EXPERT INSIGHT

python™

Mastering Python

Write powerful and efficient code using the full range of Python's capabilities

Second Edition

Rick van Hattem

Packt›

Mastering Python

Second Edition

Write powerful and efficient code using the full range of Python's capabilities

Rick van Hattem

Packt>
BIRMINGHAM—MUMBAI

Mastering Python
Second Edition

Senior Publishing Product Manager: Dr. Shailesh Jain

Contracting Acquisition Editor: Ben Renow-Clarke

Acquisition Editor – Peer Reviews: Suresh Jain

Project Editor: Janice Gonsalves

Content Development Editor: Lucy Wan, Joanne Lovell

Copy Editor: Safis Editing

Technical Editor: Aditya Sawant

Proofreader: Safis Editing

Indexer: Tejal Daruwale Soni

Presentation Designer: Ganesh Bhadwalkar
First published: April 2016
Second edition: May 2022

Production reference: 1120522

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-80020-772-1

www.packt.com

Contributors

About the author

Rick van Hattem is an entrepreneur who has founded and sold several successful start-ups. His expertise is in designing and scaling system architectures to many millions of users and/or large amounts of data that need to be accessed in real time. He's been programming for well over 25 years and has over 20 years of experience with Python. Rick has done consulting for many companies including (Y-Combinator) start-ups, banks, and airports. One of the start-ups he founded, Fashiolista.com, was one of the largest social networks for fashion in the world, featuring millions of users. He also wrote *Mastering Python, First Edition*, and he was one of the technical reviewers for *PostgreSQL Server Programming, Second Edition*.

For my wife, who is always there for me. For my sister, who always goes above and beyond to help. For my mother, who raised me to be inquisitive. And for my sweet children, who pique my curiosity and allow me to learn every day.

About the reviewer

Alexander Afanasyev is a software engineer with about 15 years of diverse experience in a variety of different domains and roles. Currently, Alexander is an independent contractor pursuing ideas in the space of computer vision, NLP, and building advanced data collections systems in the cyber and physical threat intelligence domains. Outside of daily work, he is an active contributor to Stack Overflow and GitHub. Previously, Alexander helped review the *Selenium Testing Cookbook* and *Advanced Natural Language Processing with Transformers* books by Packt Publishing.

I would like to thank the author of the book for the incredibly hard work and comprehensive content; the wonderful team of editors and coordinators with excellent communication skills; and my family, who was and always are supportive of my ideas and my work.

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:
`https://discord.gg/QMzJenHuJf`

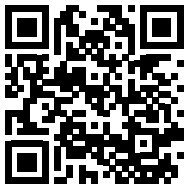


Table of Contents

Preface

Python is a language that is easy to learn and anyone can get started with a "Hello, World!" script within minutes. Mastering Python, however, is a completely different question.

Every programming problem has multiple possible solutions and choosing the Pythonic (idiomatic Python) solution is not always obvious; it can also change with time. This book will not only illustrate a range of different and new techniques but also explain where and when a method should be applied. To quote *The Zen of Python* by Tim Peters:

> *"There should be one—and preferably only one—obvious way to do it. Although that way may not be obvious at first unless you're Dutch."*

Even though it does not always help, the author of this book is actually Dutch.

This book is not a beginner's guide to Python. It is a book that can teach you about the more advanced techniques possible within Python, such as `asyncio`. It even includes Python 3.10 features, such as structural pattern matching (Python's `switch` statement), in great detail.

As a Python programmer with many years of experience, I will attempt to rationalize the choices made in this book with relevant background information. These rationalizations are in no way strict guidelines, however, as several of these cases boil down to personal style in the end. Just know that they stem from experience and are, in many cases, the solutions recommended by the Python community.

Some of the references in this book might not be obvious to you if you are not a fan of Monty Python. This book regularly uses `spam` and `eggs` instead of `foo` and `bar` in code samples because the Python programming language was named after Monty Python. To provide some background information about `spam` and `eggs`, I would recommend you watch the *Spam* sketch from Monty Python. It is positively silly.

Who this book is for

This book is meant for programmers who are already experienced in Python and want to learn more about the advanced features that Python offers. With the depth of this book, I can guarantee that almost anyone can learn something new here if they wish.

If you only know the basics of Python, however, don't worry. The book starts off relatively slow and builds to the more advanced subjects, so you should be fine.

What this book covers

Chapter 1, *Getting Started – One Environment per Project*, demonstrates several options for managing Python versions, virtual environments, and package dependencies.

Chapter 2, *Interactive Python Interpreters*, explores Python interpreter options. Python's default interpreter is perfectly functional, but better alternatives are available. With a few modifications or a replacement, you can get auto-completion, syntax highlighting, and graphical output.

Chapter 3, *Pythonic Syntax and Common Pitfalls*, discusses Pythonic coding, which is the art of writing beautiful and readable Python code. This chapter is not the holy grail, but it is filled with tips and best practices to achieve something along those lines.

Chapter 4, *Pythonic Design Patterns*, continues on the theme of *Chapter 3*. Writing Pythonic code is not just about code style, but also about using the right design patterns and data structures. This chapter tells you about the data structures available and their performance characteristics.

Chapter 5, *Functional Programming – Readability Versus Brevity*, covers functional programming. Functional programming is considered a bit of a black art by some, but when applied correctly it can be a really powerful tool that makes code reuse trivial. It is probably as close to the underlying mathematics as you can get within programming.

Chapter 6, *Decorators – Enabling Code Reuse by Decorating*, discusses decorators, an amazing tool for reusing a method. With decorators, you can wrap functions and classes with some other function to modify their parameters and return values – an extremely useful tool.

Chapter 7, *Generators and Coroutines – Infinity, One Step at a Time*, discusses generators. Lists and tuples are fantastic if you already know that you are going to use every element, but the faster alternative is to only calculate the elements you actually need. That is what a generator does for you: generate items on demand.

Chapter 8, *Metaclasses – Making Classes (not Instances) Smarter*, explores metaclasses, the classes that make other classes. It is a magic you rarely need, but it does have practical uses cases such as plugin systems.

Chapter 9, *Documentation – How to Use Sphinx and reStructuredText*, gives you some documentation-related tips. Writing documentation might not be the favorite activity for most programmers, but it is useful. This chapter shows you how to make that easier by using Sphinx and reStructuredText to generate large portions automatically.

Chapter 10, *Testing and Logging – Preparing for Bugs*, covers how to implement tests and logging to prevent and detect bugs. Bugs are inevitable and by using logging, we can trace the cause. Often, bugs can be prevented by using tests.

Chapter 11, *Debugging – Solving the Bugs*, builds on *Chapter 10*. The previous chapter helped us find the bugs; now we need to solve them. Debuggers can be a huge help when hunting down difficult bugs,

and this chapter shows you several debugging options.

Chapter 12, *Performance – Tracking and Reducing Your Memory and CPU Usage*, discusses the performance of your code. A common problem programmers have is trying to optimize code that does not need it, a fun but generally futile exercise. This chapter helps you find the code that needs to be optimized.

Chapter 13, *asyncio – Multithreading without Threads*, covers `asyncio`. Waiting for external resources such as network resources is the most common bottleneck for applications. With `asyncio`, we can stop waiting for those bottlenecks and switch to another task instead.

Chapter 14, *Multiprocessing – When a Single CPU Core Is Not Enough*, discusses performance from a different perspective. With multiprocessing, we can use multiple processors (even remotely) in parallel. When your processor is the bottleneck, this can help a lot.

Chapter 15, *Scientific Python and Plotting*, covers the most important libraries for scientific computing. Python has become the language of choice for scientific purposes.

Chapter 16, *Artificial Intelligence*, shows many AI algorithms and the libraries available for implementing them. In addition to being the language of choice for scientific purposes, most AI libraries are currently being built using Python as well.

Chapter 17, *Extensions in C/C++, System Calls, and C/C++ Libraries*, shows you how to use existing C/C++ libraries from Python, which not only allows reuse but can also speed up execution greatly. Python is a wonderful language, but it is often not the fastest solution.

Chapter 18, *Packaging – Creating Your Own Libraries or Applications*, will help you package your code into a fully functioning Python package that others can use. After building your wonderful new library, you might want to share it with the world.

To get the most out of this book

Depending on your level of experience you should start reading from the beginning, or gloss over the chapters to skip to sections that are interesting for you. This book is suitable for intermediate to expert level Python programmers, but not all sections will be equally useful for everyone.

As an example, the first two chapters are about setting up your environment and Python interpreter and seem like chapters you can skip entirely as an advanced or expert Python programmer, but I would advise against fully skipping them, as a few useful utilities and libraries are covered which you might not be familiar with.

The chapters of this book do build on each other to a certain degree, but there is no strict reading order and you can easily cherry-pick the parts you wish to read. If there is a reference to an earlier chapter, it is clearly indicated.

The most up-to-date version of the code samples can always be found at `https://github.com/mastering-python/code_2`.

The code in this repository is automatically tested and, if you have any suggestions, pull requests are always welcome.

Most chapters of this book also include exercises at the end that will allow you to test what you have learned. Since there are always multiple solutions to problems, you, and every other reader of this book, can submit and compare your solutions on GitHub: `https://github.com/mastering-python/exercises`

You are encouraged to create a pull request with your solution to the problems. And you can learn from others here as well, of course.

Download the example code files

The code bundle for the book is hosted on GitHub at `https://github.com/mastering-python/code_2` and pull requests with improvements are welcome. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://static.packt-cdn.com/downloads/9781800207721_ColorImages.pdf`.

Conventions used

There are a number of text conventions used throughout this book.

While this book largely adheres to the PEP8 styling conventions, there are a few concessions made due to the space limitations of a book format. Simply put, code samples that span multiple pages are hard to read, so some parts use less whitespace than you would usually expect. The full version of the code is available on GitHub and is automatically tested to be PEP8-compliant.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example: "The `itertools.chain()` generator is one of the simplest yet one of the most useful generators in the Python library."

A block of code is set as follows:

```
from . import base

class A(base.Plugin):
    pass
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
    :show-inheritance:
    :private-members:
    :special-members:
    :inherited-members:
```

Any command-line input or output is written as follows:

```
$ pip3 install -U mypy
```

Bold: Indicates a new term, an important word, or words that you see on the screen, for example, in menus or dialog boxes. For example: "Sometimes interactive interpreters are referred to as **REPL**. This stands for **Read-Eval-Print-Loop**."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email `feedback@packtpub.com`, and mention the book's title in the subject of your message. If you have questions about any aspect of this book, please email us at `questions@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packtpub.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `http://authors.packtpub.com`.

Share your thoughts

Once you've read *Mastering Python, Second Edition*, we'd love to hear your thoughts! Please `click here to go straight to the Amazon review page` for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

1

Getting Started — One Environment per Project

In this chapter, you'll learn about the different ways of setting up Python environments for your projects and how to use multiple Python versions on a single system outside of what your package manager offers.

After the environment is set up, we will continue with the installation of packages using both the **Python Package Index** (**PyPI**) and `conda-forge`, the package index that is coupled with Anaconda.

Lastly, we will look at several methods of keeping track of project dependencies.

To summarize, the following topics will be covered:

- Creating environments using `venv`, `pipenv`, `poetry`, `pyenv`, and `anaconda`
- Package installation through `pip`, `poetry`, `pipenv`, and `conda`
- Managing dependencies using `requirements.txt`, `poetry`, and `pipenv`

Virtual environments

The Python ecosystem offers many methods of installing and managing packages. You can simply download and extract code to your project directory, use the package manager from your operating system, or use a tool such as `pip` to install a package. To make sure your packages don't collide, it is recommended that you use a virtual environment. A virtual environment is a lightweight Python installation with its own package directories and a Python binary copied (or linked) from the binary used to create the environment.

Why virtual environments are a good idea

It might seem like a hassle to create a virtual environment for every Python project, but it offers enough advantages to do so. More importantly, there are several reasons why installing packages globally using `pip` is a really bad idea:

- Installing packages globally usually requires elevated privileges (such as `sudo`, `root`, or `administrator`), which is a huge security risk. When executing `pip install <package>`, the `setup.py` of that package is executed as the user that executed the `pip install` command. That means that if the package contains malware, it now has superuser privileges to do whatever it wants. Don't forget that anyone can upload a package to PyPI (`pypi.org`) without any vetting. As you will see later in this book, it only takes a couple of minutes for anyone to create and upload a package.
- Depending on how you installed Python, it can mess with the existing packages that are installed by your package manager. On an Ubuntu Linux system, that means you could break `pip` or even `apt` itself because a `pip install -U <package>` installs and updates both the package and all of the dependencies.
- It can break your other projects. Many projects try their best to remain backward compatible, but every `pip install` could pull in new/updated dependencies that could break compatibility with other packages and projects. The Django Web Framework, for example, changes enough between versions that many projects using Django will need several changes after an upgrade to the latest release. So, when you're upgrading Django on your system to the latest version and have a project that was written for a previous version, your project will most likely be broken.
- It pollutes your list of packages, making it hard to keep track of your project's dependencies.

In addition to alleviating the issues above, there is a major advantage as well. You can specify the Python version (assuming you have it installed) when creating the virtual environment. This allows you to test and debug your projects in multiple Python versions easily while keeping the exact same package versions beyond that.

Using venv and virtualenv

You are probably already familiar with `virtualenv`, a library used to create a virtual environment for your Python installation. What you might not know is the `venv` command, which has been included with Python since version 3.3 and can be used as a drop-in replacement for `virtualenv` in most cases. To keep things simple, I recommend creating a directory where you keep all of your environments. Some people opt for an `env`, `.venv`, or `venv` directory within the project, but I advise against that for several reasons:

- Your project files are important, so you probably want to back them up as often as possible. By keeping the bulky environment with all of the installed packages outside of your backups, your backups become faster and lighter.
- Your project directory stays portable. You can even keep it on a remote drive or flash drive without having to worry that the virtual environment will only work on a single system.
- It prevents you from accidentally adding the virtual environment files to your source control system.

If you do decide to keep your virtual environment inside your project directory, make sure that you add that directory to your `.gitignore` file (or similar) for your version control system. And if you want to keep your backups faster and lighter, exclude it from the backups. With correct dependency tracking, the virtual environment should be easy enough to rebuild.

Creating a venv

Creating a venv is a reasonably simple process, but it varies slightly according to the operating system being used.

The following examples use the `virtualenv` module directly, but for ease I recommend using `poetry` instead, which is covered later in this chapter. This module will automatically create a virtual environment for you when you first use it. Before you make the step up to `poetry`, however, it is important to understand how virtual environments work.

Since Python 3.6, the `pyvenv` command has been deprecated in favor of `python -m venv`.

In the case of Ubuntu, the `python3-venv` package has to be installed through `apt` because the Ubuntu developers have mutilated the default Python installation by not including `ensurepip`.

For Linux/Unix/OS X, using `zsh` or `bash` as a shell, it is:

```
$ python3 -m venv envs/your_env
$ source envs/your_env/bin/activate
(your_env) $
```

And for Windows `cmd.exe` (assuming `python.exe` is in your `PATH`), it is:

```
C:\Users\wolph>python.exe -m venv envs\your_env
C:\Users\wolph>envs\your_env\Scripts\activate.bat
(your_env) C:\Users\wolph>
```

PowerShell is also supported and can be used in a similar fashion:

```
PS C:\Users\wolph>python.exe -m venv envs\your_env
PS C:\Users\wolph> envs\your_env\Scripts\Activate.ps1
(your_env) PS C:\Users\wolph>
```

The first command creates the environment and the second activates the environment. After activating the environment, commands such as `python` and `pip` use the environment-specific versions, so `pip install` only installs within your virtual environment. A useful side effect of activating the environment is the prefix with the name of your environment, which is `(your_env)` in this case.

Note that we are **not** using `sudo` or other methods of elevating privileges. Elevating privileges is both unnecessary and a potential security risk, as explained in the *Why virtual environments are a good idea* section.

Using `virtualenv` instead of `venv` is as simple as replacing the following command:

```
$ python3 -m venv envs/your_env
```

with this one:

```
$ virtualenv envs/your_env
```

An additional advantage of using `virtualenv` instead of `venv`, in that case, is that you can specify the Python interpreter:

```
$ virtualenv -p python3.8 envs/your_env
```

Whereas with the `venv` command, it uses the currently running Python installation, so you need to change it through the following invocation:

```
$ python3.8 -m venv envs/your_env
```

Activating a venv/virtualenv

Every time you get back to your project after closing the shell, you need to reactivate the environment. The activation of a virtual environment consists of:

- Modifying your `PATH` environment variable to use `envs\your_env\Script` or `envs/your_env/bin` for Windows or Linux/Unix, respectively
- Modifying your prompt so that instead of `$`, you see `(your_env) $`, indicating that you are working in a virtual environment

In the case of `poetry`, you can use the `poetry shell` command to create a new shell with the activated environment.

While you can easily modify those manually, an easier method is to run the `activate` script that was generated when creating the virtual environment.

For Linux/Unix with `zsh` or `bash` as the shell, it is:

```
$ source envs/your_env/bin/activate
(your_env) $
```

For Windows using `cmd.exe`, it is:

```
C:\Users\wolph>envs\your_env\Scripts\activate.bat
(your_env) C:\Users\wolph>
```

For Windows using PowerShell, it is:

```
PS C:\Users\wolph> envs\your_env\Scripts\Activate.ps1
(your_env) PS C:\Users\wolph>
```

By default, the PowerShell permissions might be too restrictive to allow this. You can change this policy for the current PowerShell session by executing:

```
Set-ExecutionPolicy Unrestricted -Scope Process
```

If you wish to permanently change it for every PowerShell session for the current user, execute:

```
Set-ExecutionPolicy Unrestricted -Scope CurrentUser
```

Different shells, such as `fish` and `csh`, are also supported by using the `activate.fish` and `activate.csh` scripts, respectively.

When not using an interactive shell (with a cron job, for example), you can still use the environment by using the Python interpreter in the `bin` or `scripts` directory for Linux/Unix or Windows, respectively. Instead of running `python script.py` or `/usr/bin/python script.py`, you can use:

```
/home/wolph/envs/your_env/bin/python script.py
```

Note that commands installed through `pip` (and `pip` itself) can be run in a similar fashion:

```
/home/wolph/envs/your_env/bin/pip
```

Installing packages

Installing packages within your virtual environment can be done using `pip` as normal:

```
$ pip3 install <package>
```

The great advantage comes when looking at the list of installed packages:

```
$ pip3 freeze
```

Because our environment is isolated from the system, we only see the packages and dependencies that we have explicitly installed.

Fully isolating the virtual environment from the system Python packages can be a downside in some cases. It takes up more disk space and the package might not be in sync with the C/C++ libraries on the system. The PostgreSQL database server, for example, is often used together with the `psycopg2` package. While binaries are available for most platforms and building the package from the source is fairly easy, it can sometimes be more convenient to use the package that is bundled with your system. That way, you are certain that the package is compatible with both the installed Python and PostgreSQL versions.

To mix your virtual environment with system packages, you can use the `--system-site-packages` flag when creating the environment:

```
$ python3 -m venv --system-site-packages envs/your_env
```

When enabling this flag, the environment will have the system Python environment `sys.path` appended to your virtual environment's `sys.path`, effectively providing the system packages as a fallback when an `import` from the virtual environment fails.

Explicitly installing or updating a package within your virtual environment will effectively hide the system package from within your virtual environment. Uninstalling the package from your virtual environment will make it reappear.

As you might suspect, this also affects the results of `pip freeze`. Luckily, `pip freeze` can be told to only list the packages local to the virtual environment, which excludes the system packages:

```
$ pip3 freeze --local
```

Later in this chapter, we will discuss `pipenv`, which transparently handles the creation of the virtual environment for you.

Using pyenv

The `pyenv` library makes it really easy to quickly install and switch between multiple Python versions. A common issue with many Linux and Unix systems is that the package managers opt for stability over recency. In most cases, this is definitely an advantage, but if you are running a project that requires the latest and greatest Python version, or a really old version, it requires you to compile and install it manually. The `pyenv` package makes this process really easy for you but does still require the compiler to be installed.

A nice addition to `pyenv` for testing purposes is the `tox` library. This library allows you to run your tests on a whole list of Python versions simultaneously. The usage of `tox` is covered in *Chapter 10, Testing and Logging – Preparing for Bugs*.

To install `pyenv`, I recommend visiting the `pyenv` project page, since it depends highly on your operating system and operating system version. For Linux/Unix, you can use the regular `pyenv` installation manual or the `pyenv-installer` (`https://github.com/pyenv/pyenv-installer`) one-liner, if you deem it safe enough:

```
$ curl https://pyenv.run | bash
```

Make sure that you follow the instructions given by the installer. To ensure `pyenv` works properly, you will need to modify your `.zshrc` or `.bashrc`.

Windows does not support `pyenv` natively (outside of Windows Subsystem for Linux) but has a `pyenv` fork available: `https://github.com/pyenv-win/pyenv-win#installation`

After installing `pyenv`, you can view the list of supported Python versions using:

```
$ pyenv install --list
```

The list is rather long, but can be shortened with `grep` on Linux/Unix:

```
$ pyenv install --list | grep 3.10
  3.10.0
  3.10-dev
...
```

Once you've found the version you like, you can install it through the `install` command:

```
$ pyenv install 3.10-dev
Cloning https://github.com/python/cpython...
Installing Python-3.10-dev...
Installed Python-3.10-dev to /home/wolph/.pyenv/versions/3.10-dev
```

The `pyenv install` command takes an optional `--debug` parameter, which builds a debug version of Python that makes debugging C/C++ extensions possible using a debugger such as `gdb`.

Once the Python version has been built, you can activate it globally, but you can also use the `pyenv-virtualenv` plugin (`https://github.com/pyenv/pyenv-virtualenv`) to create a `virtualenv` for your newly created Python environment:

```
$ pyenv virtualenv 3.10-dev your_pyenv
```

you can see in the preceding example, as opposed to the `venv` and `virtualenv` commands, `pyenv virtualenv` automatically creates the environment in the `~/.pyenv/versions/<version>/envs/` directory so you're not allowed to fully specify your own path. You can change the base path (`~/.pyenv/`) through the `PYENV_ROOT` environment variable, however. Activating the environment using the `activate` script in the environment directory is still possible, but more complicated than it needs to be since there's an easy shortcut:

```
$ pyenv activate your_pyenv
```

Now that the environment is activated, you can run environment-specific commands, such as `pip`, and they will only modify your environment.

Using Anaconda

Anaconda is a distribution that supports both the Python and R programming languages. It is much more than simply a virtual environment manager, though; it's a whole different Python distribution with its own virtual environment system and even a completely different package system. In addition to supporting PyPI, it also supports `conda-forge`, which features a very impressive number of packages focused on scientific computing.

For the end user, the most important difference is that packages are installed through the `conda` command instead of `pip`. This brings a much more advanced dependency check when installing packages. Whereas `pip` will simply install a package and all of its dependencies without regard for other installed packages, `conda` will look at all of the installed packages and make sure it won't install a version that is not supported by the installed packages.

The `conda` package manager is not alone in smart dependency checking. The `pipenv` package manager (discussed later in this chapter) does something similar.

Getting started with Anaconda Navigator

Installing Anaconda is quite easy on all common platforms. For Windows, OS X, and Linux, you can go to the Anaconda site and download the (graphical) installer: `https://www.anaconda.com/products/distribution#Downloads`

Once it's installed, the easiest way to continue is by launching Anaconda Navigator, which should look something like this:

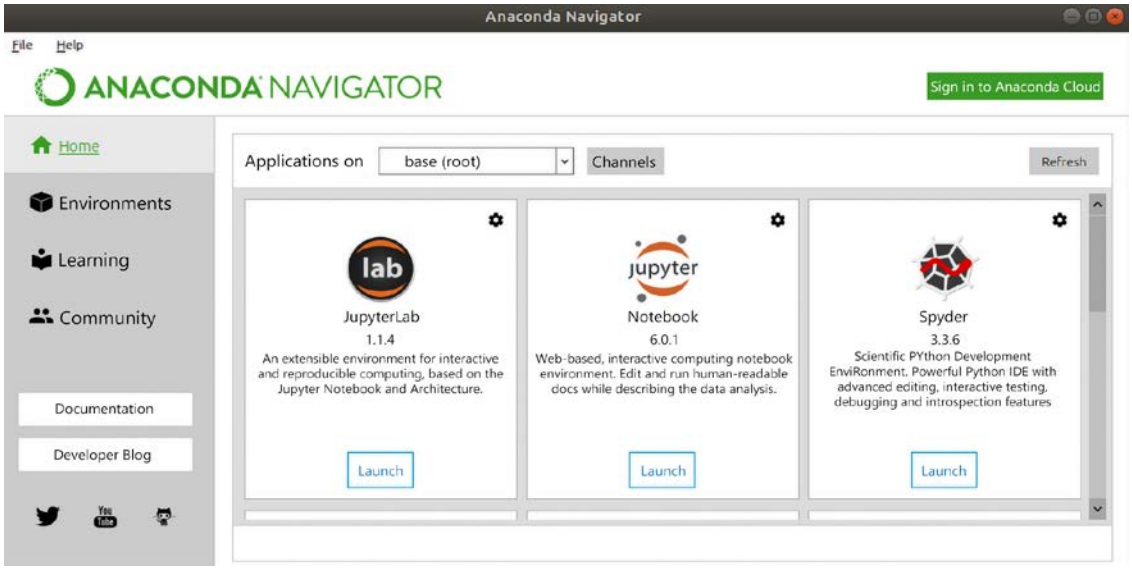


Figure 1.1: Anaconda Navigator – Home

Creating an environment and installing packages is pretty straightforward as well:

1. Click on the **Environments** button on the left.
2. Click on the **Create** button below.
3. Enter your name and Python version.
4. Click on **Create** to create your environment and wait a bit until Anaconda is done:

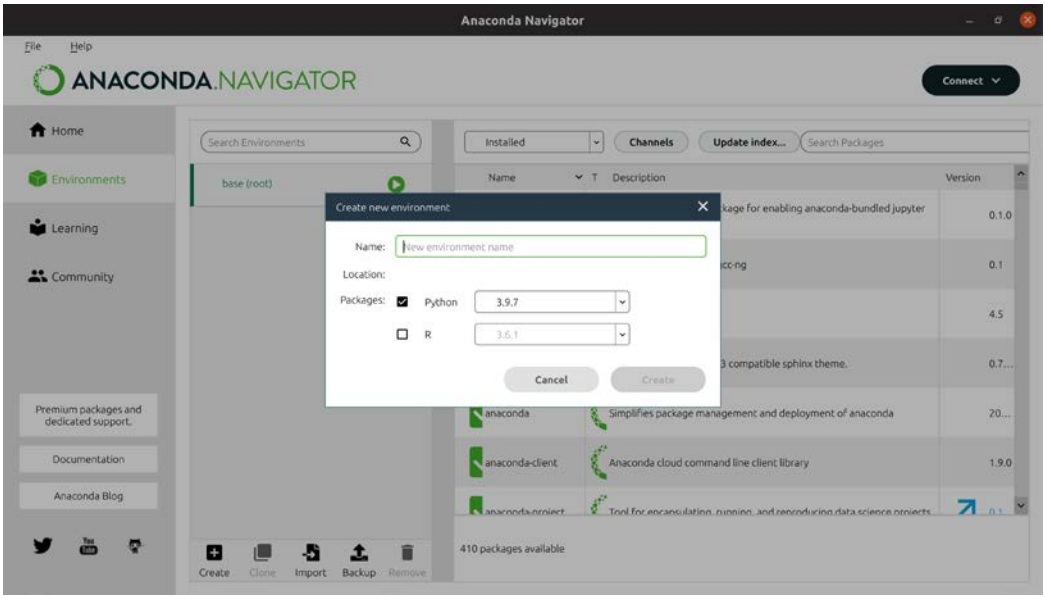


Figure 1.2: Anaconda Navigator – Creating an environment

Once Anaconda has finished creating your environment, you should see a list of installed packages. Installing packages can be done by changing the filter of the package list from **Installed** to **All**, marking the checkbox near the packages you want to install, and applying the changes.

While creating an environment, Anaconda Navigator shows you where the environment will be created.

Getting started with conda

While Anaconda Navigator is a really nice tool to use to get an overview, being able to run your code from the command line can be convenient too. With the `conda` command, that is luckily very easy.

First, you need to open the `conda` shell. You can do this from Anaconda Navigator if you wish, but you can also run it straightaway. On Windows, you can open Anaconda Prompt or Anaconda PowerShell Prompt from the start menu. On Linux and OS X, the most convenient method is to initialize the shell integration. For zsh, you can use:

```
$ conda init zsh
```

For other shells, the process is similar. Note that this process modifies your shell configuration to automatically activate the `base` environment every time you open a shell. This can be disabled with a simple configuration option:

```
$ conda config --set auto_activate_base false
```

If automatic activation is not enabled, you will need to run the `activate` command to get back into the `conda base` environment:

```
$ conda activate
(base) $
```

If, instead of the `conda base` environment, you wish to activate the environment you created earlier, you need to specify the name:

```
$ conda activate conda_env
(conda_env) $
```

If you have not created the environment yet, you can do so using the command line as well:

```
$ conda create --name conda_env
Collecting package metadata (current_repodata.json): done
Solving environment: done
...
Proceed ([y]/n)? y

Preparing transaction: done
Verifying transaction: done
Executing transaction: done
...
```

To list the available environments, you can use the `conda info` command:

```
$ conda info --envs
# conda environments
#
base                  *  /usr/local/anaconda3
conda_env                /usr/local/anaconda3/envs/conda_env
```

Installing conda packages

Now it's time to install a package. For `conda` packages, you can simply use the `conda install` command. For example, to install the `progressbar2` package that I maintain, use:

```
(conda_env) $ conda install progressbar2
Collecting package metadata (current_repodata.json): done
Solving environment: done
```

```
## Package Plan ##
  environment location: /usr/local/anaconda3/envs/conda_env

  added / updated specs:
    - progressbar2
The following packages will be downloaded:
...
The following NEW packages will be INSTALLED:
...
Proceed ([y]/n)? y

Downloading and Extracting Packages
...
```

Now you can run Python and see that the package has been installed and is working properly:

```
(conda_env) $ python
Python 3.8.0 (default, Nov  6 2019, 15:49:01)
[Clang 4.0.1 (tags/RELEASE_401/final)] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import progressbar

>>> for _ in progressbar.progressbar(range(5)): pass
...
100% (5 of 5) |############################| Elapsed Time: 0:00:00 Time:
0:00:00
```

Another way to verify whether the package has been installed is by running the `conda list` command, which lists the installed packages similarly to `pip list`:

```
(conda_env) $ conda list
# packages in environment at /usr/local/anaconda3/envs/conda_env:
#
# Name                    Version                   Build  Channel
...
```

Installing PyPI packages

With PyPI packages, we have two options within the Anaconda distribution. The most obvious is using `pip`, but this has the downside of partially circumventing the `conda` dependency checker. While `conda install` will take the packages installed through PyPI into consideration, the `pip` command might upgrade packages undesirably. This behavior can be improved by enabling the `conda`/`pip` interoperability setting, but this seriously impacts the performance of `conda` commands:

```
$ conda config --set pip_interop_enabled True
```

Depending on how important fixed versions or `conda` performance is for you, you can also opt for converting the package to a `conda` package:

```
(conda_env) $ conda skeleton pypi progressbar2
Warning, the following versions were found for progressbar2
...
Use --version to specify a different version.
...
## Package Plan ##
...
The following NEW packages will be INSTALLED:
...
INFO:conda_build.config:--dirty flag and --keep-old-work not specified.
Removing build/test folder after successful build/test.
```

Now that we have a package, we can modify the files if needed, but using the automatically generated files works most of the time. All that is left now is to build and install the package:

```
(conda_env) $ conda build progressbar2
...
(conda_env) $ conda install --use-local progressbar2
Collecting package metadata (current_repodata.json): done
Solving environment: done
...
```

And now we are done! The package has been installed through `conda` instead of `pip`.

Sharing your environment

When collaborating with others, it is essential to have environments that are as similar as possible to avoid debugging local issues. With `pip`, we can simply create a requirements file by using `pip freeze`, but that will not include the `conda` packages. With `conda`, there's actually an even better solution, which stores not only the dependencies and versions but also the installation channels, environment name, and environment location:

```
(conda_env) $ conda env export -file environment.yml
(conda_env) $ cat environment.yml
name: conda_env
channels:
  - defaults
dependencies:
...
prefix: /usr/local/anaconda3/envs/conda_env
```

Installing the packages from that environment file can be done while creating the environment:

```
$ conda env create --name conda_env –file environment.yml
```

Or they can be added to an existing environment:

```
(conda_env) $ conda env update --file environment.yml
Collecting package metadata (repodata.json): done
...
```

Managing dependencies

The simplest way of managing dependencies is storing them in a `requirements.txt` file. In its simplest form, this is a list of package names and nothing else. This file can be extended with version requirements and can even support environment-specific installations.

A fancier method of installing and managing your dependencies is by using a tool such as `poetry` or `pipenv`. Internally, these use the regular `pip` installation method, but they build a full dependency graph of all the packages. This makes sure that all package versions are compatible with each other and allows the parallel installation of non-dependent packages.

Using pip and a requirements.txt file

The `requirements.txt` format allows you to list all of the dependencies of your project as broadly or as specifically as you feel is necessary. You can easily create this file yourself, but you can also tell `pip` to generate it for you, or even to generate a new file based on a previous `requirements.txt` file so you can view the changes. I recommend using `pip freeze` to generate an initial file and cherry-picking the dependencies (versions) you want.

For example, assuming that we run `pip freeze` in our virtual environment from before:

```
(your_env) $ pip3 freeze
pkg-resources==0.0.0
```

If we store that file in a `requirements.txt` file, install a package, and look at the difference, we get this result:

```
(your_env) $ pip3 freeze > requirements.txt
(your_env) $ pip3 install progressbar2
Collecting progressbar2
...
Installing collected packages: six, python-utils, progressbar2
Successfully installed progressbar2-3.47.0 python-utils-2.3.0 six-1.13.0
(your_env) $ pip3 freeze -r requirements.txt
pkg-resources==0.0.0
## The following requirements were added by pip freeze:
```

```
progressbar2==3.47.0
python-utils==2.3.0
six==1.13.0
```

As you can see, the `pip freeze` command automatically detected the addition of the `six`, `progressbar2`, and `python-utils` packages, and it immediately pinned those versions to the currently installed ones.

The lines in the `requirements.txt` file are understood by `pip` on the command line as well, so to install a specific version, you can run:

```
$ pip3 install 'progressbar2==3.47.0'
```

Version specifiers

Often, pinning a version as strictly as that is not desirable, however, so let's change the requirements file to only contain what we actually care about:

```
# We want a progressbar that is at least version 3.47.0 since we've tested
that.
# But newer versions are ok as well.
progressbar2>=3.47.0
```

If someone else wants to install all of the requirements in this file, they can simply tell `pip` to include that requirement:

```
(your_env) $ pip3 install -r requirements.txt
Requirement already satisfied: progressbar2>=3.47.0 in your_env/lib/python3.9/
site-packages (from -r requirements.txt (line 1))
Requirement already satisfied: python-utils>=2.3.0 in your_env/lib/python3.9/
site-packages (from progressbar2>=3.47.0->-r requirements.txt (line 1))
Requirement already satisfied: six in your_env/lib/python3.9/site-packages
(from progressbar2>=3.47.0->-r requirements.txt (line 1))
```

In this case, `pip` checks to see whether all packages are installed and will install or update them if needed.

`-r requirements.txt` works recursively, allowing you to include multiple requirements files.

Now let's assume we've encountered a bug in the latest version and we wish to skip it. We can assume that only this specific version is affected, so we will only blacklist that version:

```
# Progressbar 2 version 3.47.0 has a silly bug but anything beyond 3.46.0 still
works with our code
progressbar2>=3.46,!=3.47.0
```

Lastly, we should talk about wildcards. One of the most common scenarios is needing a specific major version number but still wanting the latest security update and bug fixes. There are a few ways to specify these:

```
# Basic wildcard:
progressbar2 ==3.47.*
# Compatible release:
progressbar2 ~=3.47.1
# Compatible release above is identical to:
progressbar2 >=3.47.1, ==3.47.*
```

With the compatible release pattern (~=), you can select the newest version that is within the same major release but is at least the specified version.

The version identification and dependency specification standard is described thoroughly in PEP 440:

`https://peps.python.org/pep-0440/`

Installing through source control repositories

Now let's say that we're really unlucky and there is no working release of the package yet, but it has been fixed in the `develop` branch of the Git repository. We can install that either through `pip` or through a `requirements.txt` file, like this:

```
(your_env) $ pip3 install --editable 'git+https://github.com/wolph/python-
progressbar@develop#egg=progressbar2'
Obtaining progressbar2 from git+https://github.com/wolph/python-progressbar@
develop#egg=progressbar2
  Updating your_env/src/progressbar2 clone (to develop)
Requirement already satisfied: python-utils>=2.3.0 in your_env/lib/python3.9/
site-packages (from progressbar2)
Requirement already satisfied: six in your_env/lib/python3.9/site-packages
(from progressbar2)
Installing collected packages: progressbar2
  Found existing installation: progressbar2 3.47.0
    Uninstalling progressbar2-3.47.0:
      Successfully uninstalled progressbar2-3.47.0
  Running setup.py develop for progressbar2
Successfully installed progressbar2
```

You may notice that `pip` not only installed the package but actually did a `git clone` to `your_env/src/progressbar2`. This is an optional step caused by the `--editable` (short option: `-e`) flag, which has the additional advantage that every time you re-run the command, the `git` clone will be updated. It also makes it rather easy to go to that directory, modify the code, and create a pull request with a fix.

In addition to Git, other source control systems such as Bazaar, Mercurial, and Subversion are also supported.

Additional dependencies using extras

Many packages offer optional dependencies for specific use cases. In the case of the `progressbar2` library, I have added `tests` and `docs` extras to install the test or documentation building dependencies needed to run the tests for the package. Extras can be specified using square brackets separated by commas:

```
# Install the documentation and test extras in addition to the progressbar
progressbar2[docs,tests]

# A popular example is the installation of encryption libraries when using the
requests library:
requests[security]
```

Conditional dependencies using environment markers

If your project needs to run on multiple systems, you will most likely encounter dependencies that are not required on all systems. One example of this is libraries that are required on some operating systems but not on others. An example of this is the `portalocker` package I maintain; on Linux/Unix systems, the locking mechanisms needed are supported out of the box. On Windows, however, they require the `pywin32` package to work. The `install_requires` part of the package (which uses the same syntax as `requirements.txt`) contains this line:

```
pywin32!=226; platform_system == "Windows"
```

This specifies that on Windows, the `pywin32` package is required, and version `226` was blacklisted due to a bug.

In addition to `platform_system`, there are several more markers, such as `python_version` and `platform_machine` (contains architecture `x86_64`, for example).

The full list of markers can be found in PEP 496: `https://peps.python.org/pep-0496/`.

One other useful example of this is the `dataclasses` library. This library has been included with Python since version 3.7, so we only need to install the backport for older Python versions:

```
dataclasses; python_version < '3.7'
```

Automatic project management using poetry

The `poetry` tool provides a really easy-to-use solution for creating, updating, and sharing your Python projects. It's also very fast, which makes it a fantastic starting point for a project.

Creating a new poetry project

Starting a new project is very easy. It will automatically handle virtual environments, dependencies, and other project-related tasks for you. To start, we will use the `poetry init` wizard:

```
$ poetry init
This command will guide you through creating your pyproject.toml config.

Package name [t_00_poetry]:
Version [0.1.0]:
Description []:
Author [Rick van Hattem <Wolph@wol.ph>, n to skip]:
License []:
Compatible Python versions [^3.10]:

Would you like to define your main dependencies interactively? (yes/no) [yes]
no
Would you like to define your development dependencies interact...? (yes/no)
[yes] no
...
Do you confirm generation? (yes/no) [yes]
```

Following these few questions, it automatically creates a `pyproject.toml` file for us that contains all the data we entered and some automatically generated data. As you may have noticed, it automatically prefilled several values for us:

- The project name. This is based on the current directory name.
- The version. This is fixed to `0.1.0`.
- The author field. This looks at your `git` user information. This can be set using:

  ```
  $ git config --global user.name "Rick van Hattem"
  $ git config --global user.email "Wolph@wol.ph"
  ```

- The Python version. This is based on the Python version you are running `poetry` with, but it can be customized using `poetry init --python=...`

Looking at the generated `pyproject.toml`, we can see the following:

```
[tool.poetry]
name = "t_00_poetry"
version = "0.1.0"
```

```
description = ""
authors = ["Rick van Hattem <Wolph@wol.ph>"]

[tool.poetry.dependencies]
python = "^3.10"

[tool.poetry.dev-dependencies]

[build-system]
requires = ["poetry-core>=1.0.0"]
build-backend = "poetry.core.masonry.api"
```

Adding dependencies

Once we have the project up and running, we can now add dependencies:

```
$ poetry add progressbar2
Using version ^3.55.0 for progressbar2
...
Writing lock file
...
  • Installing progressbar2 (3.55.0)
```

This automatically installs the package, adds it to the `pyproject.toml` file, and adds the specific version to the `poetry.lock` file. After this command, the `pyproject.toml` file has a new line added to the `tool.poetry.dependencies` section:

```
[tool.poetry.dependencies]
python = "^3.10"
progressbar2 = "^3.55.0"
```

The `poetry.lock` file is a bit more specific. Whereas the `progressbar2` dependency could have a wildcard version, the `poetry.lock` file stores the exact version, the file hashes, and all the dependencies that were installed:

```
[[package]]
name = "progressbar2"
version = "3.55.0"
...
[package.dependencies]
python-utils = ">=2.3.0"
...
[package.extras]
docs = ["sphinx (>=1.7.4)"]
...
```

```
[metadata]
lock-version = "1.1"
python-versions = "^3.10"
content-hash =
"c4235fba0428ce7877f5a94075e19731e5d45caa73ff2e0345e5dd269332bff0"

[metadata.files]
progressbar2 = [
    {file = "progressbar2-3.55.0-py2.py3-none-any.whl", hash = "sha256:..."},
    {file = "progressbar2-3.55.0.tar.gz", hash = "sha256:..."},
]
...
```

By having all this data, we can build or rebuild a virtual environment for a `poetry`-based project on another system exactly as it was created on the original system. To install, upgrade, and/or downgrade the packages exactly as specified in the `poetry.lock` file, we need a single command:

```
$ poetry install
Installing dependencies from lock file
...
```

This is very similar to how the `npm` and `yarn` commands work if you are familiar with those.

Upgrading dependencies

In the previous examples, we simply added a dependency without specifying an explicit version. Often this is a safe approach, as the default version requirement will allow for any version within that major version.

If the project uses normal Python versioning or semantic versioning (more about that in *Chapter 18, Packaging - Creating Your Own Libraries or Applications*), that should be perfect. At the very least, all of my projects (such as progressbar2) are generally both backward and largely forward compatible, so simply fixing the major version is enough. In this case, `poetry` defaulted to version `^3.55.0`, which means that any version newer than or equal to 3.55.0, up to (but not including) 4.0.0, is valid.

Due to the `poetry.lock` file, a `poetry install` will result in those exact versions being installed instead of the new versions, however. So how can we upgrade the dependencies? For this purpose, we will start by installing an older version of the `progressbar2` library:

```
$ poetry add 'progressbar2=3.1.0'
```

Now we will relax the version in the `pyproject.toml` file to `^3.1.0`:

```
[tool.poetry.dependencies]
progressbar2 = "^3.1.0"
```

Once we have done this, a `poetry install` will still keep the `3.1.0` version, but we can make `poetry` update the dependencies for us:

```
$ poetry update
...
  • Updating progressbar2 (3.1.0 -> 3.55.0)
```

Now, poetry has nicely updated the dependencies in our project while still adhering to the requirements we set in the `pyproject.toml file`. If you set the version requirements of all packages to `*`, it will always update everything to the latest available versions that are compatible with each other.

Running commands

To run a single command using the `poetry` environment, you can use `poetry run`:

```
$ poetry run pip
```

For an entire development session, however, I would suggest using the `shell` command:

```
$ poetry shell
```

After this, you can run all Python commands as normal, but these will now be running from the activated virtual environment.

For cron jobs this is similar, but you will need to make sure that you change directories first:

```
0 3 * * *       cd /home/wolph/workspace/poetry_project/ && poetry run python
script.py
```

This command runs every day at 03:00 (24-hour clock, so A.M.).

Note that cron might not be able to find the `poetry` command due to having a different environment. In that case, I would recommend using the absolute path to the `poetry` command, which can be found using `which`:

```
$ which poetry
/usr/local/bin/poetry
```

Automatic dependency tracking using pipenv

For large projects, your dependencies can change often, which makes the manual manipulation of the `requirements.txt` file rather tedious. Additionally, having to create a virtual environment before you can install your packages is also a pretty repetitive task if you work on many projects. The `pipenv` tool aims to transparently solve these issues for you, while also making sure that all of your dependencies are compatible and updated. And as a final bonus, it combines the strict and loose dependency versions so you can make sure your production environment uses the exact same versions you tested.

Initial usage is simple; go to your project directory and install a package. Let's give it a try:

```
$ pipenv install progressbar2
Creating a virtualenv for this project...
...
Using /usr/local/bin/python3 (3.10.4) to create virtualenv...
...
✔ Successfully created virtual environment!
...
Creating a Pipfile for this project...
Installing progressbar2...
Adding progressbar2 to Pipfile's [packages]...
✔ Installation Succeeded
Pipfile.lock not found, creating...
...
✔ Success!
Updated Pipfile.lock (996b11)!
Installing dependencies from Pipfile.lock (996b11)...
  🐍  ████████████████████████████████ 0/0 — 00:00:0
```

That's quite a bit of output even when abbreviated. But let's look at what happened:

- A virtual environment was created.
- A `Pipfile` was created, which contains the dependency as you specified it. If you specify a specific version, that will be added to the `Pipfile`; otherwise, it will be a wildcard requirement, meaning that any version will be accepted as long as there are no conflicts with other packages.
- A `Pipfile.lock` was created containing the exact list of packages and versions as installed. This allows an identical install on a different machine with the exact same versions.

The generated `Pipfile` contains the following:

```
[[source]]
name = "pypi"
url = "https://pypi.org/simple"
verify_ssl = true

[dev-packages]

[packages]
progressbar2 = "*"

[requires]
python_version = "3.10"
```

And the `Pipfile.lock` is a bit larger, but immediately shows another advantage of this method:

```
{
    ...
    "default": {
        "progressbar2": {
            "hashes": [
                "sha256:14d3165a1781d053...",
                "sha256:2562ba3e554433f0..."
            ],
            "index": "pypi",
            "version": "==4.0.0"
        },
        "python-utils": {
            "hashes": [
                "sha256:4dace6420c5f50d6...",
                "sha256:93d9cdc8b8580669..."
            ],
            "markers": "python_version >= '3.7'",
            "version": "==3.1.0"
        },
        ...
    },
    "develop": {}
}
```

As you can see, in addition to the exact package versions, the `Pipfile.lock` contains the hashes of the packages as well. In this case, the package provides both a `.tar.gz` (source) and a `.whl` (wheel) file, which is why there are two hashes. Additionally, the `Pipfile.lock` contains all packages installed by `pipenv`, including all dependencies.

Using these hashes, you can be certain that during a deployment, you will receive the exact same file and not some corrupt or even malicious file.

Because the versions are completely fixed, you can also be certain that anyone deploying your project using the `Pipfile.lock` will get the exact same package versions. This is very useful when working together with other developers.

To install all the necessary packages as specified in the `Pipfile` (even for the initial install), you can simply run:

```
$ pipenv install
Installing dependencies from Pipfile.lock (5c99e1)…
  🐍 ████████████████████████████████ 3/3 — 00:00:00
```

```
To activate this project's virtualenv, run pipenv shell.
Alternatively, run a command inside the virtualenv with pipenv run.
```

Any time you run `pipenv install package`, the `Pipfile` will be automatically modified with your changes and checked for incompatible packages. The big downside is that `pipenv` can become terribly slow for large projects. I have encountered multiple projects where a no-op `pip install` would take several minutes due to the fetching and checking of the entire dependency graph. In most cases, it's still worth it, however; the added functionality can save you a lot of headaches.

Don't forget to run your regular Python commands with the `pipenv run` prefix or from `pipenv shell`.

Updating your packages

Because of the dependency graph, you can easily update your packages without having to worry about dependency conflicts. With one command, you're done:

```
$ pipenv update
```

Should you still encounter issues with the versions because some packages haven't been checked against each other, you can fix that by specifying the versions of the package you do or do not want:

```
$ pipenv install 'progressbar2!=3.47.0'
Installing progressbar2!=3.47.0…
Adding progressbar2 to Pipfile's [packages]…
✔ Installation Succeeded
Pipfile.lock (c9327e) out of date, updating to (5c99e1)…
✔ Success!
Updated Pipfile.lock (c9327e)!
Installing dependencies from Pipfile.lock (c9327e)…
  🐍 ████████████████████████████████ 3/3 — 00:00:00
```

By running that command, the `packages` section of the `Pipfile` changes to:

```
[packages]
progressbar2 = "!=3.47.0"
```

Deploying to production

Getting the exact same versions on all of your production servers is absolutely essential to prevent hard-to-trace bugs. For this very purpose, you can tell `pipenv` to install everything as specified in the `Pipenv.lock` file while still checking to see whether `Pipfile.lock` is out of date. With one command, you have a fully functioning production virtual environment with all packages installed.

Let's create a new directory and see if it all works out:

```
$ mkdir ../pipenv_production
$ cp Pipfile Pipfile.lock ../pipenv_production/
$ cd ../pipenv_production/
$ pipenv install --deploy
Creating a virtualenv for this project...
Pipfile: /home/wolph/workspace/pipenv_production/Pipfile
Using /usr/bin/python3 (3.10.4) to create virtualenv...
...
✔ Successfully created virtual environment!
...
Installing dependencies from Pipfile.lock (996b11)...
  🐍 ▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉ 2/2 — 00:00:01
$ pipenv shell
Launching subshell in virtual environment...
(pipenv_production) $ pip3 freeze
progressbar2==4.0.0
python-utils==3.1.0
```

All of the versions are exactly as expected and ready for use.

Running cron commands

To run your Python commands outside of the `pipenv shell`, you can use the `pipenv run` prefix. Instead of `python`, you would run `pipenv run python`. In normal usage, this is a lot less practical than activating the `pipenv shell`, but for non-interactive sessions, such as cron jobs, this is an essential feature. For example, a cron job that runs at 03:00 (24-hour clock, so A.M.) every day would look something like this:

```
0 3 * * *       cd /home/wolph/workspace/pipenv_project/ && pipenv run python
script.py
```

Exercises

Many of the topics discussed in this chapter already gave full examples, leaving little room for exercises. There are additional resources to discover, however.

Reading the Python Enhancement Proposals (PEPs)

A good way to learn more about the topics discussed in this chapter (and all the following chapters) is to read the PEP pages. These proposals were written before the changes were accepted into the Python core. Note that not all of the PEPs on the Python site have been accepted, but they will remain on the Python site:

- PEP 440 – Version Identification and Dependency Specification: `https://peps.python.org/pep-0440/`
- PEP 496 – Environment Markers: `https://peps.python.org/pep-0496/`

Combining pyenv and poetry or pipenv

Even though the chapter did not cover it, there is nothing stopping you from telling `poetry` or `pipenv` to use a `pyenv`-based Python interpreter. Give it a try!

Converting an existing project to a poetry project

Part of this exercise should be to either create a brand new `pyproject.toml` or to convert an existing `requirements.txt` file to a `pyproject.toml`.

Summary

In this chapter, you learned why virtual environments are useful and you discovered several implementations of them and their advantages. We explored how to create virtual environments and how to install multiple different Python versions. Finally, we covered how to manage the dependencies for your Python projects.

Since Python is an interpreted language, it is easily possible to run code from the interpreter directly instead of through a Python file.

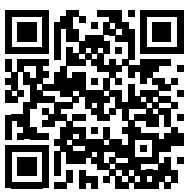
The default Python interpreter already features command history and depending on your install, basic autocompletion.

But with alternative interpreters we can have many more features in our interpreter such as syntax highlighting, smart autocompletion which includes documentation, and more.

The next chapter will show us several alternative interpreters and their advantages.

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:
`https://discord.gg/QMzJenHuJf`



2

Interactive Python Interpreters

Now that we have a working Python installation, we need to run some code. The most obvious way is to create a Python file and execute it. Often, it can be faster to interactively develop code from an interactive Python interpreter, however. While the standard Python interpreter is already quite powerful, many enhancements and alternatives are available.

The alternative interpreters/shells offer features such as:

- Smart autocompletion
- Syntax highlighting
- Saving and loading sessions
- Automatic indenting
- Graphing/charting output

In this chapter, you will learn about:

- Alternative interpreters:
    - `bpython`
    - `ptpython`
    - `ipython`
    - `jupyter`
- How to enhance interpreters

The Python interpreter

The standard Python interpreter is already fairly powerful, but more options are available through customization. First, let's start with a `'Hello world!'`. Because the interpreter uses REPL, all output will be automatically printed and we can simply create a string.

Sometimes interactive interpreters are referred to as **REPL**. This stands for **Read-Eval-Print-Loop**. This effectively means that all of your statements will be executed and printed to your screen immediately.

First, we need to start the interpreter; after that, we can type our commands:

```
$ python3
Python 3.9.0
[GCC 7.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 'Hello world!'
'Hello world!'
```

That was easy enough. And note that we didn't have to use `print('Hello world!')` to show the output.

Many interpreters have only limited support for Windows. While they all work to some degree, your experience will be better with Linux or OS X systems. I recommend trying them from a (virtual) Linux/Unix machine at least once to experience the full range of features.

Modifying the interpreter

As our first enhancement, we will add a few convenient shortcuts to the scope of the interpreter. Instead of having to type `import pprint; pprint.pprint(...)` to pretty-print our output, it would be useful to use `pp(...)` instead without having to run an `import` statement every time we start our interpreter. To do this, we will create a Python file that will be executed every time we run Python. On Linux and OS X systems, I would recommend `~/.config/python/init.py`; on Windows, something like `C:\Users\rick\AppData\Local\Python\init.py` might be more suitable. Within this file, we can add regular Python code that will be executed.

Python won't find the file automatically; you need to tell Python where to look for the file by using the `PYTHONSTARTUP` environment variable. On Linux and OS X, you can change the `~/.zshrc`, `~/.bashrc` file, or whatever your shell has, and add:

```
$ export PYTHONSTARTUP=~/.config/python/init.py
```

This file is automatically executed every time you open a new shell session. So, once you open a new shell session, you are done.

If you want to activate this for your current shell, you can also run the export line above in your current shell.

On Windows, you need to find the Advanced System Settings and change the environment variables on that screen.

Now we can add these lines to the file to make pretty print (`pprint`/`pp`) and pretty format (`pformat`/`pf`) available by default:

```
from pprint import pprint as pp
from pprint import pformat as pf
```

When we run the Python interpreter, now we will have `pp` and `pf` available in our scope:

```
>>> pp(dict(spam=0xA, eggs=0xB))
{'eggs': 11, 'spam': 10}
>>> pf(dict(spam=0xA, eggs=0xB))
"{'eggs': 11, 'spam': 10}"
```

With a few of these minor changes, you can make your life a lot easier. You could modify your `sys.path` to include a directory with custom libraries, for example. And you can also change your prompt using the `sys.ps1` and `sys.ps2` variables. To illustrate, we'll look at the interpreter before our changes:

```
# Modifying prompt
>>> if True:
...     print('Hello!')
Hello!
```

And now we will modify `sys.ps1` and `sys.ps2` and run the exact same code again:

```
>>> import sys

>>> sys.ps1 = '> '
>>> sys.ps2 = '. '

# With modified prompt
> if True:
.     print('Hello!')
Hello!
```

The configuration above shows that you can easily change the interpreter to a slightly different output if you wish. For consistency purposes, however, it might be better to keep it the same.

Enabling and enhancing autocompletion

One of the most useful additions to the interpreter is the `rlcompleter` module. This module enables tab-activated autocompletion in your interpreter and is automatically activated if the `readline` module is available.

The `rlcompleter` module depends on the availability of the `readline` module, which is not bundled with Python on Windows systems. Luckily, an alternative can be installed easily:

```
$ pip3 install pyreadline
```

It would be very useful to add some extra options to the autocompletion. First, look at the default output:

```
>>> sandwich = dict(spam=2, eggs=1, sausage=1)
>>> sandwich.<TAB>
sandwich.clear(       sandwich.fromkeys(    sandwich.items(       sandwich.pop(
sandwich.setdefault(  sandwich.values(      sandwich.copy(        sandwich.get(
sandwich.keys(        sandwich.popitem(     sandwich.update(
>>> sandwich[<TAB>
```

As you can see, the tab completion for "." works perfectly, but the tab completion for "[" does nothing. It would be useful to know the available items, so now we will work on adding that feature. It should be noted that this example uses a few techniques that are explained in later chapters, but that shouldn't matter for now:

```
import __main__
import re
import atexit
import readline
import rlcompleter

class Completer(rlcompleter.Completer):
    ITEM_RE = re.compile(r'(?P<expression>.+?)\[(?P<key>[^\[]*)')

    def complete(self, text, state):
        # Init namespace. From 'rlcompleter.Completer.complete'
        if self.use_main_ns:
            self.namespace = __main__.__dict__

        # If we find a [, try and return the keys
        if '[' in text:
            # At state 0 we need to prefetch the matches, after
            # that we use the cached results
            if state == 0:
                self.matches = list(self.item_matches(text))

            # Try and return the match if it exists
            try:
                return self.matches[state]
            except IndexError:
                pass
        else:
```

```
            # Fallback to the normal completion
            return super().complete(text, state)

    def item_matches(self, text):
        # Look for the pattern expression[key
        match = self.ITEM_RE.match(text)
        if match:
            search_key = match.group('key').lstrip()
            expression = match.group('expression')

            # Strip quotes from the key
            if search_key and search_key[0] in {"'", '"'}:
                search_key = search_key.strip(search_key[0])

            # Fetch the object from the namespace
            object_ = eval(expression, self.namespace)

            # Duck typing, check if we have a 'keys()' attribute
            if hasattr(object_, 'keys'):
                # Fetch the keys by executing the 'keys()' method
                # Can you guess where the bug is?
                keys = object_.keys()
                for i, key in enumerate(keys):
                    # Limit to 25 items for safety, could be infinite
                    if i >= 25:
                        break

                    # Only return matching results
                    if key.startswith(search_key):
                        yield f'{expression}[{key!r}]'

# By default readline doesn't call the autocompleter for [ because
# it's considered a delimiter. With a little bit of work we can
# fix this however :)
delims = readline.get_completer_delims()
# Remove [, ' and " from the delimiters
delims = delims.replace('[', '').replace('"', '').replace("'", '')
# Set the delimiters
readline.set_completer_delims(delims)
```

```
# Create and set the completer
completer = Completer()
readline.set_completer(completer.complete)
# Add a cleanup call on Python exit
atexit.register(lambda: readline.set_completer(None))
print('Done initializing the tab completer')
```

That was quite a bit of code, and if you look carefully, you'll notice multiple potential bugs in this limited example. I'm just trying to show a working example here without introducing too much complexity, so several edge cases are not considered. To make the script work, we need to store it in the `PYTHONSTARTUP` file as we discussed earlier. You should see the result from `print()` after opening the interpreter so you can verify whether the script was loaded. With this addition, we can now complete dictionary keys as well:

```
Done initializing the tab completer
>>> sandwich = dict(spam=2, eggs=1, sausage=1)
>>> sandwich['<TAB>
sandwich['eggs']     sandwich['sausage']  sandwich['spam']
```

Naturally, you could expand this to include colors, other completions, and many more useful features.

Since this completion calls `object.keys()`, there is a potential risk here. This code could be dangerous if, for some reason, the `object.keys()` method code is not safe to execute. Perhaps you are running on an external library, or your code has overridden the `keys()` method to execute a heavy database function. And if `object.keys()` is a generator that is exhausted after executing once, you won't have any results when running your actual code afterward.

Additionally, the `eval()` function can be dangerous to execute on unknown code. In this case, `eval()` is only executing the line we typed ourselves, so that is less of an issue here.

Alternative interpreters

Now that you have seen some of the features of the regular Python interpreter, let's look at some enhanced alternatives. There are many options available, but we will limit ourselves to the most popular ones here:

- `bpython`
- `ptpython`
- `ipython`
- `jupyter` (web-based `ipython`)

Let's get started.

bpython

The `bpython` interpreter is a `curses` interface for the Python interpreter that offers many useful features, while still being very similar to the regular Python interpreter.

The `curses` library allows you to create a fully functioning **text-based user interface** (**TUI**). A TUI gives you full control over where you want to write to the screen. The regular Python interpreter is a **command-line interface** (**CLI**), which normally only allows you to append to the screen. With a TUI, you can write to any position on the screen, making its features somewhat comparable to a **graphical user interface** (**GUI**).

Some key features of `bpython`:

- As-you-type autocompletion (as opposed to tab completion with `rlcompleter`)
- In-line syntax highlighting while typing
- Automatic function parameter documentation
- A undo/rewind feature that removes the last line
- Easy reloading of imported modules, so your external code changes can be tested without restarting the interpreter
- Quick changing of code in an external editor (convenient for multiline functions/code blocks)
- The ability to save the session to file/pastebin

Most of these features work fully automatically and transparently for you. Before we can start with `bpython`, we need to install it. A simple `pip install` should suffice:

```
$ pip3 install bpython
```

To illustrate the automatically enabled features, here is the output of the code we used for the regular Python interpreter completion:

```
$ bpython
bpython version 0.21 on top of Python 3.9.6
>>> sandwich = dict(spam=2, eggs=1, sausage=1)
┌──────────────────────────────────────────────────────────────┐
│ dict: (self, *args, **kwargs)                                │
│ Initialize self.  See help(type(self)) for accurate signature. │
└──────────────────────────────────────────────────────────────┘
>>> sandwich.
┌──────────────────────────────────────────────────────────────┐
│ clear                copy                 fromkeys           │
│ get                  items                keys               │
│ pop                  popitem              setdefault         │
│ update               values                                  │
└──────────────────────────────────────────────────────────────┘
```

```
>>> sandwich[
┌──────────────────────────────────────────────────────────────────────┐
│ 'eggs'     'sausage'  'spam'                                         │
└──────────────────────────────────────────────────────────────────────┘
```

If you ran this code on your own system, you would see highlighting as well as the intermediate states of autocompletion. I encourage you to give it a try; the preceding excerpt does not show enough.

Rewinding your session

As for the more advanced features, let's give those a try as well. First, let's start with the rewind feature. While it appears to simply remove the last line, in the background it actually replays your entire history, except for the last line. This means that if your code is not safe to be run more than once, it can cause errors. The following code illustrates the usage and limitations of the rewind feature:

```
>>> with open('bpython.txt', 'a') as fh:
...     fh.write('x')
...
1

>>> with open('bpython.txt') as fh:
...     print(fh.read())
...
x

>>> sandwich = dict(spam=2, eggs=1, sausage=1)
```

Now if we press *Ctrl* + *R* to "rewind" the last line, we get the following output:

```
>>> with open('bpython.txt', 'a') as fh:
...     fh.write('x')
...
1

>>> with open('bpython.txt') as fh:
...     print(fh.read())
...
xx

>>>
```

As you can see, the last line is gone now, but that's not all; the output of the `fh.read()` line is now `xx` instead of `x`, which means that the line that writes `x` was executed twice. Additionally, the partial line will be executed as well, so when rewinding an indented block of code, you will see an error until you've executed valid code again.

Reloading modules

Often, when developing, I will write code in my regular editor and test the execution in the Python shell.

When developing like this, a very useful feature of Python is the ability to reload imported modules using `importlib.reload()`. When you have multiple (nested) modules, this can get tedious fast, however. This is where the reload shortcut in `bpython` can help a lot. By using the *F6* button on your keyboard, `bpython` will not only run `importlib.reload()` on all modules in `sys.modules`, but it will also rerun the code in your session in a similar way to the rewind feature you saw earlier.

To demonstrate this, we will start by creating a file named `bpython_reload.py` with the following code:

```
with open('reload.txt', 'a+') as fh:
    fh.write('x')
    fh.seek(0)
    print(fh.read())
```

This opens the `reload.txt` file for reading and writing in append mode. This means that `fh.write('x')` will append to the end of the file. The `fh.seek(0)` will jump to the beginning of the file (position 0) so that `print(fh.read())` can print the entire file content to the screen.

Now we open the `bpython` shell and import the module:

```
>>> import bpython_reload
x
```

If we press the *F6* button within that same shell, we will see that an extra character has been written and the code has been re-executed:

```
>>> import bpython_reload
xx
Reloaded at ... by user.
```

This is an extremely useful feature with the same caveat as the rewind feature that not all code is safe to re-execute without side effects.

ptpython

The `ptpython` interpreter is younger (available since 2014) than `bpython` (available since 2009), so it might be slightly less mature and feature rich. It is, however, very actively developing and certainly worth mentioning. While there is (currently) no code reload feature similar to the one in `bpython`, there are several other useful features that `bpython` currently lacks:

- Multiline code editing
- Mouse support
- Both Vi and Emacs key bindings
- Syntax checking while typing
- A history browser
- Output highlighting

These features are all ones you need to experience yourself, though; a book is not the right medium for a demonstration in this case. In any case, this interpreter is certainly worth looking at.

Installation can be done with a simple `pip install`:

```
$ pip3 install ptpython
```

After installing, you can run it using the `ptpython` command:

```
$ ptpython
>>>
```

Once the interpreter is running, you can configure `ptpython` using the built-in menu (press *F2*). In that menu, you can configure and enable/disable features such as completion for dictionaries, completion while typing, input validation, color depth, and highlighting colors.

IPython and Jupyter

The IPython interpreter is a completely different beast from the previously mentioned interpreters. In addition to being the interpreter with the most features, it is part of a whole ecosystem of packages that includes parallel computing, integrations with visual toolkits, interactive widgets, and a web-based interpreter (Jupyter).

Some key features of the IPython interpreter:

- Easy object introspection
- Output formatting (instead of `repr()`, IPython calls `pprint.pformat()`)
- Command history can be accessed through variables and magic methods from both new and old sessions
- Saving and loading sessions
- A whole range of magic commands and shortcuts
- Access to regular shell commands such as `cd` and `ls`
- Extensible tab completion, supporting not just Python methods and functions but filenames as well

Several of the other features of the IPython project are covered in the chapters about debugging, multiprocessing, scientific programming, and machine learning.

The basic installation of IPython can be done using a `pip install`:

```
$ pip3 install ipython
```

Installing through Anaconda is also a good option, though, especially if you are planning to use a lot of data science packages, which are often far easier to install and manage through `conda`:

```
$ conda install ipython
```

Basic interpreter usage

The IPython interpreter can be used in a similar way to the other interpreters, but has somewhat different output from the other interpreters. Here's an example covering some of the key features:

```
$ ipython
Python 3.9.6 (default, Jun 29 2021, 05:25:02)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.25.0 -- An enhanced Interactive Python. Type '?' for help.
In [1]: sandwich = dict(spam=2, eggs=1, sausage=1)

In [2]: sandwich
Out[2]: {'spam': 2, 'eggs': 1, 'sausage': 1}

In [3]: sandwich = dict(spam=2, eggs=1, sausage=1, bacon=1, chees
   ...: e=2, lettuce=1, tomatoes=3, pickles=1)

In [4]: sandwich
Out[4]:
{'spam': 2,
 'eggs': 1,
 'sausage': 1,
 'bacon': 1,
 'cheese': 2,
 'lettuce': 1,
 'tomatoes': 3,
 'pickles': 1}

In [5]: _i1
Out[5]: 'sandwich = dict(spam=2, eggs=1, sausage=1)'

In [6]: !echo "$_i2"
sandwich
```

The first line is a simple variable declaration; nothing special there. The second line shows the print output for the variable declared in the first line.

Now we declare a similar dictionary with more items in it. You can see that the output is now automatically formatted and split over multiple lines for readability if the line is too long for the screen. This effectively comes down to `print()` versus `pprint.pprint()`.

At `In [5]: _i1`, we see one of the useful internal variables, the input line. The `_i<N>` and `_ih[<N>]` variables give you the lines you wrote. Similarly, the last three entered lines are available through `_i`, `_ii`, and `_iii`, respectively.

If the command generated output, it will be available through `_<N>`. And the last three output results are available through `_`, `__`, and `___`.

Finally, we call the external shell function `echo` by prefixing the line with `!` while passing along the Python variable `_i2`. When executing external shell functions, we can pass along Python variables by prefixing them with a `$`.

Saving and loading sessions

The ability to save and load a session so you can always come back to it is an incredibly useful feature. As is usually the case with IPython, there are several ways of achieving this goal. First of all, every session is already automatically saved for you, requiring no effort whatsoever. To load the previous session, you can run:

```
In [1]: %load ~1/

In [2]: # %load ~1/
   ...: sandwich = dict(spam=2, eggs=1, sausage=1)

In [3]: sandwich
Out[3]: {'spam': 2, 'eggs': 1, 'sausage': 1}
```

This command uses the same syntax as the `%history` command. Here is a quick overview of how the `%history` syntax works:

- `5`: Line 5
- `-t 5`: Line 5 as pure Python (without IPython magic)
- `10-20`: Lines 10 to 20
- `10/20`: Session 10, line 20
- `~0/`: Current session
- `~1/10-20`: Previous session lines 10 to 20
- `~5/-~2`: Everything from 5 sessions ago to 2 sessions ago

If you know that a session will be an important one and you want to make sure it gets saved, use `%logstart`:

```
In [1]: %logstart
Activating auto-logging. Current session state plus future input saved.
Filename       : ipython_log.py
Mode           : rotate
Output logging : False
Raw input log  : False
Timestamping   : False
State          : active
```

As can be seen in the output, this feature is configurable. By default, it will write to (and rotate, if it exists) `ipython_log.py`. As soon as you run this command again, the previous logfile will be renamed to `ipython_log.001~` and so on for the older files.

Loading is done using the `%load` command and will immediately reactivate auto-logging since it's replaying that line as well:

```
In [1]: %load ipython_log.py

In [2]: # %load ipython_log.py
   ...: # IPython log file
   ...:
   ...: get_ipython().run_line_magic('logstart', '')
   ...:
Activating auto-logging. Current session state plus future input saved.
Filename       : ipython_log.py
Mode           : rotate
Output logging : False
Raw input log  : False
Timestamping   : False
State          : active
```

Naturally, manually saving is also an option using `%save`. I would recommend adding the `-r` parameter so the session is saved as raw instead of a regular Python file. Let's illustrate the difference:

```
In [1]: %save session_filename ~0/
The following commands were written to file 'session_filename.py':
get_ipython().run_line_magic('save', 'session_filename ~0/')

In [2]: %save -r raw_session ~0/
The following commands were written to file 'raw_session.ipy':
%save session_filename ~0/
%save -r raw_session ~0/
```

If you don't need to run the session from a regular Python interpreter, using the raw files is somewhat more legible.

Regular Python prompt/doctest mode

The default `ipython` prompt is very useful but it can feel a little verbose at times and you can't easily copy the results to a file for doctests (we will cover more about doctests in *Chapter 10*, *Testing and Logging – Preparing for Bugs*). Because of that, it can be convenient to activate the `%doctest_mode` magic function so your prompt looks like the familiar Python interpreter:

```
In [1]: sandwich = dict(spam=2, eggs=1, sausage=1, bacon=1, chees
   ...: e=2, lettuce=1, tomatoes=3, pickles=1)
```

```
In [2]: sandwich
Out[2]:
{'spam': 2,
 'eggs': 1,
 'sausage': 1,
 'bacon': 1,
 'cheese': 2,
 'lettuce': 1,
 'tomatoes': 3,
 'pickles': 1}

In [3]: %doctest_mode
Exception reporting mode: Plain
Doctest mode is: ON
>>> sandwich
{'spam': 2, 'eggs': 1, 'sausage': 1, 'bacon': 1, 'cheese': 2, 'lettuce': 1,
'tomatoes': 3, 'pickles': 1}
```

As you can see, this also influences how the output is formatted, so it's really similar to the regular Python shell. While magic functions can still be used, the output is nearly identical to the regular Python shell.

Introspection and help

One of the most useful shortcuts of IPython is `?`. That is the shortcut for accessing the IPython help, object help, and object introspection. If you're looking for an up-to-date overview of the IPython interpreter features, start by typing `?` and start reading. If you're planning to use IPython, I definitely recommend doing so.

The `?` and `??` can be used both as a suffix and as a prefix. So, both `?history` and `history?` will return in the documentation for the `%history` command.

Because the `?` shortcut shows the documentation, it is useful for both regular Python objects and the magic functions in IPython. The magic functions are really not that magic; besides having a name that's prefixed with a `%`, they are just regular Python functions. In addition to `?`, there is also `??`, which attempts to show the source of the object:

```
In [1]: import pathlib

In [2]: pathlib.Path.name?
Type:        property
String form: <property object at 0x10c540ef0>
```

```
Docstring:   The final path component, if any.

In [3]: pathlib.Path.name??
Type:        property
String form: <property object at 0x10c540ef0>
Source:
# pathlib.Path.name.fget
@property
def name(self):
    """The final path component, if any."""
    parts = self._parts
    if len(parts) == (1 if (self._drv or self._root) else 0):
        return ''
    return parts[-1]
```

Autocompletion

Autocompletion is where `ipython` really gets interesting. In addition to the regular code completion, `ipython` will complete filenames and LaTeX/Unicode for special characters as well.

The really useful part starts when creating your own objects, though. While regular automatic autocompletion will work without a hitch, you can customize the autocompletion to only return specific items, or do dynamic lookups from a database if needed. Usage is certainly easy enough:

```
In [1]: class CompletionExample:
   ...:     def __dir__(self):
   ...:         return ['attribute', 'autocompletion']
   ...:
   ...:     def _ipython_key_completions_(self):
   ...:         return ['key', 'autocompletion']
   ...:

In [2]: completion = CompletionExample()

In [3]: completion.a<TAB>
                    attribute
                    autocompletion

In [4]: completion['aut<TAB>
                       %autoawait     %autoindent
                       %autocall      %automagic
                       autocompletion
```

Now for the LaTeX/Unicode character completion. While this might not be something you need to use that often, I find it really useful in the cases that you do need it:

```
In [1]: '\pi<TAB>'

In [1]: 'π
```

Jupyter

The Jupyter project offers an amazing web-based interpreter (Jupyter Notebook) that makes Python much more accessible for people who need to write some scripts but aren't programmers by trade. It allows a seamless mix of Python code, LaTeX, and other markup.

The web-based interpreter isn't the only or even most important feature of the Jupyter project, though. The biggest advantage of the Jupyter project is that it allows you to connect to remote systems (called "kernels") from your local machine.

Originally, the project was part of the IPython project when `ipython` was still a large monolithic application that contained all components internally. Since then, the IPython project has been split into multiple IPython projects and several projects under the Jupyter name. Internally, they are still using much of the same code base and Jupyter heavily depends on IPython.

Before we continue, we should look at the current structure of the Jupyter and IPython projects and describe the most important projects:

- `jupyter`: The metapackage that contains all the Jupyter projects.
- `notebook`: The web-based interpreter, which is part of the Jupyter project.
- `lab`: The next-generation web-based interpreter offering multiple notebooks side by side and even supporting code embedded in other languages such as Markdown, R, and LaTeX.
- `ipython`: The Python terminal interface with the magic functions.
- `jupyter_console`: The Jupyter version of `ipython`.
- `ipywidgets`: Interactive widgets that can be used as user input in `notebook`.
- `ipyparallel`: The library for easy parallel execution of Python code across multiple servers. There will be more about this in *Chapter 14*, *Multiprocessing - When a Single CPU Core Is Not Enough*.
- `traitlets`: The config system used by IPython and Jupyter, which allows you to create configurable objects with validation. There will be more about this in *Chapter 8*, *Metaclasses - Making Classes (Not Instances) Smarter*.

Figure 2.1 shows the complexity and the size of the Jupyter and IPython projects and how they work together:

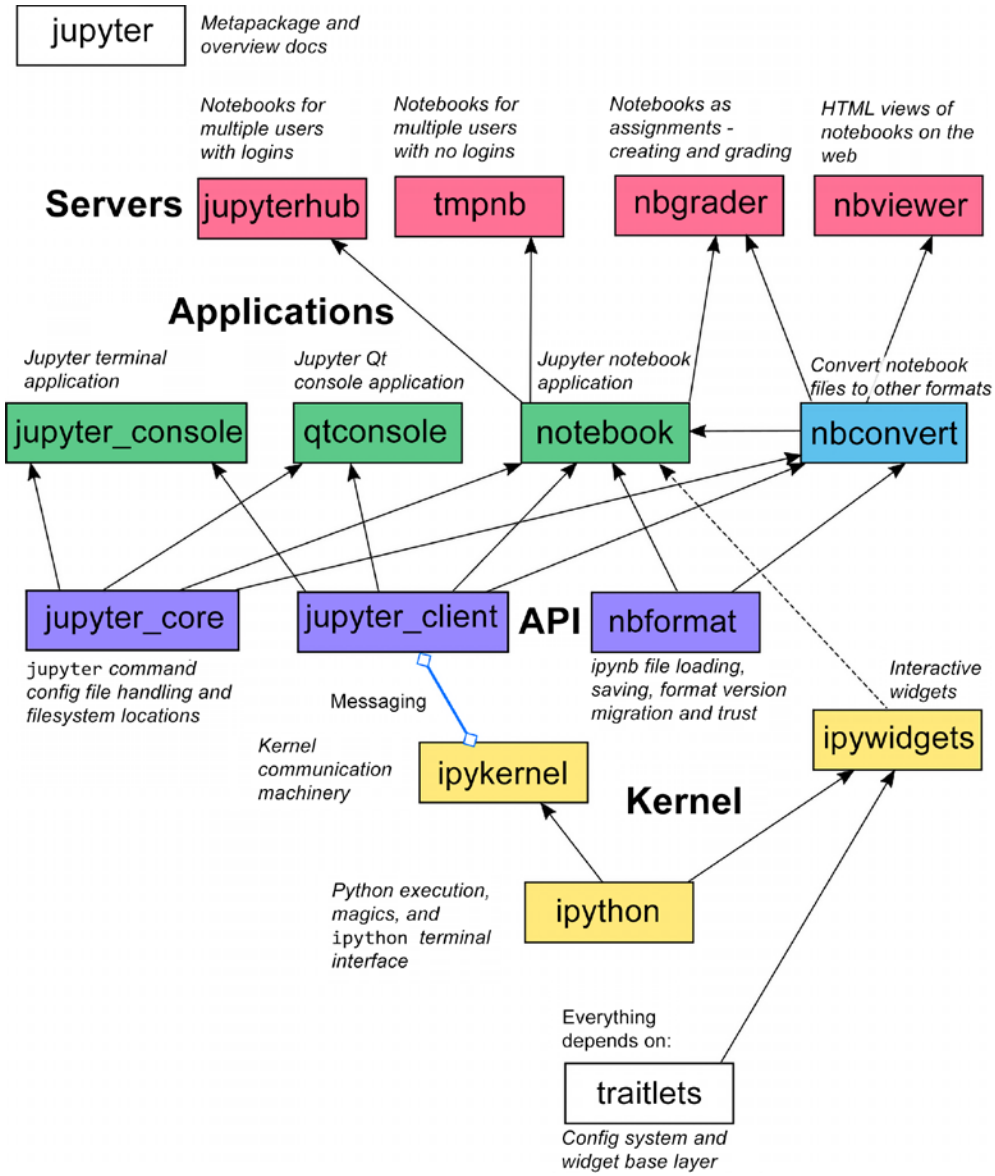


Figure 2.1: Jupyter and IPython project structure

From this overview, you might wonder why both `ipython` and `jupyter console` exist. The difference is that `ipython` runs completely locally in a single process, and `jupyter console` runs everything on a remote kernel. When running locally, this means that Jupyter will automatically start a background processing kernel that any Jupyter application can connect to.

The Jupyter project could easily fill several books by itself so we will cover only the most common features in this chapter. Additionally, *Chapter 14* covers the multiprocessing aspect in more detail. And *Chapter 15*, *Scientific Python and Plotting*, depends on Jupyter Notebook as well.

Installing Jupyter

First, let's start with the installation. The installation is easy enough with a simple `pip install` or `conda install`:

```
$ pip3 install --upgrade jupyterlab
```

Now, all that's left is to start it. Once you run the following command, your web browser should automatically open:

```
$ jupyter lab
```

Docker images are available as well if, for some reason, the installation gives you trouble or if you want an easy installation for a lot of dependency-heavy packages. For the data science chapter later in the book, the `jupyter/tensorflow-notebook` Docker image is used:

```
$ docker run -p 8888:8888 jupyter/tensorflow-notebook
```

This will run the Docker image and forward port `8888` to the running `jupyter lab` so you can access it. Note that because of the default security, you will need to open `jupyter lab` through the links provided in the console, which contains the randomly generated security token. It should look something like this:

```
http://127.0.0.1:8888/?token=..........
```

Once you have it up and running, you should see something like this in your browser:

Figure 2.2: Jupyter dashboard

Now you can create a new notebook:

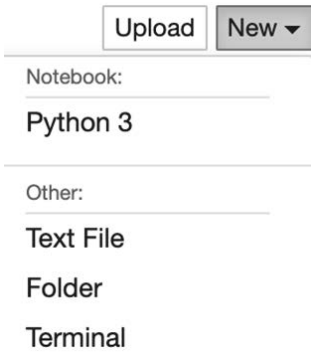


Figure 2.3: A new file in Jupyter

And start typing with tab completion and all the features that are similar to `ipython`:

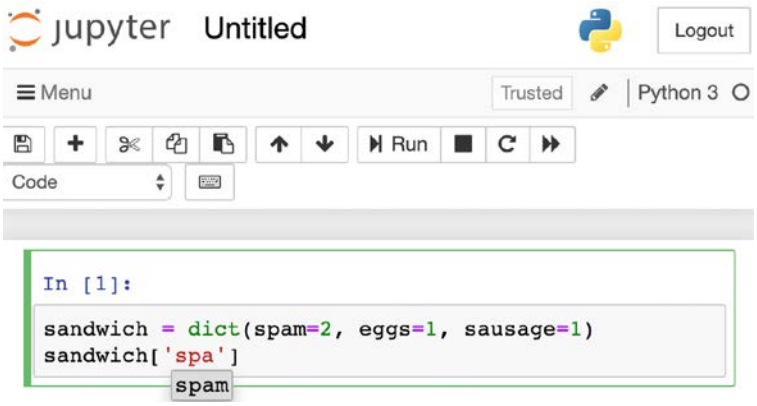


Figure 2.4: Jupyter tab completion

Within a notebook, you can have multiple cells. Each cell can have multiple lines of code and behave similarly to the IPython interpreter with one key difference: *only the last line* decides what is returned as the output, instead of each line being printed separately. But that doesn't prevent you from using `print()` functions.

```
In [3]: sandwich = dict(spam=2, eggs=1, sausage=1)
        sandwich
        sandwich
Out[3]: {'spam': 2, 'eggs': 1, 'sausage': 1}
```

Figure 2.5: Jupyter output

Each of these cells can be (re-)executed separately if needed, or all at once, to make sure the notebook still functions properly. In addition to code cells, Jupyter also supports several types of markup languages, such as Markdown, to add nicely formatted documentation.

And because it's a web-based format, you can attach all sorts of objects, such as videos, audio files, PDF files, images, and renders. LaTeX formulas, for example, are mostly impossible to render in a normal interpreter, but with Jupyter, rendering a LaTeX formula is easily possible:

```
In [6]: from IPython import display
        display.Math(r'''
        F(k) = \int_{-\infty}^{\infty}
               f(x) e^{2\pi i k} dx')''')
```

Out[6]: $$F(k) = \int_{-\infty}^{\infty} f(x)e^{2\pi ik}\,dx')$$

Figure 2.6: A LaTeX formula in Jupyter

Lastly, we have interactive widgets, which are one of the best features of using notebooks over a regular shell session:

```
In [6]: import ipywidgets

        def square(n):
            return n ** 2

        ipywidgets.interact(square, n=10)
```

n 6

36

```
Out[6]: <function __main__.square(n)>
```

Figure 2.7: Jupyter widgets

By moving the slider, the function will be called again and the result will be immediately updated. This is extremely useful when debugging functions. In the chapter about user interfaces, you will learn how to create our own.

IPython summary

The entire list of features in the IPython and Jupyter projects could easily fill several books by itself, so we have only glossed over a very small portion of what the interpreter supports.

Later chapters will cover some other parts of the project, but the IPython documentation is your friend. The documentation is really detailed and largely up to date.

An overview of some of the shortcuts/magic functions that you'll want to look at follows:

- `%quickref`: A quick reference for most of the interpreter features and a list of the magic functions.
- `%cd`: Change the current working directory for your `ipython` session.
- `%paste`: Paste a pre-formatted code block from the clipboard so your indentation is pasted correctly and not mutilated/clobbered due to auto-indentation.
- `%edit`: Open an external editor for easy editing of code blocks. This is very useful when quickly testing multiline code blocks. The `%edit -p` command, for example, will re-edit the previous (`-p`) code block.
- `%timeit`: A shortcut to quickly benchmark a line of Python code using the `timeit` module.
- `?`: Look at the documentation for any object.
- `??`: Look at the source for any Python object. Native methods such as `sum()` are compiled C code, so the source can't be fetched easily.

Exercises

1. The `rlcompleter` enhancement we created currently only handles dictionaries. Try and extend the code so it supports lists, strings, and tuples as well.
2. Add colors to the completer (hint: use `colorama` for the coloring).
3. Instead of manually completing using our own object introspection, try and use the `jedi` library for autocompletion, which does static code analysis.

Static code analysis inspects code without executing it. This means it's entirely safe to run, even on foreign code, as opposed to the autocompletion we wrote earlier, which runs the code in `object.keys()`.

4. Try to create a `Hello <ipywidget>` so the name of the person can be edited through a notebook without code changes.
5. Try and create a script that will look for a given pattern through all of your previous `ipython` sessions.

Example answers for these exercises can be found on GitHub: `https://github.com/mastering-python/exercises`. You are encouraged to submit your own solutions and learn about alternative solutions from others.

Summary

This chapter has shown you several of the available Python interpreters and some of the pros and cons. Additionally, you have had a small glimpse of what IPython and Jupyter can offer us. *Chapter 15*, *Scientific Python and Plotting*, almost exclusively uses Jupyter Notebooks and demonstrates a few more powerful features, such as plotting integration.

For most generic Python programmers, I would suggest using either `bpython` or `ptpython`, since they are really fast and lightweight interpreters to (re-)start that still offer a lot of useful features.

If your focus is more on scientific programming and/or handling large datasets in your shell, then IPython or JupyterLab are probably more useful. These are far more powerful tools, but they come at the cost of having slightly higher start up times and system requirements. I personally use both depending on the use case. When testing a few simple lines of Python and/or verifying the behavior of a small code block, I mostly use `bpython`/`ptpython`. When working with larger blocks of code and/or data, I tend to use IPython (or `ptipython`) or even JupyterLab.

The next chapter covers the Python style guide, which rules are important, and why they matter. Readability is one of the most important aspects of the Python philosophy, and you will learn methods and styles for writing cleaner and more readable Python code. In short, you will learn what Pythonic code is and how to write it.

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:
`https://discord.gg/QMzJenHuJf`

3

Pythonic Syntax and Common Pitfalls

In this chapter, you will learn how to write Pythonic code, along with finding out about some of the common pitfalls of Python and how to work around them. The pitfalls range from passing a list or dictionary (which are mutable) as an argument to more advanced pitfalls, such as late-binding in closures. You will also see how to fix or work around circular imports in a clean way. Some of the techniques used in the examples in this chapter might seem a bit too advanced for such an early chapter. Do not worry, though, as the inner workings will be covered later on.

We will explore the following topics in this chapter:

- Code style (PEP 8, `pyflakes`, `flake8`, and more)
- Common pitfalls (lists as function arguments, pass by value versus pass by reference, and inheritance behavior)

The definition of Pythonic code used in this chapter is based on commonly accepted coding guidelines and my subjective opinions. When working on a project, it is most important to stay consistent with the coding styles of that project.

A brief history of Python

The Python project started in December 1989 as a hobby project for Guido van Rossum during his week off around Christmas. His goal was to write an easy-to-use successor for the ABC programming language and to fix the issues that limited the applicability of the it. One of the main design goals of Python is, and has always been, readability. That is what the first part of the chapter is about: readability.

To facilitate new features and to maintain that readability, the **Python Enhancement Proposal** (**PEP**) process was developed. This process allows **anyone** to submit a PEP for a new feature, library, or other addition. After a discussion on the Python mailing lists and some improvements, a decision is made to either accept or reject the proposal.

The Python style guide (PEP 8: `https://peps.python.org/pep-0008/`) was once submitted as one of those PEPs, was accepted, and has been improved regularly ever since. It has a lot of great and widely accepted conventions, as well as a few disputed ones. In particular, the maximum line length of 79 characters is a topic of much discussion. Limiting a line to 79 characters does have some merits, however. Originally, this choice was made because terminals were 80 characters wide, but these days, larger monitors allow you to place multiple files next to each other. For docstrings and comments, a 72-character limit is recommended to increase readability. Additionally, it's the common convention for Linux/Unix man (manual) pages.

While just the style guide itself does not make code Pythonic, as *The Zen of Python* (PEP 20: `https://peps.python.org/pep-0020/`) elegantly puts it: "Beautiful is better than ugly." PEP 8 defines how code should be formatted in an exact way, while PEP 20 is more of a philosophy and mindset than anything else.

For almost 30 years, all major decisions for the Python project were made by Guido van Rossum, lovingly called the **BDFL** (**Benevolent Dictator For Life**). Unfortunately, the "For Life" part of BDFL was not to be after a heated debate over PEP 572. PEP 572 (covered later in this chapter) was a proposal about assignment operators, the ability to set a variable inside an `if` statement, a common practice in languages such as C, C++, C# and others. Guido van Rossum was not a fan of the syntax and opposed the PEP. This triggered a huge debate and he was met with such resistance that it moved him to step down as BDFL. It saddened many people that Guido van Rossum, universally loved by the community, felt he had to do this. I, for one, will certainly miss his insights as the decision-maker. I hope we will still see his "Time Machine" in action a few times. Guido van Rossum is thought to have a time machine, as he has repeatedly answered feature requests with "I just implemented that last night."

Without the BDFL to make the final decisions, the Python community had to come up with a new way of decision-making, and a whole list of proposals have been written to solve this issue:

- PEP 8010: Continue status quo (ish): `https://peps.python.org/pep-8010/`
- PEP 8011: Like status quo but with three co-leaders: `https://peps.python.org/pep-8011/`
- PEP 8012: No central authority: `https://peps.python.org/pep-8012/`
- PEP 8013: Non-core oversight: `https://peps.python.org/pep-8013/`
- PEP 8014: Core oversight: `https://peps.python.org/pep-8014/`
- PEP 8015: Organization of the Python community: `https://peps.python.org/pep-8015/`
- PEP 8016: The Steering Council Model: `https://peps.python.org/pep-8016/`

After a small debate, PEP 8016 - the steering council model - was accepted as the solution. PEP 81XX has been reserved for future elections of the steering council, with PEP 8100 for the 2019 election, PEP 8101 for the 2020 election, and so on.

Code style – What is Pythonic code?

When you first hear of Pythonic code, you might think it is a programming paradigm, similar to object-oriented or functional programming. It is actually more of a design philosophy. Python leaves you free to choose to program in an object-oriented, procedural, functional, aspect-oriented, or even logic-oriented way. These freedoms make Python a great language to write in, but they have the drawback of requiring more discipline to keep code clean and readable. PEP 8 tells us how to format code and PEP 20 is about style and how to write Pythonic code. PEP 20, the Pythonic philosophy, is about code that is:

- Clean
- Simple
- Beautiful
- Explicit
- Readable

Most of these sound like common sense, and I think they should be. There are cases, however, where there is not a single obvious way to write your code (unless you're Dutch, of course, as you'll read later in this chapter). That is the goal of this chapter—to help you to learn how to write beautiful Python code and understand why certain decisions have been made in the Python style guide.

Let's get started.

Whitespace instead of braces

One of the most common complaints about Python for non-Python programmers is the use of whitespace instead of braces. Something can be said for both cases, and in the end, it doesn't matter that much. Since nearly every programming language already defaults to similar indenting rules even with braces, why not skip the braces altogether and make things more readable? That's what Guido van Rossum must have thought when designing the Python language.

At one point, some programmers asked Guido van Rossum whether Python would ever support braces. Since that day, braces have been available through a `__future__` import. Just give it a try:

```
>>> from __future__ import braces
```

Next, let's talk about formatting strings.

Formatting strings – printf, str.format, or f-strings?

Python has supported both the printf style (`%`) and `str.format` for a long time, so you are most likely familiar with both already. With the introduction of Python 3.6, an extra option became available, the f-string (PEP 498). The f-string is a convenient shorthand for `str.format`, which helps with brevity (and therefore, I would argue, readability).

PEP 498 – Literal String Interpolation: `https://peps.python.org/pep-0498/`

The previous edition of this book mainly used the printf style because brevity is important in code samples. While the maximum line length as per PEP 8 is 79 characters, this book is limited to 66 characters before wrapping occurs. With f-strings, we finally have a concise alternative to the printf style.

Tip for running the code in this book

Since a large portion includes the >>> prefix, simply copy/paste it into IPython and it will execute the code as regular Python code.

Alternatively, the GitHub repository for the book has a script to automatically convert a sample from doctest style to regular Python: `https://github.com/mastering-python/code_2/blob/master/doctest_to_python.py`

To show the power of f-strings, let's see a few examples of `str.format` and the printf style next to each other.

The examples in this chapter show the output as returned by the Python console. For a regular Python file, you need to add `print()` to see the output.

Simple formatting

Formatting a simple string:

```
# Simple formatting
>>> name = 'Rick'

>>> 'Hi %s' % name
'Hi Rick'

>>> 'Hi {}'.format(name)
'Hi Rick'
```

Formatting a floating-point number with two decimals:

```
>>> value = 1 / 3

>>> '%.2f' % value
'0.33'
```

```
>>> '{:.2f}'.format(value)
'0.33'
```

The first real advantage comes when using a variable multiple times. That is not possible with the printf style without resorting to named values:

```
>>> name = 'Rick'
>>> value = 1 / 3

>>> 'Hi {0}, value: {1:.3f}. Bye {0}'.format(name, value)
'Hi Rick, value: 0.333. Bye Rick'
```

As you can see, we used `name` twice by using the reference `{0}`.

Named variables

Using named variables is fairly similar and this is where we get introduced to the magic of f-strings:

```
>>> name = 'Rick'

>>> 'Hi %(name)s' % dict(name=name)
'Hi Rick'

>>> 'Hi {name}'.format(name=name)
'Hi Rick'

>>> f'Hi {name}'
'Hi Rick'
```

As you can see, with the f-strings, the variables are fetched from the scope automatically. It's basically a shorthand for:

```
>>> 'Hi {name}'.format(**globals())
'Hi Rick'
```

Arbitrary expressions

Arbitrary expressions are where the real power of f-strings becomes visible. The features of f-strings go far beyond the string interpolation of the printf-style features. The f-strings also support full Python expressions, which means they support complex objects, calling methods, `if` statements, and even loops:

```
## Accessing dict items
>>> username = 'wolph'
>>> a = 123
>>> b = 456
>>> some_dict = dict(a=a, b=b)
```

```
>>> f'''a: {some_dict['a']}'''
'a: 123'

>>> f'''sum: {some_dict['a'] + some_dict['b']}'''
'sum: 579'

## Python expressions, specifically an inline if statement
>>> f'if statement: {a if a > b else b}'
'if statement: 456'

## Function calls
>>> f'min: {min(a, b)}'
'min: 123'

>>> f'Hi {username}. And in uppercase: {username.upper()}'
'Hi wolph. And in uppercase: WOLPH'

## Loops
>>> f'Squares: {[x ** 2 for x in range(5)]}'
'Squares: [0, 1, 4, 9, 16]'
```

PEP 20, the Zen of Python

The Zen of Python, as mentioned in the *A brief history of Python* section earlier, is about code that not only works, but is Pythonic. Pythonic code is readable, concise, and maintainable. PEP 20 says it best:

> *"Long time Pythoneer Tim Peters succinctly channels the BDFL's guiding principles for Python's design into 20 aphorisms, only 19 of which have been written down."*

The next few paragraphs will explain the intentions of these 19 aphorisms with some example code.

For clarity, let's see these aphorisms before we begin:

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
```

```
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Beautiful is better than ugly

Beauty is subjective, of course, but there are still some style rules that are good to adhere to. Rules such as (from PEP 8):

- Indent using spaces instead of tabs
- Line length limits
- Each statement on a separate line
- Each import on a separate line

When in doubt, always keep in mind that consistency is more important than fixed rules. If a project prefers to use tabs instead of spaces, or vice versa, it's better to keep the tabs/spaces like that than to potentially break existing code (and revision control history) by replacing the tabs/spaces.

In short, instead of hard-to-read code like this, which shows all odd numbers below 10:

```
>>> filter_modulo = lambda i, m: (i[j] for j in \
...                               range(len(i)) if i[j] % m)
>>> list(filter_modulo(range(10), 2))
[1, 3, 5, 7, 9]
```

I would prefer:

```
>>> def filter_modulo(items, modulo):
...     for item in items:
...         if item % modulo:
...             yield item
...

>>> list(filter_modulo(range(10), 2))
[1, 3, 5, 7, 9]
```

It is simpler, easier to read, and a bit more beautiful!

These examples are an early introduction to generators. Generators will be discussed more thoroughly in *Chapter 7, Generators and Coroutines – Infinity, One Step at a Time*.

Explicit is better than implicit

Imports, arguments, and variable names are just some of the many cases where explicit code is far easier to read at the cost of a little bit more effort and/or verbosity when writing the code.

Here is an example of how this can go wrong:

```
>>> from os import *
>>> from asyncio import *

>>> assert wait
```

Where does `wait` come from, in this case? You might say that it's obvious—it comes from `os`. But you would be wrong, sometimes. On Windows, the `os` module doesn't have a `wait` function, so it would be `asyncio.wait` instead.

It could be even worse: many editors and code clean-up tools have a sort-imports feature. If the sort order of your import changes, the behavior of your project will change.

The immediate fix is simple enough:

```
>>> from os import path
>>> from asyncio import wait

>>> assert wait
```

With this method, we have at least a way to find out where `wait` came from. But I would recommend going a step further and importing by module instead, so the executing code immediately shows which function is executed:

```
>>> import os
>>> import asyncio

>>> assert asyncio.wait
>>> assert os.path
```

The same can be said for `*args` and `**kwargs`. While they are very useful, they can make the usage of your functions and classes a lot less obvious:

```
>>> def spam(eggs, *args, **kwargs):
...     for arg in args:
...         eggs += arg
```

```
...     for extra_egg in kwargs.get('extra_eggs', []):
...         eggs += extra_egg
...     return eggs

>>> spam(1, 2, 3, extra_eggs=[4, 5])
15
```

Without looking at the code within the function, you cannot know what to pass as `**kwargs` or what `*args` does. A reasonable function name can help here, of course:

```
>>> def sum_ints(*args):
...     total = 0
...     for arg in args:
...         total += arg
...     return total

>>> sum_ints(1, 2, 3, 4, 5)
15
```

Documentation can obviously help for cases like these, and I use `*args` and `**kwargs` very often, but it is definitely a good idea to keep at least the most common arguments explicit. Even when it requires you to repeat the arguments for a parent class, it just makes the code much clearer. When refactoring the parent class in the future, you'll know whether there are subclasses that still use some parameters.

Simple is better than complex

> *"Simple is better than complex. Complex is better than complicated."*

Keeping things simple is often much harder than you would expect. Complexity has a tendency to creep up on you. You start with a beautiful little script and, before you know it, feature creep has turned it into a complex (or worse, complicated) mess:

```
>>> import math
>>> import itertools

>>> def primes_complicated():
...     sieved = dict()
...     i = 2
...
...     while True:
...         if i not in sieved:
...             yield i
...             sieved[i * i] = [i]
...         else:
```

```
...             for j in sieved[i]:
...                 sieved.setdefault(i + j, []).append(j)
...             del sieved[i]
...
...         i += 1

>>> list(itertools.islice(primes_complicated(), 10))
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

At first glance, this code might look a bit difficult. If you're familiar with the sieve of Eratosthenes however, you'll quickly realize what is happening. With just a little bit of effort, you will see that the algorithm isn't all that complicated but uses a few tricks to reduce the necessary computations.

We can do better, however; let's see a different example featuring the Python 3.8 assignment operator:

```
>>> def primes_complex():
...     numbers = itertools.count(2)
...     while True:
...         yield (prime := next(numbers))
...         numbers = filter(prime.__rmod__, numbers)

>>> list(itertools.islice(primes_complex(), 10))
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

This algorithm looks a bit less intimidating, but I wouldn't call it immediately obvious at first glance. The `prime := next(numbers)` is the Python 3.8 version of setting a variable and immediately returning it in the same statement. The `prime.__rmod__` does a modulo with the given number to sieve in a similar fashion to the previous example.

What might be confusing, however, is that the `numbers` variable is being reassigned with added filters on each iteration. Let's see a better solution:

```
>>> def is_prime(number):
...     if number == 0 or number == 1:
...         return False
...     for modulo in range(2, number):
...         if not number % modulo:
...             return False
...     else:
...         return True

>>> def primes_simple():
...     for i in itertools.count():
...         if is_prime(i):
...             yield i
```

```
>>> list(itertools.islice(primes_simple(), 10))
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

Now we've come to one of the most obvious methods of generating prime numbers. The `is_prime` function is really simple and immediately shows what `is_prime` is doing. And the `primes_simple` function is nothing more than a loop with a filter.

Unless you have a really good reason to go for the complicated approach, try to keep your code as simple as you can. You (and perhaps someone else) will be grateful when reading your code in the future.

Flat is better than nested

Nested code quickly becomes unreadable and hard to understand. There are no strict rules here, but generally, when you have multiple levels of nested loops, it is time to refactor.

Just take a look at the following example, which prints a list of two-dimensional matrices. While nothing specific is wrong here, splitting it into a few more functions might make it easier to understand the purpose and also make it easier to test:

```
>>> def between_and_modulo(value, a, b, modulo):
...     if value >= a:
...         if value <= b:
...             if value % modulo:
...                 return True
...     return False

>>> for i in range(10):
...     if between_and_modulo(i, 2, 9, 2):
...         print(i, end=' ')
3 5 7 9
```

Here's the flatter version:

```
>>> def between_and_modulo(value, a, b, modulo):
...     if value < a:
...         return False
...     elif value > b:
...         return False
...     elif not value % modulo:
...         return False
...     else:
...         return True

>>> for i in range(10):
...     if between_and_modulo(i, 2, 9, 2):
```

```
...         print(i, end=' ')
3 5 7 9
```

This example might be a bit contrived, but the idea is sound. Deeply nested code can easily become very unreadable and splitting code into multiple lines or even functions can help readability a lot.

Sparse is better than dense

Whitespace is generally a good thing. Yes, it will make your files longer and your code will take up more space, but it can help a lot with readability if you split your code logically. Let's take an example:

```
>>> f=lambda x:0**x or x*f(x-1)
>>> f(40)
815915283247897734345611269596115894272000000000
```

By looking at the output and the code, you might be able to guess that this is the factorial function. But its workings are probably not immediately obvious. Let's try rewriting:

```
>>> def factorial(x):
...     if 0 ** x:
...         return 1
...     else:
...         return x * factorial(x - 1)

>>> factorial(40)
815915283247897734345611269596115894272000000000
```

By using a proper name, expanding the `if` statement, and explicitly returning `1`, it is suddenly much more obvious what is happening.

Readability counts

Shorter does not always mean easier to read. Let's take the Fibonacci numbers. There are many ways of writing this code, many of them hard to read:

```
>>> from functools import reduce

>>> fib=lambda n:n if n<2 else fib(n-1)+fib(n-2)
>>> fib(10)
55


>>> fib=lambda n:reduce(lambda x,y:(x[0]+x[1],x[0]),[(1,1)]*(n-1))[0]
>>> fib(10)
55
```

Even though there is a kind of beauty and elegance in the solutions, they are not readable. With just a few minor changes, we can change these functions to more readable functions that function similarly:

```
>>> def fib(n):
...     if n < 2:
...         return n
...     else:
...         return fib(n - 1) + fib(n - 2)

>>> fib(10)
55


>>> def fib(n):
...     a = 0
...     b = 1
...     for _ in range(n):
...         a, b = b, a + b
...
...     return a

>>> fib(10)
55
```

Practicality beats purity

> *"Special cases aren't special enough to break the rules. Although practicality beats purity."*

Breaking the rules can be tempting at times, but it's a slippery slope. If your quick fix is going to break the rules, you should really try to refactor it immediately. Chances are that you won't have the time to fix it later and will regret it.

No need to go overboard, though. If the solution is good enough and refactoring would be much more work, then choosing the working method might be better. Even though all of these examples pertain to imports, this guideline applies to nearly all cases.

To prevent long lines, imports can be made shorter by using a few methods, adding a backslash, adding parentheses, or just shortening the imports. I will illustrate some options next:

```
>>> from concurrent.futures import ProcessPoolExecutor, \
...     CancelledError, TimeoutError
```

This case can easily be avoided by using parentheses:

```
>>> from concurrent.futures import (
...     ProcessPoolExecutor, CancelledError, TimeoutError)
```

Or my personal preference, importing modules instead of the separate objects:

```
>>> from concurrent import futures
```

But what about really long imports?

```
>>> from concurrent.futures.process import \
...     ProcessPoolExecutor
```

In that case, I would recommend using parentheses. If you need to split the imports across multiple lines, I would recommend one line per import for readability:

```
>>> from concurrent.futures.process import (
...     ProcessPoolExecutor
... )


>>> from concurrent.futures import (
...     ProcessPoolExecutor,
...     CancelledError,
...     TimeoutError,
... )
```

Errors should never pass silently

> *"Errors should never pass silently. Unless explicitly silenced."*

Handling errors the right way is really difficult and there is no one method that works for every situation. There are, however, better and worse methods to catch errors.

Bare or too-broad exception catching can be a quick way to make your life a bit more difficult in the case of bugs. Not passing exception info at all can make you (or some other person working on the code) wonder for ages about what is happening.

To illustrate a bare exception, the worst option is as follows:

```
>>> some_user_input = '123abc'

>>> try:
...     value = int(some_user_input)
... except:
...     pass
```

A much better solution is to explicitly capture only the error you need:

```
>>> some_user_input = '123abc'

>>> try:
```

```
...     value = int(some_user_input)
... except ValueError:
...     pass
```

Alternatively, if you really need to capture all exceptions, make sure to log them properly:

```
>>> import logging

>>> some_user_input = '123abc'

>>> try:
...     value = int(some_user_input)
... except Exception as exception:
...     logging.exception('Uncaught: {exception!r}')
```

When using multiple lines inside a `try` block, the issue of tracing bugs is aggravated even further because there is even more code that could be responsible for the hidden exception. The tracing of bugs also becomes much more difficult when the `except` is accidently capturing exceptions from functions a few levels deep. For example, consider the following code block:

```
>>> some_user_input_a = '123'
>>> some_user_input_b = 'abc'

>>> try:
...     value = int(some_user_input_a)
...     value += int(some_user_input_b)
... except:
...     value = 0
```

If an exception is raised, which line is causing it? With silent catching of the error, there is no way to know without running the code in a debugger. The exception could even be caused a few levels deeper in the code if, instead of `int()`, you are using a more complex function.

If you are testing for a specific exception in a specific block of code, the safer method is using the `else` in the `try/except`. The `else` is only executed if there was no exception.

To illustrate the full strength of the `try/except:`, here is an example of all variants including the `else`, `finally`, and `BaseException`:

```
>>> try:
...     1 / 0  # Raises ZeroDivisionError
... except ZeroDivisionError:
...     print('Got zero division error')
... except Exception as exception:
...     print(f'Got unexpected exception: {exception}')
... except BaseException as exception:
```

```
...     # Base exceptions are a special case for keyboard
...     # interrupts and a few other exceptions that are not
...     # technically errors.
...     print(f'Got base exception: {exception}')
... else:
...     print('No exceptions happened, we can continue')
... finally:
...     # Useful cleanup functions such as closing a file
...     print('This code is _always_ executed')
Got zero division error
This code is _always_ executed
```

In the face of ambiguity, refuse the temptation to guess

While guesses will work in many cases, they can bite you if you're not careful. As already demonstrated in the *Explicit is better than implicit* section, when you have a few `from ... import *`, you cannot always be certain which module is providing you with the variable you were expecting.

Clear and unambiguous code generates fewer bugs so it's always a good idea to think about what happens when someone else reads your code. A prime example of ambiguity is function calling. Take, for example, the following two function calls:

```
>>> fh_a = open('spam', 'w', -1, None, None, '\n')
>>> fh_b = open(file='spam', mode='w', buffering=-1, newline='\n')
```

These two calls have the exact same result. However, it's obvious in the second call that the `-1` is configuring the buffer. You probably know the first two arguments of `open()` by heart but the others are less common.

Regardless, without seeing `help(open)` or viewing the documentation in another manner, it's impossible to say whether the two are identical.

Note that I don't think you should use keyword arguments in all cases, but if there are many arguments involved and/or hard-to-identify parameters (such as numbers), it can be a good idea. A good alternative is using good variable names, which make the function call a lot more obvious:

```
>>> filename = 'spam'
>>> mode = 'w'
>>> buffers = -1

>>> fh_b = open(filename, mode, buffers, newline='\n')
```

One obvious way to do it

> *"There should be one—and preferably only one—obvious way to do it. Although that way may not be obvious at first unless you're Dutch."*

In general, after thinking about a difficult problem for a while, you will find that there is one solution that is clearly preferable over the alternatives. There are times where this is not the case, however, and in such instances, it can be useful if you're Dutch. The joke here is that Guido van Rossum, the original author of Python, is Dutch (as am I) and that only Guido knows the obvious way in some cases.

The other joke is that the Perl programming language slogan is the opposite: "There's more than one way to do it."

Now is better than never

> *"Now is better than never. Although never is often better than *right* now."*

It's better to fix a problem right now than push it into the future. There are cases, however, where fixing it right away is not an option. In those cases, a good alternative can be to mark a function as deprecated instead so that there is no chance of accidentally forgetting the problem:

```
>>> import warnings

>>> warnings.warn('Something deprecated', DeprecationWarning)
```

Hard to explain, easy to explain

> *"If the implementation is hard to explain, it's a bad idea. If the implementation is easy to explain, it may be a good idea."*

As always, keep things as simple as you can. While complicated code can be nice to test with, it is more prone to bugs. The simpler you can keep things, the better.

Namespaces are one honking great idea

> *"Namespaces are one honking great idea—let's do more of those!"*

Namespaces can make code a lot clearer to use. Naming them properly makes it even better. For example, assume the `import` isn't on your screen in a larger file. What does the `loads` line do?

```
>>> from json import loads

>>> loads('{}')
{}
```

Now let's take the version with the namespace:

```
>>> import json

>>> json.loads('{}')
{}
```

Now it is obvious that `loads()` is the `json` loader and not any other type of loader.

Namespace shortcuts are still useful, though. Let's look at the `User` class in Django, which is used in nearly every Django project. The `User` class is stored in `django.contrib.auth.models.User` by default (can be overridden). Many projects use the object in the following way:

```
from django.contrib.auth.models import User
# Use it as: User
```

While this is fairly clear, projects might be using multiple classes named `User`, which obscures the import. Also, it might make someone think that the `User` class is local to the current class. Doing the following instead lets people know that it is in a different module:

```
from django.contrib.auth import models
# Use it as: models.User
```

This quickly clashes with other models' imports, though, so I personally use the following instead:

```
from django.contrib.auth import models as auth_models
# Use it as auth_models.User
```

Or the shorter version:

```
import django.contrib.auth.models as auth_models
# Use it as auth_models.User
```

Now you should have some idea of what the Pythonic ideology is about—creating code that is:

- Beautiful
- Readable
- Unambiguous
- Explicit enough
- Not completely void of whitespace

So let's move on to some more examples of how to create beautiful, readable, and simple code using the Python style guide.

Explaining PEP 8

The previous sections have already shown a lot of examples of using PEP 20 as a reference, but there are a few other important guidelines to note as well. The PEP 8 style guide specifies the standard Python coding conventions.

Simply following the PEP 8 standard doesn't make your code Pythonic, though, but it is most certainly a good start. Which style you use is really not that much of a concern as long as you are consistent. The only thing worse than not using a proper style guide is being inconsistent with it.

Duck typing

Duck typing is a method of handling variables by behavior. To quote Alex Martelli (one of my Python heroes, also nicknamed the MartelliBot by many):

> *"Don't check whether it IS-a duck: check whether it QUACKS-like-a duck, WALKS-like-a duck, etc, etc, depending on exactly what subset of duck-like behavior you need to play your language-games with. If the argument fails this specific-ducklyhood-subset-test, then you can shrug, ask "why a duck?"*

In many cases, when people make a comparison such as `if spam != '':`, they are actually just looking for anything that is considered a true value. While you can compare the value to the string value `''`, you generally don't have to make it so specific. In many cases, simply doing `if spam:` is more than enough and actually functions better.

For example, the following lines of code use the value of `timestamp` to generate a filename:

```
>>> timestamp = 12345

>>> filename = f'{timestamp}.csv'
```

Because the variable is named `timestamp`, you might be tempted to check whether it is actually a `date` or `datetime` object, like this:

```
>>> import datetime

>>> timestamp = 12345

>>> if isinstance(timestamp, datetime.datetime):
...     filename = f'{timestamp}.csv'
... else:
...     raise TypeError(f'{timestamp} is not a valid datetime')
Traceback (most recent call last):
...
TypeError: 12345 is not a valid datetime
```

While this is not inherently wrong, comparing types is considered a bad practice in Python, as there is often no need.

In Python, the commonly used style is **EAFP** (**easier to ask for forgiveness than permission**: `https://docs.python.org/3/glossary.html#term-eafp`), which assumes no errors but catches them if needed. Within the Python interpreter, a `try`/`except` block is extremely efficient if no exception is raised. Actually catching an exception is expensive, however, so this approach is mainly recommended when you don't expect the `try` to fail often.

The opposite of EAFP is **LBYL** (**look before you leap**: `https://docs.python.org/3/glossary.html#term-lbyl`), which tests for pre-conditions before other calls or lookups are made. The notable downside of this method is the potential for race conditions in multi-threaded environments. While you are checking for the existence of a key in a `dict`, another thread may have removed it already.

That's why in Python, duck typing is often preferred. Just test the variable for the features you need and don't worry about the actual type. To illustrate how little difference this can make to the end result, see the following code:

```
>>> import datetime

>>> timestamp = datetime.date(2000, 10, 5)
>>> filename = f'{timestamp}.csv'
>>> print(f'Filename from date: {filename}')
Filename from date: 2000-10-05.csv
```

Versus a string instead of a `date`:

```
>>> timestamp = '2000-10-05'
>>> filename = f'{timestamp}.csv'
>>> print(f'Filename from str: {filename}')
Filename from str: 2000-10-05.csv
```

As you can see, the result is identical.

The same goes for converting a number to a float or an integer; instead of enforcing a certain type, just require certain features. Need something that can pass as a number? Just try to convert to `int` or `float`. Need a `file` object? Why not just check whether there is a `read` method with `hasattr`?

Differences between value and identity comparisons

There are many methods of comparing objects in Python: greater than, bitwise operators, equal, etc., but there is one comparator that is special: the identity comparison operator. Instead of using `if spam == eggs`, you would use `if spam is eggs`. The first compares the value and the second compares the identity or **memory address**. Because it only compares the memory address, it's one of the lightest and strictest lookups you can get. Whereas a value check needs to make sure that the types are comparable and perhaps check the sub-values, the identity check just checks whether the unique identifier is the same.

If you've ever written Java, you should be familiar with this principle. In Java, a regular string comparison (`spam == eggs`) will use the identity instead of the value. To compare the value, you need to use `spam.equals(eggs)` to get the correct results.

These comparisons are recommended to be used when the identity of the object is expected to be constant. One obvious example of this is a comparison with `True`, `False`, or `None`. To demonstrate this behavior, let's look at values that evaluate to `True` or `False` when comparing by value, but are actually different:

```
>>> a = 1
>>> a == True
True
>>> a is True
False

>>> b = 0
>>> b == False
True
>>> b is False
False
```

Similarly, you need to be careful with `if` statements and `None` values, which is a common pattern with default function arguments:

```
>>> def some_unsafe_function(arg=None):
...     if not arg:
...         arg = 123
...
...     return arg

>>> some_unsafe_function(0)
123
>>> some_unsafe_function(None)
123
```

The second one indeed needed the default argument, but the first one had an actual value that should have been used:

```
>>> def some_safe_function(arg=None):
...     if arg is None:
...         arg = 123
...
```

```
...     return arg

>>> some_safe_function(0)
0
>>> some_safe_function(None)
123
```

Now we actually get the value that we passed along because we used an identity instead of a value check for `arg`.

There are a few gotchas with the identities, though. Let's look at an example that doesn't make any sense:

```
>>> a = 200 + 56
>>> b = 256
>>> c = 200 + 57
>>> d = 257

>>> a == b
True
>>> a is b
True
>>> c == d
True
>>> c is d
False
```

While the values are the same, the identities are different. The catch is that Python keeps an internal array of integer objects for all integers between `-5` and `256`; that's why it works for `256` but not for `257`.

To look at what Python is actually doing internally with the `is` operator, you can use the `id` function. When executing `if spam is eggs`, Python will execute the equivalent of `if id(spam) == id(eggs)` internally and `id()` (at least for CPython) returns the memory address.

Loops

Coming from other languages, one might be tempted to use `for` loops or `while` loops with counters to process the items of a `list`, `tuple`, `str`, and so on. While valid, it is more complex than needed. For example, consider this code:

```
>>> my_range = range(5)
>>> i = 0
>>> while i < len(my_range ):
...     item = my_range [i]
...     print(i, item, end=', ')
...     i += 1
0 0, 1 1, 2 2, 3 3, 4 4,
```

Within Python, there is no need to build a custom loop: you can simply loop the iterable object instead. Although enumerating including a counter is easily possible too:

```
>>> my_range  = range(5)
>>> for item in my_range :
...     print(item, end=', ')
0, 1, 2, 3, 4,

>>> for i, item in enumerate(my_range ):
...     print(i, item, end=', ')
0 0, 1 1, 2 2, 3 3, 4 4,
```

This can be written even shorter, of course (albeit not 100% identically, since we're not using `print`), but I wouldn't recommend that for the sake of readability in most cases:

```
>>> my_range  = range(5)
>>> [(i, item) for i, item in enumerate(my_range)]
[(0, 0), (1, 1), (2, 2), (3, 3), (4, 4)]
```

The last option might be clear to some but not all. A common recommendation is to limit the usage of `list`/`dict`/`set` comprehensions and `map`/`filter` statements to cases where the entire statement fits on a single line.

Maximum line length

Many Python programmers think 79 characters is too constricting and just keep the lines longer. While I am not going to argue for 79 characters specifically, setting a low limit is a good idea so you can easily keep multiple editors side by side. I often have four Python files open next to each other. If the line width were more than 79 characters, that simply wouldn't fit.

PEP 8 tells us to use backslashes in cases where lines get too long. While I agree that backslashes are preferable over long lines, I still think they should be avoided, if possible, since they easily generate syntax errors when manipulating code by copying/pasting and rearranging. Here's an example from PEP 8:

```
with open('/path/to/some/file/you/want/to/read') as file_1, \
        open('/path/to/some/file/being/written', 'w') as file_2:
    file_2.write(file_1.read())
```

Instead of using backslashes, I would reformat the code by introducing extra variables so all lines are easy to read:

```
filename_1 = '/path/to/some/file/you/want/to/read'
filename_2 = '/path/to/some/file/being/written'
with open(filename_1) as file_1, open(filename_2, 'w') as file_2:
    file_2.write(file_1.read())
```

Or in this specific case of filenames, by using `pathlib`:

```
import pathlib
filename_1 = pathlib.Path('/path/to/some/file/you/want/to/read')
filename_2 = pathlib.Path('/path/to/some/file/being/written')
with filename_1.open() as file_1, filename_2.open('w') as file_2:
    file_2.write(file_1.read())
```

This is not always an option, of course, but it's a good consideration to keep the code short and readable. It actually provides a bonus of adding more information to the code. If, instead of `filename_1`, you use a name that conveys the goal of the filename, it immediately becomes clearer what you are trying to do.

Verifying code quality, pep8, pyflakes, and more

There are many tools for checking code quality and style in Python. The options range from `pycodestyle` (previously named `pep8`) for checking rules pertaining to PEP 8, to tools such as `flake8`, which bundles a lot of tools and can help refactor code and track down bugs in code that appears to work.

Let's go into more detail.

pycodestyle/pep8

The `pycodestyle` package (previously named `pep8`) is the default code style checker to start with. The `pycodestyle` checker attempts to validate many of the rules suggested in PEP 8 that are considered to be the standard by the community. It doesn't check everything that is in the PEP 8 standard, but it goes a long way and is still updated regularly to add new checks. Some of the most important things checked by `pycodestyle` are as follows:

- Indentation: While Python will not check how many spaces you use to indent, it does not help with the readability of your code
- Missing whitespace, such as `spam=123`
- Too much whitespace, such as `def eggs(spam = 123):`
- Too many or too few blank lines
- Too long lines
- Syntax and indentation errors
- Incorrect and/or superfluous comparisons (`not in`, `is not`, `if spam is True`, and type comparisons without `isinstance`)

If some of the specific rules are not to your liking, you can easily tweak them to fit your purpose. Beyond that, the tool is not too opinionated, which makes it an ideal starting point for any Python project.

An honorable mention goes out to the `black` project, which is a Python formatter that automatically formats your code to largely adhere to the PEP 8 style. The name `black` stems from Henry Ford's quote: "Any customer can have a car painted any color that he wants so long as it is black."

That immediately shows the downside of `black`: it offers very little in the way of customization. If you don't like one of the rules, you are most likely out of luck.

pyflakes

The `pyflakes` checker is meant to detect errors and potential bugs in your code by parsing (not importing) the code. This makes it ideal for editor integration, but it can also be used to warn you about potential issues in your code beyond that. It will warn you about:

- Unused imports
- Wildcard imports (`from module import *`)
- Incorrect `__future__` imports (after other imports)

More importantly, it warns you about potential bugs, such as the following:

- Redefinitions of names that were imported
- Usage of undefined variables
- Referencing variables before assignment
- Duplicate argument names
- Unused local variables

pep8-naming

The last bit of PEP 8 is covered by the `pep8-naming` package. It makes sure that your naming is close to the standard dictated by PEP 8:

- Class names as `CapWord`
- Function, variable, and argument names all in lowercase
- Constants as full uppercase and being treated as constants
- The first argument of instance methods and class methods as `self` and `cls`, respectively

McCabe

Lastly, there is the McCabe complexity. It checks the complexity of code by looking at the **Abstract Syntax Tree** (**AST**), which Python builds from the source code internally. It finds out how many lines, levels, and statements are there and warns you if your code has more complexity than a preconfigured threshold. Generally, you will use McCabe through `flake8`, but a manual call is possible as well. Using the following code:

```
def noop():
    pass
```

```
def yield_cube_points(matrix):
    for x in matrix:
        for y in x:
            for z in y:
                yield (x, y, z)

def print_cube(matrix):
    for x in matrix:
        for y in x:
            for z in y:
                print(z, end='')
            print()
        print()
```

McCabe will give us the following output:

```
$ pip3 install mccabe
...
$ python3 -m mccabe T_16_mccabe.py
1:0: 'noop' 1
5:0: 'yield_cube_points' 4
12:0: 'print_cube' 4
```

At first, when you look at the `1` generated by `noop`, you might think `mccabe` counts the lines of code. Upon further inspection, you can see this isn't the case. Having multiple `noop` operators does not increase the count and nor do the `print` statements in the `print_cube` function.

The `mccabe` tool checks the cyclomatic complexity of code. In a nutshell, this means that it counts the number of possible execution paths. Code without any control flow statements such as `if`/`for`/`while` counts as 1, as you can see in the `noop` function. A simple `if` or `if`/`else` results in two options: one where the `if` statement is `True` and one where the `if` statement is `False`. If there is a nested `if` or an `elif`, this would increase further. Loops count as 2 since there is the flow of going inside the loop if there are items, and not going into the loop if there are no items.

The warning threshold for `mccabe` is set to 10 by default, but is configurable. If your code actually has a score of more than 10, it is time for some refactoring. Remember the advice from PEP 20.

Mypy

Mypy is a tool used to check the variable types within your code. While specifying fixed types goes against duck typing, there are certainly cases where this is useful and where it will protect you from bugs.

Taking the following code, for example:

```
some_number: int
some_number = 'test'
```

The `mypy` command will tell us we've made a mistake:

```
$ mypy T_17_mypy.py
T_17_mypy.py:2: error: Incompatible types in assignment (expression has type
"str", variable has type "int")
Found 1 error in 1 file (checked 1 source file)
```

Note that this syntax depends on the type hinting introduced in Python 3.5. For older Python versions, you can use comments for type hints instead:

```
some_number = 'test'  # type: int
```

Even if you're not using code hinting in your own code, this can still be useful to check whether your calls to external libraries are correct. If the arguments for a function of an external library changed with an update, this can quickly tell you something is wrong at the location of the mistake instead of having to trace a bug throughout your code.

flake8

To run all of these tests combined, you can use `flake8`, a tool that runs `pycodestyle`, `pyflakes`, and `mccabe` by default. After running these commands, `flake8` combines their outputs into a single report. Some of the warnings generated by `flake8` might not fit your taste, so each and every one of the checks can be disabled, both per file and for the entire project if needed. For example, I personally disable `W391` for all my projects, which warns you about blank lines at the end of a file.

This is something I find useful while working on code so that I can easily jump to the end of the file and start writing code instead of having to append a few lines first.

There are also many plugins available to make `flake8` even more powerful.

Some example plugins are:

- `pep8-naming`: Tests PEP naming conventions
- `flake8-docstrings`: Tests whether docstrings follow the PEP 257, NumPy, or Google convention. More about these conventions will be in the chapter about documentation.
- `flake8-bugbear`: Finds likely bugs and design problems in your code, such as bare excepts.
- `flake8-mypy`: Tests whether the types of values are consistent with the declared types.

In general, before committing your code and/or putting it online, just run `flake8` from your source directory to check everything recursively.

Here is a demonstration with some poorly formatted code:

```
def spam(a,b,c):print(a,b+c)
def eggs():pass
```

It results in the following:

```
$ pip3 install flake8
...
$ flake8 T_18_flake8.py
T_18_flake8.py:1:11: E231 missing whitespace after ','
T_18_flake8.py:1:13: E231 missing whitespace after ','
T_18_flake8.py:1:16: E231 missing whitespace after ':'
T_18_flake8.py:1:24: E231 missing whitespace after ','
T_18_flake8.py:2:11: E231 missing whitespace after ':'
```

Recent additions to the Python syntax

The Python syntax has remained largely unchanged in the last decade, but we have seen a few additions, such as the f-strings, type hinting, and async functions, of course. We already covered f-strings at the beginning of this chapter, and the other two are covered by *Chapter 9* and *Chapter 13*, respectively, but there have a been a few other recent additions to the Python syntax that you might have missed. Additionally, in *Chapter 4* you will see the dictionary merge operators added in Python 3.9.

PEP 572: Assignment expressions/the walrus operator

We already covered this briefly earlier in this chapter, but since Python 3.8, we have assignment expressions. If you have experience with C or C++, you have most likely seen something like this before:

```
if((fh = fopen("filename.txt", "w")) == NULL)
```

Within C, this opens a file using `fopen()`, stores the result of `fopen()` in `fh`, and checks whether the result of the `fopen()` call is `NULL`. Until Python 3.8, we always had to split these two operations into an assignment and an `if` statement, assuming we also had `fopen()` and `NULL` available in our Python code:

```
fh = fopen("filename.txt", "w")
if fh == NULL:
```

Since Python 3.8, we can use assignment expressions to do this in a single line, similar to C:

```
if (fh := fopen("filename.txt", "w")) == NULL:
```

With the := operator you can assign and check the result in one operation. This can be useful when reading user input, for example:

```
    while (line := input('Please enter a line: ')) != '':
        # Process the line here
    # The last line was empty, continue the script
```

This operator is often called the walrus operator because it looks slightly like the eyes and tusks of a walrus (:=).

PEP 634: Structural pattern matching, the switch statement

Many programmers who are new to Python wonder why it does not have a switch statement like most common programming languages. Often the lack of a switch statement has been addressed with dictionary lookups or, simply, a chain of `if`/`elif`/`elif`/`elif`/`else` statements. While those solutions work fine, I personally feel that at times my code could have been prettier and more readable with a switch statement.

Since Python 3.10, we finally have a feature that is very comparable to a switch statement but so much more powerful. As is the case with the Python ternary operator (i.e. `true_value if condition else false_value`), the syntax is far from a literal copy of other languages. In this case, especially, this is for the better. With most programming languages, it can be really easy to forget the `break` statement in a switch, which can cause unintended side effects.

At a glance, the Python implementation appears much simpler in syntax and features. Without the `break` statement, you might wonder how you can match multiple patterns in a single go. Stay tuned and find out! The pattern matching feature is *very powerful* and offers many more features than you might expect.

The basic match statement

First, let's look at a basic example. This one offers little benefit but can still be easier to read than a regular `if`/`elif`/`else` statement:

```
>>> some_variable = 123

>>> match some_variable:
...     case 1:
...         print('Got 1')
...     case 2:
...         print('Got 2')
...     case _:
...         print('Got something else')
Got something else

>>> if some_variable == 1:
...     print('Got 1')
... elif some_variable == 1:
...     print('Got 2')
... else:
...     print('Got something else')
Got something else
```

Since we have both the `if` and the `match` statement here, you can easily compare them. In this case, I would go for the `if` statement, but the main advantage of not having to repeat the `some_variable ==` part can still be useful.

The _ is the special wild card case for the match statement. It matches any value, so it can be seen as the equivalent of the `else` statement.

Storing the fallback as a variable

A slightly more useful example is to automatically store the result when it doesn't match. The previous example uses an underscore (_), which is not actually stored in _ because it is a special case, but if we name the variable differently, we can store the result:

```
>>> some_variable = 123

>>> match some_variable:
...     case 1:
...         print('Got 1')
...     case other:
...         print('Got something else:', other)
Got something else: 123
```

In this case we store the `else` case in the `other` variable. Note that you cannot use _ and a variable name at the same time since they do the same thing, which would be useless.

Matching from variables

You saw that a case such as `case other:` will store the result in `other` instead of comparing it with the value of `other`, so you might be wondering if we can do the equivalent of:

```
if some_variable == some_value:
```

The answer is that we can, with a caveat. Since any bare `case variable:` will result in storing into a variable, we need to have something that does not match that pattern. The common way to work around this limitation is by introducing a dot:

```
>>> class Direction:
...     LEFT = -1
...     RIGHT = 1

>>> some_variable = Direction.LEFT

>>> match some_variable:
...     case Direction.LEFT:
...         print('Going left')
...     case Direction.RIGHT:
...         print('Going right')
Going left
```

As long as it cannot be interpreted as a variable name, this will work for you. When comparing with a local variable, an `if` statement can always be used as well, of course.

Matching multiple values in a single case

If you're familiar with the `switch` statement in many other programming languages, you might be wondering whether you can have multiple `case` statements before you `break`, like this, for example (C++):

```
switch(variable){
    case Direction::LEFT:
    case Direction::RIGHT:
        cout << "Going horizontal" << endl;
        break;
    case Direction::UP:
    case Direction::DOWN:
        cout << "Going vertical" << endl;
}
```

This roughly means that if `variable` is either equal to `LEFT` or `RIGHT`, print the `"Going horizontal"` line and `break`. Since the Python `match` statement does not have a `break`, how can we match something like this? Well, some syntax was introduced specifically for that:

```
>>> class Direction:
...     LEFT = -1
...     UP = 0
...     RIGHT = 1
...     DOWN = 2

>>> some_variable = Direction.LEFT

>>> match some_variable:
...     case Direction.LEFT | Direction.RIGHT:
...         print('Going horizontal')
...     case Direction.UP | Direction.DOWN:
...         print('Going vertical')
Going horizontal
```

As you can see, using the `|` operator (which is also used for bitwise operations), you can test for multiple values at the same time.

Matching values with guards or extra conditions

There are times when you want a more advanced comparison such as `if variable > value:`. Luckily, even that is possible with the `match` statement using a feature called guards:

```
>>> values = -1, 0, 1

>>> for value in values:
...     print('matching', value, end=': ')
```

```
...     match value:
...         case negative if negative < 0:
...             print(f'{negative} is smaller than 0')
...         case positive if positive > 0:
...             print(f'{positive} is greater than 0')
...         case _:
...             print('no match')
matching -1: -1 is smaller than 0
matching 0: no match
matching 1: 1 is greater than 0
```

Note that this uses the variable name that I just introduced, but it's a regular Python expression, so you could also compare something else. However, you always need to have the variable name before the `if`. This will *not* work: `case if ...`.

Matching lists, tuples, and other sequences

If you are familiar with `tuple` unpacking, you can probably guess how sequence matching works:

```
>>> values = (0, 1), (0, 2), (1, 2)

>>> for value in values:
...     print('matching', value, end=': ')
...     match value:
...         case 0, 1:
...             print('exactly matched 0, 1')
...         case 0, y:
...             print(f'matched 0, y with y: {y}')
...         case x, y:
...             print(f'matched x, y with x, y: {x}, {y}')
matching (0, 1): exactly matched 0, 1
matching (0, 2): matched 0, y with y: 2
matching (1, 2): matched x, y with x, y: 1, 2
```

The first case explicitly matches both of the given values, which is identical to `if value == (0, 1):`.

The second case explicitly matches `0` for the first value, but leaves the second value as a variable and stores it in `y`. Effectively this comes down to `if value[0] == 0: y = value[1]`.

The last case stores a variable for both the `x` and `y` values and will match any sequence with exactly two items.

Matching sequence patterns

If you thought the previous example with the unpacking of the variables was useful, you will love this section. One of the really powerful features of the `match` statement is matching based on patterns.

Let's assume we have a function that takes up to three parameters, `host`, `port`, and `protocol`. For `port` and `protocol`, we can assume `443` and `https`, respectively, so that only leaves the `hostname` as a required parameter. How can we match this so one, two, three, or more parameters are all supported and work correctly? Let's find out:

```
>>> def get_uri(*args):
...     # Set defaults so we only have to store changed variables
...     protocol, port, paths = 'https', 443, ()
...     match args:
...         case (hostname,):
...             pass
...         case (hostname, port):
...             pass
...         case (hostname, port, protocol, *paths):
...             pass
...         case _:
...             raise RuntimeError(f'Invalid arguments {args}')
...
...     path = '/'.join(paths)
...     return f'{protocol}://{hostname}:{port}/{path}'

>>> get_uri('localhost')
'https://localhost:443/'
>>> get_uri('localhost', 12345)
'https://localhost:12345/'
>>> get_uri('localhost', 80, 'http')
'http://localhost:80/'
>>> get_uri('localhost', 80, 'http', 'some', 'paths')
'http://localhost:80/some/paths'
```

As you can see, the `match` statement also handles different length sequences, which is a very useful tool to have. You could do this with `if` statements as well, but I've never found a way to handle that in a really pretty fashion. Naturally you could still combine this with the earlier examples, so you could have a `case` such as: `case (hostname, port, 'http'):` if you want to invoke specific behavior. You can also apply `*variable` to capture all extra variables. The `*` matches 0 or more extra items in the sequence.

Capturing sub-patterns

In addition to specifying a variable name to save all values into, you can also store explicit value matches:

```
>>> values = (0, 1), (0, 2), (1, 2)
```

```
>>> for value in values:
...     print('matching', value, end=': ')
...     match value:
...         case 0 as x, (1 | 2) as y:
...             print(f'matched x, y with x, y: {x}, {y}')
...         case _:
...             print('no match')
matching (0, 1): matched x, y with x, y: 0, 1
matching (0, 2): matched x, y with x, y: 0, 2
matching (1, 2): no match
```

In this case we explicitly match `0` as the first part of `value`, and `1` or `2` as the second part of `value`. And we store those in the variables `x` and `y`, respectively.

It is important to note here that within the context of a `case` statement the `|` operator will always work as a or for the `case`, instead of a bitwise or for the variables/values. Normally `1 | 2` would result in `3` because in binary `1 = 0001`, `2 = 0010` , and the combination of those is `3 = 0011`.

Matching dictionaries and other mappings

Naturally it is also possible to match mappings (such as `dict`) by key:

```
>>> values = dict(a=0, b=0), dict(a=0, b=1), dict(a=1, b=1)

>>> for value in values:
...     print('matching', value, end=': ')
...     match value:
...         case {'a': 0}:
...             print('matched a=0:', value)
...         case {'a': 0, 'b': 0}:
...             print('matched a=0, b=0:', value)
...         case _:
...             print('no match')
matching {'a': 0, 'b': 0}: matched a=0: {'a': 0, 'b': 0}
matching {'a': 0, 'b': 1}: matched a=0: {'a': 0, 'b': 1}
matching {'a': 1, 'b': 1}: no match
```

Note that `match` only checks for the given keys and values and does not care about extra keys in the mapping. This is why the first case matches both of the first two items.

As you can see in the preceding example, matching happens sequentially and it will stop at the first match, not the best match. The second case is never reached in this scenario.

Matching using isinstance and attributes

If you thought the previous examples of the `match` statement were impressive, get ready to be completely amazed. The way the `match` statement can match instances including properties is amazingly powerful and can be incredibly useful. Just look at the following example and try to understand what is happening:

```
>>> class Person:
...     def __init__(self, name):
...         self.name = name

>>> values = Person('Rick'), Person('Guido')

>>> for value in values:
...     match value:
...         case Person(name='Rick'):
...             print('I found Rick')
...         case Person(occupation='Programmer'):
...             print('I found a programmer')
...         case Person() as person:
...             print('I found a person:', person.name)
I found Rick
I found a person: Guido
```

While I will admit that the syntax is slightly confusing and, dare I say it, unPythonic, it is so useful that it still makes sense.

Firstly, we will look at the `case Person() as person:`. We're discussing this first because it is important to understand what is happening here before we continue with the other examples. This line is effectively identical to `if isinstance(value, Person):`. It does *not* actually instantiate the `Person` class at this point, which is a bit confusing.

Secondly, the `case Person(name='Rick')` matches the instance type `Person` and it requires the instance to have an attribute `name` with value `Rick`.

Lastly, the `case Person(occupation='Programmer')` matches `value` to be a `Person` instance and have an attribute called `occupation` with the value `Programmer`. Since that attribute does not exist, it ignores that issue silently.

Note that this also works for built-in types and supports nesting:

```
>>> class Person:
...     def __init__(self, name):
...         self.name = name

>>> value = Person(123)
```

```
>>> match value:
...     case Person(name=str() as name):
...         print('Found person with str name:', name)
...     case Person(name=int() as name):
...         print('Found person with int name:', name)
Found person with int name: 123
```

We have covered several examples of how the new pattern matching feature works, but you could think of many more. Since all parts can be nested, the possibilities really are endless. It might not be the perfect solution for everything, and the syntax might feel a little odd, but it is such a powerful solution that I would recommend any Python programmer learns it by heart.

Common pitfalls

Python is a language meant to be clear and readable without any ambiguities and unexpected behaviors. Unfortunately, these goals are not achievable in all cases, and that is why Python does have a few corner cases where it might do something different than what you were expecting.

This section will show you some issues that you might encounter when writing Python code.

Scope matters!

There are a few cases in Python where you might not be using the scope that you are actually expecting. Some examples are when declaring a class and with function arguments, but the most annoying one is accidentally trying to overwrite a `global` variable.

Global variables

A common problem when accessing variables from the `global` scope is that setting a variable makes it local, even when accessing the `global` variable.

This works:

```
>>> g = 1

>>> def print_global():
...     print(f'Value: {g}')

>>> print_global()
Value: 1
```

But the following does not:

```
>>> g = 1

>>> def print_global():
...     g += 1
```

```
...     print(f'Value: {g}')

>>> print_global()
Traceback (most recent call last):
    ...
UnboundLocalError: local variable 'g' referenced before assignment
```

The problem is that `g += 1` actually translates to `g = g + 1`, and anything containing `g =` makes the variable local to your scope. Since the local variable is being assigned at that point, it has no value yet and you are trying to use it.

For these cases, there is the `global` statement, although it is generally recommended to avoid writing to `global` variables altogether because it can make your life a lot more difficult while debugging. Modern editors can help a lot to track who or what is writing to your `global` variables, but restructuring your code so it purposefully passes and modifies values in a clear path can help you to avoid many bugs.

Pass by reference with mutable variables

Within Python, variables are passed by reference. This means that when you do something like `x = y`, both `x` and `y` will point to the same variable. When you change the value (not the object) of either `x` or `y`, the other will change as well.

Since most variable types such as strings, integers, floats, and tuples are immutable, this is not a problem. Doing `x = 123` won't affect `y` since we aren't changing the value of `x`, but we are replacing `x` with a new object that has the value `123`.

With mutable variables, however, we can change the value of the object. Let's illustrate this behavior and how to work around it:

```
>>> x = []
>>> y = x
>>> z = x.copy()

>>> x.append('x')
>>> y.append('y')
>>> z.append('z')

>>> x
['x', 'y']
>>> y
['x', 'y']
>>> z
['z']
```

Unless you explicitly copy the variable as we did with `z`, your new variable will point to the same object.

Now you might be wondering whether `copy()` always works. As you might suspect, it doesn't. The `copy()` function only copies the object itself, not the values within the object. For that we have `deepcopy()`, which even handles recursion safely:

```
>>> import copy

>>> x = [[1], [2, 3]]
>>> y = x.copy()
>>> z = copy.deepcopy(x)

>>> x.append('a')
>>> x[0].append(x)

>>> x
[[1, [...]], [2, 3], 'a']
>>> y
[[1, [...]], [2, 3]]
>>> z
[[1], [2, 3]]
```

Mutable function default arguments

While the issues with mutable arguments can be easily avoided and seen in most cases, the scenario of default arguments for functions is a lot less obvious:

```
>>> def append(list_=[], value='value'):
...     list_.append(value)
...     return list_

>>> append(value='a')
['a']
>>> append(value='b')
['a', 'b']
```

Note that this is the case for `dict`, `list`, `set`, and several of the types in `collections`. Additionally, the classes you define yourself are mutable by default.

To work around this issue, you could consider changing the function to the following instead:

```
>>> def append(list_=None, value='value'):
...     if list_ is None:
...         list_ = []
...     list_.append(value)
...     return list_
```

```
>>> append(value='a')
['a']
>>> append(value='b')
['b']
```

Note that we had to use `if list_ is None` here. If we had done `if not list_` instead, it would have ignored the given `list_` if an empty `list` was passed.

Class properties

The problem of mutable variables also occurs when defining classes. It is very easy to mix class attributes and instance attributes. This can be confusing, especially when you are coming from other languages such as C#. Let's illustrate it:

```
>>> class SomeClass:
...     class_list = []
...
...     def __init__(self):
...         self.instance_list = []

>>> SomeClass.class_list.append('from class')
>>> instance = SomeClass()
>>> instance.class_list.append('from instance')
>>> instance.instance_list.append('from instance')

>>> SomeClass.class_list
['from class', 'from instance']
>>> SomeClass.instance_list
Traceback (most recent call last):
...
AttributeError: ... 'SomeClass' has no attribute 'instance_list'

>>> instance.class_list
['from class', 'from instance']
>>> instance.instance_list
['from instance']
```

As with the function arguments, the list and dictionaries are shared. So if you want a mutable property for a class that isn't shared between all instances, you will need to define it from within the `__init__` or any other instance method.

Another important thing to note when dealing with classes is that a class property will be inherited, and that's where things might prove to be confusing. When inheriting, the original properties will stay references (unless overwritten) to the original values, even in subclasses:

```
>>> class Parent:
...     pass

>>> class Child(Parent):
...     pass

>>> Parent.parent_property = 'parent'
>>> Child.parent_property
'parent'

>>> Child.parent_property = 'child'
>>> Parent.parent_property
'parent'
>>> Child.parent_property
'child'

>>> Child.child_property = 'child'
>>> Parent.child_property
Traceback (most recent call last):
...
AttributeError: ... 'Parent' has no attribute 'child_property'
```

While this is to be expected due to inheritance, someone else using the class might not expect the variable to change in the meantime. After all, we modified `Parent`, not `Child`.

There are two easy ways to prevent this. It is obviously possible to simply set the properties for every class separately. But the better solution is never to modify class properties outside of the class definition. It's easy to forget that the property will change in multiple locations, and if it has to be modifiable anyway, it's usually better to put it in an instance variable instead.

Overwriting and/or creating extra built-ins

While it can be useful in some cases, generally you will want to avoid overwriting global functions. The PEP 8 convention for naming your functions—similar to built-in statements, functions, and variables—is to use a trailing underscore.

So, do not use this:

```
list = [1, 2, 3]
```

Instead, use the following:

```
list_ = [1, 2, 3]
```

For lists and such, this is just a good convention. For statements such as `from`, `import`, and `with`, it's a requirement. Forgetting about this can lead to very confusing errors:

```
>>> list = list((1, 2, 3))
>>> list
[1, 2, 3]

>>> list((4, 5, 6))
Traceback (most recent call last):
    ...
TypeError: 'list' object is not callable

>>> import = 'Some import'
Traceback (most recent call last):
    ...
SyntaxError: invalid syntax
```

If you actually want to define a built-in that is available everywhere without requiring an import, that is possible. For debugging purposes, I've been known to add this code to a project while developing:

```
import builtins
import inspect
import pprint
import re

def pp(*args, **kwargs):
    '''PrettyPrint function that prints the variable name when
    available and pprints the data'''
    name = None
    # Fetch the current frame from the stack
    frame = inspect.currentframe().f_back
    # Prepare the frame info
    frame_info = inspect.getframeinfo(frame)

    # Walk through the lines of the function
    for line in frame_info[3]:
        # Search for the pp() function call with a fancy regexp
        m = re.search(r'\bpp\s*\(\s*([^)]*)\s*\)', line)
        if m:
            print('# %s:' % m.group(1), end=' ')
```

```
            break

    pprint.pprint(*args, **kwargs)

builtins.pf = pprint.pformat
builtins.pp = pp
```

This is much too hacky for production code, but it is still useful when working on a large project where you need print statements to debug. Alternative (and better) debugging solutions can be found in *Chapter 11*, *Debugging – Solving the Bugs*.

The usage is quite simple:

```
x = 10
pp(x)
```

Here is the output:

```
# x: 10
```

Modifying while iterating

At one point or another, you will run into this problem: while iterating through some mutable objects such as `dict` and `set`, you cannot modify them. All of these result in a `RuntimeError` telling you that you cannot modify the object during iteration:

```
>>> dict_ = dict(a=123)
>>> set_ = set((456,))

>>> for key in dict_:
...     del dict_[key]
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: dictionary changed size during iteration

>>> for item in set_:
...     set_.remove(item)
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: Set changed size during iteration
```

For a list, it does work, but can result in very strange results, so it should definitely be avoided as well:

```
>>> list_ = list(range(10))
>>> list_
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> for item in list_:
...     print(list_.pop(0), end=', ')
0, 1, 2, 3, 4,

>>> list_
[5, 6, 7, 8, 9]
```

While these issues can be avoided by copying the collections before usage, in many cases you are doing something wrong if you run into this issue. If manipulation is actually needed, building a new collection is often the easier way to go because the code will look more obvious. Whenever someone looks at code like this in the future, they might try to refactor it by removing the `list()` since it looks futile at first glance:

```
>>> list_ = list(range(10))

>>> for item in list(list_):
...     print(list_.pop(0), end=', ')
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
```

Catching and storing exceptions

When catching and storing exceptions in Python, you must keep in mind that for performance reasons, the stored exception is local to the `except` block. The result is that you need to explicitly store the exception in a *different* variable. Simply declaring the variable before the `try`/`except` block does not work and will make your variable disappear:

```
>>> exception = None

>>> try:
...     1 / 0
... except ZeroDivisionError as exception:
...     pass

>>> exception
Traceback (most recent call last):
    ...
NameError: name 'exception' is not defined
```

Storing the result in a new variable does work:

```
>>> try:
...     1 / 0
... except ZeroDivisionError as exception:
```

```
...     new_exception = exception

>>> new_exception
ZeroDivisionError('division by zero')
```

As you can probably see already, this code does have a bug now. If we don't end up in an exception, `new_exception` will not be defined yet. We will either need to add an `else` to the `try`/`except` or, better yet, pre-declare the variable before the `try`/`except`.

We really need to save it explicitly because Python 3 automatically deletes anything saved with `as variable` at the end of the `except` statements. The reason for this is that exceptions in Python 3 contain a `__traceback__` attribute. Having this attribute makes it much more difficult for the garbage collector to detect which memory should be freed as it introduces a recursive self-referencing cycle.

Specifically, this is `exception -> traceback -> exception -> traceback ...`.

This does mean that you should keep in mind that storing these exceptions can introduce memory leaks into your program.

The Python garbage collector is smart enough to understand that the variables are not visible anymore and will delete the variable eventually, but it can take a lot more time because it is a far more complicated garbage collection procedure. How the garbage collection actually works is covered in *Chapter 12, Performance – Tracking and Reducing Your Memory and CPU Usage*.

Late binding and closures

Closures are a method of implementing local scopes in code. They make it possible to locally define variables without overriding variables in the parent (or global) scope and hide the variables from the outside scope later. The problem with closures in Python is that Python tries to bind its variables as late as possible for performance reasons. While generally useful, it does have some unexpected side effects:

```
>>> functions = [lambda: i for i in range(3)]

>>> for function in functions:
...     print(function(), end=', ')
2, 2, 2,
```

You were probably expecting `0, 1, 2` instead. Due to late binding, however, all functions get the last value of `i` instead, which is `2`.

What should we do instead? As with the cases in earlier paragraphs, the variable needs to be made local. One option is to force immediate binding by currying the function with `partial`:

```
>>> from functools import partial

>>> functions = [partial(lambda x: x, i) for i in range(3)]
```

```
>>> for function in functions:
...     print(function(), end=', ')
0, 1, 2,
```

A better solution would be to avoid binding problems altogether by not introducing extra scopes (the `lambda`) that use external variables. If `i` is specified as an argument to `lambda`, this will not be a problem.

Circular imports

Even though Python is fairly tolerant of circular imports, there are some cases where you will get errors.

Let's assume we have two files:

`T_28_circular_imports_a.py`:

```
import T_28_circular_imports_b

class FileA:
    pass

class FileC(T_28_circular_imports_b.FileB):
    pass
```

`T_28_circular_imports_b.py`:

```
import T_28_circular_imports_a

class FileB(T_28_circular_imports_a.FileA):
    pass
```

Running either of these files results in a circular import error:

```
Traceback (most recent call last):
  File "T_28_circular_imports_a.py", line 1, in <module>
    import T_28_circular_imports_b
  File "T_28_circular_imports_b.py", line 1, in <module>
    import T_28_circular_imports_a
  File "T_28_circular_imports_a.py", line 8, in <module>
    class FileC(T_28_circular_imports_b.FileB):
AttributeError: partially initialized module 'T_28_circular_imports_b' has no
attribute 'FileB' (most likely due to a circular import)
```

There are several ways to work around this problem. The simplest solution is to move the `import` statement so the circular import doesn't occur anymore. In this case, the `import` in import `T_28_circular_imports_a.py` needs to be moved between `FileA` and `FileB`.

In most cases, the better solution is to restructure the code, however. Move the common base class to a separate file so there is no need for a circular import anymore. For the example above, that would look something like this:

`T_29_circular_imports_a.py`:

```
class FileA:
    pass
```

`T_29_circular_imports_b.py`:

```
import T_29_circular_imports_a

class FileB(T_29_circular_imports_a.FileA):
    pass
```

`T_29_circular_imports_c.py`:

```
import T_29_circular_imports_b

class FileC(T_29_circular_imports_b.FileB):
    pass
```

If that is also not possible, it can be useful to import from a function at runtime instead of at import time. Naturally this is not an easy option for class inheritance, but if you only need the import at runtime, you can defer the importing.

Lastly, there is the option of dynamic imports, such as what the Django framework uses for the `ForeignKey` fields. In addition to actual classes, the `ForeignKey` fields also support strings, which will be imported automatically when needed.

While this is a very effective way of working around the problem, it does mean that your editor, linting tools, and other tools won't understand the object you are dealing with. To those tools, it will look like a string, so unless specific hacks are added to those, they will not assume the value to be anything besides a string.

In addition, because the `import` only happens at runtime, you will not notice import problems until you execute the function. That means that errors that normally would have presented themselves as soon as you run the script or application will now only show up when the function is called. This is a great recipe for hard-to-trace bugs that won't occur for you but will for other users of the code.

The pattern is still useful for cases such as plugin systems, however, as long as care is taken to avoid the caveats mentioned. Here's a simple example to import dynamically:

```
>>> import importlib

>>> module_name = 'sys'
>>> attribute = 'version_info'
```

```
>>> module = importlib.import_module(module_name)
>>> module
<module 'sys' (built-in)>
>>> getattr(module, attribute).major
3
```

Using `importlib`, it is fairly easy to dynamically `import` a module and by using `getattr`, you can get a specific object from the module.

Import collisions

One problem that can be extremely confusing is having colliding imports—multiple packages/modules with the same name. I have had more than a few bug reports on my packages about cases like these.

My `numpy-stl` project, for example, houses the code in a package named `stl`. Many people create a test file named `stl.py`. When importing `stl` from `stl.py`, it will import itself instead of the `stl` package.

In addition to this, there is also the problem of packages being incompatible with each other. Common names might be used by several packages, so be careful when installing a bunch of similar packages since they might be sharing the same name. When in doubt, just create a new virtual environment and try again. Doing this can save you a lot of debugging.

Summary

This chapter showed you what the Pythonic philosophy is all about and some of the reasoning behind it. Additionally, you have learned about the Zen of Python and what is considered beautiful and ugly within the Python community. While code style is highly personal, Python has a few very helpful guidelines that at least keep people mostly on the same page and style.

In the end, we are all consenting adults; everyone has the right to write code as they sees fit. But I do request that you please read through the style guides and try to adhere to them unless you have a really good reason not to.

With all that power comes great responsibility, and a few pitfalls, though there aren't too many. Some are tricky enough to fool me regularly and I've been writing Python for a long time! Python improves all the time though. Many pitfalls have been taken care of since Python 2, but some will always remain. For example, circular imports and definitions can easily bite you in most languages that support them, but that doesn't mean we'll stop trying to improve Python.

A good example of the improvements in Python over the years is the `collections` module. It contains many useful collections that have been added by users because there was a need. Most of them are actually implemented in pure Python, and because of that, they are easy enough to be read by anyone. Understanding them might take a bit more effort, but I truly believe that if you make it to the end of this book, you will have no problem understanding what the collections do. Fully understanding how the internals work is something I cannot promise, though; some parts of that speak more to generic computer science than Python mastery.

The next chapter will show you some of the collections available in Python and how they are constructed internally. Even though you are undoubtedly familiar with collections such as lists and dictionaries, you might not be aware of the performance characteristics involved with some of the operations. If some of the examples in this chapter were less than clear, you don't have to worry. The next chapter will at least revisit some of them, and more will come in later chapters.

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:
`https://discord.gg/QMzJenHuJf`

4

Pythonic Design Patterns

The previous chapter covered a lot of guidelines for what to do and what to avoid in Python. Next, we will explore a few examples of how to work in a Pythonic way using the modules included with Python.

Design patterns are largely dependent on storing data; for this, Python comes bundled with several very useful collections. The most basic collections such as `list`, `tuple`, `set`, and `dict` will already be familiar to you, but Python also comes bundled with more advanced collections. Most of these simply combine the basic types for more powerful features. In this chapter, I will explain how to use these data types and collections in a Pythonic fashion.

Before we can properly discuss data structures and related performance, a basic understanding of time complexity (and specifically the big O notation) is required. The concept is really simple, but without it, I cannot easily explain the performance characteristics of operations and why seemingly nice-looking code can perform horribly.

In this chapter, once the big O notation is clear, we will discuss some data structures and I will show you some example design patterns, along with how to use them. We will start with the following basic data structures:

- `list`
- `dict`
- `set`
- `tuple`

Building on the basic data structures, we will continue with more advanced collections, such as the following:

- Dictionary-like types:
    - `ChainMap`
    - `Counter`
    - `Defaultdict`
    - `OrderedDict`

- List types: `heapq`
- Tuple types: `dataclass`
- Other types: `enum`

Time complexity – The big O notation

Before we can begin with this chapter, there is a simple notation that you need to understand. This chapter uses the big O notation to indicate the time complexity for an operation. Feel free to skip this section if you are already familiar with this notation. While the notation sounds really complicated, the concept is actually quite simple.

The big O letter refers to the capital version of the Greek letter Omicron, which means small-o (micron o).

When we say that a function takes `O(1)` time, it means that it generally only takes `1` step to execute. Similarly, a function with `O(n)` time would take `n` steps to execute, where `n` is generally the size (or length) of the object. This time complexity is just a basic indication of what to expect when executing the code, as it is generally what matters most.

In addition to O, several other characters might pop up in literature. Here's an overview of the characters used:

- `O` Big Omicron: The upper bound/worst-case scenario.
- `Ω` Big Omega: The lower bound/best-case scenario.
- `Θ` Big Theta: The tight bound, which means both O and Ω are identical.

A good example of an algorithm where these differ a lot is the quicksort algorithm. The quicksort algorithm is one of the most widely used sorting algorithm, which is surprising if you only look at time complexity according to the (big) O. The worst case for quicksort is `O(n**2)` and the best case is either `Ω(n log n)` or `Ω(n)`, depending on the implementation.

Given the worst case of `O(n**2)`, you might not expect the algorithm to be used a lot, but it has been the default sorting algorithm for many programming languages. Within C, it is still the default; for Java, it was the default up to Java 6; and Python used it up to 2002. So, why is/was quicksort so popular? For quicksort, it is very important to look at the average case, which is far more likely to occur than the worst case. Indeed, the average case is `O(n log n)`, which is really good for a sorting algorithm.

The purpose of the big O notation is to indicate the approximate performance of an operation based on the number of steps that need to be executed. A piece of code that executes a single step 1,000 times faster but needs to execute `O(2**n)` steps will still be slower than another version of it that takes only `O(n)` steps for a value of `n` equal to 10 or more.

This is because `2**n` for `n=10` is `2**10=1024`, which is 1,024 steps to execute the same code. This makes choosing the right algorithm very important, even when using languages such as `C/C++`, which are generally expected to perform better than Python with the CPython interpreter. If the code uses the wrong algorithm, it will still be slower for a non-trivial `n`.

For example, suppose you have a list of 1,000 items and you walk through them. This will take `O(n)` time because there are `n=1000` items. Checking to see whether an item exists in a list means silently walking through the items in a similar way, which means it also takes `O(n)`, so that's 1,000 steps.

If you do the same with a `dict` or `set` that has 1,000 keys/items, it will only take `O(1)` step because of how a `dict`/`set` is structured. How the `dict` and `set` are structured internally will be covered later in this chapter.

This means that if you want to check the existence of 100 items in that `list` or `dict`, it will take you `100*O(n)` for the `list` and `100*O(1)` for the `dict` or `set`. That is the difference between 100 steps and 100,000 steps, which means that the `dict`/`set` is `n` or 1,000 times faster in this case.

Even though the code seems very similar, the performance characteristics vary enormously:

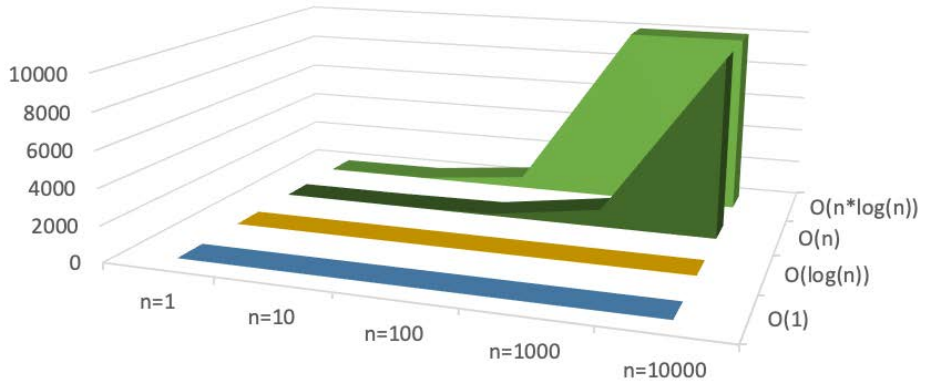
```
>>> n = 1000
>>> a = list(range(n))
>>> b = dict.fromkeys(range(n))

>>> for i in range(100):
...     assert i in a  # takes n=1000 steps
...     assert i in b  # takes 1 step
```

To illustrate `O(1)`, `O(n)`, and `O(n**2)` functions:

```
>>> def o_one(items):
...     return 1  # 1 operation so O(1)

>>> def o_n(items):
...     total = 0
...     # Walks through all items once so O(n)
...     for item in items:
...         total += item
...     return total

>>> def o_n_squared(items):
...     total = 0
...     # Walks through all items n*n times so O(n**2)
...     for a in items:
...         for b in items:
```

```
...             total += a * b
...     return total

>>> n = 10
>>> items = range(n)
>>> o_one(items)  # 1 operation
1
>>> o_n(items)  # n = 10 operations
45
>>> o_n_squared(items)  # n*n = 10*10 = 100 operations
2025
```

To illustrate this, we will look at some slower-growing functions first:

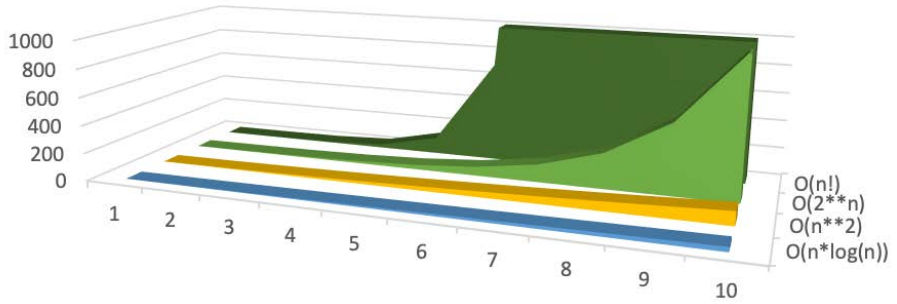


	n=1	n=10	n=100	n=1000	n=10000
O(1)	1	1	1	1	1
O(log(n))	0.0	3.3	6.6	10.0	13.3
O(n)	1	10	100	1000	10000
O(n*log(n))	0	33	664	9966	132877

Figure 4.1: Time complexity of slow-growing functions with n=1 to n=10,000

As you can see, the `O(log(n))` function scales really well with larger numbers; this is why a binary search is so incredibly fast, even for large datasets. Later in this chapter, you will see an example of a binary search algorithm.

The `O(n*log(n))` result shows a rather fast growth, which is undesirable, but better than some of the alternatives, as you can see in *Figure 4.2* with faster-growing functions:



	1	2	3	4	5	6	7	8	9	10
O(n*log(n))	0	2	5	8	12	16	20	24	29	33
O(n**2)	1	4	9	16	25	36	49	64	81	100
O(2**n)	2	4	8	16	32	64	128	256	512	1024
O(n!)	1	2	6	24	120	720	5040	40320	362880	3628800

Figure 4.2: Time complexity of fast-growing functions with n=1 to n=10

Looking at these charts, the `O(n*log(n))` looks quite good by comparison. As you will see later in this chapter, many sorting algorithms use `O(n*log(n))` functions and some use `O(n**2)`.

These algorithms quickly grow to an incalculable size; the `O(2**n)` function, for example, already takes 1,024 steps with 10 items and doubles with every step. A famous example of this is the current solution to the Towers of Hanoi problem, where `n` is the number of disks.

The `O(n!)` factorial function is far worse and becomes impossibly large after just a few steps. One of the most famous examples of this is the Traveling Salesman problem: finding the shortest route covering a list of cities exactly once.

Next, we'll dive into core collections.

Core collections

Before we can look at the more advanced combined collections later in this chapter, you need to understand the workings of the core Python collections. This is not just about their usage; it is also about the time complexities involved, which can strongly affect how your application will behave as it grows. If you are well versed in the time complexities of these objects and know the possibilities of Python 3's tuple packing and unpacking by heart, then feel free to jump to the *Advanced collections* section.

list – A mutable list of items

The `list` is most likely the container structure that you've used most in Python. It is simple in terms of its usage, and for most cases, it exhibits great performance.

While you may already be very familiar with the usage of `list`, you might not be aware of the time complexities of the `list` object. Luckily, many of the time complexities of `list` are very low; `append`, get operations, set operations, and `len` all take `O(1)` time—the best possible. However, you may not know that `remove` and `insert` have `O(n)` worst-case time complexity. So, to delete a single item out of 1,000 items, Python might have to walk through 1,000 items. Internally, the `remove` and `insert` operations execute something along these lines:

```
>>> def remove(items, value):
...     new_items = []
...     found = False
...     for item in items:
...         # Skip the first item which is equal to value
...         if not found and item == value:
...             found = True
...             continue
...         new_items.append(item)
...
...     if not found:
...         raise ValueError('list.remove(x): x not in list')
...
...     return new_items

>>> def insert(items, index, value):
...     new_items = []
...     for i, item in enumerate(items):
...         if i == index:
...             new_items.append(value)
...         new_items.append(item)
...     return new_items

>>> items = list(range(10))
>>> items
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> items = remove(items, 5)
>>> items
[0, 1, 2, 3, 4, 6, 7, 8, 9]

>>> items = insert(items, 2, 5)
>>> items
[0, 1, 5, 2, 3, 4, 6, 7, 8, 9]
```

To remove or insert a single item from/into the list, Python needs to shift the rest of the list after the insertion/deletion point. For a large `list`, this can become a performance burden and, if possible, should be avoided by using `append` instead of `insert`. When executing this only once, it is, of course, not all that bad. But when executing a large number of `remove` operations, a `filter` or `list` comprehension is a much faster solution because, if properly structured, it needs to copy the list only once.

For example, suppose we wish to remove a specific set of numbers from the list. We have quite a few options for this. The first is a solution using `remove`, which becomes slower if the number of items to remove becomes larger.

Next up is constructing a new list, a list comprehension, or a `filter` statement. *Chapter 5*, *Functional Programming – Readability Versus Brevity*, will explain `list` comprehensions and the `filter` statement in more detail. But first, let's check out some examples:

```
>>> primes = set((1, 2, 3, 5, 7))

# Classic solution
>>> items = list(range(10))
>>> for prime in primes:
...     items.remove(prime)
>>> items
[0, 4, 6, 8, 9]

# List comprehension
>>> items = list(range(10))
>>> [item for item in items if item not in primes]
[0, 4, 6, 8, 9]

# Filter
>>> items = list(range(10))
>>> list(filter(lambda item: item not in primes, items))
[0, 4, 6, 8, 9]
```

The latter two examples are much faster for large lists of items. This is because the operations are much faster. To compare using `n=len(items)` and `m=len(primes)`, the first example takes `O(m*n)=5*10=50` operations, whereas the latter two take `O(n*1)=10*1=10` operations.

The first method is actually slightly better than stated because `n` decreases during the loop. So, it's effectively `10+9+8+7+6=40`, but this is an effect that is negligible enough to ignore. In the case of `n=1000`, that would be the difference between `1000+999+998+997+996=4990` and `5*1000=5000`, which makes no real-world difference.

Of course, `min`, `max`, and `in` all take `O(n)` as well, but that is expected for a structure that is not optimized for these types of lookups.

They can be implemented like this:

```
>>> def in_(items, value):
...     for item in items:
...         if item == value:
...             return True
...     return False

>>> def min_(items):
...     current_min = items[0]
...     for item in items[1:]:
...         if current_min > item:
...             current_min = item
...     return current_min

>>> def max_(items):
...     current_max = items[0]
...     for item in items[1:]:
...         if current_max < item:
...             current_max = item
...     return current_max

>>> items = range(5)
>>> in_(items, 3)
True
>>> min_(items)
0
>>> max_(items)
4
```

With these examples, it's also clear that the `in` operator is a good example of where the best, worst, and average cases are vastly different. The best case is `O(1)`, which is being lucky and finding our value at the first item. The worst case is `O(n)` because it might not exist or it could be the last item. From this, you might expect the average case to be `O(n/2)`, but you would be wrong. The average case is still `O(n)` since there is a large likelihood of the item not existing in the list at all.

dict — A map of items

The `dict` is probably the container structure you will choose to use the most. You might not realize that you are using it constantly without explicitly using `dict`. Every function call and variable access goes through a `dict` to look up the name from the `local()` or `global()` scope dictionaries.

The `dict` is fast, simple to use, and very effective for a wide range of use cases. The average time complexity is `O(1)` for the `get`, `set`, and `delete` operations.

There are exceptions to this time complexity that you need to be aware of, however. The way a `dict` works is by converting the key into a hash using the `hash` function (which calls the `__hash__` method of the object given as a key) and storing it in a hash table.

Magic methods such as `__hash__` are called either **magic methods** or **dunder methods**, where **dunder** is short for double-underscore.

There are two problems with hash tables, however. The first and the most obvious is that the items will be sorted by hash, which appears at random in most cases. The second problem with hash tables is that they can have hash collisions, and the result of a hash collision is that in the worst case, all the former operations can take `O(n)` instead. Hash collisions are not all that likely to occur, but they can occur, and if a large `dict` performs below par, that is the place to look.

Since Python 3.6, the default `dict` implementation in CPython has changed to a version that is sorted by insertion. Since Python 3.7, this is guaranteed behavior since other Python versions such as Jython and PyPy could use different implementations before version 3.7.

Let's see how this actually works in practice. For the sake of this example, I will use one of the simplest hashing algorithms I can think of, which uses the most significant digit of a number. So, for the case of `12345`, this hashing function will return `1`, and for `56789`, it will return `5`:

```
>>> def most_significant(value):
...     while value >= 10:
...         value //= 10
...     return value

>>> most_significant(12345)
1
>>> most_significant(99)
9
>>> most_significant(0)
0
```

Now, we will emulate a `dict` using a `list` of lists with this hashing method. We know that our hashing method can only return numbers from `0` to `9`, so we need only 10 buckets in our list. Now, we will add a few values and see how a `contains` function could work:

```
>>> def add(collection, key, value):
...     index = most_significant(key)
...     collection[index].append((key, value))
```

```
>>> def contains(collection, key):
...     index = most_significant(key)
...     for k, v in collection[index]:
...         if k == key:
...             return True
...     return False

# Create the collection of 10 lists
>>> collection = [[], [], [], [], [], [], [], [], [], []]

# Add some items, using key/value pairs
>>> add(collection, 123, 'a')
>>> add(collection, 456, 'b')
>>> add(collection, 789, 'c')
>>> add(collection, 101, 'c')

# Look at the collection
>>> collection
[[], [(123, 'a'), (101, 'c')], [], [],
 [(456, 'b')], [], [], [(789, 'c')], [], []]

# Check if the contains works correctly
>>> contains(collection, 123)
True
>>> contains(collection, 1)
False
```

This code is obviously not identical to the `dict` implementation, but it is similar. Since we can just get item `1` for a value of `123` by simple indexing, we have only `O(1)` lookup costs in the general case. However, since both keys, `123` and `101`, are within the `1` bucket, the runtime can actually increase to `O(n)` in the worst case, where all keys have the same hash. As mentioned, that is a hash collision. To alleviate hash collisions beyond what the `hash()` function already does, the Python `dict` uses a probing sequence to automatically shift hashes if needed. The details of this method are well explained in the `dictobject.c` file of the Python source.

To debug hash collisions, you can use the `hash()` function paired with `collections.Counter`. This will quickly show you where hash collisions occur but it does not take the `dict` probing sequence into consideration.

In addition to the hash collision performance problem, there is another behavior that might surprise you. When deleting items from a dictionary, it won't actually resize the dictionary in memory. The result is that both copying and iterating over the entire dictionary take `O(m)` time (where `m` is the maximum size of the dictionary); `n`, the current number of items, is not used. So, if you add 1,000 items to a `dict` and remove 999, iterating and copying will still take 1,000 steps. The only way to work around this issue is by recreating the dictionary, which is something that both the `copy` and `insert` operations do. Note that recreation during an `insert` operation is not guaranteed and depends on the number of free slots available internally.

set — Like a dict without values

A `set` is a structure that uses the `hash()` function to get a unique collection of values. Internally, it is very similar to a `dict`, with the same hash collision problem, but there are a few handy features of `set` that need to be shown:

```
# All output in the table below is generated using this function
>>> def print_set(expression, set_):
...     'Print set as a string sorted by letters'
...     print(expression, ''.join(sorted(set_)))

>>> spam = set('spam')
>>> print_set('spam:', spam)
spam: amps

>>> eggs = set('eggs')
>>> print_set('eggs:', eggs)
eggs: egs
```

The first few are pretty much as expected. When we get to the operators, it gets interesting:

Expression	Output	Explanation
`spam`	`amps`	All unique items. A `set` doesn't allow for duplicates.
`eggs`	`egs`	
`spam & eggs`	`s`	Every item in both.
`spam | eggs`	`aegmps`	Every item in either or both.
`spam ^ eggs`	`aegmp`	Every item in either but not in both.
`spam - eggs`	`amp`	Every item in the first but not the latter.
`eggs - spam`	`eg`	
`spam > eggs`	`False`	True if every item in the latter is in the first.
`eggs > spam`	`False`	
`spam > sp`	`True`	
`spam < sp`	`False`	True if every item in the first is contained in the latter.

One useful example of `set` operations is calculating the differences between two objects. For example, let's assume we have two lists:

- `current_users`: The current users in a group
- `new_users`: The new list of users in a group

In permission systems, this is a very common scenario—mass adding and/or removing users from a group. Within many permission databases, it's not easily possible to set the entire list at once, so you need a list to insert and a list to delete. This is where `set` comes in really handy:

```
# The set function takes a sequence as argument so the double ( is required.
>>> current_users = set((
...     'a',
...     'b',
...     'd',
... ))

>>> new_users = set((
...     'b',
...     'c',
...     'd',
...     'e',
... ))

>>> to_insert = new_users - current_users
>>> sorted(to_insert)
['c', 'e']
>>> to_delete = current_users - new_users
>>> sorted(to_delete)
['a']
>>> unchanged = new_users & current_users
>>> sorted(unchanged)
['b', 'd']
```

Now, we have lists of all users who were added, removed, and unchanged. Note that `sorted` is only needed for consistent output, since a `set` has no predefined sort order.

tuple — The immutable list

A `tuple` is another object that you probably use very often without even noticing it. When you look at it initially, it seems like a useless data structure. It's like a list that you can't modify, so why not just use a `list`? In fact, there are a few cases where a `tuple` offers some really useful functionalities that a `list` does not.

Firstly, they are hashable. This means that you can use a `tuple` as a key in a `dict` or as an item of a `set`, which is something a `list` can't do:

```
>>> spam = 1, 2, 3
>>> eggs = 4, 5, 6

>>> data = dict()
>>> data[spam] = 'spam'
>>> data[eggs] = 'eggs'

>>> import pprint  # Using pprint for consistent and sorted output

>>> pprint.pprint(data)
{(1, 2, 3): 'spam', (4, 5, 6): 'eggs'}
```

However, tuples can contain more than simple numbers. You can use nested tuples, strings, numbers, and anything else for which the `hash()` function returns a consistent result:

```
>>> spam = 1, 'abc', (2, 3, (4, 5)), 'def'
>>> eggs = 4, (spam, 5), 6

>>> data = dict()
>>> data[spam] = 'spam'
>>> data[eggs] = 'eggs'

>>> import pprint  # Using pprint for consistent and sorted output

>>> pprint.pprint(data)
{(1, 'abc', (2, 3, (4, 5)), 'def'): 'spam',
 (4, ((1, 'abc', (2, 3, (4, 5)), 'def'), 5), 6): 'eggs'}
```

You can make these as complex as you need. As long as all the parts of the tuple are hashable, you will have no problem hashing the tuple as well. You can still construct a tuple containing a `list` or any other unhashable type without a problem, but that will make the tuple unhashable.

Perhaps even more useful is the fact that tuples also support tuple packing and unpacking:

```
# Assign using tuples on both sides
>>> a, b, c = 1, 2, 3
>>> a
1

# Assign a tuple to a single variable
>>> spam = a, (b, c)
```

```
>>> spam
(1, (2, 3))

# Unpack a tuple to two variables
>>> a, b = spam
>>> a
1
>>> b
(2, 3)
```

In addition to regular packing and unpacking, from Python 3 onward, we can actually pack and unpack objects with a variable number of items:

```
# Unpack with variable length objects which assigns a list instead
# of a tuple
>>> spam, *eggs = 1, 2, 3, 4
>>> spam
1
>>> eggs
[2, 3, 4]

# Which can be unpacked as well of, course
>>> a, b, c = eggs
>>> c
4

# This works for ranges as well
>>> spam, *eggs = range(10)
>>> spam
0
>>> eggs
[1, 2, 3, 4, 5, 6, 7, 8, 9]

# And it works both ways
>>> a, b, *c = a, *eggs
>>> a, b
(2, 1)
>>> c
[2, 3, 4, 5, 6, 7, 8, 9]
```

Packing and unpacking can be applied to function arguments:

```
>>> def eggs(*args):
...     print('args:', args)

>>> eggs(1, 2, 3)
args: (1, 2, 3)
```

They are equally useful when returning from a function:

```
>>> def spam_eggs():
...     return 'spam', 'eggs'

>>> spam, eggs = spam_eggs()
>>> spam
'spam'
>>> eggs
'eggs'
```

Now that you have seen the core Python collections and their limitations, you should understand a bit better when certain collections are a good (or bad) idea. And more importantly, if a data structure doesn't perform as you expect it to, you will understand why.

Unfortunately, often real-world problems are not as simple as the ones you have seen in this chapter, so you will have to weigh up the pros and the cons of the data structures and choose the best solution for your case. Alternatively, you can build a more advanced data structure by combining a few of these structures. Before you start building your own structures, however, keep reading because we will now dive into more advanced collections that do just that: combine the core collections.

Pythonic patterns using advanced collections

The following collections are mostly just extensions of base collections; some of them are fairly simple, while others are a bit more advanced. For all of them, though, it is important to know the characteristics of the underlying structures. Without understanding them, it will be difficult to comprehend the characteristics of the collections.

There are a few collections that are implemented in native C code for performance reasons, but all of them can easily be implemented in pure Python as well. The following examples will show you not only the features and characteristics of these collections, but also a few example design patterns where they can be useful. Naturally, this is not an exhaustive list, but it should give you an idea of the possibilities.

Smart data storage with type hinting using dataclasses

One of the most useful recent additions to Python (since 3.5) is type hinting. With the type annotations, you can give type hints to your editor, documentation generator, and others reading your code.

Within Python, we are generally expected to be "consenting adults," which means the hints are not enforced in any way. This is similar to how private and protected variables in Python are not enforced. This means that we can easily give a completely different type from what our hint would suggest:

```
>>> spam: int
>>> __annotations__['spam']
<class 'int'>

>>> spam = 'not a number'
>>> __annotations__['spam']
<class 'int'>
```

Even with the `int` type hint, we can still insert a `str` if we want to.

The `dataclasses` module, which was introduced in Python 3.7 (backports available for Python 3.6), uses the type hinting system to automatically generate classes, including documentation and constructors based on these types:

```
>>> import dataclasses

>>> @dataclasses.dataclass
... class Sandwich:
...     spam: int
...     eggs: int = 3

>>> Sandwich(1, 2)
Sandwich(spam=1, eggs=2)

>>> sandwich = Sandwich(4)
>>> sandwich
Sandwich(spam=4, eggs=3)
>>> sandwich.eggs
3
>>> dataclasses.asdict(sandwich)
{'spam': 4, 'eggs': 3}
>>> dataclasses.astuple(sandwich)
(4, 3)
```

The basic class looks quite simple and like it's nothing special, but if you look carefully, the `dataclass` has generated multiple methods for us. Which ones are generated becomes obvious when looking at the `dataclass` arguments:

```
>>> help(dataclasses.dataclass)
Help on ... dataclass(..., *, init=True, repr=True, eq=True,
order=False, unsafe_hash=False, frozen=False) ...
```

As you can see, `dataclass` has several Boolean flags that decide what to generate.

First, the `init` flag tells `dataclass` to create an `__init__` method that looks something like this:

```
>>> def __init__(self, spam, eggs=3):
...     self.spam = spam
...     self.eggs = eggs
```

Further, `dataclass` has flags for:

- `repr`: This generates a `__repr__` magic function that generates a nice and readable output like `Sandwich(spam=1, eggs=2)` instead of something like `<__main__.Sandwich object at 0x...>`.
- `eq`: This generates an automatic comparison method that compares two instances of `Sandwich` by their value when doing `if sandwich_a == sandwich_b`.
- `order`: This generates a whole range of methods so that comparison operators such as `>=` and `<` work by comparing the output of `dataclasses.astuple`.
- `unsafe_hash`: This will force the generation of a `__hash__` method so that you use the `hash()` function on it. By default, a `__hash__` function is only generated when all parts of the object are considered immutable. The reason for this is that `hash()` should *always* be consistent. If you wish to store an object in a `set`, it needs to have a consistent hash. Since a `set` uses `hash()` to decide which memory address to use, if the object changes, the `set` would need to move the object as well.
- `frozen`: This will prevent changes after the instance has been created. The main use for this is to make sure the `hash()` of the object remains consistent.
- `slots`: This automatically adds a `__slots__` attribute which makes attribute access and storage faster and more efficient. More about slots in *Chapter 12, Performance – Tracking and Reducing Your Memory and CPU Usage*.

The only flag that adds validation is the `frozen` flag, which makes everything read-only and prevents us from changing the `__setattr__` and `__getattr__` methods, which could be used to modify the instance otherwise.

The type hinting system still only provides hints; however, these hints are not enforced in any way. In *Chapter 6, Decorators – Enabling Code Reuse by Decorating*, you will see how we can add these types of enforcements to our code using custom decorators.

For a more useful example that includes dependence, let's say that we have some users who all belong to one or multiple groups in a system:

```
>>> import typing

>>> @dataclasses.dataclass
... class Group:
...     name: str
```

```
...     parent: 'Group' = None

>>> @dataclasses.dataclass
... class User:
...     username: str
...     email: str = None
...     groups: typing.List[Group] = None

>>> users = Group('users')
>>> admins = Group('admins', users)
>>> rick = User('rick', groups=[admins])
>>> gvr = User('gvanrossum', 'guido@python.org', [admins])

>>> rick.groups
[Group(name='admins', parent=Group(name='users', parent=None))]

>>> rick.groups[0].parent
Group(name='users', parent=None)
```

In addition to linking dataclasses to each other, this also shows how to create a collection as a field and how to have recursive definitions. As you can see, the `Group` class references its own definition as a parent.

These dataclasses are especially useful when used for reading data from databases or CSV files. You can easily extend the behavior of dataclasses to include custom methods, which makes them a very useful basis for storing your custom data models.

Combining multiple scopes with ChainMap

Introduced in Python 3.3, `ChainMap` allows you to combine multiple mappings (dictionaries, for example) into one. This is especially useful when combining multiple contexts. For example, when looking for a variable in your current scope, by default, Python will search in `locals()`, `globals()`, and, lastly, `builtins`.

To explicitly write code to do this, we could do something like this:

```
>>> import builtins

>>> builtin_vars = vars(builtins)

>>> key = 'something to search for'

>>> if key in locals():
...     value = locals()[key]
... elif key in globals():
...     value = globals()[key]
... elif key in builtin_vars:
```

```
...     value = builtin_vars[key]
... else:
...     raise NameError(f'name {key!r} is not defined')
Traceback (most recent call last):
...
NameError: name 'something to search for' is not defined
```

This works, but it's ugly to say the least. We can make it prettier by removing some of the repeated code:

```
>>> mappings = locals(), globals(), vars(builtins)

>>> for mapping in mappings:
...     if key in mapping:
...         value = mapping[key]
...         break
... else:
...     raise NameError(f'name {key!r} is not defined')
Traceback (most recent call last):
...
NameError: name 'something to search for' is not defined
```

That's a lot better! Moreover, this can actually be considered a nice solution. But since Python 3.3, it's even easier. Now, we can simply use the following code:

```
>>> import collections

>>> mappings = collections.ChainMap(
...     locals(), globals(), vars(builtins))
>>> mappings[key]
Traceback (most recent call last):
...
KeyError: 'something to search for'
```

As you can see, the `ChainMap` class is automatically coalescing the requested value through every given `dict` until it finds a match. And if the value is not available, a `KeyError` is raised since it behaves like a `dict`.

This is very useful for reading configurations from multiple sources and simply getting the first matching item. For a command-line application, this could start with the command-line arguments, followed by the local configuration file, followed by the global configuration file, and lastly the defaults. To illustrate a bit of code similar to what I use in small command-line scripts:

```
>>> import json
>>> import pathlib
>>> import argparse
```

```
>>> import collections

>>> DEFAULT = dict(verbosity=1)

>>> config_file = pathlib.Path('config.json')
>>> if config_file.exists():
...     config = json.load(config_file.open())
... else:
...     config = dict()

>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('-v', '--verbose', action='count',
...                     dest='verbosity')
_CountAction(...)

>>> args, _ = parser.parse_known_args()
>>> defined_args = {k: v for k, v in vars(args).items() if v}
>>> combined = collections.ChainMap(defined_args, config, DEFAULT)
>>> combined['verbosity']
1

>>> args, _ = parser.parse_known_args(['-vv'])
>>> defined_args = {k: v for k, v in vars(args).items() if v}
>>> combined = collections.ChainMap(defined_args, config, DEFAULT)
>>> combined['verbosity']
2
```

The inheritance can clearly be seen here. When a specific command-line argument is given (`-vv`), that result is used. Otherwise, the code falls back to the one in `DEFAULTS` or any other available variable.

Default dictionary values using defaultdict

The `defaultdict` is one of my favorite objects in the `collections` package. Before it was added to the core, I wrote similar objects several times. While it is a fairly simple object, it is extremely useful for all sorts of design patterns. Instead of having to check for the existence of a key and adding a value every time, you can just declare the default from the beginning, and there is no need to worry about the rest.

For example, let's say we are building a very basic graph structure from a list of connected nodes.

This is our list of connected nodes (one way):

```
nodes = [
    ('a', 'b'),
    ('a', 'c'),
    ('b', 'a'),
```

```
    ('b', 'd'),
    ('c', 'a'),
    ('d', 'a'),
    ('d', 'b'),
    ('d', 'c'),
]
```

Now, let's put this graph into a normal dictionary:

```
>>> graph = dict()
>>> for from_, to in nodes:
...     if from_ not in graph:
...         graph[from_] = []
...     graph[from_].append(to)

>>> import pprint

>>> pprint.pprint(graph)
{'a': ['b', 'c'],
 'b': ['a', 'd'],
 'c': ['a'],
 'd': ['a', 'b', 'c']}
```

Some variations are possible, of course, such as using `setdefault`. However, they remain more complex than they need to be.

The truly Pythonic version uses `defaultdict` instead:

```
>>> import collections

>>> graph = collections.defaultdict(list)
>>> for from_, to in nodes:
...     graph[from_].append(to)

>>> import pprint

>>> pprint.pprint(graph)
defaultdict(<class 'list'>,
            {'a': ['b', 'c'],
             'b': ['a', 'd'],
             'c': ['a'],
             'd': ['a', 'b', 'c']})
```

Isn't that a beautiful bit of code? The `defaultdict` can also be used as a basic version of the `Counter` object. It's not as fancy and doesn't have all the bells and whistles that `Counter` has, but it does the job in many cases:

```
>>> counter = collections.defaultdict(int)
>>> counter['spam'] += 5
>>> counter
defaultdict(<class 'int'>, {'spam': 5})
```

The default value for `defaultdict` needs to be a callable object. In the previous cases, these were `int` and `list`, but you can easily define your own functions to use as a default value. That's what the following example uses, although I don't recommend production usage since it lacks a bit of readability. I do believe, however, that it is a beautiful example of the power of Python.

This is how we create a `tree` in a single line of Python:

```
import collections

def tree(): return collections.defaultdict(tree)
```

Brilliant, isn't it? Here's how we can actually use it:

```
>>> import json
>>> import collections

>>> def tree():
...     return collections.defaultdict(tree)

>>> colours = tree()
>>> colours['other']['black'] = 0x000000
>>> colours['other']['white'] = 0xFFFFFF
>>> colours['primary']['red'] = 0xFF0000
>>> colours['primary']['green'] = 0x00FF00
>>> colours['primary']['blue'] = 0x0000FF
>>> colours['secondary']['yellow'] = 0xFFFF00
>>> colours['secondary']['aqua'] = 0x00FFFF
>>> colours['secondary']['fuchsia'] = 0xFF00FF

>>> print(json.dumps(colours, sort_keys=True, indent=4))
{
    "other": {
        "black": 0,
        "white": 16777215
    },
```

```
    "primary": {
        "blue": 255,
        "green": 65280,
        "red": 16711680
    },
    "secondary": {
        "aqua": 65535,
        "fuchsia": 16711935,
        "yellow": 16776960
    }
}
```

The nice thing is that you can make it go as deep as you like. Because of the `defaultdict` base, it generates itself recursively.

enum — A group of constants

The `enum` package introduced in Python 3.4 is quite similar in its workings to enums in many other programming languages, such as C and C++. It helps to create reusable constants for your module so you can avoid arbitrary constants. A basic example is as follows:

```
>>> import enum

>>> class Color(enum.Enum):
...     red = 1
...     green = 2
...     blue = 3

>>> Color.red
<Color.red: 1>
>>> Color['red']
<Color.red: 1>
>>> Color(1)
<Color.red: 1>
>>> Color.red.name
'red'
>>> Color.red.value
1
>>> isinstance(Color.red, Color)
True
>>> Color.red is Color['red']
True
>>> Color.red is Color(1)
True
```

A few of the handy features of the `enum` package are that the objects are iterable, accessible through both numeric and textual representation of the values, and, with proper inheritance, even comparable to other classes.

The following code shows the usage of a basic API:

```
>>> for color in Color:
...     color
<Color.red: 1>
<Color.green: 2>
<Color.blue: 3>

>>> colors = dict()
>>> colors[Color.green] = 0x00FF00
>>> colors
{<Color.green: 2>: 65280}
```

One of the lesser-known possibilities of the `enum` package is that you can make value comparisons work in addition to the identity comparisons you would normally use. And this works for every type—not just integers but (your own) custom types as well.

With a regular `enum`, only an identity check (that is, `a is b`) works:

```
>>> import enum

>>> class Spam(enum.Enum):
...     EGGS = 'eggs'

>>> Spam.EGGS == 'eggs'
False
```

When we make the `enum` inherit `str` as well, it starts comparing the values in addition to the identity:

```
>>> import enum

>>> class Spam(str, enum.Enum):
...     EGGS = 'eggs'

>>> Spam.EGGS == 'eggs'
True
```

In addition to the preceding examples, the `enum` package has a few other variants such as `enum.Flag` and `enum.IntFlag`, which allow for bitwise operations. These can be useful for representing permissions as follows: `permissions = Perm.READ | Perm.Write`.

Whenever you have a list of constants that can be grouped together, consider using the `enum` package. It makes validation much cleaner than having to use `if/elif/elif/else` several times.

Sorting collections using heapq

The `heapq` module is a great little module that makes it very easy to create a priority queue in Python. It is a data structure that will always make the smallest (or largest, depending on the implementation) item available with minimum effort. The API is quite simple, and one of the best examples of its usage can be seen in the `OrderedDict` object. While you might not need it often, it is a very useful structure if you need it. And understanding the inner workings is important if you wish to understand the workings of classes such as `OrderedDict`.

If you are looking for a structure to keep your list always sorted, try the `bisect` module instead, which is covered in the next section.

The basic usage of `heapq` is simple but somewhat confusing initially:

```
>>> import heapq

>>> heap = [1, 3, 5, 7, 2, 4, 3]
>>> heapq.heapify(heap)
>>> heap
[1, 2, 3, 7, 3, 4, 5]

>>> while heap:
...     heapq.heappop(heap), heap
(1, [2, 3, 3, 7, 5, 4])
(2, [3, 3, 4, 7, 5])
(3, [3, 5, 4, 7])
(3, [4, 5, 7])
(4, [5, 7])
(5, [7])
(7, [])
```

One important thing to note here—something that you have probably already understood from the preceding example—is that the `heapq` module does not create a special object. It consists of a few methods to treat a regular list as a `heap`. That doesn't make it less useful, but it is something to take into consideration.

The really confusing part, at first glance, is the sort order. The array is actually sorted but not as a list; it is sorted as a tree. To illustrate this, take a look at the following tree, which shows how the tree is supposed to be read:

```
   1
 2   3
7 3 4 5
```

The smallest number is always at the top and the biggest numbers are always at the bottom row of the tree. Because of that, it's really easy to find the smallest number, but finding the largest is not as easy. To get the sorted version of the heap, we simply need to keep removing the top of the tree until all the items are gone. Therefore, the heapsort algorithm can be implemented as follows:

```
>>> def heapsort(iterable):
...     heap = []
...     for value in iterable:
...         heapq.heappush(heap, value)
...
...     while heap:
...         yield heapq.heappop(heap)

>>> list(heapsort([1, 3, 5, 2, 4, 1]))
[1, 1, 2, 3, 4, 5]
```

With `heapq` doing the heavy lifting, it becomes incredibly easy to write your own version of the `sorted()` function.

Since the `heappush` and `heappop` functions both have `O(log(n))` time complexity, they can be considered really fast. Combining those for the `n` items in the preceding iterable gives us `O(n*log(n))` for the `heapsort` function. The `heappush` method uses `list.append()` internally and swaps the items in the `list` to avoid the `O(n)` time complexity of `list.insert()`.

The `log(n)` refers to the base 2 logarithm function. To calculate this value, the `math.log2()` function can be used. This results in an increase of 1 every time the number doubles in size. For `n=2`, the value of `log(n)` is `1`, and consequently for `n=4` and `n=8`, the log values are 2 and 3, respectively. And `n=1024` results in a log of only `10`.

This means that a 32-bit number, which is `2**32 = 4294967296`, has a log of `32`.

Searching through sorted collections using bisect

The `heapq` module in the previous section gave us an easy way to sort a structure and keep it sorted. But what if we want to search through a sorted collection to see whether the item exists? Or what's the next biggest/smallest item if it doesn't? That's where the `bisect` algorithm helps us.

The `bisect` module inserts items in an object in such a way that they stay sorted and are easily searchable. If your primary purpose is searching, then `bisect` should be your choice. If you're modifying your collection a lot, `heapq` might be better for you.

As is the case with `heapq`, `bisect` does not really create a special data structure. The `bisect` module expects a `list` and expects that `list` to always be sorted. It is important to understand the performance implications of this. While appending items to a `list` has `O(1)` time complexity, inserting has `O(n)` time complexity, making it a very heavy operation. Effectively, creating a sorted list using bisect takes `O(n*n)`, which is quite slow, especially because creating the same sorted list using `heapq` or `sorted()` takes `O(n*log(n))` instead.

If you have a sorted structure and you only need to add a single item, then the `bisect` algorithm can be used for insertion. Otherwise, it's generally faster to simply append the items and call `list.sort()` or `sorted()` afterward.

To illustrate, we have these lines:

```
>>> import bisect

# Using the regular sort:
>>> sorted_list = []
>>> sorted_list.append(5)  # O(1)
>>> sorted_list.append(3)  # O(1)
>>> sorted_list.append(1)  # O(1)
>>> sorted_list.append(2)  # O(1)
>>> sorted_list.sort()  # O(n * log(n)) = 4 * log(4) = 8
>>> sorted_list
[1, 2, 3, 5]

# Using bisect:
>>> sorted_list = []
>>> bisect.insort(sorted_list, 5)  # O(n) = 1
>>> bisect.insort(sorted_list, 3)  # O(n) = 2
>>> bisect.insort(sorted_list, 1)  # O(n) = 3
>>> bisect.insort(sorted_list, 2)  # O(n) = 4
>>> sorted_list
[1, 2, 3, 5]
```

For a small number of items, the difference is negligible, but the number of operations needed to sort using `bisect` quickly grows to a point where the difference will be large. For `n=4`, the difference is just between `4 * 1 + 8 = 12` and `1 + 2 + 3 + 4 = 10`, making the `bisect` solution faster. But if we were to insert 1,000 items, it would be `1000 + 1000 * log(1000) = 10966` versus `1 + 2 + … 1000 = 1000 * (1000 + 1) / 2 = 500500`. So, be very careful while inserting many items.

Searching within the list is very fast, though; because it is sorted, we can use a very simple binary search algorithm. For example, what if we want to check whether a few numbers exist within the list? The simplest algorithm, shown as follows, simply loops through the list and checks all items, resulting in `O(n)` worst-case performance:

```
>>> sorted_list = [1, 2, 5]

>>> def contains(sorted_list, value):
...     for item in sorted_list:
...         if item > value:
...             break
...         elif item == value:
```

```
...             return True
...     return False

>>> contains(sorted_list, 2)  # Need to walk through 2 items, O(n) = 2
True
>>> contains(sorted_list, 4)  # Need to walk through 3 items, O(n) = 3
False
>>> contains(sorted_list, 6)  # Need to walk through 3 items, O(n) = 3
False
```

With the `bisect` algorithm, though, there is no need to walk through the entire list:

```
>>> import bisect

>>> sorted_list = [1, 2, 5]
>>> def contains(sorted_list, value):
...     i = bisect.bisect_left(sorted_list, value)
...     return i < len(sorted_list) and sorted_list[i] == value

>>> contains(sorted_list, 2)  # Found it after the first step, O(log(n)) = 1
True
>>> contains(sorted_list, 4)  # No result after 2 steps, O(log(n)) = 2
False
>>> contains(sorted_list, 6)  # No result after 2 steps, O(log(n)) = 2
False
```

The `bisect_left` function tries to find the position at which the number is supposed to be. This is actually what `bisect.insort` does as well; it inserts the number at the correct position by searching for the location of the number.

The biggest difference between these methods is that `bisect` does a binary search internally, which means that it starts in the middle and jumps to the middle of the left or right section, depending on whether the value in the list is bigger or smaller than the value we are looking for. To illustrate, we will search for `4` in a list of numbers from `0` to `14`:

```
sorted_list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
Step 1: 4 > 7                       ^
Step 2: 4 > 3           ^
Step 3: 4 > 5                 ^
Step 4: 4 > 5               ^
```

As you can see, after only four steps, we have found the number we searched for. Depending on the number (`7`, for example), it may go faster, but it will never take more than `O(log(n))` steps to find a number.

With a regular list, a search will simply walk through all the items until it finds the desired item. If you're lucky, it could be the first number you encounter, but if you're unlucky, it could be the last item. In the case of 1,000 items, that would be the difference between 1,000 steps and `log(1000) = 10` steps.

While very fast and efficient, the `bisect` module doesn't feel Pythonic at all. Let's fix that by creating our own `SortedList` class:

```
>>> import bisect
>>> import collections

>>> class SortedList:
...     def __init__(self, *values):
...         self._list = sorted(values)
...
...     def index(self, value):
...         i = bisect.bisect_left(self._list, value)
...         if i < len(self._list) and self._list[i] == value:
...             return index
...
...     def delete(self, value):
...         del self._list[self.index(value)]
...
...     def add(self, value):
...         bisect.insort(self._list, value)
...
...     def __iter__(self):
...         for value in self._list:
...             yield value
...
...     def __exists__(self, value):
...         return self.index(value) is not None

>>> sorted_list = SortedList(1, 3, 6, 2)
>>> 3 in sorted_list
True
>>> 5 in sorted_list
False
>>> sorted_list.add(5)
>>> 5 in sorted_list
True
>>> list(sorted_list)
[1, 2, 3, 5, 6]
```

While functional, this implementation is obviously still a tad limited. But it's certainly a nice starting point in case you need a structure like this.

Global instances using Borg or Singleton patterns

Most programmers will be familiar with the Singleton pattern, which ensures that only a single instance of a class will ever exist. Within Python, a common alternative solution to this is the Borg pattern, named after the Borg in Star Trek. Where a Singleton enforces a single instance, the Borg pattern enforces a single state for all instances and subclasses as well. Due to the way class creation works in Python, the Borg pattern is a tiny bit easier to implement and modify than the Singleton pattern as well.

To illustrate an example of both:

The Borg class:

```
>>> class Borg:
...     _state = {}
...     def __init__(self):
...         self.__dict__ = self._state

>>> class SubBorg(Borg):
...     pass

>>> a = Borg()
>>> b = Borg()
>>> c = Borg()
>>> a.a_property = 123
>>> b.a_property
123
>>> c.a_property
123
```

The Singleton class:

```
>>> class Singleton:
...     def __new__(cls):
...         if not hasattr(cls, '_instance'):
...             cls._instance = super(Singleton, cls).__new__(cls)
...
...         return cls._instance

>>> class SubSingleton(Singleton):
...     pass
```

```
>>> a = Singleton()
>>> b = Singleton()
>>> c = SubSingleton()
>>> a.a_property = 123
>>> b.a_property
123
>>> c.a_property
123
```

The Borg pattern works by overriding the `__dict__` of the instance that contains the instance state. The Singleton overrides the `__new__` (note, not `__init__`) method so that we only ever return a single instance of the class.

No need for getters and setters with properties

Within many languages (notably Java), a common design pattern for accessing instance variables is using getters and setters so that you can modify the behavior when needed in the future. Within Python, we can transparently change the behavior of attributes for existing classes without the need to touch the calling code:

```
>>> class Sandwich:
...     def __init__(self, spam):
...         self.spam = spam
...
...     @property
...     def spam(self):
...         return self._spam
...
...     @spam.setter
...     def spam(self, value):
...         self._spam = value
...         if self._spam >= 5:
...             print('You must be hungry')
...
...     @spam.deleter
...     def spam(self):
...         self._spam = 0

>>> sandwich = Sandwich(2)
>>> sandwich.spam += 1
>>> sandwich.spam += 2
You must be hungry
```

The calling code doesn't need to be changed at all. We can simply change the behavior of the property in a completely transparent way.

Dict union operators

This is not actually a separate advanced collection, but it is advanced usage of the `dict` collection. Since Python 3.9, we have a few easy options for combining multiple `dict` instances. The "old" solution was to use `dict.update()`, possibly combined with `dict.copy()` to create a new instance. While that works fine, it is rather verbose and a tad clunky.

Since this is a case where a few examples are much more useful than just explanation, let's see how the old solution works:

```
>>> a = dict(x=1, y=2)
>>> b = dict(y=1, z=2)

>>> c = a.copy()
>>> c
{'x': 1, 'y': 2}
>>> c.update(b)

>>> a
{'x': 1, 'y': 2}
>>> b
{'y': 1, 'z': 2}
>>> c
{'x': 1, 'y': 1, 'z': 2}
```

That solution works great, but with Python 3.9 and above we can do it in a much easier and shorter way:

```
>>> a = dict(x=1, y=2)
>>> b = dict(y=1, z=2)

>>> a | b
{'x': 1, 'y': 1, 'z': 2}
```

This is a feature that can be very convenient when specifying arguments to a function, especially if you want to automatically fill in keyword arguments with default arguments:

```
some_function(**(default_arguments | given_arguments))
```

Now that you have seen a few of the more advanced collections bundled with Python, you should have a pretty good idea of when to apply which type of collection. You may also have learned about a few new Python design patterns.

Exercises

In addition to enhancing the examples in this chapter, there are many other exercises:

- Create a `SortedDict` collection that takes a `keyfunc` to decide the sort order.
- Create a `SortedList` collection that has `O(log(n))` inserts and always returns a sorted list during each iteration.
- Create a Borg pattern that has a state per subclass.

Example answers for these exercises can be found on GitHub: `https://github.com/mastering-python/exercises`. You are encouraged to submit your own solutions and learn about alternative solutions from others.

Summary

Python is a bit unlike other languages in some aspects and several design patterns that are common in other languages make little sense in Python. In this chapter, you have seen some common Python design patterns, but many more patterns exist. Before you start implementing your own collections based on these patterns, quickly search the web to see whether there is an existing solution already. In particular, the `collections` module receives a lot of updates, so it is possible that your problem has already been solved.

If you are ever wondering how these structures work, have a look at the following source: `https://github.com/python/cpython/blob/master/Lib/collections/__init__.py`.

After finishing this chapter, you should be aware of the time complexities of the basic Python structures. You should also be familiar with a few Pythonic methods of tackling certain problems. Many of these examples use the `collections` module, but this chapter does not list all of the classes in the `collections` module.

Selecting the correct data structure within your applications is by far the most important performance factor for your code. This makes basic knowledge about performance characteristics essential for any serious programmer.

In the next chapter, we will continue with functional programming, which covers `lambda` functions, `list` comprehensions, `dict` comprehensions, `set` comprehensions, and an array of related topics. Additionally, you will learn about the mathematic background of functional programming.

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:
`https://discord.gg/QMzJenHuJf`

5

Functional Programming – Readability Versus Brevity

This chapter will show you some of the cool tricks that functional programming in Python gives you, and it will explain some of the limitations of Python's implementation. For learning and entertainment, we will also briefly discuss the mathematical equivalent using lambda calculus, using the **Y combinator** as an example.

The last few paragraphs will list and explain the usage of the `functools` and `itertools` libraries. If you are familiar with these libraries, feel free to skip them, but note that some of these will be used heavily in the later chapters about decorators (*Chapter 6*), generators (*Chapter 7*), and performance (*Chapter 12*).

These are the topics covered in this chapter:

- The theory behind functional programming
- `list`, `dict`, and `set` comprehensions
- `lambda` functions
- `functools` (`partial` and `reduce`)
- `itertools` (`accumulate`, `chain`, `dropwhile`, `starmap`, and so on)

First, we will begin with a bit of history about functional programming in Python and what functional programming actually means.

Functional programming

Functional programming is a paradigm that originates from the lambda calculus (λ-calculus), a formal system in mathematics that can be used to simulate any Turing machine. Without diving too much into the λ-calculus, this means that computation is performed using only the function arguments as input and that the output consists of a new variable without mutating the input variables. With a strictly functional programming language this behavior would be enforced, but since Python is not a strictly functional language, this doesn't necessarily hold true.

It is still a good idea to adhere to this paradigm since mixing paradigms can cause unforeseen bugs, as discussed in *Chapter 3, Pythonic Syntax and Common Pitfalls*.

Purely functional

Purely functional programming expects functions to have no side effects. That means that arguments given to the function should not be mutated, and neither should any other external states. Let's illustrate this with a simple example:

```
>>> def add_value_functional(items, value):
...     return items + [value]

>>> items = [1, 2, 3]
>>> add_value_functional(items, 5)
[1, 2, 3, 5]
>>> items
[1, 2, 3]

>>> def add_value_regular(items, value):
...     items.append(value)
...     return items

>>> add_value_regular(items, 5)
[1, 2, 3, 5]
>>> items
[1, 2, 3, 5]
```

That essentially shows the difference between a regular function and a purely functional one. The first function returns a *new* value purely based on the input, without any other side effects. This is in comparison to the second function, which modifies the given input or even variables outside of its scope.

Even outside of functional programming, limiting your changes to local variables only is a good idea. Keeping functions purely functional (relying only on the given input) makes code clearer, easier to understand, and better to test as there are fewer dependencies. Well-known examples can be found within the `math` module. These functions (`sin`, `cos`, `pow`, `sqrt`, and so on) have an input and an output that is strictly dependent on the input.

Functional programming and Python

Python is one of the few, or at least earliest, non-functional programming languages to add functional programming features. The initial few functional programming functions were introduced around 1993, and these were `lambda`, `reduce`, `filter`, and `map`. Since that time, Guido van Rossum has been less than happy with their existence because they often make readability suffer. Additionally, functions such as `map` and `filter` can easily be replicated using `list` comprehensions. Because of this, Guido wanted to remove these functions with the Python 3 release, but after a lot of resistance he opted for moving at least the `reduce` function to `functools.reduce`.

Since then, several other functional programming features have been added to Python:

- `list`/`dict`/`set` comprehensions
- Generator expressions
- Generator functions
- Coroutines

There are also a host of useful functions in the `functools` and `itertools` modules.

Advantages of functional programming

The big question is, of course, why would you want to use functional programming instead of regular/procedural programming? There are multiple advantages to writing code in a functional style:

- One major advantage of writing purely functional code is that it becomes trivially easy to run in parallel. Because there are no external variables needed and no external variables changed, you can easily parallelize the code to run on multiple processors or even on multiple machines. Assuming you can easily transfer the input variables and output results, of course.
- Because the functions are self-contained and don't have any side effects, they mitigate several kinds of bugs. Mutating function arguments in-place, for example, is a great source of bugs. Additionally, a seemingly useless function call that modifies a variable in the parent scope couldn't exist in a purely functional codebase.
- It makes testing much easier. If a function only has a given input and output and does not touch anything outside of those, you can test without having to set up an entire environment for that function. It also omits the need for sandboxing functions while testing them.

Naturally, functional programming also comes with a few drawbacks, several of which are caused by the same advantages.

In some cases it can be a hassle to pass along all useful arguments all of the time. When modifying a database for example, you need to get the database connection somehow. If you decide to pass the database connection as an argument and did not prepare for that, you will need to modify not just that function but all the calling functions as well to pass along that argument. In those cases a globally accessible variable containing the database connection could save you a lot of work.

Another often-touted downside of functional programming is recursion. While recursion is a very useful tool, it can make it much harder to trace the code execution path, which can be a problem when solving bugs.

Functional programming has its place and its time. It's not suited for every situation but when applied correctly it is a very useful tool for your toolbox. Now let's continue with some examples of functional programming.

list, set, and dict comprehensions

The Python `list`, `set`, and `dict` comprehensions are a very easy way to apply a function or filter to a list of items.

When used correctly, `list/set/dict` comprehensions can be really useful for quick filtering or transforming of lists, sets, and dicts. The same results can be achieved using the "functional" functions `map` and `filter`, but `list/set/dict` comprehensions are often easier to use and also easier to read.

Basic list comprehensions

Let's dive right into a few examples. The basic premise of a `list` comprehension looks like this:

```
>>> squares = [x ** 2 for x in range(10)]
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

We can easily expand this with a filter:

```
>>> odd_squares = [x ** 2 for x in range(10) if x % 2]
>>> odd_squares
[1, 9, 25, 49, 81]
```

This brings us to the version that is common in most functional languages using `map` and `filter`:

```
>>> def square(x):
...     return x ** 2

>>> def odd(x):
...     return x % 2

>>> squares = list(map(square, range(10)))
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

>>> odd_squares = list(filter(odd, map(square, range(10))))
>>> odd_squares
[1, 9, 25, 49, 81]
```

After seeing this it becomes slightly more obvious why Guido van Rossum wanted to remove these from the language. In particular, the version using both `filter` and `map` isn't all that readable given the number of parentheses, unless you're used to the Lisp programming language, that is.

The most important application of `map` is actually not using `map` itself, but using one of the `map`-like functions such as `multiprocessing.pool.Pool.map` and variants such as `map_async`, `imap`, `starmap`, `starmap_async`, and `imap_unordered`, which automatically execute the functions in parallel on multiple processors.

While I am personally not against `map` or `filter`, I think their usage should be reserved for cases where you have an existing function available to use in the `map` or `filter` call. A somewhat more useful example would be:

```
>>> import os

>>> directories = filter(os.path.isdir, os.listdir('.'))
# Versus:
>>> directories = [x for x in os.listdir('.') if os.path.isdir(x)]
```

In this case, the `filter` version might be slightly more readable than the `list` comprehension.

As for the `list` comprehensions, the syntax is pretty close to regular Python for loops, but the `if` statement and automatic storing of results make it quite useful to condense code slightly. The regular Python equivalent is not much longer:

```
>>> odd_squares = []
>>> for x in range(10):
...     if x % 2:
...         odd_squares.append(x ** 2)

>>> odd_squares
[1, 9, 25, 49, 81]
```

set comprehensions

In addition to `list` comprehensions, we can also use a `set` comprehension, which has the same syntax but returns a unique and unordered (all sets are unordered) set instead:

```
# List comprehension
>>> [x // 2 for x in range(3)]
[0, 0, 1]

# Set comprehension
>>> numbers = {x // 2 for x in range(3)}
>>> sorted(numbers)
[0, 1]
```

dict comprehensions

Lastly, we have `dict` comprehensions, which return a `dict` instead of a `list` or `set`.

Beyond the return type, the only real difference is that you need to return both a key and a value. The following is a basic example:

```
>>> {x: x ** 2 for x in range(6)}
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

>>> {x: x ** 2 for x in range(6) if x % 2}
{1: 1, 3: 9, 5: 25}
```

Since the output is a dictionary, the key needs to be hashable for the `dict` comprehension to work. We covered hashing in *Chapter 4*, but the short version is that `hash(key)` needs to return a consistent value for your object. That means that hashing mutable objects such as lists is not possible.

The funny thing is that you can mix these two, of course, for even more unreadable magic:

```
>>> {x ** 2: [y for y in range(x)] for x in range(5)}
{0: [], 1: [0], 4: [0, 1], 16: [0, 1, 2, 3], 9: [0, 1, 2]}
```

Obviously, you need to be careful with these. They can be very useful if used correctly, but the output quickly becomes unreadable, even with proper whitespace.

Comprehension pitfalls

When using comprehensions, some care must be taken. Some types of operations are not as obvious as you might expect. This time, we are looking for random numbers greater than `0.5`:

```
>>> import random

>>> [random.random() for _ in range(10) if random.random() >= 0.5]
[0.5211948104577864, 0.650010512129705, 0.021427316545174158]
```

See that last number? It's actually less than `0.5`. This happens because the first and the last random calls are actually separate calls and return different results.

One way to counter this is by creating the list separately from the filter:

```
>>> import random

>>> numbers = [random.random() for _ in range(10)]
>>> [x for x in numbers if x >= 0.5]
[0.715510247827078, 0.8426277505519564, 0.5071133900377911]
```

That obviously works, but it's not all that pretty. So what other options are there? Well, there are a few but the readability is a bit questionable, so these are not the solutions that I would recommend. It's good to see them at least once, however.

Here is a `list` comprehension within a `list` comprehension:

```
>>> import random

>>> [x for x in [random.random() for _ in range(10)] if x >= 0.5]
```

And here's one that quickly becomes an incomprehensible `list` comprehension:

```
>>> import random

>>> [x for _ in range(10) for x in [random.random()] if x >= 0.5]
```

Caution is needed with these options as the double list comprehension actually works like a nested `for` loop would, so it quickly generates a lot of results. To elaborate on this, consider:

```
>>> [(x, y) for x in range(3) for y in range(3, 5)]
[(0, 3), (0, 4), (1, 3), (1, 4), (2, 3), (2, 4)]
```

This effectively does the following:

```
>>> results = []
>>> for x in range(3):
...     for y in range(3, 5):
...         results.append((x, y))
...
>>> results
[(0, 3), (0, 4), (1, 3), (1, 4), (2, 3), (2, 4)]
```

These can be useful for some cases, but I would strongly recommend against nesting comprehensions as this quickly results in unreadable code. Understanding what is happening is still useful, however, so let's look at one more example. The following `list` comprehension swaps the column and row counts, so a 3 x 4 matrix becomes 4 x 3:

```
>>> matrix = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]

>>> reshaped_matrix = [
...     [
...         [y for x in matrix for y in x][i * len(matrix) + j]
...         for j in range(len(matrix))
...     ]
...     for i in range(len(matrix[0]))
... ]

>>> import pprint

>>> pprint.pprint(reshaped_matrix, width=40)
[[1, 2, 3],
 [4, 5, 6],
 [7, 8, 9],
 [10, 11, 12]]
```

Even with the extra indentation, the `list` comprehension just isn't all that readable. With four nested loops, that is expectedly so, of course. There are rare cases where nested list comprehensions might be justified, such as very basic matrix manipulation. In the general case, however, I would not recommend using nested comprehensions.

Next up, we will look at `lambda` functions, which can be combined with `map` and `filter` for short convenient functions.

lambda functions

The `lambda` statement in Python is simply an anonymous function. Due to the syntax, it is slightly more limited than regular functions, but a lot can be done through it. As always though, readability counts, so generally it is a good idea to keep it as simple as possible. One of the more common use cases is as the `sort` key for the `sorted` function:

```
>>> import operator

>>> values = dict(one=1, two=2, three=3)

>>> sorted(values.items())
[('one', 1), ('three', 3), ('two', 2)]

>>> sorted(values.items(), key=lambda item: item[1])
[('one', 1), ('two', 2), ('three', 3)]

>>> get_value = operator.itemgetter(1)
>>> sorted(values.items(), key=get_value)
[('one', 1), ('two', 2), ('three', 3)]
```

The first version sorts by key and the second sorts by the value. The last one shows an alternative option using `operator.itemgetter` to generate a function that gets a specific item.

The regular (non-lambda) function wouldn't be much more verbose but in these cases, a lambda function is a very useful shorthand. For completeness, let's look at both identical functions:

```
>>> key = lambda item: item[1]

>>> def key(item):
...     return item[1]
```

Do note that `PEP8` dictates that assigning a lambda to a variable is a bad idea (`https://peps.python.org/pep-0008/#programming-recommendations`). And logically, it is. The idea of an anonymous function is that it is just that—anonymous and without a name. If you are giving it an identity, you should define it as a normal function.

In my opinion, the only valid use case for a `lambda` function is as an anonymous one-line argument to a function such as `sorted()`.

The Y combinator

This section can easily be skipped. It is mostly an example of the mathematical value of the lambda statement.

The Y combinator is probably the most famous example of the λ-calculus:

$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

All this looks very complicated, but that's mostly because it uses the lambda calculus notation, which is not all that difficult if you look beyond the special characters.

To illustrate, you should read this syntax, $\lambda x.x^2$, as an anonymous (lambda) function that takes x as an input and returns x^2. In Python, this would be expressed almost exactly as it is in the original lambda calculus, except for replacing λ with `lambda` and `.` with `:`, so it results in `lambda x: x**2`.

With some algebra (`https://en.wikipedia.org/wiki/Fixed-point_combinator#Fixed-point_combinators_in_lambda_calculus`), the Y combinator can be reduced to $Yf = f(Yf)$, or a function that takes the f function and applies it to itself. The λ-calculus notation of this function is as follows:

$$\lambda x.f(xx)$$

Here is the Python notation for the `lambda` functions:

```
Y = lambda f: lambda *args: f(Y(f))(*args)
```

Or the regular function version:

```
def Y(f):
    def y(*args):
        y_function = f(Y(f))
        return y_function(*args)
    return y
```

This all comes down to a function that accepts a function `f` which gets called with that function as an argument using the Y combinator.

This might still be a bit unclear, so let's look at an example that actually uses it:

```
>>> Y = lambda f: lambda *args: f(Y(f))(*args)

>>> def factorial(combinator):
...     def _factorial(n):
...         if n:
...             return n * combinator(n - 1)
...         else:
```

```
...             return 1
...     return _factorial

>>> Y(factorial)(5)
120
```

The following is the short version, where the power of the Y combinator becomes more apparent, with a recursive anonymous function:

```
>>> Y = lambda f: lambda *args: f(Y(f))(*args)

>>> Y(lambda c: lambda n: n and n * c(n - 1) or 1)(5)
120
```

Note that the `n and n * c(n - 1) or 1` part is short for the `if` statement used in the longer version of the function. Alternatively, this can be written using the Python ternary operator:

```
>>> Y = lambda f: lambda *args: f(Y(f))(*args)

>>> Y(lambda c: lambda n: n * c(n - 1) if n else 1)(5)
120
```

You might be wondering about the point of this entire exercise. You could easily write a factorial function in regular Python that is shorter, easier and more idiomatic. So what is the point of the Y combinator? The Y combinator allows us to make a non-recursive function execute in a recursive way.

More importantly, however, I think it is an interesting demonstration of the power of Python — how you can implement something as fundamental as the lambda-calculus in a few lines of Python. I think it has a certain kind of beauty in its implementation.

One final example of the `Y` combinator will be given by the definition of `quicksort` in a few lines:

```
>>> quicksort = Y(lambda f:
...     lambda x: (
...         f([item for item in x if item < x[0]])
...         + [y for y in x if x[0] == y]
...         + f([item for item in x if item > x[0]])
...     ) if x else [])

>>> quicksort([1, 3, 5, 4, 1, 3, 2])
[1, 1, 2, 3, 3, 4, 5]
```

While the Y combinator most likely doesn't have much practical use in Python, it does show the power of the `lambda` statement and how close Python is to the fundamental mathematics behind it. Essentially, the difference is only in the notation and not in the functionality.

Now that we know how to write our own `lambda` and functional functions, we will take a look at the bundled functional functions in Python.

functools

In addition to the `list/dict/set` comprehensions, Python also has a few (more advanced) functions that can be really convenient when coding functionally. The `functools` library is a collection of functions that return callable objects. Some of these functions are used as decorators (we'll cover more about that in *Chapter 6*, *Decorators – Enabling Code Reuse by Decorating*), but the ones that we are going to talk about are used as straight-up functions to make your life easier.

partial — Prefill function arguments

The `partial` function is really convenient for adding some default arguments to a function that you use often but can't (or don't want to) redefine. With object-oriented code, you can usually work around cases similar to these, but with procedural code, you will often have to repeat your arguments. Let's take the `heapq` functions from *Chapter 4*, *Pythonic Design Patterns*, as an example:

```
>>> import heapq

>>> heap = []
>>> heapq.heappush(heap, 1)
>>> heapq.heappush(heap, 3)
>>> heapq.heappush(heap, 5)
>>> heapq.heappush(heap, 2)
>>> heapq.heappush(heap, 4)
>>> heapq.nsmallest(3, heap)
[1, 2, 3]
```

Almost all of the `heapq` functions require a `heap` argument, so we are going to make a shortcut that automatically fills the `heap` variable for us. This could easily be done with a regular function of course:

```
>>> def push(*args, **kwargs):
...     return heapq.heappush(heap, *args, **kwargs)
```

There is an easier method, however. Python comes bundled with a function called `functools.partial` that generates a function with pre-filled arguments:

```
>>> import functools
>>> import heapq

>>> heap = []
>>> push = functools.partial(heapq.heappush, heap)
>>> smallest = functools.partial(heapq.nsmallest, iterable=heap)

>>> push(1)
```

```
>>> push(3)
>>> push(5)
>>> push(2)
>>> push(4)
>>> smallest(3)
[1, 2, 3]
```

With `functools.partial` we can automatically fill in positional and/or keyword arguments for us. So a call to `push(...)` is automatically expanded to `heapq.heappush(heap, ...)`.

Why should we use `partial` instead of writing a `lambda` argument? Well, it's mostly about convenience, but it also helps solve the late binding problem discussed in *Chapter 3*, *Pythonic Syntax and Common Pitfalls*. Additionally, partial functions still behave somewhat similarly to the original function, which means they still have the documentation available and can be pickled, whereas `lambda` statements cannot.

The `pickle` module in Python allows serialization of many complex Python objects, but not all by default. The `lambda` functions have no defined `pickle` method by default, but this can be worked around by defining your own lambda-pickle method in `copy_reg.dispatch_table`. An easy way to achieve this is by using the `dill` library, which contains a whole range of `pickle` helpers.

To illustrate the difference between `lambda` and `functools.partial`, look at the following example:

```
>>> lambda_push = lambda x: heapq.heappush(heap, x)

>>> heapq.heappush
<built-in function heappush>
>>> push
functools.partial(<built-in function heappush>, [1, 2, 5, 3, 4])
>>> lambda_push
<function <lambda> at ...>

>>> heapq.heappush.__doc__
'Push item onto heap, maintaining the heap invariant.'
>>> push.__doc__
'partial(func, *args, **keywords) - new function ...'
>>> lambda_push.__doc__
```

Note how the `lambda_push.__doc__` doesn't return anything and the `lambda` only has a very unhelpful `<function <lambda> ...>` representation string. This is one of the reasons that `functools.partial` is far more convenient to use in practice. It shows the documentation from the reference function; the representation string shows exactly what it is doing and it can be pickled with no modification.

In *Chapter 6*, *Decorators – Enabling Code Reuse by Decorating* (specifically, in the section about `functools.wraps`), we will see how we can make functions copy attributes from other functions in a similar fashion to how `functools.partial` copies the documentation.

reduce — Combining pairs into a single result

The `reduce` function implements a mathematical technique called folding. It applies a pair of the previous result and the next item in the given list to the function that is passed.

The `reduce` function is supported by many languages but in most cases using different names such as `curry`, `fold`, `accumulate`, or `aggregate`. Python has actually supported `reduce` for a very long time, but since Python 3, it has been moved from the global scope to the `functools` library. Some code can be simplified beautifully using the `reduce` statement; whether it's readable or not is debatable, however.

Implementing a factorial function

One of the most used examples of `reduce` is for calculating factorials, which is indeed quite simple:

```
>>> import operator
>>> import functools

>>> functools.reduce(operator.mul, range(1, 5))
24
```

The preceding code uses `operator.mul` instead of `lambda a, b: a * b`. While they produce the same results, the former can be much faster.

Internally, the `reduce` function will do the following:

```
>>> from operator import mul

>>> mul(mul(mul(1, 2), 3), 4)
24
```

Or, creating a reduce function that automatically loops would look like:

```
>>> import operator

>>> def reduce(function, iterable):
...     print(f'iterable={iterable}')
...     # Fetch the first item to prime 'result'
...     result, *iterable = iterable
...
...     for item in iterable:
...         old_result = result
...         result = function(result, item)
...         print(f'{old_result} * {item} = {result}')
...
```

```
...     return result

>>> iterable = list(range(1, 5))
>>> iterable
[1, 2, 3, 4]

>>> reduce(operator.mul, iterable)
iterable=[1, 2, 3, 4]
1 * 2 = 2
2 * 3 = 6
6 * 4 = 24
24
```

Using the form `a, *b = c`, we can split an iterable between the first item and the remaining ones. Which means that `a, *b = [1, 2, 3]` will result in `a=1, b=[2, 3]`.

In this example, this means that we start by priming the `result` variable so it contains the initial value and continue to call the function with the current result and the next item until the iterable is exhausted.

Effectively, this comes down to:

1. `iterable = [1, 2, 3, 4]`
2. `result, *iterable = iterable`

   This gives us `result=1` and `iterable = [2, 3, 4]`.
3. Next up is the first call to operator.mul with the arguments result and item, which is stored in result. This is the big difference between reduce and map. Whereas map applies the function only to the given item, reduce applies both the previous result and the item to the function. So effectively, it runs result = operator.mul(result, item). Filling in the variables gives us `result = 1 * 2 = 2`.
4. The next call effectively repeats the process, but because of the previous call our initial `result` value is now `2` and the next `item` is `3`: `result = 2 * 3 = 6`.
5. We repeat this one more time because our `iterable` is now exhausted. The last call will run `result = 6 * 4 = 24`.

Processing trees

Trees are a case where the `reduce` function really shines. Remember the one-line tree definition using a `defaultdict` from *Chapter 4*, *Pythonic Design Patterns*. What would be a good way to access the keys inside of that object? Given a path of a tree item, we can use `reduce` to easily access the items inside. First, let's build a tree:

```
>>> import json
>>> import functools
>>> import collections
```

```
>>> def tree():
...     return collections.defaultdict(tree)

# Build the tree:
>>> taxonomy = tree()
>>> reptilia = taxonomy['Chordata']['Vertebrata']['Reptilia']
>>> reptilia['Squamata']['Serpentes']['Pythonidae'] = [
...     'Liasis', 'Morelia', 'Python']

# The actual contents of the tree
>>> print(json.dumps(taxonomy, indent=4))
{
    "Chordata": {
        "Vertebrata": {
            "Reptilia": {
                "Squamata": {
                    "Serpentes": {
                        "Pythonidae": [
                            "Liasis",
                            "Morelia",
                            "Python"
                        ]
                    }
                }
            }
        }
    }
}
```

First, we created a `tree` structure by using a recursive definition with `collections.defaultdict`. This allows us to nest the `tree` many levels deep without the need for explicit definitions.

To provide somewhat readable output, we use the `json` module to export the `tree` (which is effectively a list of nested dicts).

Now it's time for the lookup:

```
# Let's build the lookup function
>>> import operator

>>> def lookup(tree, path):
...     # Split the path for easier access
...     path = path.split('.')
```

```
...
...     # Use 'operator.getitem(a, b)' to get 'a[b]'
...     # And use reduce to recursively fetch the items
...     return functools.reduce(operator.getitem, path, tree)

>>> path = 'Chordata.Vertebrata.Reptilia.Squamata.Serpentes'
>>> dict(lookup(taxonomy, path))
{'Pythonidae': ['Liasis', 'Morelia', 'Python']}

# The path we wish to get
>>> path = 'Chordata.Vertebrata.Reptilia.Squamata'
>>> lookup(taxonomy, path).keys()
dict_keys(['Serpentes'])
```

Now we have a very simple way of walking through the `tree` structure recursively in just a few short lines of code.

Reducing in the other direction

People that are familiar with functional programming might wonder why Python only has the equivalent of `fold_left` and no `fold_right`. You honestly don't really need both of them as you can easily reverse the operation. To be fair, however, the same can be said of `reduce` as well since it is trivial to implement, as we have seen in the previous paragraph.

The regular `reduce`—the `fold left` operation:

```
fold_left = functools.reduce(
    lambda x, y: function(x, y),
    iterable,
    initializer,
)
```

The reverse—the `fold right` operation:

```
fold_right = functools.reduce(
    lambda x, y: function(y, x),
    reversed(iterable),
    initializer,
)
```

There may not be too many useful cases for `reduce`, but there are definitely a few. In particular, traversing recursive data structures is far more easily done using `reduce`, since it would otherwise involve more complicated loops or recursive functions.

Now that we have seen a few of the functional functions in Python, it is time to take a look at a few methods that focus on iterables instead.

itertools

The `itertools` library contains iterable functions inspired by those available in functional languages. All of these are iterable and have been constructed in such a way that only a minimal amount of memory is required to process even the largest of datasets. While you can easily write most of these functions yourself, I would still recommend using the ones available in the `itertools` library. These are all fast, memory efficient, and—perhaps more importantly—tested. We're going to explore a few now: `accumulate`, `chain`, `compress`, `dropwhile/takewhile`, `count`, and `groupby`.

accumulate — reduce with intermediate results

The `accumulate` function is very similar to the `reduce` function, which is why some languages actually have `accumulate` instead of `reduce` as the folding operator.

The major difference between the two is that the `accumulate` function returns the immediate results. This can be useful when summing the results of a company's sales, for example:

```
>>> import operator
>>> import itertools

# Sales per month
>>> months = [10, 8, 5, 7, 12, 10, 5, 8, 15, 3, 4, 2]
>>> list(itertools.accumulate(months, operator.add))
[10, 18, 23, 30, 42, 52, 57, 65, 80, 83, 87, 89]
```

It should be noted that the `operator.add` function is actually optional in this case as the default behavior of `accumulate` is to sum the results. In some other languages and libraries, this function is sometimes called `cumsum` (cumulative sum).

chain — Combining multiple results

The `chain` function is a simple but useful function that combines the results of multiple iterators. Very simple but also very useful if you have multiple lists, iterators, and so on—just combine them with a simple chain:

```
>>> import itertools

>>> a = range(3)
>>> b = range(5)
>>> list(itertools.chain(a, b))
[0, 1, 2, 0, 1, 2, 3, 4]
```

It should be noted that there is a small variant of `chain` that accepts an iterable containing iterables, namely `chain.from_iterable`. This works nearly identically, except for the fact that you need to pass along an iterable item instead of passing a list of arguments.

Your initial response might be that this can be achieved simply by unpacking the (`*args`) tuple, as we will see in *Chapter 7*, *Generators and Coroutines – Infinity, One Step at a Time*. However, this is not always the case. For now, just remember that if you have an iterable containing iterables, the easiest method is to use `itertools.chain.from_iterable`. The usage is as you would expect:

```
>>> import itertools

>>> iterables = [range(3), range(5)]
>>> list(itertools.chain.from_iterable(iterables))
[0, 1, 2, 0, 1, 2, 3, 4]
```

compress – Selecting items using a list of Booleans

The `compress` function is one of those that you won't need too often, but it can be very useful when you do need it. It applies a Boolean filter to your iterable, making it return only the elements you actually need. The most important thing to note here is that `compress` executes lazy and that `compress` will stop if either the data is exhausted, or no elements are being fetched anymore. So, even with infinite ranges, it works without a hitch:

```
>>> import itertools

>>> list(itertools.compress(range(1000), [0, 1, 1, 1, 0, 1]))
[1, 2, 3, 5]
```

The `compress` function can be useful if you want to make a filtered view of a larger iterable without modifying the original iterable. If calculating the filter is a heavy operation and the actual values inside the iterable can change, this can be very useful. To build on the example above:

```
>>> primes = [0, 0, 1, 1, 0, 1, 0, 1]
>>> odd = [0, 1, 0, 1, 0, 1, 0, 1]
>>> numbers = ['zero', 'one', 'two', 'three', 'four', 'five']

# Primes:
>>> list(itertools.compress(numbers, primes))
['two', 'three', 'five']

# Odd numbers
>>> list(itertools.compress(numbers, odd))
['one', 'three', 'five']

# Odd primes
>>> list(itertools.compress(numbers, map(all, zip(odd, primes))))
['three', 'five']
```

In this case, both the filters and the iterable are predefined and very small. But if you have a large set that takes a lot of time to compute (or fetch from an external resource), this method can be useful to quickly filter without having to recalculate everything, especially since the filters can be combined easily using a combination of `map`, `all`, and `zip`. You can use `any` instead of `all` if you want to see the results from both.

dropwhile/takewhile – Selecting items using a function

The `dropwhile` function will drop all results until a given predicate evaluates to true. This can be useful if you are waiting for a device to finally return an expected result. That's a bit difficult to demonstrate in a book, so we only have an example with the basic usage—waiting for a number greater than `3`:

```
>>> import itertools

>>> list(itertools.dropwhile(lambda x: x <= 3, [1, 3, 5, 4, 2]))
[5, 4, 2]
```

As you might expect, the `takewhile` function is the reverse of this. It will simply return all rows until the predicate turns false:

```
>>> import itertools

>>> list(itertools.takewhile(lambda x: x <= 3, [1, 3, 5, 4, 2]))
[1, 3]
```

Adding the results from `dropwhile` and `takewhile` will give you all the elements again as they are each other's opposites.

count – Infinite range with decimal steps

The `count` function is quite similar to the `range` function, but there are two significant differences:

- The first is that this range is infinite, so don't even try to do `list(itertools.count())`. You'll definitely run out of memory immediately and it might even freeze your system.
- The second difference is that, unlike the `range` function, you can actually use floating-point numbers here, so there is no need for whole/integer numbers.

Since listing the entire range will kill our Python interpreter, we'll limit the results using the `itertools.islice` function, which is similar to regular slicing (e.g. `some_list[10:20]`) but works on infinitely large inputs as well.

The infinitely large functions such as `count` are not sliceable because they are infinite generators, a topic we will discuss in *Chapter 7, Generators and Coroutines – Infinity, One Step at a Time*.

The `count` function takes two optional parameters: a `start` parameter, which defaults to `0`, and a `step` parameter, which defaults to `1`:

```
>>> import itertools

>>> list(itertools.islice(itertools.count(), 10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> list(itertools.islice(itertools.count(), 5, 10, 2))
[5, 7, 9]

>>> list(itertools.islice(itertools.count(10, 2.5), 5))
[10, 12.5, 15.0, 17.5, 20.0]
```

groupby – Grouping your sorted iterable

The `groupby` function is a really convenient function for grouping results. It allows you to convert a list of objects into a list of groups given a specific grouping function.

A basic example of `groupby` usage:

```
>>> import operator
>>> import itertools

>>> words = ['aa', 'ab', 'ba', 'bb', 'ca', 'cb', 'cc']

# Gets the first element from the iterable
>>> getter = operator.itemgetter(0)

>>> for group, items in itertools.groupby(words, key=getter):
...     print(f'group: {group}, items: {list(items)}')
group: a, items: ['aa', 'ab']
group: b, items: ['ba', 'bb']
group: c, items: ['ca', 'cb', 'cc']
```

We can see here how the words are grouped by the first character with very little effort. This can be a really convenient utility for grouping employees by department in a user interface, for example.

There are some important things to keep in mind when using this function, however:

- The input needs to be sorted by the `group` parameter. Otherwise, every repeated group will be added as a separate group.
- The results are available for use only once. So, after processing a group, it will not be available anymore. If you wish to iterate the results twice, wrap the results in `list()` or `tuple()`.

Here is an example of groupby including the side effects of not sorting:

```
>>> import itertools

>>> raw_items = ['spam', 'eggs', 'sausage', 'spam']

>>> def keyfunc(group):
...     return group[0]

>>> for group, items in itertools.groupby(raw_items, key=keyfunc):
...     print(f'group: {group}, items: {list(items)}')
group: s, items: ['spam']
group: e, items: ['eggs']
group: s, items: ['sausage', 'spam']

>>> raw_items.sort()
>>> for group, items in itertools.groupby(raw_items, key=keyfunc):
...     print(f'group: {group}, items: {list(items)}')
group: e, items: ['eggs']
group: s, items: ['sausage', 'spam', 'spam']
```

The groupby function is definitely a very useful one that you can use in a wide variety of scenarios. Grouping output for a user, for example, can make results much easier to read.

Exercises

Now that you know how to use some of the functional programming features in Python, perhaps you can try writing the quicksort algorithm as (a collection of) regular functions instead of the hard-to-read Y-combinator version.

You can also try and write a groupby function yourself that isn't affected by sorting and returns lists of results that can be used multiple times rather than just once.

Example answers for these exercises can be found on GitHub: `https://github.com/mastering-python/exercises`. You are encouraged to submit your own solutions and learn about alternative solutions from others.

Summary

Functional programming is a paradigm that scares many people initially, but really it shouldn't. The most important difference between functional and procedural programming (within Python) is the mindset. Everything is executed using simple functions that depend only on their input variables and don't produce any side effects outside of the local scope.

The main advantages are:

- Because there are fewer side-effects and code influencing each other, you will get fewer bugs.
- Because the functions always have a predictable input and output, they can be easily parallelized across multiple processors or even multiple machines.

This chapter covered the basics of functional programming within Python and a tiny portion of the mathematics behind it. In addition to this, some of the many useful libraries that can be used in a very convenient way by using functional programming were covered.

The most important takeaways should be the following:

- Lambda statements are not inherently bad, but it would be best to make them use variables from the local scope only, and they should not be longer than a single line.
- Functional programming can be very powerful, but has a tendency to become unreadable. Care must be taken.
- `list/dict/set` comprehensions are very useful but have a tendency to quickly become unreadable. In particular, nested comprehensions are hard to read in nearly all cases and should mostly be avoided.

Ultimately, it is a matter of preference. For the sake of readability, I recommend limiting the usage of the functional paradigm when there is no obvious benefit. Having said that, when executed correctly, it can be a thing of beauty.

Next up are decorators – methods to wrap your functions and classes in other functions and/or classes to modify their behavior and extend their functionality.

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:
`https://discord.gg/QMzJenHuJf`

6

Decorators – Enabling Code Reuse by Decorating

In this chapter, you are going to learn about Python decorators. The previous chapters have already shown the usage of a few decorators, but you will now find out more about them. Decorators are essentially function/class wrappers that can be used to modify the input, output, or even the function/class itself before executing it. This type of wrapping can just as easily be achieved by having a separate function that calls the inner function, or via inheriting small feature classes commonly called **mixins**. As is the case with many Python constructs, decorators are not the only way to reach the goal but are definitely convenient in many cases.

While you can get along fine without knowing too much about decorators, they give you a lot of "reuse power" and are therefore used heavily in framework libraries such as web frameworks. Python actually comes bundled with some useful decorators, most notably the `@property`, `@classmethod`, and `@staticmethod` decorators.

There are, however, some particularities to take note of: wrapping a function creates a new function and makes it harder to reach the inner function and its properties. One example of this is the `help(function)` functionality of Python; by default, you, your editor, and your documentation generator can lose function properties such as the help text and the module the function exists in.

This chapter will cover the usage of both function and class decorators, as well as the intricate details you need to know when decorating functions within classes.

The following are the topics covered:

- Decorating functions
- Decorating class functions
- Decorating classes
- Useful decorators in the Python Standard Library

Decorating functions

Decorators are functions or classes that wrap other functions and/or classes. In its most basic form, you can view a regular function call as `add(1, 2)`, which transforms into `decorator(add(1, 2))` when applying a decorator. There's slightly more to it, but we will come to that later. Let's implement that `decorator()` function:

```
>>> def decorator(function):
...     return function

>>> def add(a, b):
...     return a + b

>>> add = decorator(add)
```

To make the syntax easier to use, Python has a special syntax for this case. So, instead of adding a line such as the preceding one below the function, you can decorate a function using the `@` operator as a shortcut:

```
>>> @decorator
... def add(a, b):
...     return a + b
```

This example shows the simplest and most useless decorator you can get: simply returning the input function and doing nothing else.

From this, you might wonder what the use of a decorator is and what is so special about them. Some possibilities of decorators are:

- Registering a function/class
- Modifying function/class input
- Modifying function/class output
- Logging function calls/class instantiations

All of these will be covered later in this chapter, but let's start simple for now.

Our first decorator will show how we can modify both the input and the output of a function call. Additionally, it adds some `logging` calls so we can see what is happening:

```
>>> import functools

>>> def decorator(function):
...     # This decorator makes sure we mimic the wrapped function
...     @functools.wraps(function)
...     def _decorator(a, b):
...         # Pass the modified arguments to the function
```

```
...         result = function(a, b + 5)
...
...         # Log the function call
...         name = function.__name__
...         print(f'{name}(a={a}, b={b}): {result}')
...
...         # Return a modified result
...         return result + 4
...
...     return _decorator

>>> @decorator
... def func(a, b):
...     return a + b

>>> func(1, 2)
func(a=1, b=2): 8
12
```

This should show you how powerful decorators can be. We can modify, add, and/or remove arguments. We can modify the return value or even call a completely different function if we want to. And we can easily log all behavior if needed, which can be very useful when debugging. Instead of `return function(...)`, we can return something completely different if we wish.

More extensive examples of how to log using decorators are covered in *Chapter 12, Debugging – Solving the Bugs*.

Generic function decorators

The decorator we wrote earlier explicitly used the `a` and `b` arguments so it only works with functions that have a signature very similar to taking `a` and `b` arguments. If we want to make the generator more generic, we can replace `a, b` with `*args` and `**kwargs` to get the arguments and keyword arguments, respectively. That introduces a new problem, however. We either need to make sure to only use regular arguments or keyword arguments, or the checking will become increasingly difficult:

```
>>> import functools

>>> def decorator(function):
...     @functools.wraps(function)
...     def _decorator(*args, **kwargs):
...         a, b = args
...         return function(a, b + 5)
...
```

```
...     return _decorator

>>> @decorator
... def func(a, b):
...     return a + b

>>> func(1, 2)
8

>>> func(a=1, b=2)
Traceback (most recent call last):
...
ValueError: not enough values to unpack (expected 2, got 0)
```

As can be seen, in this case, keyword arguments are broken. To work around this issue, we have a few different methods. We can change the arguments to positional-only or keyword-only arguments:

This code uses positional-only arguments (the `/` as the last function argument), which have been supported since Python 3.8. For older versions, you can emulate this behavior using `*args` instead of explicit arguments.

```
>>> def add(a, b, /):
...     return a + b

>>> add(a=1, b=2)
Traceback (most recent call last):
...
TypeError: add() got some positional-only arguments passed ...

>>> def add(*, a, b):
...     return a + b

>>> add(1, 2)
Traceback (most recent call last):
...
TypeError: add() takes 0 positional arguments but 2 were given
```

Or we can make Python automatically take care of this by fetching the signature and binding it to the given arguments:

```
>>> import inspect
>>> import functools
```

```
>>> def decorator(function):
...     # Use the inspect module to get function signature. More
...     # about this in the logging chapter
...     signature = inspect.signature(function)
...
...     @functools.wraps(function)
...     def _decorator(*args, **kwargs):
...         # Bind the arguments to the given *args and **kwargs.
...         # If you want to make arguments optional, use
...         # signature.bind_partial instead.
...         bound = signature.bind(*args, **kwargs)
...
...         # Apply the defaults so b is always filled
...         bound.apply_defaults()
...
...         # Extract the filled arguments. If the number of
...         # arguments is still expected to be fixed, you can use
...         # tuple unpacking: 'a, b = bound.arguments.values()'
...         a = bound.arguments['a']
...         b = bound.arguments['b']
...         return function(a, b + 5)
...
...     return _decorator

>>> @decorator
... def func(a, b=3):
...     return a + b

>>> func(1, 2)
8

>>> func(a=1, b=2)
8

>>> func(a=1)
9
```

By using this method, the function has become a lot more versatile. We could easily add arguments to the `add` function and still be sure that the decorator functions.

The importance of functools.wraps

Whenever you are writing a decorator, always be sure to add `functools.wraps` to wrap the inner function. Without wrapping it, you will lose all properties from the original function, which can lead to confusion and unexpected behavior. Take a look at the following code without `functools.wraps`:

```
>>> def decorator(function):
...     def _decorator(*args, **kwargs):
...         return function(*args, **kwargs)
...
...     return _decorator

>>> @decorator
... def add(a, b):
...     '''Add a and b'''
...     return a + b

>>> help(add)
Help on function _decorator in module ...:
<BLANKLINE>
_decorator(*args, **kwargs)
<BLANKLINE>

>>> add.__name__
'_decorator'
```

Now, our `add` method has no documentation anymore and the name is gone. It has been renamed `_decorator`. Since we are indeed calling `_decorator`, this is understandable, but it's very inconvenient for code that relies on this information. Now we will try the same code with a minor difference; we will use `functools.wraps`:

```
>>> import functools

>>> def decorator(function):
...     @functools.wraps(function)
...     def _decorator(*args, **kwargs):
...         return function(*args, **kwargs)
...
...     return _decorator

>>> @decorator
... def add(a, b):
...     '''Add a and b'''
```

```
...     return a + b

>>> help(add)
Help on function add in module ...:
<BLANKLINE>
add(a, b)
    Add a and b
<BLANKLINE>

>>> add.__name__
'add'
```

Without any further changes, we now have documentation and the expected function name. The working of `functools.wraps` is nothing magical; it copies and updates several attributes. Specifically, the following attributes are copied:

- `__doc__`
- `__name__`
- `__module__`
- `__annotations__`
- `__qualname__`

Also, `__dict__` is updated using `_decorator.__dict__.update(add.__dict__)`, and a new property called `__wrapped__` is added, which contains the original function (`add`, in this case). The actual `wraps` function is available in the `functools.py` file of your Python distribution.

Chaining or nesting decorators

Since we're wrapping functions, there is nothing stopping us from adding multiple wrappers. The order is important to keep in mind, though, because the decorators are initialized starting from the inside, but are called starting from the outside. Additionally, the teardown starts from the inside again:

```
>>> import functools

>>> def track(function=None, label=None):
...     # Trick to add an optional argument to our decorator
...     if label and not function:
...         return functools.partial(track, label=label)
...
...     print(f'initializing {label}')
...
...     @functools.wraps(function)
...     def _track(*args, **kwargs):
...         print(f'calling {label}')
```

```
...         function(*args, **kwargs)
...         print(f'called {label}')
...
...     return _track

>>> @track(label='outer')
... @track(label='inner')
... def func():
...     print('func')
initializing inner
initializing outer

>>> func()
calling outer
calling inner
func
called inner
called outer
```

As you can see in the output, the decorators are called from outer to inner before running the function and running from inner to outer when processing the results.

Registering functions using decorators

We have seen how calls can be tracked, arguments can be modified, and return values can be changed. Now it is time to see how we can use decorators to register a function that can be useful for registering plugins, callbacks, and so on.

One situation where this is very useful is a user interface. Let us assume we have a GUI that has a button that can be clicked. By creating a system that can register callbacks, we can make the button fire a "clicked" signal and connect functions to that event.

To create an event manager like that, we will now create a class that keeps track of all of the registered functions and allows the firing of events:

```
>>> import collections

>>> class EventRegistry:
...     def __init__(self):
...         self.registry = collections.defaultdict(list)
...
...     def on(self, *events):
...         def _on(function):
...             for event in events:
```

```
...                 self.registry[event].append(function)
...             return function
...
...         return _on
...
...     def fire(self, event, *args, **kwargs):
...         for function in self.registry[event]:
...             function(*args, **kwargs)

>>> events = EventRegistry()

>>> @events.on('success', 'error')
... def teardown(value):
...     print(f'Tearing down got: {value}')

>>> @events.on('success')
... def success(value):
...     print(f'Successfully executed: {value}')

>>> events.fire('non-existing', 'nothing to see here')
>>> events.fire('error', 'Oops, some error here')
Tearing down got: Oops, some error here
>>> events.fire('success', 'Everything is fine')
Tearing down got: Everything is fine
Successfully executed: Everything is fine
```

Firstly, we create the `EventRegistry` class to handle all of the events and store all the callbacks. After that, we register a few functions with the registry. Lastly, we fire a few events to see if it works as expected.

While this example is rather basic, this pattern can be applied to many scenarios: handling events for a web server, letting plugins register themselves for events, letting plugins register themselves in an application, and so on.

Memoization using decorators

Memoization is a simple trick for remembering results to make code run a lot faster in specific scenarios. The basic trick here is to store a mapping of the input and expected output so that you have to calculate a value only once. One of the most common examples of this technique is the naïve (recursive) Fibonacci function.

The Fibonacci sequence starts from 0 or 1 (depending how you look at it) and each consecutive number consists of the sum of the previous two numbers. To illustrate the pattern starting from the additions of the initial `0` and `1`:

```
1 = 0 + 1
2 = 1 + 1
3 = 1 + 2
5 = 2 + 3
8 = 3 + 5
```

I will now show how you can build a very basic memoization function decorator, and how it can be used:

```
>>> import functools

>>> def memoize(function):
...     # Store the cache as attribute of the function so we can
...     # apply the decorator to multiple functions without
...     # sharing the cache.
...     function.cache = dict()
...
...     @functools.wraps(function)
...     def _memoize(*args):
...         # If the cache is not available, call the function
...         # Note that all args need to be hashable
...         if args not in function.cache:
...             function.cache[args] = function(*args)
...         return function.cache[args]
...
...     return _memoize
```

The memoize decorator has to be used without arguments and the cache can be introspected as well:

```
>>> @memoize
... def fibonacci(n):
...     if n < 2:
...         return n
...     else:
...         return fibonacci(n - 1) + fibonacci(n - 2)

>>> for i in range(1, 7):
...     print(f'fibonacci {i}: {fibonacci(i)}')
fibonacci 1: 1
fibonacci 2: 1
```

```
fibonacci 3: 2
fibonacci 4: 3
fibonacci 5: 5
fibonacci 6: 8

>>> fibonacci.__wrapped__.cache
{(1,): 1, (0,): 0, (2,): 1, (3,): 2, (4,): 3, (5,): 5, (6,): 8}
```

When arguments are given, it breaks because the decorator is not built to support them:

```
# It breaks keyword arguments:
>>> fibonacci(n=2)
Traceback (most recent call last):
...
TypeError: _memoize() got an unexpected keyword argument 'n'
```

Additionally, the arguments need to be hashable to work with this implementation:

```
# Unhashable types don't work as dict keys:
>>> fibonacci([123])
Traceback (most recent call last):
...
TypeError: unhashable type: 'list'
```

While examples with a small `n` will work easily without memoization, for larger numbers it will run for an extremely long time. For `n=2`, the function would execute `fibonacci(n - 1)` and `fibonacci(n - 2)` recursively, resulting in exponential time complexity. For `n=30`, the Fibonacci function would already be called 2,692,537 times; at `n=50`, it will stall or even crash your system.

Without memoization, the call stack becomes a tree that very quickly grows. To illustrate, let's assume we want to calculate `fibonacci(4)`.

First, `fibonacci(4)` calls `fibonacci(3)` and `fibonacci(2)`. There's nothing special here.

Now, `fibonacci(3)` calls `fibonacci(2)` and `fibonacci(1)`. You will notice that we got `fibonacci(2)` for the second time now. `fibonacci(4)` also executed it.

That split with each call is exactly the problem. Each function call starts two new function calls, which means it doubles for every call. And those double again and again until we have reached the end of the calculation.

Because the memoized version caches the results and only needs to calculate every number once, it doesn't even break a sweat and only needs to execute `31` times for `n=30`.

This decorator also shows how a context can be attached to a function itself. In this case, the cache property becomes an attribute of the internal (wrapped `fibonacci`) function so that an extra `memoize` decorator for a different object won't clash with any of the other decorated functions.

Note, however, that implementing the memoization function yourself is generally not that useful anymore since Python introduced `lru_cache` (**least recently used cache**) in Python 3.2. The `lru_cache` is similar to the preceding memoize decorator function but a bit more advanced. It maintains a fixed cache size (`128` by default) to save memory, and stores statistics so you can check whether the cache size should be increased.

If you are only looking for statistics and have no need for caching, you can also set the `maxsize` to `0`. Or if you want to forego the LRU algorithm and save everything, you can pass `None` as `maxsize`. With a fixed size, the `lru_cache` will keep only the most recently accessed items and discard the oldest once it is full.

In most cases, I would suggest using `lru_cache` over your own decorator, but if you always need to store all items or if you need to process the keys before storing them, you can always roll your own. At the very least, it is useful to know how to write a decorator like this.

To demonstrate how `lru_cache` works internally, we will calculate `fibonacci(100)`, which would keep our computer busy until the end of the universe without any caching. Moreover, to make sure that we can actually see how many times the `fibonacci` function is being called, we'll add an extra decorator that keeps track of the count, as follows:

```
>>> import functools

# Create a simple call counting decorator
>>> def counter(function):
...     function.calls = 0
...     @functools.wraps(function)
...     def _counter(*args, **kwargs):
...         function.calls += 1
...         return function(*args, **kwargs)
...
...     return _counter

# Create a LRU cache with size 3
>>> @functools.lru_cache(maxsize=3)
... @counter
... def fibonacci(n):
...     if n < 2:
...         return n
...     else:
...         return fibonacci(n - 1) + fibonacci(n - 2)

>>> fibonacci(100)
354224848179261915075

# The LRU cache offers some useful statistics
```

```
>>> fibonacci.cache_info()
CacheInfo(hits=98, misses=101, maxsize=3, currsize=3)

# The result from our counter function which is now wrapped both by
# our counter and the cache
>>> fibonacci.__wrapped__.__wrapped__.calls
101
```

You might wonder why we need only 101 calls with a cache size of `3`. That's because we recursively require only `n - 1` and `n - 2`, so we have no need for a larger cache in this case. If your cache is not performing as expected, the cache size might be the culprit.

Additionally, this example shows the usage of two decorators for a single function. You can see these as the layers of an onion. When calling `fibonacci`, the execution order is as follows:

1. `functools.lru_cache`
2. `counter`
3. `fibonacci`

Returning the values works in the reverse order, of course; `fibonacci` returns its value to `counter`, which passes the value along to `lru_cache`.

Decorators with (optional) arguments

The previous examples mostly used simple decorators without any arguments. As you have already seen with `lru_cache`, decorators can accept arguments as well since they are just regular functions, but this adds an extra layer to a decorator. This means that we need to check the decorator arguments to see if they are the decorated method or a regular argument. The only caveat is that the optional argument should not be callable. If the argument has to be callable, you will need to pass it as a keyword argument instead.

The upcoming code shows a decorator that has an optional (keyword) argument to the decorator:

```
>>> import functools

>>> def add(function=None, add_n=0):
...     # function is not callable so it's probably 'add_n'
...     if not callable(function):
...         # Test to make sure we don't pass 'None' as 'add_n'
...         if function is not None:
...             add_n = function
...         return functools.partial(add, add_n=add_n)
...
...     @functools.wraps(function)
...     def _add(n):
...         return function(n) + add_n
```

```
...
...     return _add

>>> @add
... def add_zero(n):
...     return n

>>> @add(1)
... def add_one(n):
...     return n

>>> @add(add_n=2)
... def add_two(n):
...     return n

>>> add_zero(5)
5

>>> add_one(5)
6

>>> add_two(5)
7
```

This decorator uses the `callable()` test to see whether the argument is a callable such as a function. This method works in many cases, but if for some reason your argument to the `add()` decorator is callable, this will break because it will be called instead of the function.

Whenever you have the choice available, I recommend that you either have a decorator with arguments or without them. Having optional arguments makes the flow of the function less obvious and slightly harder to debug when issues arise.

Creating decorators using classes

Similar to how we create regular function decorators, it is also possible to create decorators using classes instead. As is always the case with classes, this makes storing data, inheriting, and reuse more convenient than with functions. After all, a function is just a callable object and a class can implement the callable interface as well. The following decorator works similarly to the `debug` decorator we used earlier, but uses a class instead of a regular function:

```
>>> import functools

>>> class Debug(object):
```

```
...
...     def __init__(self, function):
...         self.function = function
...         # functools.wraps for classes
...         functools.update_wrapper(self, function)
...
...     def __call__(self, *args, **kwargs):
...         output = self.function(*args, **kwargs)
...         name = self.function.__name__
...         print(f'{name}({args!r}, {kwargs!r}): {output!r}')
...         return output

>>> @Debug
... def add(a, b=0):
...     return a + b
...

>>> output = add(3)
add((3,), {}): 3

>>> output = add(a=4, b=2)
add((), {'a': 4, 'b': 2}): 6
```

The only notable difference between functions and classes is that `functools.wraps` is now replaced with `functools.update_wrapper` in the `__init__` method.

Since class methods have a `self` argument in addition to the regular arguments, you might wonder whether decorators will function in that scenario. The next section will cover decorator usage within classes.

Decorating class functions

Decorating class functions is very similar to regular functions, but you need to be aware of the required first argument, `self`—the class instance. You have most likely already used a few class function decorators. The `classmethod`, `staticmethod`, and `property` decorators, for example, are used in many different projects. To explain how all this works, we will build our own versions of the `classmethod`, `staticmethod`, and `property` decorators. First, let's look at a simple decorator for class functions to demonstrate the difference from regular decorators:

```
>>> import functools

>>> def plus_one(function):
```

```
...     @functools.wraps(function)
...     def _plus_one(self, n, *args):
...         return function(self, n + 1, *args)
...
...     return _plus_one

>>> class Adder(object):
...     @plus_one
...     def add(self, a, b=0):
...         return a + b

>>> adder = Adder()
>>> adder.add(0)
1
>>> adder.add(3, 4)
8
```

As is the case with regular functions, the class function decorator now gets passed along `self` as the instance. Nothing unexpected!

Skipping the instance – classmethod and staticmethod

The difference between a `classmethod` and a `staticmethod` is fairly simple. The `classmethod` passes a class object instead of a class instance (`self`), and `staticmethod` skips both the class and the instance entirely. This effectively makes `staticmethod` very similar to a regular function outside of a class.

In the following examples, we will use `pprint.pprint(... width=60)` to account for the width of the book. Additionally, `locals()` is a Python built-in that shows all local variables. Similarly, a `globals()` function is also available.

Before we recreate `classmethod` and `staticmethod`, we need to take a look at the expected behavior of these methods:

```
>>> import pprint

>>> class Spam(object):
...     def some_instancemethod(self, *args, **kwargs):
...         pprint.pprint(locals(), width=60)
...
...     @classmethod
...     def some_classmethod(cls, *args, **kwargs):
...         pprint.pprint(locals(), width=60)
...
```

```
...     @staticmethod
...     def some_staticmethod(*args, **kwargs):
...         pprint.pprint(locals(), width=60)

# Create an instance so we can compare the difference between
# executions with and without instances easily
>>> spam = Spam()
```

The following examples will use the example above to illustrate the difference between a regular (class instance) method, a `classmethod`, and a `staticmethod`. Be wary of the difference between `spam` (lowercase) the instance and `Spam` (capitalized) the class:

```
# With an instance (note the lowercase spam)
>>> spam.some_instancemethod(1, 2, a=3, b=4)
{'args': (1, 2),
 'kwargs': {'a': 3, 'b': 4},
 'self': <__main__.Spam object at ...>}

# Without an instance (note the capitalized Spam)
>>> Spam.some_instancemethod()
Traceback (most recent call last):
    ...
TypeError: some_instancemethod() missing ... argument: 'self'

# But what if we add parameters? Be very careful with these!
# Our first argument is now used as an argument, this can give
# very strange and unexpected errors
>>> Spam.some_instancemethod(1, 2, a=3, b=4)
{'args': (2,), 'kwargs': {'a': 3, 'b': 4}, 'self': 1}
```

In particular, the last example is rather tricky. Because we passed some arguments to the function, these have automatically been passed as the `self` argument. Similarly, the last example shows how you can use this argument handling to call a method using a given instance. `Spam.some_instancemethod(spam)` is identical to `spam.some_instancemethod()`.

Now let's look at the `classmethod`:

```
# Classmethods are expectedly identical
>>> spam.some_classmethod(1, 2, a=3, b=4)
{'args': (1, 2),
 'cls': <class '__main__.Spam'>,
 'kwargs': {'a': 3, 'b': 4}}

>>> Spam.some_classmethod()
```

```
{'args': (), 'cls': <class '__main__.Spam'>, 'kwargs': {}}

>>> Spam.some_classmethod(1, 2, a=3, b=4)
{'args': (1, 2),
 'cls': <class '__main__.Spam'>,
 'kwargs': {'a': 3, 'b': 4}}
```

The main difference here is that instead of `self` we now have `cls`, which contains the class (`Spam`) instead of the instance (`spam`).

The names `self` and `cls` are conventions and are not enforced in any way. You could easily call them `s` and `c` or something completely different instead.

Next up is the `staticmethod`. The `staticmethod` behaves identically to a regular function outside of a class.

```
# Staticmethods are also identical
>>> spam.some_staticmethod(1, 2, a=3, b=4)
{'args': (1, 2), 'kwargs': {'a': 3, 'b': 4}}

>>> Spam.some_staticmethod()
{'args': (), 'kwargs': {}}

>>> Spam.some_staticmethod(1, 2, a=3, b=4)
{'args': (1, 2), 'kwargs': {'a': 3, 'b': 4}}
```

Before we can continue with decorators, you need to be aware of how Python descriptors function. Descriptors can be used to modify the binding behavior of object attributes. This means that if a descriptor is used as the value of an attribute, you can modify which value is being set, got, and deleted when these operations are called on the attribute. Here is a basic example of this behavior:

```
>>> class Spam:
...     def __init__(self, spam=1):
...         self.spam = spam
...
...     def __get__(self, instance, cls):
...         return self.spam + instance.eggs
...
...     def __set__(self, instance, value):
...         instance.eggs = value - self.spam

>>> class Sandwich:
```

```
...     spam = Spam(5)
...
...     def __init__(self, eggs):
...         self.eggs = eggs

>>> sandwich = Sandwich(1)
>>> sandwich.eggs
1
>>> sandwich.spam
6

>>> sandwich.eggs = 10
>>> sandwich.spam
15
```

As you can see, whenever we set or get values from `sandwich.spam`, it actually calls `__get__` or `__set__` on `Spam`, which has access not only to its own variables, but also the calling class. A very useful feature for automatic conversions and type checking, the `property` decorator we will see in the next section is just a more convenient implementation of this technique.

Now that you know how descriptors work, we can continue with creating the `classmethod` and `staticmethod` decorators. For these two, we simply need to modify `__get__` instead of `__call__` so that we can control which type of instance (or none at all) is passed along:

```
>>> import functools

>>> class ClassMethod(object):
...     def __init__(self, method):
...         self.method = method
...
...     def __get__(self, instance, cls):
...         @functools.wraps(self.method)
...         def method(*args, **kwargs):
...             return self.method(cls, *args, **kwargs)
...
...         return method

>>> class StaticMethod(object):
...     def __init__(self, method):
...         self.method = method
...
...     def __get__(self, instance, cls):
...         return self.method
```

```
>>> class Sandwich:
...     spam = 'class'
...
...     def __init__(self, spam):
...         self.spam = spam
...
...     @ClassMethod
...     def some_classmethod(cls, arg):
...         return cls.spam, arg
...
...     @StaticMethod
...     def some_staticmethod(arg):
...         return Sandwich.spam, arg

>>> sandwich = Sandwich('instance')
>>> sandwich.spam
'instance'
>>> sandwich.some_classmethod('argument')
('class', 'argument')
>>> sandwich.some_staticmethod('argument')
('class', 'argument')
```

The `ClassMethod` decorator still features a sub-function to actually produce a working decorator. Looking at the function, you can most likely guess how it functions. Instead of passing `instance` as the first argument to `self.method`, it passes `cls`.

`StaticMethod` is even simpler, because it completely ignores both the `instance` and the `cls`. It can just return the original method unmodified. Because it returns the original method without any modifications, we have no need for the `functools.wraps` call either.

Properties — Smart descriptor usage

The `property` decorator is probably the most used decorator in Python land. It allows you to add getters/setters to existing instance properties so that you can add validators and modify your values before setting them to your instance properties.

The `property` decorator can be used both as an assignment and as a decorator. The following example shows both syntaxes so that you know what to expect from the `property` decorator.

Python 3.8 added `functools.cached_property`, which functions the same as `property` but executes only once per instance.

```
>>> import functools

>>> class Sandwich(object):
```

```
...     def get_eggs(self):
...         print('getting eggs')
...         return self._eggs
...
...     def set_eggs(self, eggs):
...         print('setting eggs to %s' % eggs)
...         self._eggs = eggs
...
...     def delete_eggs(self):
...         print('deleting eggs')
...         del self._eggs
...
...     eggs = property(get_eggs, set_eggs, delete_eggs)
...
...     @property
...     def spam(self):
...         print('getting spam')
...         return self._spam
...
...     @spam.setter
...     def spam(self, spam):
...         print('setting spam to %s' % spam)
...         self._spam = spam
...
...     @spam.deleter
...     def spam(self):
...         print('deleting spam')
...         del self._spam
...
...     @functools.cached_property
...     def bacon(self):
...         print('getting bacon')
...         return 'bacon!'

>>> sandwich = Sandwich()

>>> sandwich.eggs = 123
setting eggs to 123

>>> sandwich.eggs
getting eggs
```

```
123
>>> del sandwich.eggs
deleting eggs
>>> sandwich.bacon
getting bacon
'bacon!'
>>> sandwich.bacon
'bacon!'
```

Similar to how we implemented the `classmethod` and `staticmethod` decorators, we need the Python descriptors again. This time, we require the full power of the descriptors, not just `__get__` but `__set__` and `__delete__` as well. For brevity, however, we will skip handling the documentation and some error handling:

```
>>> class Property(object):
...     def __init__(self, fget=None, fset=None, fdel=None):
...         self.fget = fget
...         self.fset = fset
...         self.fdel = fdel
...
...     def __get__(self, instance, cls):
...         if instance is None:
...             # Redirect class (not instance) properties to self
...             return self
...         elif self.fget:
...             return self.fget(instance)
...
...     def __set__(self, instance, value):
...         self.fset(instance, value)
...
...     def __delete__(self, instance):
...         self.fdel(instance)
...
...     def getter(self, fget):
...         return Property(fget, self.fset, self.fdel)
...
...     def setter(self, fset):
...         return Property(self.fget, fset, self.fdel)
...
...     def deleter(self, fdel):
...         return Property(self.fget, self.fset, fdel)
```

That doesn't look all that complicated, does it? The descriptors make up most of the code, which is fairly straight to the point. Only the `getter`/`setter`/`deleter` functions might look a bit strange, but they're actually fairly straightforward as well.

To make sure the `property` still works as expected, the class returns a new `Property` instance while copying the other methods. The only small caveat to make this work here is the `return self` in the `__get__` method.

```
>>> class Sandwich:
...     @Property
...     def eggs(self):
...         return self._eggs
...
...     @eggs.setter
...     def eggs(self, value):
...         self._eggs = value
...
...     @eggs.deleter
...     def eggs(self):
...         del self._eggs

>>> sandwich = Sandwich()
>>> sandwich.eggs = 5
>>> sandwich.eggs
5
```

As expected, our `Property` decorator works as it should. But note that this is a more limited version of the built-in `property` decorator; our version has no checking for edge cases.

Naturally, being Python, there are more methods of achieving the effect of properties. In the previous examples, you saw the bare descriptor implementation, and in our previous example, you saw the `property` decorator. Now we will look at a generic solution by implementing `__getattr__` or `__getattribute__`. Here's a simple demonstration:

```
>>> class Sandwich(object):
...     def __init__(self):
...         self.registry = {}
...
...     def __getattr__(self, key):
...         print('Getting %r' % key)
...         return self.registry.get(key, 'Undefined')
...
...     def __setattr__(self, key, value):
...         if key == 'registry':
...             object.__setattr__(self, key, value)
```

```
...         else:
...             print('Setting %r to %r' % (key, value))
...             self.registry[key] = value
...
...     def __delattr__(self, key):
...         print('Deleting %r' % key)
...         del self.registry[key]

>>> sandwich = Sandwich()

>>> sandwich.a
Getting 'a'
'Undefined'

>>> sandwich.a = 1
Setting 'a' to 1

>>> sandwich.a
Getting 'a'
1

>>> del sandwich.a
Deleting 'a'
```

The `__getattr__` method looks for existing attributes, for example, it checks whether the key exists in `instance.__dict__`, and is called only if it does not exist. That's why we never see a `__getattr__` for the registry attribute. The `__getattribute__` method is called in all cases, which makes it a bit more dangerous to use. With the `__getattribute__` method, you will need a specific exclusion for `registry` since it will be executed infinitely through recursion if you try to access `self.registry`.

There is rarely a need to look at descriptors, but they are used by several internal Python processes, such as the `super()` method when inheriting classes.

Now that you know how to create decorators for regular functions and class methods, let's continue by decorating entire classes.

Decorating classes

Python 2.6 introduced the class decorator syntax. As is the case with the function decorator syntax, this is not really a new technique either. Even without the syntax, a class can be decorated simply by executing `DecoratedClass = decorator(RegularClass)`. After the previous sections, you should be familiar with writing decorators. Class decorators are no different from regular ones, except for the fact that they take a class instead of a function. As is the case with functions, this happens at declaration time and *not* at instantiating/calling time.

Because there are quite a few alternative ways to modify how classes work, such as standard inheritance, mixins, and metaclasses (read more in *Chapter 8*, *Metaclasses – Making Classes (Not Instances) Smarter*), class decorators are never strictly needed. This does not reduce their usefulness, but it does offer an explanation of why you will most likely not see too many examples of class decorating in the wild.

Singletons — Classes with a single instance

Singletons are classes that always allow only a single instance to exist. So, instead of getting an instance specifically for your call, you always get the same one. This can be very useful for things such as a database connection pool, where you don't want to keep opening connections all of the time but want to reuse the original ones:

```
>>> import functools

>>> def singleton(cls):
...     instances = dict()
...     @functools.wraps(cls)
...     def _singleton(*args, **kwargs):
...         if cls not in instances:
...             instances[cls] = cls(*args, **kwargs)
...         return instances[cls]

...     return _singleton

>>> @singleton
... class SomeSingleton(object):
...     def __init__(self):
...         print('Executing init')

>>> a = SomeSingleton()
Executing init
>>> b = SomeSingleton()

>>> a is b
True

>>> a.x = 123
>>> b.x
123
```

As you can see in the `a is b` comparison, both objects have the same identity, so we can conclude that they are indeed the same object. As is the case with regular decorators, due to the `functools.wraps` functionality, we can still access the original class through `Spam.__wrapped__` if needed.

The `is` operator compares objects by identity, which is implemented as the memory address in CPython. If `a is b` returns `True`, we can conclude that both `a` and `b` are the same instance.

Total ordering – Making classes sortable

At some point or the other, you have probably needed to sort data structures. While this is easily achievable using the key parameter of the `sorted` function, there is a more convenient way if you need to do this often—by implementing the `__gt__`, `__ge__`, `__lt__`, `__le__`, and `__eq__` functions. That seems a bit verbose, doesn't it? If you want the best performance, it's still a good idea, but if you can take a tiny performance hit and some slightly more complicated stack traces, then `total_ordering` might be a nice alternative.

The `total_ordering` class decorator can implement all required sort functions based on a class that possesses an `__eq__` function and one of the comparison functions (`__lt__`, `__le__`, `__gt__`, or `__ge__`). This means you can seriously shorten your function definitions. Let's compare the regular function definition and the function definition using the `total_ordering` decorator:

```
>>> import functools

>>> class Value(object):
...     def __init__(self, value):
...         self.value = value
...
...     def __repr__(self):
...         return f'<{self.__class__.__name__} {self.value}>'

>>> class Spam(Value):
...     def __gt__(self, other):
...         return self.value > other.value
...
...     def __ge__(self, other):
...         return self.value >= other.value
...
...     def __lt__(self, other):
...         return self.value < other.value
...
...     def __le__(self, other):
...         return self.value <= other.value
...
...     def __eq__(self, other):
...         return self.value == other.value
```

```
>>> @functools.total_ordering
... class Egg(Value):
...     def __lt__(self, other):
...         return self.value < other.value
...
...     def __eq__(self, other):
...         return self.value == other.value
```

As you can see, without `functools.total_ordering`, it's quite a bit of work to create a fully sortable class. Now we will test whether they actually sort in a similar way:

```
>>> numbers = [4, 2, 3, 4]
>>> spams = [Spam(n) for n in numbers]
>>> eggs = [Egg(n) for n in numbers]

>>> spams
[<Spam 4>, <Spam 2>, <Spam 3>, <Spam 4>]

>>> eggs
[<Egg 4>, <Egg 2>, <Egg 3>, <Egg 4>]

>>> sorted(spams)
[<Spam 2>, <Spam 3>, <Spam 4>, <Spam 4>]

>>> sorted(eggs)
[<Egg 2>, <Egg 3>, <Egg 4>, <Egg 4>]

# Sorting using key is of course still possible and in this case
# perhaps just as easy:
>>> values = [Value(n) for n in numbers]
>>> values
[<Value 4>, <Value 2>, <Value 3>, <Value 4>]

>>> sorted(values, key=lambda v: v.value)
[<Value 2>, <Value 3>, <Value 4>, <Value 4>]
```

Now, you might be wondering, "Why isn't there a class decorator to make a class sortable using a specified key property?" Well, that might indeed be a good idea for the `functools` library, but it isn't there yet. So, let's see how we would implement something like it while still using `functools.total_ordering`:

```
>>> def sort_by_attribute(attr, keyfunc=getattr):
...     def _sort_by_attribute(cls):
```

```
...         def __lt__(self, other):
...             return getattr(self, attr) < getattr(other, attr)
...
...         def __eq__(self, other):
...             return getattr(self, attr) <= getattr(other, attr)
...
...         cls.__lt__ = __lt__
...         cls.__eq__ = __eq__
...
...         return functools.total_ordering(cls)
...
...     return _sort_by_attribute

>>> class Value(object):
...     def __init__(self, value):
...         self.value = value
...
...     def __repr__(self):
...         return f'<{self.__class__.__name__} {self.value}>'

>>> @sort_by_attribute('value')
... class Spam(Value):
...     pass

>>> numbers = [4, 2, 3, 4]
>>> spams = [Spam(n) for n in numbers]
>>> sorted(spams)
[<Spam 2>, <Spam 3>, <Spam 4>, <Spam 4>]
```

Certainly, this greatly simplifies the making of a sortable class. And if you would rather have your own key function instead of `getattr`, it's even easier. Simply replace the `getattr(self, attr)` call with `key_function(self)`, do that for `other` as well, and change the argument for the decorator to your function. You can even use that as the base function and implement `sort_by_attribute` by simply passing a wrapped `getattr` function.

Now that you know how to create all types of decorators, let's look at a few useful decorator examples bundled with Python.

Useful decorators

In addition to the ones already mentioned in this chapter, Python comes bundled with a few other useful decorators. There are some that aren't in the standard library (yet?).

Single dispatch — Polymorphism in Python

If you've used C++ or Java before, you're probably used to having ad hoc polymorphism available—different functions being called depending on the argument types. Python being a dynamically typed language, most people would not expect the possibility of a single dispatch pattern. Python, however, is a language that is not only dynamically typed but also strongly typed, which means we can rely on the type we receive.

A dynamically typed language does not require strict type definitions. While a language such as C would require the following to declare an integer:

```
int some_integer = 123;
```

Python simply accepts that our value has a type:

```
some_integer = 123
```

Although with type hinting we could also do:

```
some_integer: int = 123
```

As opposed to languages such as JavaScript and PHP, however, Python does very little implicit type conversion. In Python, the following will return an error, whereas JavaScript would execute it without any problems:

```
'spam' + 5
```

In Python, the result is a `TypeError`. In JavaScript, it's `'spam5'`.

The idea of single dispatch is that depending on the type you pass, the correct function is called. Since `str + int` results in an error in Python, this can be very convenient to automatically convert your arguments before passing them to your function. This can be useful for separating the actual workings of your function from the type conversions.

Since Python 3.4, there is a decorator that makes it easily possible to implement the single dispatch pattern in Python. This decorator is useful if you need to execute different functions depending on the `type()` of your input variable. Here is a basic example:

```
>>> import functools

>>> @functools.singledispatch
... def show_type(argument):
...     print(f'argument: {argument}')

>>> @show_type.register(int)
... def show_int(argument):
...     print(f'int argument: {argument}')
```

```
>>> @show_type.register
... def show_float(argument: float):
...     print(f'float argument: {argument}')

>>> show_type('abc')
argument: abc

>>> show_type(123)
int argument: 123

>>> show_type(1.23)
float argument: 1.23
```

The `singledispatch` decorator automatically calls the correct function for the type passed as the first argument. As you can see in the example, this works both when using type annotations and if explicit types are passed to the `register` function.

Let's see how we could make a simplified version of this method ourselves:

```
>>> import functools

>>> registry = dict()

>>> def register(function):
...     # Fetch the first type from the type annotation but be
...     # careful not to overwrite the 'type' function
...     type_ = next(iter(function.__annotations__.values()))
...     registry[type_] = function
...
...     @functools.wraps(function)
...     def _register(argument):
...         # Fetch the function using the type of argument, and
...         # fall back to the main function
...         new_function = registry.get(type(argument), function)
...         return new_function(argument)
...
...     return _register

>>> @register
... def show_type(argument: any):
...     print(f'argument: {argument}')
```

```
>>> @register
... def show_int(argument: int):
...     print(f'int argument: {argument}')

>>> show_type('abc')
argument: abc

>>> show_type(123)
int argument: 123
```

Naturally, this method is a bit basic and it uses a single global registry, which limits its application. But this exact pattern can be used for registering plugins or callbacks.

When naming the functions, make sure that you do not overwrite the original `singledispatch` function. If you named `show_int` as just `show_type`, it would overwrite the initial `show_type` function. This would make it impossible to access the original `show_type` function and make all `register` operations after that fail as well.

Now, a slightly more useful example—differentiating between a filename and a file handler:

```
>>> import json
>>> import functools

>>> @functools.singledispatch
... def write_as_json(file, data):
...     json.dump(data, file)

>>> @write_as_json.register(str)
... @write_as_json.register(bytes)
... def write_as_json_filename(file, data):
...     with open(file, 'w') as fh:
...         write_as_json(fh, data)

>>> data = dict(a=1, b=2, c=3)
>>> write_as_json('test1.json', data)
>>> write_as_json(b'test2.json', 'w')
>>> with open('test3.json', 'w') as fh:
...     write_as_json(fh, data)
```

So now we have a single `write_as_json` function; it calls the right code depending on the type. If it's a `str` or `bytes` object, it will automatically open the file and call the regular version of `write_as_json`, which accepts file objects.

Writing a decorator that does this is not that hard to do, of course, but it's still quite convenient to have the `singledispatch` decorator in the base library. It most certainly beats manually checking the given argument types with a list of `isinstance()` `if`/`elif`/`elif`/`else` statements.

To see which function will be called, you can use the `write_as_json.dispatch` function with a specific type. When passing along a `str`, you will get the `write_as_json_filename` function. It should be noted that the names of the dispatched functions are completely arbitrary. They are accessible as regular functions, of course, but you can name them anything you like.

To check the registered types, you can access the registry, which is a dictionary, through `write_as_json.registry`:

```
>>> write_as_json.registry.keys()
dict_keys([<class 'bytes'>, <class 'object'>, <class 'str'>])
```

contextmanager — with statements made easy

Using the `contextmanager` class, we can make the creation of a context wrapper very easy. Context wrappers are used whenever you use a `with` statement. One example is the `open` function, which works as a context wrapper as well, allowing you to use the following code:

```
with open(filename) as fh:
    pass
```

Let's just assume for now that the `open` function is not usable as a context manager and that we need to build our own function to do this. The standard method of creating a context manager is by creating a class that implements the `__enter__` and `__exit__` methods:

```
>>> class Open:
...     def __init__(self, filename, mode):
...         self.filename = filename
...         self.mode = mode
...
...     def __enter__(self):
...         self.handle = open(self.filename, self.mode)
...         return self.handle
...
...     def __exit__(self, exc_type, exc_val, exc_tb):
...         self.handle.close()

>>> with Open('test.txt', 'w') as fh:
...     print('Our test is complete!', file=fh)
```

While that works perfectly, it's a tad verbose. With `contextlib.contextmanager`, we can have the same behavior in just a few lines:

```
>>> import contextlib

>>> @contextlib.contextmanager
... def open_context_manager(filename, mode='r'):
...     fh = open(filename, mode)
...     yield fh
...     fh.close()

>>> with open_context_manager('test.txt', 'w') as fh:
...     print('Our test is complete!', file=fh)
```

Simple, right? However, I should mention that for this specific case—the closing of objects—there is a dedicated function in `contextlib`, and it is even easier to use.

With `file` objects, database connections, and connections, it is important to always have a `close()` call to clean up resources. In the case of a `file`, it tells the operating system to write the data to disk (as opposed to temporary buffers), and in the case of network connections and database connections, it releases the network connection and related resources on both ends. With database connections, it will also notify the server that the connection is no longer needed so that part is also handled gracefully.

Without these calls, you can quickly run into "too many open files" or "too many connections" errors.

Let's demonstrate it with the most basic example of when `closing()` would be useful:

```
>>> import contextlib

>>> with contextlib.closing(open('test.txt', 'a')) as fh:
...     print('Yet another test', file=fh)
```

For a `file` object, you can usually also use `with open(...)` because it is a context manager by itself, but if some other part of the code handles the opening, you don't always have that luxury, and in those cases, you will need to close it yourself. Additionally, some objects such as requests made by `urllib` don't support automatic closing in that manner and benefit from this function.

But wait; there's more! In addition to being usable in a `with` statement, the results of a `contextmanager` are actually usable as decorators since Python 3.2. In older Python versions, the `contextmanager` was simply a small wrapper, but since Python 3.2 it's based on the `ContextDecorator` class, which makes it a decorator.

The `open_context_manager` context manager isn't really suitable as a decorator since it has a `yield <value>` as opposed to an empty `yield` (more about that in *Chapter 7*, *Generators and Coroutines – Infinity, One Step at a Time*), but we can think of other functions:

```
>>> @contextlib.contextmanager
... def debug(name):
...     print(f'Debugging {name}:')
...     yield
...     print(f'Finished debugging {name}')

>>> @debug('spam')
... def spam():
...     print('This is the inside of our spam function')

>>> spam()
Debugging spam:
This is the inside of our spam function
Finished debugging spam
```

There are quite a few nice use cases for this, but at the very least, it's just a convenient way to wrap a function in a context without all the (nested) `with` statements.

Validation, type checks, and conversions

While checking for types is usually not the best way to go in Python, at times it can be useful if you know that you will need a specific type (or something that can be cast to that type). To facilitate this, Python 3.5 introduced a type hinting system so that you can do the following:

```
>>> def sandwich(bacon: float, eggs: int):
...     pass
```

In some cases, it can be useful to change these hints into requirements. Instead of using an `isinstance()`, we will simply try to enforce the types by casting, which is more along the lines of duck-typing.

The essence of duck-typing is: if it looks like a duck, walks like a duck, and quacks like a duck, it might be a duck. Essentially, this means that we don't care if the value is a `duck` or something else, only if it supports the `quack()` method that we need.

To enforce the type hints, we can create a decorator:

```
>>> import inspect
>>> import functools

>>> def enforce_type_hints(function):
...     # Construct the signature from the function which contains
...     # the type annotations
```

```
...     signature = inspect.signature(function)
...
...     @functools.wraps(function)
...     def _enforce_type_hints(*args, **kwargs):
...         # Bind the arguments and apply the default values
...         bound = signature.bind(*args, **kwargs)
...         bound.apply_defaults()
...
...         for key, value in bound.arguments.items():
...             param = signature.parameters[key]
...             # The annotation should be a callable
...             # type/function so we can cast as validation
...             if param.annotation:
...                 bound.arguments[key] = param.annotation(value)
...
...         return function(*bound.args, **bound.kwargs)
...
...     return _enforce_type_hints

>>> @enforce_type_hints
... def sandwich(bacon: float, eggs: int):
...     print(f'bacon: {bacon!r}, eggs: {eggs!r}')

>>> sandwich(1, 2)
bacon: 1.0, eggs: 2
>>> sandwich(3, 'abc')
Traceback (most recent call last):
...
ValueError: invalid literal for int() with base 10: 'abc'
```

This is a fairly simple yet very versatile type enforcer that should work with most type annotations.

Useless warnings — How to ignore them safely

When writing in Python, warnings are often very useful when you're actually writing the code. When executing it, however, it is not useful to get that same message every time you run your script/application. So, let's create some code that allows easy hiding of the expected warnings, but not all of them so that we can easily catch new ones:

```
>>> import warnings
>>> import functools

>>> def ignore_warning(warning, count=None):
...     def _ignore_warning(function):
```

```
...         @functools.wraps(function)
...         def __ignore_warning(*args, **kwargs):
...             # Execute the code while catching all warnings
...             with warnings.catch_warnings(record=True) as ws:
...                 # Catch all warnings of the given type
...                 warnings.simplefilter('always', warning)
...                 # Execute the function
...                 result = function(*args, **kwargs)
...
...             # Re-warn all warnings beyond the expected count
...             if count is not None:
...                 for w in ws[count:]:
...                     warnings.warn(w.message)
...
...             return result
...
...         return __ignore_warning
...
...     return _ignore_warning

>>> @ignore_warning(DeprecationWarning, count=1)
... def spam():
...     warnings.warn('deprecation 1', DeprecationWarning)
...     warnings.warn('deprecation 2', DeprecationWarning)

# Note, we use catch_warnings here because doctests normally
# capture the warnings quietly
>>> with warnings.catch_warnings(record=True) as ws:
...     spam()
...
...     for i, w in enumerate(ws):
...         print(w.message)
deprecation 2
```

Using this method, we can catch the first (expected) warning and still see the second (unexpected) warning.

Now that you have seen some examples of useful decorators, it is time to continue with a few exercises and see how much you can write yourself.

Exercises

Decorators have a huge range of uses, so you can probably think of some yourself after reading this chapter, but you can easily elaborate on some of the decorators we wrote earlier:

- Extend the `track` function to monitor execution time.
- Extend the `track` function with min/max/average execution time and call count.
- Modify the memoization function to function with unhashable types.
- Modify the memoization function to have a cache per function instead of a global one.
- Create a version of `functools.cached_property` that can be recalculated as needed.
- Create a single-dispatch decorator that considers all or a configurable number of arguments instead of only the first one.
- Enhance the `type_check` decorator to include additional checks such as requiring a number to be greater than or less than a given value.

Example answers for these exercises can be found on GitHub: `https://github.com/mastering-python/exercises`. You are encouraged to submit your own solutions and learn about alternative solutions from others.

Summary

This chapter showed you some of the places where decorators can be used to make our code simpler and add some fairly complex behavior to very simple functions. Truthfully, most decorators are more complex than the regular function would have been by simply adding the functionality directly, but the added advantage of applying the same pattern to many functions and classes is generally well worth it.

Decorators have so many uses to make your functions and classes smarter and more convenient to use:

- Debugging
- Validation
- Argument convenience (pre-filling or converting arguments)
- Output convenience (converting the output to a specific type)

The most important takeaway of this chapter should be to never forget `functools.wraps` when wrapping a function. Debugging decorated functions can be rather difficult because of (unexpected) behavior modification, but losing attributes as well can make that problem much worse.

The next chapter will show you how and when to use `generators` and `coroutines`. This chapter has already shown you the usage of the `with` statement briefly, but `generators` and `coroutines` go much further with this. We will still be using decorators often, both in this book and when using Python in general, so make sure you have a good understanding of how they work.

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:
`https://discord.gg/QMzJenHuJf`

7

Generators and Coroutines – Infinity, One Step at a Time

Generator functions are functions that behave like iterators by generating the return values one by one. While traditional methods build and return a `list` or `tuple` of items with a fixed length, a generator will `yield` a single value only when requested by the caller. The side effect is that these generators can be infinitely large because you can keep yielding forever.

In addition to generators, there is a variation to the generator's syntax that creates coroutines. Coroutines are functions that allow multitasking without requiring multiple threads or processes. Whereas generators can only yield values to the caller based on the initial arguments, coroutines enable two-way communication with the calling function while running. The modern implementation of coroutines in Python is through the `asyncio` module, which is covered extensively in *Chapter 13, asyncio – Multithreading without Threads*, but the basics stem from the coroutines discussed in this chapter. If coroutines or `asyncio` work for your case, they can offer a tremendous performance improvement.

In this chapter, we will cover the following topics:

- Advantages and disadvantages of generators
- The characteristics and quirks of generators
- Creating generators using regular functions
- Generator comprehensions similar to `list`, `dict`, and `set` comprehensions
- Creating generators using classes
- Generators bundled with Python
- A basic implementation of coroutines and a few of their quirks

Generators

Generators are a very useful tool but they come with a set of rules to keep in mind.

First, let's explore the advantages of generators:

- Generators are often simpler to write than list-generating functions. Instead of having to declare a `list`, `list.append(value)`, and `return`, you only need `yield value`.
- Memory usage. Items can be processed one at a time, so there is generally no need to keep the entire list in memory.
- Results can depend on outside factors. Instead of having a static list, you generate the value when it is being requested. Think of processing a queue/stack, for example.
- Generators are lazy. This means that if you're using only the first five results of a generator, the rest won't even be calculated. Additionally, between fetching the items, the generator is completely frozen.

The most important disadvantages are:

- Results are available only once. After processing the results of a generator, it cannot be used again.
- The size is unknown. Until you are done processing, you cannot get any information about the size of the generator. It might even be infinite. This makes `list(some_infinite_generator)` a dangerous operation. It can quickly crash your Python interpreter or even your entire system.
- Slicing is not possible, so `some_generator[10:20]` will not work. You can work around this using `itertools.islice` as you will see later in this chapter, but that effectively discards the unused indices.
- Indexing generators, similar to slicing, is also not possible. This means that the following will not work: `some_generator[5]`.

Now that you know what to expect, let's create a few generators.

Creating generators

The simplest generator is a function containing a `yield` statement instead of a `return` statement. The key difference with regular functions containing a `return` is that you can have many `yield` statements in your function.

An example of a generator with a few fixed `yield` statements and how it behaves with several operations is as follows:

```
>>> def generator():
...     yield 1
...     yield 'a'
...     yield []
...     return 'end'

>>> result = generator()

>>> result
<generator object generator at ...>
```

```
>>> len(result)
Traceback (most recent call last):
    ...
TypeError: object of type 'generator' has no len()

>>> result[:10]
Traceback (most recent call last):
    ...
TypeError: 'generator' object is not subscriptable

>>> list(result)
[1, 'a', []]

>>> list(result)
[]
```

A few of the downsides of generators become immediately apparent in this example. The `result` does not offer much meaningful information when looking at its `repr()`, getting `len()` (length), or slicing. And trying to do `list()` to get the values a second time does not work because the generator is already exhausted.

Additionally, you may have noticed that the `return` value of the function appears to have completely disappeared. This is actually not the case; the value of `return` is still used, but as the value for the `StopIteration` exception raised by the generator to indicate that the generator has been exhausted:

```
>>> def generator_with_return():
...     yield 'some_value'
...     return 'The end of our generator'

>>> result = generator_with_return()

>>> next(result)
'some_value'
>>> next(result)
Traceback (most recent call last):
    ...
StopIteration: The end of our generator
```

The following example demonstrates the lazy execution of generators:

```
>>> def lazy():
...     print('before the yield')
...     yield 'yielding'
```

```
...     print('after the yield')

>>> generator = lazy()

>>> next(generator)
before the yield
'yielding'

>>> next(generator)
Traceback (most recent call last):
    ...
StopIteration
```

As you can see in this example, the code after the `yield` isn't executed. This is caused by the `StopIteration` exception; if we properly catch this exception, the code will be executed:

```
>>> def lazy():
...     print('before the yield')
...     yield 'yielding'
...     print('after the yield')

>>> generator = lazy()

>>> next(generator)
before the yield
'yielding'

>>> try:
...     next(generator)
... except StopIteration:
...     pass
after the yield

>>> for item in lazy():
...     print(item)
before the yield
yielding
after the yield
```

To properly handle generators, you always need to either catch the `StopIteration` yourself, or use a loop or another structure that handles the `StopIteration` implicitly.

Creating infinite generators

Creating an endless generator (such as the `itertools.count` iterator discussed in *Chapter 5*, *Functional Programming – Readability Versus Brevity*) is easy as well. If, instead of having the fixed `yield <value>` lines like in the previous function, we `yield` from inside of an infinite loop, we can easily make an infinite generator.

As opposed to the `itertools.count()` generator, we will add a `stop` parameter to make testing easier:

```
>>> def count(start=0, step=1, stop=None):
...     n = start
...     while stop is not None and n < stop:
...         yield n
...         n += step

>>> list(count(10, 2.5, 20))
[10, 12.5, 15.0, 17.5]
```

Due to the potentially infinite nature of generators, caution is required. Without the `stop` variable, simply doing `list(count())` would result in an infinite loop that results in an out-of-memory situation quite fast.

So, how does this work? Essentially it is just a normal loop, but the big difference between this and the regular method of returning a list of items is that the `yield` statement returns the items one at a time, which means you only have to calculate the requested items and you don't have to keep all results in memory.

Generators wrapping iterables

While generators are already quite useful when generating values from scratch, the real power comes when wrapping other iterables. To illustrate this, we will create a generator that automatically squares all numbers from the given input:

```
>>> def square(iterable):
...     for i in iterable:
...         yield i ** 2

>>> list(square(range(5)))
[0, 1, 4, 9, 16]
```

Naturally, there is nothing stopping you from adding extra `yield` statements outside of the loop:

```
>>> def padded_square(iterable):
...     yield 'begin'
...     for i in iterable:
```

```
...         yield i ** 2
...     yield 'end'

>>> list(padded_square(range(5)))
['begin', 0, 1, 4, 9, 16, 'end']
```

Because these generators are iterable, you can chain them together by wrapping them as many times as you like. A basic example of chaining a `square()` and an `odd()` generator together is:

```
>>> import itertools

>>> def odd(iterable):
...     for i in iterable:
...         if i % 2:
...             yield i

>>> def square(iterable):
...     for i in iterable:
...         yield i ** 2

>>> list(square(odd(range(10))))
[1, 9, 25, 49, 81]
```

If we analyze how the code is executed, we need to start from the inside to the outside:

1. The `range(10)` statement generates 10 numbers for us.
2. The `odd()` generator filters the input values, so from the `[0, 1, 2 … ]` values it only returns `[1, 3, 5, 7, 9]`.
3. The `square()` function squares the given input, which is the list of odd numbers as generated by `odd()`.

The real power of chaining is that the generators will only do something when we request a value. If we request a single value with `next()` instead of `list()`, it will mean that only the first iteration of the loop in `square()` will be run. For `odd()` and `range()`, however, it will have to process two values because `odd()` will discard the first value given by `range()` and not `yield` anything.

Generator comprehensions

In the previous chapters, you saw `list`, `dict`, and `set` comprehensions, which generate collections. With a generator comprehension we can make similar collections, but make them lazy so they are only evaluated as needed. The basic premise is identical to the `list` comprehension but using round brackets/parentheses instead of square brackets:

```
>>> squares = (x ** 2 for x in range(4))

>>> squares
```

```
<generator object <genexpr> at 0x...>

>>> list(squares)
[0, 1, 4, 9]
```

This is very useful when you need to wrap the results of a different generator because it only calculates the values you asked for:

```
>>> import itertools

>>> result = itertools.count()
>>> odd = (x for x in result if x % 2)
>>> sliced_odd = itertools.islice(odd, 5)
>>> list(sliced_odd)
[1, 3, 5, 7, 9]

>>> result = itertools.count()
>>> sliced_result = itertools.islice(result, 5)
>>> odd = (x for x in sliced_result if x % 2)
>>> list(odd)
[1, 3]
```

As you can probably surmise from this result, this can be dangerous with infinite-sized generators such as `itertools.count()`. The order of operations is very important because the `itertools.islice()` function slices the result at that point, not the original generator. This means that if we replace `odd()` with a function that never evaluates to `True` for the given collection, it will run forever because it will never `yield` any results.

Class-based generators and iterators

In addition to creating generators as regular functions and through generator comprehensions, we can also create generators using classes. This can be beneficial for more complex generators where you need to remember the state or where inheritance can be used.

First, let's look at an example of creating a basic generator `class` that mimics the behavior of `itertools.count()` with an added `stop` parameter:

```
>>> class CountGenerator:
...     def __init__(self, start=0, step=1, stop=None):
...         self.start = start
...         self.step = step
...         self.stop = stop
...
...     def __iter__(self):
```

```
...         i = self.start
...         while self.stop is None or i < self.stop:
...             yield i
...             i += self.step

>>> list(CountGenerator(start=2.5, step=0.5, stop=5))
[2.5, 3.0, 3.5, 4.0, 4.5]
```

Now let's convert the generator class into an iterator with more features:

```
>>> class CountIterator:
...     def __init__(self, start=0, step=1, stop=None):
...         self.i = start
...         self.start = start
...         self.step = step
...         self.stop = stop
...
...     def __iter__(self):
...         return self
...
...     def __next__(self):
...         if self.stop is not None and self.i >= self.stop:
...             raise StopIteration
...
...         # We need to return the value before we increment to
...         # maintain identical behavior
...         value = self.i
...         self.i += self.step
...         return value

>>> list(CountIterator(start=2.5, step=0.5, stop=5))
[2.5, 3.0, 3.5, 4.0, 4.5]
```

The most important distinction between the generator and the iterator is that instead of a simple iterable object, we now have a fully fledged class that acts as an iterator, which means we can also expand it beyond the capabilities of regular generators.

A few of the limitations of regular generators are that they don't have a length and we cannot slice them. With an iterator, we can explicitly define the behavior in these scenarios if needed:

```
>>> import itertools

>>> class AdvancedCountIterator:
...     def __init__(self, start=0, step=1, stop=None):
```

```
...         self.i = start
...         self.start = start
...         self.step = step
...         self.stop = stop
...
...     def __iter__(self):
...         return self
...
...     def __next__(self):
...         if self.stop is not None and self.i >= self.stop:
...             raise StopIteration
...
...         value = self.i
...         self.i += self.step
...         return value
...
...     def __len__(self):
...         return int((self.stop - self.start) // self.step)
...
...     def __contains__(self, key):
...         # To check 'if 123 in count'.
...         # Note that this does not look at 'step'!
...         return self.start < key < self.stop
...
...     def __repr__(self):
...         return (
...             f'{self.__class__.__name__}(start={self.start}, '
...             f'step={self.step}, stop={self.stop})')
...
...     def __getitem__(self, slice_):
...         return itertools.islice(self, slice_.start,
...                                 slice_.stop, slice_.step)
```

Now that we have our advanced count iterator with support for features such as `len()`, `in`, and `repr()`, we can test to see if it works as expected:

```
>>> count = AdvancedCountIterator(start=2.5, step=0.5, stop=5)

# Pretty representation using '__repr__'
>>> count
AdvancedCountIterator(start=2.5, step=0.5, stop=5)

# Check if item exists using '__contains__'
```

```
>>> 3 in count
True
>>> 3.1 in count
True
>>> 1 in count
False

# Getting the length using '__len__'
>>> len(count)
5

# Slicing using '__getitem__' with a slice as a parameter
>>> count[:3]
<itertools.islice object at 0x...>

>>> list(count[:3])
[2.5, 3.0, 3.5]

>>> list(count[:3])
[4.0, 4.5]
```

In addition to working around some of the limitations, in the last example, you can also see a very useful feature of generators. We can exhaust the items one by one and stop/start whenever we want. And since we still have full access to the object, we could alter `count.i` to restart the iterator.

Generator examples

Now that you know how generators can be created, let's look at a few useful generators and examples of how to use them.

Before you start writing a generator for your project, always make sure to look at the Python `itertools` module. It features a host of useful generators that cover a vast array of use cases. The following sections show some custom generators and a few of the most useful generators in the standard library.

These generators work on all iterables, not just generators. So, you could also apply them to a `list`, `tuple`, `string`, or other kinds of iterables.

Breaking an iterable up into chunks/groups

When executing large amounts of queries in a database or when running tasks via multiple processes, it is often more efficient to chunk the operations. Having a single huge operation could result in out-of-memory issues; having many tiny operations can be slow due to start-up/teardown sequences.

To make things more efficient, a good method is to split the input into chunks. The Python documentation (`https://docs.python.org/3/library/itertools.html?highlight=chunk#itertools-recipes`) already comes with an example of how to do this by using `itertools.zip_longest()`:

```
>>> import itertools

>>> def grouper(iterable, n, fillvalue=None):
...     '''Collect data into fixed-length chunks or blocks'''
...     args = [iter(iterable)] * n
...     return itertools.zip_longest(*args, fillvalue=fillvalue)

>>> list(grouper('ABCDEFG', 3, 'x'))
[('A', 'B', 'C'), ('D', 'E', 'F'), ('G', 'x', 'x')]
```

This code is a very nice example of how easy it is to chunk your data, but it has to hold the entire chunk in memory. To work around that, we can create a version that generates sub-generators for the chunks:

```
>>> def chunker(iterable, chunk_size):
...     # Make sure 'iterable' is an iterator
...     iterable = iter(iterable)
...
...     def chunk(value):
...         # Make sure not to skip the given value
...         yield value
...         # We already yielded a value so reduce the chunk_size
...         for _ in range(chunk_size - 1):
...             try:
...                 yield next(iterable)
...             except StopIteration:
...                 break
...
...     while True:
...         try:
...             # Check if we're at the end by using 'next()'
...             yield chunk(next(iterable))
...         except StopIteration:
...             break

>>> for chunk in chunker('ABCDEFG', 3):
...     for value in chunk:
...         print(value, end=', ')
...     print()
A, B, C,
```

```
D, E, F,
G,
```

Because we need to catch the `StopIteration` exceptions, this example does not look very pretty in my opinion. Part of the code could be improved by using `itertools.islice()` (which is covered next) but that will still leave us with the problem that we cannot know when we have reached the end.

If you are interested, an implementation using `itertools.islice()` and `itertools.chains()` can be found on this book's GitHub: `https://github.com/mastering-python/code_2`.

itertools.islice – Slicing iterables

One limitation of generators is that they cannot be sliced. You can work around this by converting the generator into a `list` before slicing, but that is not possible with infinite generators, and it can be inefficient if you only need a few values.

To solve this, the `itertools` library has an `islice()` function, which can slice any iterable object. The function is the generator version of the slicing operators and similarly to slicing supports a `start`, `stop`, and `step` parameter. The following illustrates how regular slicing and `itertools.islice()` compare:

```
>>> import itertools

>>> some_list = list(range(1000))
>>> some_list[:5]
[0, 1, 2, 3, 4]
>>> list(itertools.islice(some_list, 5))
[0, 1, 2, 3, 4]

>>> some_list[10:20:2]
[10, 12, 14, 16, 18]
>>> list(itertools.islice(some_list, 10, 20, 2))
[10, 12, 14, 16, 18]
```

It is very important to note that while the output is identical, these methods are far from equivalent internally. Regular slicing only works on objects that are sliceable; effectively, this means the object has to implement the `__getitem__(self, slice)` method.

Additionally, we expect that slicing objects is a fast and efficient operation. For `list` and `tuple` this is certainly the case, but for a given generator this might not be the case.

If for a list with size `n=1000` we take any slice of any `k=10` elements, we can expect the time complexity of that to be only `O(k)`; that is, 10 steps. It doesn't matter whether we do `some_list[:10]` or `some_list[900:920:2]`.

For `itertools.islice()` this is not the case because the only assumption it makes is that the input is iterable. That means that getting the first 10 items is easy; simply loop through the items, return the first 10, and stop. So `itertools.islice(some_list, 10)` also takes 10 steps. Getting items 900 to 920, however, means walking through and discarding the first 900 items, and only returning 10 of the next 20 items. So that is 920 steps instead.

To illustrate this, here's a slightly simplified implementation of `itertools.islice()` that expects to always have a `stop` available:

```
>>> def islice(iterable, start, stop=None, step=1):
...     # 'islice' has signatures: 'islice(iterable, stop)' and:
...     # 'islice(iterable, start, stop[, step])'
...     # 'fill' stop with 'start' if needed
...     if stop is None and step == 1 and start is not None:
...         start, stop = 0, start
...
...     # Create an iterator and discard the first 'start' items
...     iterator = iter(iterable)
...     for _ in range(start):
...         next(iterator)
...
...     # Enumerate the iterator making 'i' start at 'start'
...     for i, item in enumerate(iterator, start):
...         # Stop when we've reached 'stop' items
...         if i >= stop:
...             return
...         # Use modulo 'step' to discard non-matching items
...         if i % step:
...             continue
...         yield item

>>> list(islice(range(1000), 10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> list(islice(range(1000), 900, 920, 2))
[900, 902, 904, 906, 908, 910, 912, 914, 916, 918]

>>> list(islice(range(1000), 900, 910))
[900, 901, 902, 903, 904, 905, 906, 907, 908, 909]
```

As you can see, both the `start` and the `step` sections discard items that are not needed. This does not mean you should not use `itertools.islice()`, but be wary of the internals. Also, as you might expect, this generator does not support negative values for the indices and expects all values to be positive.

itertools.chain — Concatenating multiple iterables

The `itertools.chain()` generator is one of the simplest yet one of the most useful generators in the Python library. It simply returns every item from every passed iterable in sequential order and can be implemented in just three lines:

```
>>> def chain(*iterables):
...     for iterable in iterables:
...         yield from iterable

>>> a = 1, 2, 3
>>> b = [4, 5, 6]
>>> c = 'abc'
>>> list(chain(a, b, c))
[1, 2, 3, 4, 5, 6, 'a', 'b', 'c']

>>> a + b + c
Traceback (most recent call last):
    ...
TypeError: can only concatenate tuple (not "list") to tuple
```

As you might notice, this also introduces a feature not yet discussed: the `yield from` expression. `yield from` does exactly what you can expect from the name and yields all items from the given iterable. So `itertools.chain()` can also be replaced with the slightly more verbose:

```
>>> def chain(*iterables):
...     for iterable in iterables:
...         for i in iterable:
...             yield i
```

Interestingly, this method is more powerful than adding the collections because it doesn't care about the types as long as they are iterable—duck typing at its finest.

itertools.tee — Using an output multiple times

As mentioned before, one of the biggest disadvantages of generators is that the results are usable only once. Luckily, Python has a function that allows you to copy the output to several generators. The name `tee` might be familiar to you if you are used to working in a Linux/Unix command-line shell. The `tee` program allows you to write outputs to both the screen and a file, so you can store an output while still maintaining a live view of it.

The Python version, `itertools.tee()`, does a similar thing except that it returns several iterators, allowing you to process the results separately.

By default, `tee` will split your generator into a tuple containing two different generators, which is why tuple unpacking works nicely here. By passing along the `n` parameter, you can tell `itertools.tee()` to create more than two generators. Here is an example:

```
>>> import itertools

>>> def spam_and_eggs():
...     yield 'spam'
...     yield 'eggs'

>>> a, b = itertools.tee(spam_and_eggs())
>>> next(a)
'spam'
>>> next(a)
'eggs'
>>> next(b)
'spam'
>>> next(b)
'eggs'
>>> next(b)
Traceback (most recent call last):
    ...
StopIteration
```

After seeing this code, you might be wondering about the memory usage of `tee`. Does it need to store the entire list for you? Luckily, no. The `tee` function is pretty smart in handling this. Assume you have a generator that contains 1,000 items, and you read the first 100 items from `a` and the first `75` items from `b` simultaneously. Then `tee` will only keep the difference (`100 - 75 = 25` items) in memory and drop the rest while you are iterating the results.

Whether `tee` is the best solution in your case or not depends, of course. If instance `a` is read from the beginning to (nearly) the end before instance `b` is read, then it would not be a great idea to use `tee`. Simply converting the generator into a `list` would be faster since it involves much fewer operations.

contextlib.contextmanager — Creating context managers

You have already seen context managers in *Chapter 5, Functional Programming – Readability Versus Brevity*, and *Chapter 6, Decorators – Enabling Code Reuse by Decorating*, but there are many more useful things to be done with context managers. While the `contextlib.contextmanager()` generator is not meant to be a result-generating generator like the examples you saw earlier in this chapter, it does use `yield`, so it's a nice example of non-standard generator usage.

Some useful examples to log your output to a file and measure function execution time are:

```
>>> import time
>>> import datetime
>>> import contextlib

# Context manager that shows how long a context was active
```

```
>>> @contextlib.contextmanager
... def timer(name):
...     start_time = datetime.datetime.now()
...     yield
...     stop_time = datetime.datetime.now()
...     print('%s took %s' % (name, stop_time - start_time))

>>> with timer('basic timer'):
...     time.sleep(0.1)
basic timer took 0:00:00.1...

# Write standard print output to a file temporarily
>>> @contextlib.contextmanager
... def write_to_log(name):
...     with open(f'{name}.txt', 'w') as fh:
...         with contextlib.redirect_stdout(fh):
...             with timer(name):
...                 yield

# Using as a decorator also works in addition to with-statements
>>> @write_to_log('some_name')
... def some_function():
...     print('This will be written to 'some_name.txt'')

>>> some_function()
```

This all works perfectly, but the code could be prettier. Having three levels of context managers tends to get a bit unreadable, which is something you could generally solve using decorators, as covered in *Chapter 6*. In this case, however, we need the output from one context manager as the input for the next, which would make for a more complicated decorator setup.

That's where the `ExitStack` context manager comes in. It allows the easy combining of multiple context managers without increasing the indentation level:

```
>>> import contextlib

>>> @contextlib.contextmanager
... def write_to_log(name):
...     with contextlib.ExitStack() as stack:
...         fh = stack.enter_context(open(f'{name}.txt', 'w'))
...         stack.enter_context(contextlib.redirect_stdout(fh))
```

```
...         stack.enter_context(timer(name))
...         yield

>>> @write_to_log('some_name')
... def some_function():
...     print('This will be written to 'some_name.txt'')

>>> some_function()
```

Looks a bit simpler, doesn't it? While this example is still reasonably legible without the `ExitStack` context manager, the convenience of `ExitStack` becomes quickly apparent when you need to do specific teardowns. In addition to the automatic handling, as seen before, it's also possible to transfer the contexts to a new `ExitStack` to manually handle the closing:

```
>>> import contextlib

>>> with contextlib.ExitStack() as stack:
...     fh = stack.enter_context(open('file.txt', 'w'))
...     # Move the context(s) to a new ExitStack
...     new_stack = stack.pop_all()

>>> bytes_written = fh.write('fh is still open')

# After closing we can't write anymore
>>> new_stack.close()
>>> fh.write('cant write anymore')
Traceback (most recent call last):
    ...
ValueError: I/O operation on closed file.
```

Most of the `contextlib` functions have extensive documentation available in the Python manual. `ExitStack` in particular is documented using many examples at `https://docs.python.org/3/library/contextlib.html#contextlib.ExitStack`. I recommend keeping an eye on the `contextlib` documentation as it is improving greatly with every Python version.

Now that we have covered regular generators, it is time to continue with coroutines.

Coroutines

Coroutines are subroutines that offer non-pre-emptive multitasking through multiple entry points. The basic premise is that coroutines allow two functions to communicate with each other while running within a single thread. Normally, this type of communication is reserved only for multitasking or multithreading solutions, but coroutines offer a relatively simple way of achieving this at almost no added performance cost.

Since generators are lazy by default, you might be able to guess how coroutines function. Until a result is consumed, the generator sleeps; but while consuming a result, the generator becomes active. The difference between regular generators and coroutines is that with coroutines the communication goes both ways; the coroutine can receive values as well as `yield` them to the calling function.

If you are familiar with `asyncio` you might notice a strong similarity between `asyncio` and coroutines. That is because `asyncio` is built on the idea of coroutines and has evolved from a little bit of syntactic sugar into a whole ecosystem. For practical purposes I would suggest using `asyncio` instead of the coroutine syntax explained here; for educational purposes, however, it is very useful to understand how they work. The `asyncio` module is under very active development and has a much less awkward syntax.

A basic example

In the previous sections, you saw how regular generators can `yield` values. But generators can do more; they can actually receive values through `yield` as well. The basic usage is fairly simple:

```
>>> def generator():
...     value = yield 'value from generator'
...     print('Generator received:', value)
...     yield f'Previous value: {value!r}'

>>> g = generator()
>>> print('Result from generator:', next(g))
Result from generator: value from generator

>>> print(g.send('value from caller'))
Generator received: value from caller
Previous value: 'value from caller'
```

And that's all there is to it. The function is frozen until the `send` method is called, at which point it will process up to the next `yield` statement. One limitation you can see from this is that the coroutine can't wake up by itself. The value exchanges can only happen when the calling code runs `next(generator)` or `generator.send()`.

Priming

Since generators are lazy, you can't just send a value to a brand-new generator. Before a value can be sent to the generator, either a result must be fetched using `next()` or a `send(None)` has to be issued so that the code is actually reached. This is understandable, but a bit tedious at times. Let's create a simple decorator to omit the need for this:

```
>>> import functools

>>> def coroutine(function):
...     # Copy the 'function' description with 'functools.wraps'
```

```
...     @functools.wraps(function)
...     def _coroutine(*args, **kwargs):
...         active_coroutine = function(*args, **kwargs)
...         # Prime the coroutine and make sure we get no values
...         assert not next(active_coroutine)
...         return active_coroutine
...
...     return _coroutine

>>> @coroutine
... def our_coroutine():
...     while True:
...         print('Waiting for yield...')
...         value = yield
...         print('our coroutine received:', value)

>>> generator = our_coroutine()
Waiting for yield...

>>> generator.send('a')
our coroutine received: a
Waiting for yield...
```

As you've probably noticed, even though the generator is still lazy, it now automatically executes all of the code until it reaches the `yield` statement again. At that point, it will stay dormant until new values are sent.

Note that the `coroutine` decorator will be used throughout this chapter from this point onward. For brevity, the coroutine function definition will be omitted from the following examples.

Closing and throwing exceptions

Unlike regular generators, which simply exit as soon as the input sequence is exhausted, coroutines generally employ infinite `while` loops, which means that they won't be torn down the normal way. That's why coroutines also support both the `close` and `throw` methods, which will exit the function. The important thing here is not the closing but the possibility of adding a teardown method. Essentially, it is very comparable to how context wrappers function with an `__enter__` and `__exit__` method, but with coroutines in this case.

The following example shows a coroutine with normal and exception exit cases using the `coroutine` decorator from the previous paragraph:

```
>>> from coroutine_decorator import coroutine

>>> @coroutine
... def simple_coroutine():
...     print('Setting up the coroutine')
...     try:
...         while True:
...             item = yield
...             print('Got item:', item)
...     except GeneratorExit:
...         print('Normal exit')
...     except Exception as e:
...         print('Exception exit:', e)
...         raise
...     finally:
...         print('Any exit')
```

This `simple_coroutine()` function can show us some of the internal flow of coroutines and how they are interrupted. The `try`/`finally` behavior might surprise you in particular:

```
>>> active_coroutine = simple_coroutine()
Setting up the coroutine
>>> active_coroutine.send('from caller')
Got item: from caller
>>> active_coroutine.close()
Normal exit
Any exit

>>> active_coroutine = simple_coroutine()
Setting up the coroutine
>>> active_coroutine.throw(RuntimeError, 'caller sent an error')
Traceback (most recent call last):
    ...
RuntimeError: caller sent an error

>>> active_coroutine = simple_coroutine()
Setting up the coroutine
>>> try:
...     active_coroutine.throw(RuntimeError, 'caller sent an error')
... except RuntimeError as exception:
```

```
...     print('Exception:', exception)
Exception exit: caller sent an error
Any exit
Exception: caller sent an error
```

Most of this output is as you would expect, but as was the case with the `StopIteration` in generators, you have to catch the exception to be sure the teardown is handled correctly.

Mixing generators and coroutines

While generators and coroutines appear to be very similar due to the `yield` statements, they are somewhat different beasts. Let's create a two-way pipeline to process the given input and pass this along to multiple coroutines along the way:

```
# The decorator from the Priming section in this chapter
>>> from coroutine_decorator import coroutine

>>> lines = 'some old text', 'really really old', 'old old old'

>>> @coroutine
... def replace(search, replace):
...     while True:
...         item = yield
...         print(item.replace(search, replace))

>>> old_replace = replace('old', 'new')
>>> for line in lines:
...     old_replace.send(line)
some new text
really really new
new new new
```

Given this example, you might be wondering why we are now printing the value instead of yielding it. We can `yield` the value, but remember that generators freeze until a value is yielded. Let's see what will happen if we simply `yield` the value instead of calling `print`. By default, you might be tempted to do this:

```
>>> @coroutine
... def replace(search, replace):
...     while True:
...         item = yield
...         yield item.replace(search, replace)

>>> old_replace = replace('old', 'new')
>>> for line in lines:
```

```
...     old_replace.send(line)
'some new text'
'new new new'
```

Half of the values have disappeared now; our "`really really new`" line has disappeared. Notice that the second `yield` isn't storing the results, and that `yield` effectively makes this a generator and not a coroutine. We need to store the results from that `yield` as well:

```
>>> @coroutine
... def replace(search, replace):
...     item = yield
...     while True:
...         item = yield item.replace(search, replace)

>>> old_replace = replace('old', 'new')
>>> for line in lines:
...     old_replace.send(line)
'some new text'
'really really new'
'new new new'
```

But even this is far from optimal. We are essentially using coroutines to mimic the behavior of generators right now. It works, but it is a bit pointless and offers no real benefit.

Let's make a real pipeline this time where the coroutines send the data to the next coroutine or coroutines. This demonstrates the real power of coroutines, which is being able to chain multiple coroutines together:

```
>>> @coroutine
... def replace(target, search, replace):
...     while True:
...         target.send((yield).replace(search, replace))

# Print will print the items using the provided formatstring
>>> @coroutine
... def print_(formatstring):
...     count = 0
...     while True:
...         count += 1
...         print(count, formatstring.format((yield)))
# tee multiplexes the items to multiple targets
>>> @coroutine
```

```
... def tee(*targets):
...     while True:
...         item = yield
...         for target in targets:
...             target.send(item)
```

Now that we have our coroutine functions, let's see how we can link these together:

```
# Because we wrap the results we need to work backwards from the
# inner layer to the outer layer.

# First, create a printer for the items:
>>> printer = print_('print: {}')

# Create replacers that send the output to the printer
>>> old_replace = replace(printer, 'old', 'new')
>>> current_replace = replace(printer, 'old', 'current')

# Send the input to both replacers
>>> branch = tee(old_replace, current_replace)

# Send the data to the tee routine for processing
>>> for line in lines:
...     branch.send(line)
1 print: some new text
2 print: some current text
3 print: really really new
4 print: really really current
5 print: new new new
6 print: current current current
```

This makes the code much simpler and more readable and shows how you can send a single input source to multiple destinations simultaneously. At first glance, this example does not look that exciting, but the exciting part is that even though we split the input using `tee()` and processed it through two separate `replace()` instances, we still ended up at the same `print_()` function with the same state. This means that it's possible to route and modify your data along whichever way is convenient for you while still having it end up at the same endpoint with no effort whatsoever.

For now, the most important takeaway is that mixing generators and coroutines is not a good idea in most cases since it can have very strange side effects if used incorrectly. Even though both use the `yield` statement, they are significantly different creatures with different behavior. The next section will demonstrate one of the few cases where mixing coroutines and generators can be useful.

Using the state

Now that you know how to write basic coroutines and which pitfalls you have to take care of, how about writing a function where remembering the state is required? That is, a function that always gives you the average value of all sent values. This is one of the few cases where it is still relatively safe and useful to combine the coroutine and generator syntax:

```
>>> import itertools

>>> @coroutine
... def average():
...     total = yield
...     for count in itertools.count(start=1):
...         total += yield total / count

>>> averager = average()
>>> averager.send(20)
20.0
>>> averager.send(10)
15.0
```

It still requires some extra logic to work properly, though. We need to prime our coroutine using `yield`, but we don't send any data at that point because the first `yield` is the primer and is executed before we get the value. Once that's all set up, we can easily yield the average value while summing. It's not all that bad, but the pure coroutine version is slightly simpler to understand since we only have a single execution path because we don't have to worry about priming. To illustrate this, here is the pure coroutine version:

```
>>> import itertools

>>> @coroutine
... def print_(formatstring):
...     while True:
...         print(formatstring.format((yield)))

>>> @coroutine
... def average(target):
...     total = 0
...     for count in itertools.count(start=1):
...         total += yield
...         target.send(total / count)

>>> printer = print_('{:.1f}')
```

```
>>> averager = average(printer)
>>> averager.send(20)
20.0
>>> averager.send(10)
15.0
```

While that example is a few lines longer than the version that includes a generator, it is much easier to understand. Let's analyze it to make sure the workings are clear:

1. We set `total` to `0` to start counting.
2. We keep track of the measurement count by using `itertools.count()`, which we configure to start counting from 1.
3. We fetch the next value using `yield`.
4. We send the average to the given coroutine instead of returning the value to make the code less confusing.

Another nice example is `itertools.groupby`, which is also quite simple to recreate using coroutines. For comparison, I will once again show both the generator coroutine and the pure coroutine version:

```
>>> @coroutine
... def groupby():
...     # Fetch the first key and value and initialize the state
...     # variables
...     key, value = yield
...     old_key, values = key, []
...     while True:
...         # Store the previous value so we can store it in the
...         # list
...         old_value = value
...         if key == old_key:
...             key, value = yield
...         else:
...             key, value = yield old_key, values
...             old_key, values = key, []
...         values.append(old_value)

>>> grouper = groupby()
>>> grouper.send('a1')
>>> grouper.send('a2')
>>> grouper.send('a3')
>>> grouper.send('b1')
('a', ['1', '2', '3'])
>>> grouper.send('b2')
```

```
>>> grouper.send('a1')
('b', ['1', '2'])
>>> grouper.send('a2')
>>> grouper.send((None, None))
('a', ['1', '2'])
```

As you can see, this function uses a few tricks. Firstly, we store the previous `key` and `value` so that we can detect when the group (`key`) changes. Secondly, we obviously cannot recognize a group until the group has changed, so only after the group has changed will the results be returned. This means that the last group will be sent only if a different group is sent after it, hence the `(None, None)`.

The example uses tuple unpacking for the string, splitting `'a1'` into group `'a'` and value `'1'`. Alternatively, you could also use `grouper.send(('a', 1))`.

Now here is the pure coroutine version:

```
>>> @coroutine
... def print_(formatstring):
...     while True:
...         print(formatstring.format(*(yield)))

>>> @coroutine
... def groupby(target):
...     old_key = None
...     while True:
...         key, value = yield
...         if old_key != key:
...             # A different key means a new group so send the
...             # previous group and restart the cycle.
...             if old_key and values:
...                 target.send((old_key, values))
...             values = []
...             old_key = key
...         values.append(value)

>>> grouper = groupby(print_('group: {}, values: {}'))
>>> grouper.send('a1')
>>> grouper.send('a2')
>>> grouper.send('a3')
>>> grouper.send('b1')
group: a, values: ['1', '2', '3']
```

```
>>> grouper.send('b2')
>>> grouper.send('a1')
group: b, values: ['1', '2']
>>> grouper.send('a2')
>>> grouper.send((None, None))
group: a, values: ['1', '2']
```

While the functions are fairly similar, the coroutine version has a less complex control path and only needs to `yield` in one spot. This is because we don't have to think about priming and potentially losing values.

Exercises

Generators have a multitude of uses so you can probably start using them in your own code right away. Nevertheless, the following exercises might help you understand the features and the limitations a bit better:

- Create a generator similar to `itertools.islice()` that allows for a negative step so you can execute `some_list[20:10:-1]`.
- Create a class that wraps a generator so it becomes sliceable by using `itertools.islice()` internally.
- Write a generator for the Fibonacci numbers.
- Write a generator that uses the sieve of Eratosthenes to generate prime numbers.

Example answers for these exercises can be found on GitHub: `https://github.com/mastering-python/exercises`. You are encouraged to submit your own solutions and learn about alternative solutions from others.

Summary

This chapter showed you how to create generators and both the strengths and weaknesses that they possess. Additionally, it should now be clear how to work around their limitations and the implications of doing so.

In general, I would always recommend the use of generators over traditional collection-generating functions. They are easier to write, consume less memory, and, if needed, the downsides can be mitigated by replacing `some_generator()` with `list(some_generator())`, or a decorator that handles that for you.

While the paragraphs about coroutines provided some insights into what they are and how they can be used, they were just a mild introduction to coroutines. Both the pure coroutines and the coroutine generator combinations are still somewhat clunky, which is why the `asyncio` library was created. *Chapter 13, - asyncio – Multithreading without Threads,* covers `asyncio` in detail and also introduces the `async` and `await` statements, which make coroutine usage much more intuitive compared to `yield`.

In the previous chapter, you saw how we can modify classes using class decorators. In the next chapter, we will cover the creation of classes using metaclasses. Using metaclasses, you can modify classes during the creation of the class itself. Note that I am not talking about the instances of the class, but the actual class object. Using this technique, you can create automatically registering plugin systems, add extra attributes to classes, and more.

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:
`https://discord.gg/QMzJenHuJf`

8

Metaclasses – Making Classes (Not Instances) Smarter

The previous chapters have already shown us how to modify classes and functions using decorators. But that's not the only option to modify or extend a class. An even more advanced technique for modifying your classes before creation is the usage of metaclasses. The name already gives you a hint as to what it could be; a metaclass is a class containing meta information about a class.

The basic premise of a metaclass is a class that generates another class for you at definition time, so generally you wouldn't use it to change the class instances, but only the class definitions. By changing the class definitions, it is possible to automatically add some properties to a class, validate whether certain properties are set, change inheritance, automatically register the class with a manager, and many other things.

Although metaclasses are generally considered to be a more powerful technique than (class) decorators, effectively they don't differ too much in possibilities. The choice usually comes down to either convenience or personal preference.

In this chapter, we will cover the following topics:

- Basic dynamic class creation
- Metaclasses with arguments
- Abstract base classes, examples, and inner workings
- Automatic plugin systems using metaclasses
- Internals of class creation and the order of operations
- Storing the definition order of class attributes

Dynamically creating classes

Metaclasses are the factories that create new classes in Python. In fact, even though you may not be aware of it, Python will always execute the `type` metaclass whenever you create a class.

A few common examples where metaclasses are used internally are `abc` (abstract base classes), `dataclasses`, and the Django framework, which heavily relies on metaclasses for the `Model` class.

When creating classes in a procedural way, the `type` metaclass is used as a function that takes three arguments: `name`, `bases`, and `dict`.`name` will become the `__name__` attribute, `bases` is the list of inherited base classes and will be stored in `__bases__`, and `dict` is the namespace dictionary that contains all variables and will be stored in `__dict__`.

It should be noted that the `type()` function has another use as well. Given the arguments documented above, it will create a class with those specifications. Given a single argument with the instance of a class (for example, `type(spam)`), it will return the class object/definition.

Your next question might be, what happens if I call `type()` on a class definition instead of a class instance? Well, that returns the metaclass for the class, which is `type` by default.

Let's clarify this using a few examples:

```
>>> class Spam(object):
...     eggs = 'my eggs'

>>> Spam = type('Spam', (object,), dict(eggs='my eggs'))
```

The above two definitions of `Spam` are completely identical; they both create a class with an instantiated property of `eggs` and `object` as a base. Let's test whether this actually works as you would expect:

```
>>> class Spam(object):
...     eggs = 'my eggs'

>>> spam = Spam()
>>> spam.eggs
'my eggs'
>>> type(spam)
<class ' ...Spam'>
>>> type(Spam)
<class 'type'>

>>> Spam = type('Spam', (object,), dict(eggs='my eggs'))

>>> spam = Spam()
>>> spam.eggs
'my eggs'
>>> type(spam)
<class '...Spam'>
>>> type(Spam)
<class 'type'>
```

As expected, the results for the two are the same. When creating a class, Python will silently add the `type` metaclass, and custom metaclasses are classes that inherit `type`. A simple class definition has a silent metaclass, making a simple definition such as:

```
class Spam(object):
    pass
```

essentially identical to:

```
class Spam(object, metaclass=type):
    pass
```

This raises the question: if every class is created by a (silent) metaclass, what is the metaclass of `type`? This is a recursive definition; the metaclass of `type` is `type`. That is the essence of what a custom metaclass is: a class that inherits `type` to allow class modification without needing to modify the class definition itself.

A basic metaclass

Since metaclasses can modify any class attribute, you can do absolutely anything you wish. Before we continue with more advanced metaclasses, let's create a metaclass that does the following:

1. Makes the class inherit `int`
2. Adds a `lettuce` attribute to the class
3. Changes the name of the class

First we create the metaclass. After that, we create a class both with and without the metaclass:

```
# The metaclass definition, note the inheritance of type instead
# of object
>>> class MetaSandwich(type):
...     # Notice how the __new__ method has the same arguments
...     # as the type function we used earlier?
...     def __new__(metaclass, name, bases, namespace):
...         name = 'SandwichCreatedByMeta'
...         bases = (int,) + bases
...         namespace['lettuce'] = 1
...         return type.__new__(metaclass, name, bases, namespace)
```

First, the regular Sandwich:

```
>>> class Sandwich(object):
...     pass

>>> Sandwich.__name__
'Sandwich'
```

```
>>> issubclass(Sandwich, int)
False
>>> Sandwich.lettuce
Traceback (most recent call last):
    ...
AttributeError: type object 'Sandwich' has no attribute 'lettuce'
```

Now, the meta-Sandwich:

```
>>> class Sandwich(object, metaclass=MetaSandwich):
...     pass

>>> Sandwich.__name__
'SandwichCreatedByMeta'
>>> issubclass(Sandwich, int)
True
>>> Sandwich.lettuce
1
```

As you can see, the class with the custom metaclass now inherits `int`, has the `lettuce` attribute, and has a different name.

With metaclasses, you can modify any aspect of the class definition. That makes them a tool that is both very powerful and potentially very confusing. With just a few small modifications, you can cause the strangest of bugs in your (or others') code.

Arguments to metaclasses

The possibility of adding arguments to a metaclass is a little-known feature, but very useful nonetheless. In many cases, simply adding attributes or methods to a class definition is enough to detect what to do, but there are cases where it is useful to be more specific:

```
>>> class AddClassAttributeMeta(type):
...     def __init__(metaclass, name, bases, namespace, **kwargs):
...         # The kwargs should not be passed on to the
...         # type.__init__
...         type.__init__(metaclass, name, bases, namespace)
...
...     def __new__(metaclass, name, bases, namespace, **kwargs):
...         for k, v in kwargs.items():
...             # setdefault so we don't overwrite attributes
...             namespace.setdefault(k, v)
...
```

```
...         return type.__new__(metaclass, name, bases, namespace)

>>> class WithArgument(metaclass=AddClassAttributeMeta, a=1234):
...     pass

>>> WithArgument.a
1234
>>> with_argument = WithArgument()
>>> with_argument.a
1234
```

This simplistic example may not be useful, but the possibilities are. For example, a metaclass that automatically registers a plugin in a plugin registry could use this to specify plugin name aliases.

With this feature, instead of having to include all class-creating parameters as attributes and methods on the class, you can pass these arguments without polluting your class. The only thing you need to keep in mind is that both the `__new__` and `__init__` methods need to be extended in order for this to work because the arguments are passed to the metaclass constructor (`__init__`).

Since Python 3.6, however, we have had a simpler alternative to get this effect. Python 3.6 introduced the `__init_subclass__` magic method, which allows for similar modifications in a slightly easier way:

```
>>> class AddClassAttribute:
...     def __init_subclass__(cls, **kwargs):
...         super().__init_subclass__()
...
...         for k, v in kwargs.items():
...             setattr(cls, k, v)

>>> class WithAttribute(metaclass=AddClassAttributeMeta, a=1234):
...     pass

>>> WithAttribute.a
1234
>>> with_attribute = WithAttribute()
>>> with_attribute.a
1234
```

Several of the metaclasses in this chapter could be replaced with the `__init_subclass__` method, and it is a very useful option for small modifications. For larger changes, I would recommend using a full metaclass instead to make the distinction between the regular class and the metaclass slightly more obvious.

Accessing metaclass attributes through classes

When using metaclasses, it might be confusing that the class actually does more than simply construct the class; it's actually inheriting the class during the creation. To illustrate:

```
>>> class Meta(type):
...     @property
...     def some_property(cls):
...         return 'property of %r' % cls
...
...     def some_method(self):
...         return 'method of %r' % self

>>> class SomeClass(metaclass=Meta):
...     pass

# Accessing through the class definition
>>> SomeClass.some_property
"property of <class '...SomeClass'>"
>>> SomeClass.some_method
<bound method Meta.some_method of <class '__main__.SomeClass'>>
>>> SomeClass.some_method()
"method of <class '__main__.SomeClass'>"

# Accessing through an instance
>>> some_class = SomeClass()
>>> some_class.some_property
Traceback (most recent call last):
    ...
AttributeError: 'SomeClass' object has no attribute 'some_property'
>>> some_class.some_method
Traceback (most recent call last):
    ...
AttributeError: 'SomeClass' object has no attribute 'some_method'
```

As can be seen in the preceding example, these methods are only available for the class objects and not the instances. The `some_property` and `some_method` are not accessible through the instance, while they are accessible through the class. This can be useful for making some functions class- (as opposed to instance-) only, and it keeps your class namespace cleaner.

In the general case, however, I suspect this only adds confusion, so I would typically recommend against it.

Abstract classes using collections.abc

The abstract base classes (also known as interface classes) module is one of the most useful and most widely used examples of metaclasses in Python, as it makes it easy to ensure that a class adheres to a certain interface without a lot of manual checks. We have already seen some examples of abstract base classes in previous chapters, but now we will also look at their inner workings and some more advanced features, such as custom abstract base classes (ABCs).

Internal workings of the abstract classes

First, let's demonstrate the usage of the regular abstract base class:

```
>>> import abc

>>> class AbstractClass(metaclass=abc.ABCMeta):
...     @abc.abstractmethod
...     def some_method(self):
...         raise NotImplemented()

>>> class ConcreteClass(AbstractClass):
...     pass

>>> ConcreteClass()
Traceback (most recent call last):
    ...
TypeError: Can't instantiate abstract class ConcreteClass with
abstract methods some_method

>>> class ImplementedConcreteClass(ConcreteClass):
...     def some_method():
...         pass

>>> instance = ImplementedConcreteClass()
```

As you can see, the abstract base class blocks us from instantiating the classes until all abstract methods have been inherited. This is really useful when your code expects certain properties or methods to be available, but a sane default value is not an option. A common example of this is with base classes for plugins and data models.

In addition to regular methods, `property`, `staticmethod`, and `classmethod` are also supported:

```
>>> import abc

>>> class AbstractClass(object, metaclass=abc.ABCMeta):
...     @property
...     @abc.abstractmethod
...     def some_property(self):
...         raise NotImplemented()
...
...     @classmethod
...     @abc.abstractmethod
...     def some_classmethod(cls):
...         raise NotImplemented()
...
...     @staticmethod
...     @abc.abstractmethod
...     def some_staticmethod():
...         raise NotImplemented()
...
...     @abc.abstractmethod
...     def some_method():
...         raise NotImplemented()
```

So what does Python do internally? You could, of course, read the `abc.py` source code, but I think a simple explanation would be better.

First, the `abc.abstractmethod` sets the `__isabstractmethod__` property on the function to `True`. So if you don't want to use the decorator, you could simply emulate the behavior by doing something along the lines of:

```
some_method.__isabstractmethod__ = True
```

After that, the `abc.ABCMeta` metaclass walks through all of the items in the `namespace` and looks for objects where the `__isabstractmethod__` attribute evaluates to `True`. In addition to that, it will walk through all `bases` and check the `__abstractmethods__` set for every base class, in case the class inherits an abstract class. All of the items where `__isabstractmethod__` still evaluates to `True` will be added to the `__abstractmethods__` set that is stored in the class as a `frozenset`.

Note that we don't use `abc.abstractproperty`, `abc.abstractclassmethod`, and `abc.abstractstaticmethod`. Since Python 3.3, these have been deprecated as the `classmethod`, `staticmethod`, and `property` decorators are recognized by `abc.abstractmethod`, so a simple `property` decorator followed by an `abc.abstractmethod` is recognized as well. Take care when ordering the decorators; `abc.abstractmethod` needs to be the innermost decorator for this to work properly.

The next question now is where the actual checks come in, the checks to see whether the classes are completely implemented. This actually functions through a few Python internals:

```
>>> class AbstractMeta(type):
...     def __new__(metaclass, name, bases, namespace):
...         cls = super().__new__(metaclass, name, bases,
...                               namespace)
...         cls.__abstractmethods__ = frozenset(('something',))
...         return cls

>>> class ConcreteClass(metaclass=AbstractMeta):
...     pass

>>> ConcreteClass()
Traceback (most recent call last):
    ...
TypeError: Can't instantiate abstract class ConcreteClass with
abstract methods something
```

We can easily emulate the same behavior with a metaclass ourselves, but it should be noted that `abc.ABCMeta` actually does more, which we will demonstrate in the next section. To illustrate the behavior as described above, let's create an abstract base metaclass that mimics `abc.ABCMeta`:

```
>>> import functools

>>> class AbstractMeta(type):
...     def __new__(metaclass, name, bases, namespace):
...         # Create the class instance
...         cls = super().__new__(metaclass, name, bases,
...                               namespace)
...
...         # Collect all local methods marked as abstract
...         abstracts = set()
...         for k, v in namespace.items():
...             if getattr(v, '__abstract__', False):
...                 abstracts.add(k)
...
...         # Look for abstract methods in the base classes and
...         # add them to the list of abstracts
...         for base in bases:
...             for k in getattr(base, '__abstracts__', ()):
...                 v = getattr(cls, k, None)
...                 if getattr(v, '__abstract__', False):
```

```
...                     abstracts.add(k)
...
...         # Store the abstracts in a frozenset so they cannot be
...         # modified
...         cls.__abstracts__ = frozenset(abstracts)
...
...         # Decorate the __new__ function to check if all
...         # abstract functions were implemented
...         original_new = cls.__new__
...         @functools.wraps(original_new)
...         def new(self, *args, **kwargs):
...             for k in self.__abstracts__:
...                 v = getattr(self, k)
...                 if getattr(v, '__abstract__', False):
...                     raise RuntimeError(
...                         '%r is not implemented' % k)
...
...             return original_new(self, *args, **kwargs)
...
...         cls.__new__ = new
...         return cls

# Create a decorator that sets the '__abstract__' attribute
>>> def abstractmethod(function):
...     function.__abstract__ = True
...     return function
```

Now that we have the metaclass and decorator for creating abstract classes, let's see if it works as expected:

```
>>> class ConcreteClass(metaclass=AbstractMeta):
...     @abstractmethod
...     def some_method(self):
...         pass

# Instantiating the function, we can see that it functions as the
# regular ABCMeta does
>>> ConcreteClass()
Traceback (most recent call last):
    ...
RuntimeError: 'some_method' is not implemented
```

The actual implementation is much more complicated since it needs to handle decorators such as `property`, `classmethod`, and `staticmethod`. It also has some caching to features, but this code covers the most useful part of the implementation. One of the most important tricks to note here is that the actual check is executed by decorating the `__new__` function of the actual class. This method is only executed once within a class, so we can avoid the overhead of these checks for multiple instantiations.

The actual implementation of the abstract methods can be found by looking for `__isabstractmethod__` in the Python source code in the following files: `Objects/descrobject.c`, `Objects/funcobject.c`, and `Objects/object.c`. The Python part of the implementation can be found in `Lib/abc.py`.

Custom type checks

Defining your own interfaces using abstract base classes is great, of course. But it can also be very convenient to tell Python what your class actually resembles and what kind of types are similar. For that, `abc.ABCMeta` offers a register function that allows you to specify which types are similar. For example, a custom `list` that sees the `list` type as similar:

```
>>> import abc

>>> class CustomList(abc.ABC):
...     '''This class implements a list-like interface'''

>>> class CustomInheritingList(list, abc.ABC):
...     '''This class implements a list-like interface'''

>>> issubclass(list, CustomList)
False
>>> issubclass(list, CustomInheritingList)
False

>>> CustomList.register(list)
<class 'list'>

# We can't make it go both ways, however
>>> CustomInheritingList.register(list)
Traceback (most recent call last):
    ...
RuntimeError: Refusing to create an inheritance cycle

>>> issubclass(list, CustomList)
True
>>> issubclass(list, CustomInheritingList)
```

```
False

# We need to inherit list to make it work the other way around
>>> issubclass(CustomList, list)
False
>>> isinstance(CustomList(), list)
False
>>> issubclass(CustomInheritingList, list)
True
>>> isinstance(CustomInheritingList(), list)
True
```

As demonstrated by the last eight lines, this is a one-way relationship. The other way around requires inheriting `list`, but due to inheritance cycles, it can't be done both ways. Otherwise, `CustomInheritingList` would inherit `list` and `list` would inherit `CustomInheritingList`, which could recurse forever during the `issubclass()` call.

To be able to handle cases like these, there is another useful feature in `abc.ABCMeta`. When subclassing `abc.ABCMeta`, the `__subclasshook__` method can be extended to customize the behavior of `issubclass` and with that, `isinstance`:

```
>>> import abc

>>> class UniversalClass(abc.ABC):
...     @classmethod
...     def __subclasshook__(cls, subclass):
...         return True

>>> issubclass(list, UniversalClass)
True
>>> issubclass(bool, UniversalClass)
True
>>> isinstance(True, UniversalClass)
True
>>> issubclass(UniversalClass, bool)
False
```

The `__subclasshook__` should return `True`, `False`, or `NotImplemented`, which results in `issubclass` returning `True`, `False`, or the usual behavior when `NotImplemented` is returned.

Automatically registering plugin systems

One very useful way to use metaclasses is to have classes automatically register themselves as plugins/handlers.

Instead of manually adding a register call after creating the class or by adding a decorator, you can make it completely automatic for the user. That means that the user of your library or plugin system cannot accidentally forget to add the register call.

Note the distinction between registering and importing. While this first example shows automatic registering, automatic importing is covered in later sections.

Examples of these can be seen in many projects such as web frameworks. The Django web framework, for example, uses metaclasses for its database models (effectively tables) to automatically generate the table and column names based on the class and attribute names.

The actual code base of projects like these is too extensive to usefully explain here though. Hence, we'll show a simpler example that demonstrates the power of metaclasses as a self-registering plugin system:

```
>>> import abc

>>> class Plugins(abc.ABCMeta):
...     plugins = dict()
...
...     def __new__(metaclass, name, bases, namespace):
...         cls = abc.ABCMeta.__new__(metaclass, name, bases,
...                                   namespace)
...         if isinstance(cls.name, str):
...             metaclass.plugins[cls.name] = cls
...         return cls
...
...     @classmethod
...     def get(cls, name):
...         return cls.plugins[name]

>>> class PluginBase(metaclass=Plugins):
...     @property
...     @abc.abstractmethod
...     def name(self):
...         raise NotImplemented()

>>> class PluginA(PluginBase):
...     name = 'a'

>>> class PluginB(PluginBase):
...     name = 'b'
```

```
>>> Plugins.get('a')
<class '...PluginA'>

>>> Plugins.plugins
{'a': <class '...PluginA'>,
 'b': <class '...PluginB'>}
```

This example is a tad simplistic of course, but it's the basis for many plugin systems.

While metaclasses run at definition time, the module still needs to be **imported** to work. There are several options for doing this; loading on-demand through the `get` method would have my vote if possible, as that also doesn't add load time if the plugin is not used.

The following examples will use the following file structure to get reproducible results. All files will be contained in a `plugins` directory. Note that all the code for this book, including this example, can be found on GitHub: `https://github.com/mastering-python/code_2`.

The `__init__.py` file is used to create shortcuts, so a simple `import plugins` will result in having `plugins.Plugins` available, instead of requiring the import of `plugins.base` explicitly:

```
# plugins/__init__.py
from .base import Plugin
from .base import Plugins

__all__ = ['Plugin', 'Plugins']
```

Here's the `base.py` file containing the `Plugins` collection and the `Plugin` base class:

```
# plugins/base.py
import abc

class Plugins(abc.ABCMeta):
    plugins = dict()

    def __new__(metaclass, name, bases, namespace):
        cls = abc.ABCMeta.__new__(
            metaclass, name, bases, namespace)
        metaclass.plugins[name.lower()] = cls
        return cls

    @classmethod
    def get(cls, name):
```

```
        return cls.plugins[name]

class Plugin(metaclass=Plugins):
    pass
```

And two simple plugins, `a.py` and `b.py` (omitted since it's functionally identical to `a.py`):

```
from . import base

class A(base.Plugin):
    pass
```

Now that we have set up the plugins and the automatic registering, we need to take care of the loading of `a.py` and `b.py`. While `A` and `B` will automatically register within `Plugins`, if you forget to `import` them, they will not be registered. To solve this, we have several options; first we will look at on-demand loading.

Importing plugins on-demand

The first of the solutions for the import problem is simply taking care of it in the `get` method of the `Plugins` metaclass. Whenever the plugin is not found in the registry, the `get` method should automatically `import` the module from the `plugins` directory.

The advantages of this approach are that the plugins don't explicitly need to be preloaded, but also that the plugins are only loaded when the need is there. Unused plugins won't be touched, so this method can help in reducing your applications' load times.

The downsides are that the code will not be run or tested, so it might be completely broken and you won't know about it until it is finally loaded. Solutions for this problem will be covered in the chapter on testing, *Chapter 10*. The other problem is that if the code self-registers into other parts of an application, then that code won't be executed either, unless you add the required `import` in other parts of the code, that is.

Modifying the `Plugins.get` method, we get the following:

```
import importlib

# Plugins class omitted for brevity
class PluginsOnDemand(Plugins):
    @classmethod
    def get(cls, name):
        if name not in cls.plugins:
            print('Loading plugins from plugins.%s' % name)
            importlib.import_module('plugins.%s' % name)
        return cls.plugins[name]
```

Now we run this from a Python file:

```
import plugins

print(plugins.PluginsOnDemand.get('a'))
print(plugins.PluginsOnDemand.get('a'))
```

Which results in:

```
Loading plugins from plugins.a
<class 'plugins.a.A'>
<class 'plugins.a.A'>
```

As you can see, this approach only results in running the `import` once; the second time, the plugin will be available in the plugins dictionary, so no loading will be necessary.

Importing plugins through configuration

While only loading the required plugins is useful because it reduces your initial load time and memory overhead, there is something to be said about preloading the plugins you will likely need. As dictated by the Zen of Python, explicit is better than implicit, so an explicit list of plugins to load is generally a good solution. The added advantages of this method are that you are able to make the registration a bit more advanced as you are guaranteed it is run, and that you can load plugins from multiple packages. The disadvantage is, of course, that you need to explicitly define which plugins to load, which could be considered a violation of the DRY (Don't Repeat Yourself) principle.

Instead of importing in the `get` method, we will add a `load` method this time, which imports all the given module names:

```
# PluginsOnDemand class omitted for brevity
class PluginsThroughConfiguration(PluginsOnDemand):
    @classmethod
    def load(cls, *plugin_names):
        for plugin_name in plugin_names:
            cls.get(plugin_name)
```

Which can be called using the following code:

```
import plugins

plugins.PluginsThroughConfiguration.load(
    'a',
    'b',
)

print('After load')
```

```
print(plugins.PluginsThroughConfiguration.get('a'))
print(plugins.PluginsThroughConfiguration.get('a'))
```

This results in the following output:

```
Loading plugins from plugins.a
Loading plugins from plugins.b
After load
<class 'plugins.a.A'>
<class 'plugins.a.A'>
```

A fairly simple and straightforward system to load the plugins based on settings, this could easily be combined with any type of settings system to fill the `load` method. An example of this method is `INSTALLED_APPS` in Django.

Importing plugins through the filesystem

The most convenient method of loading plugins is one you don't have to think about because it happens automatically. While this is very convenient, very important caveats should be considered.

First, they often make debugging much more difficult. Similar automatic import systems in Django have caused me a fair share of headaches, as they tend to obfuscate errors or even completely hide them, making you debug for hours.

Second, it can be a security risk. If someone has write access to one of your plugin directories, they can effectively execute code within your application.

Having that said, especially for beginners and/or new users of your framework, automatic plugin loading can be very convenient and certainly warrants a demonstration.

This time, we inherit the `PluginsThroughConfiguration` class we created in the previous example, and add an `autoload` method to detect available plugins.

```
import re
import pathlib
import importlib

CURRENT_FILE = pathlib.Path(__file__)
PLUGINS_DIR = CURRENT_FILE.parent
MODULE_NAME_RE = re.compile('[a-z][a-z0-9_]*', re.IGNORECASE)

class PluginsThroughFilesystem(PluginsThroughConfiguration):
    @classmethod
    def autoload(cls):
        for filename in PLUGINS_DIR.glob('*.py'):
            # Skip __init__.py and other non-plugin files
            if not MODULE_NAME_RE.match(filename.stem):
```

```
                continue
                cls.get(filename.stem)

            # Skip this file
            if filename == CURRENT_FILE:
                continue

            # Load the plugin
            cls.get(filename.stem)
```

Now, let's give this code a try:

```
import pprint
import plugins

plugins.PluginsThroughFilesystem.autoload()

print('After load')
pprint.pprint(plugins.PluginsThroughFilesystem.plugins)
```

This results in:

```
Loading plugins from plugins.a
Loading plugins from plugins.b
After load
{'a': <class 'plugins.a.A'>,
 'b': <class 'plugins.b.B'>,
 'plugin': <class 'plugins.base.Plugin'>}
```

Now every file in the `plugins` directory will automatically be loaded. But note that it can obscure certain errors. For example, if one of your plugins imports a library that you do not have installed, you will get the `ImportError` from the plugin, not the actual library.

To make this system a bit smarter (even importing packages outside of your Python path), you can create a plugin loader using the abstract base classes in `importlib.abc`; note that you will most likely still need to somehow list the files and/or directories though. To improve this, you could also take a look at the loaders in `importlib`. Using these loaders, you can load plugins from ZIP files and other sources as well.

Now that we are done with plugin systems, it is time to look at how `dataclasses` could be implemented using metaclasses instead of decorators.

Dataclasses

In *Chapter 4, Pythonic Design Patterns*, we already saw the `dataclasses` module, which makes it possible to implement easy type hinting and even enforce some structure in your classes.

Now let's look at how we can implement our own version using a metaclass. The actual `dataclasses` module mostly relies on a class decorator, but that is no issue. Metaclasses can be seen as a more powerful version of a class decorator, so they will work fine. With metaclasses, you can use inheritance to reuse them, or make the class inherit other classes, but above all, they allow you to modify the class object, instead of the instance with decorators.

The `dataclasses` module has several tricks up its sleeve that are non-trivial to replicate. Beyond adding documentation and some utility methods, it also generates an `__init__` method with a `signature` that matches the fields of the `dataclass`. Since the entire `dataclasses` module is roughly 1,300 lines, we will not get close with our implementation. So we will implement the `__init__()` method, including a generated `signature` and `__annotations__` for type hinting, and a `__repr__` method to show the results:

```
import inspect

class Dataclass(type):
    def _get_signature(namespace):
        # Get the annotations from the class
        annotations = namespace.get('__annotations__', dict())

        # Signatures are immutable so we need to build the
        # parameter list before creating the signature
        parameters = []
        for name, annotation in annotations.items():
            # Create Parameter shortcut for readability
            Parameter = inspect.Parameter

            # Create the parameter with the correct type
            # annotation and default. You could also choose to
            # make the arguments keyword/positional only here
            parameters.append(Parameter(
                name=name,
                kind=Parameter.POSITIONAL_OR_KEYWORD,
                default=namespace.get(name, Parameter.empty),
                annotation=annotation,
            ))

        return inspect.Signature(parameters)

    def _create_init(namespace, signature):
        # If init exists we don't need to do anything
        if '__init__' in namespace:
            return
```

```
        # Create the __init__ method and use the signature to
        # process the arguments
        def __init__(self, *args, **kwargs):
            bound = signature.bind(*args, **kwargs)
            bound.apply_defaults()

            for key, value in bound.arguments.items():
                # Convert to the annotation to enforce types
                parameter = signature.parameters[key]
                # Set the casted value
                setattr(self, key, parameter.annotation(value))

        # Override the signature for __init__ so help() works
        __init__.__signature__ = signature

        namespace['__init__'] = __init__

    def _create_repr(namespace, signature):
        def __repr__(self):
            arguments = []
            for key, value in vars(self).items():
                arguments.append(f'{key}={value!r}')
            arguments = ', '.join(arguments)
            return f'{self.__class__.__name__}({arguments})'

        namespace['__repr__'] = __repr__

    def __new__(metaclass, name, bases, namespace):
        signature = metaclass._get_signature(namespace)
        metaclass._create_init(namespace, signature)
        metaclass._create_repr(namespace, signature)

        cls = super().__new__(metaclass, name, bases, namespace)

        return cls
```

At first glance, this might look complicated, but the general process is actually fairly simple:

1. We generate a signature from the `__annotations__` and defaults in the class.
2. We generate an `__init__` method based on the signature.
3. We make the `__init__` method use the signature to automatically bind the arguments passed to the function and apply those to the instance.

4. We generate a `__repr__` method, which simply prints the class name and the values stored in the instance. Note that this method is rather limited and will show anything you've added to the class.

Note that as an extra little touch, we have a cast to the annotated type to enforce the type correctly.

Let's see if it works as expected by using the `dataclass` example from *Chapter 4* with a few small additions to test the type conversions:

```
>>> from T_10_dataclasses import Dataclass

>>> class Sandwich(metaclass=Dataclass):
...     spam: int
...     eggs: int = 3

>>> Sandwich(1, 2)
Sandwich(spam=1, eggs=2)

>>> sandwich = Sandwich(4)
>>> sandwich
Sandwich(spam=4, eggs=3)
>>> sandwich.eggs
3

>>> help(Sandwich.__init__)
Help on function __init__ in ...
<BLANKLINE>
__init__(spam: int, eggs: int = 3)
<BLANKLINE>

>>> Sandwich('a')
Traceback (most recent call last):
    ...
ValueError: invalid literal for int() with base 10: 'a'

>>> Sandwich('1234', 56.78)
Sandwich(spam=1234, eggs=56)
```

That all functions as expected, with similar output to the original `dataclass`. Naturally, it's far more limited in features, but it shows how you can generate your own classes and functions dynamically, and how easy it is to add automatic annotation-based type casting to your code.

Next up is a deep dive into the creation and instantiation of classes.

Order of operations when instantiating classes

The order of operations during class instantiation is very important to keep in mind when debugging issues with dynamically created and/or modified classes. Assuming an incorrect order can cause difficult-to-trace bugs. The instantiation of a class happens in the following order:

1. Finding the metaclass
2. Preparing the namespace
3. Executing the class body
4. Creating the class object
5. Executing the class decorators
6. Creating the class instance

We will go through each of these now.

Finding the metaclass

The metaclass comes from either the explicitly given metaclass on the class or `bases`, or by using the default `type` metaclass.

For every class, the class itself and the bases, the first matching of the following will be used:

- Explicitly given metaclass
- Explicit metaclass from `bases`
- `type()`

Note that if no metaclass is found that is a subtype of all of the candidate metaclasses, a `TypeError` will be raised. This scenario is not that likely to occur, but is certainly a possibility when using multiple inheritance/mixins with metaclasses.

Preparing the namespace

The class namespace is prepared through the metaclass selected above. If the metaclass has a `__prepare__` method, it will be called as `namespace = metaclass.__prepare__(names, bases, **kwargs)` where the `**kwargs` originate from the class definition. If no `__prepare__` method is available, the result will be `namespace = dict()`.

Note that there are multiple ways of achieving custom namespaces. As we saw in the previous section, the `type()` function call also takes a `dict` argument, which can be used to alter the namespace as well.

Executing the class body

The body of the class is executed very similarly to normal code execution with one key difference: the separate namespace. Since a class has a separate namespace, which shouldn't pollute the `globals()`/`locals()` namespaces, it is executed within that context. The resulting call looks something like this:

```
exec(body, globals(), namespace)
```

where the `namespace` is the previously produced namespace.

Creating the class object (not instance)

Now that we have all components ready, the actual class object can be produced. This is done through the `class_ = metaclass(name, bases, namespace, **kwargs)` call, which is, as you can see, actually identical to the `type()` call previously discussed. The `**kwargs` here are the same as the ones passed to the `__prepare__` method earlier.

It might be useful to note that this is also the object that will be referenced from the `super()` call without arguments.

Executing the class decorators

Now that the class object is actually done already, the class decorators will be executed. Since this is only executed after everything else in the class object has already been constructed, it becomes difficult to modify class attributes such as which classes are being inherited and the name of the class. By modifying the `__class__` object, you can still modify or overwrite these, but it is, at the very least, more difficult.

Creating the class instance

From the class object produced above, we can now finally create the actual instances as you normally would with a class. It should be noted that, unlike the steps above, this step and the class decorators step, are the only ones that are executed every time you instantiate a class. The steps before these two are only executed once per class definition.

Example

Enough theory – let's illustrate the creation and instantiation of the class objects so we can check the order of operations:

```
>>> import functools

>>> def decorator(name):
...     def _decorator(cls):
...         @functools.wraps(cls)
...         def __decorator(*args, **kwargs):
...             print('decorator(%s)' % name)
...             return cls(*args, **kwargs)
...
...         return __decorator
...
...     return _decorator

>>> class SpamMeta(type):
...     @decorator('SpamMeta.__init__')
```

```
...     def __init__(self, name, bases, namespace, **kwargs):
...         print('SpamMeta.__init__()')
...         return type.__init__(self, name, bases, namespace)
...
...     @staticmethod
...     @decorator('SpamMeta.__new__')
...     def __new__(cls, name, bases, namespace, **kwargs):
...         print('SpamMeta.__new__()')
...         return type.__new__(cls, name, bases, namespace)
...
...     @classmethod
...     @decorator('SpamMeta.__prepare__')
...     def __prepare__(cls, names, bases, **kwargs):
...         print('SpamMeta.__prepare__()')
...         namespace = dict(spam=5)
...         return namespace
```

With the created class and decorator, we can now illustrate when methods such as `__prepare__` and `__new__` are called:

```
>>> @decorator('Spam')
... class Spam(metaclass=SpamMeta):
...     @decorator('Spam.__init__')
...     def __init__(self, eggs=10):
...         print('Spam.__init__()')
...         self.eggs = eggs
decorator(SpamMeta.__prepare__)
SpamMeta.__prepare__()
decorator(SpamMeta.__new__)
SpamMeta.__new__()
decorator(SpamMeta.__init__)
SpamMeta.__init__()

# Testing with the class object
>>> spam = Spam
>>> spam.spam
5
>>> spam.eggs
Traceback (most recent call last):
  ...
  File "<doctest T_11_order_of_operations.rst[6]>", line 1, in ...
AttributeError: 'function' object has no attribute 'eggs'
```

```
# Testing with a class instance
>>> spam = Spam()
decorator(Spam)
decorator(Spam.__init__)
Spam.__init__()
>>> spam.spam
5
>>> spam.eggs
10
```

The example clearly shows the creation order of the class:

1. Preparing the namespace through `__prepare__`
2. Creating the class body using `__new__`
3. Initializing the metaclass using `__init__` (note: this is not the class `__init__`)
4. Initializing the class through the class decorator
5. Initializing the class through the class `__init__` function

One thing we can note from this is that class decorators are executed each and every time the class is actually instantiated and not before that. This can be both an advantage and a disadvantage, of course, but if you wish to build a register of all subclasses, it is definitely more convenient to use a metaclass since the decorator will not register until you instantiate the class.

In addition to this, having the power to modify the namespace before actually creating the class object (not the instance) can be very powerful as well. This can be convenient for sharing a certain scope between several class objects, for example, or to easily ensure that certain items are always available in the scope.

Storing class attributes in definition order

There are cases where the definition order makes a difference. For example, let's assume we are creating a class that represents a CSV (Comma-Separated Values) format. The CSV format expects fields to have a particular order. In some cases, this will be indicated by a header, but it's still useful to have a consistent field order. Similar systems are used in ORM systems such as SQLAlchemy to store the column order for table definitions, and for the input field order within forms in Django.

The classic solution without metaclasses

An easy way to store the order of the fields is by giving the field instances a special `__init__` method that increments for every definition, so the fields have an incrementing index property. This solution could be considered the classic solution, as it would also work in Python 2:

```
>>> import itertools

>>> class Field(object):
```

```
...     counter = itertools.count()
...
...     def __init__(self, name=None):
...         self.name = name
...         self.index = next(Field.counter)
...
...     def __repr__(self):
...         return '<%s[%d] %s>' % (
...             self.__class__.__name__,
...             self.index,
...             self.name,
...         )

>>> class FieldsMeta(type):
...     def __new__(metaclass, name, bases, namespace):
...         cls = type.__new__(metaclass, name, bases, namespace)
...         fields = []
...         for k, v in namespace.items():
...             if isinstance(v, Field):
...                 fields.append(v)
...                 v.name = v.name or k
...
...         cls.fields = sorted(fields, key=lambda f: f.index)
...         return cls

>>> class Fields(metaclass=FieldsMeta):
...     spam = Field()
...     eggs = Field()

>>> Fields.fields
[<Field[0] spam>, <Field[1] eggs>]

>>> fields = Fields()
>>> fields.eggs.index
1
>>> fields.spam.index
0
>>> fields.fields
[<Field[0] spam>, <Field[1] eggs>]
```

For convenience, and to make things prettier, we have added the `FieldsMeta` class.

It is not strictly required here, but it automatically takes care of filling in the `name` if needed, and adds the `fields` list, which contains a sorted list of fields.

Using metaclasses to get a sorted namespace

The previous solution is a bit more straightforward and supports Python 2 as well, but with Python 3 we have more options. As you have seen in the previous section, Python 3 gave us the `__prepare__` method, which returns the namespace. From *Chapter 4*, you might remember `collections.OrderedDict`, so let's see what happens when we combine them:

```
>>> import collections

>>> class Field(object):
...     def __init__(self, name=None):
...         self.name = name
...
...     def __repr__(self):
...         return '<%s %s>' % (
...             self.__class__.__name__,
...             self.name,
...         )

>>> class FieldsMeta(type):
...     @classmethod
...     def __prepare__(metaclass, name, bases):
...         return collections.OrderedDict()
...
...     def __new__(metaclass, name, bases, namespace):
...         cls = type.__new__(metaclass, name, bases, namespace)
...         cls.fields = []
...         for k, v in namespace.items():
...             if isinstance(v, Field):
...                 cls.fields.append(v)
...                 v.name = v.name or k
...
...         return cls

>>> class Fields(metaclass=FieldsMeta):
...     spam = Field()
...     eggs = Field()

>>> Fields.fields
```

```
[<Field spam>, <Field eggs>]
>>> fields = Fields()
>>> fields.fields
[<Field spam>, <Field eggs>]
```

As you can see, the fields are indeed in the order we defined them. Spam first, eggs after that. Since the class namespace is now a `collections.OrderedDict` instance, we know that the order is guaranteed. It should be noted that, since Python 3.6, the order of the regular `dict` is also consistent, but the usage example of `__prepare__` is still useful. It demonstrates how convenient metaclasses can be to extend your classes in a generic way. Another big advantage of metaclasses instead of a custom `__init__` method is that users won't lose the functionality if they forget to call the parent `__init__` method. The metaclass will always be executed, unless a different metaclass is added, that is.

Exercises

The most important point of this chapter is to teach you how metaclasses work internally: a metaclass is just a class that creates a class, which, in turn, is created by another metaclass (eventually ending up recursively at `type`). If you want to challenge yourself, however, there is more you can do with metaclasses:

- Validation is one of the most prominent examples of where metaclasses can be useful. You can validate to check if attributes/methods are available, you can check if required classes are inherited, and so on. The possibilities are endless.
- Build a metaclass that wraps every method with a decorator (could be useful for logging/debugging purposes), something with a signature like this:

```
class SomeClass(metaclass=WrappingMeta, wrapper=some_wrapper):
```

Example answers for these exercises can be found on GitHub: `https://github.com/mastering-python/exercises`. You are encouraged to submit your own solutions and learn about alternative solutions from others.

Summary

The Python metaclass system is something every Python programmer uses all the time, perhaps without even knowing about it. Every class is created through some (subclass of) `type`, which allows for endless customization and magic.

Instead of statically defining your class, you can now have it created as you normally would and dynamically add, modify, or remove attributes from your class during definition; very magical but very useful. The magic component, however, is also the reason why metaclasses should be used with a lot of caution. While they can be used to make your life much easier, they are also among the easiest ways of producing completely incomprehensible code.

Regardless, there are some great use cases for metaclasses, and many libraries such as SQLAlchemy and Django use metaclasses to make your code work much more easily and arguably better. Actually comprehending the magic that is used inside is generally not needed for the usage of these libraries, which makes the cases defendable.

The question becomes whether a much better experience for beginners is worth some dark magic internally, and looking at the success of these libraries, I would say *yes* in this case.

To conclude, when thinking about using metaclasses, keep in mind what Tim Peters once said:

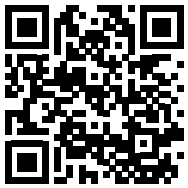
> "Metaclasses are deeper magic than 99% of users should ever worry about. If you wonder whether you need them, you don't."

With the introduction of class decorators and methods such as `__init_subclass__` and `__set_name__`, the need for metaclasses has dwindled even further. So when in doubt, you probably have no real need for them.

Now we will continue with a solution to remove some of the magic that metaclasses generate – documentation. The next chapter will show us how your code can be documented, how that documentation can be tested, and most importantly, how the documentation can be made smarter by annotating types.

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:
`https://discord.gg/QMzJenHuJf`



9

Documentation – How to Use Sphinx and reStructuredText

Documenting code can be both fun and useful! I will admit that many programmers have a strong dislike for documenting code and understandably so. Writing documentation can be a boring job and, traditionally, only others reap the benefits of that effort. The tools available for Python, however, make it almost trivial to generate useful and up-to-date documentation with little to no effort at all. Generating documentation has actually become so easy that I often create and generate documentation before using a Python package. Assuming it wasn't available already, that is.

In addition to simple text documentation explaining what a function does, it is also possible to add metadata, such as type hints. These type hints can be used to make the arguments and return types of a function or class clickable in the documentation. But more importantly, many modern IDEs and editors, such as VIM, have plugins available that parse the type hints and use them for intelligent autocompletion. So if you type `'some_string.'`, your editor will automatically complete the specific attributes and methods of a string object, something that is traditionally only viable with statically typed languages such as Java, C, and C++.

This chapter will explain the types of documentation available in Python and how easily a full set of documentation can be created. With the amazing tools that Python provides, you can have fully functioning documentation within minutes.

The topics covered in this chapter are as follows:

- Type hinting
- The reStructuredText syntax
- The Markdown syntax
- Setting up documentation using Sphinx
- Sphinx-, Google-, and NumPy-style docstrings

Type hinting

Since Python 3.5, we've had a feature called type hinting, which is arguably one of the most useful additions to Python 3. It allows you to specify the types of variables and return values, which means your editor will be able to give you smart autocompletion. This makes it useful for all Python programmers, regardless of level, and can make your life much easier when paired with a good editor.

Basic example

Most editors are already smart enough to recognize basic types in regular variables such as these:

```
>>> a = 123
>>> b = 'test'
>>> c = True
```

It becomes a lot harder for an editor when, instead of `a = 123`, we have something like `a = some_function()`. In some cases, the return type of a function is obvious (i.e. `return True`), but if the return type depends on the input variables or is not consistent, it becomes much harder for the editor to understand what is happening.

As the Zen of Python tells us, explicit is better than implicit. In the case of function return types, this is often the case and can be implemented with very little effort:

```
>>> def pow(base: int, exponent: int) -> int:
...     return base ** exponent

>>> help(pow)
Help on function pow in module __main__:
<BLANKLINE>
pow(base: int, exponent: int) -> int
<BLANKLINE>

>>> pow.__annotations__
{'base': <class 'int'>,
 'exponent': <class 'int'>,
 'return': <class 'int'>}

>>> pow(2, 10)
1024
>>> pow(pow(9, 2) + pow(19, 2) / 22, 0.25)
3.1415926525826463
```

That works as expected. With a simple `-> type`, you can specify the function return type, which is automatically reflected in the `__annotations__`, which is also visible in the `help()`. And the arguments (and variables) can be type-specified using `name: type`.

In this case, you may notice that even though we specified the function to return an `int`, it can actually return a `float` as well, since Python only has type hints, not type constraints/enforcements.

While basic types such as `int`, `float`, `str`, `dict`, `list`, and `set` can be specified with `variable: int` alone, for more advanced types, we need the `typing` module.

Since Python 3.9, you can use `variable: list[int]`. For older versions of Python, you need to use `variable: typing.List[int]` for all collection types such as `dict`/`list`/`set` that require the getitem ([]) operator.

The `typing` module contains types such as `typing.Any` to allow everything, `typing.Optional` to allow for `None`, and `typing.Union` to specify multiple allowed types, which we will now demonstrate:

```
>>> import typing

>>> int_or_float = typing.Union[int, float]

>>> def pow(base: int, exponent: int) -> int_or_float:
...     return base ** exponent

>>> help(pow)
Help on function pow in module __main__:
<BLANKLINE>
pow(base: int, exponent: int) -> Union[int, float]
<BLANKLINE>
```

With `typing.Union`, we can specify a list of types that apply. Similarly, an optional type can be specified using `typing.Optional[int]` to indicate that the type can be either `int` or `None`, effectively being equivalent to `typing.Union[int, None]`. Additionally, since Python 3.10 we can write this as `int | None`.

Custom types

Since regular Python objects are their own type, you usually don't even have to think about what type they are. Simply specify the object and it will work:

```
>>> class Sandwich:
...     pass

>>> def get_sandwich() -> Sandwich:
...     return Sandwich()
```

But what would happen with circular definitions or other circumstances where you do not have the type available yet? In that case, you can work around the issue by specifying the type as a string:

```
>>> class A:
...     @staticmethod
...     def get_b() -> 'B':
...         return B()

>>> class B:
...     @staticmethod
...     def get_a() -> A:
...         return A()
```

Whenever possible, I would recommend against this method because it gives you no guarantee that the type can actually be resolved:

```
# Works without an issue
>>> some_variable: 'some_non_existing_type'

# Error as expected
>>> some_variable: some_non_existing_type
Traceback (most recent call last):
    ...
NameError: name 'some_non_existing_type' is not defined
```

Naturally, this will only check whether the type actually exists. For proper type checking, we can use tools such as `mypy`, which will be covered in the next section. To make sure that your type checker can resolve the type, you can encase your imports in an `if typing.TYPE_CHECKING` block like so:

```
>>> if typing.TYPE_CHECKING:
...     # Add your import for some_non_existing_type here
...     ...
```

The `typing.TYPE_CHECKING` constant is not normally set, but can be set by type checkers such as `mypy` to make sure all types are working correctly.

In the examples above, we have seen custom classes as custom types, but what if we want to create a custom type out of an existing built-in type? That is also possible using `typing.NewType`, which creates a new type that behaves like the base type, but can be checked by static type checkers:

```
>>> import typing

>>> Username = typing.NewType('Username', str)

>>> rick = Username('Rick')

>>> type(rick)
<class 'str'>
```

Here we created a type called `Username`, which is treated as a subclass of `str` in this case.

Generics

In some cases, you don't want to statically specify the type of a function, but make it depend on the input instead. For this reason, the Python type system supports generics. If you're familiar with Java, C++, or C#, you might be familiar with them.

Generics allow you to create a generic type whose only constraint is that it is the same in all cases. This means that if you specify a generic type as both the input and the output for a function, it will be assumed to be the same; if you input an `int` into a function, you will receive an `int`.

First, we need to specify a generic type and, after that, we can specify it as parameters for our functions:

```
>>> import typing

>>> T = typing.TypeVar('T', int, str)

>>> def add(a: T, b: T) -> T:
...     return a + b

>>> add(1, 2)
3
>>> add('a', 'b')
'ab'
```

In this case, we created a generic type with the constraint that it needs to be either `int` or `str`. When the type checker runs, it will check if `a`, `b`, and the `return` value have the same type. This means that even though an `int` is valid for type `T`, if you make `a` a `str`, `b` and the output have to be `str` as well.

Type checking

Now that we know how to specify and create type hints, it's time to run a type checker. The reference implementation for type checking is the `mypy` tool. It can thoroughly check your code and warn about potential problems.

First, we need to install `mypy` – luckily, that's easy enough with `pip`:

```
$ pip3 install -U mypy
```

Now we will use `mypy` to check some of the earlier examples with a few errors added:

```
import typing

def pow(base: int, exponent: int) -> int:
    return base ** exponent

pow(2.5, 10)
```

Since we hinted `base` to be an `int`, `2.5` is not a valid value since it is a `float`:

```
$ mypy T_01_type_hinting.py
T_01_type_hinting.py:8: error: Argument 1 to "pow" has incompatible type
"float"; expected "int"
```

Now an example with a custom type:

```
Username = typing.NewType('Username', str)

rick = Username('Rick')

def print_username(username: Username):
    print(f'Username: {username}')

print_username(rick)
print_username(str(rick))
```

Here we specified that `print_username()` should receive a `Username` type. Even though `Username` inherits `str`, it is not considered valid:

```
$ mypy T_01_type_hinting.py
T_01_type_hinting.py:22: error: Argument 1 to "print_username" has incompatible
type "str"; expected "Username"
```

Lastly, we will create a generic type:

```
T = typing.TypeVar('T')

def to_string(value: T) -> T:
    return str(value)

to_string(1)
```

Since `to_string()` received an `int`, it should return an `int`, which is not the case. Let's run `mypy` to see what's wrong:

```
error: Incompatible return value type (got "str", expected "T")
```

While writing code, `mypy` can save you a lot of debugging by warning you about incorrect type usage.

Python type interface files

Python type hint files (`.pyi`), also called stub files, are files that allow you to specify all type hints for a file without touching the original file. This is useful for libraries that you do not have write access to, or if you do not want to clutter your files with type hints.

The files use the regular Python syntax, but the functions are not meant to contain anything beyond stubs that only hint the types. An example stub for the `print_username()` function mentioned above could be:

```
import typing

Username = typing.NewType('Username', str)

def print_username(username: Username): ...
```

The files are nothing special, but they can be especially useful when interacting with libraries that lack type hinting. If your regular file is named `test.py`, the `pyi` file would be named `test.pyi`.

Type hinting conclusion

Within this section, you have seen a few very basic examples of how type hinting can be applied and how the types can be checked. The Python `typing` module is still getting enhanced quite a lot and `mypy` has really extensive documentation that can be useful if you are applying this to your own code. Make sure to look at the documentation if you have any specific issues; it is high quality and very useful.

When it comes to using type hinting in your own projects, my suggestion is to use it wherever it enhances your workflow but not to go overboard. In many cases, your editor will be smart enough to figure out the arguments automatically, or it won't really matter too much. But when passing along more advanced classes where you tend to forget the methods available for that class, it becomes a really useful feature. Having smart autocompletion can really save you a lot of time.

Now that we have type hints covered, it is time to continue with documenting our code and the markup languages available for that task.

reStructuredText and Markdown

The **reStructuredText** format (also known as **RST**, **ReST**, or **reST**) was developed in 2002 as a language that implements enough markup to be usable, but is simple enough to be readable as plain text. These two features make it readable enough to use in code, yet still versatile enough to generate pretty and useful documentation.

The Markdown format is really similar to reStructuredText and largely comparable. While reStructuredText is slightly older (2012) than Markdown (2014), the Markdown format has gained a bit more popularity because it's a bit simpler and less Python-focused. Both standards are excellent for writing text that is legible straightaway and can easily be converted to other formats such as HTML or PDF files.

The main advantages of reST are:

- A very extensive feature set
- A strictly defined standard
- Easy extensibility

The main advantages of Markdown are:

- It is less Python-centric, which caused it to gain more widespread adoption
- A more forgiving and less strict parser, which makes it easier to write

The greatest thing about both reStructuredText and Markdown is that they are very intuitive to write and natively supported by most (social) coding platforms such as GitHub, GitLab, BitBucket, and PyPI.

Even without knowing anything about the standard, you can easily write documentation in this style. However, more advanced techniques, such as images and links, do require some explanation.

For Python documentation itself, reStructuredText is the most convenient standard since it's well supported by tools such as Sphinx and docutils. For readme files on sites such as GitHub and the Python Package Index, the Markdown standard is generally better supported.

To easily convert between formats such as reStructuredText and Markdown, use the Pandoc tool, available at `https://pandoc.org/`.

The basic syntax reads just like text and the next few paragraphs will show some of the more advanced features. However, let us start with a simple example demonstrating how simple a reStructuredText or Markdown file can be:

```
Documentation, how to use Sphinx and reStructuredText
#####################################################################

Documenting code can be both fun and useful! ...

Additionally, adding ...

... So that typing 'some_string.' will automatically ...

Topics covered in this chapter are as follows:

 - The reStructuredText syntax
 - Setting up documentation using Sphinx
 - Sphinx style docstrings
 - Google style docstrings
 - NumPy style docstrings

The reStructuredText syntax
**********************************************************************

The reStructuredText format (also known as ...
```

That's how easy it is to convert the text of this chapter so far to reStructuredText or Markdown. The example above works in both. But for the Markdown file to look similar, we need to modify the headers slightly:

```
# Documentation, how to use Sphinx and reStructuredText

...

## The reStructuredText syntax

...
```

The following paragraphs will cover the following features:

1. Inline markup (italic, bold, code, and links)
2. Lists
3. Headers
4. Advanced links
5. Images
6. Substitutions
7. Blocks containing code, math, and others

Getting started with reStructuredText

To quickly convert a reStructuredText file to HTML, we can use the `docutils` library. The `sphinx` library discussed later in this chapter actually uses the `docutils` library internally, but has some extra features that we won't need initially. To get started, we just need to install `docutils`:

```
$ pip3 install docutils
```

After that, we can easily convert reStructuredText into PDF, LaTeX, HTML, and other formats. For the examples in this paragraph, we'll use the HTML format, which is easily generated using the following command:

```
$ rst2html.py file.rst file.html
```

The reStructuredText language has two basic components:

- Roles that allow for **inline** modifications of the output, such as `:code:`, `:math:`, `:emphasis:`, and `:literal:`.
- Directives that generate markup **blocks**, such as code samples with multiple lines. These look like this:

  ```
  .. code:: python

    print('Hello world')
  ```

Within pure reStructuredText, the directives are the most important, but we will see many uses for the roles in the section on *Sphinx roles* later in this chapter.

Getting started with Markdown

To quickly convert a Markdown file to HTML we have many options available. But, because we are using Python, we will use the `markdown` package:

```
$ pip3 install markdown
```

Now we can convert our file to HTML with the following command:

```
$ markdown_py file.md -f file.html
```

It should be noted that this converter only supports plain Markdown, not the GitHub flavored Markdown, which also supports code syntax highlighting.

The `grip` (GitHub Readme Instant Preview) Python package supports live rendering of GitHub flavored Markdown by using the GitHub servers and can be useful while writing Markdown.

Inline markup

Inline markup is the markup that is used within a regular line of text. Examples of these are emphasis, inline code examples, links, images, and bullet lists.

Within reStructuredText, these are implemented through roles, but often have useful shorthands. Instead of `:emphasis:'text'`, you can also use `*text*`.

Emphasis, for example, can be added by encapsulating the words between one or two asterisk signs. This sentence, for example, could add a little bit of `*emphasis*` by adding a single asterisk on both sides, or a lot of `**emphasis**` by adding two asterisks on both sides. There are many different inline markup directives so we will list only the most common ones. A full list can always be found through the reStructuredText home page at `https://docutils.sourceforge.io/docs/` and the Markdown home page at `https://daringfireball.net/projects/markdown/syntax`, respectively.

The following are some examples that work for both reST and Markdown:

- Emphasis (italic) text: `*emphasis for this phrase*`.
- Extra emphasis (bold) text: `**extra emphasis for this phrase**`.
- For lists without numbers, a simple dash with a space after it:
    - `- item 1`

    - `- item 2`

Note

The space after the dash is required for reStructuredText to recognize the list.

- For lists with numbers, the number followed by a period and a space:
    - `1. item 1`
    - `2. item 2`
- For numbered lists, the period after the number is required.
- Interpreted text: These are domain-specific. Within Python documentation, the default role is code, which means that surrounding text with backticks will convert your code to use code tags, for example, `'if spam and eggs:'`.
- Inline literals: This is formatted with a monospace font, which makes it ideal for inline code. Just add two backticks to `''add some code''`. For Markdown, there is no noticeable difference between single and double backticks in output, but it can be used to escape single backticks: `''some code ' with backticks''`.
- Escaping in reST can be done using a \, similar to escaping in Python: `''some code \' with backticks''`.

For reStructuredText, there are a few extra options using roles, similar to the interpreted text role we saw earlier. These roles can be set through role prefixes or suffixes depending on your preference; for example, `:math:'E=mc^2'` to show mathematical equations.

References can be added through a trailing underscore. They can point to headers, links, labels, and more. The next section will cover more about these, but the basic syntax is simply `reference_`, or enclosed in backticks when the reference contains spaces – `'some reference link'_`.

There are many more available, but these are the ones you will use the most when writing reStructuredText.

Headers

The headers are used to indicate the start of a document, section, chapter, or paragraph. It is therefore the first structure you need in a document. While not strictly needed, its usage is highly recommended as it serves several purposes:

1. The headers are consistently formatted according to their level.
2. A table of contents (TOC) tree can be generated from the headers.
3. All headers automatically function as labels, which means you can create links to them.

The format required to make headers overlaps a little between reST and Markdown, but for clarity, we will cover them separately.

Headers with reStructuredText

When creating headers, consistency is one of the few constraints; the number of characters used is fairly arbitrary, as is the number of levels.

Personally, I default to a simple system with a fixed-size header, but I recommend at least following the default of the Python documentation in terms of the parts, chapters, sections, subsections, sub-subsections, and paragraphs, something along the lines of the following:

```
Part
######################################################################

Chapter
**********************************************************************

Section
======================================================================

Subsection
----------------------------------------------------------------------

Subsubsection
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Paragraph
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""

Content
```

This creates the following output:

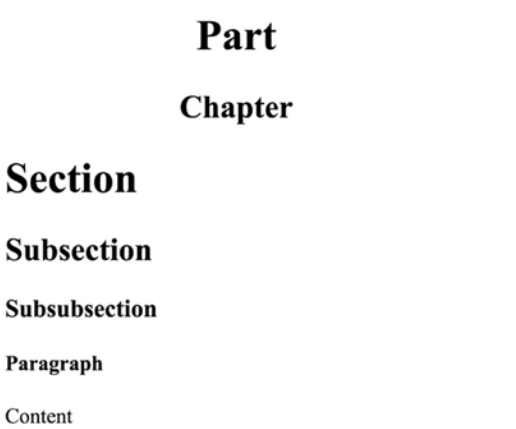


Figure 9.1: Headers with reStructuredText

That is just the common usage of the headers, but the main idea of reStructuredText is that you can use just about anything that feels natural to you, which means that you can use any of the following characters: `= - ' : ' " ~ ^ _ * + # <>`. It also supports both underlines and overlines, so if you prefer that, they are options as well:

```
######################################################################
Part
######################################################################

**********************************************************************
Chapter
**********************************************************************

======================================================================
Section
======================================================================

----------------------------------------------------------------------
Subsection
----------------------------------------------------------------------

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Subsubsection
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
Paragraph
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""

Content
```

While I try to keep the number of characters fixed to 78 characters as `PEP8` (*Chapter 3, Pythonic Syntax and Common Pitfalls*) recommends for Python, the number of characters used is mostly arbitrary, but it does have to be at least as long as the text of the header. This allows it to accept the following result:

```
Section
=======
```

But not this:

```
Section
====
```

Headers with Markdown

With Markdown, you have several options for headers depending on what you feel like. Similar to reST, you can use the = and – characters to underline, but only those, and the length and blank lines after them do not matter:

```
Part
=
Chapter
-
```

If you want more levels, you can use up to 6 levels by using the # prefix and optional suffixes:

```
# Part
## Chapter
### Section
#### Subsection
##### Subsubsection
###### Paragraph
Content
###### Paragraph with suffix ######
Content
```

This results in:

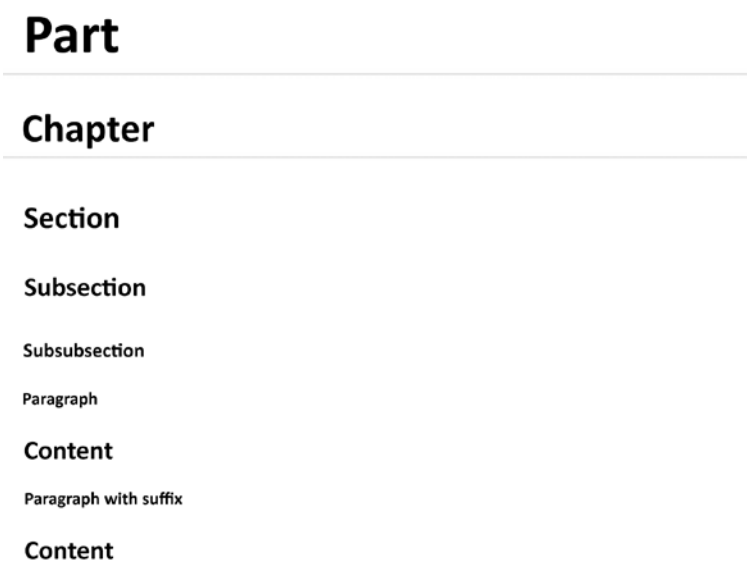


Figure 9.2: Headers in Markdown

As you can see, Markdown is slightly less flexible than reStructuredText when it comes to headers, but in most cases, it offers enough features to be perfectly usable.

Lists

The reStructuredText format has several styles of lists:

1. Enumerated
2. Bulleted
3. Options
4. Definitions

The simplest forms of lists were already displayed in the introduction section, but it's actually possible to use many different characters, such as letters, Roman numerals, and others, for enumeration. After demonstrating the basic list types, we will continue with the nesting of lists and structures, which makes them even more powerful. Care must be taken with the amount of whitespace, as one space too many can cause a structure to be recognized as regular text instead of a structure.

Enumerated lists

Enumerated lists are convenient for all sorts of enumerations. The basic premise for enumerated lists is an alphanumeric character followed by a period, a right parenthesis, or parentheses on both sides. Additionally, the # character functions as an automatic enumeration. For example:

```
1. With
2. Numbers

a. With
#. letters

i. Roman
#. numerals

(1) With
(2) Parenthesis
```

The output is perhaps a bit simpler than you would expect. The reason is that it depends on the output format. The following figure shows the rendered HTML output, which has no support for parentheses. If you output LaTeX, for example, the difference can be made visible.

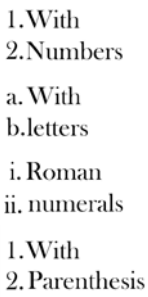


Figure 9.3: Enumerated lists generated with the HTML output format

Markdown also supports enumerated lists, but it is a bit more limited in its options. It only supports regular numbered lists. It's more convenient in how it supports them though; there is no need for explicit numbering, and repeating `1.` works without a problem:

```
1. With
1. Numbers
```

Bulleted lists

If the order of the list is not relevant and you simply need a list of items without enumeration, then the bulleted list is what you should use. To create a simple list using bullets only, the bulleted items need to start with a `*`, `+`, `-`, `•`, `‣`, or `⁃`. This list is mostly arbitrary and can be modified by extending Sphinx or Docutils. For example:

```
- dashes
- and more dashes

* asterisk
* stars

+ plus
+ and plus
```

As you can see in the following figure, with the HTML output, all bullets again look identical.

When generating documentation as LaTeX (and consecutively, PDF or Postscript), these can differ.

Since web-based documentation is by far the most common output format for Sphinx, we default to that output instead. The rendered HTML output is as follows:

- dashes
- and more dashes
- asterisk
- stars
- plus
- and plus

Figure 9.4: Bulleted lists with HTML output

As you can see, all bulleted lists are rendered the same in this case. This is dependent on the renderer, however, so it's a good idea to check the output to see if it matches your preference.

Option lists

The `option` list is one meant specifically for documenting the command-line arguments of a program. The only special thing about the syntax is that the comma space is recognized as a separator for options:

```
-s, --spam  This is the spam option
--eggs      This is the eggs option
```

The following is the output:

-s, --spam	This is the spam option
--eggs	This is the eggs option

Figure 9.5: Option list

In Markdown, there is no support for option lists, but you can achieve similar results by creating a table:

```
| Argument     | Help                    |
|--------------|-------------------------|
| '-s, --spam' | This is the spam option |
| '--eggs'     | This is the eggs option |
```

Note that in most Markdown implementations, the headers for a table are required. But the header alignment as is done here is optional, and the following would render the same:

```
| Argument | Help |
|-|-|
| '-s, --spam' | This is the spam option |
| '--eggs' | This is the eggs option |
```

Definition lists (reST only)

The definition list is a bit more obscure than the other types of lists, since the actual structure consists of whitespace only. It's therefore pretty straightforward to use, but not always as easy to identify in a file, and it is only supported by reST:

```
spam
    Spam is a canned pork meat product
eggs
    Is, similar to spam, also food
```

The following is the output:

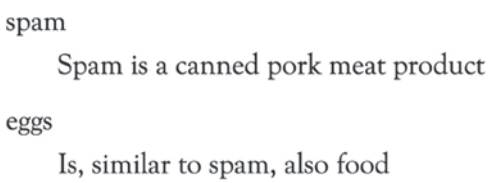


Figure 9.6: Definitions list

The definition list is especially useful when explaining the meaning of certain keywords in your documentation.

Nested lists

Nesting items is actually not limited to lists and can be done with multiple types of blocks, but the idea is the same. You could nest a code block within a bulleted list, for example. Just be careful to keep the indenting at the correct level. If you don't, it either won't be recognized as a separate level or you will get an error:

```
1. With
2. Numbers

   (food) food

    spam
        Spam is a canned pork meat product
```

```
    eggs
        Is, similar to spam, also food

    (other) non-food stuff
```

The following figure shows the output:

1. With

2. Numbers

(food) food

spam
Spam is a canned pork meat product

eggs
Is, similar to spam, also food

(other) non-food stuff

Figure 9.7: Nested lists

For Markdown, the same kind of nesting is possible, as long as the right list types are used.

Links, references, and labels

The links syntax is quite different between Markdown and reStructuredText, but they offer similar features. Both support inline links and links using a list of references.

The simplest links with protocols such as `http://python.org` will automatically be recognized by most parsers for both Markdown and reStructuredText. For custom labels, the syntax is a bit different:

- reStructuredText: `'Python <http://python.org>'_`
- Markdown: `[Python](http://python.org)`

Both of these are nice for simple links that won't be repeated too often, but generally, it's more convenient to attach labels to links so they can be reused and don't clog up the text too much.

For example, refer to the following reStructuredText example:

```
The switch to reStructuredText and Sphinx was made with the
'Python 2.6 <https://docs.python.org/whatsnew/2.6.html>'_
release.
```

Now compare it with the following:

```
The switch to reStructuredText and Sphinx was made with the
'python 2.6'_ release.
```

```
.. _'Python 2.6': https://docs.python.org/whatsnew/2.6.html
```

The output is as follows:

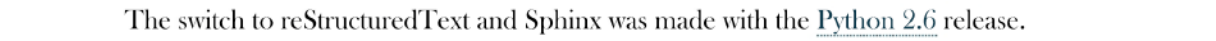


Figure 9.8: Link with a custom label

And the Markdown equivalents:

```
The switch to reStructuredText and Sphinx was made with the [Python 2.6]
(https://docs.python.org/whatsnew/2.6.html) release.

The switch to reStructuredText and Sphinx was made with the [Python 2.6]
release.

[Python 2.6]: https://docs.python.org/whatsnew/2.6.html
```

Using labels, you can easily have a list of references at a designated location without making the actual text harder to read.

For reStructuredText, these labels can be used for more than external links, however. Similar to the `GOTO` statements found in older programming languages, you can create labels and refer to them from other parts of the documentation:

```
.. _label:
```

Within HTML or PDF output, this can be used to create a clickable link from anywhere in the text using the underscore links. Creating a clickable link to the label is as simple as having `label_` in the text.

Note that reStructuredText ignores case differences, so both uppercase and lowercase links work just fine. Even though we're not likely to make this mistake, having the same label in a single document with only case differences results in an error, to make sure that duplicates never occur.

The usage of references in conjunction with the headers works in a very natural way; you can just refer to them as you normally would and add an underscore to make it a link:

```
The introduction section
================================================================

This section contains:

- 'chapter 1'_
- :ref:'chapter2'

  1. my_label_

  2. 'And a label link with a custom title <my_label>'_
```

```
Chapter 1
---------------------------------------------------------------

Jumping back to the beginning of 'chapter 1'_ is also possible.
Or jumping to :ref:'Chapter 2 <chapter2>'

.. _chapter2:

Chapter 2 With a longer title
---------------------------------------------------------------

The next chapter.

.. _my_label:

The label points here.

Back to 'the introduction section'_
```

The output is as follows:

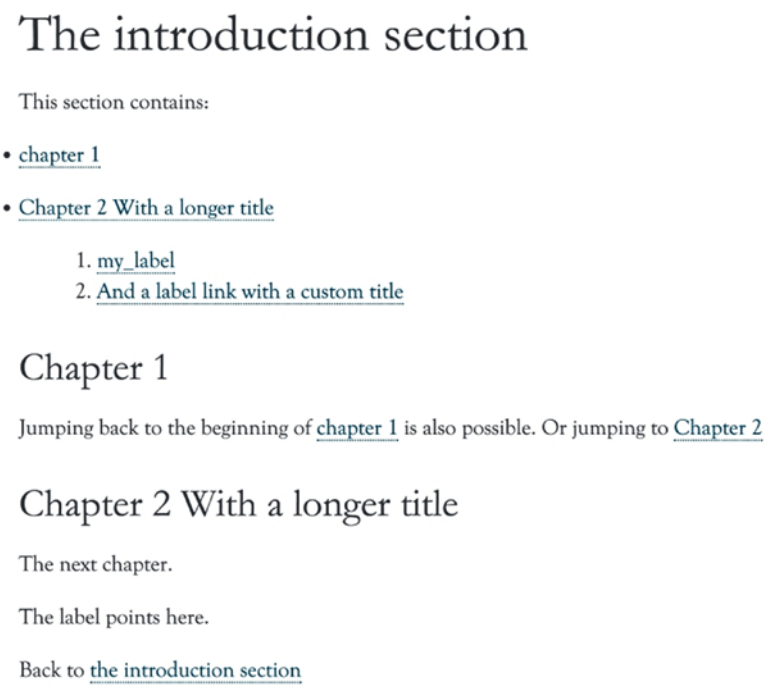


Figure 9.9: Links, labels, and references

For Markdown, you can partially get similar results depending on the renderer that is used. In the case of the GitHub parser, all headers are automatically converted to HTML anchors, so a header like `# Some header` can be linked to by using `[name of the link](#some-header)`.

While this method is convenient for simple cases, it comes with a number of drawbacks:

- When the header changes, all links to it are broken
- When multiple headers have the same name, only the first one can be linked to
- Only headers can be linked to

Images

Images is a feature that is implemented quite differently between reStructuredText and Markdown.

Images with reStructuredText

In reStructuredText, the image directive looks very similar to the label syntax. They're actually a bit different, but the pattern is quite similar. The image directive is just one of the many directives that are supported by reStructuredText. We will see more about that later on when we cover Sphinx and reStructuredText extensions. For the time being, it is enough to know that the directives start with two periods followed by a space, the name of the directive, and two colons:

```
.. name_of_directive::
```

In the case of the image, the directive is called `image` of course:

```
.. image:: python.png
```

Here is the scaled output, as the actual image is much larger:

Figure 9.10: Image output with reStructuredText

Note the double colon after the directives.

But how about specifying the size and other properties? The image directive has many other options (as do most other directives) that can be used: `https://docutils.sourceforge.io/docs/ref/rst/directives.html#images`; they are mostly fairly obvious, however. To specify the width and height or the scale (in percent) of the image:

```
.. image:: python.png
   :width: 150
   :height: 100

.. image:: python.png
   :scale: 10
```

The following is the output:

Figure 9.11: Scaled image with reStructuredText

The `scale` option uses the `width` and `height` options if available and falls back to the PIL (Python Imaging Library) or Pillow library to detect the image. If neither `width`/`height` nor PIL/Pillow are available, the `scale` option will be ignored silently.

In addition to the `image` directive, there is also the `figure` directive. The difference is that `figure` adds a caption to the image. Beyond that, the usage is the same as `image`:

```
.. figure:: python.png
   :scale: 10

   The Python logo
```

The output is as follows:

Figure 9.12: Adding a figure caption with reStructuredText

Now, let's compare what we've just seen with how to deal with images using Markdown.

Images with Markdown

The support for images in Markdown is similar to the support for links, but you need to add a `!` in front of it:

```
![python](python.png)
```

As is the case with links, you can also use references:

```
![python]

[python]: python.png
```

However, changing other properties such as the size is not supported by most Markdown implementations.

Substitutions

When writing documentation, you will often have to use the same images and links over and over again. While you can add those inline, it is often very verbose, tedious, and hard to maintain.

Within reStructuredText pages, we already have the internal labeling system that handles a lot of cases for us. For external links and images, however, we need to use one of the other reStructuredText features. With substitution definitions, you can shorten directives so they can easily be re-used. In the common Markdown implementations, there is no equivalent feature for this.

Let's assume we have a logo that we use quite often within a bit of text. Instead of typing the entire `.. image:: <url>`, it would be very handy to have a shorthand to make it easier. That's where the substitutions are very useful:

```
.. |python| image:: python.png
   :scale: 1

The Python programming language uses the logo: |python|
```

As you can see, you can use the pipe character to create and use substitutions anywhere in your text. As is usual in most languages, you can escape the character with a backslash (\) if you need to use a pipe outside of substitutions.

The output is as follows:

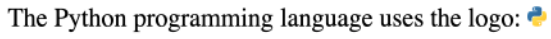


Figure 9.13: Rendered reStructuredText using a substitution for an image directive

These substitutions can be used with many directives, though they are particularly useful for outputting a variable in many places of a document. For example:

```
.. |author| replace:: Rick van Hattem

This book was written by |author|
```

The following is the output:

Figure 9.14: Rendered reStructuredText using a text substitution for an author's name

These types of substitutions are really useful while writing documentation because they make your reStructuredText files more readable, but they also allow you to change your entire documentation by updating a single variable. As opposed to a search/replace, which is generally an error-prone operation.

While writing this chapter, a substitution for `|rest|` to return `reStructuredText` would have been very useful.

Blocks, code, math, comments, and quotes

When writing documentation, a common scenario is the need for blocks that contain different types of content, explanations with mathematical formulae, code examples, and more.

The usage of these directives is similar to the image directive. The following is an example of a code block:

```
.. code:: python

   def spam(*args):
       print('spam got args', args)
```

The output is as follows:

```
def spam(*args):
    print('spam got args', args)
```

Figure 9.15: Code block output

This is one of the cases where Markdown is a bit simpler to use. With plain Markdown, a code block only requires indenting:

```
Code below:

    def spam(*args):
        print('spam got args', args)
```

Or with the GitHub flavored Markdown with syntax highlighting:

```
'''python
def spam(*args):
    print('spam got args', args)
'''
```

With reStructuredText you have more options, however. You can also display mathematical formulae using the LaTeX syntax. Here's the fundamental theorem of calculus, for example:

```
.. math::

    \int_a^b f(x)\,dx = F(b) - F(a)
```

The following is the output:

$$\int_a^b f(x)dx = F(b) - F(a)$$

Figure 9.16: Mathematical formula output

Commenting a bunch of text/commands is easily achieved by using the "empty" directive followed by an indent. Effectively, this means two dots, as is the case with any directive, but with the `directive::` bit omitted:

```
Before comments

.. Everything here will be commented

   And this as well
   .. code:: python
      def even_this_code_sample():
```

```
          pass  # Will be commented

After comments
```

The output is as follows:

Before comments

After comments

Figure 9.17: Output (with hidden comments)

With Markdown, you have no real method of adding comments, but you can use links as a hack around this limitation in a few limited cases:

```
[_]: <> (this will not be shown)
```

While this method works, it is still far from pretty, of course. Often, you are better off moving the contents to a separate scratch file, or removing the content instead of commenting it and using a version control system such as Git to retrieve the data if you need it later.

Quoting text is supported by both reStructuredText and Markdown, but the syntax conflicts. Within reStructuredText, you can create a block quote using indentation, which would result in code formatting in Markdown:

```
Normal text

    Quoted text
```

The output is as follows:

Normal text

Quoted text

Figure 9.18: Quoting text

Within Markdown, the format is comparable to how text-based email clients generally quote replies:

```
Normal text
> Quoted text
```

Conclusion

Both reStructuredText and Markdown are very useful languages for creating some documentation. A large portion of the syntax comes naturally when writing plain text notes. A full guide to all the intricacies of reST, however, could fill a separate book. The previous demonstrations should have given enough of an introduction to do at least 90 percent of the work you will need when documenting your projects. Beyond that, Sphinx will help a lot, as we will see in the next sections.

In general, I would suggest using reStructuredText for actual documentation because it has many more features than Markdown. However, Markdown is generally more convenient for the basic readme files on PyPI and GitHub, mainly because you can use the same readme file for both cases, and GitHub supports Markdown slightly better than reStructuredText.

The Sphinx documentation generator

The Sphinx documentation generator was created in 2008 for the Python 2.6 release to replace the old LaTeX documentation for Python. It's a generator that makes it almost trivial to generate documentation for programming projects, but even outside of the programming world, it can be easily used. Within programming projects, there is specific support for the following domains (programming languages):

- Python
- C
- C++
- JavaScript
- reStructuredText

Outside of these languages, there are extensions available for many other languages, such as CoffeeScript, MATLAB, PHP, Ruby Lisp, Go, and Scala. And if you're simply looking for snippet code highlighting, the Pygments highlighter, which is used internally, supports over 120 languages and is easily extendible for new languages if needed.

The most important advantage of Sphinx is that almost everything can be automatically generated from your source code. The result is that your documentation is always up to date.

Getting started with Sphinx

First of all, we have to make sure we install Sphinx. Even though the Python core documentation is written using Sphinx, it is still a separately maintained project and must be installed separately. Luckily, that's easy enough using pip:

```
$ pip3 install sphinx
```

After installing Sphinx, there are two ways of getting started with a project: the `sphinx-quickstart` script, and the `sphinx-apidoc` script.

If you want to create and customize an entire Sphinx project, then I would recommend the `sphinx-quickstart` command, as it assists you in configuring a fully featured Sphinx project.

If you want to start quickly and generate some API documentation for an **existing** Python project, then `sphinx-apidoc` might be better suited since it takes a single command and no further input to create a project. After running it, you will have fully functioning documentation based on your Python source.

In the end, both are valid options for creating Sphinx projects, and personally I usually end up generating the initial configuration using `sphinx-quickstart` and call the `sphinx-apidoc` command every time I add a Python module to add the new module.

The `sphinx-apidoc` command does not overwrite any files by default, making it a safe operation to run repeatedly.

Using sphinx-quickstart

The `sphinx-quickstart` script interactively asks you about the most important decisions in your Sphinx project. There is no need to worry about typos; the configuration is stored in a `conf.py` file and can be modified like a regular Python file.

Usage is easy enough. As a default, I would recommend creating the documentation in a separate `docs` directory, as is the convention for many projects. The output uses the following conventions:

- Inline comments start with #
- User input lines start with >
- Cropped output is indicated with `...` and all questions skipped in between use the default settings

Note the `docs` after the command:

```
$ sphinx-quickstart docs
Welcome to the Sphinx 3.2.1 quickstart utility.

Please enter values for the following settings (just press Enter to
accept a default value, if one is given in brackets).

Selected root path: docs

You have two options for placing the build directory for Sphinx output.
Either, you use a directory "_build" within the root path, or you separate
"source" and "build" directories within the root path.
> Separate source and build directories (y/n) [n]:

The project name will occur in several places in the built documentation.
> Project name: Mastering Python
> Author name(s): Rick van Hattem
> Project release []:

...
```

You should now populate your master file, `docs/index.rst`, and create other documentation source files. Use the Makefile to build the docs, like so:

```
$ make <builder>
```

Where "<builder>" is one of the supported builders, for example, `html`, `latex`, or `linkcheck`. After running this, we should have a `docs` directory containing the Sphinx project. Let's see what the command actually created for us:

```
$ find docs
docs
docs/index.rst
docs/_templates
docs/Makefile
docs/conf.py
docs/_static
docs/make.bat
docs/_build
```

The `_build`, `_static`, and `_templates` directories are initially empty and can be ignored for now. The `_build` directory is used to output the generated documentation, whereas the `_static` directory can be used to easily include custom CSS files and such. The `_templates` directory makes it possible to style the HTML output to your liking as well. Examples of these can be found in the Sphinx Git repository at `https://www.sphinx-doc.org/en/master/usage/theming.html#builtin-themes`.

`Makefile` and `make.bat` can be used to generate the documentation output. `Makefile` can be used for any operating system that supports the make utility, and `make.bat` is there to support Windows systems out of the box. Now let's look at the `index.rst` source:

```
Welcome to Mastering Python's documentation!
============================================

.. toctree::
   :maxdepth: 2
   :caption: Contents:



Indices and tables
==================

* :ref:'genindex'
* :ref:'modindex'
* :ref:'search'
```

We see the document title as expected, followed by `toctree` (table of contents tree; more about that later in this chapter), and the links to the indices and search. `toctree` automatically generates a tree out of the headers of all available documentation pages.

The indices and tables are automatically generated Sphinx pages, which are very useful, but nothing we need to worry about in terms of settings.

Now it's time to generate the HTML output:

```
$ cd docs
$ make html
```

The `make html` command generates the documentation for you and the result is placed in `_build/html/`. Just open `index.html` in your browser to see the results. You should now have something looking similar to the following:

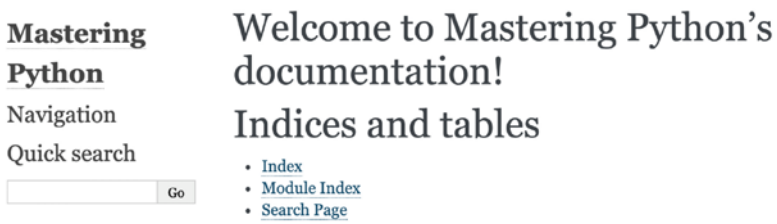


Figure 9.19: Viewing index.html

With just that single command and by answering a few questions, we now have a documentation project with an index, search, and table of contents on all the pages.

In addition to the HTML output, there are quite a few other formats supported by default, although some require external libraries to actually work:

```
$ make help
Sphinx v3.2.1
Please use 'make target' where target is one of
  html        to make standalone HTML files
  dirhtml     to make HTML files named index.html in directories
  singlehtml  to make a single large HTML file
  pickle      to make pickle files
  json        to make JSON files
  htmlhelp    to make HTML files and an HTML help project
  qthelp      to make HTML files and a qthelp project
  devhelp     to make HTML files and a Devhelp project
  epub        to make an epub
  latex       to make LaTeX files, you can set PAPER=a4 or ...
  latexpdf    to make LaTeX and PDF files (default pdflatex)
  latexpdfja  to make LaTeX files and run them through platex/...
  text        to make text files
  man         to make manual pages
  texinfo     to make Texinfo files
```

```
  info         to make Texinfo files and run them through makeinfo
  gettext      to make PO message catalogs
  changes      to make an overview of all changed/added/... items
  xml          to make Docutils-native XML files
  pseudoxml    to make pseudoxml-XML files for display purposes
  linkcheck    to check all external links for integrity
  doctest      to run all doctests embedded in the documentation
  coverage     to run coverage check of the documentation
```

Using sphinx-apidoc

The `sphinx-apidoc` command is generally used together with `sphinx-quickstart`. It is possible to generate an entire project with the `--full` parameter, but it's generally a better idea to generate the entire project using `sphinx-quickstart` and simply add the API documentation using `sphinx-apidoc`.

To properly demonstrate the `sphinx-apidoc` command, we need some Python files, so we'll create two files within a project called `apidoc_example`.

The first one is `apidoc_example/a.py`, containing a class called `A` with some methods:

```
class A(object):
    def __init__(self, arg, *args, **kwargs):
        pass

    def regular_method(self, arg):
        pass

    @classmethod
    def decorated_method(self, arg):
        pass

    def _hidden_method(self):
        pass
```

Next, we have `apidoc_example/b.py` containing a `B` class that inherits `A`:

```
from . import a

class B(a.A):
    def regular_method(self):
        '''This regular method overrides
        :meth:'a.A.regular_method'
        '''
        pass
```

Now that we have our source files, it's time to generate the actual API documentation:

```
$ sphinx-apidoc apidoc_example -o docs
Creating file docs/apidoc_example.rst.
Creating file docs/modules.rst.
```

This alone is not enough to include the API in the documentation. It needs to be added to `toctree`. Luckily, that's as simple as adding modules to `toctree` in the `index.rst` file to look something like this:

```
.. toctree::
   :maxdepth: 2

   modules
```

The `toctree` directive is discussed in further detail later in this chapter.

We also have to make sure that the modules can be imported, otherwise Sphinx won't be able to read the Python files. To do that, we simply add the parent directory (as seen from the `docs` directory) to `sys.path`; this can be put anywhere in the `conf.py` file:

```
import os
import sys

sys.path.insert(0, os.path.abspath('..'))
```

Additionally, the `autodoc` module needs to be enabled in `conf.py`:

```
extensions = [
    'sphinx.ext.autodoc',
]
```

Now it's time to generate the documentation again by using the `html` builder:

```
$ make html
Running Sphinx v3.2.1
making output directory... done
building [mo]: targets for 0 po files that are out of date
building [html]: targets for 3 source files that are out of date
updating environment: [new config] 3 added, 0 changed, 0 removed
reading sources... [100%] modules
looking for now-outdated files... none found
pickling environment... done
checking consistency... done
preparing documents... done
writing output... [100%] modules
generating indices...  genindex py-modindexdone
writing additional pages...  searchdone
```

```
copying static files... ... done
copying extra files... done
dumping search index in English (code: en)... done
dumping object inventory... done
build succeeded.

The HTML pages are in _build/html.
```

Open the `docs/_build/index.html` file again. For the sake of brevity, the repeated parts of the document will be omitted from the screenshots. The cropped output is as follows:

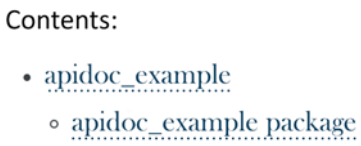


Figure 9.20: Viewing the Contents

But it actually generated quite a bit more. When running the `sphinx-apidoc` command, it looks at all the Python modules in the specified directory recursively and generates an `rst` file for each of them. After generating all those files, it adds all of them to a file called `modules.rst`, which makes it easy to add them to your documentation.

The `modules.rst` file is really straight to the point; nothing more than a list of modules with the package name as the title:

```
apidoc_example
==============

.. toctree::
   :maxdepth: 4

   apidoc_example
```

The `apidoc_example` page output is as follows:

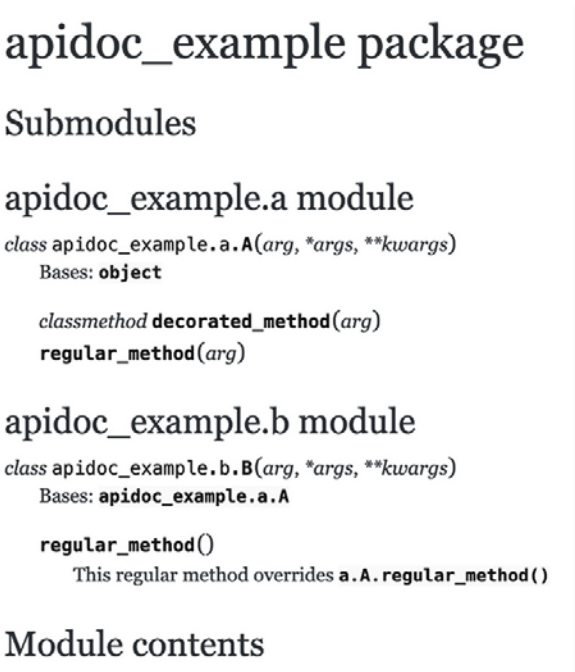


Figure 9.21: The apidoc_example page

The `apidoc_example.rst` file simply lists all the documented modules in `automodule` directives with a few settings:

```
apidoc\_example package
=======================

Submodules
----------

apidoc\_example.a module
------------------------
```

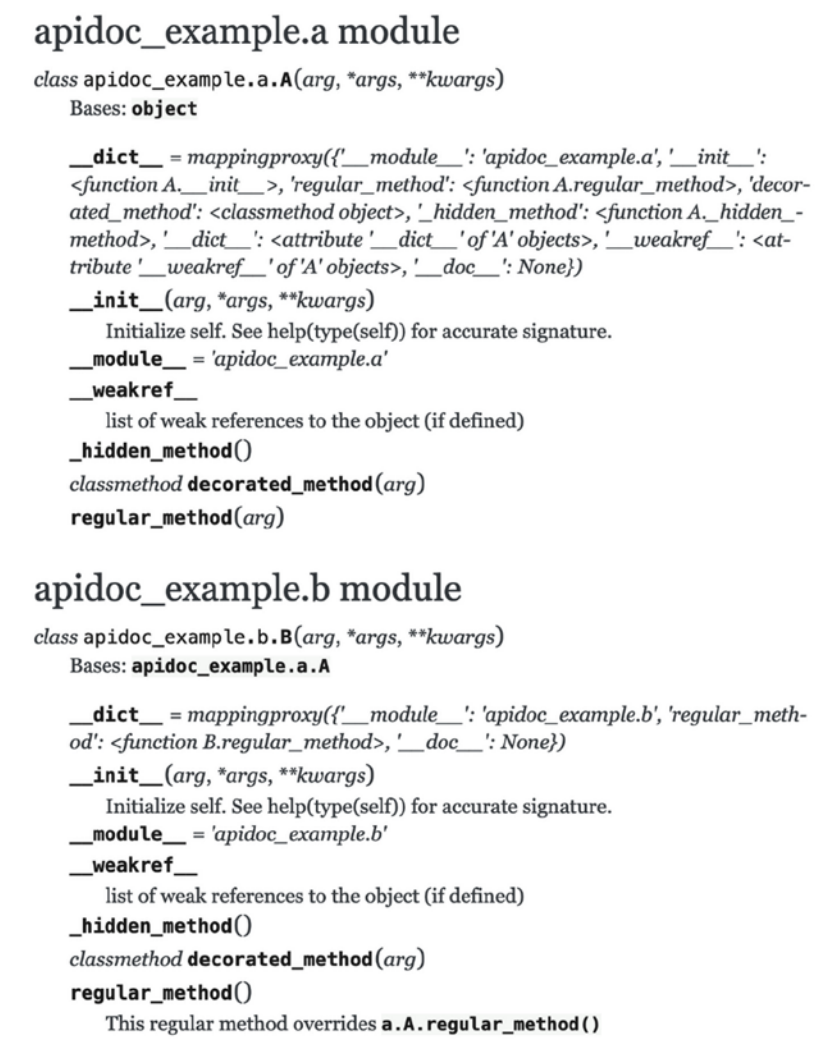
```
.. automodule:: apidoc_example.a
   :members:
   :undoc-members:
   :show-inheritance:

apidoc\_example.b module
------------------------

.. automodule:: apidoc_example.b
   :members:
   :undoc-members:
   :show-inheritance:

Module contents
---------------

.. automodule:: apidoc_example
   :members:
   :undoc-members:
   :show-inheritance:
```

But as you have seen in the previous screenshot, it does not include hidden or magic methods. By adding some extra arguments to the `automodule` directive, we can change this:

```
apidoc\_example package
=======================

Submodules
----------

apidoc\_example.a module
```

```
------------------------

.. automodule:: apidoc_example.a
   :members:
   :undoc-members:
   :show-inheritance:
   :private-members:
   :special-members:
   :inherited-members:

apidoc\_example.b module
------------------------

.. automodule:: apidoc_example.b
   :members:
   :undoc-members:
   :show-inheritance:
   :private-members:
   :special-members:
   :inherited-members:

Module contents
---------------

.. automodule:: apidoc_example
   :members:
   :undoc-members:
   :show-inheritance:
   :private-members:
   :special-members:
   :inherited-members:
```

With these extra settings (`private-members`, `special-members`, and `inherited-members`), we get a lot of extra and arguably useful documentation:



apidoc_example.a module

class apidoc_example.a.A(*arg*, **args*, ***kwargs*)

Bases: object

__dict__ = *mappingproxy({'__module__': 'apidoc_example.a', '__init__': <function A.__init__>, 'regular_method': <function A.regular_method>, 'decorated_method': <classmethod object>, '_hidden_method': <function A._hidden_method>, '__dict__': <attribute '__dict__' of 'A' objects>, '__weakref__': <attribute '__weakref__' of 'A' objects>, '__doc__': None})*

__init__(*arg*, **args*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

__module__ = *'apidoc_example.a'*

__weakref__

list of weak references to the object (if defined)

_hidden_method()

classmethod decorated_method(*arg*)

regular_method(*arg*)

apidoc_example.b module

class apidoc_example.b.B(*arg*, **args*, ***kwargs*)

Bases: apidoc_example.a.A

__dict__ = *mappingproxy({'__module__': 'apidoc_example.b', 'regular_method': <function B.regular_method>, '__doc__': None})*

__init__(*arg*, **args*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

__module__ = *'apidoc_example.b'*

__weakref__

list of weak references to the object (if defined)

_hidden_method()

classmethod decorated_method(*arg*)

regular_method()

This regular method overrides a.A.regular_method()

Figure 9.22: The updated apidoc_example page

Which of these settings are useful for you depends on your use case, of course. But it shows how easily we can generate full documentation for classes with barely any effort. And all references such as the bases and the overridden methods are clickable as well.

New files won't be added to your docs automatically. It is safe to rerun the `sphinx-apidoc` command to add the new files, but it won't update your existing files. Even though the `--force` option can be used to force overwriting the files, within existing files I recommend manually editing them instead. As we will see in the next sections, there are quite a few reasons to manually modify the generated files afterward.

Sphinx directives

Sphinx adds a few directives on top of the default ones in reStructuredText and an easy API to add new directives yourself. Most of them are generally not that relevant to modify but, as one would expect, Sphinx has pretty good documentation in case you need to know more about them.

We have already seen the labels, images, math, substitutions, code, and comment reST directives. But there are also quite a few Sphinx-specific directives. Most of them are not too important to talk about, but perhaps interesting to take a look at: `https://www.sphinx-doc.org/en/master/usage/restructuredtext/directives.html`.

We've already covered the most important one of all, the `autodoc` module, which is used by the `automodule` directive. There is another one that requires a tiny bit of coverage, however: the `toctree` directive. We have already seen it in use earlier, but it has a few interesting configuration options that are really useful for larger projects.

The `toctree` directive is one of the most important directives in Sphinx; it generates the table of contents tree. The `toctree` directive has a couple of options, but the most important one is probably `maxdepth`, which specifies how deep the tree needs to go. The top level of `toctree` has to be specified manually by specifying the files to be read, but beyond that, every level within a document (section, chapter, paragraph, and so on) can be another level in `toctree`, depending on the depth, of course. Even though the `maxdepth` option is optional, without it all the available levels will be shown, which is usually more than required. In most cases, a `maxdepth` of 2 is a good default value, which makes the basic example look like this:

```
.. toctree::
   :maxdepth: 2
```

The items in `toctree` are the `.rst` files in the same directory without the extension. This can include subdirectories, in which case the directories are separated with a `.` (period):

```
.. toctree::
   :maxdepth: 2

   module.a
   module.b
   module.c
```

Another very useful option is the `glob` option. It tells `toctree` to use the `glob` module in Python to automatically add all the documents matching a pattern. By simply adding a directory with a `glob` pattern, you can add all the files in that directory. This makes the `toctree` we had before as simple as:

```
.. toctree::
   :maxdepth: 2
   :glob:

   module.*
```

If, for some reason, the document title is not as you would have liked, you can easily change the title to something customized:

```
.. toctree::
   :maxdepth: 2

   The A module <module.a>
```

Sphinx roles

We have seen Sphinx directives, which are separate blocks. Now we will discuss Sphinx roles, which can be used inline. A role allows you to tell Sphinx how to parse some input. Examples of these roles are links, math, code, and markup. But the most important ones are the roles within the Sphinx domains for referencing other classes, even for external projects. Within Sphinx, the default domain is the Python one, so a role such as `:py:meth:` can be used as `:meth:` as well. These roles are really useful to link to different packages, modules, classes, methods, and other objects. The basic usage is simple enough. To link to a class, use the following:

```
Spam: :class:'spam.Spam'
```

The output is:

Spam: **spam.Spam**

Figure 9.23: Linking to a class

The same goes for just about any other object, functions, exceptions, attributes, and so on. The Sphinx documentation offers a list of supported objects: `https://www.sphinx-doc.org/domains.html#cross-referencing-python-objects`.

One of the nicer features of Sphinx is that these references can extend beyond your project. Similar to how we added a link to the class above, adding a reference to the `int` object in the standard Python documentation is easily possible using `:obj:'int'`. Adding references to your own projects in other documentation sets and on other websites is done in a similar fashion.

For inter-project links, you will need to enable the `intersphinx` module in `conf.py`:

```
extensions = [
    'sphinx.ext.autodoc',
    'sphinx.ext.intersphinx',
]
```

After that, we need to tell `intersphinx` where it can find the documentation of the other projects by adding `intersphinx_mapping` to `conf.py`:

```
intersphinx_mapping = {
    'python': ('https://docs.python.org/', None),
```

```
    'sphinx': ('https://www.sphinx-doc.org/', None),
}
```

Now we can easily link to the documentation on the Sphinx home page:

```
Link to the intersphinx module: :mod:'sphinx.ext.intersphinx'
```

The following is the output:

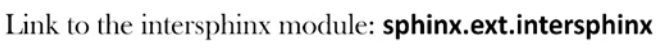


Figure 9.24: Linking to another project

This links to `https://www.sphinx-doc.org/en/master/ext/intersphinx.html`.

Now that we know how to use Sphinx to generate documentation from our code, let's enhance that documentation to be even more useful.

Documenting code

There are currently three different documentation styles supported by Sphinx: the original Sphinx style and the more recent NumPy and Google styles. The differences between them are mainly in style, but it's actually slightly more than that.

The Sphinx style was developed using a bunch of reStructuredText roles, a very effective method, but it can be detrimental for readability when used a lot. You can probably tell what the following does, but it's not the nicest syntax:

```
:param number: The number of eggs to return
:type number: int
```

The Google style was (as the name suggests) developed by Google. The goal was to have a simple/readable format that works both as in-code documentation and is parseable for Sphinx. In my opinion, it comes closer to the original idea of reStructuredText, a format that's very close to how you would document instinctively. This example has the same meaning as the Sphinx style example shown earlier:

```
Args:
    number (int): The number of eggs to return
```

The NumPy style was created specifically for the NumPy project. The NumPy project has many functions, with a huge amount of documentation, and generally a lot of documentation per argument. It is slightly more verbose than the Google format, but quite easy to read as well:

```
Parameters
----------
number : int
    The number of eggs to return
```

With the type hint annotations introduced in Python 3.5, at least the argument type part of these syntaxes has become less useful. Since Sphinx 3.0 you can tell Sphinx to use the type hints instead of manually adding the type by adding this line to your Sphinx `conf.py`:

```
autodoc_typehints = 'description'
```

Documenting a class with the Sphinx style

First of all, let's look at the traditional style, the Sphinx style. While it's easy to understand what all the parameters mean, it's a bit verbose, which reduces the readability somewhat. Nonetheless, the meaning is immediately clear and it is definitely not a bad style to use:

```
class Eggs:
    pass

class Spam(object):
    '''
    The Spam object contains lots of spam

    :param arg: The arg is used for ...
    :type arg: str
    :param '*args': The variable arguments are used for ...
    :param '**kwargs': The keyword arguments are used for ...
    :ivar arg: This is where we store arg
    :vartype arg: str
    '''

    def __init__(self, arg: str, *args, **kwargs):
        self.arg: str = arg

    def eggs(self, number: int, cooked: bool) -> Eggs:
        '''We can't have spam without eggs, so here are the eggs

        :param number: The number of eggs to return
        :type number: int
        :param bool cooked: Should the eggs be cooked?
        :raises: :class:'RuntimeError': Out of eggs

        :returns: A bunch of eggs
        :rtype: Eggs
        '''
        pass
```

The output looks like this:

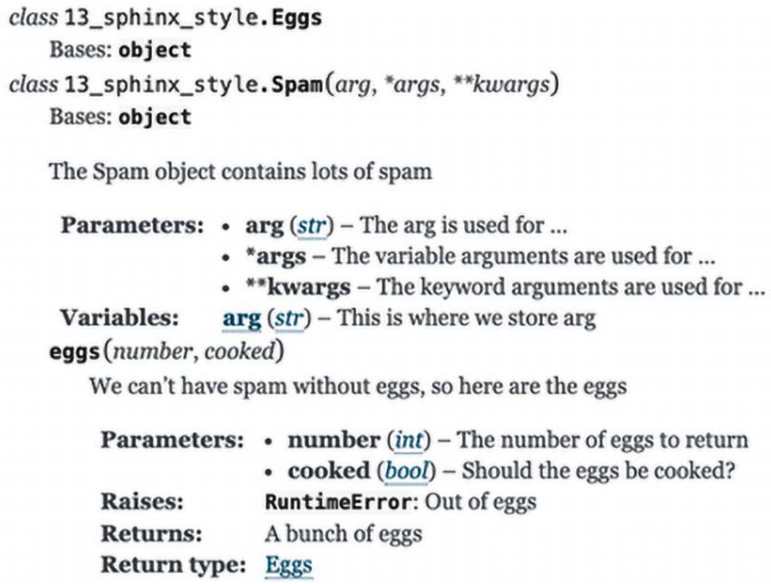


Figure 9.25: The Sphinx style documentation

This is a very useful output indeed, with documented functions, classes, and arguments. And, more importantly, the types are documented as well, resulting in a clickable link to the actual type. An added advantage of specifying the type is that many editors understand the documentation and will provide autocompletion based on the given types.

You might have also noticed that we specified the variable types both as documentation and using type hinting. While this is technically not needed, they apply to different parts of the documentation. The types shown in the function itself are done through type hinting: `eggs(number: int, cooked: bool) -> 13_sphinx_style.Eggs`. The `Parameters` and `Return type` are specified through the `:type` in the documentation.

To explain what's actually happening here, Sphinx has a few roles within the docstrings that offer hints as to what we are documenting.

The `param` role paired with a name sets the documentation for the parameter with that name. The `type` role paired with a name tells Sphinx the data type of the parameter. Both the roles are optional and the parameter simply won't have any added documentation if they are omitted, but the `param` role is always required for any documentation to show. Simply adding the `type` role without the `param` role will result in no output whatsoever, so take note to always pair them.

The `returns` role is similar to the `param` role with regards to documenting. While the `param` role documents a parameter, the `returns` role documents the returned object. They are slightly different, however. As opposed to the `param` role, the `returns` role is not dependent on the `rtype` role, or vice versa. They both work independently of each other, making it possible to use either or both of the roles.

The `rtype`, as you can expect, tells Sphinx (and several editors) what type of object is returned from the function. With the introduction of type hinting, however, the `rtype` role is pretty useless since you have an easier way of specifying the return type.

Documenting a class with the Google style

The Google style is just a more legible version of the Sphinx style documentation. It doesn't actually support more or less, but it's a lot more intuitive to use. Here's the Google style version of the `Spam` class:

```
class Eggs:
    pass

class Spam(object):
    r'''
    The Spam object contains lots of spam

    Args:
        arg: The arg is used for ...
        \*args: The variable arguments are used for ...
        \*\*kwargs: The keyword arguments are used for ...

    Attributes:
        arg: This is where we store arg,
    '''

    def __init__(self, arg: str, *args, **kwargs):
        self.arg: str = arg

    def eggs(self, number: int, cooked: bool) -> Eggs:
        '''We can't have spam without eggs, so here are the eggs

        Args:
            number (int): The number of eggs to return
            cooked (bool): Should the eggs be cooked?

        Raises:
            RuntimeError: Out of eggs

        Returns:
            Eggs: A bunch of eggs
        '''
        pass
```

This is easier on the eyes than the Sphinx style and has the same number of possibilities. For longer argument documentation, it's less than convenient though. Just imagine how a multiline description of `number` would look. That is why the NumPy style was developed, providing a lot of documentation for its arguments.

Documenting a class with the NumPy style

The NumPy style is meant for having a lot of documentation. Honestly, most people are too lazy for that, so for most projects it would not be a good fit. If you do plan to have extensive documentation of your functions and all their parameters, the NumPy style might be a good option for you. It's a bit more verbose than the Google style, but it's very legible, especially with more detailed documentation. The following is the NumPy version of the `Spam` class:

```
class Eggs:
    pass

class Spam(object):
    r'''
    The Spam object contains lots of spam

    Parameters
    ----------
    arg : str
        The arg is used for ...
    \*args
        The variable arguments are used for ...
    \*\*kwargs
        The keyword arguments are used for ...
    Attributes
    ----------
    arg : str
        This is where we store arg,
    '''

    def __init__(self, arg, *args, **kwargs):
        self.arg = arg

    def eggs(self, number, cooked):
        '''We can't have spam without eggs, so here are the eggs

        Parameters
        ----------
        number : int
```

```
            The number of eggs to return
        cooked : bool
            Should the eggs be cooked?

        Raises
        ------
        RuntimeError
            Out of eggs

        Returns
        -------
        Eggs
            A bunch of eggs
        '''
        pass
```

While the NumPy style definitely isn't bad, it's just very verbose. This example alone is about 1.5 times as long as the alternatives. So, for longer and more detailed documentation it's a very good choice, but if you're planning to have short documentation anyhow, just use the Google style instead.

Which style to choose

For most projects, the Google style is the best choice since it is readable but not too verbose. If you are planning to use large amounts of documentation per parameter, then the NumPy style might be a good option as well.

The only reason for choosing the Sphinx style is legacy. Even though the Google style might be more legible, consistency is more important.

Exercises

To practice a little with Python type hinting, it would be good to add some of this documentation to your own projects.

Some examples of less trivial type hints would be:

- Dictionaries
- Nested or even recursive types
- Generating stubs for documenting external projects that don't have type hinting

Example answers for these exercises can be found on GitHub: `https://github.com/mastering-python/exercises`. You are encouraged to submit your own solutions and learn about alternative solutions from others.

Summary

In this chapter, you have learned how to add, use, and test type hinting in your code using both the built-in types and your own custom types. You learned how to write Markdown and reStructuredText to document your projects and your code itself. Lastly, you learned how to use the Sphinx documentation generator to generate fully functioning documentation for your projects.

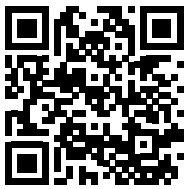
Documentation can help greatly in a project's popularity, and bad documentation can kill productivity. I think there are few aspects of a library that have more impact on the usage by third parties than documentation. Thus, in many cases, documentation is a more important factor in deciding the usage of a project than the actual code quality. That's why it is very important to always try to have good documentation available. Sphinx is a great help in this case because it makes it much easier to keep your documentation up to date and matching your code. The only thing worse than no documentation is incorrect and/or out-of-date documentation.

With Sphinx, it is easy to generate documentation. With just a few minutes of your time, you can have a fully functioning website with documentation available, or a PDF, or ePub, or one of the many other output formats. There really is no excuse for having no documentation anymore. And even if you don't use the documentation that much yourself, offering type hints to your editor can help a lot with productivity as well. Making your editor smarter should always help with productivity. I, for one, have added type hints to several external projects simply to increase my productivity.

The next chapter will explain how code can be tested in Python and some part of the documentation will return there. Using `doctest`, it is possible to have example code, documentation, and tests in one.

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:
`https://discord.gg/QMzJenHuJf`



10

Testing and Logging – Preparing for Bugs

When programming, most developers plan a bit and immediately start writing code. After all, we all expect to write bug-free code! Unfortunately, we don't. At some point, an incorrect assumption, a misinterpretation, or just a silly mistake is bound to happen. Debugging (covered in *Chapter 11, Debugging – Solving the Bugs*) will always be required at some point, but there are several methods that you can use to prevent bugs or, at the very least, make it much easier to solve them when they do occur.

To prevent bugs from occurring in the first place, test-driven development or, at the very least, functional/regression/unit tests, are very useful. The standard Python installation alone offers several options such as the `doctest`, `unittest`, and `test` modules. The `doctest` module allows you to combine tests with example documentation. The `unittest` module allows you to easily write regression tests. The `test` module is meant for internal usage only, so unless you are planning to modify the Python core, you probably won't need this one.

The test modules we will discuss in this chapter are:

- `doctest`
- `py.test` (and why it's more convenient than `unittest`)
- `unittest.mock`

The `py.test` module has roughly the same purpose as the `unittest` module, but it's much more convenient to use and has many more options and plugins available.

After learning how to avoid the bugs, it'll be time to take a look at logging so that we can inspect what is happening in our program and why. The `logging` module in Python is highly configurable and can be adjusted for just about any use case. If you've ever written Java code, you should feel right at home with the `logging` module, as its design is largely based on the `log4j` module and is very similar in both implementation and naming. The latter makes it a bit of an odd module in Python as well, as it is one of the few modules that does not follow the `pep8` naming standards.

This chapter will explain the following topics:

- Combining documentation with tests using `doctest`
- Regression and unit tests using `py.test` and `unittest`
- Testing with fake objects using `unittest.mock`
- Testing multiple environments using `tox`
- Using the `logging` module effectively
- Combining `logging` and `py.test`

Using documentation as tests with doctest

The `doctest` module is one of the most useful modules within Python. It allows you to combine documenting your code with tests to make sure that it keeps working as it is supposed to.

By now the format should be very familiar to you; most of the code samples in this book use the `doctest` format, which offers the advantage that both the input and the output are shown intertwined. Especially in demonstrations, this is much more convenient than having a block of code followed by the output.

A simple doctest example

Let's start with a quick example: a function that squares the input. The following example is a fully functional command-line application, containing not only code but also functioning tests. The first few tests cover how the function is supposed to behave when executing normally, followed by a few tests to demonstrate the expected errors:

```
def square(n: int) -> int:
    '''
    Returns the input number, squared

    >>> square(0)
    0
    >>> square(1)
    1
    >>> square(2)
    4
    >>> square(3)
    9
    >>> square()
    Traceback (most recent call last):
    ...
    TypeError: square() missing 1 required positional argument: 'n'
    >>> square('x')
    Traceback (most recent call last):
    ...
    TypeError: can't multiply sequence by non-int of type 'str'
```

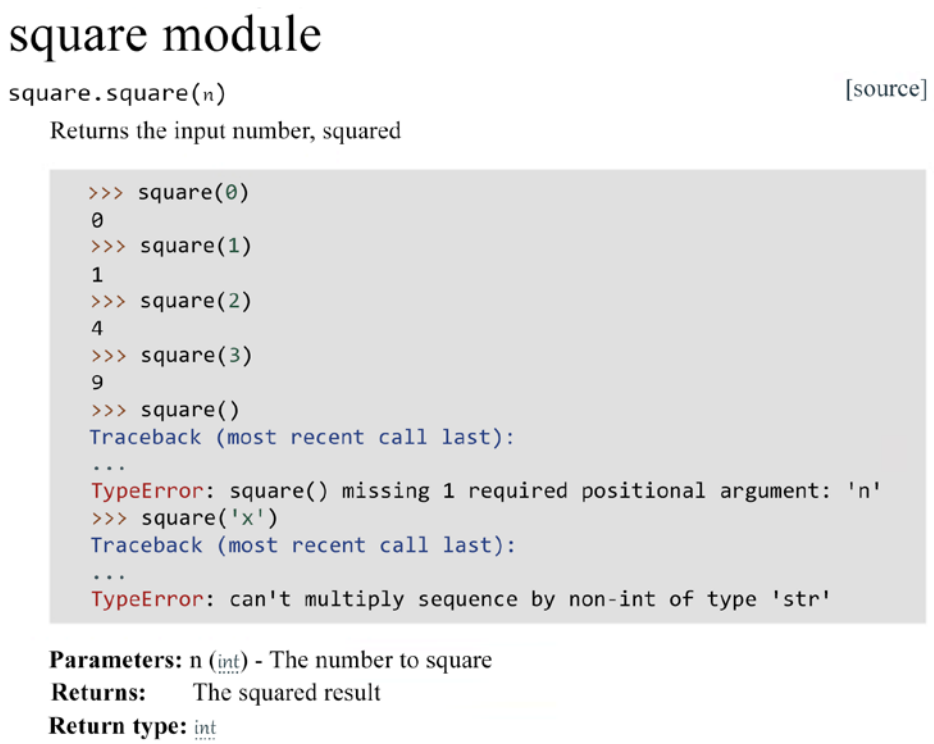
```
    Args:
        n (int): The number to square

    Returns:
        int: The squared result
    '''
    return n * n

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

It can be executed as any Python script, but the regular command won't give any output as all tests are successful. The `doctest.testmod` function takes verbosity parameters, luckily:

```
$ python3 T_00_simple_doctest.py -v
Trying:
    square(0)
Expecting:
    0
ok
Trying:
    square(1)
Expecting:
    1
ok
Trying:
    square(2)
Expecting:
    4
ok
Trying:
    square(3)
Expecting:
    9
ok
Trying:
    square()
Expecting:
    Traceback (most recent call last):
    ...
```

```
    TypeError: square() missing 1 required positional argument: 'n'
ok
Trying:
    square('x')
Expecting:
    Traceback (most recent call last):
    ...
    TypeError: can't multiply sequence by non-int of type 'str'
ok
1 items had no tests:
    __main__
1 items passed all tests:
   6 tests in __main__.square
6 tests in 2 items.
6 passed and 0 failed.
Test passed.
```

Additionally, since it uses the Google syntax (as discussed in *Chapter 9*, *Documentation – How to Use Sphinx and reStructuredText*, the documentation chapter), we can generate pretty documentation using Sphinx:



square module

`square.square(n)` [source]

Returns the input number, squared

```
>>> square(0)
0
>>> square(1)
1
>>> square(2)
4
>>> square(3)
9
>>> square()
Traceback (most recent call last):
...
TypeError: square() missing 1 required positional argument: 'n'
>>> square('x')
Traceback (most recent call last):
...
TypeError: can't multiply sequence by non-int of type 'str'
```

Parameters: n (int) - The number to square
Returns: The squared result
Return type: int

Figure 10.1: Documentation generated using Sphinx

However, the code is not always correct, of course. What will happen if we modify the code so that the tests do not pass anymore?

This time, instead of `n * n`, we use `n ** 2`. Both square a number, so the results must be identical. Right? These are the types of assumptions that create bugs, and the types of assumptions that are trivial to catch using a few basic tests. Since most results are the same we will skip them in the example, but one test has different results now:

```
def square(n: int) -> int:
    '''
    >>> square('x')
    Traceback (most recent call last):
    ...
    TypeError: unsupported operand type(s) for ** or pow(): ...
    '''
    return n ** 2

if __name__ == '__main__':
    import doctest
    doctest.testmod(optionflags=doctest.ELLIPSIS)
```

The only modification we made to the code was replacing `n * n` with `n ** 2`, which translates to the power function. Since multiplication is not the same as taking the power of a number, the results are slightly different, but similar enough in practice that most programmers wouldn't notice the difference.

Because of that difference, however, the error changed from `can't multiply sequence ...` to `unsupported operand type(s) for ** or pow(): ...`. It's an innocent mistake, but a quick optimization by a programmer could have changed this unintentionally with possibly wrong results. If the `__pow__` method was overloaded with different behavior, for example, this could result in bigger problems.

This example has shown us how useful these tests can be. When rewriting or optimizing code, an incorrect assumption is easily made, and that is where tests are very useful—knowing you are breaking code as soon as you break it instead of finding out months later.

Writing doctests

Perhaps you have noticed from the preceding examples that the syntax is very similar to the regular Python console, and that is because it is. The `doctest` input is nothing more than the output of a regular Python shell session. This is what makes testing with this module so intuitive; simply write the code in the Python console and copy the output into a docstring to get tests. Here is an example:

```
$ python3
>>> from square import square

>>> square(5)
```

```
25
>>> square()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: square() missing 1 required positional argument: 'n'
```

That's why this is probably the easiest way to test code. With almost no effort, you can check whether your code is working as you would expect it, add tests, and add documentation at the same time. Simply copy the output from the interpreter to your function or class documentation and you have functioning doctests.

Testing with documentation

The docstrings in functions, classes, and modules are usually the most obvious way to add doctests to your code, but they are not the only way. The Sphinx documentation, as we discussed in the previous chapter, also supports the `doctest` module.

To enable `doctest` support in Sphinx, you need to add the `sphinx.ext.doctest` extension in Sphinx, which tells Sphinx to run those tests as well. Since not all the examples in the code are useful, let's see whether we can split them into the ones that are actually useful and the ones that are only relevant for documentation. Moreover, to see the results, we will add an error to the documentation.

`square.py`

```
def square(n: int) -> int:
    '''
    Returns the input number, squared

    >>> square(2)
    4

    Args:
        n (int): The number to square

    Returns:
        int: The squared result
    '''
    return n * n

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

`square.rst`

```
square module
=============

.. automodule:: square
    :members:
    :undoc-members:
    :show-inheritance:

Examples:

.. testsetup::

    from square import square

.. doctest::

    # pytest does not recognize testsetup
    >>> from square import square

    >>> square(100)
    10000
    >>> square(0)
    0
    >>> square(1)
    1
    >>> square(3)
    9
    >>> square()
    Traceback (most recent call last):
    ...
    TypeError: square() missing 1 required positional argument: 'n'

    >>> square('x')
    Traceback (most recent call last):
    ...
    TypeError: can't multiply sequence by non-int of type 'str'
```

Now, it's time to execute the tests. In the case of Sphinx, there is a specific command for this:

```
$ make doctest
Running Sphinx v3.2.1
loading pickled environment... done
building [mo]: targets for 0 po files that are out of date
building [doctest]: targets for 2 source files that are out of date
updating environment: 0 added, 0 changed, 0 removed
looking for now-outdated files... none found
running tests...

Document: square
---------------
1 items passed all tests:
   8 tests in default
8 tests in 1 items.
8 passed and 0 failed.
Test passed.

Doctest summary
===============
    8 tests
    0 failures in tests
    0 failures in setup code
    0 failures in cleanup code
build succeeded.

Testing of doctests in the sources finished, look at the results in _build/
doctest/output.txt.
```

As expected, we are getting an error for the incomplete `doctest`, but beyond that, all tests executed correctly. To make sure that the tests know what `square` is, we had to add the `testsetup` directive, and this still generates a pretty output:

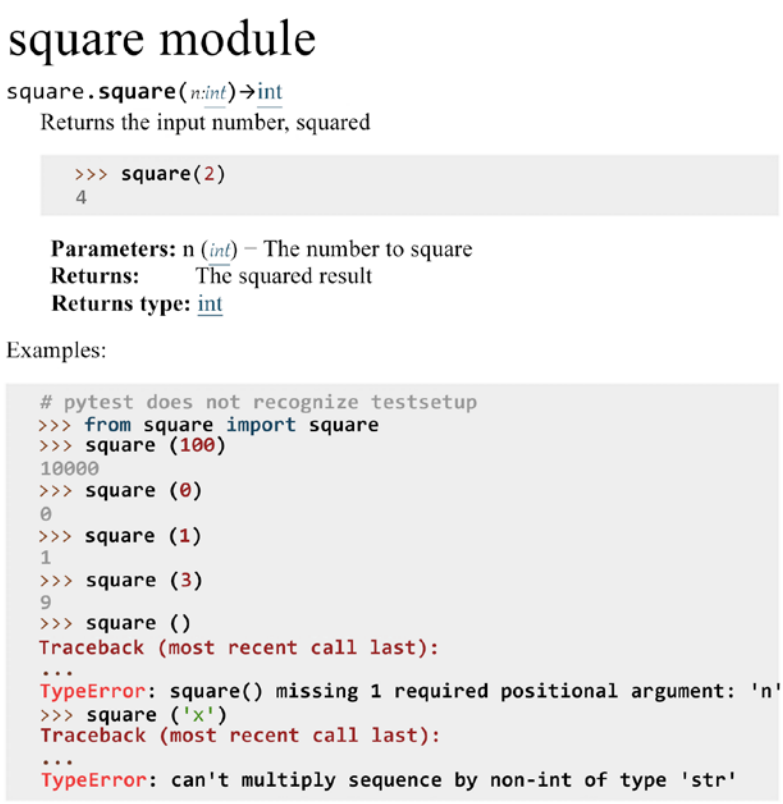


Figure 10.2: Rendered Sphinx output

Sphinx nicely renders both the documentation for the code and the highlighted code samples.

The doctest flags

The `doctest` module features several option flags that affect how `doctest` processes the tests. These option flags can be passed globally using your test suite, through command-line parameters while running the tests, and through inline commands. For this book, I have globally enabled the following option flags through a `pytest.ini` file (we will cover more about `py.test` later in this chapter):

```
doctest_optionflags = ELLIPSIS NORMALIZE_WHITESPACE
```

Without these option flags, some of the examples in this book will not function properly. This is because they have to be reformatted to fit. The next few paragraphs will cover the following option flags:

- `DONT_ACCEPT_TRUE_FOR_1`
- `NORMALIZE_WHITESPACE`
- `ELLIPSIS`

There are several other option flags available with varying degrees of usefulness, but these are better left to the Python documentation: `https://docs.python.org/3/library/doctest.html#option-flags`

True and False versus 1 and 0

Having `True` evaluating to `1` and `False` evaluating to `0` is useful in most cases, but it can give unexpected results if you were actually expecting a `bool` instead of an `int`. To demonstrate the difference, we have these lines:

```
'''
>>> False
0
>>> True
1
'''
if __name__ == '__main__':
    import doctest
    doctest.testmod()
    doctest.testmod(optionflags=doctest.DONT_ACCEPT_TRUE_FOR_1)
```

When we run this, it will run the tests both without and with the `DONT_ACCEPT_TRUE_FOR_1` flag:

```
$ python3 T_03_doctest_true_for_1_flag.py -v
Trying:
    False
Expecting:
    0
ok
Trying:
    True
Expecting:
    1
ok
1 items passed all tests:
   2 tests in __main__
2 tests in 1 items.
2 passed and 0 failed.
Test passed.
```

```
Trying:
    False
Expecting:
    0
**********************************************************************
File "T_03_doctest_true_for_1_flag.py", line 2, in __main__
Failed example:
    False
Expected:
    0
Got:
    False
Trying:
    True
Expecting:
    1
**********************************************************************
File "T_03_doctest_true_for_1_flag.py", line 4, in __main__
Failed example:
    True
Expected:
    1
Got:
    True
**********************************************************************
1 items had failures:
   2 of   2 in __main__
2 tests in 1 items.
0 passed and 2 failed.
***Test Failed*** 2 failures.
```

As you can see, the `DONT_ACCEPT_TRUE_FOR_1` flag makes `doctest` reject `1` as a valid response for `True` as well as `0` for `False`.

Normalizing whitespace

Since doctests are used for both documentation and test purposes, it is pretty much a requirement to keep them readable. Without normalizing whitespace, this can be tricky, however. Consider the following example:

```
>>> [list(range(5)) for i in range(3)]
[[0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4]]
```

While not all that bad, this output isn't the best for readability. With whitespace normalizing, here is what we can do instead:

```
>>> # doctest: +NORMALIZE_WHITESPACE
... [list(range(5)) for i in range(3)]
[[0, 1, 2, 3, 4],
 [0, 1, 2, 3, 4],
 [0, 1, 2, 3, 4]]
```

Formatting the output in this manner is both more readable and convenient for keeping your lines shorter.

Ellipsis

The `ELLIPSIS` flag is very useful but also a bit dangerous, as it can easily lead to incorrect matches. It makes `...` match any substring, which is very useful for exceptions but dangerous in other cases:

```
>>> {10: 'a', 20: 'b'}  # doctest: +ELLIPSIS
{...}
>>> [True, 1, 'a']  # doctest: +ELLIPSIS
[...]
>>> True,  # doctest: +ELLIPSIS
(...)
>>> [1, 2, 3, 4]  # doctest: +ELLIPSIS
[1, ..., 4]
>>> [1, 0, 0, 0, 0, 0, 4]  # doctest: +ELLIPSIS
[1, ..., 4]
```

These cases are not too useful in real scenarios, but they demonstrate how the `ELLIPSIS` option flag functions. They also indicate the danger. Both `[1, 2, 3, 4]` and `[1, 0, ... , 4]` match the `[1, ..., 4]` test, which is probably unintentional, so be very careful while using `ELLIPSIS`.

A more useful case is when documenting class instances:

```
>>> class Spam(object):
...     pass

>>> Spam()  # doctest: +ELLIPSIS
<__main__.Spam object at 0x...>
```

Without the `ELLIPSIS` flag, the memory address (the `0x...` part) would never be what you expect. Let's demonstrate an actual run in a normal CPython instance:

```
Failed example:
    Spam()
Expected:
    <__main__.Spam object at 0x...>
Got:
```

```
    <__main__.Spam object at 0x10d9ad160>
```

Doctest quirks

The three option flags discussed earlier take care of quite a few quirks found in doctests, but there are several more cases that require care. In these cases, you just need to be a bit careful and work around the limitations of the `doctest` module. The `doctest` module effectively uses the representation string, and those are not always consistent.

The representation string can be generated using `repr(object)` and uses the `__repr__` magic method internally. On regular classes without a specific `__repr__` method, this will look like `<module.className instance at 0x....>`, where the `0x...` is the memory address of the object, which changes with each run and each object.

The most important cases are floating-point inaccuracies, and random values, such as timers. With the following example, the floating-point example will return consistent results for your system, but on a different system it is likely to fail. The `time` example will almost certainly always fail:

```
>>> 1./7.
0.14285714285714285

>>> import time

>>> time.time() - time.time()
-9.5367431640625e-07
```

All the problems have several possible solutions, which differ mostly in style and your personal preference.

Testing dictionaries

Since the implementation of dictionaries has changed in recent Python versions, this exact issue is probably one you will not encounter anymore. However, there are still situations where similar solutions are useful.

The problem with dictionaries used to be that they had an effectively random representation order. Since the `doctest` system requires a representation string that is identical in meaning (save for certain `doctest` flags, of course) to the `docstring`, this does not work. Naturally, there are several workaround options available and all have some advantages and disadvantages.

The first is using the `pprint` (pretty print) library to format the dictionary in a pretty and consistent way:

```
>>> import pprint

>>> data = dict.fromkeys('spam')
>>> pprint.pprint(data)
{'a': None, 'm': None, 'p': None, 's': None}
```

Since the `pprint` library always sorts the items before outputting, this solves the problem with random representation orders. However, it does require an extra import and function call, which some people prefer to avoid.

Another option is manual sorting of the items:

```
>>> data = dict.fromkeys('spam')
>>> sorted(data.items())
[('a', None), ('m', None), ('p', None), ('s', None)]
```

The downside here is that it is not visible from the output that `data` is a dictionary, which makes the output less readable.

Lastly, comparing the `dict` with a different `dict` comprising the same elements works as well:

```
>>> data = dict.fromkeys('spam')
>>> data == {'a': None, 'm': None, 'p': None, 's': None}
True
```

A perfectly okay solution, of course! But `True` is not really the clearest output, especially if the comparison doesn't work:

```
Failed example:
    data == {'a': None, 'm': None, 'p': None}
Expected:
    True
Got:
    False
```

On the other hand, the other options presented previously show both the expected value and the returned value correctly:

```
Failed example:
    sorted(data.items())
Expected:
    [('a', None), ('m', None), ('p', None)]
Got:
    [('a', None), ('m', None), ('p', None), ('s', None)]

Failed example:
    pprint.pprint(data)
Expected:
    {'a': None, 'm': None, 'p': None}
Got:
    {'a': None, 'm': None, 'p': None, 's': None}
```

Personally, out of the solutions presented, I would recommend using `pprint`, as I find it the most readable solution, but all the solutions have some merits to them.

Testing floating-point numbers

For the same reason a floating-point comparison can be problematic (that is, `1/3 == 0.333`), a representation string comparison is also problematic. The easiest solution is to round or clip the value, but the `ELLIPSIS` flag is also an option here. Here is a list of several solutions:

```
>>> 1/3  # doctest: +ELLIPSIS
0.333...
>>> '%.3f' % (1/3)
'0.333'
>>> '{:.3f}'.format(1/3)
'0.333'
>>> round(1/3, 3)
0.333
>>> 0.333 < 1/3 < 0.334
True
```

Which solution you choose should depend on your own preference or consistency with the project you are working on. In general, my choice would be to enable the `ELLIPSIS` option flag globally and go for that solution, as it looks the cleanest to me.

Times and durations

For timings, the problems that you will encounter are quite similar to the floating-point issues. When measuring the execution time of a code snippet, there will always be some variation present. That's why limiting the precision is the easiest solution for time dependent tests. To achieve this we can check whether the delta (difference) between the two times is smaller than a certain number:

```
>>> import time

>>> a = time.time()
>>> b = time.time()
>>> (b - a) < 0.01
True
```

For the `timedelta` objects, however, it's slightly more complicated. Yet, this is where the `ELLIPSIS` flag definitely comes in handy again:

```
>>> import datetime

>>> a = datetime.datetime.now()
>>> b = datetime.datetime.now()
>>> str(b - a)  # doctest: +ELLIPSIS
'0:00:00.000...
```

The alternative to the `ELLIPSIS` option flag would be comparing the days, hours, minutes, and microseconds in `timedelta` separately. Or you can use `timedelta.total_seconds()` to convert the `timedelta` into seconds and use a regular floating-point comparison.

In a later paragraph, we will see a completely stable solution for problems like these using mock objects. For doctests, however, that is generally overkill.

Now that we are done with `doctest`, it is time to continue with more explicit tests using `py.test`.

Testing with py.test

The `py.test` tool makes it very easy to write tests and run them. There are a few other options such as `nose2` and the bundled `unittest` module available, but the `py.test` library offers a very good combination of usability and active development. In the past, I was an avid `nose` user but have since switched to `py.test` as it tends to be easier to use and has better community support, in my experience at least. Regardless, `nose2` is still a good choice, and if you're already using either `nose` or `nose2`, there is little reason to switch and rewrite all of your tests. When writing tests for a new project, however, `py.test` can be much more convenient.

Now, we will run the doctests from the previously discussed `square.py` file using `py.test`.

First, start by installing `py.test`, of course:

```
$ pip3 install pytest pytest-flake8
```

We also installed `pytest-flake8` here because the default `pytest.ini` for this project depends on it. We will discuss what it does and how it can be configured later in this chapter.

Now you can do a test run, so let's give the doctests we have in `square.py` a try:

```
$ py.test --doctest-modules -v square.py
==================== test session starts ====================
collected 2 items

square.py::square.square PASSED [100%]

==================== 1 passed in 0.03s =====================
```

We can see that `py.test` was able to find two tests for the given file: the test in `square.square` itself, and a `flake8` test from the `pytest-flake8` plugin that we will see later in this chapter.

The difference between the unittest and py.test output

We have the doctests in `square.py`. Let's create a new class called `cube` and create a proper set of tests outside of the code.

First of all, we have the code of `cube.py`, similar to `square.py` but minus the doctests, since they mostly won't work anyway:

```
def cube(n: int) -> int:
    '''
    Returns the input number, cubed

    Args:
        n (int): The number to cube

    Returns:
        int: The cubed result
    '''
    return n ** 3
```

Now let's start with the `unittest` example, `T_09_test_cube.py`:

```
import cube
import unittest

class TestCube(unittest.TestCase):
    def test_0(self):
        self.assertEqual(cube.cube(0), 0)

    def test_1(self):
        self.assertEqual(cube.cube(1), 1)

    def test_2(self):
        self.assertEqual(cube.cube(2), 8)

    def test_3(self):
        self.assertEqual(cube.cube(3), 27)

    def test_no_arguments(self):
        with self.assertRaises(TypeError):
            cube.cube()

    def test_exception_str(self):
        with self.assertRaises(TypeError):
            cube.cube('x')

if __name__ == '__main__':
    unittest.main()
```

This can be executed by executing the file itself:

```
$ python3 T_09_test_cube.py -v
test_0 (__main__.TestCube) ... ok
test_1 (__main__.TestCube) ... ok
test_2 (__main__.TestCube) ... ok
test_3 (__main__.TestCube) ... ok
test_exception_str (__main__.TestCube) ... ok
test_no_arguments (__main__.TestCube) ... ok

----------------------------------------------------------------
Ran 6 tests in 0.000s

OK
```

Alternatively, it can be done through the `unittest` module:

```
$ python3 -m unittest -v T_09_test_cube.py
...
```

But it also works with other tools such as `py.test`:

```
$ py.test -v T_09_test_cube.py
===================== test session starts ======================
collected 7 items

T_09_test_cube.py::FLAKE8 SKIPPED                         [ 14%]
T_09_test_cube.py::TestCube::test_0 PASSED                [ 28%]
T_09_test_cube.py::TestCube::test_1 PASSED                [ 42%]
T_09_test_cube.py::TestCube::test_2 PASSED                [ 57%]
T_09_test_cube.py::TestCube::test_3 PASSED                [ 71%]
T_09_test_cube.py::TestCube::test_exception_str PASSED    [ 85%]
T_09_test_cube.py::TestCube::test_no_arguments PASSED     [100%]

================= 6 passed, 1 skipped in 0.08s =================
```

And other tools such as `nose` are also possible. First, we need to install it using pip:

```
$ pip3 install nose
```

After that, we can use the `nosetests` command to run:

```
$ nosetests -v T_09_test_cube.py
test_0 (T_09_test_cube.TestCube) ... ok
test_1 (T_09_test_cube.TestCube) ... ok
```

```
test_2 (T_09_test_cube.TestCube) ... ok
test_3 (T_09_test_cube.TestCube) ... ok
test_exception_str (T_09_test_cube.TestCube) ... ok
test_no_arguments (T_09_test_cube.TestCube) ... ok

----------------------------------------------------------------
Ran 6 tests in 0.001s

OK
```

As long as all the results are successful, the differences between the output from `unittest` and `py.test` are slim. This time around, however, we are going to break the code to show the difference when it actually matters. Instead of the `cube` code, we will add the `square` code, returning `n ** 2` from `square`, instead of `n ** 3`.

To reduce the amount of output, we will not be running the verbose variants of the commands here.

First of all, we have the regular `unittest` output:

```
$ python3 T_09_test_cube.py
..FF..
======================================================================
FAIL: test_2 (__main__.TestCube)
----------------------------------------------------------------------
Traceback (most recent call last):
  File " T_09_test_cube.py", line 14, in test_2
    self.assertEqual(cube.cube(2), 8)
AssertionError: 4 != 8

======================================================================
FAIL: test_3 (__main__.TestCube)
----------------------------------------------------------------------
Traceback (most recent call last):
  File " T_09_test_cube.py", line 17, in test_3
    self.assertEqual(cube.cube(3), 27)
AssertionError: 9 != 27

----------------------------------------------------------------------
Ran 6 tests in 0.001s

FAILED (failures=2)
```

Not all that bad, as each test returns a nice stack trace that includes the values and everything. Yet, we can observe a small difference here when compared with the `py.test` run:

```
$ py.test T_09_test_cube.py
==================== test session starts ======================
collected 7 items

T_09_test_cube.py s..FF..                                 [100%]

========================== FAILURES ===========================
_____________________ TestCube.test_2 ______________________

self = <T_09_test_cube.TestCube testMethod=test_2>

    def test_2(self):
>       self.assertEqual(cube.cube(2), 8)
E       AssertionError: 4 != 8

T_09_test_cube.py:14: AssertionError
_____________________ TestCube.test_3 ______________________

self = <T_09_test_cube.TestCube testMeth
od=test_3>

    def test_3(self):
>       self.assertEqual(cube.cube(3), 27)
E       AssertionError: 9 != 27

T_09_test_cube.py:17: AssertionError
================== short test summary info ====================
FAILED T_09_test_cube.py::TestCube::test_2 - AssertionError: 4..
FAILED T_09_test_cube.py::TestCube::test_3 - AssertionError: 9..
============ 2 failed, 4 passed, 1 skipped in 0.17s ============
```

In small cases such as these, the difference is not all that apparent, but when testing complicated code with large stack traces, it becomes even more useful. However, for me personally, seeing the surrounding test code is a big advantage.

In the example that was just discussed, the `self.assertEqual(...)` line shows the entire test, but in many other cases, you will need more information. The difference between the regular `unittest` module and the `py.test` module is that with `py.test` you can see the entire function with all of the code and the output. Later in this chapter, we will see how powerful this can be when writing more advanced tests.

To truly appreciate the `py.test` output, we need colors as well. Unfortunately, that is not possible within the constraints of this book, but I strongly encourage you to give `py.test` a try if you aren't using it already.

Perhaps you are wondering now, "Is that all? The only difference between `py.test` and `unittest` is a bit of color and a slightly different output?" Well, far from it; there are many other differences, but this alone is enough reason to give it a try.

The difference between unittest and py.test tests

The improved output does help a bit, but the combination of improved output and a much easier way to write tests is what makes `py.test` so useful. There are quite a few methods for making the tests simpler and more legible, and in many cases, you can choose which you prefer. As always, readability counts, so choose wisely and try not to over-engineer the solutions.

Simplifying assertions

Where the `unittest` library requires the usage of `self.assertEqual` to compare variables, `py.test` allows the use of a regular `assert` statement while still understanding the comparison between the variables.

The following test file contains three styles of tests, so they can be compared easily:

```
import unittest
import cube

n = 2
expected = 8

# Regular unit test
class TestCube(unittest.TestCase):
    def test_2(self):
        self.assertEqual(cube.cube(n), expected)

    def test_no_arguments(self):
        with self.assertRaises(TypeError):
            cube.cube()

# py.test class
class TestPyCube:
    def test_2(self):
        assert cube.cube(n) == expected

# py.test functions
def test_2():
    assert cube.cube(n) == expected
```

To convert to `py.test`, we simply replaced `self.assertEqual` with `assert ... == ...`. A minor improvement indeed, but the actual benefit is seen in the failure output. The first two use the `unittest` style and the latter two use the `py.test` style both inside a class and as separate functions:

```
$ py.test T_10_simplifying_assertions.py
...
=========================== FAILURES ===========================
_____________________ TestCube.test_2 _____________________

self = <TestCube testMethod=test_2>

    def test_2(self):
>       self.assertEqual(cube.cube(n), expected)
E       AssertionError: 4 != 8

T_10_simplifying_assertions.py:12: AssertionError
_____________________ TestPyCube.test_2 _____________________

self = <TestPyCube object at 0x...>

    def test_2(self):
>       assert cube.cube(n) == expected
E       assert 4 == 8
E        +  where 4 = <function cube at 0x...>(2)
E        +    where <function cube at 0x...> = cube.cube

T_10_simplifying_assertions.py:23: AssertionError
_____________________________ test_2 _____________________________

    def test_2():
>       assert cube.cube(n) == expected
E       assert 4 == 8
E        +  where 4 = <function cube at 0x...>(2)
E        +    where <function cube at 0x...> = cube.cube

T_10_simplifying_assertions.py:28: AssertionError
================== short test summary info ====================
FAILED T_10_simplifying_assertions.py::TestCube::test_2 - Ass...
FAILED T_10_simplifying_assertions.py::TestPyCube::test_2 - a...
FAILED T_10_simplifying_assertions.py::test_2 - assert 4 == 8
============ 3 failed, 1 passed, 1 skipped in 0.15s ============
```

In addition to seeing the values that were compared, we can actually see the function that was called and which input parameters it received. With the regular `unittest` we have no way of knowing that `2` was entered as a parameter to the `cube()` function.

The standard `py.test` behavior works for most test cases, but it may not be enough for some custom types. For example, let's say that we have a `User` object with a `name` attribute that should be compared with the `name` attribute on another object. This part can easily be achieved by implementing the `__eq__` method on `User`, but it does not improve clarity. Since `name` is the attribute that we compare, it would be useful if the tests showed `name` when errors were displayed.

First is the class with two tests, one working and one broken to demonstrate the regular output:

`T_11_representing_assertions.py`

```
class User:
    def __init__(self, name):
        self.name = name

    def __eq__(self, other):
        return self.name == other.name

def test_user_equal():
    a = User('Rick')
    b = User('Guido')

    assert a == b
```

And here is the regular `py.test` output:

```
____________________ test_user_equal ____________________

    def test_user_equal():
        a = User('Rick')
        b = User('Guido')

>       assert a == b
E       assert <T_11_representing_assertions.User object at 0x...> == <T_11_
representing_assertions.User object at 0x...>

T_11_representing_assertions.py:13: AssertionError
=================== short test summary info ====================
FAILED T_11_representing_assertions.py::test_user_equal - asse...
================= 1 failed, 1 skipped in 0.17s =================
```

The default test output is still usable since the function is fairly straightforward, and the value for `name` is visible due to it being available in the constructor. However, it would have been more useful if we could explicitly see the value of `name`. By adding a `pytest_assertrepr_compare` function to the `conftest.py` file, we can modify the behavior of the `assert` statements.

The `conftest.py` file is a special file for `py.test` that can be used to override or extend `py.test`. Note that this file will automatically be loaded by every test run in that directory, so we need to test the types of both the left-hand side and the right-hand side of the operator. In this case, it's `a` and `b`.

`conftest.py`

```
from T_12_assert_representation import User

def is_user(value):
    return isinstance(value, User)

def pytest_assertrepr_compare(config, op, left, right):
    if is_user(left) and is_user(right) and op == '==':
        return [
            'Comparing User instances:',
            f'    name: {left.name} != {right.name}',
        ]
```

The preceding function will be used as the output for our test. So when it fails, this time we get our own, slightly more useful, output:

```
    def test_user_equal():
        a = User('Rick')
        b = User('Guido')

>       assert a == b
E       assert Comparing User instances:
E             name: Rick != Guido

T_12_assert_representation.py:13: AssertionError
```

In this case, we could have easily changed the `__repr__` function of `User` as well, but there are many cases where modifying the `py.test` output can be useful – if you need more debug output, for example. Similar to this, there is specific support for many types, such as sets, dictionaries, and texts.

Parameterizing tests

So far, we have specified every test separately, but we can simplify tests a lot by parameterizing them. The square and cube tests are very similar; a certain input gave a certain output.

You could solve this by creating a loop in a test, but a loop in a test will be executed as a single test. This means that it will fail in its entirety if a single test iteration of the loop fails, which means you can't easily see what exactly broke if you compare older and newer test output. In this example with the numbers, the result is obvious, but if you were to apply a list of filenames to a complicated processing test, it would be far less obvious what happened.

In these cases, parameterized tests can help a lot. After creating a list of parameters and the expected output data, you can make it run the test function for every parameter combination separately:

```
import cube
import pytest

cubes = (
    (0, 0),
    (1, 1),
    (2, 8),
    (3, 27),
)

@pytest.mark.parametrize('n,expected', cubes)
def test_cube(n, expected):
    assert cube.cube(n) == expected
```

This outputs the following, as you might have already expected:

```
============================ FAILURES ============================
____________________________ test_cube[2-8] ____________________________

n = 2, expected = 8
    @pytest.mark.parametrize('n,expected', cubes)
    def test_cube(n, expected):
>       assert cube.cube(n) == expected
E       assert 4 == 8
E        +  where 4 = <function cube at 0x...>(2)
E        +    where <function cube at 0x...> = cube.cube

T_13_parameterizing_tests.py:15: AssertionError
____________________________ test_cube[3-27] ____________________________

n = 3, expected = 27

    @pytest.mark.parametrize('n,expected', cubes)
    def test_cube(n, expected):
```

```
>       assert cube.cube(n) == expected
E       assert 9 == 27
E        +  where 9 = <function cube at 0x...>(3)
E        +    where <function cube at 0x...> = cube.cube

T_13_parameterizing_tests.py:15: AssertionError
=================== short test summary info ====================
FAILED T_13_parameterizing_tests.py::test_cube[2-8] - assert ...
FAILED T_13_parameterizing_tests.py::test_cube[3-27] - assert...
============ 2 failed, 2 passed, 1 skipped in 0.16s ============
```

With the parameterized tests, we can see the parameters clearly, which means we can see all inputs and outputs without any extra effort.

Generating the list of tests dynamically at runtime is also possible with a global function. Similar to the `pytest_assertrepr_compare` function that we added to `conftest.py` earlier, we can add a `pytest_generate_tests` function, which generates tests.

Creating the `pytest_generate_tests` function can be useful only to test a subset of options depending on the configuration options. If possible, however, I recommend trying to configure selective tests using fixtures instead, as they are somewhat more explicit. We will cover this in the following section. The problem with functions such as `pytest_generate_tests` is that they are global and don't discriminate between specific tests, resulting in strange behavior if you are not expecting that.

Automatic arguments using fixtures

The `py.test` fixture system is one of the most magical features of `py.test`. It magically executes a fixture function with the same **name** as your arguments. Let's create a basic fixture to demonstrate this:

```
import pytest

@pytest.fixture
def name():
    return 'Rick'

def test_something(name):
    assert name == 'Rick'
```

When the `test_something()` test is executed, the `name` argument will be filled with the output from the `name()` function automatically.

Because arguments are automatically filled by fixtures, the naming of the arguments becomes very important, as fixtures can easily collide with other fixtures. To prevent collisions, the scope is set to `function` by default. However, `class`, `module`, and `session` are also valid options for the scope. There are several fixtures available by default, some of which you will use often, and others most likely never. A complete list can always be generated with the following command:

```
$ py.test --quiet --fixtures
...
capsys
    enables capturing of writes to sys.stdout/sys.stderr and
    makes captured output available via ''capsys.readouterr()''
    method calls which return a ''(out, err)'' tuple.
...
monkeypatch
    The returned ''monkeypatch'' funcarg provides these helper
    methods to modify objects, dictionaries or os.environ::

    monkeypatch.setattr(obj, name, value, raising=True)
    monkeypatch.delattr(obj, name, raising=True)
    monkeypatch.setitem(mapping, name, value)
    monkeypatch.delitem(obj, name, raising=True)
    monkeypatch.setenv(name, value, prepend=False)
    monkeypatch.delenv(name, value, raising=True)
    monkeypatch.syspath_prepend(path)
    monkeypatch.chdir(path)

    All modifications will be undone after the requesting
    test function has finished. The ''raising''
    parameter determines if a KeyError or AttributeError
    will be raised if the set/deletion operation has no target.
...
tmpdir
    return a temporary directory path object which is unique to
    each test function invocation, created as a sub directory of
    the base temporary directory. The returned object is a
    'py.path.local'_ path object.
```

The next few paragraphs demonstrate some fixture usage, and the `monkeypatch` fixture is covered later in the chapter.

cache

The `cache` fixture is as simple as it is useful; there is a `get` function and a `set` function, and the `cache` state remains between separate `py.test` runs. To illustrate how to get and set values from `cache`:

```
def test_cache(cache):
    counter = cache.get('counter', 0) + 1
    assert counter
    cache.set('counter', counter)
```

The default value (`0` in this case) is required for the `cache.get` function.

The cache can be cleared through the `--cache-clear` command-line parameter, and all caches can be shown through `--cache-show`. Internally, the `cache` fixture uses the `json` module to encode/decode the values, so anything JSON encodable will work.

Custom fixtures

Bundled fixtures are quite useful, but within most projects, you will want to create your own fixtures to make things easier. Fixtures make it trivial to repeat code that is needed more often. You are most likely wondering how this is different from a regular function, context wrapper, or something else, but the special thing about fixtures is that they themselves can accept fixtures as well. So, if your function needs the `pytestconfig` variables, it can ask for them without needing to modify the calling functions.

You can create a fixture out of anything that would be useful to reuse. The basic premise is simple enough: a function with the `pytest.fixture` decorator, which returns a value that will be passed along as an argument. Also, the function can take parameters and fixtures just as any test can.

The only notable variation is `pytest.yield_fixture`. This fixture variation has one small difference: the actual test will be executed at the `yield` (more than one `yield` results in errors) and the code before/after functions as setup/teardown code, which is useful for things like database connections and file handles. A basic example of a `fixture` and a `yield_fixture` looks like this:

```
import pytest

@pytest.yield_fixture
def some_yield_fixture():
    with open(__file__ + '.txt', 'w') as fh:
        # Before the function
        yield fh
        # After the function

@pytest.fixture
def some_regular_fixture():
    # Do something here
    return 'some_value_to_pass_as_parameter'

def some_test(some_yield_fixture, some_regular_fixture):
    some_yield_fixture.write(some_regular_fixture)
```

These fixtures take no parameters and simply pass a parameter to the `py.test` functions. A more useful example would be setting up a database connection and executing a query in a transaction:

```
import pytest
import sqlite3

@pytest.fixture(params=[':memory:'])
def connection(request):
    return sqlite3.connect(request.param)

@pytest.yield_fixture
def transaction(connection):
    with connection:
        yield connection

def test_insert(transaction):
    transaction.execute('create table test (id integer)')
    transaction.execute('insert into test values (1), (2), (3)')
```

First we have the `connection()` fixture, which uses the special parameter `params`. Instead of using the `:memory:` database in `sqlite3`, we can use a different database name or multiple names as well. That is why `params` is a `list`; the test will be executed for each value in `params`.

The `transaction()` fixture uses the `connection()` to open the database connection, `yield` it to the user of that fixture, and take care of the cleanup after. This could easily have been omitted and done in `transation()` immediately, but it saves an indentation level and it allows you to further customize the connection at a single location if needed.

Lastly, the `test_insert()` function uses the `transaction()` fixture to execute the queries on the database. It is important to note that if we had passed more values to `params`, this test would have been executed for each value.

Print statements and logging

Even though print statements are generally not the most optimal way to debug code, I admit that it is still my default method of debugging. This means that when running and trying tests, I will include many print statements. However, let's see what happens when we try this with `py.test`. Here is the testing code:

```
import os
import sys
import logging

def test_print():
    print('Printing to stdout')
```

```
    print('Printing to stderr', file=sys.stderr)
    logging.debug('Printing to debug')
    logging.info('Printing to info')
    logging.warning('Printing to warning')
    logging.error('Printing to error')
    # We don't want to display os.environ so hack around it
    fail = 'FAIL' in os.environ
    assert not fail
```

The following is the actual output:

```
$ py.test -v T_15_print_statements_and_logging.py
T_15_print_statements_and_logging.py::test_print PASSED  [100%]

================= 1 passed, 1 skipped in 0.06s =================
```

So, all of our print statements and logging got trashed? Well, not really. In this case, `py.test` assumed that it wouldn't be relevant to you, so it ignored the output. But what about the same run with an error?

```
$ FAIL=true py.test -v T_15_print_statements_and_logging.py
============================ FAILURES ============================
_____________________________ test_print ______________________________

    def test_print():
        print('Printing to stdout')
        print('Printing to stderr', file=sys.stderr)
        logging.debug('Printing to debug')
        logging.info('Printing to info')
        logging.warning('Printing to warning')
        logging.error('Printing to error')
        # We don't want to display os.environ so hack around it
        fail = 'FAIL' in os.environ
>       assert not fail
E       assert not True

T_15_print_statements_and_logging.py:15: AssertionError
---------------------- Captured stdout call ----------------------
Printing to stdout
---------------------- Captured stderr call ----------------------
Printing to stderr
----------------------- Captured log call ------------------------
WARNING  root:T_15_print_statements_and_logging.py:11 Printing t
o warning
ERROR    root:T_15_print_statements_and_logging.py:12 Printing t
```

```
o error
=================== short test summary info ====================
FAILED T_15_print_statements_and_logging.py::test_print - ass...
================= 1 failed, 1 skipped in 0.16s =================
```

As we see here, when it's actually useful, we do get the `stdout` and `stderr` output. Additionally, logging with a level of `WARNING` or higher is visible now. `DEBUG` and `INFO` still won't be visible, but we'll see more about that later in this chapter, in the *Logging* section.

There is one big caveat to using print statements for debugging, however: since they write to `stdout` they can break quickly break your doctests. Because `doctest` looks at all generated output, your print statements will be included as expected output.

Plugins

One of the most powerful features of `py.test` is the plugin system. Within `py.test`, nearly everything can be modified using the available hooks; the result of this is that writing plugins is almost simple. Actually, if you've been typing along, you already wrote a few plugins in the previous paragraphs without realizing it. By packaging `conftest.py` in a different package or directory, it becomes a `py.test` plugin. We will explain more about packaging in *Chapter 18*, *Packaging – Creating Your Own Libraries or Applications*.

Generally, it won't be required to write your own plugin because the odds are that the plugins you seek are already available. A small list of plugins can be found on the `py.test` website at `https://pytest.org/latest/plugins.html`, an automatically generated list with plugins here: `https://docs.pytest.org/en/latest/reference/plugin_list.html`, and a longer completely uncurated list (currently over 8,000) can be found through the Python Package Index at `https://pypi.org/search/?q=pytest-`.

By default, `py.test` does cover quite a bit of the desirable features, so you can easily do without plugins, but within the packages that I write myself, I generally default to the following list:

- `pytest-cov`
- `pytest-flake8`
- `pytest-mypy`

By using these plugins, it becomes much easier to maintain the code quality of your project. In order to understand why, we will take a closer look at these packages in the following paragraphs.

pytest-cov

Using the `pytest-cov` package, you can see whether your code is properly covered by tests or not. Internally, it uses the `coverage` package to detect how much of the code is being tested.

Make sure you have `pytest-cov` installed:

```
$ pip3 install pytest-cov
```

To demonstrate the principle, we will check the coverage of a `cube_root` function.

First of all, let's create a `.coveragerc` file with some useful defaults:

```
[report]
# The test coverage you require. Keeping to 100% is not easily
# possible for all projects but it's a good default for new projects.
fail_under = 100

# These functions are generally only needed for debugging and/or
# extra safety so we want to ignore them in the coverage
# requirements
exclude_lines =
    # Make it possible to ignore blocks of code
    pragma: no cover

    # Generally only debug code uses this
    def __repr__

    # If a debug setting is set, skip testing
    if self\.debug:
    if settings.DEBUG

    # Don't worry about safety checks and expected errors
    raise AssertionError
    raise NotImplementedError

    # Do not complain about code that will never run
    if 0:
    if __name__ == .__main__.:
    @abc.abstractmethod

[run]
# Make sure we require that all branches of the code are covered.
# So both the if and the else
branch = True

# No need to require coverage of testing code
omit =
    test_*.py
```

Since Linux and Mac systems hide files starting with a `.` (such as `.coveragerc`), the filename in the GitHub repository is `_coveragerc`. To use the file, you can either choose to copy/rename it, or set the `COVERAGE_RCFILE` environment variable to override the filename.

Which defaults are good for your project is of course a personal decision, but I find the defaults above quite useful. Be sure to read through these instead of blindly copying them, however; perhaps you want to make sure all of your `AssertionErrors` are tested instead of silently ignoring them from the coverage output.

Here is the `cube_root.py` code:

```
def cube_root(n: int) -> int:
    '''
    Returns the cube root of the input number

    Args:
        n (int): The number to cube root

    Returns:
        int: The cube root result
    '''
    if n >= 0:
        return n ** (1 / 3)
    else:
        raise ValueError('A number larger than 0 was expected')
```

And the `T_16_test_cube_root.py` code:

```
import pytest
import cube_root

cubes = (
    (0, 0),
    (1, 1),
    (8, 2),
    (27, 3),
)

@pytest.mark.parametrize('n,expected', cubes)
def test_cube_root(n, expected):
    assert cube_root.cube_root(n) == expected
```

Now, let's see what happens when we run this with coverage enabled:

```
$ py.test --cov-report=html --cov-report=term-missing \
  --cov=cube_root --cov-branch T_16_test_cube_root.py
Name           Stmts   Miss Branch BrPart  Cover   Missing
----------------------------------------------------------
cube_root.py       4      1      2      1    67%   14
Coverage HTML written to dir htmlcov
================= 4 passed, 1 skipped in 0.12s =================
```

What happened here? It looks like we forgot to test some part of the code: line `14` and the branch that goes from line `11` to line `14`. This output isn't all that readable, and that's why we added `--cov-report=html` to get easily readable HTML output in the `htmlcov` directory as well:

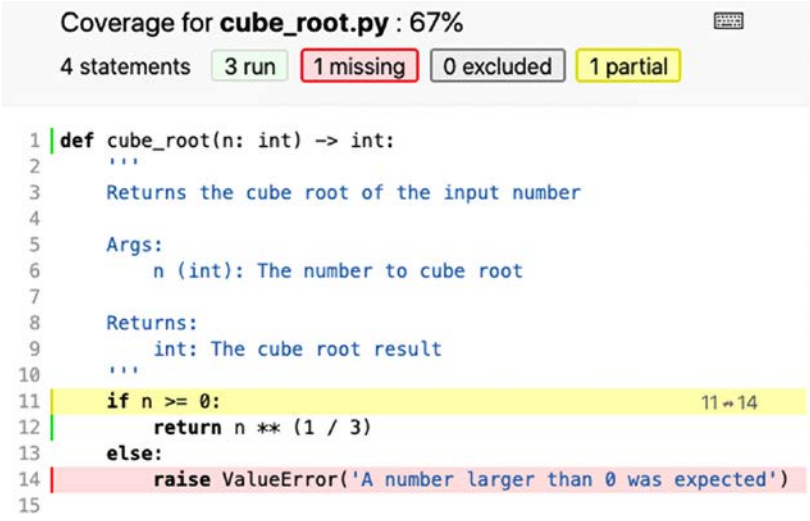


Figure 10.3: Coverage report generated by --cov-report=html

Perfect! So now we know – we forgot to test for values smaller than `0`.

The yellow line (line 11) indicates that only one part of the branch was executed (`(n >= 0) == True`) and not the other (`(n >= 0) == False`). This occurs with `if` statements, loops, and other things where at least one of the branches is not covered. For example, if a loop over an empty array is an impossible scenario, then the test can be partially skipped using a comment:

```
for i in range(10):  # pragma: no branch
```

But since we know the problem, that is, the missing test for `ValueError`, let's add the test case:

```
# Previous test cases omitted
...
```

```
def test_cube_root_below_zero():
    with pytest.raises(ValueError):
        cube_root.cube_root(-1)
```

Then we run the test again:

```
$ py.test --cov-report=html --cov-report=term-missing \
  --cov=cube_root --cov-branch T_17_test_cube_root_subzero.py
Name           Stmts   Miss Branch BrPart  Cover   Missing
----------------------------------------------------------
cube_root.py       4      0      2      0   100%
Coverage HTML written to dir htmlcov
================= 5 passed, 1 skipped in 0.12s =================
```

Perfect! Now we have 100% test coverage of our function. At least, in theory. I can think of several other test cases with different types of values that are not covered. So keep in mind that 100% test coverage is still no guarantee for bug-free code.

But what if we have a branch that really doesn't need testing because it is intentionally not implemented? If we raise a `NotImplementedError` instead of raising a `ValueError` for values below `0`, we also get 100% test coverage without adding that test.

This is because we added `raise NotImplementedError` to the ignore list in the `.coveragerc` file. Even if we were to test for the `NotImplementedError` in the test file, the coverage report would still ignore the line.

pytest-flake8

Code quality testing tools are very useful for making your code readable, consistent, and `pep8` compliant. The `pytest-flake8` plugin automatically executes these checks before running the actual tests. To install it, simply execute this line:

```
$ pip3 install pytest-flake8
```

We already installed `pytest-flake8` earlier in this chapter because the default configuration for the code in this book depends on it.

Now we'll create some bad code:

```
import os
def test(a,b):
    return c
```

After that, we can check the code using the `pytest-flake8` plugin by adding it to the `pytest.ini`, or by running `py.test` like this:

```
$ py.test --flake8 T_18_bad_code.py
========================== FAILURES ==========================
_____________ FLAKE8-check(ignoring W391 E402 F811) _____________
T_18_bad_code.py:1:1: F401 'os' imported but unused
T_18_bad_code.py:2:1: E302 expected 2 blank lines, found 0
T_18_bad_code.py:2:11: E231 missing whitespace after ','
T_18_bad_code.py:3:12: F821 undefined name 'c'

---------------------- Captured log call -----------------------
WARNING  flake8.options.manager:manager.py:207 option --max-complexity: please
update from optparse string 'type=' to argparse callable 'type=' -- this will
be an error in the future
WARNING  flake8.checker:checker.py:119 The multiprocessing module is not
available. Ignoring --jobs arguments.
```

The output of `pytest-flake8` is, as expected, very similar to the output from the `flake8` command that is called internally and combines the `pyflakes` and `pep8` commands to test code quality.

Depending on your situation, you might opt for having the code quality tests before you commit to the repository, or you could only run it on-demand if code quality isn't that important to you. After all, while code quality considerations are important, it does not mean the code does not function without them, and a good editor will already notify you of code quality issues while typing.

pytest-mypy

The `pytest-mypy` plugin runs the `mypy` static type checker, which uses the type hints to check if the input and output are as expected. First, we need to install it using `pip`:

```
$ pip3 install pytest-mypy
```

When we apply this to our `cube_root.py` file, we can already see a possible error:

```
$ py.test --mypy cube_root.py
========================== FAILURES ==========================
_____________________________ cube_root.py _____________________________
12: error: Incompatible return value type (got "float", expected
 "int")
```

As opposed to the `cube.py` file, which will return an `int` when given an `int`, the cube root of a number does not have to be an integer when an integer is passed. While the cube root of `8` is `2`, the cube root of `4` returns a floating-point number of approximately `1.587`.

This is an error that is easily overlooked without a tool such as `mypy`.

Configuring plugins

To make sure that all the plugins get executed and to configure them, simply add the settings to the `pytest.ini` file. The following example can be a reasonable default for development, but for production releases, you will probably want to take care of the `UnusedImport` warnings.

`pytest.ini`

```
[pytest]
python_files =
    your_project_source/*.py
    tests/*.py

addopts =
    --doctest-modules
    --cov your_project_source
    --cov-report term-missing
    --cov-report html
    --flake8
    --mypy

# W391 is the error about blank lines at the end of a file
flake8-ignore =
    *.py W391
```

Using the `addopts` setting in the `pytest.ini`, you can add options to the `py.test` command as if you had added them to the command while running.

When debugging to find out why a test is failing, it can be useful to simply look at the first test that fails. The `py.test` module offers both a `-x`/`--exitfirst` flag to stop after the first failure and `--maxfail=n` to stop after *n* failures.

Additionally, the `--ff`/`--failed-first` option is useful to run the previously failed tests first.

Or you can use the `--lf`/`--last-failed` option to only run previously failed tests.

Now that we have a good understanding of the `py.test` possibilities, it is time to continue writing tests. Next up is the subject of faking objects using `mock`.

Mock objects

When writing tests, you will often find that you are not only testing your own code, but also the interaction with external resources, such as hardware, databases, web hosts, servers, and others. Some of these can be run safely, but certain tests are too slow, too dangerous, or even impossible to run. In those cases, mock objects are your friends; they can be used to fake anything, so you can be certain that your code still returns the expected results without having any variation from external factors.

Using unittest.mock

The `unittest.mock` library provides two base objects, `Mock` and `MagicMock`, to easily mock any external resources. The `Mock` object is just a general generic mock object and `MagicMock` is mostly the same, but it has all the Python magic methods such as `__contains__` and `__len__` defined. In addition to this, it can make your life even easier. This is because in addition to creating mock objects manually, it is possible to patch objects directly using the `patch` decorator/context manager.

The following function uses `random` to return `True` or `False` with probabilities governed by a certain probability distribution. Due to the random nature of a function like this, it is notoriously difficult to test, but not with `unittest.mock`. With the use of `unittest.mock`, it's easy to get repeatable results:

```
from unittest import mock
import random

@mock.patch('random.random')
def test_random(mock_random):
    # Specify our mock return value
    mock_random.return_value = 0.1
    # Test for the mock return value
    assert random.random() == 0.1
    assert mock_random.call_count == 1

def test_random_with():
    with mock.patch('random.random') as mock_random:
        mock_random.return_value = 0.1
        assert random.random() == 0.1
```

Wonderful, isn't it? Without having to modify the original code, we can make sure that `random.random()` now returns `0.1` instead of some random number. If you have an `if` statement in your code so it only runs 10% of the time (`if random.random() < 0.1`), you can now test explicitly what happens in both cases of the `if`.

The possibilities with mock objects are nearly endless. They vary from raising exceptions on access to faking entire APIs and returning different results on multiple calls. For example, let's fake deleting a file:

```
import os
from unittest import mock

def delete_file(filename):
    while os.path.exists(filename):
        os.unlink(filename)
```

```
@mock.patch('os.path.exists', side_effect=(True, False, False))
@mock.patch('os.unlink')
def test_delete_file(mock_exists, mock_unlink):
    # First try:
    delete_file('some non-existing file')

    # Second try:
    delete_file('some non-existing file')
```

Quite a bit of magic in this example! The `side_effect` parameter tells `mock` to return those values in that sequence, making sure that the first call to `os.path.exists` returns `True` and the other two return `False`. The `mock.patch` call without specific arguments simply returns a callable that does nothing and accepts anything.

Using py.test monkeypatch

The `monkeypatch` object in `py.test` is a fixture that allows mocking as well. While it may seem useless after seeing the possibilities with `unittest.mock`, in summary, it's not. Some of the functionality does overlap, but while `unittest.mock` focuses on controlling and recording the actions of an object, the `monkeypatch` fixture focuses on simple and temporary environmental changes. Some examples of these are given in the following list:

- Setting and deleting attributes using `monkeypatch.setattr` and `monkeypatch.delattr`
- Setting and deleting dictionary items using `monkeypatch.setitem` and `monkeypatch.delitem`
- Setting and deleting environment variables using `monkeypatch.setenv` and `monkeypatch.delenv`
- Inserting an extra path to `sys.path` before all others using `monkeypatch.syspath_prepend`
- Changing the directory using `monkeypatch.chdir`

To undo all modifications, simply use `monkeypatch.undo`. Naturally, at the end of your test function, `monkeypatch.undo()` will be called automatically.

For example, let's say that for a certain test, we need to work from a different directory. With `mock`, your options would be to mock pretty much all file functions, including the `os.path` functions, and even in that case, you will probably forget about a few. So, it's definitely not useful in this case. Another option would be to put the entire test into a `try...finally` block and just do an `os.chdir` before and after the testing code. This is quite a good and safe solution, but it's a bit of extra work, so let's compare the two methods:

```
import os

def test_chdir_monkeypatch(monkeypatch):
    monkeypatch.chdir('/')
    assert os.getcwd() == '/'
```

```
def test_chdir():
    original_directory = os.getcwd()
    try:
        os.chdir('/')
        assert os.getcwd() == '/'
    finally:
        os.chdir(original_directory)
```

They effectively do the same, but one needs a single line of code to temporarily change directory whereas the other needs four, or five if you count the `os` import as well. All of these can easily be worked around with a few extra lines of code, of course, but the simpler the code is, the fewer mistakes you can make and the more readable it is.

Now that we know how to fake objects, let's look at how we can run our tests on multiple platforms simultaneously using `tox`.

Testing multiple environments with tox

Now that we have written our tests and are able to run them for our own environment, it's time to make sure that others can easily run the tests too. `tox` can create sandboxed environments for all specified Python versions (assuming they are installed) and runs them automatically and in parallel if needed. This is especially useful to test if your dependency specification is up to date. While you may have a lot of packages installed in your local environment, someone else might not have those packages.

Getting started with tox

Before we can do anything, we need to install the `tox` command. A simple pip install will suffice:

```
$ pip3 install --upgrade tox
```

After the install, we can start by creating a `tox.ini` file to specify what we want to run. The easiest way is by using `tox-quickstart`, but if you already have a functioning `tox.ini` from a different project you can easily copy and modify that:

```
$ tox-quickstart
Welcome to the tox 3.20.1 quickstart utility.
This utility will ask you a few questions and then generate a simple
configuration file to help get you started using tox.
Please enter values for the following settings (just press Enter to accept a
default value, if one is given in brackets).

What Python versions do you want to test against?
            [1] py37
            [2] py27, py37
            [3] (All versions) py27, py35, py36, py37, pypy, jython
            [4] Choose each one-by-one
```

```
> Enter the number of your choice [3]: 1
What command should be used to test your project? Examples:
            - pytest
            - python -m unittest discover
            - python setup.py test
            - trial package.module
> Type the command to run your tests [pytest]:
What extra dependencies do your tests have?
default dependencies are: ['pytest']
> Comma-separated list of dependencies: pytest-flake8,pytest-mypy,pytest-cov
Finished: ./tox.ini has been created. For information on this file, see
https://tox.readthedocs.io/en/latest/config.html
Execute 'tox' to test your project.
```

Now we have our first `tox` configuration finished. The `tox-quickstart` command has made a `tox.ini` file with a few sane defaults.

When looking at the output of `tox-quickstart`, you might be wondering why newer Python versions are not listed. The reason is that the Python versions are hardcoded in the `tox-quickstart` command at the time of writing. This issue is expected to be solved in the near future but should not be a big issue in either case, as the versions can be changed in the `tox.ini` file quite easily.

The tox.ini config file

The `tox.ini` file is very basic by default:

```
[tox]
envlist = py37

[testenv]
deps =
    pytest-flake8
    pytest-mypy
    pytest-cov
    pytest
commands =
    pytest
```

The `tox.ini` file usually consists of two main types of sections, the `tox` and `testenv` sections.

The `tox` section configures the `tox` command itself and specifies options such as:

- `envlist`: Specifies the default list of environments to run, can be overridden by running `tox -e <env>`.
- `requires`: Specifies which packages (and specific versions) are required alongside `tox`. This can be useful for specifying a specific `setuptools` version so your package can be installed correctly.
- `skip_missing_interpreters`: A very useful feature that allows you to test all available environments on your system but skip the ones that are not installed.

The `testenv` section configures your actual environment. Some of the most useful options are:

- `basepython`: The Python executable to run, useful if your Python binary has a non-standard name but more commonly useful when using custom environment names.
- `commands`: Commands to run when testing, in our case `pytest`.
- `install_command`: Command to run to install the package, defaults to `python -m pip install {opts} {packages}(ARGV)`.
- `allowlist_externals`: Which external commands such as `make`, `rm`, `ls`, and `cd` to allow so they can be run from the package or the scripts.
- `changedir`: Switch to a specific directory before running tests; to the directory containing the tests, for example.
- `deps`: Which Python packages to install, uses the `pip` command syntax. A `requirements.txt` file can be specified through `-rrequirements.txt`.
- `platform`: Restrict the environment to a specific value of `sys.platform`.
- `setenv`: Set environment variables, very useful to let tests know that they are being run from `tox`, for example.
- `skipsdist`: With this flag enabled, you can test a regular directory instead of only installable Python packages.

The most interesting part of the configuration is the `testenv` section prefix. While the `testenv` options above can be configured globally for all environments, you can use a section such as `[testenv:my_custom_env]` to only apply to your custom environment. In those cases, you will need to specify the `basepython` option so `tox` knows what to execute.

Additionally to a single environment, you can also expand the pattern to configure multiple environments simultaneously with a pattern such as `[testenv:py{27,38}]` to specify both the `py27` and `py38` environments.

Expansions such as `py{27,38}` are also possible for all other options, so to specify a whole list of Python environments, you could do:

```
envlist = py27, py3{7,8,9}, docs, coverage, flake8
```

Furthermore, all options in the `tox.ini` also allow for variable interpolation based on a whole range of available variables, such as `{envname}`, but also based on options from other environments. The next example shows how to copy the `basepython` variable from the `py39` environment:

```
[testenv:custom_env]
basepython = {[py39]basepython}
```

Naturally, interpolating from environment variables is also possible:

```
{env:NAME_OF_ENV_VARIABLE}
```

With an optional default:

```
{env:NAME_OF_ENV_VARIABLE:some default value}
```

Running tox

Now that we know some of the basic config options for `tox`, let's run a simple test to illustrate how convenient it can be.

First we need to create a `tox.ini` file to configure `tox`:

```
[tox]
envlist = py3{8,9}
skipsdist = True

[testenv]
deps =
    pytest
commands =
    pytest test.py
```

Next, we will create a `test.py` file containing the Python 3.9 `dict` merge operator:

```
def test_dict_merge():
    a = dict(a=123)
    b = dict(b=456)
    assert a | b
```

Now when running `tox`, it will show us that this syntax failed on Python 3.8 and works on Python 3.9 as expected:

```
$ tox
py38 installed: ...
py38 run-test: commands[0] | pytest test.py
==================== test session starts ======================
============================ FAILURES ============================
_____________________ test_dict_merge _____________________
```

```
    def test_dict_merge():
        a = dict(a=123)
        b = dict(b=456)
>       assert a | b
E       TypeError: unsupported operand type(s) for |: 'dict' and 'dict'
...
ERROR:   py38: commands failed
  py39: commands succeeded
```

That all looks good – an error for Python 3.8 and a fully working Python 3.9 run. This is where `tox` is really useful; you can easily test multiple Python versions and multiple environments simultaneously, even in parallel if you use the `tox -p<processes>` parameter. And best of all, since it creates a completely blank Python environment, you are testing your requirements specification as well.

Now that we know how to run our tests on multiple Python environments simultaneously, it is time to continue with `logging`, the last section of this chapter. While a simple print statement can be very useful in debugging, when working on larger or distributed systems it is often not the most convenient option anymore. This is where the `logging` module can help you greatly to debug your issues.

Logging

The Python `logging` module is one of those modules that are extremely useful, but it tends to be very difficult to use correctly. The result is often that people just disable logging completely and use `print` statements instead. While it is somewhat understandable, this is a waste of the very extensive logging system in Python.

The Python `logging` module is largely based on the Java `log4j` library so it might be familiar to you if you've written Java before. That is also one of the biggest problems with the `logging` module in my opinion; Python is not Java and the `logging` module feels pretty un-Pythonic because of it. That does not make it a bad library, but it takes a little effort to get used to its design.

The most important objects of the `logging` module are the following:

- **Logger**: The actual logging interface
- **Handler**: This processes the log statements and outputs them
- **Formatter**: This formats the input data into a string
- **Filter**: This allows filtering of certain messages

Within these objects, you can set the logging levels to one of the default levels:

- `CRITICAL: 50`
- `ERROR: 40`
- `WARNING: 30`
- `INFO: 20`
- `DEBUG: 10`
- `NOTSET: 0`

The numbers are the numeric values of these log levels. While you can generally ignore them, the order is obviously important while setting the minimum level. Also, when defining custom levels, you will have to overwrite existing levels if they have the same numeric value.

Configuration

There are several ways to configure the logging system, ranging from pure code to JSON files or even remote configuration. The examples will use parts of the `logging` module discussed later in this chapter, but the usage of the config system is all that matters here. If you are not interested in the internal workings of the `logging` module, you should be able to get by with just this paragraph of the *Logging* section.

Basic logging configuration

The most basic logging configuration is, of course, no configuration, but that will not get you much useful output:

```
import logging

logging.debug('debug')
logging.info('info')
logging.warning('warning')
logging.error('error')
logging.critical('critical')
```

With the default log level, you will only see a `WARNING` and up:

```
$ python3 T_23_logging_basic.py
WARNING:root:warning
ERROR:root:error
CRITICAL:root:critical
```

A quick and easy start for a configuration is `logging.basicConfig()`. I recommend using this if you just need some quick logging for a script you're writing, but not for a full-blown application. While you can configure pretty much anything you wish, once you get a more complicated setup, there are usually more convenient options. We will talk more about that in later paragraphs, but first, we have `logging.basicConfig()`, which creates a `logging.StreamHandler` that is added to the root logger and configured to write all output to `sys.stderr` (standard error). Note that if the root logger already has handlers, the `logging.basicConfig()` function does nothing (unless `force=True`).

If no log handlers are configured for the root logger, the logging functions (`debug()`, `info()`, `warning()`, `error()`, and `critical()`) will automatically call `logging.basicConfig()` to set up a logger for you. This means that if you have a log statement before your `logging.basicConfig()` call, it will be ignored.

To illustrate the usage of `basicConfig()` with a few customizations:

```
import logging

log_format = '%(levelname)-8s %(name)-12s %(message)s'

logging.basicConfig(
    filename='debug.log',
    format=log_format,
    level=logging.DEBUG,
)

formatter = logging.Formatter(log_format)
handler = logging.StreamHandler()
handler.setLevel(logging.WARNING)
handler.setFormatter(formatter)
logging.getLogger().addHandler(handler)
```

Now we can test the code:

```
logging.debug('debug')
logging.info('info')
some_logger = logging.getLogger('some')
some_logger.warning('warning')
some_logger.error('error')
other_logger = some_logger.getChild('other')
other_logger.critical('critical')
```

This will give us the following output on our screen:

```
$ python3 T_24_logging_basic_formatted.py
WARNING  some         warning
ERROR    some         error
CRITICAL some.other   critical
```

And here is the output in the `debug.log` file:

```
DEBUG    root         debug
INFO     root         info
WARNING  some         warning
ERROR    some         error
CRITICAL some.other   critical
```

This configuration shows how log outputs can be configured with separate configurations, log levels, and, if you choose so, formatting. It tends to become unreadable though, which is why it's usually a better idea to use `basicConfig` only for simple configurations that don't involve multiple handlers.

Dictionary configuration

`dictConfig` makes it possible to name all parts so that they can be reused easily, for example, a single formatter for multiple loggers and handlers. Let's rewrite our previous configuration using `dictConfig`:

```
from logging import config

config.dictConfig({
    'version': 1,
    'formatters': {
        'standard': {
            'format': '%(levelname)-8s %(name)-12s %(message)s',
        },
    },
    'handlers': {
        'file': {
            'filename': 'debug.log',
            'level': 'DEBUG',
            'class': 'logging.FileHandler',
            'formatter': 'standard',
        },
        'stream': {
            'level': 'WARNING',
            'class': 'logging.StreamHandler',
            'formatter': 'standard',
        },
    },
    'loggers': {
        '': {
            'handlers': ['file', 'stream'],
            'level': 'DEBUG',
        },
    },
})
```

You can probably see the similarities with the `logging.basicConfig()` call we used earlier. It is merely a different syntax for a `logging` configuration.

The nice thing about the dictionary configuration is that it's very easy to extend and/or overwrite the logging configuration. For example, if you want to change the formatter for all of your logging, you can simply change the `standard` formatter or even loop through `handlers`.

JSON configuration

Since `dictConfig` takes any type of dictionary, it is actually quite simple to implement a different type of reader employing JSON or YAML files. This is especially useful as they tend to be a bit friendlier toward non-Python programmers. As opposed to Python files, they are easily readable and writable from outside of Python.

Let's assume that we have a `T_26_logging_json_config.json` file such as the following:

```
{
    "version": 1,
    "formatters": {
        "standard": {
            "format": "%(levelname)-8s %(name)-12s %(message)s"
        }
    },
    "handlers": {
        "file": {
            "filename": "debug.log",
            "level": "DEBUG",
            "class": "logging.FileHandler",
            "formatter": "standard"
        },
        "stream": {
            "level": "WARNING",
            "class": "logging.StreamHandler",
            "formatter": "standard"
        }
    },
    "loggers": {
        "": {
            "handlers": ["file", "stream"],
            "level": "DEBUG"
        }
    }
}
```

We can simply use this code to read the config:

```
import json
from logging import config

with open('T_26_logging_json_config.json') as fh:
    config.dictConfig(json.load(fh))
```

Naturally, you could use any source that can generate a `dict`, but be mindful of the source. Since the `logging` module will import the specified class, it can be a potential security risk.

ini file configuration

The file configuration is probably the most readable format for non-programmers. It uses the `ini-style` configuration format and uses the `configparser` module internally. The downside is that it is perhaps a little verbose, but it is clear enough and makes it easy to combine several configuration files without us having to worry too much about overwriting other configurations. Having said that, if `dictConfig` is an option, then it is most likely a better option. This is because `fileConfig` is slightly limited and awkward at times. Just look at the handlers as an example:

```
[formatters]
keys=standard

[handlers]
keys=file,stream

[loggers]
keys=root

[formatter_standard]
format=%(levelname)-8s %(name)-12s %(message)s

[handler_file]
level=DEBUG
class=FileHandler
formatter=standard
args=('debug.log',)

[handler_stream]
level=WARNING
class=StreamHandler
formatter=standard
args=(sys.stderr,)

[logger_root]
handlers=file,stream
level=DEBUG
```

Reading the files is extremely easy though:

```
from logging import config

config.fileConfig('T_27_logging_ini_config.ini')
```

One thing to make note of, however, is that if you look carefully, you will see that this config is slightly different from the other configs. With `fileConfig` you can't just use keyword arguments alone. The args is required for both `FileHandler` and `StreamHandler`.

The network configuration

The network configuration is a rarely used but very convenient way to configure your loggers across multiple processes. This type of configuration is quite esoteric and if you have no use for such a setup, feel free to skip to the *Logger* section.

The major caveat of the network configuration is that it can be dangerous because it allows you to configure your logger on the fly while your application/script is still running. The dangerous part is that the config is (partially) read by using the `eval` function, which allows people to potentially execute code within your application remotely. Even though `logging.config.listen` only listens to local connections, it can still be dangerous if you execute the code on a shared/unsafe host where others can run code as well.

If your system is unsafe, you can pass `verify` as a callable argument to `listen()`, which could implement signature verification or encryption of the configurations before they are evaluated. By default, the `verify` function is analogous to `lambda config: config`. As the most simple verification method, you could use something along these lines:

```
def verify(config):
    if config.pop('secret', None) != 'some secret':
        raise RuntimeError('Access denied')
    return config
```

To show the workings of the network configuration, we need two scripts. One script will continuously print a few messages to the loggers and the other will change the logging configuration. We will start with the same test code that we had before, but keep it running in an endless loop with a `sleep` in between:

```
import sys

def receive():
    import time
    import logging
    from logging import config

    listener = config.listen()
    listener.start()

    try:
        while True:
            logging.debug('debug')
            logging.info('info')
```

```
            some_logger = logging.getLogger('some')
            some_logger.warning('warning')
            some_logger.error('error')
            other_logger = some_logger.getChild('other')
            other_logger.critical('critical')

            time.sleep(5)

    except KeyboardInterrupt:
        # Stop listening and finish the listening thread
        config.stopListening()
        listener.join()

def send():
    import os
    import struct
    import socket
    from logging import config

    ini_filename = os.path.splitext(__file__)[0] + '.ini'
    with open(ini_filename, 'rb') as fh:
        data = fh.read()

    # Open the socket
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    # Connect to the server
    sock.connect(('127.0.0.1',
                  config.DEFAULT_LOGGING_CONFIG_PORT))
    # Send the magic logging packet
    sock.send(struct.pack('>L', len(data)))
    # Send the config
    sock.send(data)
    # And close the connection again
    sock.close()

if __name__ == '__main__':
    if sys.argv[-1] == 'send':
        send()
    elif sys.argv[-1] == 'receive':
```

```
        receive()
    else:
        print(f'Usage: {sys.argv[0]} [send/receive]')
```

Now we need to run both of the scripts at the same time. First, we start the `receive` script, which will start outputting data:

```
$ python3 T_28_logging_network_config.py receive
WARNING:some:warning
ERROR:some:error
CRITICAL:some.other:critical
# The send command was run here
WARNING  some         warning
ERROR    some         error
CRITICAL some.other   critical
```

In the meantime, we run the `send` command:

```
$ python3 T_28_logging_network_config.py send
```

As you can see, the logging configuration was updated while the code was still running. This can be very useful for long-running scripts that you need to debug but don't want to restart.

In addition to the output on the screen, the extra output was sent to the `debug.log` file, which looks something like this now:

```
DEBUG    root         debug
INFO     root         info
WARNING  some         warning
ERROR    some         error
CRITICAL some.other   critical
```

This allows you to redirect mostly useless debug output to a separate log file while still keeping the most important messages on the screen.

Logger

The main object that you will be using all the time with the `logging` module is the `Logger` object. This object contains all the APIs that you will need to do the actual logging. Most are simple enough but some require attention.

First of all, loggers inherit the parent settings by default. As we have seen previously, with the `propagate` setting, by default, all settings will propagate from the parent. This is really useful when incorporating loggers within your files.

Assuming your modules are using sane names and import paths, I recommend the following style of naming your loggers:

```
import logging

logger = logging.getLogger(__name__)

class MyClass(object):
    def __init__(self, count):
        self.logger = logger.getChild(self.__class__.__name__)
```

By using this style, your loggers will get names such as `main_module.sub_module.ClassName`. Not only does this make your logs easier to read, but also it is easily possible to enable or disable logging per module with the propagation of log settings. To create a new log file that logs everything from `main_module.sub_module`, we can simply do this:

```
import logging

logger = logging.getLogger('main_module.sub_module')
logger.addHandler(logging.FileHandler('sub_module.log'))
```

Alternatively, you can configure it using your chosen configuration option, of course. The relevant point is that with sub-loggers, you have very fine-grained control over your loggers.

This includes increasing the log level:

```
import logging

logger = logging.getLogger('main_module.sub_module')
logger.setLevel(logging.DEBUG)
```

Usage

The usage of the `Logger` object is mostly identical to that of the bare `logging` module, but `Logger` actually supports a bit more. This is because the bare `logging` module just calls the functions on the root logger. The `Logger` object has a few very useful properties, although most of these are undocumented in the library:

- `propagate`: Whether to pass events to this logger or to the handlers of the parent loggers. Without this, a log message to `main_module.sub_module` won't be logged by `main_module`.
    - The `handle` method will keep looking for parent handlers as long as those loggers have `propagate` set to `true`, which is the default.
- `filters`: These are the filters attached to the logger. They can be set through `addFilter` and `removeFilter`. To see whether a message will be filtered, the `filter` method can be used.

- `disabled`: By setting this property, it's possible to disable a certain logger. The regular API only allows the disabling of all loggers below a certain level. This offers some fine-grained control.
- `handlers`: These are the handlers attached to the logger. They can be added through `addHandler` and `removeHandler`. The existence of any (inherited) handlers can be checked through the `hasHandlers` function.
- `level`: This is really an internal one as it simply has a numeric value and not a name. But beyond that, it doesn't take inheritance into account, so it's better to avoid the property and use the `getEffectiveLevel` function instead. To check whether the setting is enabled for a `DEBUG`, for example, you can simply do `logger.isEnabledFor(logging.DEBUG)`. Setting the property is possible through the `setLevel` function, of course.
- `name`: As this property's name suggests, it is very useful for your own reference, of course.

Now that you know about the properties, it is time to discuss the logging functions themselves. The functions you will use most often are the `log`, `debug`, `info`, `warning`, `error`, and `critical` log functions. They can be used quite simply, but they support string formatting as well, which is very useful:

```
import logging

logger = logging.getLogger()
exception = 'Oops...'
logger.error('Some horrible error: %r', exception)
```

Formatting

When seeing the previous examples, you might wonder why we use `logger.error('error: %r', error)` instead of regular string formatting with f-strings, `%`, or `string.format` instead. The reason is that when parameters are used instead of preformatted strings, the handler gets them as parameters. The result is that you can group log messages by the original string, which is what tools such as Sentry (`https://github.com/getsentry/sentry`) use.

There is more to it, however. In terms of parameters, `*args` are only for string formatting, but it's possible to add extra parameters to a log object using the `extra` keyword parameter:

```
import logging

logger = logging.getLogger()
logger.error('simple error', extra=dict(some_variable='my value'))
```

These `extra` parameters can be used in the logging formatter to display extra information just like the standard formatting options:

```
import logging

logging.basicConfig(format='%(some_variable)s: %(message)s')
logger = logging.getLogger()
logger.error('the message', extra=dict(some_variable='my value'))
```

This results in the following:

```
$ python3 T_30_formatting.py
simple error
my value: the message
```

However, one of the most useful features is the support for exceptions:

```
import logging

logging.basicConfig()
logger = logging.getLogger()

try:
    raise RuntimeError('some runtime error')
except Exception as exception:
    logger.exception('Got an exception: %s', exception)

logger.error('And an error')
```

This results in a stack trace for the exception, but it will not kill the code:

```
$ python3 T_31_exception.py
ERROR:root:Got an exception: some runtime error
Traceback (most recent call last):
  File "T_31_exception.py", line 7, in <module>
    raise RuntimeError('some runtime error')
RuntimeError: some runtime error
ERROR:root:And an error
```

Modern formatting using f-strings and str.format

The Python `logging` module is still largely based on the "old" formatting syntax and doesn't have much support for `str.format`. For the `Formatter` itself, you can easily use the new style formatting, but that's ultimately mostly useless since you rarely modify the `Formatter` and mainly need formatting when logging messages instead.

Regardless, the syntax is simple enough to enable:

```
import logging

formatter = logging.Formatter('{levelname} {message}', style='{')
handler = logging.StreamHandler()
handler.setFormatter(formatter)

logging.error('formatted message?')
```

Which results in:

```
$ python3 T_32_str_format.py
ERROR:root:formatted message?
```

For actual messages requiring formatting, we need to implement something ourselves, however. A logging adapter is the easiest solution:

```
import logging

class FormattingMessage:
    def __init__(self, message, kwargs):
        self.message = message
        self.kwargs = kwargs

    def __str__(self):
        return self.message.format(**self.kwargs)

class FormattingAdapter(logging.LoggerAdapter):
    def process(self, msg, kwargs):
        msg, kwargs = super().process(msg, kwargs)
        return FormattingMessage(msg, kwargs), dict()

logger = FormattingAdapter(logging.root, dict())
logger.error('Hi {name}', name='Rick')
```

When executing the code, this results in the following output:

```
$ python3 T_33_logging_format.py
Hi Rick
```

The solution still doesn't look that pretty in my opinion, but it works. Because the formatting of log messages cannot be overridden easily in the `logging` module, we have created a separate `FormattingMessage` that formats itself whenever `str(message)` is called. This way we can override the formatting using a simple `logging.LoggerAdapter` without having to replace large portions of the `logging` library.

Please note that if you want to send the value of `kwargs` to a logger such as Sentry, you will need to make sure the order of operations is correct, since this method cannot pass the `kwargs` along or the standard log formatter would complain.

Additionally, you might be wondering why we used the `FormattingMessage` instead of running `msg.format(**kwargs)` in the `process()` method. The reason is that we want to avoid string formatting for as long as possible.

If the logger doesn't have an active handler or the handler ignores messages of this level, it means we would have done useless work. Depending on the implementation, string formatting can be a very heavy operation and the `logging` system is meant to be as light as possible until enabled.

Logging pitfalls

The logging propagation is one of the most useful features and also the biggest problem with the `logging` module. We have already seen how logging settings are inherited from parent loggers, but what if you override them? Well, let's find out:

```
import logging

a = logging.getLogger('a')
ab = logging.getLogger('a.b')

ab.error('before setting level')
a.setLevel(logging.CRITICAL)
ab.error('after setting level')
```

When we run this code, we get this output:

```
$ python3 T_34_logging_pitfalls.py
before setting level
```

In this case it's obvious that the `a.setLevel(...)` caused the issue, but if that happens in some external code that you didn't know about, you could be searching for a long time.

And the reverse can also happen; an explicit level on a logger will ignore your parent level:

```
import logging

a = logging.getLogger('a')
ab = logging.getLogger('a.b')
ab.setLevel(logging.ERROR)

ab.error('before setting level')
a.setLevel(logging.CRITICAL)
ab.error('after setting level')
```

When we execute this, we notice that setting the level is completely ignored:

```
$ python3 T_35_logging_propagate_pitfalls.py
before setting level
after setting level
```

Once again, not a problem in this case, but if that happens in some external library without your knowledge it can certainly cause a headache.

Debugging loggers

The most important rule about loggers is that they inherit the settings from the parent loggers unless you override them. If your logging isn't working as you expect it, most of the time it's caused by some inheritance issue and that can be difficult to debug.

The logging flow according to the Python manual looks like this:

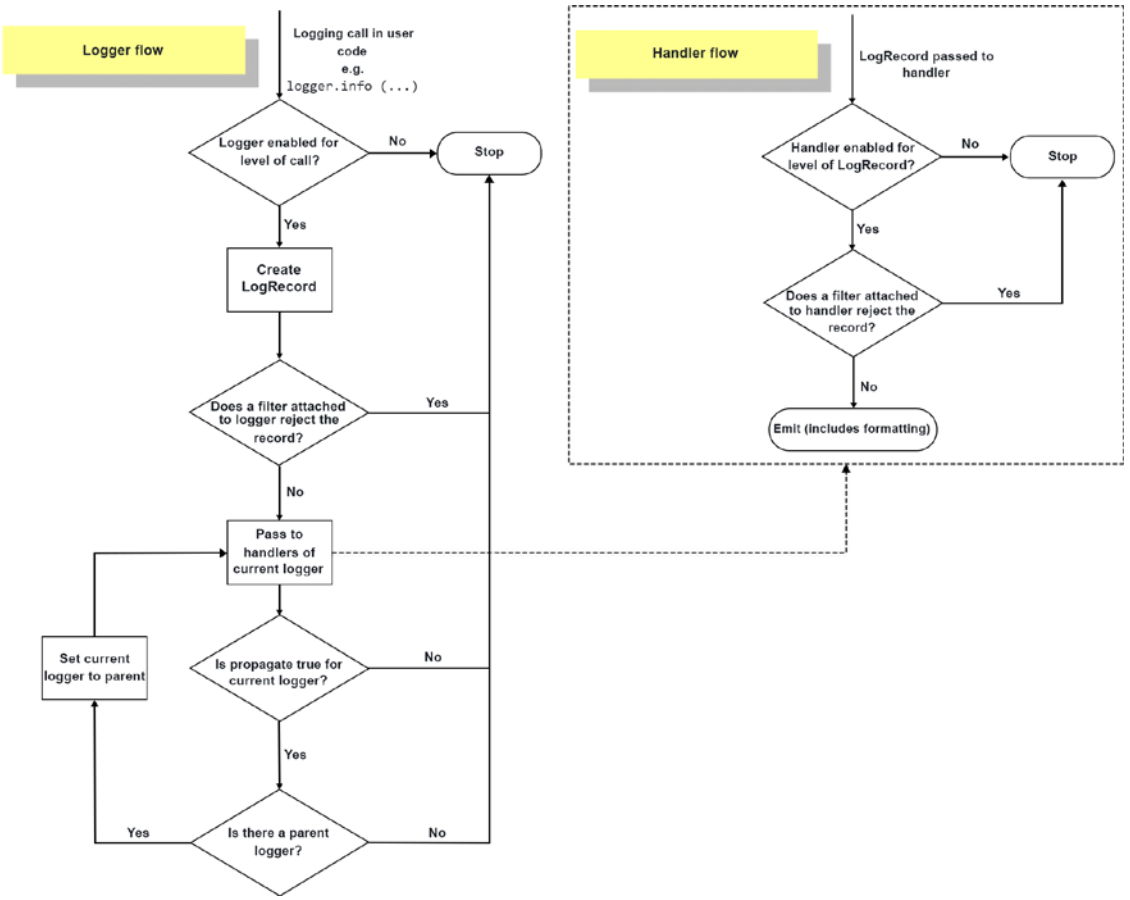


Figure 10.4: Logging flow. Copyright © 2001-2021 Python Software Foundation; All Rights Reserved

Now that we know how the logging flow is supposed to go, we can start creating a method to show our current logger structure and the settings:

```
import logging

def get_handlers(logger):
    handlers = []
    # Walk through the loggers and their parents recursively to
    # fetch the handlers
```

```
    while logger:
        handlers += logger.handlers

        if logger.propagate:
            logger = logger.parent
        else:
            break

    # Python has a lastResort handler in case no handlers are
    # defined
    if not handlers and logging.lastResort:
        handlers.append(logging.lastResort)

    return handlers

def debug_loggers():
    logger: logging.Logger
    for name, logger in logging.root.manager.loggerDict.items():
        # Placeholders are loggers without settings
        if isinstance(logger, logging.PlaceHolder):
            print('skipping', name)
            continue

        level = logging.getLevelName(logger.getEffectiveLevel())
        handlers = get_handlers(logger)
        print(f'{name}@{level}: {handlers}')

if __name__ == '__main__':
    a = logging.getLogger('a')
    a.setLevel(logging.INFO)

    handler = logging.StreamHandler()
    handler.setLevel(logging.INFO)
    ab = logging.getLogger('a.b')
    ab.setLevel(logging.DEBUG)
    ab.addHandler(handler)

    debug_loggers()
```

The `get_handlers()` function recursively walks through a logger and all of its parents to collect all propagated handlers. The `debug_loggers()` function walks through the internal config of the `logging` module to list all configured loggers and fetch the matching handlers through `get_handlers()`.

This is just a basic debugging function of course, but it can really help you when you're wondering why your logging is not working as expected. The output looks something like this:

```
$ python3 T_36_logger_debugging.py
a@INFO: [<_StderrHandler <stderr> (WARNING)>]
a.b@DEBUG: [<StreamHandler <stderr> (INFO)>]
```

Now we can see that the `a` logger has level `INFO` but only has a handler at a `WARNING` level. So, none of our `INFO` messages will show. Similarly, the `a.b` logger has a `DEBUG` level but a handler at level `INFO` so it will only show `INFO` and higher levels.

Exercises

Now that you have seen several testing and logging options, it's time to try it yourself.

A few challenges:

- Create a function that tests the doctests of a given function/class.
- For a greater challenge, create a function that recursively tests all doctests of every function and class in a given module.
- Create a `py.test` plugin that checks if all tested files have file-level documentation. Hint: use `pytest_collect_file`.
- Create a custom `tox` environment to run `flake8` or `mypy` on your project.
- Create a `LoggerAdapter` that combines multiple messages into a single message based on some task ID. This can be useful when debugging long-running tasks.

Example answers for these exercises can be found on GitHub: `https://github.com/mastering-python/exercises`. You are encouraged to submit your own solutions and learn about alternative solutions from others.

Summary

This chapter showed us how to write doctests, make use of the shortcuts provided by `py.test`, and use the `logging` module. With testing, there is never a one-size-fits-all solution. While the `doctest` system is very useful in many cases for providing both documentation and tests at the same time, in many functions there are edge cases that simply don't matter for documentation but still need to be tested. This is where regular unit tests come in and where `py.test` helps a lot.

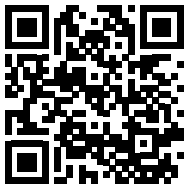
We have also seen how we can use `tox` to run tests in multiple sandboxed environments. If you ever have a project that also has to run on different computers or even on different Python versions, I would highly encourage you to use it.

The `logging` module is extremely useful when configured correctly and if your project becomes somewhat larger, it quickly becomes useful to do so. The usage of the logging system should be clear enough for most of the common use cases now, and as long as you keep the `propagate` parameter in check, you should be fine when implementing a logging system.

Next up is debugging, where testing helps prevent bugs. We will see how to solve them effectively. In addition, the logging that we added in this chapter will help a lot in that area.

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:
`https://discord.gg/QMzJenHuJf`



11

Debugging – Solving the Bugs

The previous chapter showed you how to add logging and tests to your code, but no matter how many tests you have, you will always have bugs. The biggest problem will always be external variables such as user input and different environments. At some point sooner or later, we will need to debug issues with our code, or worse, the code that was written by someone else.

There are many debugging techniques and, most certainly, you have already used a few of them. Within this chapter, we are going to focus on print/trace debugging and interactive debugging.

Debugging using print statements, stack traces, and logging is one of the most versatile methods to work with, and it is most likely the first type of debugging you ever used. Even a `print('Hello world')` can be considered this type, as the output will show you that your code is being executed correctly. There is obviously no point in explaining how and where to place print statements to debug your code, but there are quite a few nice tricks using decorators and other Python modules that render this type of debugging a lot more useful, such as `faulthandler`.

Interactive debugging is a more complicated debugging method. It allows you to debug a program while it's still running. Using this method, it's even possible to change variables while the application is running and pause the application at any point desired. The downside is that it requires some knowledge about the debugger commands to be really useful.

To summarize, we will cover the following topics:

- Non-interactive debugging using `print`, `trace`, `logging`, and `faulthandler`
- Interactive debugging using `pdb`, `ipython`, `jupyter`, and other debuggers and debugging services

Non-interactive debugging

The most basic form of debugging is adding a simple print statement into your code to see what is still working and what isn't. This is useful in a variety of cases and likely to help solve most of your issues.

Later in this chapter, we will show some interactive debugging methods, but those are not always suitable. Interactive debugging tends to become difficult or even impossible in cases such as:

- Multithreaded environments
- Multiple servers
- Bugs that are hard (or take a long time) to reproduce
- Closed-off remote servers such as Google App Engine or Heroku

Both interactive and non-interactive debugging methods have their merits, but I personally opt for non-interactive debugging 90% of the time, since a simple print/log statement is usually enough to analyze the cause of a problem. I find interactive debugging to be mostly helpful when writing code which uses large and complicated external libraries, where it can be hard to analyze which attributes, properties, and methods are available for objects.

A basic example of this (I've been known to do similar) with a generator can be as follows:

```
>>> def hiding_generator():
...     print('a')
...     yield 'first value'
...     print('b')
...     yield 'second value'
...     print('c')

>>> generator = hiding_generator()

>>> next(generator)
a
'first value'

>>> next(generator)
b
'second value'

>>> next(generator)
Traceback (most recent call last):
...
StopIteration
```

This shows exactly where the code does, and consequently, does not reach. Without this example, you might have expected the first `print` to come immediately after the `hiding_generator()` call. Since it's a generator, however, nothing will be executed until we `yield` an item. Assuming you would have some setup code before the first `yield`, it won't run until `next` is actually called. Additionally, `print('c')` is never executed and can be considered unreachable code.

Although this is one of the simplest ways to debug functions using `print` calls, it's not always the most convenient way. We can start by making an auto-print function that prints the line of code that it's going to execute:

```
>>> import os
>>> import inspect
>>> import linecache

>>> def print_code():
...     while True:
...         info = inspect.stack()[1]
...         lineno = info.lineno + 1
...         function = info.function
...         # Fetch the next line of code
...         code = linecache.getline(info.filename, lineno)
...         print(f'{lineno:03d} {function}: {code.strip()}')
...         yield

# Always prime the generator
>>> print_code = print_code()

>>> def some_test_function(a, b):
...     next(print_code)
...     c = a + b
...     next(print_code)
...     return c

>>> some_test_function('a', 'b')
003 some_test_function: c = a + b
005 some_test_function: return c
'ab'
```

As you can see, it automatically prints the line number, the name of the function, and the line of code it will execute next for you. That way, if you have a slow bit of code, you can see which line is stalling because it will be printed before execution.

With this specific instance, there's no real use for a generator, but you could easily incorporate some timings so you can see the delay between two `next(print_code)` statements. Or perhaps a counter to see how often this particular bit of code has been run.

Inspecting your script using trace

Simple print statements are useful in a lot of cases since you can easily incorporate print statements in nearly every application. It does not matter whether it's remote or local, threaded or using multiprocessing. It works almost everywhere, making it the most universal solution available – in addition to logging, that is. The general solution is often not the best solution for every situation, however. A nice alternative to our previous function is the `trace` module. It offers you a way to trace every executed line, including the runtime. The downside of tracing so much data is that it can quickly become overly verbose, as we will see in the next example.

To demonstrate, we will use our previous code but without print statements:

```
def some_test_function(a, b):
    c = a + b
    return c

print(some_test_function('a', 'b'))
```

Now we execute the code with the `trace` module:

```
$ python3 -m trace --trace --timing T_01_trace.py
 --- modulename: T_01_trace, funcname: <module>
0.00 T_01_trace.py(1): def some_test_function(a, b):
0.00 T_01_trace.py(6): print(some_test_function('a', 'b'))
 --- modulename: T_01_trace, funcname: some_test_function
0.00 T_01_trace.py(2):     c = a + b
0.00 T_01_trace.py(3):     return c
ab
```

The `trace` module shows you exactly which line is being executed with function names and, more importantly, which line was caused by which statement (or statements). Additionally, it shows you at what time it was executed relative to the start time of the program. This is due to the `--timing` flag.

And it still seems fairly reasonable in terms of output, right? Within this example, it does because this is about the most basic code there is. As soon as you add an `import`, for example, your screen will be flooded with output. In spite of the fact that you can opt to ignore specific modules and directories by using command-line parameters, it is still too verbose in many cases.

We can also enable the `trace` module selectively with a little bit of effort:

```
import sys
import trace as trace_module
import contextlib

@contextlib.contextmanager
def trace(count=False, trace=True, timing=True):
```

```
    tracer = trace_module.Trace(
        count=count, trace=trace, timing=timing)
    sys.settrace(tracer.globaltrace)
    yield tracer
    sys.settrace(None)

    result = tracer.results()
    result.write_results(show_missing=False, summary=True)

def some_test_function(a, b):
    c = a + b
    return c

with trace():
    print(some_test_function('a', 'b'))
```

This code shows a context manager that temporarily enables and disables the `trace` module to selectively trace code. In this example, we used `sys.settrace` with `tracer.globaltrace` as an argument, but you could also hook to your own tracing functions to customize the output.

When executing this, we get this output:

```
$ python3 T_02_selective_trace.py
 --- modulename: T_02_selective_trace, funcname: some_test_function
0.00 T_02_selective_trace.py(19):     c = a + b
0.00 T_02_selective_trace.py(20):     return c
ab
 --- modulename: contextlib, funcname: __exit__
0.00 contextlib.py(122):         if type is None:
0.00 contextlib.py(123):             try:
0.00 contextlib.py(124):                 next(self.gen)
 --- modulename: T_02_selective_trace, funcname: trace
0.00 T_02_selective_trace.py(12):     sys.settrace(None)
```

Now, to illustrate, if we were to run the same code with the trace module enabled, we would get a lot of output:

```
$ python3 -m trace --trace --timing T_02_selective_trace.py | wc
    256    2940   39984
```

The `wc` (word count) command shows us that this command gave us `252` lines, `2881` words, or `38716` characters of output, so I would generally recommend using the context decorator instead. Executing a trace on any reasonably sized script will generate a scary amount of output.

There are a few extra options with the `trace` module, such as showing which code is (not) executed, which can be useful to detect code coverage.

In addition to the arguments we already passed to `trace`, we can easily change the output or add extra filters by wrapping or replacing `tracer.globaltrace` as the `sys.settrace()` argument. As arguments, the function needs to accept `frame`, `event`, and `arg`.

The `frame` is a Python stack frame that contains references to the code and the filename and can be used to inspect the scope at that point in the stack. This is the same frame you can extract when using the `traceback` module.

The `event` argument is a string that can have the following values (from the standard Python documentation):

Parameter	Description
`call`	A function is called (or some other code block entered). The global trace function is called; `arg` is `None`. The return value specifies the local trace function.
`line`	The interpreter is about to execute a new line of code or re-execute the condition of a loop. The local trace function is called; `arg` is `None`; the return value specifies the new local trace function. See `Objects/lnotab_notes.txt` (in the Python source repository) for a detailed explanation of how this works. Per-line events may be disabled for a frame by setting `f_trace_lines` to `False` on that `frame`.
`return`	A function (or another code block) is about to return. The local trace function is called; `arg` is the value that will be returned, or `None` if the event is caused by an exception being raised. The trace function's return value is ignored.
`exception`	This means an exception has occurred. The local trace function is called; `arg` is a tuple (`exception`, `value`, `traceback`). The return value specifies the new local trace function.
`opcode`	The interpreter is about to execute a new opcode (see the `dis` module for opcode details). The local trace function is called; `arg` is `None`; the return value specifies the new local trace function. Per-opcode events are not emitted by default: they must be explicitly requested by setting `f_trace_opcodes` to `True` on the frame.

Lastly, the `arg` argument depends on the `event` argument as illustrated by the documentation above. In general, if `arg` is `None`, the return value of this function will be used as the local trace function, allowing you to override this for a specific scope. With `exception` events, it will be a tuple containing `exception`, `value`, and `traceback`.

Now let's create a little snippet that can selectively trace our code by filtering on filename:

```
import sys
import trace as trace_module
```

```
import contextlib

@contextlib.contextmanager
def trace(filename):
    tracer = trace_module.Trace()

    def custom_trace(frame, event, arg):
        # Only trace for the given filename
        if filename != frame.f_code.co_filename:
            return custom_trace

        # Let globaltrace handle the rest
        return tracer.globaltrace(frame, event, arg)

    sys.settrace(custom_trace)
    yield tracer
    sys.settrace(None)

    result = tracer.results()
    result.write_results(show_missing=False, summary=True)

def some_test_function(a, b):
    c = a + b
    return c

# Pass our current filename as '__file__'
with trace(filename=__file__):
    print(some_test_function('a', 'b'))
```

By using the `frame` argument, we can retrieve the code we are currently executing, and from that, the filename the code currently exists in. Naturally, you could also filter for different functions or only filter to a specific depth. Since we hand the tracing and outputting off to `tracer.globaltrace()`, we only check the `filename` for a place up in the stack. You could return `trace()` instead and handle the `print()` yourself.

When executing this code, you should get:

```
$ python3 T_03_filename_trace.py
 --- modulename: T_03_filename_trace, funcname: some_test_function
T_03_filename_trace.py(27):     c = a + b
T_03_filename_trace.py(28):     return c
ab
 --- modulename: T_03_filename_trace, funcname: trace
```

```
T_03_filename_trace.py(20):     sys.settrace(None)
lines   cov%   module   (path)
    3   100%   T_03_filename_trace   (T_03_filename_trace.py)
```

As you can see, this excludes the code from `contextlib`, which we saw in the earlier example.

Debugging using logging

In *Chapter 10*, *Testing and Logging – Preparing for Bugs*, we saw how to create custom loggers, set the levels for them, and add handlers to specific levels. We are going to use the `logging.DEBUG` level to log now, which is nothing special by itself, but with a few decorators, we can add some very useful debug-only code.

Whenever I'm debugging, I always find it very useful to know the input and output for a function. The basic version with a decorator is simple enough to write; just print the `args`, `kwargs`, and `return` value and you are done. The following example goes a little further. By using the `inspect` module, we can retrieve the default arguments as well, making it possible to show all arguments with the argument names and values in all cases, even if the argument was not specified:

```
import pprint
import inspect
import logging
import functools

def debug(function):
    @functools.wraps(function)
    def _debug(*args, **kwargs):
        # Make sure 'result' is always defined
        result = None
        try:
            result = function(*args, **kwargs)
            return result
        finally:
            # Extract the signature from the function
            signature = inspect.signature(function)
            # Fill the arguments
            arguments = signature.bind(*args, **kwargs)
            # NOTE: This only works for Python 3.5 and up!
            arguments.apply_defaults()
            logging.debug('%s(%s): %s' % (
                function.__qualname__,
                ', '.join('%s=%r' % (k, v) for k, v in
                          arguments.arguments.items()),
                pprint.pformat(result),
```

```
            ))

    return _debug

@debug
def add(a, b=123):
    return a + b

if __name__ == '__main__':
    logging.basicConfig(level=logging.DEBUG)

    add(1)
    add(1, 456)
    add(b=1, a=456)
```

Let's analyze how this code executes:

1. The decorator executes `function()` as normal with the given `*args` and `**kwargs` passed along unmodified, while storing the result to both display and `return` later.
2. The `finally` section of the `try`/`finally` generates an `inspect.Signature()` object from `function()`.
3. Now we generate an `inspect.BoundArguments()` object by binding `*args` and `**kwargs` using the previously generated `signature`.
4. Now we can tell the `inspect.BoundArguments()` object to apply the default arguments so we can see the value of arguments not passed in `*args` and `**kwargs`.
5. Lastly, we output the full function name, the formatted arguments, and the `result`.

When we execute the code, we should see the following:

```
$ python3 T_04_logging.py
DEBUG:root:add(a=1, b=123): 124
DEBUG:root:add(a=1, b=456): 457
DEBUG:root:add(a=456, b=1): 457
```

Very nice of course, as we have a clear sight of when the function is called, which parameters were used, and what is returned. However, this is something you will probably only execute when you are actively debugging your code.

You can also make the regular `logging.debug` statements in your code quite a bit more useful by adding a debug-specific logger, which shows more information. Simply replace the logging config of the preceding example with this:

```
import logging

log_format = (
```

```
    '[%(relativeCreated)d %(levelname)s] '
    '%(filename)s:%(lineno)d:%(funcName)s: %(message)s'
)
logging.basicConfig(level=logging.DEBUG, format=log_format)
```

Then your result will be something like this:

```
$ python3 T_05_logging_config.py
[DEBUG] T_05_logging_config.py:20:_debug: add(a=1, b=123): 124
[DEBUG] T_05_logging_config.py:20:_debug: add(a=1, b=456): 457
[DEBUG] T_05_logging_config.py:20:_debug: add(a=456, b=1): 457
```

It shows the time relative to the start of the application in milliseconds and the log level. This is followed by an identification block that shows the filename, line number, and function name that originated the logs. Of course, there is a `message` at the end, which contains the result of our log call.

Showing the call stack without exceptions

When looking at how and why a piece of code is being run, it's often useful to see the entire stack trace. Simply raising an exception is, of course, an option. However, that will kill the current code execution, which is generally not something we are looking for. This is where the `traceback` module comes in handy. With just a simple call to `traceback.print_stack()`, we get a full stack list:

```
import sys
import traceback

class ShowMyStack:
    def run(self, limit=None):
        print('Before stack print')
        traceback.print_stack(limit=limit)
        print('After stack print')

class InheritShowMyStack(ShowMyStack):
    pass

if __name__ == '__main__':
    show_stack = InheritShowMyStack()

    print('Stack without limit')
    show_stack.run()
    print()

    print('Stack with limit 1')
    show_stack.run(1)
```

The `ShowMyStack.run()` function shows a regular `traceback.print_stack()` call, which shows the entire stack trace to that point in the stack. You could place `traceback.print_stack()` anywhere in your code to see where it is being called from.

Since the full stack trace can be quite large, it is often useful to use the `limit` argument to only show a few levels, which is what we do in the second run.

This results in the following:

```
$ python3 T_06_stack.py
Stack without limit
Before stack print
  File "T_06_stack.py", line 20, in <module>
    show_stack.run()
  File "T_06_stack.py", line 8, in run
    traceback.print_stack(limit=limit)
After stack print

Stack with limit 1
Before stack print
  File "T_06_stack.py", line 8, in run
    traceback.print_stack(limit=limit)
After stack print
```

As you can see, the traceback simply prints without any exceptions. The `traceback` module actually has quite a few other methods for printing tracebacks based on exceptions and such, but you probably won't need them often. The most useful one is probably the `limit` parameter we've demonstrated. A positive limit number shows you only a specific number of frames. In most cases, you don't need a full stack trace, so this can be quite useful to limit the output.

Alternatively, we can also specify a negative limit, which trims the stack from the other side. This is mostly useful when printing the stack from a decorator where you want to hide the decorator from the trace. If you want to limit both sides, you will have to do it manually using `format_list(stack)` with a stack from `extract_stack(f, limit)`, the usage of which is similar to the `print_stack()` function.

The negative limit support was added in Python 3.5. Before that, only positive limits were supported.

Handling crashes using faulthandler

The `faulthandler` module helps when debugging really low-level crashes, that is, crashes that should only be possible when using low-level access to memory, such as C extensions.

For example, here's a bit of code that *will* cause your Python interpreter to crash:

```
import ctypes

# Get memory address 0, your kernel shouldn't allow this:
ctypes.string_at(0)
```

It results in something similar to the following:

```
$ python3 T_07_faulthandler.py
zsh: segmentation fault  python3 T_07_faulthandler.py
```

That's quite an ugly response, of course, and gives you no possibility to handle the error. Just in case you are wondering, having a `try/except` structure won't help you in these cases either. The following code will crash in exactly the same way:

```
import ctypes

try:
    # Get memory address 0, your kernel shouldn't allow this:
    ctypes.string_at(0)
except Exception as e:
    print('Got exception:', e)
```

This is where the `faulthandler` module helps. It will still cause your interpreter to crash, but at least you will see a proper error message raised, so it's a good default if you (or any of the sub-libraries) have any interaction with raw memory:

```
import ctypes
import faulthandler

faulthandler.enable()

# Get memory address 0, your kernel shouldn't allow this:
ctypes.string_at(0)
```

It results in something along these lines:

```
$ python3 T_09_faulthandler_enabled.py
Fatal Python error: Segmentation fault

Current thread 0x0000000110382e00 (most recent call first):
  File python3.9/ctypes/__init__.py", line 517 in string_at
  File T_09_faulthandler.py", line 7 in <module>
zsh: segmentation fault  python3 T_09_faulthandler_enabled.py
```

Obviously, it's not desirable to have a Python application exit in this manner as the code won't exit with a normal cleanup. Resources won't be closed cleanly and your exit handler won't be called. If you somehow need to catch this behavior, your best bet is to wrap the Python executable in a separate script using something like `subprocess.run([sys.argv[0], 'T_09_faulthandler_enabled.py'])`.

Interactive debugging

Now that we have discussed basic debugging methods that will always work, we will look at interactive debugging for some more advanced debugging techniques. The previous debugging methods made variables and stacks visible through modifying the code and/or foresight. This time around, we will look at a slightly smarter method, which constitutes doing the same thing interactively, but once the need arises.

Console on demand

When testing some Python code, you may have used the interactive console a couple of times, since it's a simple yet effective tool for testing your Python code. What you might not have known is that it is actually simple to start your own shell from within your code. So, whenever you want to drop into a regular shell from a specific point in your code, that's easily possible:

```
import code

def start_console():
    some_variable = 123
    print(f'Launching console, some_variable: {some_variable}')
    code.interact(banner='console:', local=locals())
    print(f'Exited console, some_variable: {some_variable}')

if __name__ == '__main__':
    start_console()
```

When executing that, we will drop into an interactive console halfway:

```
$ python3 T_10_console.py
Launching console, some_variable: 123
console:
>>> some_variable = 456
>>>
now exiting InteractiveConsole...
Exited console, some_variable: 123
```

To exit this console, we can use ^*d* (*Ctrl* + *D*) on Linux/Mac systems and ^*z* (*Ctrl* + *Z*) on Windows systems.

One important thing to note here is that the local scope is not shared between the two. Even though we passed along `locals()` to share the local variables for convenience, this relation is not bidirectional.

The result is that even though we set `some_variable` to `456` in the interactive session, it does not carry over to the outside function. You can modify variables in the outside scope through direct manipulation (for example, setting the properties) if you wish, but all variables declared locally will remain local.

Naturally, modifying mutable variables will affect both scopes.

Debugging using Python debugger (pdb)

When it comes to actually debugging code, the regular interactive console just isn't suited. With a bit of effort, you can make it work, but it's just not all that convenient for debugging since you can only see the current scope and can't jump around the stack easily. With `pdb` (Python debugger), this is easily possible. So, let's look at a simple example of using `pdb`:

```
import pdb

def go_to_debugger():
    some_variable = 123
    print('Starting pdb trace')
    pdb.set_trace()
    print(f'Finished pdb, some_variable: {some_variable}')

if __name__ == '__main__':
    go_to_debugger()
```

This example is pretty much identical to the one in the previous paragraph, except that this time we end up in the `pdb` console instead of a regular interactive console. So let's give the interactive debugger a try:

```
$ python3 T_11_pdb.py
Starting pdb trace
> T_11_pdb.py(8)go_to_debugger()
-> print(f'Finished pdb, some_variable: {some_variable}')
(Pdb) some_variable
123
(Pdb) some_variable = 456
(Pdb) continue
Finished pdb, some_variable: 456
```

As you can see, we've actually modified the value of `some_variable` now. In this case, we used the full `continue` command, but all the `pdb` commands have short versions as well. So, using `c` instead of `continue` gives the same result. Just typing `some_variable` (or any other variable) will show the contents and setting the variable will simply set it, just as we would expect from an interactive session.

To get started with `pdb`, first of all, a list of the most useful (full) stack movement and manipulation commands with shorthands is shown here:

Command	Explanation
`h(elp)`	This shows the list of commands (this list).
`h(elp) command`	This shows the help for the given command.
`w(here)`	Current stack trace with an arrow at the current frame.
`d(own)`	Move down/to a newer frame in the stack.
`u(p)`	Move up/to an older frame in the stack.
`s(tep)`	Execute the current line and stop as soon as possible.
`n(ext)`	Execute the current line and stop at the next line within the current function.
`r(eturn)`	Continue execution until the function returns.
`c(ont(inue))`	Continue execution up to the next breakpoint.
`l(ist) [first[, last]]`	List the lines of source code (by default, 11 lines) around the current line.
`ll | longlist`	List all of the source code for the current function or frame.
`source expression`	List the source code for the given object. This is similar to `longlist`.
`a(rgs)`	Print the arguments for the current function.
`pp expression`	Pretty-print the given expression.
`! statement`	Execute the statement at the current point in the stack. Normally, the `!` sign is not needed, but this can be useful if there are collisions with debugger commands. For example, try `b = 123`.
`interact`	Open an interactive Python shell session similar to the previous paragraph.

Many more commands are available and some of them will be covered by the following paragraphs. All commands are covered by the built-in help, however, so be sure to use the `h/help [command]` command if needed.

Breakpoints

Breakpoints are points where the debugger will halt the code execution and allow you to debug from that point. We can create breakpoints using either code or commands. First, let's enter the debugger using `pdb.set_trace()`. This is effectively a hardcoded breakpoint:

```
import pdb

def print_value(value):
    print('value:', value)

if __name__ == '__main__':
```

```
    pdb.set_trace()
    for i in range(5):
        print_value(i)
```

So far, nothing new has happened, but let's now open the interactive debugging session and try a few breakpoint commands. Here's a list of the most useful breakpoint commands before we start:

Command	Explanation
`b(reak)`	Show the list of breakpoints.
`b(reak) [filename:] lineno`	Place a breakpoint at the given line number and, optionally, file.
`b(reak) function[, condition]`	Place a breakpoint at the given function. The condition is an expression that must evaluate to `True` for the breakpoint to work.
`cl(ear) [filename:] lineno`	Clear the breakpoint (or breakpoints) at this line.
`cl(ear) breakpoint [breakpoint ...]`	Clear the breakpoint (or breakpoints) with these numbers.

Now let's execute this code and enter the interactive debugger to try the commands:

```
$ python3 T_12_pdb_loop.py
> T_12_pdb_loop.py (10)<module>()
-> for i in range(5):
(Pdb) source print_value  # View the source of print_value
  4     def print_value(value):
  5         print('value:', value)
(Pdb) b 5  # Add a breakpoint to line 5
Breakpoint 1 at T_12_pdb_loop.py:5
(Pdb) w  # Where shows the current line
> T_12_pdb_loop.py (10)<module>()
-> for i in range(5):
(Pdb) c  # Continue (until the next breakpoint or exception)
> T_12_pdb_loop.py(5)print_value()
-> print('value:', value)
(Pdb) w  # Where shows the current line and the calling functions
  T_12_pdb_loop.py(11)<module>()
-> print_value(i)
> T_12_pdb_loop.py(5)print_value()
-> print('value:', value)
(Pdb) ll  # List the lines of the current function
  4     def print_value(value):
```

```
  5 B->     print('value:', value)
(Pdb) b  # Show the breakpoints
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at T_12_pdb_loop.py:5
        breakpoint already hit 1 time
(Pdb) cl 1  # Clear breakpoint 1
Deleted breakpoint 1 at T_12_pdb_loop.py:5
(Pdb) c  # Continue the application until the end
value: 0
value: 1
value: 2
value: 3
value: 4
```

That was a lot of output, but it's actually not as complex as it seems:

1. First, we used the `source print_value` command to see the source for the `print_value` function.
2. After that, we knew the line number of the first `print` statement, which we used to place a breakpoint (`b 5`) at line 5.
3. To check whether we were still at the right position, we used the `w` command.
4. Since the breakpoint was set, we used `c` to continue up to the next breakpoint.
5. Having stopped at the breakpoint at line 5, we used `w` again to confirm that and show the current stack.
6. We listed the code of the current function using `ll`.
7. We listed the breakpoints using `b`.
8. We removed the breakpoint again using `cl 1` with the breakpoint number from the previous command.
9. We continued (`c`) until the program exits or reaches the next breakpoint if available.

It all seems a bit complicated in the beginning, but you'll see that it's actually a very convenient way of debugging once you've tried a few times.

To make it even better, this time we will execute the breakpoint only when `value = 3`:

```
$ python3 T_12_pdb_loop.py
> T_12_pdb_loop.py(10)<module>()
-> for i in range(5):
# print the source to find the variable name and line number:
(Pdb) source print_value
  4     def print_value(value):
  5         print('value:', value)
(Pdb) b 5, value == 3  # add a breakpoint at line 5 when value=3
```

```
Breakpoint 1 at T_12_pdb_loop.py:5
(Pdb) c  # continue until breakpoint
value: 0
value: 1
value: 2
> T_12_pdb_loop.py(5)print_value()
-> print('value:', value)
(Pdb) a  # show the arguments for the function
value = 3
(Pdb) value = 123  # change the value before the print
(Pdb) c  # continue, we see the new value now
value: 123
value: 4
```

To list what we have done:

1. First, using `source print_value`, we looked for the line number and variable name.
2. After that, we placed a breakpoint with the `value == 3` condition.
3. Then we continued execution using `c`. As you can see, the values `0`, `1`, and `2` are printed as normal.
4. The breakpoint was reached at value `3`. To verify, we used `a` to see the function arguments.
5. We changed the variable before `print()` was executed.
6. We continued to execute the rest of the code.

Catching exceptions

All of these have been manual calls to the `pdb.set_trace()` function, but in general, you are just running your application and not really expecting issues. This is where exception catching can be very handy. In addition to importing `pdb` yourself, you can run scripts through `pdb` as a module as well. Let's examine this bit of code, which dies as soon as it reaches zero division:

```
print('This still works')
1 / 0
print('We will never reach this')
```

If we run it through the `pdb` module, we can end up in the Python debugger whenever it crashes:

```
$ python3 -m pdb T_13_pdb_catching_exceptions
> T_13_pdb_catching_exceptions(1)<module>()
-> print('This still works')
(Pdb) w  # Where
  bdb.py(431)run()
-> exec(cmd, globals, locals)
```

```
  <string>(1)<module>()
> T_13_pdb_catching_exceptions(1)<module>()
-> print('This still works')
(Pdb) s  # Step into the next statement
This still works
> T_13_pdb_catching_exceptions(2)<module>()
-> 1/0
(Pdb) c  # Continue
Traceback (most recent call last):
  File "pdb.py", line 1661, in main
    pdb._runscript(mainpyfile)
  File "pdb.py", line 1542, in _runscript
    self.run(statement)
  File "bdb.py", line 431, in run
    exec(cmd, globals, locals)
  File "<string>", line 1, in <module>
  File "T_13_pdb_catching_exceptions", line 2, in <module>
    1/0
ZeroDivisionError: division by zero
Uncaught exception. Entering post mortem debugging
Running 'cont' or 'step' will restart the program
> T_13_pdb_catching_exceptions(2)<module>()
-> 1/0
```

A useful little trick within `pdb` is to use the *Enter* button, which, by default, will execute the previously executed command again. This is very useful when stepping through the program.

Aliases

Aliases can be a really useful feature to make your life easier. If you "live" in a Linux/Unix shell like I do, you are probably already familiar with them, but essentially an alias is just a shorthand to save you from having to type (or even remember) a long and complicated command.

Which aliases are useful for you depends on your preferences of course, but I personally like an alias for the `pprint` (pretty print) module. Within my projects, I often use `pf=pprint.pformat` and `pp=pprint.pprint` as aliases, but the same goes for `pdb` where I find `pd` a useful shorthand for pretty printing the `__dict__` for a given object.

The pdb commands for aliases are relatively straightforward and very easy to use:

`alias`	List all aliases.
`alias name command`	Create an alias. The command can be any valid Python expression, so you can do the following to print all properties for an object: `alias pd pp %1.__dict__`
`unalias name`	Remove an alias.

Make sure to use these to your advantage. Within Linux/Unix systems, you have probably noticed that many commands (`ls`, `rm`, `cd`) are very short to save you some typing; you can do the same with these aliases.

commands

The `commands` command is a little complicated but very useful. It allows you to execute commands whenever a specific breakpoint is encountered. To illustrate this, let's start from a simple example again:

```
def do_nothing(i):
    pass

for i in range(10):
    do_nothing(i)
```

The code is simple enough, so now we'll add the breakpoint and the commands, as follows:

```
$ python3 -m pdb T_14_pdb_commands.py
> T_14_pdb_commands.py(1)<module>()
-> def do_nothing(i):
(Pdb) b do_nothing  # Add a breakpoint to function do_nothing
Breakpoint 1 at T_14_pdb_commands.py:1
(Pdb) commands 1  # add command to breakpoint 1
(com) print(f'The passed value: {i}')
(com) end  # end command
(Pdb) c  # continue
The passed value: 0
> 16_pdb_commands.py(2)do_nothing()
-> pass
(Pdb) q  # quit
```

As you can see, we can easily add commands to the breakpoint. After removing the breakpoint, these commands won't be executed anymore because they are linked to the breakpoint.

These can be really useful to add some automatic debug `print` statements to your breakpoint; for example, to see the value of all of the variables in the local scope. You can always manually do a `print(locals())` of course, but these can save you a lot of time while debugging.

Debugging with IPython

While the generic Python console is useful, it can be a little rough around the edges. The IPython console offers a whole new world of extra features, which make it a much nicer console to work with. One of those features is a more convenient debugger.

First, make sure you have `ipython` installed:

```
$ pip3 install ipython
```

Next, let's try the debugger with a very basic script:

```
def print_value(value):
    print('value:', value)

if __name__ == '__main__':
    for i in range(5):
        print_value(i)
```

Next, we run IPython and tell it to run the script in debug mode:

```
$ ipython
Python 3.10.0
Type 'copyright', 'credits' or 'license' for more information
IPython 7.19.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: %run -d T_15_ipython.py
Breakpoint 1 at T_15_ipython.py:1
NOTE: Enter 'c' at the ipdb>  prompt to continue execution.
> T_15_ipython.py(1)<module>()
1---> 1 def print_value(value):
      2     print('value:', value)
      3
      4
      5 if __name__ == '__main__':

ipdb> b print_value, value == 3  # Add a breakpoint when value=3
Breakpoint 2 at T_15_ipython.py:1
ipdb> c
value: 0
value: 1
value: 2
> T_15_ipython.py(2)print_value()
2     1 def print_value(value):
----> 2     print('value:', value)
```

```
      3
      4
      5 if __name__ == '__main__':

ipdb> value
3
ipdb> value = 123  # Change the value
ipdb> c  # Continue
value: 123
value: 4
```

As you can see, not all that different from `pdb`. But it automatically shows the surrounding code in a readable format, which is very useful. Additionally, the shown code has syntax highlighting, which helps with readability as well.

If you install the `ipdb` module, you get features similar to the `pdb` module, which allow for triggering breakpoints from your code.

Debugging with Jupyter

Jupyter is amazing for ad hoc development and makes it really easy to see what's going on in your code for small scripts. For larger scripts, it can quickly become more difficult because you normally only get the non-interactive stack trace and have to resort to a different method for changing external code.

Since 2020, however, Jupyter has added a (currently experimental) visual debugger to make it possible to debug your code as it happens in a very convenient way. To get started, make sure you have a recent version of Jupyter and install both the `@jupyterlab/debugger` extension and the `xeus-python` (XPython) kernel for Jupyter. To make sure everything works without too much effort, I strongly recommend using `conda` for this operation:

```
$ conda create -n jupyter-debugger -c conda-forge xeus-python=0.8.6 notebook=6
jupyterlab=2 ptvsd nodejs
...
## Package Plan ##

  added / updated specs:
    - jupyterlab=2
    - nodejs
    - notebook=6
```

```
    - ptvsd
    - xeus-python=0.8.6
...
$ conda activate jupyter-debugger

(jupyter-debugger) $ jupyter labextension install @jupyterlab/debugger

Building jupyterlab assets (build:prod:minimize)
```

The current installation instructions for Conda can be found on the JupyterLab debugger GitHub page: `https://jupyterlab.readthedocs.io/en/latest/user/debugger.html`

For a regular Python virtual environment, you can try the binary wheel (`.whl`) packages so you don't have to compile anything. Due to the currently experimental nature of this feature, it is not supported in all environments yet. At the time of writing, binary wheels are available for Python 3.6, 3.7 and 3.8 for OS X, Linux, and Windows. A list of available versions can be found here: `https://pypi.org/project/xeus-python/#files`

Now we can start `jupyter lab` as normal:

```
(jupyter-debugger) $ jupyter lab
[I LabApp] JupyterLab extension loaded from jupyterlab
[I LabApp] Jupyter Notebook 6.1.4 is running at:
[I LabApp] http://localhost:8888/?token=...
[I LabApp] Use Control-C to stop this server and shut down all kernels (twice
to skip confirmation).
```

If everything is working as expected, you should see the JupyterLab launcher now, with both Python 3 and the XPython kernels available:

Figure 11.1: JupyterLab Python and XPython kernels

Since only `xeus-python` (XPython) currently supports debugging, we will have to open that one. Now we will add our script from before so we can demonstrate the debugger. If everything is working correctly, you should see the debug buttons at the top-right part of your screen:

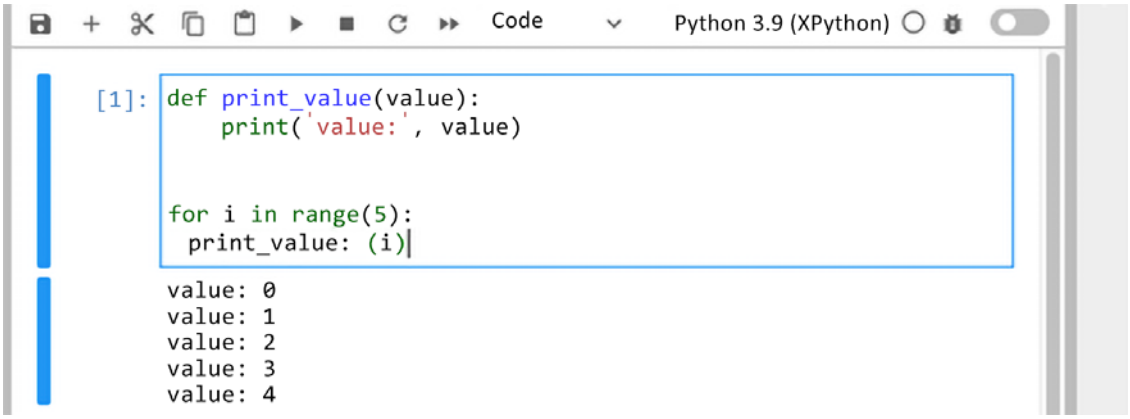


Figure 11.2: Regular Jupyter console output

Now we can start debugging by following these steps:

1. Enable the debug toggle at the top right.
2. Click on a line to add a breakpoint.
3. Run the code.

If everything is set up correctly, it should look something like this:

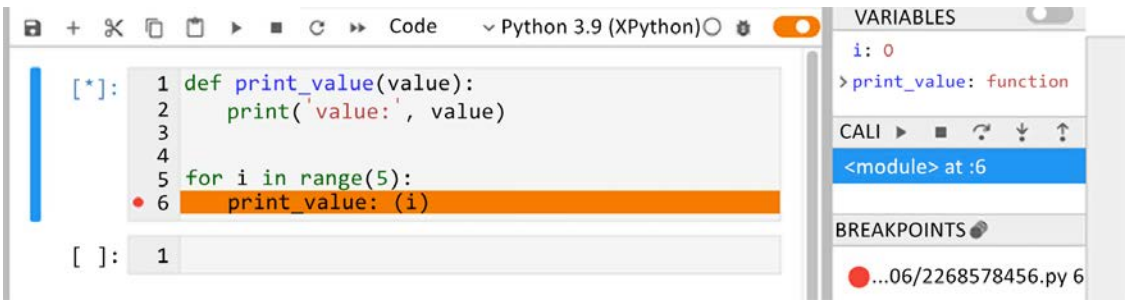


Figure 11.3: Debugging using Jupyter

From this point on, you can use the buttons in the debugging pane on the right to step over/in/out of the next statement to walk through the code.

Other debuggers

The `pdb` debugger is simply the Python default, but far from the only option to debug Python code. Some of the currently noteworthy debuggers are as follows:

- `ipdb`: The `pdb` debugger wrapped in an IPython shell
- `pudb`: A full-screen command-line debugger

- `pdbpp` (pdb++): An extension to the regular `pdb` module, which adds tab completion, syntax highlighting, and a few other useful features to `pdb`
- `Werkzeug`: A web-based debugger that allows debugging of web applications while they are running

There are many others, of course, and there isn't a single one that's the absolute best. As is the case with all tools, they all have their advantages and their flaws, and the one that is best for your current purpose can be properly decided only by you. Chances are that your current Python IDE already has an integrated debugger. The PyCharm IDE, for example, even offers built-in remote debugging so you can debug applications running on cloud providers from your local graphical interface.

Debugging services

In addition to debugging when you encounter a problem, there are times when you simply need to keep track of errors for later debugging. This can be especially difficult if your application is running on remote servers or on computers not controlled by you. For this type of error tracking, there are a few very useful open-source packages available.

Elastic APM

Elastic APM is part of the Elastic Stack and can keep track of errors, performance, logs, and other data for you. This system can help you track not only Python applications but supports a whole range of other languages and applications as well. The Elastic Stack (which is built around Elasticsearch) is an extremely versatile and very well-maintained stack of software which I highly recommend.

The only downside of the Elastic Stack is that it is a very heavy set of applications, which quickly requires a number of dedicated servers to maintain reasonable performance. It does scale very well, however; if you ever need more processing power, you can simply add a new machine to your cluster and everything will automatically rebalance for you.

Sentry

Sentry is an open-source error management system that allows you to collect errors from a wide range of languages and frameworks. Some notable features are:

- Grouping of errors so you only get one (or a configurable number of) notification of errors per type of error
- Being able to mark an error as "fixed" so it re-alerts you when it occurs again while still showing you the previous occurrences
- Showing a full stack trace including surrounding code
- Keeping track of code versions/releases so you know which version (re-)introduced an error
- Assign errors to a specific **developer** to fix

While the Sentry application is mainly focused on web applications, it can easily be used for regular applications and scripts as well.

Historically, Sentry started as a small error-grouping application that could be used as an app within an existing Django application, or as a separate installation depending on your needs. Since that time, very little of that lightweight structure remains; it has grown into a fully fledged error tracking system that has native support for many programming languages and frameworks.

Over time, Sentry has gravitated more and more toward the commercial hosted platform, however, so hosting the application yourself has become more difficult with that. The time that a simple `pip install sentry` was enough to get it running is long gone. These days, Sentry is a heavy application that relies on the following running services:

- PostgreSQL
- Redis
- Memcached
- Symbolicator
- Kafka
- Snuba

So if you wish to try Sentry, I would recommend trying the free tier of the hosted Sentry to see if you like it first. Manually installing is not really a valid option anymore, so if you wish to run it self-hosted, your only realistic option is to use the `docker-compose` files provided.

When self-hosting, you should keep in mind that it is a heavy application that requires a significant amount of resources to run and can easily fill a decently sized dedicated server. It is still lighter than Elastic APM, however.

In my experience, you need at least about 2-3 GiB of RAM and about 2 CPU cores to run current versions of Sentry. Depending on your load, you might need something much heavier, but that is the bare minimum.

Exercises

For local development, a few small utility functions can make your life much easier. We have already seen an example of this with the `print_code` generator and the `trace` context wrapper. See if you can extend one of these to:

- Execute code with a timeout so you can see where your application is stalling
- Measure the duration of the execution
- Show how often that specific bit of code has been executed

Example answers for these exercises can be found on GitHub: `https://github.com/mastering-python/exercises`. You are encouraged to submit your own solutions and learn about alternative solutions from others.

Summary

This chapter explained a few different debugging techniques and gotchas. There is, of course, much more that can be said about debugging, but I hope you have acquired a nice vantage point for debugging your Python code now. Interactive debugging techniques are very useful for single-threaded applications and locations where interactive sessions are available.

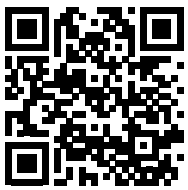
But since that's not always the case, we also discussed some non-interactive options.

To recap, in this chapter, we talked about non-interactive debugging with `print` statements, `logging`, `trace`, `traceback`, `asyncio`, and `faulthandler`. We also explored interactive debugging with the Python debugger, IPython, and Jupyter, as well as learning about alternative debuggers.

In the next chapter, we will see how to monitor and improve both CPU and memory performance, as well as finding and fixing memory leaks.

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:
`https://discord.gg/QMzJenHuJf`



12

Performance – Tracking and Reducing Your Memory and CPU Usage

Before we talk about performance, there is a quote by *Donald Knuth* you need to consider first:

> *"The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; premature optimization is the root of all evil (or at least most of it) in programming".*

Donald Knuth is often called the father of algorithm analysis. His book series, *The Art of Computer Programming*, can be considered the Bible of all fundamental algorithms.

As long as you pick the correct data structures with the right algorithms, performance should not be something to worry about. That does not mean you should ignore performance entirely, but just make sure you pick the right battles and optimize only when it is actually needed. Micro/premature optimizations can definitely be fun, but are only very rarely useful.

We have seen the performance characteristics of many data structures in *Chapter 2*, *Pythonic Syntax and Common Pitfalls*, already, so we won't discuss that, but we will show you how performance can be measured and how problems can be detected. There are cases where micro optimizations make a difference, but you won't know until you measure the performance.

Within this chapter, we will cover:

- Profiling CPU usage
- Profiling memory usage

- Learning how to correctly compare performance metrics
- Optimizing performance
- Finding and fixing memory leaks

Globally, the chapter is split between CPU usage and/or CPU time, and memory usage. The first half of the chapter mainly concerns CPU/time; the second half covers memory usage.

What is performance?

Performance is a very broad term. It has many different meanings and, in many cases, it is defined incorrectly. Within this chapter, we will attempt to measure and improve performance in terms of CPU usage/time and memory usage. Many of the examples here are a trade-off between execution time and memory usage. Note that a fast algorithm that can only use a single CPU core can be outperformed in terms of execution time by a slower algorithm that is easily parallelizable given enough CPU cores.

When it comes to incorrect statements about performance, you have probably heard statements similar to "Language X is faster than Python." That statement is inherently wrong. Python is neither fast nor slow; Python is a programming language, and a language has no performance metrics whatsoever. If you were to say that the CPython interpreter is faster or slower than interpreter Y for language X, that would be possible. The performance characteristics of code can vary greatly between different interpreters. Just take a look at this small test (which uses ZSH shell script):

```
$ export SCRIPT='"".join(str(i) for i in range(10000))'

$ for p in pypy3 pyston python3.{8..10}; do echo -n "$p: "; $p -m timeit
"$SCRIPT"; done
pypy3: ... 2000 loops, average of 7: 179 +- 6.05 usec per loop ...
pyston: 500 loops, best of 5: 817 usec per loop
python3.8: 200 loops, best of 5: 1.21 msec per loop
python3.9: 200 loops, best of 5: 1.64 msec per loop
python3.10: 200 loops, best of 5: 1.14 msec per loop
```

Five different Python interpreters, each with a different performance! All are Python, but the interpreters obviously vary.

You might not have heard of the PyPy3 and Pyston interpreters yet.

The PyPy3 interpreter is an alternative Python interpreter that uses JIT (Just-In-Time) compiling to perform much better than CPython in many, but certainly not all, cases. The big caveat of PyPy3 is that code that has speedups in C and depends on CPython extensions (which is a large portion of performance-critical libraries) either does not support PyPy3 or suffers a performance hit.

Pyston attempts to be a drop-in replacement for CPython with JIT compiling added to it. While JIT compiling might be added to CPython pretty soon, as of Python 3.10, that is not the case yet. This is why Pyston can offer a great performance benefit over CPython. The downside is that it is currently only supported on Unix/Linux systems.

Looking at this benchmark, you might be tempted to drop the CPython interpreter completely and only use PyPy3. The danger with benchmarks such as these is that they rarely offer any meaningful results. For this limited example, the Pypy interpreter was about 200 times faster than the CPython3.10 interpreter, but that has very little relevance for the general case. The only conclusion that can safely be drawn here is that this specific version of the PyPy3 interpreter is much faster than this specific version of CPython3 **for this exact test**. For any other test and interpreter version, the results could be vastly different.

Measuring CPU performance and execution time

When talking about performance you can measure a great number of things. When it comes to CPU performance, we can measure:

- The "wall time" (the absolute time on the clock).
- Relative time (when comparing multiple runs or multiple functions)
- Used CPU time. Due to multithreading, multiprocessing, or asynchronous processing, this can be vastly different from the wall time.
- When inspecting really low-level performance, measuring the number of CPU cycles and loop counts.

In addition to all these different measurement options, you should also consider the observer effect. Simply put, measuring takes time, and depending on how you are measuring the performance, the impact can be huge.

Within this section, we will be exploring several methods to inspect the CPU performance and execution time of your code. Tricks to improve your performance after measuring will come later in the chapter.

Timeit — comparing code snippet performance

Before we can start improving execution/CPU times, we need a reliable method to measure them. Python has a really nice module (`timeit`) with the specific purpose of measuring the execution times of bits of code. It executes a bit of code many times to make sure there is as little variation as possible and to make the measurement fairly clean. It's very useful if you want to compare a few code snippets. Following are some example executions:

```
$ python3 -m timeit 'x=[]; [x.insert(0, i) for i in range(10000)]'
10 loops, best of 3: 30.2 msec per loop
$ python3 -m timeit 'x=[]; [x.append(i) for i in range(10000)]'
1000 loops, best of 3: 1.01 msec per loop
$ python3 -m timeit 'x=[i for i in range(10000)]'
1000 loops, best of 3: 381 usec per loop
$ python3 -m timeit 'x=list(range(10000))'
10000 loops, best of 3: 212 usec per loop
```

These few examples demonstrate the performance difference between `list.insert`, `list.append`, a list comprehension, and the `list` function. As we have seen in *Chapter 4*, doing `list.insert` is very inefficient and that quickly shows here, in this case being 30 times slower than `list.append`.

More importantly, however, the code demonstrates how we can use the `timeit` module and how it works. As you can see in the output, the `list.append` variant was executed only `10` times, whereas the `list` call was executed `10000` times. That is one of the most convenient features of the `timeit` module: it automatically figures out some useful parameters for you, and it shows the "best of 3" to try and reduce the amount of variance in your tests.

The `timeit` module is great at comparing the performance of similar bits of code within a code base. Comparing the execution time between different Python interpreters using `timeit` is generally useless because it is rarely representative of the performance of your whole application.

Naturally, the command can be used with regular scripts as well, but that won't automatically determine the number of repetitions like the command-line interface does. So we will have to do that ourselves:

```
import timeit

def test_list():
    return list(range(10000))

def test_list_comprehension():
    return [i for i in range(10000)]

def test_append():
    x = []
    for i in range(10000):
        x.append(i)

    return x

def test_insert():
    x = []
    for i in range(10000):
        x.insert(0, i)

    return x

def benchmark(function, number=100, repeat=10):
    # Measure the execution times. Passing the globals() is an
```

```
    # easy way to make the functions available.
    times = timeit.repeat(function, number=number,
                          globals=globals())
    # The repeat function gives 'repeat' results so we take the
    # min() and divide it by the number of runs
    time = min(times) / number
    print(f'{number} loops, best of {repeat}: {time:9.6f}s :: ',
          function.__name__)

if __name__ == '__main__':
    benchmark(test_list)
    benchmark(test_list_comprehension)
    benchmark(test_append)
    benchmark(test_insert)
```

When executing this, you will get something along the following lines:

```
$ python3 T_00_timeit.py
100 loops, best of 10:  0.000168s ::  test_list
100 loops, best of 10:  0.000322s ::  test_list_comprehension
100 loops, best of 10:  0.000573s ::  test_append
100 loops, best of 10:  0.027552s ::  test_insert
```

As you may have noticed, this script is still a bit basic. While the command-line version of `timeit` keeps trying until it reaches 0.2 seconds or more, this script just has a fixed number of executions. Since Python 3.6, we do have the option of using `timeit.Timer.autorange` to replicate this behavior, but it is a bit less convenient to use and would produce a lot more output in our current case. Depending on your use case, however, it could be useful to try this benchmark code instead:

```
def autorange_benchmark(function):
    def print_result(number, time_taken):
        # The autorange function keeps trying until the total
        # runtime (time_taken) reaches 0.2 seconds. To get the
        # time per run we need to divide it by the number of runs
        time = time_taken / number
        name = function.__name__
        print(f'{number} loops, average: {time:9.6f}s :: {name}')

    # Measure the execution times. Passing the globals() is an
    # easy way to make the functions available.
    timer = timeit.Timer(function, globals=globals())
    timer.autorange(print_result)
```

If you want to use `timeit` interactively, I would recommend using IPython, since it has a magic `%timeit` command that shows even more useful output:

```
$ ipython
In [1]: %timeit x=[]; [x.insert(0, i) for i in range(100000)]
2.5 s ± 112 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

In [2]: %timeit x=[]; [x.append(i) for i in range(100000)]
6.67 ms ± 252 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

In this case, IPython automatically takes care of the string wrapping and passing of `globals()`. Still, this is all very limited and useful only for comparing multiple methods of doing the same thing. When it comes to full Python applications, there are more methods available, as we will see later in this chapter.

To view the source of both IPython functions and regular modules, entering `object??` in the IPython shell returns the source. In this case, just enter `timeit??` to view the `timeit` IPython function definition.

The easiest way you can implement a function similar to the `%timeit` function is to call `timeit.main`:

```
import timeit

timeit.main(args=['[x for x in range(1000000)]'])
```

This effectively does the same as:

```
$ python3 -m timeit '[x for x in range(1000000)]'
```

The internals of the `timeit` module are nothing too special, but take care to minimize a few sources of inaccuracy, such as the setup and the teardown code. Additionally, the module reports the fastest run because other processes on your system can interfere with the measurement.

A basic version can be implemented with a few calls to `time.perf_counter` (the highest resolution timer available in Python), which is also used by `timeit` internally. The `timeit.default_timer` function is simply a reference to `time.perf_counter`. This basic implementation of the `timeit` function is comparable to the internals of the `timeit` module:

```
import gc
import time
import functools

assert time

TIMEIT_TEMPLATE = '''
```

```
def run(number):
    {setup}
    start = time.perf_counter()
    for i in range(number):
        {statement}
    stop = time.perf_counter()
    return stop - start
'''

def timeit(statement, setup='', number=1000000, globals_=None):
    # Get or create globals
    globals_ = globals() if globals_ is None else globals_

    # Create the test code so we can separate the namespace
    src = TIMEIT_TEMPLATE.format(
        statement=statement,
        setup=setup,
        number=number,
    )
    # Compile the source
    code = compile(src, '<source>', 'exec')

    # Define locals for the benchmarked code
    locals_ = {}

    # Execute the code so we can get the benchmark fuction
    exec(code, globals_, locals_)

    # Get the run function from locals() which was added by 'exec'
    run = functools.partial(locals_['run'], number=number)

    # Disable garbage collection to prevent skewing results
    gc.disable()
    try:
        result = run()
    finally:
        gc.enable()

    return result
```

The actual `timeit` code is a bit more advanced in terms of checking the input, but this example roughly shows how the `timeit.timeit` function can be implemented, including several of the features added for more precision:

- First, we can see that the code has a `number` parameter that defaults to 1 million. This has been done to reduce the result variance a little, as we will see when running the code.
- Second, the code disables the Python garbage collector so we don't get any slowdowns from Python deciding to clean up its memory.

When we actually call this code, we will see why a high value for `number` can be important:

```
>>> from T_02_custom_timeit import timeit

>>> statement = '[x for x in range(100)]'

>>> print('{:.7f}'.format(timeit(statement, number=1)))
0.0000064
>>> print('{:.7f}'.format(timeit(statement) / 1000000))
0.0000029
>>> print('{:.7f}'.format(timeit(statement, number=1)))
0.0000287
>>> print('{:.7f}'.format(timeit(statement) / 1000000))
0.0000029
```

Even though we called the exact same code each time, the single repetition took more than two times as long in the first run and more than 10 times as long in the second run compared to the 1 million repetitions version. To make your results more consistent and reliable between runs, it is always good to repeat your tests several times and `timeit` can certainly help with that.

The `timeit.repeat` function simply calls the `timeit.timeit` function several times and can be emulated using a list comprehension:

```
[timeit(statement) for _ in range(repeat)]
```

Now that we know how to test simple code statements, let's look at how to find slow statements in our code.

cProfile – Finding the slowest components

The `profile` and `cProfile` modules make it easily possible to analyze the relative CPU cycles used in a script/application. Be very careful not to compare these with the results from the `timeit` module. The `timeit` module tries as best as possible to give an accurate benchmark of the **absolute** amount of time it takes to execute a code snippet; the `profile`/`cProfile` modules are only useful for **relative** results because profiling increases the runtime. There are ways to make the results more accurate, but more about that later.

The `profile` and `cProfile` modules offer the exact same interface, but the latter is written in C and is much faster. I would recommend using `cProfile` if it is available on your system. If not, you can safely replace any occurrence of `cProfile` with `profile` in the following examples.

First profiling run

Let's profile our Fibonacci function from *Chapter 6*, *Decorators – Enabling Code Reuse by Decorating*, both with and without the cache function. First, the code:

```
import sys
import functools

@functools.lru_cache()
def fibonacci_cached(n):
    if n < 2:
        return n
    else:
        return fibonacci_cached(n - 1) + fibonacci_cached(n - 2)

def fibonacci(n):
    if n < 2:
        return n
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)

if __name__ == '__main__':
    n = 30
    if sys.argv[-1] == 'cache':
        fibonacci_cached(n)
    else:
        fibonacci(n)
```

For the sake of readability, all `cProfile` statistics will be stripped of the `percall` columns in all `cProfile` outputs. These columns contain the duration per function call, which is irrelevant for these examples since they will be either 0 or identical to the `cumtime` (cumulative time) column in nearly all cases.

First, we'll execute the function without cache:

```
$ python3 -m cProfile T_03_profile_fibonacci.py no_cache
   2692557 function calls (21 primitive calls) in 0.596 seconds

   Ordered by: standard name

   ncalls tottime cumtime filename:lineno(function)
        1   0.000   0.596 T_03_profile_fibonacci.py:1(<module>)
2692537/1   0.596   0.596 T_03_profile_fibonacci.py:13(fibonacci)
        1   0.000   0.000 functools.py:35(update_wrapper)
        1   0.000   0.000 functools.py:479(lru_cache)
        1   0.000   0.000 functools.py:518(decorating_function)
        1   0.000   0.596 {built-in method builtins.exec}
        7   0.000   0.000 {built-in method builtins.getattr}
        1   0.000   0.000 {built-in method builtins.isinstance}
        5   0.000   0.000 {built-in method builtins.setattr}
        1   0.000   0.000 {method 'disable' of '_lsprof.Profile...
        1   0.000   0.000 {method 'update' of 'dict' objects}
```

We see `2692557` calls in total, which is quite a lot of calls. We called the `test_fibonacci` function nearly 3 million times. That is where the profiling modules provide a lot of insight. Let's analyze the metrics a bit further, in the order they appear:

- `ncalls`: The number of calls that were made to the function.
- `tottime`: The total time spent in this function, **excluding** the sub-functions.
- `percall`: The time per call without sub-functions: `tottime / ncalls`.
- `cumtime`: The total time spent in this function, **including** sub-functions.
- `percall`: The time per call including sub-functions: `cumtime / ncalls`. This is distinct from the `percall` metric above, despite having the same name.

Which is the most useful depends on your use case. It's quite simple to change the sort order using the `-s` parameter within the default output. But now let's see what the result is with the cached version. Once again, with stripped output:

```
$ python3 -m cProfile T_03_profile_fibonacci.py cache
         51 function calls (21 primitive calls) in 0.000 seconds

   Ordered by: standard name

ncalls tottime cumtime filename:lineno(function)
     1  0.000  0.000 T_03_profile_fibonacci.py:1(<module>)
  31/1  0.000  0.000 T_03_profile_fibonacci.py:5(fibonacci_cached)
```

```
     1  0.000  0.000 functools.py:35(update_wrapper)
     1  0.000  0.000 functools.py:479(lru_cache)
     1  0.000  0.000 functools.py:518(decorating_function)
     1  0.000  0.000 {built-in method builtins.exec}
     7  0.000  0.000 {built-in method builtins.getattr}
     1  0.000  0.000 {built-in method builtins.isinstance}
     5  0.000  0.000 {built-in method builtins.setattr}
     1  0.000  0.000 {method 'disable' of '_lsprof.Profiler' ...}
     1  0.000  0.000 {method 'update' of 'dict' objects}
```

This time, we see a `tottime` of `0.000` because it's just too fast to measure. But also, while the `fibonacci_cached` function is still the most executed function, it's only being executed 31 times instead of 3 million times.

Calibrating your profiler

To illustrate the difference between `profile` and `cProfile`, let's try the uncached run again with the `profile` module instead. Just a heads up: this is much slower, so don't be surprised if it stalls a little:

```
$ python3 -m profile T_03_profile_fibonacci.py no_cache
         2692558 function calls (22 primitive calls) in 4.541 seconds

   Ordered by: standard name

   ncalls  tottime cumtime filename:lineno(function)
        1    0.000   4.530 :0(exec)
        7    0.000   0.000 :0(getattr)
        1    0.000   0.000 :0(isinstance)
        5    0.000   0.000 :0(setattr)
        1    0.010   0.010 :0(setprofile)
        1    0.000   0.000 :0(update)
        1    0.000   4.530 T_03_profile_fibonacci.py:1(<module>)
2692537/1    4.530   4.530 T_03_profile_fibonacci.py:13(fibonacci)
        1    0.000   0.000 functools.py:35(update_wrapper)
        1    0.000   0.000 functools.py:479(lru_cache)
        1    0.000   0.000 functools.py:518(decorating_function)
        1    0.000   4.541 profile:0(<code object <module> at ...
        0    0.000   0.000 profile:0(profiler)
```

The code now runs nearly 10 times more slowly, and the only difference is using the pure Python `profile` module instead of the `cProfile` module. This does indicate a big problem with the `profile` module. The overhead from the module itself is great enough to skew the results, which means we should account for that offset.

That's what the `Profile.calibrate()` function takes care of, as it calculates the performance bias incurred by the profile module. To calculate the bias, we can use the following script:

```
import profile

if __name__ == '__main__':
    profiler = profile.Profile()
    for i in range(10):
        print(profiler.calibrate(100000))
```

The numbers will vary slightly, but you should be able to get a fair estimate of the performance bias that the `profile` module introduces to your code. It effectively runs a bit of code both with and without profiling enabled and calculates a multiplier to apply to all results so they are closer to the actual duration.

If the numbers still vary a lot, you can increase the trials from `100000` to something even larger.

Note that with many modern processors, the burst CPU performance (the first few seconds) can vary greatly from the sustained CPU performance (2 minutes or more).

The CPU performance is also highly temperature-dependent, so if your system has a large CPU cooler or is water-cooled, it can take up to 20 minutes at 100% CPU load before the CPU performance becomes consistent. The bias after that 20 minutes would be completely unusable as a bias for a cold CPU.

This type of calibration only works for the `profile` module and should help a lot in achieving more accurate results. The bias can be set globally for all newly created profilers:

```
import profile

# The number here is bias calculated earlier
profile.Profile.bias = 9.809351906482531e-07
```

Or for a specific `Profile` instance:

```
import profile

profiler = profile.Profile(bias=9.809351906482531e-07)
```

Note that in general, a smaller bias is better to use than a large one because a large bias could cause very strange results. If the bias is large enough, you will even get negative timings. Let's give it a try for our Fibonacci code:

```
import sys
import pstats
```

```
import profile

...

if __name__ == '__main__':
    profiler = profile.Profile(bias=9.809351906482531e-07)
    n = 30

    if sys.argv[-1] == 'cache':
        profiler.runcall(fibonacci_cached, n)
    else:
        profiler.runcall(fibonacci, n)

    stats = pstats.Stats(profiler).sort_stats('calls')
    stats.print_stats()
```

While running it, it indeed appears that we've used a bias that's too large:

```
$ python3 T_05_profiler_large_bias.py
      2692539 function calls (3 primitive calls) in -0.746 seconds

   Ordered by: call count

   ncalls tottime cumtime filename:lineno(function)
2692537/1  -0.747  -0.747 T_05_profiler..._bias.py:15(fibonacci)
        1   0.000  -0.746 profile:0(<function fibonacci at ...>)
        1   0.000   0.000 :0(setprofile)
        0   0.000   0.000 profile:0(profiler)
```

Still, it shows how the code can be used properly. You can even incorporate the bias calculation within the script using a snippet like this:

```
import profile

if __name__ == '__main__':
    profiler = profile.Profile()
    profiler.bias = profiler.calibrate(100000)
```

It is not a bad idea to always have a snippet like this enabled when using the `profile` module. The only cost is the duration of the `calibrate()` run, and with a small number of trials (say, `10000`), it only takes about 0.2 seconds on my current system while still greatly increasing the accuracy of the results. Because of this properly calculated bias, the results can actually be more accurate than the `cProfile` module.

Selective profiling using decorators

Calculating simple timings is easy enough using decorators, but profiling can show a lot more and can also be applied selectively using decorators or context wrappers. Let's look at a `timer` and a `profiler` decorator:

```
import cProfile
import datetime
import functools

def timer(function):
    @functools.wraps(function)
    def _timer(*args, **kwargs):
        start = datetime.datetime.now()
        try:
            return function(*args, **kwargs)
        finally:
            end = datetime.datetime.now()
            print(f'{function.__name__}: {end - start}')

    return _timer

def profiler(function):
    @functools.wraps(function)
    def _profiler(*args, **kwargs):
        profiler = cProfile.Profile()
        try:
            profiler.enable()
            return function(*args, **kwargs)
        finally:
            profiler.disable()
            profiler.print_stats()

    return _profiler
```

Now that we have created the decorators, we can profile and time our functions with them:

```
@profiler
def profiled_fibonacci(n):
    return fibonacci(n)

@timer
def timed_fibonacci(n):
    return fibonacci(n)
```

```
def fibonacci(n):
    if n < 2:
        return n
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)

if __name__ == '__main__':
    timed_fibonacci(32)
    profiled_fibonacci(32)
```

The code is simple enough: just a basic `timer` and `profiler` decorator printing some default statistics. Which functions best for you depends on your use case, of course. The `timer()` decorator is very useful for quick performance tracking and/or a sanity check while developing. The `profiler()` decorator is great while you are actively working on the performance characteristics of a function.

The added advantage of this selective profiling is that the output is more limited, which helps with readability, albeit still much more verbose than the `timer()` decorator:

```
$ python3 T_06_selective_profiling.py
timed_fibonacci: 0:00:00.744912
         7049157 function calls (3 primitive calls) in 1.675 seconds

   Ordered by: standard name

   ncalls  tottime cumtime filename:lineno(function)
        1    0.000   1.675 T_06_select...py:31(profiled_fibonacci)
7049155/1    1.675   1.675 T_06_selec...profiling.py:41(fibonacci)
        1    0.000   0.000 {method 'disable' of '_lsprof.Profil...
```

As you can see, the profiler still makes the code about twice as slow, but it's definitely usable.

Using profile statistics

To get slightly more interesting profiling results, we will profile using the `pyperformance.benchmarks.bm_float` script.

The `pyperformance` library is the official Python benchmarks library optimized for the CPython interpreter. It contains a large (ever-growing) list of benchmarks to monitor the performance of the CPython interpreter under many scenarios.

It can be installed through `pip`:

```
$ pip3 install pyperformance
```

First, let's create the statistics using this script:

```
import sys
import pathlib
import pstats
import cProfile

import pyperformance

# pyperformance doesn't expose the benchmarks anymore so we need
# to manually add the path
pyperformance_path = pathlib.Path(pyperformance.__file__).parent
sys.path.append(str(pyperformance_path / 'data-files'))

# Now we can import the benchmark
from benchmarks.bm_float import run_benchmark as bm_float  # noqa


def benchmark():
    for i in range(10):
        bm_float.benchmark(bm_float.POINTS)


if __name__ == '__main__':
    profiler = cProfile.Profile()
    profiler.runcall(benchmark)
    profiler.dump_stats('bm_float.profile')

    stats = pstats.Stats('bm_float.profile')
    stats.strip_dirs()
    stats.sort_stats('calls', 'cumtime')
    stats.print_stats(10)
```

When executing the script, you should get something like this:

```
$ python3 T_07_profile_statistics.py
Sun May  1 06:14:26 2022    bm_float.profile

         6000012 function calls in 2.501 seconds
```

```
   Ordered by: call count, cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
  1000000    0.446    0.000    0.682    0.000 run_benchmark.py:15(__init__)
  1000000    0.525    0.000    0.599    0.000 run_benchmark.py:23(normalize)
  1000000    0.120    0.000    0.120    0.000 {built-in method math.cos}
  1000000    0.116    0.000    0.116    0.000 {built-in method math.sin}
  1000000    0.073    0.000    0.073    0.000 {built-in method math.sqrt}
   999990    0.375    0.000    0.375    0.000 run_benchmark.py:32(maximize)
       10    0.625    0.063    2.446    0.245 run_benchmark.py:46(benchmark)
       10    0.165    0.017    0.540    0.054 run_benchmark.py:39(maximize)
        1    0.055    0.055    2.501    2.501 T_07_profile_statistics.
py:17(benchmark)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.
Profiler' objects}
```

After running the script, you should have a `bm_float.profile` file containing the profiling results. As we can see in the script, these statistics can be viewed through the `pstats` module.

In some cases, it can be interesting to combine the results from multiple measurements. That is possible by specifying multiple files or by using `stats.add(*filenames)`.

The main advantage of saving these profile results to files is that several applications support this output and can visualize it in a clearer way. One option is SnakeViz, which uses your web browser to render the profile results interactively. Also, we have QCacheGrind, a very nice visualizer for profile statistics, but which requires some manual compiling to get running or some searching for binaries of course.

Let's look at the output from QCacheGrind. In the case of Windows, the QCacheGrindWin package provides a binary, whereas within Linux it is most likely available through your package manager, and with OS X you can try `brew install qcachegrind`.

However, there is one more package you will require: the `pyprof2calltree` package. It transforms the `profile` output into a format that QCacheGrind understands. So, after a simple `pip install pyprof2calltree`, we can now convert the `profile` file into a `callgrind` file:

```
$ pyprof2calltree -i bm_float.profile -o bm_float.callgrind
writing converted data to: bm_float.callgrind
$ qcachegrind bm_float.callgrind
```

This results in the running of the `QCacheGrind` application. After switching to the appropriate tabs, you should see something like the following screenshot:

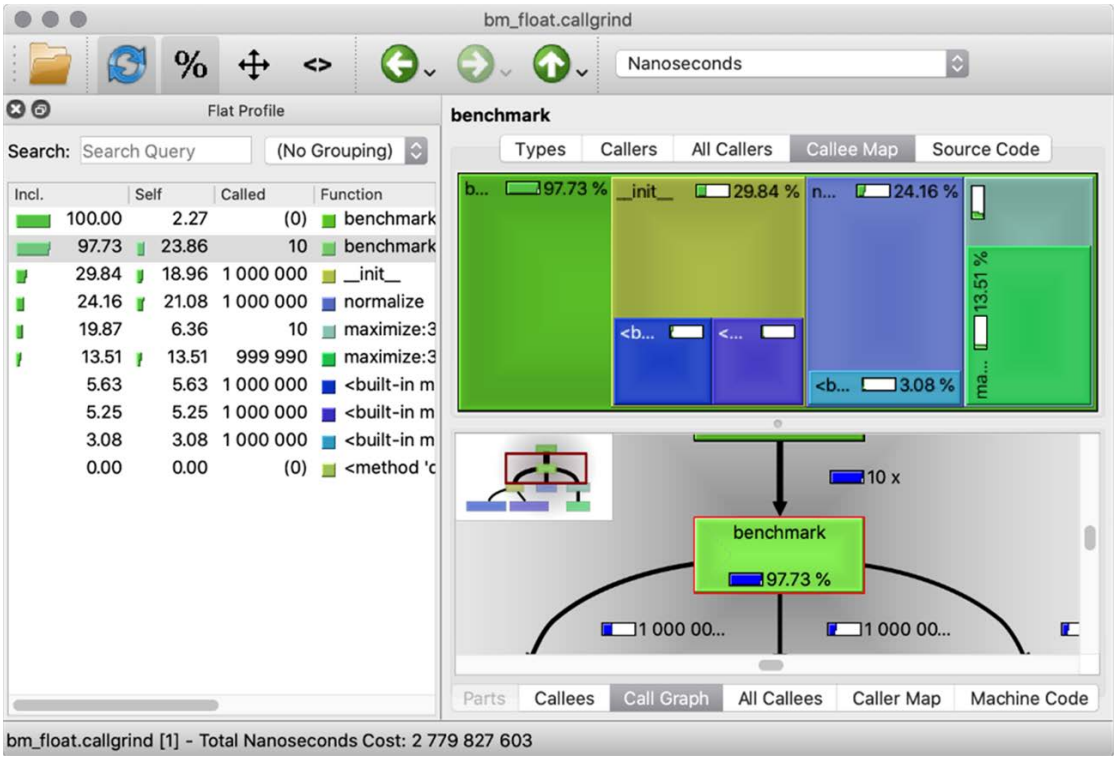


Figure 12.1: QCacheGrind

For a simple script such as this, pretty much all output works. However, with full applications, a tool such as QCacheGrind is invaluable. Looking at the output generated by QCacheGrind, it is immediately obvious which process took the most time. The structure at the top right shows bigger rectangles if the amount of time taken was greater, which is a very useful visualization of the chunks of CPU time that were used. The list at the left is very similar to `cProfile` and therefore nothing new. The tree at the bottom right can be very valuable or very useless, as it is in this case. It shows you the percentage of CPU time taken in a function and, more importantly, the relationship of that function with the other functions.

Because these tools scale depending on the input, the results are useful for just about any application. Whether a function takes 100 milliseconds or 100 minutes makes no difference – the output will show a clear overview of the slow parts, which is what we will try to fix.

Line profiler — Tracking performance per line

`line_profiler` is actually not a package that's bundled with Python, but it's far too useful to ignore. While the regular `profile` module profiles all (sub)functions within a certain block, `line_profiler` allows for profiling line *per line* within a function. The Fibonacci function is not best suited here, but we can use a prime number generator instead. But first, install `line_profiler`:

```
$ pip3 install line_profiler
```

Now that we have installed the `line_profiler` module (and with that the `kernprof` command), let's test `line_profiler`:

```
import itertools

@profile
def primes():
    n = 2
    primes = set()
    while True:
        for p in primes:
            if n % p == 0:
                break
        else:
            primes.add(n)
            yield n
        n += 1

if __name__ == '__main__':
    total = 0
    n = 2000
    for prime in itertools.islice(primes(), n):
        total += prime

    print('The sum of the first %d primes is %d' % (n, total))
```

You might be wondering where the `profile` decorator is coming from. It originates from the `line_profiler` module, which is why we have to run the script with the `kernprof` command:

```
$ kernprof --line-by-line T_08_line_profiler.py
The sum of the first 2000 primes is 16274627
Wrote profile results to T_08_line_profiler.py.lprof
```

As the command says, the results have been written to the `T_08_line_profiler.py.lprof` file, so we can now look at the output of that file. For readability, we've skipped the `Line #` column:

```
$ python3 -m line_profiler T_08_line_profiler.py.lprof
Timer unit: 1e-06 s

Total time: 1.34623 s
File: T_08_line_profiler.py
Function: primes at line 4
```

```
   Hits         Time  Per Hit   % Time  Line Contents
==============================================================
                                       @profile
                                       def primes():
      1          3.0      3.0      0.0      n = 2
      1          1.0      1.0      0.0      primes = set()
                                           while True:
2055131     625266.0      0.3     46.4          for p in primes:
2053131     707403.0      0.3     52.5              if n % p == 0:
  15388       4893.0      0.3      0.4                  break
                                               else:
   2000       1519.0      0.8      0.1              primes.add(n)
   2000        636.0      0.3      0.0              yield n
  17387       6510.0      0.4      0.5          n += 1
```

Wonderful output, isn't it? It makes it trivial to find the slow part within a bit of code. Within this code, the slowness is obviously originating from the loop, but within other code it might not be that clear.

This module can be added as an IPython extension as well, which enables the `%lprun` command within IPython. To load the extension, the `load_ext` command can be used from the IPython shell, `%load_ext line_profiler`.

We have seen several methods of measuring CPU performance and execution time. Now it's time to look at how to improve performance. Since this largely applies to CPU performance and not memory performance, we will cover that first. Later in this chapter, we will take a look at memory usage and leaks.

Improving execution time

Much can be said about performance optimization, but truthfully, if you have read the entire book up to this point, you know most of the Python-specific techniques for writing fast code. The most important factor in overall application performance will always be the choice of algorithms and, by extension, the data structures. Searching for an item within a `list (O(n))` is almost always a worse idea than searching for an item in a `dict` or `set (O(1))`, as we have seen in *Chapter 4*.

Naturally, there are more factors and tricks that can help make your application faster. The extremely abbreviated version of all performance tips is quite simple, however: do as little as possible. No matter how fast you make your calculations and operations, doing nothing at all will always be faster. The following sections cover some of the most common performance bottlenecks in Python and test a few common assumptions about performance, such as the performance of `try/except` blocks versus `if` statements, which can have a huge impact in many languages.

Some of the tricks in this section will be a trade-off between memory and execution time; others will trade readability with performance. When in doubt, go for readability by default and only improve performance if you have to.

Using the right algorithm

Within any application, the right choice of algorithm is by far the most important performance characteristic, which is why I am repeating it to illustrate the results of a bad choice. Consider the following:

```
In [1]: a = list(range(1000000))

In [2]: b = dict.fromkeys(range(1000000))

In [3]: %timeit 'x' in a
12.2 ms ± 245 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

In [4]: %timeit 'x' in b
40.1 ns ± 0.446 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
```

Checking whether an item is within a `list` is an `O(n)` operation, and checking whether an item is within a `dict` is an `O(1)` operation. This makes a huge difference when `n=1000000`; in this simple test, we can see that for 1 million items, it's 300,000 times faster.

The big-O notation (`O(...)`) is covered in more detail in *Chapter 4*, but we can provide a quick recap.

`O(n)` means that for a `list` with `len(some_list) = n`, it will take `n` steps to perform the operation. Consequently, `O(1)` means that it takes a constant amount of time regardless of the size of the collection.

All other performance tips combined might make your code twice as fast, but using the right algorithm for the job can cause a much greater improvement. Using an algorithm that takes `O(n)` time instead of $O(n^2)$ time will make your code `1000` times faster for `n=1000`, and with a larger `n`, the difference only grows further.

Global interpreter lock

One of the most obscure components of the CPython interpreter is the **global interpreter lock** (**GIL**), a **mutual exclusion lock** (**mutex**) required to prevent memory corruption. The Python memory manager is not thread-safe, which is why the GIL is needed. Without the GIL, multiple threads might alter memory at the same time, causing all sorts of unexpected and potentially dangerous results. The GIL is covered in much more detail in *Chapter 14*.

What is the impact of the GIL in a real-life application? Within single-threaded applications, it makes no difference whatsoever and is actually an extremely fast method for memory consistency.

Within multithreaded applications, however, it can slow your application down a bit because only a single thread can access the GIL at a time. If your code has to access the GIL a lot, it might benefit from some restructuring.

Luckily, Python offers a few other options for parallel processing. The `asyncio` module, which we will see in *Chapter 13*, can help a lot by switching tasks whenever you are waiting for a slow operation. In *Chapter 14*, we will see the `multiprocessing` library, which allows us to use multiple processors simultaneously.

try versus if

In many languages, a `try/except` type of block incurs quite a performance hit, but within Python, this is *not* the case as long as you don't hit the `except` block. If you do hit an `except`, it will be slightly heavier than an `if` statement, but not enough to be noticeable in most cases.

It's not that an `if` statement is heavy, but if you expect your `try/except` to succeed most of the time and only fail in rare cases, it is definitely a valid alternative. As always though, focus on readability and conveying the purpose of the code. If the intention of the code is clearer using an `if` statement, use the `if` statement. If `try/except` conveys the intention in a better way, use that.

Most programming languages depend on the use of the **Look Before You Leap** (**LBYL**) ideology. This means that you always check before you try, so if you are getting `some_key` from a `dict`, you use:

```
if some_key in some_dict:
    process_value(some_dict[some_key])
```

Because you are always doing the `if`, it hints that `some_key` is usually not part of `some_dict`.

Within Python, it is common to use the **Easier to Ask for Forgiveness than Permission** (**EAFP**) ideology when applicable. This means that the code assumes everything will work, but still catches errors:

```
try:
    process_value(some_dict[some_key])
except KeyError:
    pass
```

These two examples function mostly the same, but the latter gives the idea that you expect the key to be available and will catch errors if needed. This is one of the cases where the Zen of Python (explicit is better than implicit) applies.

The only caveat of the code above is that you might accidentally catch a `KeyError` from `process_value()`, so if you want to avoid that you should use the following code instead:

```
try:
    value = some_dict[some_key]
except KeyError:
    pass
else:
    process_value(value)
```

Which one you use comes mostly down to personal preference, but the takeaway should be that, with Python, both options are perfectly valid and will perform similarly.

Lists versus generators

Evaluating code lazily using generators is almost always a better idea than calculating the entire dataset. The most important rule of performance optimization is probably that you shouldn't calculate anything you're not going to use. If you're not sure that you are going to need it, don't calculate it.

Don't forget that you can easily chain multiple generators, so everything is calculated only when it's actually needed. Do be careful that this won't result in recalculation though; `itertools.tee()` is generally a better idea than recalculating your results completely.

To recap `itertools.tee()` from *Chapter 7*, a regular generator can only be consumed once, so if you need to process the results two or more times, you can use `itertools.tee()` to store the intermediate results:

```
>>> import itertools

# Without itertools.tee:
>>> generator = itertools.count()
>>> list(itertools.islice(generator, 5))
[0, 1, 2, 3, 4]
>>> list(itertools.islice(generator, 5))
[5, 6, 7, 8, 9]

>>> generator_a, generator_b = itertools.tee(itertools.count())
>>> list(itertools.islice(generator_a, 5))
[0, 1, 2, 3, 4]
>>> list(itertools.islice(generator_b, 5))
[0, 1, 2, 3, 4]
```

As you can see, if you forget to use `itertools.tee()` here, you would only process the results once, and both would process different values. The alternative fix is to use `list()` and store the intermediate results, but this can cost much more memory, and you are required to pre-calculate all items without knowing whether you actually need them all.

String concatenation

You might have seen benchmarks saying that using `+=` is much slower than joining strings because the `str` object (as is the case with `bytes`) is immutable. The result is that every time you do `+=` on a string, it will have to create a new object. At one point, this made quite a lot of difference indeed. With Python 3, however, most of the differences have vanished:

```
In [1]: %%timeit
   ...: s = ''
```

```
   ...: for i in range(1000000):
   ...:     s += str(i)
   ...:
1 loops, best of 3: 362 ms per loop

In [2]: %%timeit
   ...: ss = []
   ...: for i in range(1000000):
   ...:     ss.append(str(i))
   ...: s = ''.join(ss)
   ...:
1 loops, best of 3: 332 ms per loop

In [3]: %timeit ''.join(str(i) for i in range(1000000))
1 loops, best of 3: 324 ms per loop

In [4]: %timeit ''.join([str(i) for i in range(1000000)])
1 loops, best of 3: 294 ms per loop
```

There are still some differences, of course, but they are so small that I recommend you simply ignore them and choose the most readable option instead.

Addition versus generators

As is the case with string concatenation, addition from a loop was significantly slower with older Python versions, but the difference is now too small to consider:

```
In [1]: %%timeit
   ...: x = 0
   ...: for i in range(1000000):
   ...:     x += i
   ...:
10 loops, best of 3: 73.2 ms per loop

In [2]: %timeit x = sum(i for i in range(1000000))
10 loops, best of 3: 75.3 ms per loop

In [3]: %timeit x = sum([i for i in range(1000000)])
10 loops, best of 3: 71.2 ms per loop

In [4]: %timeit x = sum(range(1000000))
10 loops, best of 3: 25.6 ms per loop
```

What does help, though, is letting Python handle everything internally using native functions, as can be seen in the last example.

Map versus generators and list comprehensions

Once again, readability generally counts more than performance, so only rewrite for performance if it really makes a difference. There are a few cases where `map()` is faster than list comprehensions and generators, but only if the `map()` function can use a predefined function. As soon as you need to whip out `lambda`, it's actually slower. Not that it matters much, since readability should be key anyhow. If `map()` makes your code more readable than a generator or list comprehension, feel free to use it. Otherwise, I would not recommend it:

```
In [1]: %timeit list(map(lambda x: x/2, range(1000000)))
10 loops, best of 3: 182 ms per loop

In [2]: %timeit list(x/2 for x in range(1000000))
10 loops, best of 3: 122 ms per loop

In [3]: %timeit [x/2 for x in range(1000000)]
10 loops, best of 3: 84.7 ms per loop
```

As you can see, the list comprehension is quite a bit faster than the generator. In many cases, I would still recommend the generator over the list comprehension, though, if only because of the memory usage and the potential laziness.

If, for some reason, you are only going to use the first 10 items when generating 1,000 items, you're still wasting a lot of resources by calculating the full list of items.

Caching

We have already covered the `functools.lru_cache` decorator in *Chapter 6*, *Decorators – Enabling Code Reuse by Decorating*, but its importance should not be underestimated. Regardless of how fast and smart your code is, not having to calculate results is always better and that's what caching does. Depending on your use case, there are many options available. Within a simple script, `functools.lru_cache` is a very good contender, but between multiple executions of an application, the `cPickle` module can be a lifesaver as well.

We have already seen the effects of this with the `fibonacci_cached` function in the `cProfile` section of this chapter, which uses `functools.lru_cache()`.

There are several scenarios where you need a more powerful solution, however:

- If you need caching between multiple executions of a script
- If you need caching shared across multiple processes

- If you need caching shared across multiple servers

At least for the first two scenarios, you could write the cache to a local pickle/CSV/JSON/YAML/DBM/etc. file. This is a perfectly valid solution that I use often.

If you need a more powerful solution, however, I can highly recommend taking a look at **Redis**. The Redis server is a fully in-memory server that is extremely fast and has many useful data structures available. If you see articles or tutorials about improving performance using Memcached, simply replace Memcached with Redis everywhere. Redis is superior to Memcached in every way and, in its most basic form, the API is compatible.

Lazy imports

A common problem in application load times is that everything is loaded immediately at the start of the program while, with many applications, this is actually not needed and certain parts of the application only require loading when they are actually used. To facilitate this, you can occasionally move the imports inside of functions so they can be loaded on demand.

While it's a valid strategy in some cases, I don't generally recommend it for two reasons:

- It makes your code less clear; having all imports in the same style at the top of the file improves readability.
- It doesn't make the code faster as it just moves the load time to a different part.

Using slots

The `__slots__` feature was written by Guido van Rossum to enhance Python performance. Effectively what the `__slots__` feature does is specify a fixed list of attributes for a class. When `__slots__` are used, several changes are made to a class and several (side-)effects must be considered:

- All attributes must be explicitly named in the `__slots__`. It is not possible to do `some_instance.some_variable = 123` if `some_variable` is not in `__slots__`.
- Because the list of attributes is fixed in `__slots__`, there is no longer any need for a `__dict__` attribute, which saves memory.
- Attribute access is faster because there is no intermediate lookup through `__dict__`.
- It is not possible to use multiple inheritance if both parents have defined `__slots__`.

So, how much performance benefit can `__slots__` give us? Well, let's give it a test:

```
import timeit
import functools

class WithSlots:
    __slots__ = 'eggs',

class WithoutSlots:
    pass
```

```
with_slots = WithSlots()
no_slots = WithoutSlots()

def test_set(obj):
    obj.eggs = 5

def test_get(obj):
    return obj.eggs

timer = functools.partial(
    timeit.timeit,
    number=20000000,
    setup='\n'.join((
        f'from {__name__} import with_slots, no_slots',
        f'from {__name__} import test_get, test_set',
    )),
)
for function in 'test_set', 'test_get':
    print(function)
    print('with slots', timer(f'{function}(with_slots)'))
    print('with slots', timer(f'{function}(no_slots)'))
```

When we actually run this code, we can definitely see some improvements from using `__slots__`:

```
$ python3 T_10_slots_performance.py
test_set
with slots 1.748628467
with slots 2.0184642979999996
test_get
with slots 1.5832197570000002
with slots 1.6575410809999997
```

In most cases, I would argue that the 5-15% difference in performance isn't going to help you that much. However, if it's applied to a bit of code that is near the core of your application and executed very often, it can help.

Don't expect miracles from this method, but use it when you need it.

Using optimized libraries

This is actually a very broad tip, but useful nonetheless. If there's a highly optimized library that suits your purpose, you most likely won't be able to beat its performance without a significant amount of effort. Libraries such as `numpy`, `pandas`, `scipy`, and `sklearn` are highly optimized for performance and their native operations can be incredibly fast. If they suit your purpose, be sure to give them a try.

Before you can use `numpy`, you need to install it: `pip3 install numpy`.

Just to illustrate how fast `numpy` can be compared to plain Python, refer to the following:

```
In [1]: import numpy

In [2]: a = list(range(1000000))

In [3]: b = numpy.arange(1000000)

In [4]: %timeit c = [x for x in a if x > 500000]
10 loops, best of 3: 44 ms per loop

In [5]: %timeit d = b[b > 500000]
1000 loops, best of 3: 1.61 ms per loop
```

The `numpy` code does exactly the same as the Python code, except that it uses `numpy` arrays instead of Python lists. This little difference has made the code more than 25 times faster.

Just-in-time compiling

Just-in-time (**JIT**) compiling is a method of dynamically compiling (parts of) an application during runtime. Because there is much more information available at runtime, this can have a huge effect and make your application much faster.

When it comes to JIT compiling, you currently have three options:

- **Pyston**: An alternative, currently Linux only, CPython-compatible Python interpreter.
- **Pypy**: A really fast alternative Python interpreter without full CPython compatibility.
- **Numba**: A package that allows for JIT compiling per function and execution on either the CPU or the GPU.
- **CPython 3.12 and 3.13**? At the time of writing, there is little concrete data about the upcoming Python releases, but there are plans to greatly increase the CPython interpreter performance. How much will be achieved and how well it will work is currently unknown, but the ambitious plan is to make CPython 5x faster over the next 5 releases (with 3.10 being the first in the series). The expectation is to add JIT compiling in CPython 3.12 and extend that further in 3.13.

If you are looking for global JIT compiling in existing projects, I can currently recommend trying Pyston. It is a CPython fork that promises about a 30% performance increase without having to change any code. In addition, because it is CPython-compatible, you can still use regular CPython modules.

The downside is that it currently only supports Linux systems and, as will always be the case with forks, it's behind the current Python version. At the time of writing, CPython is at Python 3.10.1, whereas Pyston is at Python 3.8.

If compatibility with all CPython modules is not a requirement for you and you don't require Python features that are too recent, PyPy3 can also offer amazing performance in many cases. They are up to Python 3.7, whereas the main Python release is at 3.10.1 at the time of writing. That makes PyPy roughly 2-3 years behind CPython in terms of features, but I doubt this is a big issue. The differences between Python 3.7, 3.8, 3.9, and 3.10 are largely incremental and Python 3.7 is already a very well-rounded Python version.

The `numba` package provides selective JIT compiling for you, allowing you to mark the functions that are JIT compiler-compatible. Essentially, if your functions follow the functional programming paradigm of basing the calculations only on the input, then it will most likely work with the JIT compiler.

Here is a basic example of how the `numba` JIT compiler can be used:

```
import numba

@numba.jit
def sum(array):
    total = 0.0
    for value in array:
        total += value
    return value
```

If you are using `numpy` or `pandas`, you will most likely benefit from looking at `numba`.

Another very interesting fact to note is that `numba` supports not only CPU-optimized execution, but GPU as well. This means that for certain operations you can use the fast processor in your video card to process the results.

Converting parts of your code to C

We will see more about this in *Chapter 17*, *Extensions in C/C++, System Calls, and C/C++ Libraries*, but if high performance is really required, then a native C function can help quite a lot. This doesn't even have to be that difficult; the Cython module makes it trivial to write parts of your code with performance very close to native C code.

The following is an example from the Cython manual to approximate the value of pi:

```
cdef inline double recip_square(int i):
    return 1./(i*i)

def approx_pi(int n=10000000):
    cdef double val = 0.
```

```
    cdef int k
    for k in range(1,n+1):
        val += recip_square(k)
    return (6 * val)**.5
```

While there are some small differences, such as `cdef` instead of `def` and type definitions such as `int i` instead of just `i` for the values and parameters, the code is largely the same as regular Python would be, but certainly much faster.

Memory usage

So far, we have simply looked at the execution times and largely ignored the memory usage of the scripts. In many cases, the execution times are the most important, but memory usage should not be ignored. In almost all cases, CPU and memory are traded; an algorithm either uses a lot of CPU time or a lot of memory, which means that both do matter a lot.

Within this section, we are going to look at:

- Analyzing memory usage
- When Python leaks memory and how to avoid these scenarios
- How to reduce memory usage

tracemalloc

Monitoring memory usage used to be something that was only possible through external Python modules such as **Dowser** or **Heapy**. While those modules still work, they are partially obsolete now because of the `tracemalloc` module. Let's give the `tracemalloc` module a try to see how easy memory usage monitoring is nowadays:

```
import tracemalloc

if __name__ == '__main__':
    tracemalloc.start()

    # Reserve some memory
    x = list(range(1000000))

    # Import some modules
    import os
    import sys
    import asyncio

    # Take a snapshot to calculate the memory usage
    snapshot = tracemalloc.take_snapshot()
```

```
    for statistic in snapshot.statistics('lineno')[:10]:
        print(statistic)
```

This results in:

```
$ python3 T_11_tracemalloc.py
T_11_tracemalloc.py:8: size=34.3 MiB, count=999746, average=36 B
<frozen importlib._bootstrap_external>:587: size=1978 KiB, coun...
<frozen importlib._bootstrap>:228: size=607 KiB, count=5433, av...
abc.py:85: size=32.6 KiB, count=155, average=215 B
enum.py:172: size=26.2 KiB, count=134, average=200 B
collections/__init__.py:496: size=24.1 KiB, count=117, average=...
enum.py:225: size=23.3 KiB, count=451, average=53 B
enum.py:391: size=15.0 KiB, count=21, average=729 B
<frozen importlib._bootstrap_external>:64: size=14.3 KiB, count...
enum.py:220: size=12.2 KiB, count=223, average=56 B
```

You can easily see how every part of the code allocated memory and where it might be wasted. While it might still be unclear which part was actually causing the memory usage, there are options for that as well, as we will see in the following sections.

Memory Profiler

The `memory_profiler` module is very similar to `line_profiler` discussed earlier, but for memory usage instead. Installing it is as easy as `pip install memory_profiler`, but the optional `pip install psutil` is also highly recommended (and required in the case of Windows) as it increases your performance by a large amount. To test `memory_profiler`, we will use the following script:

```
import memory_profiler

@memory_profiler.profile
def main():
    n = 100000
    a = [i for i in range(n)]
    b = [i for i in range(n)]
    c = list(range(n))
    d = list(range(n))
    e = dict.fromkeys(a, b)
    f = dict.fromkeys(c, d)

if __name__ == '__main__':
    main()
```

Note that we actually import `memory_profiler` here although that is not strictly required. It can also be executed through `python3 -m memory_profiler your_scripts.py`:

```
Filename: CH_12_performance/T_12_memory_profiler.py

Mem usage  Increment  Occurrences  Line Contents
=============================================
 14.7 MiB   14.7 MiB           1  @memory_profiler.profile
                                  def main():
 14.7 MiB    0.0 MiB           1      n = 100000
 18.5 MiB    3.8 MiB      100003      a = [i for i in range(n)]
 22.4 MiB    3.9 MiB      100003      b = [i for i in range(n)]
 26.3 MiB    3.9 MiB           1      c = list(range(n))
 30.2 MiB    3.9 MiB           1      d = list(range(n))
 39.9 MiB    9.8 MiB           1      e = dict.fromkeys(a, b)
 44.9 MiB    5.0 MiB           1      f = dict.fromkeys(c, d)
 44.9 MiB    0.0 MiB           1      assert e
 44.9 MiB    0.0 MiB           1      assert f
```

Even though everything runs as expected, you might be wondering about the varying amounts of memory used by the lines of code here.

Why does `e` take `9.8 MiB` and `f` `5.0 MiB`? This is caused by the Python memory allocation code; it reserves memory in larger blocks, which is subdivided and reused internally. Another problem is that `memory_profiler` takes snapshots internally, which results in memory being attributed to the wrong variables in some cases. The variations should be small enough not to make a large difference in the end, but some changes are to be expected.

This module can be added as an IPython extension as well, which enables the `%mprun` command within IPython. To load the extension, the `load_ext` command can be used from the IPython shell: `%load_ext memory_profiler`. Another very useful command is `%memit`, which is the memory equivalent of the `%timeit` command.

Memory leaks

The usage of these modules will generally be limited to the search for memory leaks. In particular, the `tracemalloc` module has a few features to make that fairly easy. The Python memory management system is fairly straightforward; it has a simple reference counter to see whether an object is (still) used. While this works great in most cases, it can easily introduce memory leaks when circular references are involved. The basic premise of a memory leak with leak detection code looks like this:

```
import tracemalloc


class SomeClass:
    pass


if __name__ == '__main__':
    # Initialize some variables to ignore them from the leak
    # detection
    n = 100000

    tracemalloc.start()
    # Your application should initialize here

    snapshot_a = tracemalloc.take_snapshot()
    instances = []

    # This code should be the memory leaking part
    for i in range(n):
        a = SomeClass()
        b = SomeClass()
        # Circular reference. a references b, b references a
        a.b = b
        b.a = a
        # Force Python to keep the object in memory for now
        instances.append(a)

    # Clear the list of items again. Now all memory should be
    # released, right?
    del instances
    snapshot_b = tracemalloc.take_snapshot()

    statistics = snapshot_b.compare_to(snapshot_a, 'lineno')
    for statistic in statistics[:10]:
        print(statistic)
```

The line numbers in the code above are provided as a reference for the `tracemalloc` output and are not functionally part of the code.

The big problem in this code is that we have two objects that are referencing each other. As we can see, `a.b` is referencing `b`, and `b.a` is referencing `a`. This loop makes it so that Python doesn't immediately understand that the objects can be safely deleted from memory.

Let's see how badly this code is actually leaking:

```
$ python3 T_12_memory_leaks.py
T_12_memory_leaks.py:25: size=22.1 MiB (+22.1 MiB), count=199992 (+199992),
average=116 B
T_12_memory_leaks.py:24: size=22.1 MiB (+22.1 MiB), count=199992 (+199992),
average=116 B
T_12_memory_leaks.py:22: size=4688 KiB (+4688 KiB), count=100000 (+100000),
average=48 B
T_12_memory_leaks.py:21: size=4688 KiB (+4688 KiB), count=100000 (+100000),
average=48 B
tracemalloc.py:423: size=88 B (+88 B), count=2 (+2), average=44 B
tracemalloc.py:560: size=48 B (+48 B), count=1 (+1), average=48 B
tracemalloc.py:315: size=40 B (+40 B), count=1 (+1), average=40 B
T_12_memory_leaks.py:20: size=28 B (+28 B), count=1 (+1), average=28 B
```

This example shows a leak of 22.1 megabytes due to the nearly 200,000 instances of `SomeClass`. Python correctly lets us know that this memory was allocated at lines 24 and 25, which can really help when trying to ascertain what is causing the memory usage in your application.

The Python garbage collector (`gc`) is smart enough to clean circular references like these eventually, but it won't clean them until a certain limit is reached. More about that soon.

Circular references

Whenever you want to have a circular reference that does not cause memory leaks, the `weakref` module is available. It creates references that don't count toward the object reference count. Before we look at the `weakref` module, let's take a look at the object references themselves through the eyes of the Python garbage collector (`gc`):

```
import gc

class SomeClass(object):
    def __init__(self, name):
        self.name = name

    def __repr__(self):
        return f'<{self.__class__.__name__}: {self.name}'

# Create the objects
a = SomeClass('a')
b = SomeClass('b')
```

```
# Add some circular references
a.b = a
b.a = b

# Remove the objects
del a
del b

# See if the objects are still there
print('Before manual collection:')
for object_ in gc.get_objects():
    if isinstance(object_, SomeClass):
        print('\t', object_, gc.get_referents(object_))

print('After manual collection:')
gc.collect()
for object_ in gc.get_objects():
    if isinstance(object_, SomeClass):
        print('\t', object_, gc.get_referents(object_))

print('Thresholds:', gc.get_threshold())
```

First, we create two instances of `SomeClass` and add some circular references between them. Once that is done, we delete them from memory, except that they are not actually deleted until the garbage collector runs.

To verify this, we inspect the objects in memory through `gc.get_objects()`, and until we tell the garbage collector to manually collect, they stay in memory.

Once we do run `gc.collect()` to manually call the garbage collector, the objects are gone from memory:

```
$ python3 T_14_garbage_collection.py
Before manual collection:
        <SomeClass: a> [{'name': 'a', 'b': <SomeClass: a>}, <class '__main__.
SomeClass'>]
        <SomeClass: b> [{'name': 'b', 'a': <SomeClass: b>}, <class '__main__.
SomeClass'>]
After manual collection:
Thresholds: (700, 10, 10)
```

Now, you might wonder, are you always required to manually call `gc.collect()` to remove these references? No, that is not needed, as the Python garbage collector will automatically collect once thresholds have been reached.

By default, the thresholds for the Python garbage collector are set to `700, 10, 10` for the three generations of collected objects. The collector keeps track of all the memory allocations and deallocations in Python, and as soon as the number of allocations minus the number of deallocations reaches `700`, the object is either removed if it's not referenced anymore, or it is moved to the next generation if it still has a reference. The same is repeated for generations 2 and 3, albeit with the lower thresholds of 10.

This begs the question: where and when is it useful to manually call the garbage collector? Since the Python memory allocator reuses blocks of memory and only rarely releases it, for long-running scripts the garbage collector can be very useful. That's exactly where I recommend its usage: long-running scripts in memory-strapped environments and, specifically, right before you **allocate** a large amount of memory. If you call the garbage collector before doing a memory-intensive operation, you can maximize the amount of reuse of the memory that Python has previously reserved.

Analyzing memory usage using the garbage collector

The `gc` module can help you a lot when looking for memory leaks as well. The `tracemalloc` module can show you the parts that take the most memory in bytes, but the `gc` module can help you find the most commonly occurring object types (for example, `SomeClass`, `int`, and `list`). Just be careful when setting the garbage collector debug settings such as `gc.set_debug(gc.DEBUG_LEAK)`; this returns a large amount of output even if you don't reserve any memory yourself. Let's see the output for one of the most basic scripts you can get:

```
import gc
import collections
if __name__ == '__main__':
    objects = collections.Counter()
    for object_ in gc.get_objects():
        objects[type(object_)] += 1

    print(f'Different object count: {len(objects)}')
    for object_, count in objects.most_common(10):
        print(f'{count}: {object_}')
```

Now, when we run the code, you can see a bit of what has been added to our memory with such a simple script:

```
$ python3 T_15_garbage_collection_viewing.py
Different object count: 42
1058: <class 'wrapper_descriptor'>
887: <class 'function'>
677: <class 'method_descriptor'>
652: <class 'builtin_function_or_method'>
545: <class 'dict'>
484: <class 'tuple'>
431: <class 'weakref'>
```

```
251: <class 'member_descriptor'>
238: <class 'getset_descriptor'>
76: <class 'type'>
```

As you can see, there are actually 42 different types of objects that should have been shown here, but even without that, the number of different objects in memory is impressive, if you ask me. With just a little bit of extra code, the output can quickly explode and become unusable without significant filtering.

Weak references

An easy method to make the work easier for the garbage collector is to use **weak references**. These are references to variables that are not included when counting the references to a variable. Since the garbage collector removes an object from memory when its reference count gets to zero, this can help a lot with memory leaks.

In the earlier example, we saw that the objects weren't removed until we manually called `gc.collect()`. Now we will see what happens if we use the `weakref` module instead:

```
import gc
import weakref
class SomeClass(object):
    def __init__(self, name):
        self.name = name

    def __repr__(self):
        return '<%s: %s>' % (self.__class__.__name__, self.name)

def print_mem(message):
    print(message)
    for object_ in gc.get_objects():
        if isinstance(object_, SomeClass):
            print('\t', object_, gc.get_referents(object_))

# Create the objects
a = SomeClass('a')
b = SomeClass('b')

# Add some weak circular references
a.b = weakref.ref(a)
b.a = weakref.ref(b)

print_mem('Objects in memory before del:')

# Remove the objects
```

```
del a
del b

# See if the objects are still there
print_mem('Objects in memory after del:')
```

Now let's see what remained this time:

```
$ python3 T_16_weak_references.py
Objects in memory before del:
         <SomeClass: a> [{'name': 'a', 'b': ...}, ...]
         <SomeClass: b> [{'name': 'b', 'a': ...}, ...]
Objects in memory after del:
```

Perfect – no instances of `SomeClass` exist in memory after `del`, which is exactly what we had hoped for.

Weakref limitations and pitfalls

You might be wondering what happens when you still try to reference a since-removed `weakref`. As you would expect, the object is gone now, so you can no longer use it. What is more, not all objects can be used through weak references directly:

```
>>> import weakref

>>> weakref.ref(dict(a=123))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot create weak reference to 'dict' object

>>> weakref.ref([1, 2, 3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot create weak reference to 'list' object

>>> weakref.ref('test')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot create weak reference to 'str' object

>>> weakref.ref(b'test')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot create weak reference to 'bytes' object

>>> a = weakref.WeakValueDictionary(a=123)
```

```
Traceback (most recent call last):
    ...
TypeError: cannot create weak reference to 'int' object
```

We can use `weakref` for custom classes though, so we can subclass the types before we create the `weakref`:

```
>>> class CustomDict(dict):
...     pass

>>> weakref.ref(CustomDict())
<weakref at 0x...; dead>
```

For `dict` and `set` instances, the `weakref` library also has the `weakref.WeakKeyDictionary`, `weakref.WeakValueDictionary`, and `weakref.WeakSet` classes. These behave similarly to the regular instances of `dict` and `set`, but remove the values based on the key or value.

We need to be careful when using a `weakref`, of course. As soon as all regular references are deleted, the object will be inaccessible:

```
>>> class SomeClass:
...     def __init__(self, name):
...         self.name = name

>>> a = SomeClass('a')
>>> b = weakref.proxy(a)
>>> b.name
'a'
>>> del a
>>> b.name
Traceback (most recent call last):
    ...
ReferenceError: weakly-referenced object no longer exists
```

After deleting `a`, which is the only real reference to the `SomeClass` instance, we cannot use the instance anymore. While this is to be expected, you should be wary of this problem if your main reference has a chance to disappear.

Whenever you are working with large self-referencing data structures, it can be a good idea to use the `weakref` module. However, don't forget to check if your instance still exists before using it.

Reducing memory usage

In general, memory usage probably won't be your biggest problem in Python, but it can still be useful to know what you can do to reduce it. When trying to reduce memory usage, it's important to understand how Python allocates memory.

There are four concepts that you need to know about within the Python memory manager:

- First, we have the **heap**. The heap is the collection of all Python-managed memory. Note that this is separate from the regular heap and mixing the two could result in corrupt memory and crashes.
- Second are the **arenas**. These are the chunks that Python requests from the system. These chunks have a fixed size of 256 KiB each and they are the objects that make up the heap.
- Third we have the **pools**. These are the chunks of memory that make up the arenas. These chunks are 4 KiB each. Since the pools and arenas have fixed sizes, they are simple arrays.
- Fourth and last, we have the **blocks**. The Python objects get stored within these and every block has a specific format depending on the data type. Since an integer takes up more space than a character, for efficiency, a different block size is used.

Now that we know how the memory is allocated, we can also understand how it can be returned to the operating system and why this is often very hard to do.

Releasing a block back to the pool is easy enough: a simple `del some_variable` followed by a `gc.collect()` should do the trick. The problem is that this is no guarantee that the memory will be released back to the operating system yet.

To illustrate what needs to happen in order for the memory to release to the operating system:

- All blocks in a pool need to be released before the pool can be released
- All pools in an arena need to be released before the arena can be released
- Once the arena has been released to the heap, memory *might* be released to the operating system, but that depends on the C runtime and/or operating system

That is why I would always recommend running `gc.collect()` in long-running scripts right before you start allocating large blocks of memory.

It is a common and incorrect misconception that Python never releases any memory to the system. Before Python 2.5, this was indeed the case because arenas were never freed to the heap.

Let's illustrate the effects of allocating and releasing memory by allocating and releasing twice:

```
import os
import psutil

def print_usage(message):
    process = psutil.Process(os.getpid())
    usage = process.memory_info().rss / (1 << 20)
    print(f'Memory usage {message}: {usage:.1f} MiB')

def allocate_and_release():
```

```
    # Allocate large block of memory
    large_list = list(range(1000000))
    print_usage('after allocation')

    del large_list
    print_usage('after releasing')

print_usage('initial')
allocate_and_release()
allocate_and_release()
```

You might expect that the memory usage after the second block has been released will be near identical to after the first block has been released, or even back to the original state. Let's see what actually happens:

```
$ python3 T_18_freeing_memory.py
Memory usage initial: 9.4 MiB
Memory usage after allocation: 48.1 MiB
Memory usage after releasing: 17.3 MiB
Memory usage after allocation: 55.7 MiB
Memory usage after releasing: 25.0 MiB
```

That's odd, isn't it? The memory usage has grown between the two allocations. The truth is that I cherry-picked the result somewhat and that the output changes between each run, because releasing memory back to the operating system is not a guarantee that the operating system will immediately handle it. In some other cases, the memory had properly returned to 17 MiB.

The astute among you might wonder if the results are skewed because I forgot the `gc.collect()`. In this case, the answer is no because the memory allocation is large enough to immediately trigger the garbage collector by itself and the difference is negligible.

This is roughly the best case, however – just a few contiguous blocks of memory. The real challenge is when you have many variables so only parts of the pools/arenas are used. Python uses some heuristics to find space in an empty arena so it doesn't have to allocate new arenas when you are storing new variables, but that does not always succeed, of course. This is a case where running `gc.collect()` before allocation can help because it can tell Python which pools are now free.

It is important to note that the regular heap and Python heap are maintained separately, as mixing them can result in corruption and/or the crashing of applications. Unless you write your own Python extensions in C/C++, you will probably never have to worry about manual memory allocation though.

Generators versus lists

The most important tip is to use generators whenever possible. Python 3 has come a long way in replacing lists with generators already, but it really pays to keep that in mind as it saves not only memory, but CPU as well, when not all of that memory needs to be kept at the same time.

To illustrate the difference:

```
Line #    Mem usage    Increment   Line Contents
================================================
     4     11.0 MiB      0.0 MiB   @memory_profiler.profile
     5                             def main():
     6     11.0 MiB      0.0 MiB    a = range(1000000)
     7     49.7 MiB     38.6 MiB    b = list(range(1000000))
```

The `range()` generator takes such little memory that it doesn't even register, whereas the list of numbers takes `38.6 MiB`.

Recreating collections versus removing items

One very important detail about collections in Python is that many of them can only grow; they won't just shrink by themselves. To illustrate:

```
Mem usage    Increment   Line Contents
=====================================
 11.5 MiB      0.0 MiB   @memory_profiler.profile
                         def main():
                         # Generate a huge dict
 26.3 MiB     14.8 MiB   a = dict.fromkeys(range(100000))

                         # Remove all items
 26.3 MiB      0.0 MiB   for k in list(a.keys()):
 26.3 MiB      0.0 MiB   del a[k]

                         # Recreate the dict
 23.6 MiB     -2.8 MiB   a = dict((k, v) for k, v in a.items())
```

Even after removing all items from the `dict`, the memory usage remains the same. This is one of the most common memory usage mistakes made with lists and dictionaries. The only way to reclaim the memory is by recreating the object. Or, never allocate the memory at all by using generators.

Using slots

In addition to the performance benefits of using `__slots__` that we saw earlier in this chapter, `__slots__` can also help to reduce memory usage. As a recap, `__slots__` allows you to specify which fields you want to store in a class and it skips all the others by not implementing `instance.__dict__`.

While this method does save a little bit of memory in your class definitions, the effect is often limited. For a nearly empty class with just a single/tiny attribute such as a `bool` or `byte`, this can make quite a bit of difference. For classes that actually store a bit of data, the effect can diminish quickly.

The biggest caveat of `__slots__` is that multiple inheritance is impossible if both parent classes have `__slots__` defined. Beyond that, it can be used in nearly all cases.

You might wonder if `__slots__` will limit dynamic attribute assignments, effectively blocking you from doing `Spam.eggs = 123` if `eggs` was not part of `__slots__`. And you are right – partially, at least. With a standard fixed list of attributes in `__slots__`, you cannot dynamically add new attributes – but you can if you add `__dict__` to `__slots__`.

I'm embarrassed to say that it took me about 15 years before I found out about this feature, but knowing about this feature makes `__slots__` so much more versatile that I really feel like I should mention it.

Let's now illustrate the difference in memory usage:

```
import memory_profiler

class Slots(object):
    __slots__ = 'index', 'name', 'description'

    def __init__(self, index):
        self.index = index
        self.name = 'slot %d' % index
        self.description = 'some slot with index %d' % index

class NoSlots(object):
    def __init__(self, index):
        self.index = index
        self.name = 'slot %d' % index
        self.description = 'some slot with index %d' % index

@memory_profiler.profile
def main():
    slots = [Slots(i) for i in range(25000)]
    no_slots = [NoSlots(i) for i in range(25000)]
    return slots, no_slots

if __name__ == '__main__':
    main()
```

And the memory usage:

```
Mem usage Increment Occurrences Line Contents
=============================================
38.4 MiB  38.4 MiB          1 @memory_profiler.profile
                              def main():
44.3 MiB   5.9 MiB      25003     slots = [Slots(i) for i in range(25000)]
52.4 MiB   8.1 MiB      25003     no_slots = [NoSlots(i) for i in range(25000)]
52.4 MiB   0.0 MiB          1     return slots, no_slots
```

You might argue that this is not a fair comparison since they both store a lot of data, which skews the results. And you would indeed be right because the "bare" comparison, storing only `index` and nothing else, gives `2 MiB` versus `4.5 MiB`. But, let's be honest, if you're not going to store data, then what's the point in creating class instances? I'm not saying that `__slots__` have no purpose, but don't go overboard because the advantages are generally limited.

There is one more structure that's even more memory-efficient: the `array` module. It stores the data in pretty much the same way a bare memory array in C would do. Note that this is generally slower than lists and much less convenient to use. If you need to store large amounts of numbers, I would suggest looking at `numpy.array` or `scipy.sparse` instead.

Performance monitoring

So far, we have seen how to measure and improve both CPU and memory performance, but there is one part we have completely skipped over. Performance changes due to external factors such as growing amounts of data are very hard to predict. In real-life applications, bottlenecks aren't constant. They change all the time and code that was once extremely fast might bog down as soon as more load is applied.

Because of that, I recommend implementing a monitoring solution that tracks the performance of anything and everything over time. The big problem with performance monitoring is that you can't know what will slow down in the future and what the cause is going to be. I've even had websites slow down because of Memcached and Redis calls. These are memory-only caching servers that respond well within a millisecond, which makes slowdowns highly unlikely, until you do over 100 cache calls and the latency toward the cache server increases from 0.1 milliseconds to 2 milliseconds, and all of a sudden those 100 calls take 200 milliseconds instead of 10 milliseconds. Even though 200 milliseconds still sounds like very little, if your total page load time is generally below 100 milliseconds, that is, all of a sudden, an enormous increase and definitely noticeable.

To monitor performance and to be able to track changes over time and find the responsible components, I can personally recommend several systems for monitoring performance:

- For simple short-term (up to a few weeks) application performance tracking, the **Prometheus** monitoring system is very easy to set up and when paired with **Grafana**, you can create the prettiest charts to monitor your performance.

- If you want a more long-term performance tracking solution that scales well to large numbers of variables, you might be interested in **InfluxDB** instead. It can also be paired with Grafana for really useful interactive charting:

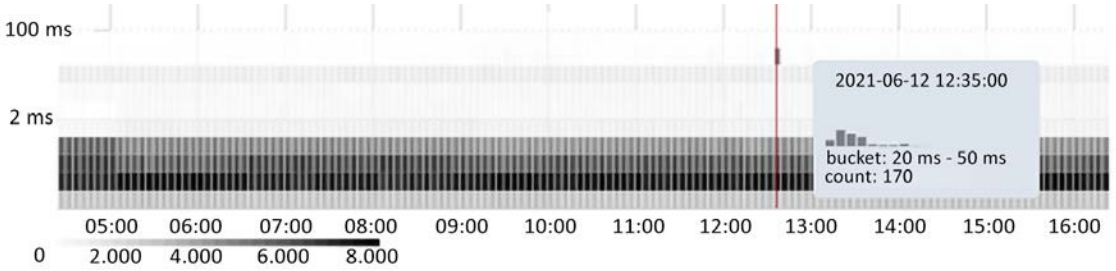


Figure 12.2: Grafana heatmap of response times

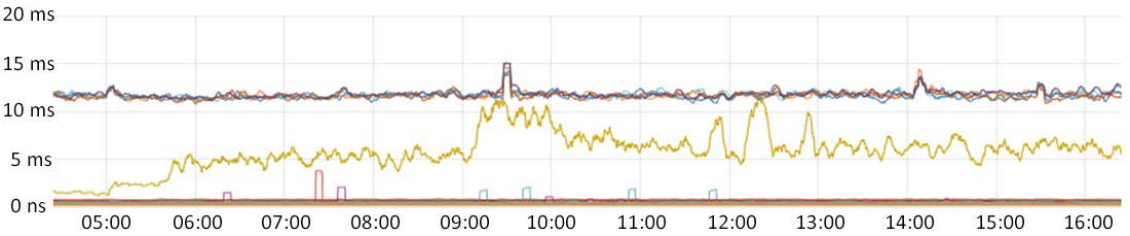


Figure 12.3: Grafana chart of request latency

To enter data into systems like these, you have several options. You can use the native APIs, but you can also use an intermediate system such as **StatsD**. The StatsD system doesn't store data itself, but it makes it really easy to fire and forget performance metrics from your system without having to worry whether the monitoring system is still up and running. Because the system commonly uses UDP to send the information, even if the monitoring server is completely down and unreachable, your application won't notice the difference.

To be able to use these, you will have to send the metrics from your application to the StatsD server. To do just that, I have written the Python-StatsD (`https://pypi.org/project/python-statsd/`) and Django-StatsD (`https://pypi.org/project/django-statsd/`) packages. These packages allow you to monitor your application from beginning to end and, in the case of Django, you will be able to monitor your performance per application or view, and within those see all of the components, such as the database, template, and caching layers. This way, you know exactly what is causing the slowdowns in your website (or application). And best of all, it is in (near) real time.

Exercises

Now that you have learned about many of the available tools for performance measuring and optimization, try and create a few useful decorators or context wrappers that will help you prevent issues:

- Try to create a decorator that monitors each run of a function and warns you if the memory usage grows each run.

- Try to create a decorator that monitors the runtime of a function and warns you if it deviates too much from the previous run. Optionally, you could make the function generate a (running) average runtime as well.
- Try to create a memory manager for your classes that warns you when more than a configured number of instances remain in memory. If you never expect more than 5 instances of a certain class, you can warn the user when that number is exceeded.

Example answers for these exercises can be found on GitHub: `https://github.com/mastering-python/exercises`. You are encouraged to submit your own solutions and learn about alternative solutions from others.

Summary

When it comes to performance, there is no holy grail, no single thing you can do to ensure peak performance in all cases. This shouldn't worry you, however, as in most cases, you will never need to tune the performance and, if you do, a single tweak could probably fix your problem. You should be able to find performance problems and memory leaks in your code now, which is what matters most, so just try to contain yourself and only tweak when it's actually needed.

Here is a quick recap of the tools in this chapter:

- Measuring CPU performance: `timeit`, `profile`/`cProfile`, and `line_profiler`
- Analyzing profiling results: SnakeViz, `pyprof2calltree`, and QCacheGrind
- Measuring memory usage: `tracemalloc`, `memory_profiler`
- Reducing memory usage and leaks: `weakref` and `gc` (garbage collector)

If you know how to use these tools, you should be able to track down and fix most performance issues in your code.

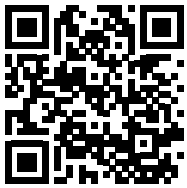
The most important takeaways from this chapter are:

- Test before you invest any effort. Making some functions faster seems like a great achievement, but it is only rarely needed.
- Choosing the correct data structure/algorithm is much more effective than any other performance optimization.
- Circular references drain the memory until the garbage collector starts cleaning.
- Slots come with several caveats, so I would recommend limited usage.

The next chapter will properly introduce us to working asynchronously using the `asyncio` module. This module makes it possible to "background" the waiting for external I/O. Instead of keeping your foreground thread busy, it can switch to a different thread when your code is waiting for endpoints such as TCP, UDP, files, and processes.

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:
`https://discord.gg/QMzJenHuJf`



13

asyncio – Multithreading without Threads

The previous chapter showed us how to track our application performance. In this chapter, we will use asynchronous programming to switch between functions whenever we have to wait for **input/output** (**I/O**) operations. This effectively fakes the effects of multiple threads or processes without introducing the overhead that comes with those solutions. In the next chapter, we will also cover multiple threads and processes for the cases where I/O is not your bottleneck or where `asyncio` is not an option.

Whenever you are dealing with external resources such as reading/writing files, interacting with APIs or databases, and other I/O operations, you can achieve great benefits from using `asyncio`. Where normally a single stalling remote connection can make your entire process hang, with `asyncio`, it will simply switch to a different part of your code.

This chapter will explain how asynchronous functions can be used in Python and how code can be restructured in such a way that it still functions, even though it doesn't follow the standard procedural coding pattern of returning the values. The potential downside is that, similar to working with multiple threads and multiple processes, the possibility exists of your code executing in an unexpected order.

The following topics are covered:

- Introduction to `asyncio`
- `asyncio` basic concepts, including coroutines, event loops, futures, and tasks
- Functions using `async def`, `async for`, `async with`, and `await`
- Parallel execution
- Examples of implementation with `asyncio`, including clients and servers
- Debugging `asyncio`

Introduction to asyncio

The `asyncio` library was created to make using asynchronous processing much easier and more predictable. It was meant as a replacement for the `asyncore` module, which has been available for a very long time (since Python 1.5 even) but was not all that usable. The `asyncio` library was officially introduced for Python 3.4 and has seen many improvements with each newer Python release since.

In a nutshell, the `asyncio` library allows you to switch to the execution of a different function whenever you need to wait for I/O operations. So instead of Python waiting for your operating system to finish reading a file for you, blocking the entire application in the process, it can do something useful in a different function in the meantime.

Backward compatibility and async/await statements

Before we continue with any examples, it is important to know how `asyncio` has changed within Python versions. Even though the `asyncio` library was only introduced in Python 3.4, a large portion of the generic syntax has been replaced in Python 3.5. Using the old Python 3.4 syntax is still possible, but an easier and therefore recommended syntax using `await` has been introduced.

This chapter will assume Python 3.7 or newer in all examples unless specified differently. If you are still running an older version, however, please look at the following sections, which illustrate how to run `asyncio` on older systems. If you have Python 3.7+, feel free to skip to the section titled *A basic example of parallel execution*.

Python 3.4

For the traditional Python 3.4 usage, a few things need to be considered:

- Functions should be declared using the `asyncio.coroutine` decorator
- Asynchronous results should be fetched using `yield from coroutine()`
- Asynchronous loops are not directly supported, but can be emulated using a `while True: yield from coroutine()`

Example:

```
import asyncio

@asyncio.coroutine
def main():
    print('Hello from main')
    yield from asyncio.sleep(1)

loop = asyncio.new_event_loop()
loop.run_until_complete(main())
loop.close()
```

Python 3.5

The Python 3.5 syntax is much more obvious than the Python 3.4 version. While the `yield from` is understandable given the origins of coroutines in earlier Python versions, it is actually the wrong name for the job. Let `yield` be used for generators and `await` be used in coroutines.

- Functions should be declared using `async def` instead of `def`
- Asynchronous results should be fetched using `await coroutine()`
- Asynchronous loops can be created using `async for ... in ...`

Example:

```
import asyncio

async def main():
    print('Hello from main')
    await asyncio.sleep(1)

loop = asyncio.new_event_loop()
loop.run_until_complete(main())
loop.close()
```

Python 3.7

Since Python 3.7 it has become slightly easier and more obvious to run `asyncio` code. If you have the luxury of a newer Python version, you can use the following to run your `async` function:

```
import asyncio

async def main():
    print('Hello from main')
    await asyncio.sleep(1)

asyncio.run(main())
```

With older Python versions, we need a fairly advanced bit of code to properly replace `asyncio.run()`, but if you are not concerned with potentially reusing existing event loops (detailed information about event loops can be found later in the chapter) and take care of shutting down your tasks yourself, you can get away with the following:

```
import asyncio

async def main():
    print('Hello from main')
    await asyncio.sleep(1)
```

```
loop = asyncio.new_event_loop()
asyncio.set_event_loop(loop)
try:
    loop.run_until_complete(main())
finally:
    # Run the loop again to finish pending tasks
    loop.run_until_complete(asyncio.sleep(0))

    asyncio.set_event_loop(None)
    loop.close()
```

Or a shorter version that is certainly not equivalent but will handle many of your test cases:

```
import asyncio

async def main():
    print('Hello from main')
    await asyncio.sleep(1)

loop = asyncio.get_event_loop()
loop.run_until_complete(main())
```

If at all possible, I would certainly recommend using `asyncio.run()`, of course. Even without `asyncio.run()`, you are likely to run into library compatibility issues with older versions of Python.

If you have to, however, you can find the source for `asyncio.run()` in the Python Git so you can implement a simplified version yourself: `https://github.com/python/cpython/blob/main/Lib/asyncio/runners.py`.

A basic example of parallel execution

When it comes to code performance, you will usually encounter one of the two following bottlenecks:

- Waiting for external I/O such as web servers, the filesystem, a database server, anything network-related, and others
- The CPU, in the case of heavy calculations

If your CPU is the bottleneck due to heavy calculations, you will need to resort to using faster algorithms, faster or more processors, or offloading the calculations to dedicated hardware such as video cards. In these cases, the `asyncio` library won't help you much.

If your code is mostly waiting for the user, the kernel, the filesystem, or external servers, `asyncio` can help you a lot while being a fairly easy solution with few side effects. As we will see in the *asyncio concepts* section, there are some caveats, however. Making existing code `asyncio`-compatible can be a lot of work.

Let's start with a very simple example to show the difference between regular and `asyncio` code when having to wait.

First, the regular Python version that executes a 1-second `sleep` two times:

```
>>> import time
>>> import asyncio

>>> def normal_sleep():
...     print('before sleep')
...     time.sleep(1)
...     print('after sleep')

>>> def normal_sleeps(n):
...     for _ in range(n):
...         normal_sleep()

# Normal execution
>>> start = time.time()
>>> normal_sleeps(2)
before sleep
after sleep
before sleep
after sleep
>>> print(f'duration: {time.time() - start:.0f}')
duration: 2
```

And now the `asyncio` version that executes a 1-second `sleep` two times:

```
>>> async def asyncio_sleep():
...     print('before sleep')
...     await asyncio.sleep(1)
...     print('after sleep')

>>> async def asyncio_sleeps(n):
...     coroutines = []
...     for _ in range(n):
...         coroutines.append(asyncio_sleep())
...
...     await asyncio.gather(*coroutines)

>>> start = time.time()
>>> asyncio.run(asyncio_sleeps(2))
```

```
before sleep
before sleep
after sleep
after sleep
>>> print(f'duration: {time.time() - start:.0f}')
duration: 1
```

As you can see, it still had to wait 1 second for the actual `sleep`, but it could run them in parallel. The `asyncio_sleep()` functions started simultaneously, as can be seen by the `before sleep` output.

Let's analyze the components used in this example:

- `async def`: This tells the Python interpreter that our function is a coroutine function instead of a regular function.
- `asyncio.sleep()`: Asynchronous version of `time.sleep()`. The big difference between these two is that `time.sleep()` will keep the Python process busy while it's sleeping, while `asyncio.sleep()` will allow switching to a different task within the event loop. This process is very similar to the workings of task switching in most operating systems.
- `asyncio.run()`: A wrapper that executes a coroutine in the default event loop. This is effectively the `asyncio` task switcher; more about this in the next section.
- `asyncio.gather()`: Wraps a sequence of awaitable objects and gathers the results for you. The wait time is configurable, as is the manner of waiting. You can choose to wait until the first result, until all results are available, or until the first exception occurs.

This immediately demonstrates a few of the caveats and pitfalls of `asyncio` code as well.

If we had accidentally used `time.sleep()` instead of `asyncio.sleep()`, the code would have taken 2 seconds to run instead and blocked the entire loop while doing so. More about this in the next section.

If we had used `await asyncio.sleep()` instead of using `await asyncio.gather()` at the end, the code would have run sequentially, and not in parallel, as you are probably looking for.

Now that we have seen a basic example of `asyncio`, we need to learn more about the internals so the limitations become more apparent.

asyncio concepts

The `asyncio` library has several basic concepts that have to be explained before venturing further into examples and uses. The example shown in the previous section actually uses several of them, but a little explanation about the how and why might still be useful.

The main concepts of `asyncio` are coroutines and event loops. Within those there are several helper classes available such as `Streams`, `Futures`, and `Processes`. The next few paragraphs will explain the basics of them so we can understand the implementations as examples in the later sections.

Coroutines, Futures, and Tasks

The `coroutine`, `asyncio.Future`, and `asyncio.Task` objects are essentially promises of a result; they return the results if they are available and can be used to cancel the execution of the promise if they have not finished processing yet. It should be noted that the creation of these objects will not guarantee that the code will be executed. The actual execution starts happening when you either `await` the results or tell an event loop to execute the promise. This is covered in the next section about event loops.

The most basic object you will encounter when using `asyncio` is the `coroutine`. The result of any regular `async def` (such as `asyncio.sleep()`) is a `coroutine` object. Once you `await` that `coroutine`, it will be executed and you will get the results.

The `asyncio.Future` and `asyncio.Task` classes can also be executed through `await`, but also allow you to register callback functions that receive the results (or exceptions) as soon as they are available. Additionally, they maintain a state variable internally, which allows an outside party to cancel the future and stop (or prevent) its execution. The API is very similar to the `concurrent.futures.Future` class, but they are not fully compatible, so make sure you do not confuse the two.

To clarify a bit further, all of these are awaitable but have different levels of abstraction:

- `coroutine`: The result of a called `async def` that has not yet been awaited. You will mostly use these.
- `asyncio.Future`: A class that represents an eventual result. It does not need to wrap a `coroutine` and the result can be set manually.
- `asyncio.Task`: An implementation of `asyncio.Future` that is meant to wrap a `coroutine` to have a convenient and consistent interface.

Usually the creation of these classes is not something you need to worry about directly; instead of creating the class yourself, the recommended way is through either `asyncio.create_task()` or `loop.create_task()`. The former actually executes `loop.create_task()` internally, but it's more convenient if you simply want to execute it on the running event loop through `asyncio.get_running_loop()` without having to specify it. If you need to extend the `Task` class for some reason, that is easily possible through the `loop.set_task_factory()` method.

Before Python 3.7, `asyncio.create_task()` was called `asyncio.ensure_future()`.

Event loops

The event loop concept is actually the most important one within `asyncio`. You might suspect that the coroutines themselves are what everything is about, but without the event loop they are useless. Event loops function as task switchers, similar to how operating systems switch between active tasks on the CPU. Even with multicore processors, there still needs to be a main process to tell the CPU which tasks to run and which need to wait or sleep for a bit. That is exactly what the event loop does: it decides which task to run.

Effectively, every time you do `await`, the event loop will look at the pending awaitables and will continue the execution of one that is currently pending. This is also where the danger of a single event loop comes in. If, for some reason, you have a slow/blocking function in your coroutine, such as accidentally using `time.sleep()` instead of `asyncio.sleep()`, it will block the entire event loop until it finishes.

In practice, this means that `await asyncio.sleep(5)` only guarantees that your code will wait *at least* 5 seconds. If, during that `await`, some other coroutine blocked the event loop for 10 seconds, the `asyncio.sleep(5)` would take at least 10 seconds.

Event loop implementations

So far we have only seen `asyncio.run()`, which uses `asyncio.get_event_loop()` internally to return the default event loop with the default event loop policy. Currently, there are two bundled event loop implementations:

- The `asyncio.SelectorEventLoop` implementation, which is used by default on Unix and Linux systems
- The `asyncio.ProactorEventLoop` implementation, which is only supported (and the default) on Windows

Internally, the `asyncio.ProactorEventLoop` implementation uses I/O completion ports, a system that is supposedly faster and more efficient than the `select` implementation of the `asyncio.SelectorEventLoop` on Windows systems.

The `asyncio.SelectorEventLoop` is based on selectors, which, since Python 3.4, are available through the `select` module in the core Python module. There are several selectors available: the traditional `selectors.SelectSelector`, which uses `select.select` internally, but also more modern solutions such as `selectors.KqueueSelector`, `selectors.EpollSelector`, and `selectors.DevpollSelector`. Even though `asyncio.SelectorEventLoop` will select the most efficient selector by default, there are cases where the most efficient one is not suitable in some way or another.

The most efficient selector is chosen by process of elimination. If the `select` module has a `kqueue` attribute, the `KqueueSelector` will be used. If `kqueue` is not available, the next best option will be chosen in the following order:

1. `KqueueSelector`: `kqueue` is an event notification interface for BSD systems. It is currently supported on FreeBSD, NetBSD, OpenBSD, DragonFly BSD, and macOS (OS X).
2. `EpollSelector`: `epoll` is the Linux kernel version of `kqueue`.
3. `DevpollSelector`: This selector uses `/dev/poll`, a system that is similar to `kqueue` and `epoll` but is supported on Solaris systems.
4. `PollSelector`: `poll()` is a system call that will call your function when an update is available. The actual implementation depends on the system.
5. `SelectSelector`: Very similar to `poll()`, but `select()` builds a bitmap for all file descriptors and walks through that list for every update, which is quite a bit less efficient than `poll()`.

In those cases, the selector event loop allows you to specify a different selector:

```
>>> import asyncio
>>> import selectors
```

```
>>> selector = selectors.SelectSelector()
>>> loop = asyncio.SelectorEventLoop(selector)
>>> asyncio.set_event_loop(loop)
```

It should be noted that the differences between these are generally too small to notice in most real-world applications. This is why I would recommend ignoring optimizations like these wherever possible, as they will most likely have very little effect and might actually cause problems if used incorrectly. The only situation I have come across where these would actually matter is when building a server that has to handle a lot of simultaneous connections. By "a lot," I refer to over 100,000 concurrent connections on a single server, which is a problem only a few people on this planet have to deal with.

If performance is important to you (and you are running Linux/OS X) I would recommend looking at `uvloop`, a really fast event loop that is built on `libuv`, an asynchronous I/O library written in C that's supported on most platforms. According to the `uvloop` benchmarks, it can make your event loop 2-4 times faster.

Event loop policies

The event loop policies are merely the constructs that store and create event loops for you and have been written with maximum flexibility in mind. The only reason I can think of for modifying the event loop policy is if you want to make specific event loops run on specific processors and/or systems, such as enabling `uvloop` only if you are running Linux or OS X. Beyond that, it offers more flexibility than most people will ever need. To make `uvloop` the default loop if installed, you could do the following:

```
import asyncio

class UvLoopPolicy(asyncio.DefaultEventLoopPolicy):
    def new_event_loop(self):
        try:
            from uvloop import Loop
            return Loop()
        except ImportError:
            return super().new_event_loop()

asyncio.set_event_loop_policy(UvLoopPolicy())
```

Beyond overriding the `new_event_loop()` to customize the creation of new event loops, you can also override how the re-use of event loops works by overriding the `get_event_loop()` and `set_event_loop()` methods. I have personally never had any use for it beyond enabling `uvloop`, however.

Event loop usage

Now that we know what event loops are, what they do, and how an event loop is selected, let's look at how they can be applied beyond `asyncio.run()`.

If you get into running your own event loops you will likely use `loop.run_forever()`, which, as you might expect, keeps running forever. Or at least until `loop.stop()` has been run. But you can also run a single task using `loop.run_until_complete()`. The latter is very useful for one-off operations, but can cause bugs in some scenarios. If you create a task from a very small/quick coroutine, odds are that the task will not have any time to run so it won't be executed until the next time you execute `loop.run_until_complete()` or `loop.run_forever()`. More about that later in this chapter, however; for now, we will assume a long-running loop using `loop.run_forever()`.

Because we have an event loop running forever now, we need to add tasks to it – this is where things get interesting. There are quite a few choices available within the default event loops:

- `call_soon()`: Add an item to the end of the (FIFO) queue so the functions will be executed in the order they were inserted.
- `call_soon_threadsafe()`: The same as `call_soon()` except for being thread-safe. The `call_soon()` method isn't thread-safe because thread safety requires the usage of the **global interpreter lock** (**GIL**), which effectively makes your program single-threaded at the moment of thread safety. *Chapter 14, Multiprocessing – When a Single CPU Core is Not Enough,* explains both the GIL and thread safety in detail.
- `call_later()`: Call the function after the given number of seconds; if two jobs would run at the same time, they will run in an undefined order. If the undefined order is an issue, you can also opt to use `asyncio.gather()` or increase the `delay` parameter for one of the two tasks slightly. Note that the `delay` is a minimum – if the event loop is locked/busy, it could run later.
- `call_at()`: Call a function at a specific time related to the output of `loop.time()`, which is the number of seconds since the `loop` started. So, if the current value of `loop.time()` is `90` (which means the `loop` started running `90` seconds ago), then you could run `loop.call_at(95, ...)` to run after `5` seconds.

All of these functions return `asyncio.Handle` objects. These objects allow the cancelation of the task if it hasn't been executed yet through the `handle.cancel()` function. Be careful with canceling from other threads, however, as cancelation is not thread-safe either. To execute it in a thread-safe way, we have to execute the cancelation function as a task as well: `loop.call_soon_threadsafe(handle.cancel)`.

Example usage:

```
>>> import time
>>> import asyncio

>>> def printer(name):
...     print(f'Started {name} at {loop.time() - offset:.1f}')
...     time.sleep(0.2)
...     print(f'Finished {name} at {loop.time() - offset:.1f}')

>>> loop = asyncio.new_event_loop()
>>> _ = loop.call_at(loop.time() + .2, printer, 'call_at')
>>> _ = loop.call_later(.1, printer, 'call_later')
```

```
>>> _ = loop.call_soon(printer, 'call_soon')
>>> _ = loop.call_soon_threadsafe(printer, 'call_soon_threadsafe')

>>> # Make sure we stop after a second
>>> _ = loop.call_later(1, loop.stop)

# Store the offset because the loop requires time to start
>>> offset = loop.time()

>>> loop.run_forever()
Started call_soon at 0.0
Finished call_soon at 0.2
Started call_soon_threadsafe at 0.2
Finished call_soon_threadsafe at 0.4
Started call_later at 0.4
Finished call_later at 0.6
Started call_at at 0.6
Finished call_at at 0.8
```

You might be wondering why we are using `time.sleep()` instead of `asyncio.sleep()` here. That is an intentional choice to show how none of these functions offer any guarantee of when the function is executed if the `loop` is somehow blocked. Even though we specified a `0.1` second delay for the `loop.call_later()` call, it took `0.4` seconds to actually start. If we had used `asyncio.sleep()` instead, the functions would have run in parallel.

The `call_soon()`, `call_soon_threadsafe()`, and `call_later()` functions are all just wrappers for `call_at()`. In the case of `call_soon()`, it just wraps `call_later()` with a delay of `0`, and `call_at()` is simply a `call_soon()` with `asyncio.time()` added to the delay.

Depending on the type of event loop, there are actually many other methods for creating connections, file handlers, and more, similar to `asyncio.create_task()`. Those will be explained with examples in the later sections, since they have less to do with the event loop and are more about programming with coroutines.

Executors

Since even a simple `time.sleep()` can completely block your event loop, you might be wondering what the practical use for `asyncio` is. It would mean you have to rewrite your entire code base to be `asyncio`-compatible, right? Ideally that would be the best solution, but we can work around this limitation by executing sync code from `asyncio` code using executors. An `Executor` creates the other type of `Future` (`concurrent.futures.Future` as opposed to `asyncio.Future`) we talked about earlier, and runs your code in a separate thread or process to provide an `asyncio` interface to synchronous code.

Here is a basic example of the synchronous `time.sleep()` executed through an executor to make it asynchronous:

```
>>> import time
>>> import asyncio

>>> def executor_sleep():
...     print('before sleep')
...     time.sleep(1)
...     print('after sleep')

>>> async def executor_sleeps(n):
...     loop = asyncio.get_running_loop()
...     futures = []
...     for _ in range(n):
...         future = loop.run_in_executor(None, executor_sleep)
...         futures.append(future)
...
...     await asyncio.gather(*futures)

>>> start = time.time()
>>> asyncio.run(executor_sleeps(2))
before sleep
before sleep
after sleep
after sleep
>>> print(f'duration: {time.time() - start:.0f}')
duration: 1
```

So, instead of running `executor_sleep()` directly, we are creating a future through `loop.run_in_executor()`. This makes `asyncio` execute this function through the default executor, which is normally a `concurrent.futures.ThreadPoolExecutor`, and return the results when it's done. You do need to be aware of thread safety because it is handled in a separate thread, but more about that topic in the next chapter.

For operations that are blocking but not CPU-bound (in other words, no heavy calculations), the default threading-based executor will work great. For CPU-bound operations it will not help you, since the operations will still be limited to a single CPU core. For those scenarios, we can use `concurrent.futures.ProcessPoolExecutor()`:

```
import time
import asyncio
import concurrent.futures

def executor_sleep():
    print('before sleep')
```

```
    time.sleep(1)
    print('after sleep')

async def executor_sleeps(n):
    loop = asyncio.get_running_loop()
    futures = []
    with concurrent.futures.ProcessPoolExecutor() as pool:
        for _ in range(n):
            future = loop.run_in_executor(pool, executor_sleep)
            futures.append(future)

        await asyncio.gather(*futures)

if __name__ == '__main__':
    start = time.time()
    asyncio.run(executor_sleeps(2))
    print(f'duration: {time.time() - start:.0f}')
```

While this example looks nearly identical to the previous example, the internal mechanism is quite different and the use of multiple Python processes instead of multiple threads comes with several caveats:

- Memory cannot easily be shared between processes. This means that anything you want to pass as an argument and anything that you need to return has to be supported by the `pickle` process so Python can send the data, through the network, to the other Python process. This is explained in detail in *Chapter 14*.
- The main script has to be run from an `if __name__ == '__main__'` block, otherwise the executor would end up in an infinite loop spawning itself.
- Most resources cannot be shared between processes. This is similar to not being able to share memory, but it goes beyond that. If you have a database connection in your main process, that connection cannot be used from the process so it will need to have its own connections.
- Killing/exiting the process can be more difficult since killing the main process is not always a guarantee of killing the child processes.
- Depending on your operating system, every new process will use its own memory, resulting in greatly increased memory usage.
- Creating a new process is generally a far heavier operation than creating a new thread, so you have a lot more overhead.
- Synchronization between processes is much slower than with threads.

All of these reasons definitely shouldn't prevent you from using `ProcessPoolExecutor`, but you should always ask yourself if you actually need it. It can be an amazing solution if you need to run many heavy calculations in parallel. If at all possible, I would recommend using functional programming with `ProcessPoolExecutor`. *Chapter 14*, *Multiprocessing – When a Single CPU Core Is Not Enough*, covers multiprocessing in much more detail.

Now that we have a basic grasp of `asyncio`, it is time to continue with some examples of where `asyncio` can be useful.

Asynchronous examples

One of the most common reasons for stalling scripts and applications is the usage of remote resources, where *remote* means any interaction with the network, filesystem, or other resources. With `asyncio`, at least a large portion of that is easily fixable. Fetching multiple remote resources and serving to multiple clients is quite a bit easier and more lightweight than it used to be. While both multithreading and multiprocessing can be used for these cases as well, `asyncio` is a much lighter alternative that is actually easier to manage in many cases.

The next few sections show a few examples of how to implement certain operations using `asyncio`.

Before you start implementing your own code and copying the examples here, I would recommend doing a quick search on the web for whichever library you are looking for and seeing if there is an `asyncio` version available.

In general, looking for "asyncio <protocol>" will give you great results. Alternatively, many libraries use the `aio` prefix for the library name, such as `aiohttp`, so that can help your search as well.

Processes

So far, we have simply executed simple `async` functions within Python such as `asyncio.sleep()`, but some things are a tad more difficult to run asynchronously. For example, let's assume we have some long-running external application that we wish to run without blocking our main thread completely.

The options for running external processes in a non-blocking mode are generally:

- Threading
- Multiprocessing
- Polling (periodically checking) for output

Both threading and multiprocessing are covered in *Chapter 14*.

Without resorting to more complex solutions such as threading and multiprocessing, which introduce variable synchronization issues, we only have polling remaining. With polling, we check if there is new output at an interval, which can slow down your results by as much as the poll interval. That is, if your poll interval is 1 second and the process generates output 0.1 seconds after the last poll, the next 0.9 seconds are wasted waiting. To alleviate this, you could reduce the poll interval, of course, but with a lower poll interval more time is wasted checking to see if there are results.

With `asyncio`, we can have the advantages of the polling method without the time wasted between the poll intervals. Using `asyncio.create_subprocess_shell` and `asyncio.create_subprocess_exec`, we can `await` output just like other coroutines. The usage of the class is very similar to `subprocess.run` except that the functions have been made asynchronous, resulting in the removal of the poll function, of course.

The examples below expect the `sleep` command to be available in your environment. On all Unix/Linux/BSD systems, this is the case by default. On Windows, it is not available by default, but can be installed easily. The `timeout` command can be used as an alternative.

If you do wish to use `sleep` and other Unix tools, the easiest method I have found is to install Git for Windows and let it install the **optional Unix tools**:

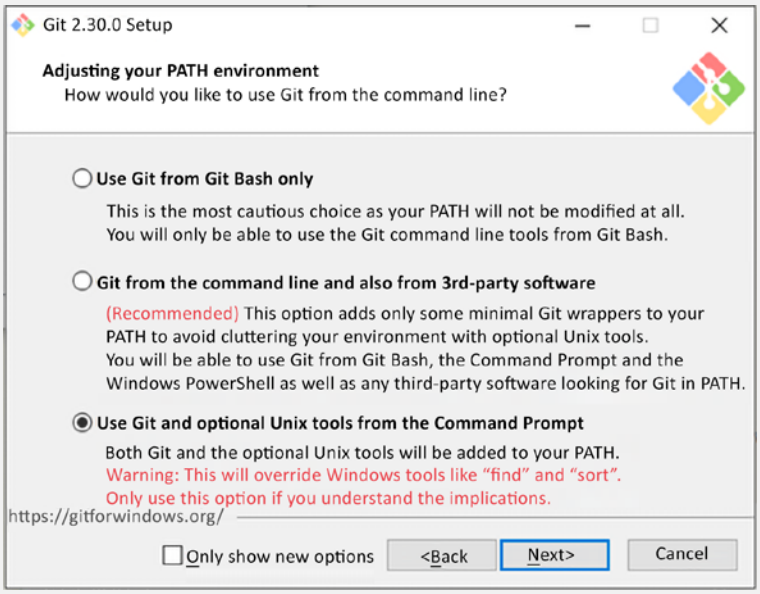


Figure 13.1: Git for Windows installer

First, let's look at the traditional sequential version of a script that runs external processes (in this case the `sleep` command) through the `subprocess` module:

```
>>> import time
>>> import subprocess

>>> def subprocess_sleep():
...     print(f'Started sleep at: {time.time() - start:.1f}')
...     process = subprocess.Popen(['sleep', '0.1'])
...     process.wait()
...     print(f'Finished sleep at: {time.time() - start:.1f}')

>>> start = time.time()
```

After the first `print()`, we use `subprocess.Popen()` to run the `sleep` command with argument `0.1` so it will sleep for `0.1` seconds. As opposed to `subprocess.run()`, which blocks your Python process and waits until the external process has finished running, `subprocess.Popen()` creates and starts the process and returns a reference to the running process, but it won't automatically wait for the output.

This allows us to explicitly call `process.wait()` to wait or poll for the results, as we will see in the next example. Internally, `subprocess.run()` is actually a convenient shortcut for a common use case of `subprocess.Popen()`.

When running the code, we get the following output, as you would expect:

```
>>> for _ in range(2):
...     subprocess_sleep()
Started sleep at: 0.0
Finished sleep at: 0.1
Started sleep at: 0.1
Finished sleep at: 0.2
```

Since everything is executed sequentially, it takes two times the 0.1 seconds that the `sleep` command is sleeping for. This is, of course, the worst-case scenario: it completely blocks your Python process while it is running.

Instead of waiting for the `sleep` command immediately after running, we are now going to start all processes in parallel and only wait for the results once they have all started in the background:

```
>>> import time
>>> import subprocess

>>> def subprocess_sleep():
...     print(f'Started sleep at: {time.time() - start:.1f}')
...     return subprocess.Popen(['sleep', '0.1'])

>>> start = time.time()
```

As you can see, we are returning the process by returning `subprocess.Popen()` without executing `process.wait()`.

Now we start all processes immediately and only wait for output after they have all started:

```
>>> processes = []
>>> for _ in range(2):
...     processes.append(subprocess_sleep())
Started sleep at: 0.0
Started sleep at: 0.0
```

The processes should be running in the background now, so let's wait for the results:

```
>>> for process in processes:
...     returncode = process.wait()
...     print(f'Finished sleep at: {time.time() - start:.1f}')
Finished sleep at: 0.1
Finished sleep at: 0.1
```

While that looks a lot better in terms of runtime, it still blocks the main process when we run `process.wait()`. It also required restructuring in such a way that the teardown (the `Finished` print statement) is not in the same block as the start process, as was the case with the earlier example. This means that if something were to go wrong with your application, you would manually need to keep track of which process was failing, which is a bit inconvenient.

With the `asyncio` version, we can once again go back to processing everything related to the `sleep` command in a single function, very similar to the first example with `subprocess.Popen()`:

```
>>> import time
>>> import asyncio

>>> async def async_process_sleep():
...     print(f'Started sleep at: {time.time() - start:.1f}')
...     process = await asyncio.create_subprocess_exec('sleep', '0.1')
...     await process.wait()
...     print(f'Finished sleep at: {time.time() - start:.1f}')

>>> async def main():
...     coroutines = []
...     for _ in range(2):
...         coroutines.append(async_process_sleep())
...     await asyncio.gather(*coroutines)

>>> start = time.time()
>>> asyncio.run(main())
Started sleep at: 0.0
Started sleep at: 0.0
Finished sleep at: 0.1
Finished sleep at: 0.1
```

As you can see, it is trivial to run multiple applications at the same time this way. The syntax is essentially the same as it would be with `subprocess` without having to block or poll.

If you are running this from a long-running `asyncio` event loop and you don't need to capture the results, you could skip the entire `asyncio.gather()` step and use `asyncio.create_task(async_process_sleep())` instead.

Interactive processes

Starting processes is the easy part; the more difficult part is interactive input and output with processes. The `asyncio` module has several measures to make that part easier, but it can still be difficult when actually working with the results.

Here's an example of calling the Python interpreter as an external subprocess, executing some code, and exiting again in a simple one-off fashion:

```
>>> import time
>>> import asyncio

>>> async def run_python_script(script):
...     print(f'Executing: {script!r}')
...     process = await asyncio.create_subprocess_exec(
...         'python3',
...         stdout=asyncio.subprocess.PIPE,
...         stdin=asyncio.subprocess.PIPE,
...     )
...     stdout, stderr = await process.communicate(script)
...     print(f'stdout: {stdout!r}')

>>> asyncio.run(run_python_script(b'print(2 ** 20)'))
Executing: b'print(2 ** 20)'
stdout: b'1048576\n'
```

In this case, we added a pipe to `stdout` (standard output) and `stdin` (standard input) so we can read from `stdout` and write to `stdin` manually. After the process has started, we can use `process.communicate()` to write to `stdin`, and `process.communicate()` will automatically read all output from `stdout` and `stderr` if they are available. Since we did not declare what `stderr` is supposed to be, Python will automatically send all `process.stderr` output to `sys.stderr` for us, so we can ignore `stderr` here as it will be `None`.

Now the actual challenge comes when we want interactive subprocesses with two-way communication through `stdin`/`stdout`/`stderr` that keep on running for a longer time. That is also possible of course, but it can be hard to avoid deadlocks in situations where both sides are waiting for input. Here's a very simple example of a Python subprocess that does effectively the same as `communicate()` above, but manually, to give you granular control over the input and output of the process:

```
>>> import asyncio

>>> async def run_script():
...     process = await asyncio.create_subprocess_exec(
...         'python3',
...         stdout=asyncio.subprocess.PIPE,
...         stdin=asyncio.subprocess.PIPE,
...     )
...
...     # Write a simple Python script to the interpreter
```

```
...     process.stdin.write(b'print("Hi~")')
...
...     # Make sure the stdin is flushed asynchronously
...     await process.stdin.drain()
...     # And send the end of file so the Python interpreter will
...     # start processing the input. Without this the process will
...     # stall forever.
...     process.stdin.write_eof()
...
...     # Fetch the lines from the stdout asynchronously
...     async for line in process.stdout:
...         # Decode the output from bytes and strip the whitespace
...         # (newline) at the right
...         print('stdout:', line.rstrip())
...
...     # Wait for the process to exit
...     await process.wait()

>>> asyncio.run(run_script())
stdout: b'Hi~'
```

The code might appear largely as you would expect, but there are a few parts that are non-obvious to use, yet required to function. While the creation of the subprocess is identical to the previous example, the writing of the code to `stdin` is slightly different.

Instead of using `process.communicate()`, we now write directly to the `process.stdin` pipe. When you run `process.stdin.write()`, Python will *try* to write to the stream, but might not be able to because the process hasn't started running yet. Because of that, we need to manually flush these buffers by using `process.stdin.drain()`. Once that is done, we send an end-of-file (`EOF`) character so the Python subprocess knows that no more input is coming.

Once the input is written, we need to read the output from the Python subprocess. We could use `process.stdout.readline()` in a loop for this, but similar to how we can do `for line in open(filename)`, we can also read `process.stdout` line by line using an `async for` loop until the stream is closed.

If at all possible, I would recommend abstaining from using `stdin` to send data to subprocesses and instead use some network, pipe, or file communication instead. As we will see in the next section covering an echo client and server, those are much more convenient to handle and less prone to deadlocks.

Echo client and server

The most basic kind of server you can get is an "echo" server, which sends all messages received back. Since we can run multiple tasks in parallel with `asyncio`, we can run both the server and the client from the same script here. Splitting them into two processes is also possible, of course.

Creating a basic client and server is easy to do:

```
>>> import asyncio

>>> HOST = '127.0.0.1'
>>> PORT = 1234

>>> async def echo_client(message):
...     # Open the connection to the server
...     reader, writer = await asyncio.open_connection(HOST, PORT)
...
...     print(f'Client sending {message!r}')
...     writer.write(message)
...
...     # We need to drain and write the EOF to stop sending
...     writer.write_eof()
...     await writer.drain()
...
...     async for line in reader:
...         print(f'Client received: {line!r}')
...
...     writer.close()

>>> async def echo(reader, writer):
...     # Read all lines from the reader and send them back
...     async for line in reader:
...         print(f'Server received: {line!r}')
...         writer.write(line)
...         await writer.drain()
...
...     writer.close()

>>> async def echo_server():
...     # Create a TCP server that listens on 'HOST'/'PORT' and
...     # calls 'echo' when a client connects.
...     server = await asyncio.start_server(echo, HOST, PORT)
...
...     # Start listening
...     async with server:
...         await server.serve_forever()
```

```
>>> async def main():
...     # Create and run the echo server
...     server_task = asyncio.create_task(echo_server())
...
...     # Wait a little for the server to start
...     await asyncio.sleep(0.01)
...
...     # Create a client and send the message
...     await echo_client(b'test message')
...
...     # Kill the server
...     server_task.cancel()

>>> asyncio.run(main())
Client sending b'test message'
Server received: b'test message'
Client received: b'test message'
```

In this example, we can see that we sent the server to the background using `asyncio.create_task()`. After that, we have to wait just a tiny amount of time for the background task to start working, which we are doing using `asyncio.sleep()`. The sleep time of `0.01` was chosen arbitrarily (and `0.001` is probably enough as well), but it should be enough for most systems to communicate with the kernel to create a listening socket. Once the server is running, we start our client to send a message and wait for the response.

Naturally, this example could have been written in many different ways. Instead of `async for`, you could use `reader.readline()` to read until the next newline, or you could use `reader.read(number_of_bytes)` to read a specific number of characters. It all depends on the protocol you wish to write. In the case of the HTTP/1.1 protocol, the server expects a `Connection: close`; in the case of the SMTP protocol, a `QUIT` message should be sent. In our case, we use the `EOF` character as an indicator.

Asynchronous file operations

One of the operations you would prefer to be asynchronous is file operations. Even though storage devices have become much faster over the years, you are not always working on fast local storage. If you want to write to a network drive over a Wi-Fi connection, for example, you can experience quite a lot of latency. By using `asyncio`, you can make sure this won't stall your entire interpreter.

Unfortunately, there is currently no easy way to do file operations through `asyncio` in a cross-platform way because most operating systems have no (scalable) asynchronous file operations support. Luckily, someone created a workaround for this issue. The `aiofiles` library uses the `threading` library internally to give you an `asyncio` interface to file operations. While you could easily use an `Executor` to handle the file operations for you, the `aiofiles` library is a very convenient wrapper that I recommend using.

First, install the library:

```
$ pip3 install aiofiles
```

Now we can use `aiofiles` to open, read, and write files in a non-blocking manner through `asyncio`:

```
>>> import asyncio
>>> import aiofiles

>>> async def main():
...     async with aiofiles.open('aiofiles.txt', 'w') as fh:
...         await fh.write('Writing to file')
...
...     async with aiofiles.open('aiofiles.txt', 'r') as fh:
...         async for line in fh:
...             print(line)

>>> asyncio.run(main())
Writing to file
```

The usage of `aiofiles` is very similar to a regular `open()` call, except with the `async` prefix in all cases.

Creating async generators to support async for

In the earlier examples, you might have wondered how to support `async for` statements. Essentially it is very easy to do so; instead of a regular generator that you could create with the `__iter__` and `__next__` magic functions in a class, you would now use `__aiter__` and `__anext__` instead:

```
>>> import asyncio

>>> class AsyncGenerator:
...     def __init__(self, iterable):
...         self.iterable = iterable
...
...     async def __aiter__(self):
...         for item in self.iterable:
...             yield item

>>> async def main():
...     async_generator = AsyncGenerator([4, 2])
...
...     async for item in async_generator:
...         print(f'Got item: {item}')
```

```
>>> asyncio.run(main())
Got item: 4
Got item: 2
```

Effectively, the code is identical to regular generators and `with` statements, but you can also access `asyncio` code from the functions. There is really nothing special about these methods except that they need the `async` prefix and the `a` in the name, so you get `__aiter__` instead of `__iter__`.

Creating async context managers to support async with

Similar to the `async` generator, we can also create an `async` context manager. Instead of the `__iter__` method, we now have to replace the `__enter__` and `__exit__` methods with `__aenter__` and `__aexit__`, respectively.

Effectively the code is identical to a `with` statement, but you can also access `asyncio` code from the functions:

```
>>> import asyncio

>>> class AsyncContextManager:
...     async def __aenter__(self):
...         print('Hi :)')
...
...     async def __aexit__(self, exc_type, exc_value, traceback):
...         print('Bye :(')

>>> async def main():
...     async_context_manager = AsyncContextManager()
...
...     print('Before with')
...     async with async_context_manager:
...         print('During with')
...     print('After with')

>>> asyncio.run(main())
Before with
Hi :)
During with
Bye :(
After with
```

Similar to the `async` generator, there really is nothing special about these methods. But the `async` context manager in particular is very useful for setup/teardown methods, as we will see in the next section.

Asynchronous constructors and destructors

At some point, you will probably want to run some asynchronous code from your constructors and/or destructors, perhaps to initialize a database connection or other type of network connection. Unfortunately, that is not really possible.

Naturally, using `__await__` or metaclasses, you could hack around this for your constructor. And with an `asyncio.run(...)` you could do something similar for your destructor. Neither is really a great solution though – I would suggest restructuring your code instead.

Depending on the scenario I would suggest using either:

- Context managers to properly enter/exit using an `async with` statement
- A Factory pattern where an `async def` generates and initializes the class for you, together with an `async def close()` as an async destructor

We have already seen the context manager in the previous section, and that would be the method I would recommend in most cases, such as creating database connections and/or transactions, since you cannot accidentally forget to run the teardown using that.

The Factory design pattern uses a function to facilitate the creation of an object. In this case, that means instead of doing `instance = SomeClass(...)`, you would have `instance = await SomeClass.create(...)` so you can have an asynchronous initialization method.

But a Factory pattern with an explicit create and close method is, of course, a good possibility too:

```
>>> import asyncio

>>> class SomeClass:
...     def __init__(self, *args, **kwargs):
...         print('Sync init')
...
...     async def init(self, *args, **kwargs):
...         print('Async init')
...
...     @classmethod
...     async def create(cls, *args, **kwargs):
...         # Create an instance of 'SomeClass' which calls the
...         # sync init: 'SomeClass.__init__(*args, **kwargs)'
...         self = cls(*args, **kwargs)
...         # Now we can call the async init:
...         await self.init(*args, **kwargs)
...         return self
...
```

```
...     async def close(self):
...         print('Async destructor')
...
...     def __del__(self):
...         print('Sync destructor')

>>> async def main():
...     # Note that we use 'SomeClass.create()' instead of
...     # 'SomeClass()' so we also run 'SomeClass().init()'
...     some_class = await SomeClass.create()
...     print('Using the class here')
...     await some_class.close()
...     del some_class

>>> asyncio.run(main())
Sync init
Async init
Using the class here
Async destructor
Sync destructor
```

With the order of operations as shown before, you can properly create and tear down an `asyncio` class that way. As a failsafe (explicitly calling `close()` is always the better solution), you can add an `async` destructor to your `__del__` by calling the loop.

For the next example, we will use the `asyncpg` library, so make sure to install it first:

```
$ pip3 install asyncpg
```

Now, an `asyncio` database connection to PostgreSQL could be implemented like this:

```
import typing
import asyncio
import asyncpg

class AsyncPg:
    _connection: typing.Optional[asyncpg.Connection]

    async def init(self):
        self._connection = asyncpg.connect(...)

    async def close(self):
        await self._connection.close()
```

```
    def __del__(self):
        if self._connection:
            loop = asyncio.get_event_loop()
            if loop.is_running():
                loop.create_task(self.close())
            else:
                loop.run_until_complete(self.close())

            self._connection = None
```

You could also create a registry to easily close all classes that were created so you can't forget to do so on exit. But if possible, I would still recommend the context manager-style solution. You could also make a convenient shortcut using a decorator by creating an `async` version of `contextlib.ContextDecorator`.

Next up, we will look at how to debug `asyncio` code and how to catch common mistakes.

Debugging asyncio

The `asyncio` module has a few special provisions to make debugging somewhat easier. Given the asynchronous nature of functions within `asyncio`, this is a very welcome feat. While the debugging of multithreaded/multiprocessing functions or classes can be difficult – since concurrent classes can easily change environment variables in parallel – with `asyncio`, it's just as difficult, if not more, because `asyncio` background tasks run in the stack of the event loop, not your own stack.

If you wish to skip this part of the chapter, I urge you to at least read the section on *Exiting before all tasks are done*. That covers a **huge** pitfall with `asyncio`.

The first and most obvious way of debugging `asyncio` is to use the event loop debug mode. We have several options for enabling the debug mode:

- Set the `PYTHONASYNCIODEBUG` environment variable to `True`
- Enable the Python development mode using the `PYTHONDEVMODE` environment variable or by executing Python with the `-X dev` command-line option
- Pass the `debug=True` argument to `asyncio.run()`
- Call `loop.set_debug()`

Of these methods, I recommend using the `PYTHONASYNCIODEBUG` or `PYTHONDEVMODE` environment variables because these are applied very early and can therefore catch several errors that the others might miss. We will see an example of that in the next section about forgotten `await` statements.

Note about setting environment variables

Within most Linux/Unix/Mac shell sessions, environment variables can be set using `variable=value` as a prefix:

```
SOME_ENVIRONMENT_VARIABLE=value python3 script.py
```

Also, environment variables can be configured for the current shell (when using ZSH or Bash) session using `export`:

```
export SOME_ENVIRONMENT_VARIABLE=value
```

The current value can be fetched using the following line:

```
echo $SOME_ENVIRONMENT_VARIABLE
```

On Windows, you can configure an environment variable for your local shell session using the `set` command:

```
set SOME_ENVIRONMENT_VARIABLE=value
```

The current value can be fetched using this line:

```
set SOME_ENVIRONMENT_VARIABLE
```

When the debug mode is enabled, the `asyncio` module will check a few common `asyncio` mistakes and issues. Specifically:

- Coroutines that have not been yielded will raise an exception.
- Calling coroutines from the "wrong" thread raises an exception. This can occur if you have code running in different threads from the thread running the current event loop. This is effectively a case of thread safety, which is covered in *Chapter 14*.
- The execution time of the selector will be logged.
- Slow coroutines (more than 100 ms) will be logged. This timeout can be modified through `loop.slow_callback_duration`.
- Warnings will be raised when resources are not closed properly.
- Tasks that were destroyed before execution will be logged.

Let's showcase a few of these mistakes.

Forgetting to await a coroutine

This is probably the most common `asyncio` bug and it has bitten me many, many times. It is so easy to do `some_coroutine()` instead of `await some_coroutine()` and you usually find out when it's already too late.

Luckily, Python can help us with this one, so let's look at what happens when you forget to `await` a coroutine with `PYTHONASYNCIODEBUG` set to `1`:

```
async def printer():
    print('This is a coroutine')

printer()
```

This results in an error for the `printer` coroutine, which we forgot to `await`:

```
$ PYTHONASYNCIODEBUG=1 python3 T_13_forgot_await.py
T_13_forgot_await.py:5: RuntimeWarning: coroutine 'printer' was never awaited
  printer()
RuntimeWarning: Enable tracemalloc to get the object allocation traceback
```

Note that this will only occur when the event loop has been closed. The event loop can't know if you intended to execute the coroutine at a later moment, so this can still be difficult to debug.

This is also one of the cases where using the `PYTHONASYNCIODEBUG` environment variable instead of `loop.set_debug(True)` can make a difference. Think about a scenario where you have multiple event loops and forget to enable debug mode for all of them, or where a forgotten coroutine is created before debug mode is enabled, which means it will not be tracked.

Slow blocking functions

Not considering that a function might be slow and blocking your loop is easy to do. If it is somewhat slow but not slow enough that you'll notice, you will probably never find out about it unless you enable the debug mode. Let's look at how the debug mode helps us here:

```
import time
import asyncio

async def main():
    # Oh no... a synchronous sleep from asyncio code
    time.sleep(0.2)

asyncio.run(main(), debug=True)
```

In this case, we "accidentally" used `time.sleep()` instead of `asyncio.sleep()`.

For these issues, `debug=True` works great, but it never hurts to use `PYTHONASYNCIODEBUG=1` when developing:

```
$ PYTHONASYNCIODEBUG=1 python3 T_14_slow_blocking_code.py
Executing <Task finished ...> took 0.204 seconds
```

As we expected, we get a warning with this slow function.

The default warning threshold is set to 100 ms and we are sleeping for 200 ms, so it is reported. The threshold can be changed through `loop.slow_callback_duration=<seconds>` if needed. This could be useful if you are working on a slower system such as a Raspberry Pi, or if you want to look for slow code.

Forgetting to check the results or exiting early

A common way to write code with `asyncio` is to use fire-and-forget with `asyncio.create_task()` without storing the resulting future. While this is not inherently wrong, if an exception unexpectedly occurs in your code, it can be very difficult to find the culprit without the debug mode enabled.

To illustrate, we are going to use the following uncaught exception and execute it both with and without debug mode:

```
import asyncio

async def throw_exception():
    raise RuntimeError()

async def main():
    # Ignoring an exception from an async def
    asyncio.create_task(throw_exception())

asyncio.run(main())
```

If we execute this without debug mode, we get the following output:

```
$ python3 T_15_forgotten_exception.py
Task exception was never retrieved
future: <Task finished ... at T_15_forgotten_exception.py:4>
exception=RuntimeError()>
Traceback (most recent call last):
  File "T_15_forgotten_exception.py", line 5, in throw_exception
    raise RuntimeError()
RuntimeError
```

While this does nicely show us where the exception occurred and what exception occurred, it does not show us who or what created the coroutine.

Now if we repeat the same with debug mode enabled, we get this:

```
$ PYTHONASYNCIODEBUG=1 python3 T_15_forgotten_exception.py
Task exception was never retrieved
future: <Task finished ... at T_15_forgotten_exception.py:4>
exception=RuntimeError() created at asyncio/tasks.py:361>
source_traceback: Object created at (most recent call last):
```

```
  File "T_15_forgotten_exception.py", line 13, in <module>
    asyncio.run(main())
  File "asyncio/runners.py", line 44, in run
    return loop.run_until_complete(main)
  File "asyncio/base_events.py", line 629, in run_until_complete
    self.run_forever()
  File "asyncio/base_events.py", line 596, in run_forever
    self._run_once()
  File "asyncio/base_events.py", line 1882, in _run_once
    handle._run()
  File "asyncio/events.py", line 80, in _run
    self._context.run(self._callback, *self._args)
  File "T_15_forgotten_exception.py", line 10, in main
    asyncio.create_task(throw_exception())
  File "asyncio/tasks.py", line 361, in create_task
    task = loop.create_task(coro)
Traceback (most recent call last):
  File "T_15_forgotten_exception.py", line 5, in throw_exception
    raise RuntimeError()
RuntimeError
```

This might still be a bit hard to read, but now we see that the exception originated from `asyncio.create_task(throw_exception())` and we can even see the `asyncio.run(main())` call.

For a slightly larger code base, this can be essential in tracing the source of your exceptions.

Exiting before all tasks are done

Pay attention here, because this issue is extremely subtle but can have huge consequences if you don't notice it.

Similar to forgetting to fetch the results, when you create a task while the loop is already tearing down, the task will *not* always run. In some cases, it does not have the chance to run and you most likely won't notice it.

Take a look at this example where we have a task spawning another task:

```
import asyncio

async def sub_printer():
    print('Hi from the sub-printer')

async def printer():
    print('Before creating the sub-printer task')
    asyncio.create_task(sub_printer())
```

```
    print('After creating the sub-printer task')

async def main():
    asyncio.create_task(printer())

asyncio.run(main())
```

In this case, even the debug mode cannot help you. To illustrate, let's look at what happens when we call this with debug mode enabled:

```
$ PYTHONASYNCIODEBUG=1 python3 T_16_early_exit.py
Before creating the sub-printer task
After creating the sub-printer task
```

The call to `sub_printer()` seems to have disappeared. It really hasn't, but we did not explicitly wait for it to finish so it never got a chance to run.

The **best** solution by far is to keep track of all futures created by `asyncio.create_task()` and do an `await asyncio.gather(*futures)` at the end of your `main()` function. But this is not always an option – you might not have access to the futures created by other libraries, or the futures might be created in a scope you cannot easily access. So what can you do?

As a very simple workaround, you can simply wait at the end of your `main()` function:

```
import asyncio

async def sub_printer():
    print('Hi from the sub-printer')

async def printer():
    print('Before creating the sub-printer task')
    asyncio.create_task(sub_printer())
    print('After creating the sub-printer task')

async def main():
    asyncio.create_task(printer())
    await asyncio.sleep(0.1)

asyncio.run(main())
```

For this case, adding that little bit of sleep time works fine:

```
$ python3 T_17_wait_for_exit.py
Before creating the sub-printer task
After creating the sub-printer task
Hi from the sub-printer
```

But this only does the trick if your task is fast enough or if you increase the sleep time. If we had a database teardown method that takes several seconds, we could still end up with an issue. As a very crude workaround, it can be useful to add this to your code since it will be more obvious when you're missing a task.

A slightly better solution is to ask `asyncio` what tasks are still running and wait until they have finished. The drawback of this method is that if you have a task that runs forever (in other words, `while True`), you will wait forever for the script to exit.

So let's look at how we could implement a feature like this, with a fixed timeout so we won't wait forever:

```
import asyncio

async def sub_printer():
    print('Hi from the sub-printer')

async def printer():
    print('Before creating the sub-printer task')
    asyncio.create_task(sub_printer())
    print('After creating the sub-printer task')

async def main():
    asyncio.create_task(printer())
    await shutdown()

async def shutdown(timeout=5):
    tasks = []
    # Collect all tasks from 'asyncio'
    for task in asyncio.all_tasks():
        # Make sure we skip our current task so we don't loop
        if task is not asyncio.current_task():
            tasks.append(task)

    for future in asyncio.as_completed(tasks, timeout=timeout):
        await future

asyncio.run(main())
```

This time, we have added a `shutdown()` method that fetches all tasks from `asyncio` using `asyncio.all_tasks()`. After collecting the tasks, we need to make sure that we don't get our current task because that would result in a chicken-and-egg problem. The `shutdown()` task will never exit while waiting for the `shutdown()` task to finish.

When all tasks are gathered, we use `asyncio.as_completed()` to wait for them to finish and return after. If the waiting takes more than `timeout` seconds, `asyncio.as_completed()` will raise an `asyncio.TimeoutError` for us.

You can easily modify this to try and cancel all tasks so all non-shielded tasks will be canceled right away. And you can also change the exception to a warning instead if the pending tasks are not critical in your use case.

`task = asyncio.shield(...)` protects against `task.cancel()` and functions like an onion. A single `asyncio.shield()` protects against a single `task.cancel()`; to protect against multiple cancelations, you will need to shield in a loop, or at least multiple times.

Lastly, it should be noted that this solution is not without its flaws either. It could happen that one of the tasks spawns new tasks while running; this is not something that is handled by this implementation, and handling it improperly might lead to waiting forever.

Now that we know how to debug the most common `asyncio` issues, it's time to end with a few exercises.

Exercises

Working with `asyncio` will require active thought throughout most of your development process. Besides `asyncio.run()` and similar methods, there is no way to run an `async def` from synchronous code. This means that every intermediate function between your main `async def` and the code that needs `asyncio` will have to be `async` as well.

You could make a synchronous function return a coroutine so one of the parent functions can run it within an event loop. But that usually results in a very confusing execution order of the code, so I would not recommend going down that route.

In short, this means that any `asyncio` project you try with the `asyncio` debug setting enabled is good practice. We can create a few challenges, however:

- Try to create a `asyncio` base class that automatically registers all instances for easy closing/destructuring when you are done
- Create an `asyncio` wrapper class for a synchronous process such as file or network operations using executors
- Convert any of your scripts or projects to `asyncio`

Example answers for these exercises can be found on GitHub: `https://github.com/mastering-python/exercises`. You are encouraged to submit your own solutions and learn about alternative solutions from others.

Summary

In this chapter, we have seen:

- The basic concepts of `asyncio` and how they interact
- How to run external processes using `asyncio`
- How to create a server and client using `asyncio`
- How to create context managers with `asyncio`
- How to create generators with `asyncio`
- How to debug common issues when using `asyncio`
- How to avoid the unfinished task pitfall

By now you should know how to keep your main loop responsive while waiting for results without having to resort to polling. In *Chapter 14*, *Multiprocessing – When a Single CPU Core Is Not Enough*, we will learn about `threading` and `multiprocessing` as an `asyncio` alternative to running multiple functions in parallel.

For new projects, I would strongly consider using `asyncio` from the ground up because it is usually the fastest solution for handling external resources. For existing scripts, however, this can be a very invasive process. So knowing about `threading` and `multiprocessing` is certainly important, also because `asyncio` can leverage them and you should be aware of thread and process safety.

When building utilities based on the `asyncio` library, make sure to search for pre-made libraries to solve your problems as `asyncio` is gaining more adoption every year. In many cases, someone has already created a library for you.

Next up is parallel execution using `threading` and `multiprocessing`.

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:
`https://discord.gg/QMzJenHuJf`

14

Multiprocessing – When a Single CPU Core Is Not Enough

In the previous chapter, we discussed `asyncio`, which can use the `threading` and `multiprocessing` modules but mainly uses single-thread/single-process parallelization. In this chapter, we will see how we can directly use multiple threads or processes to speed up our code and what caveats to keep in mind. This chapter can actually be seen as an extension to the list of performance tips.

The `threading` module makes it possible to run code in parallel in a single process. This makes `threading` very useful for I/O-related tasks such as reading/writing files or network communication, but a useless option for slow and heavy calculations, which is where the `multiprocessing` module shines.

With the `multiprocessing` module, you can run code in multiple processes, which means you can run code on multiple CPU cores, multiple processors, or even on multiple computers. This is an easy way to work around the **Global Interpreter Lock** (**GIL**) that was discussed in *Chapter 12*, *Performance – Tracking and Reducing Your Memory and CPU Usage*.

The `multiprocessing` module has a fairly easy-to-use interface with many convenience features, but the `threading` module is rather basic and requires you to manually create and manage the threads. For this, we also have the `concurrent.futures` module, which offers a simple way of executing a list of tasks either through threads or processes. This interface is also partially comparable to the `asyncio` features we saw in the previous chapter.

To summarize, this chapter covers:

- The Global Interpreter Lock (GIL)
- Multithreading versus multiprocessing
- Locking, deadlocks, and thread safety
- Data sharing and synchronization between processes
- Choosing between multithreading, multiprocessing, and single-threading
- Hyper-threading versus physical cores

- Remote multiprocessing with `multiprocessing` and `ipyparallel`

The Global Interpreter Lock (GIL)

The GIL has been mentioned in this book several times already, but we have not really covered it in detail and it really does need a bit more explanation before we continue with this chapter.

In short, the name already explains what it does. It is a global lock for the Python interpreter so it can only execute a single statement at once. A **lock** or **mutex** (**mutual exclusion**) in parallel computing is a synchronization primitive that can block parallel execution. With a lock, you can make sure that nobody can touch your variable while you are working on it.

Python offers several types of synchronization primitives, such as `threading.Lock` and `threading.Semaphore`. These are covered in more detail in the *Sharing data between threads and processes* section of this chapter.

That means that even with the `threading` module, you are still only executing a single Python statement at the same time. So, when it comes to pure Python code, your multithreaded solutions will **always** be slower than single-threaded solutions because `threading` introduces some synchronization overhead while not offering any benefit for that scenario.

Let's continue with some more in-depth information about the GIL.

The use of multiple threads

Since the GIL only allows a single Python statement to be executed at the same time, what point does threading have? The effectiveness greatly depends on your goal. Similar to the `asyncio` examples in *Chapter 13*, `threading` can give you a lot of benefit if you are waiting for external resources.

For example, if you are trying to fetch a webpage, open a file (remember that the `aiofiles` module actually uses threads), or if you want to execute something periodically, `threading` can work great.

When writing a new application, I would generally recommend that you make it ready for `asyncio` if there is even a small chance of becoming I/O-limited in the future. Rewriting for `asyncio` at a later time can be a huge amount of work.

There are several advantages to `asyncio` over `threading`:

- `asyncio` is generally faster than threading because you don't have any thread synchronization overhead.
- Since `asyncio` is normally single-threaded, you don't have to worry about thread safety (more about thread safety later in the chapter).

Why do we need the GIL?

The GIL is currently an essential part of the CPython interpreter because it makes sure that memory management is always consistent.

To explain how this works, we need to know a bit about how the CPython interpreter manages its memory.

Within CPython, the memory management system and garbage collection system rely on reference counting. This means that CPython counts how many names you have linked to a value. If you have a line of Python like this: `a = SomeObject()`, that means that this instance of `SomeObject` has 1 reference, namely, `a`. If we were to do `b = a`, the reference count would increase to 2. When the reference count reaches 0, the variable will be deleted by the garbage collector when it runs.

You can check the number of references using `sys.getrefcount(variable)`. You should note that the call to `sys.getrefcount()` increases your reference count by 1, so if it returns 2, the actual number is 1.

As the GIL makes sure that only a single Python statement can be executed simultaneously, you can never have issues where multiple bits of code manipulate memory at the same time, or where memory is being released to the system that is not actually free.

If the reference counter is not correctly managed, this could easily result in memory leaks or a crashing Python interpreter. Remember the segmentation faults we saw in *Chapter 11, Debugging – Solving the Bugs?* That is what could easily happen without the GIL, and it would instantly kill your Python interpreter.

Why do we still have the GIL?

When Python was initially created, many operating systems didn't even have a concept of threading, and all common processors only had a single core. Long story short, there are two main reasons for the GIL:

- There was initially no point in creating a complex solution that would handle threading
- The GIL is a really simple solution for a very complex problem

Luckily, it seems that is not the end of the discussion. Recently (May 2021), Guido van Rossum came out of retirement and he has plans to address the GIL limitations by creating sub-interpreters for threads. How this will work out in practice remains to be seen, of course, but the ambitious plan is to make CPython 3.15 about 5 times faster than CPython 3.10, which would be an amazing performance increase.

Now that we know when the GIL limits CPython threads, let's look at how we can create and use multiple threads and processes.

Multiple threads and processes

The `multiprocessing` module was introduced in Python 2.6, and it has been a game changer when it comes to working with multiple processes in Python. Specifically, it has made it rather easy to work around the limitations of the GIL because each process has its own GIL.

The usage of the `multiprocessing` module is largely similar to the `threading` module, but it has several really useful extra features that make much more sense with multiple processes. Alternatively, you can also use it with `concurrent.futures.ProcessPoolExecutor`, which has an interface nearly identical to `concurrent.futures.ThreadPoolExecutor`.

These similarities mean that in many cases you can simply swap out the modules and your code will keep running as expected. Don't be fooled, however; while threads can still use the same memory objects and only have thread safety and deadlocks to worry about, multiple processes also have these issues and introduce several other issues when it comes to sharing memory, objects, and results.

In either case, dealing with parallel code comes with caveats. This is also why code that uses multiple threads or processes has the reputation of being difficult to work with. Many of these issues are not as daunting as they might seem; if you follow a few rules, that is.

Before we continue with the example code, you should be aware that it is critically important to have your code in an `if __name__ == '__main__'` block when using `multiprocessing`. When the `multiprocessing` module launches the extra Python processes, it will execute the same Python script, so without using this block you will end up with an infinite loop of starting processes.

Within this section, we are going to cover:

- Basic examples using `threading`, `multiprocessing`, and `concurrent.futures`
- Cleanly exiting threads and processes
- Batch processing
- Sharing memory between processes
- Thread safety
- Deadlocks
- Thread-local variables

Several of these, such as race conditions and locking, are not exclusive to threading and could be interesting for `multiprocessing` as well.

Basic examples

To create threads and processes, we have several options:

- `concurrent.futures`: An easy-to-use interface for running functions in either threads or processes, similar to `asyncio`
- `threading`: An interface for creating threads directly
- `multiprocessing`: An interface with many utility and convenience functions to create and manage multiple Python processes

Let's look at an example of each one.

concurrent.futures

Let's start with a basic example of the `concurrent.futures` module. In this example, we run two timer jobs that run and print in parallel:

```
import time
import concurrent.futures
```

```
def timer(name, steps, interval=0.1):
    '''timer function that sleeps 'steps * interval' '''
    for step in range(steps):
        print(name, step)
        time.sleep(interval)

if __name__ == '__main__':
    # Replace with concurrent.futures.ProcessPoolExecutor for
    # multiple processes instead of threads
    with concurrent.futures.ThreadPoolExecutor() as executor:
        # Submit the function to the executor with some arguments
        executor.submit(timer, steps=3, name='a')
        # Sleep a tiny bit to keep the output order consistent
        time.sleep(0.1)
        executor.submit(timer, steps=3, name='b')
```

Before we execute the code, let's see what we did here. First, we created a `timer` function that runs `time.sleep(interval)` and does that `steps` times. Before sleeping, it prints the `name` and the current `step` so we can easily see what is happening.

Then, we create an `executor` using `concurrent.futures.ThreadPoolExecutor` to execute the functions.

Lastly, we submit the functions we want to execute with their respective arguments to start both of the threads. In between starting them, we sleep a very short time so our output in this example is consistent. If we were to execute the code without the `time.sleep(0.1)`, the output order would be random because sometimes `a` would be faster and other times `b` would be faster.

The main reason for including the tiny sleep is testing. All of the code in this book is available on GitHub (`https://github.com/mastering-python/code_2`) and is automatically tested.

Now when executing this script, we get the following:

```
$ python3 T_00_concurrent_futures.py
a 0
b 0
a 1
b 1
a 2
b 2
```

As expected, they run right next to each other, but due to the tiny `time.sleep(0.1)` we added, the results are consistently interleaved. In this case we started the `ThreadPoolExecutor` with the default arguments, which results in threads without specific names and an automatically calculated thread count.

The thread count depends on the Python version. Up to Python 3.8, the number of workers was equal to the number of hyper-threaded CPU cores in the machine multiplied by 5. So, if your machine has 2 cores with hyper-threading enabled, it would result in 4 cores * 5 = 20 threads. With a 64-core machine, that would result in 320 threads, which would probably incur more synchronization overhead than benefits.

For Python 3.8 and above, this has been changed to `min(32, cores + 4)`, which should be enough to always have at least 5 threads for I/O operations but not so much that it uses large amounts of resources on machines with many cores. For the same 64-core machine, this would still be capped at 32 threads.

In the case of the `ProcessPoolExecutor`, the number of processor cores including hyper-threading will be used. That means that if your processor has 4 cores with hyper-threading enabled, you will get a default of 8 processes.

Naturally, the traditional `threading` module is still a good option and offers a bit more control while still having an easy-to-use interface.

Before Python 3, the `thread` module was also available as a low-level API to threads. This module is still available but renamed to `_thread`. Internally, both `concurrent.futures.ThreadPoolExecutor` and `threading` are still using it, but you should generally have no need to access it directly.

threading

Now we will look at how to recreate the `concurrent.futures` example using the `threading` module:

```
import time
import threading

def timer(name, steps, interval=0.1):
    '''timer function that sleeps 'steps * interval' '''
    for step in range(steps):
        print(name, step)
        time.sleep(interval)

# Create the threads declaratively
a = threading.Thread(target=timer, kwargs=dict(name='a', steps=3))
b = threading.Thread(target=timer, kwargs=dict(name='b', steps=3))

# Start the threads
a.start()
# Sleep a tiny bit to keep the output order consistent
time.sleep(0.1)
b.start()
```

The `timer` function is identical to the previous example, so no difference there. The execution, however, is a bit different.

In this case we create the threads by instantiating `threading.Thread()` directly, but inheriting `threading.Thread` is also an option, as we will see in the next example. The arguments to the `target` function can be given by passing an `args` and/or `kwargs` argument, but these are optional if you have no need for them or if you have prefilled them using `functools.partial`.

With the earlier example, we created a `ThreadPoolExecutor()` that creates a bunch of threads and runs the functions on those threads. With this example, we are explicitly creating the threads to run a single function and exit as soon as the function is done. This is mostly useful for long-running backgrounded threads as this method requires setting up and tearing down a thread for each function. Generally, the overhead of starting a thread is very little, but it depends on your Python interpreter (CPython, PyPy, and so on) and your operating system.

Now for the same example, but inheriting `threading.Thread` instead of a declarative call to `threading.Thread()`:

```
import time
import threading

class Timer(threading.Thread):
    def __init__(self, name, steps, interval=0.1):
        self.steps = steps
        self.interval = interval
        # Small gotcha: threading.Thread has a built-in name
        # parameter so be careful not to manually override it
        super().__init__(name=name)

    def run(self):
        '''timer function that sleeps 'steps * interval' '''
        for step in range(self.steps):
            print(self.name, step)
            time.sleep(self.interval)

a = Timer(name='a', steps=3)
b = Timer(name='b', steps=3)

# Start the threads
a.start()
# Sleep a tiny bit to keep the output order consistent
time.sleep(0.1)
b.start()
```

The code is roughly the same as the procedural version where we called `threading.Thread()` directly, but there are two critical differences that you need to be aware of:

- `name` is a reserved attribute for `threading.Thread`. On Linux/Unix machines your process manager (for instance, `top`) can display this name instead of `/usr/bin/python3`.
- The default target function is `run()`. Be careful to override the `run()` method instead of the `start()` method, otherwise your code will *not* execute in a separate thread but will execute like a regular function call when you call `start()` instead.

The procedural and class-based versions use the exact same API internally and are equally powerful, so choosing between them comes down to personal preference only.

multiprocessing

Lastly, we can recreate the earlier timer scripts using `multiprocessing` as well. First with the procedural call to `multiprocessing.Process()`:

```
import time
import multiprocessing

def timer(name, steps, interval=0.1):
    '''timer function that sleeps 'steps * interval' '''
    for step in range(steps):
        print(name, step)
        time.sleep(interval)

if __name__ == '__main__':
    # Create the processes declaratively
    a = multiprocessing.Process(target=timer, kwargs=dict(name='a', steps=3))
    b = multiprocessing.Process(target=timer, kwargs=dict(name='b', steps=3))

    # Start the processes
    a.start()
    # Sleep a tiny bit to keep the output order consistent
    time.sleep(0.1)
    b.start()
```

The code looks effectively the same with a few minor changes. Instead of `threading.Thread` we used `multiprocessing.Process`, and we have to run the code from an `if __name__ == '__main__'` block. Beyond that, both the code and execution are the same in this simple example.

Lastly, for completeness, let's look at the class-based version as well:

```
import time
import multiprocessing
```

```
class Timer(multiprocessing.Process):
    def __init__(self, name, steps, interval=0.1):
        self.steps = steps
        self.interval = interval
        # Similar to threading.Thread, multiprocessing.Process
        # also supports the name parameter but you are not
        # required to use it here.
        super().__init__(name=name)

    def run(self):
        '''timer function that sleeps 'steps * interval' '''
        for step in range(self.steps):
            print(self.name, step)
            time.sleep(self.interval)

if __name__ == '__main__':
    a = Timer(name='a', steps=3)
    b = Timer(name='b', steps=3)

    # Start the process
    a.start()
    # Sleep a tiny bit to keep the output order consistent
    time.sleep(0.1)
    b.start()
```

Once again, we are required to use the `if __name__ == '__main__'` block. But beyond that, the code is virtually identical to the `threading` version. As was the case with `threading`, choosing between the procedural and class-based style depends only on your personal preference.

Now that we know how to start threads and processes, let's look at how we can cleanly shut them down again.

Cleanly exiting long-running threads and processes

The `threading` module is mostly useful for long-running threads that handle an external resource. Some example scenarios:

- When creating a server and you want to keep listening for new connections
- When connecting to HTTP WebSockets and you need the connection to stay open
- When you need to periodically save your changes

Naturally, these scenarios can also use `multiprocessing`, but `threading` is often more convenient, as we will see later.

At some point you might need to shut the thread down from **outside** of the thread; during the exit of your main script, for example. Waiting for a thread that is exiting by itself is trivial; the only thing you need to do is `future.result()` or `some_thread.join(timeout=...)` and you are done. The harder part is telling a thread to shut itself down and run the cleanup while it's still doing something.

The only real solution for this issue, which applies if you are lucky, is a simple `while` loop that keeps running until you give a stop signal like this:

```
import time
import threading

class Forever(threading.Thread):
    def __init__(self):
        self.stop = threading.Event()
        super().__init__()

    def run(self):
        while not self.stop.is_set():
            # Do whatever you need to do here
            time.sleep(0.1)

thread = Forever()
thread.start()
# Do whatever you need to do here
thread.stop.set()
thread.join()
```

This code uses `threading.Event()` as a flag to tell the thread to exit when needed. While you can use a `bool` instead of `threading.Event()` with the current CPython interpreter, there is no guarantee for this to work with future Python versions and/or other types of interpreters. The reason this is currently safe for CPython is that, due to the GIL, all Python operations are effectively single-threaded. That's why threads are useful for waiting for external resources, but have a negative effect on the performance of your Python code.

Additionally, if you were to translate this code to multiprocessing, you could simply replace `threading.Event()` with `multiprocessing.Event()` and it should keep working with no other changes, assuming you are not interacting with external variables. With multiple Python processes, you are no longer protected by the single GIL so you need to be more careful when modifying variables. More about this topic in the *Sharing data between threads and processes* section later in this chapter.

Now that we have the `stop` event, we can run `stop.set()` so the thread knows when to exit and will do so after the maximum of 0.1 seconds' sleep.

This is the ideal scenario: to have a loop where the loop condition is checked regularly and the loop interval is your maximum thread shutdown delay. What happens if the thread is busy doing some operation and doesn't check the `while` condition? As you might suspect, setting the `stop` event is useless in those scenarios and you need a more powerful method to exit the thread.

To handle this scenario, you have a few options:

- Avoid the issue entirely by using `asyncio` or `multiprocessing` instead. In terms of performance, `asyncio` is your best option by far, but `multiprocessing` can work as well if your code is suitable.
- Make the thread a daemon thread by setting `your_thread.daemon = True` *before* starting the thread. This will automatically kill the thread when the main process exits so it is not a graceful shutdown. You can still add a teardown using the `atexit` module.
- Kill the thread from the outside by either telling your operating system to send a terminate/kill signal or by raising an exception within the thread from the main thread. You might be tempted to go for this method, but I would strongly recommend against it. Not only is it unreliable, but it can cause your entire Python interpreter to crash, so it really is not an option you should ever consider.

We have already seen how to use `asyncio` in the previous chapter, so let's look at how we can terminate with `multiprocessing`. Before we start, however, you should note that the same limitations that apply to `threading` also largely apply to `multiprocessing`. While `multiprocessing` does have a built-in solution for terminating processes as opposed to threading, it is still not a clean method and it won't (reliably) run your exit handlers, `finally` clauses, and so on. This means you should *always* try an event first, but use `multiprocessing.Event` instead of `threading.Event`, of course.

To illustrate how we can forcefully terminate or kill a thread (while risking memory corruption):

```
import time
import multiprocessing

class Forever(multiprocessing.Process):
    def run(self):
        while True:
            # Do whatever you need to do here
            time.sleep(0.1)

if __name__ == '__main__':
    process = Forever()
    process.start()

    # Kill our "unkillable" process
    process.terminate()
```

```
    # Wait for 10 seconds to properly exit
    process.join(10)

    # If it still didn't exit, kill it
    if process.exitcode is None:
        process.kill()
```

In this example, we first try a regular `terminate()`, which sends a `SIGTERM` signal on Unix machines and `TerminateProcess()` on Windows. If that does not work, we try again with a `kill()`, which sends a `SIGKILL` on Unix and does not currently have a Windows equivalent, so on Windows the `kill()` and `terminate()` methods behave the same way and both effectively kill the process without teardown.

Batch processing using concurrent.futures

Starting threads or processes in a fire-and-forget fashion is easy enough, as we have seen in the prior examples. However, often, you want to spin up several threads or processes and wait until they have all finished.

This is a case where `concurrent.futures` and `multiprocessing` really shine. They allow you to call `executor.map()` or `pool.map()` very similarly to how we saw in *Chapter 5, Functional Programming – Readability Versus Brevity*. Effectively, you only need to create a list of items to process, call the `[executor/pool].map()` function, and you are done. You could build something similar with the `threading` module if you are looking for a fun challenge, but there is little use for it otherwise.

To give our system a test, let's get some information about a hostname that should use the system DNS resolving system. Since that queries an external resource, we should expect nice results when using threading, right? Well... let's give it a try and have a look:

```
import timeit
import socket
import concurrent.futures

def getaddrinfo(*args):
    # Call getaddrinfo but ignore the given parameter
    socket.getaddrinfo('localhost', None)

def benchmark(threads, n=1000):
    if threads > 1:
        # Create the executor
        with concurrent.futures.ThreadPoolExecutor(threads) \
                as executor:
            executor.map(getaddrinfo, range(n))
    else:
        # Make sure to use 'list'. Otherwise the generator will
```

```
        # not execute because it is lazy
        list(map(getaddrinfo, range(n)))

if __name__ == '__main__':
    for threads in (1, 10, 50, 100):
        print(f'Testing with {threads} threads and n={10} took: ',
              end='')
        print('{:.1f}'.format(timeit.timeit(
            f'benchmark({threads})',
            setup='from __main__ import benchmark',
            number=10,
        )))
```

Let's analyze this code. First, we have the `getaddrinfo()` function, which attempts to fetch some info about a hostname through your operating system, an external resource that could benefit from multiple threads.

Second, we have the `benchmark()` function, which uses multiple threads for the `map()` if `threads` is set to a number above 1. If not, it goes for the regular `map()`.

Lastly, we execute the benchmarks for `1`, `10`, `50`, and `100` threads where `1` is the regular non-threaded approach. So how much can threads help us here? This test strongly depends on your computer, operating system, network, etc., so your results may be different, but this is what happened on my OS X machine using CPython 3.10:

```
$ python3 T_07_thread_batch_processing.py
Testing with 1 threads and n=10 took: 2.1
Testing with 10 threads and n=10 took: 1.9
Testing with 50 threads and n=10 took: 1.9
Testing with 100 threads and n=10 took: 13.9
```

Did you expect those results? While `1` thread is indeed slower than `10` threads and `50` threads, at `100` we are clearly seeing the diminishing returns and the overhead of having `100` threads. Also, the benefit of using multiple threads is rather limited here due to `socket.getaddrinfo()` being pretty fast.

If we were to read a whole bunch of files from a slow networked filesystem or if we were to use it to fetch multiple webpages in parallel, we would see a much larger difference. That immediately shows the downside of threading: it only gives you a benefit if the external resource is slow enough to warrant the synchronization overhead. With a fast external resource, you are likely to experience slowdowns instead because the GIL becomes the bottleneck. CPython can only execute a single statement at once so that can quickly become problematic.

When it comes to performance, you should always run a benchmark to see what works best for your case, especially when it comes to thread count. As you saw in the earlier example, more is not always better and the 100-thread version is many times slower than even the single-threaded version.

So, what if we try the same using processes instead of threads? For brevity, we will skip the actual code since we effectively only need to swap out `concurrent.futures.ThreadPoolExecutor()` with `concurrent.futures.ProcessPoolExecutor()` and we are done. The tested code can be found on GitHub if you are interested. When we execute that code, we get these results:

```
$ python3 T_08_process_batch_processing.py
Testing with 1 processes and n=10 took: 2.1
Testing with 10 processes and n=10 took: 3.2
Testing with 50 processes and n=10 took: 8.3
Testing with 100 processes and n=10 took: 15.0
```

As you can see, we got universally slower results when using multiple processes. While multiprocessing can offer a lot of benefit when the GIL or a single CPU core is the limit, the overhead can cost you performance in other scenarios.

Batch processing using multiprocessing

In the previous section, we saw how we can use `concurrent.futures` to do batch processing. You might be wondering why we would want to use `multiprocessing` directly if `concurrent.futures` can handle it for us. The reason is rather simple: `concurrent.futures` is an easy-to-use and very simple interface to both `threading` and `multiprocessing`, but `multiprocessing` offers several advanced options that can be very convenient and can even help your performance in some scenarios.

In the previous examples we only saw `multiprocessing.Process`, which is the process analog to `threading.Thread`. In this case, however, we will be using `multiprocessing.Pool`, which creates a process pool very similar to the `concurrent.futures` executors but offers several additional features:

- `map_async(func, iterable, [..., callback, ...])`

  The `map_async()` method is similar to the `map()` method in `concurrent.futures`, but instead of blocking it returns a list of `AsyncResult` objects so you can fetch the results when you need them.

- `imap(func, iterable[, chunksize])`

  The `imap()` method is effectively the generator version of `map()`. It works in roughly the same way, but it doesn't preload the items from the iterable so you can safely process large iterables if needed. This can be *much* faster if you need to process many items.

- `imap_unordered(func, iterable[, chunksize])`

  The `imap_unordered()` method is effectively the same as `imap()` except that it returns the results as soon as they are processed, which can improve performance even further. If the order of your results is of no importance to you, give it a try as it can make your code even faster.

- `starmap(func, iterable[, chunksize])`

  The `starmap()` method is very similar to the `map()` method, but supports multiple arguments by passing them like `*args`. If you were to run `starmap(function, [(1, 2), (3, 4)])`, the `starmap()` method would call `function(1, 2)` and `function(3, 4)`. This can be really useful in conjunction with `zip()` to combine several lists of arguments.

- `starmap_async(func, iterable, [..., callback, ...])`

  As you can imagine, `starmap_async()` is effectively the non-blocking `starmap()` method, but it returns a list of `AsyncResult` objects so you can fetch them at your convenience.

The usage of `multiprocessing.Pool()` is largely analogous to `concurrent.future.SomeExecutor()` beyond the extra methods mentioned above. Depending on your scenario, it can be slower, a similar speed, or faster than `concurrent.futures`, so always make sure to benchmark for your specific use case. This little bit of benchmark code should give you a nice starting point:

```
import timeit
import functools
import multiprocessing
import concurrent.futures

def triangle_number(n):
    total = 0
    for i in range(n + 1):
        total += i

    return total

def bench_mp(n, count, chunksize):
    with multiprocessing.Pool() as pool:
        # Generate a generator like [n, n, n, ..., n, n]
        iterable = (n for _ in range(count))
        list(pool.imap_unordered(triangle_number, iterable,
                                 chunksize=chunksize))

def bench_ft(n, count, chunksize):
    with concurrent.futures.ProcessPoolExecutor() as executor:
        # Generate a generator like [n, n, n, ..., n, n]
        iterable = (n for _ in range(count))
        list(executor.map(triangle_number, iterable,
                          chunksize=chunksize))

if __name__ == '__main__':
    timer = functools.partial(timeit.timeit, number=5)

    n = 1000
    chunksize = 50
```

```
    for count in (100, 1000, 10000):
        # Using <6 formatting for consistent alignment
        args = ', '.join((
            f'n={n:<6}',
            f'count={count:<6}',
            f'chunksize={chunksize:<6}',
        ))
        time_mp = timer(
            f'bench_mp({args})',
            setup='from __main__ import bench_mp',
        )
        time_ft = timer(
            f'bench_ft({args})',
            setup='from __main__ import bench_ft',
        )

        print(f'{args} mp: {time_mp:.2f}, ft: {time_ft:.2f}')
```

On my machine, this gives the following results:

```
$ python3 T_09_multiprocessing_pool.py
n=1000  , count=100   , chunksize=50     mp: 0.71, ft: 0.42
n=1000  , count=1000  , chunksize=50     mp: 0.76, ft: 0.96
n=1000  , count=10000 , chunksize=50     mp: 1.12, ft: 1.40
```

Before I benchmarked this, I was not expecting `concurrent.futures` to be that much faster in some cases and that much slower in other cases. Analyzing these results, you can see that processing 1,000 items with `concurrent.futures` took more time than processing 10,000 items with multiprocessing in this particular case. Similarly, for 100 items the `multiprocessing` module was nearly twice as slow. Naturally, every run yields different results and there is not a single option that will perform well for every scenario, but it is something to keep in mind.

Now that we know how to run our code in multiple threads or processes, let's look at how we can safely share data between the threads/processes.

Sharing data between threads and processes

Data sharing is really the most difficult part about multiprocessing, multithreading, and distributed programming in general: which data to pass along, which data to share, and which data to skip. The theory is really simple, however: whenever possible, don't transfer any data, don't share any data, and keep everything local. This is essentially the **functional programming** paradigm, which is why functional programming mixes really well with multiprocessing. In practice, regrettably, this is simply not always possible. The `multiprocessing` library has several options to share data, but internally they break down to two different options:

- **Shared memory**: This is by far the fastest solution since it has very little overhead, but it can only be used for immutable types and is restricted to a select few types and custom objects that are created through `multiprocessing.sharedctypes`. This is a fantastic solution if you only need to store primitive types such as `int`, `float`, `bool`, `str`, `bytes`, and/or fixed-sized lists or dicts (where the children are primitives).
- `multiprocessing.Manager`: The `Manager` classes offer a host of different options for storing and synchronizing data, such as locks, semaphores, queues, lists, dicts, and several others. If it can be pickled, it can work with a manager.

For threading, the solution is even easier: all memory is shared so, by default, all objects are available from every thread. There is an exception called a thread-local variable, which we will see later.

Sharing memory brings its own caveats, however, as we will see in the *Thread safety* section in the case of `threading`. Since multiple threads and/or processes can write to the same piece of memory at the same time, this is an inherently risky operation. At best, your changes can become lost due to conflicting writes; at worst, your memory could become corrupted, which could even result in a crashing interpreter. Luckily, Python is pretty good at protecting you, so if you are not doing anything too exotic you do not have to worry about crashing your interpreter.

Shared memory between processes

Python offers several different structures to make memory sharing between processes a safe operation:

- `multiprocessing.Value`
- `multiprocessing.Array`
- `multiprocessing.shared_memory.SharedMemory`
- `multiprocessing.shared_memory.ShareableList`

Let's dive into a few of these types to demonstrate how to use them.

For sharing primitive values, you can use `multiprocessing.Value` and `multiprocessing.Array`. These are essentially the same, but with `Array` you can store multiple values whereas `Value` is just a single value. As arguments, these expect a typecode identical to how the `array` module works in Python, which means they map to C types. This results in `d` as a double (floating point) number, `i` as a signed integer, `b` as a signed char, etc.

For more options, look at the documentation for the `array` module: `https://docs.python.org/3/library/array.html`.

For more advanced types, you can take a look at the `multiprocessing.sharedctypes` module, which is also where the `Value` and `Array` classes originate from.

Both `multiprocessing.Value` and `multiprocessing.Array` are not difficult to use, but they do not feel very Pythonic to me:

```
import multiprocessing

some_int = multiprocessing.Value('i', 123)
with some_int.get_lock():
    some_int.value += 10
print(some_int.value)

some_double_array = multiprocessing.Array('d', [1, 2, 3])
with some_double_array.get_lock():
    some_double_array[0] += 2.5
print(some_double_array[:])
```

If you need to share memory and performance is important to you, feel free to use them. If possible, however, I would avoid them (or sharing memory at all if possible) as the usage is clunky at best.

The `multiprocessing.shared_memory.SharedMemory` object is similar to the `Array` but it is a lower-level structure. It offers you an interface to read/write to an optionally **named** block of memory so you can access it from other processes by name as well. Additionally, when you are done using it you *must* call `unlink()` to release the memory:

```
from multiprocessing import shared_memory

# From process A we could write something
name = 'share_a'
share_a = shared_memory.SharedMemory(name, create=True, size=4)
share_a.buf[0] = 10

# From a different process, or the same one, we can access the data
share_a = shared_memory.SharedMemory(name)
print(share_a.buf[0])

# Make sure to clean up after. And only once!
share_a.unlink()
```

As we can see in this example, the first call had a `create=True` parameter to ask the operating system for memory. Only after that (and before calling `unlink()`) can we reference the block from other (or the same) processes.

Once again it is not the most Pythonic interface, but it can be effective for sharing memory. Since the name is optional and automatically generated otherwise, you could omit it from the creation of the shared memory block and read it back from `share_a.name`. Also, like the `Array` and `Value` objects, this too has a fixed size and cannot be grown without replacing it.

Lastly, we have the `multiprocessing.shared_memory.ShareableList` object. While this object is slightly more convenient than `Array` and `SharedMemory` since it allows you to be flexible with types (i.e. `item[0]` could be a `str` and `item[1]` could be an `int`), it is still a hard-to-use interface and it does not allow you to resize it. While you can change the type for the items, you cannot resize the object, so swapping out a number with a larger string will not work. At least its usage is more Pythonic than the other options:

```
from multiprocessing import shared_memory

shared_list = shared_memory.ShareableList(['Hi', 1, False, None])
# Changing type from str to bool here
shared_list[0] = True
# Don't forget to unlink()
shared_list.shm.unlink()
```

Seeing all of these options for sharing memory between processes, should you be using them? Yes, if you need high performance, that is.

It should be a good indication of why it is best to keep memory local with parallel processing, however. Sharing memory between processes is a complicated problem to solve. Even with these methods, which are the fastest and least complicated available, it is a bit of a pain already.

So, how much performance impact does memory sharing have? Let's run a few benchmarks to see the difference between sharing a variable and returning it for post-processing. First, the version that does not use shared memory as a performance base:

```
import multiprocessing

def triangle_number_local(n):
    total = 0
    for i in range(n + 1):
        total += i

    return total

def bench_local(n, count):
    with multiprocessing.Pool() as pool:
        results = pool.imap_unordered(
            triangle_number_local,
            (n for _ in range(count)),
        )
        print('Sum:', sum(results))
```

The `triangle_number_local()` function calculates the sum of all numbers up to and including `n` and returns it, similar to a factorial function but with addition instead.

The `bench_local()` function calls the `triangle_number_local()` function `count` times and stores the results. After that, we `sum()` those results to verify the output.

Now let's look at the version using shared memory:

```
import multiprocessing

class Shared:
    pass

def initializer(shared_value):
    Shared.value = shared_value

def triangle_number_shared(n):
    for i in range(n + 1):
        with Shared.value.get_lock():
            Shared.value.value += i

def bench_shared(n, count):
    shared_value = multiprocessing.Value('i', 0)

    # We need to explicitly share the shared_value. On Unix you
    # can work around this by forking the process, on Windows it
    # would not work otherwise
    pool = multiprocessing.Pool(
        initializer=initializer,
        initargs=(shared_value,),
    )

    iterable = (n for _ in range(count))
    list(pool.imap_unordered(triangle_number_shared, iterable))
    print('Sum:', shared_value.value)

    pool.close()
```

In this case we have created a `Shared` class as a namespace to store the shared variable, but a `global` variable would also be an option.

To make sure the shared variable is available, we need to send it along to all workers in the `pool` using an `initializer` method argument.

Additionally, as the `+=` operation is not atomic (not a single operation, since it does *fetch, add, set*), we need to make sure to lock the variable using the `get_lock()` method.

The *Thread safety* section later in this chapter goes into more detail about when locking is and is not needed.

To run the benchmarks, we use the following code:

```
import timeit

if __name__ == '__main__':
    n = 1000
    count = 100
    number = 5

    for function in 'bench_local', 'bench_shared':
        statement = f'{function}(n={n}, count={count})'
        result = timeit.timeit(
            statement, number=number,
            setup=f'from __main__ import {function}',
        )
        print(f'{statement}: {result:.3f}')
```

Now when executing this, we see the reason for not sharing memory if possible:

```
bench_local(n=1000, count=100): 0.598
bench_shared(n=1000, count=100): 4.157
```

The code using shared memory is roughly 8 times slower, which makes sense because my machine has 8 cores. Since the shared memory example spends most of its time with locking/unlocking (which can only be done by one process at the same time), we have effectively made the code run on a single core again.

I should point out that this is pretty much the worst-case scenario for shared memory. Since all the functions do is write to the shared variable, most of the time is spent locking and unlocking the variables. If you were to do actual processing in the function and only write the results, it would be much better already.

You might be curious about how we could rewrite this example the right way while still using shared variables. In this case it is rather easy, but this largely depends on your specific use case and this might not work for you:

```
def triangle_number_shared_efficient(n):
    total = 0
    for i in range(n + 1):
        total += i

    with Shared.value.get_lock():
```

```
            Shared.value.value += total
```

This code runs almost as fast as the `bench_local()` function. As a rule of thumb, just remember to reduce the number of locks and writes as much as possible.

Sharing data between processes using managers

Now that we have seen how we can directly share memory to get the best performance possible, let's look at a far more convenient and much more flexible solution: the `multiprocessing.Manager` class.

Whereas shared memory restricted us to primitive types, with a `Manager` we can share anything that can be pickled in a very easy way if we are willing to sacrifice a little bit of performance. The mechanism it uses is very different, though; it connects through a network connection. The huge advantage of this method is that you can even use this across multiple devices (which we will see later in this chapter).

The `Manager` itself is not an object you will use much, though you will probably use the objects provided by the `Manager`. The list is plentiful so we will only cover a few in detail, but you can always look at the Python documentation for the current list of options: `https://docs.python.org/3/library/multiprocessing.html#managers`.

One of the most convenient options for sharing data with `multiprocessing` is the `multiprocessing.Namespace` object. The `Namespace` object behaves very similarly to a regular object, with the difference being that it can be accessed as a shared memory object from all processes. As long as your objects can be pickled, you can use them as attributes of a `Namespace` instance. To illustrate the usage of the `Namespace`:

```
import multiprocessing

manager = multiprocessing.Manager()
namespace = manager.Namespace()

namespace.spam = 123
namespace.eggs = 456
```

As you can see in this example, you can simply set the attributes of `namespace` as you would expect from regular objects, but they are shared between all processes. Since the locking now happens through network sockets, the overhead is even larger than with shared memory, so only write data when you must. Directly translating the earlier shared memory example to use a `Namespace` and explicit `Lock` (a `Namespace` does not have a `get_lock()` method) yields the following code:

```
def triangle_number_namespace(namespace, lock, n):
    for i in range(n + 1):
        with lock:
            namespace.total += i

def bench_manager(n, count):
```

```
    manager = multiprocessing.Manager()
    namespace = manager.Namespace()
    namespace.total = 0
    lock = manager.Lock()
    with multiprocessing.Pool() as pool:
        list(pool.starmap(
            triangle_number_namespace,
            ((namespace, lock, n) for _ in range(count)),
        ))
        print('Sum:', namespace.total)
```

As with the shared memory example, this is a really inefficient case because we are locking for each iteration of the loop, and it really shows. While the local version took about 0.6 seconds and the shared memory version took about 4 seconds, this version takes a whopping 90 seconds for effectively the same operation.

Once again, we can easily speed it up by reducing the time spent in the synchronized/locked code:

```
def triangle_number_namespace_efficient(namespace, lock, n):
    total = 0
    for i in range(n + 1):
        total += i

    with lock:
        namespace.total += i
```

When benchmarking this version with the same benchmark code as before, we can see that it is still much slower than the 0.6 seconds we got with the local version:

```
bench_local(n=1000, count=100): 0.637
bench_manager(n=1000, count=100): 1.476
```

That being said, at least this is much more acceptable than the 90 seconds we would get otherwise.

Why are these locks so incredibly slow? For a proper lock to be set, all the parties need to agree that the data is locked, which is a process that takes time. That simple fact slows down execution much more than most people would expect. The server/process that runs the `Manager` needs to confirm to the client that it has the lock; only once that has been done can the client read, write, and release the lock again.

On a regular hard disk setup, database servers aren't able to handle more than about 10 transactions per second *on the same row* due to locking and disk latency. Using lazy file syncing, SSDs, and a battery-backed RAID cache, that performance can be increased to handle, perhaps, 100 transactions per second on the same row. These are simple hardware limitations; because you have multiple processes trying to write to a single target, you need to synchronize the actions between the processes, and that takes a lot of time.

Even with the fastest hardware available, synchronization can lock all the processes and produce enormous slowdowns, so if at all possible, try to avoid sharing data between multiple processes. Put simply, if all the processes are constantly reading and writing from/to the same object, it is generally faster to use a single process instead because the locking will effectively restrict you to a single process anyway.

Redis, one of the fastest data storage systems available, was fully single-threaded for over a decade until 2020 because the locking overhead was not worth the benefit. Even the current threaded version is effectively a collection of single-threaded servers with their own memory space to avoid locking.

Thread safety

When working with threads or processes, you need to be aware that you might not be the only one modifying a variable at some point in time. There are many scenarios where this will not be an issue and often you are lucky and it won't affect you, but when it does it can cause bugs that are extremely difficult to debug.

As an example, imagine having two bits of code incrementing a number at the same time and imagine what could go wrong. Initially, let's assume the value is 10. With multiple threads, this could result in the following sequence:

1. Two threads fetch the number to local memory to increment. It is currently 10 for both.
2. Both threads increment the number in their local memory to 11.
3. Both threads write the number back from local memory (which is 11 for both) to the global one, so the global number is now 11.

Since both threads fetched the number at the same time, one overwrote the increment of the other with its own increment. So instead of incrementing twice, you now have a variable that was only incremented once.

In many cases, the current GIL implementation in CPython will protect you from these issues when using `threading`, but you should never take that protection for granted and make sure to protect your variables if multiple threads might update your variable at the same time.

Perhaps an actual code example might make the scenario a bit clearer:

```
import time
import concurrent.futures

counter = 10

def increment(name):
    global counter
    current_value = counter
    print(f'{name} value before increment: {current_value}')
    counter = current_value + 1
```

```
    print(f'{name} value after increment: {counter}')

print(f'Before thread start: {counter}')

with concurrent.futures.ThreadPoolExecutor() as executor:
    executor.map(increment, range(3))
print(f'After thread finish: {counter}')
```

As you can see, the `increment` function stores `counter` in a temporary variable, prints it, and writes to `counter` after adding 1 to it. This example is admittedly a bit contrived because you would normally do `counter += 1` instead, which reduces the odds of unexpected behaviour, but even in that case you have no guarantee that your results are correct.

To illustrate the output of this script:

```
$ python3 T_12_thread_safety.py
Before thread start: 10
0 value before increment: 10
0 value after increment: 11
1 value before increment: 11
1 value after increment: 12
2 value before increment: 11
2 value after increment: 12
4 value before increment: 12
4 value after increment: 13
3 value before increment: 12
3 value after increment: 13
After thread finish: 13
```

Why did we end up with 13 at the end? Pure luck really. Some of my attempts resulted in 15, some in 11, and others in 14. That's what makes thread safety issues so incredibly hard to debug; in a complicated codebase, it can be really hard to figure out what is causing the bug and you cannot reliably reproduce the issue.

When experiencing strange and hard-to-explain errors in a system using multiple threads/processes, make sure to see if they also occur when running single-threaded. Mistakes like these are easily made and can easily be introduced by third-party code that was not meant to be thread-safe.

To make your code thread-safe, you have a few different options:

- This might seem obvious, but if you don't update shared variables from multiple threads/processes in parallel then there is nothing to worry about.

- Use atomic operations when modifying your variables. An atomic operation is one that executes in a single instruction so no conflicts could ever arise. For example, incrementing a number could be an atomic operation where the fetching, incrementing, and updating happens in a single instruction. Within Python, an increment is usually done with `counter += 1` which is actually a shorthand for `counter = counter + 1`. Can you see the issue here? Instead of incrementing `counter` internally, Python will write a new value to the variable `counter`, which means it is not an atomic operation.
- Use locks to protect your variables.

Knowing these options for thread-safe code, you might be wondering which operations are thread-safe and which aren't. Luckily, Python does have some documentation about the issue, and I would strongly recommend looking at it as this is prone to change in the future: `https://docs.python.org/3/faq/library.html#what-kinds-of-global-value-mutation-are-thread-safe`.

For the current CPython versions (at least CPython 3.10 and below) where the GIL is protecting us, we can assume these operations to be atomic and therefore thread-safe:

```
L.append(x)
L1.extend(L2)
x = L[i]
x = L.pop()
L1[i:j] = L2
L.sort()
x = y
x.field = y
D[x] = y
D1.update(D2)
D.keys()
```

These are not atomic and not thread-safe:

```
i = i+1
L.append(L[-1])
L[i] = L[j]
D[x] = D[x] + 1
```

What could we do to make `i = i + 1` thread-safe? The most obvious solution is to use our own lock, similar to the GIL:

```
# This lock needs to be the same object for all threads
lock = threading.Lock()
i = 0

def increment():
    global i
```

```
    with lock():
        i += 1
```

As you can see, with a lock we can protect the updates for a variable easily. I should note that even though we used a `global` variable in this case, the same limitation applies for the attributes of class instances and other variables as well.

Naturally, this all applies to `multiprocessing` as well, with the subtle difference that variables are not shared by default with multiple processes, so you need to do something to explicitly cause an issue. Having said that, the earlier shared memory and `Manager` examples break immediately if you remove the locks from them.

Deadlocks

Now that you know how to update your variables in a thread-safe manner, you might be hoping that we are done with threading limitations. Unfortunately, the opposite is true. The locks we used to make our variable updates thread-safe can actually introduce another issue that can be even more devious to solve: **deadlocks**.

A deadlock can occur when threads or processes are holding a lock while waiting for another thread/process to release a lock. In some cases, you can even have a thread/process that is waiting for itself. To illustrate, let's assume that we have locks `a` and `b` and two different threads. Now the following occurs:

1. Thread 0 locks `a`
2. Thread 1 locks `b`
3. Thread 0 waits for lock `b`
4. Thread 1 waits for lock `a`

Now thread 1 is waiting for thread 0 to finish, and vice versa. Neither will ever finish because they are waiting for each other.

To illustrate this scenario:

```
import time
import threading

a = threading.Lock()
b = threading.Lock()

def thread_0():
    print('thread 0 locking a')
    with a:
        time.sleep(0.1)
        print('thread 0 locking b')
        with b:
```

```
            print('thread 0 everything locked')

def thread_1():
    print('thread 1 locking b')
    with b:
        time.sleep(0.1)
        print('thread 1 locking a')
        with a:
            print('thread 1 everything locked')

threading.Thread(target=thread_0).start()
threading.Thread(target=thread_1).start()
```

The code is relatively straightforward but warrants at least some explanation. As previously discussed, the `thread_0` function locks `a` first and `b` after and `thread_1` does this in the reverse order. This is what causes the deadlock; they will each wait for the other to finish. To be sure we actually reach the deadlock in this example, we have a small sleep to make sure `thread_0` does not finish before `thread_1` starts. In real-world scenarios, you would have some code in that bit that would take time as well.

How can we resolve locking issues like these? Locking strategies and resolving these issues could easily fill a chapter by themselves and there are several different types of locking problems and solutions. You could even have a livelock problem where both threads are attempting to resolve the deadlock problem at the same time with the same method, causing them to also wait for each other but with constantly changing locks.

An easy way to visualize a livelock is to think of a narrow part of a road where two cars are approaching from opposite sides. Both cars would attempt to drive at the same time and both would back off when they notice that the other car is moving. Repeat that and you have a livelock.

In general, there are several strategies that you can employ to avoid deadlocks:

- Deadlocks can only occur when you have multiple locks, so if your code only ever acquires a single lock at the same time, no problems can occur.
- Try to keep the lock section small so there is less chance of accidentally adding another lock within that block. This can also help performance because a lock can make your parallel code essentially single-threaded again.
- This is probably the most important tip for fixing deadlocks. *Always have a consistent locking order.* If you always lock in the same order, you can never have deadlocks. Let's explain how this helps:With the earlier example and the two locks `a` and `b`, the problem occurred because thread 0 was waiting for `b` and thread 1 was waiting for `a`.If they both had attempted to lock `a` first and `b` after, we never would have reached the deadlock state because one of the threads would lock `a` and that would cause the other thread to stall long before `b` could ever be locked.

Thread-local variables

We have seen how to lock variables so only a single thread can modify a variable simultaneously. We have also seen how we can prevent deadlocks while using locks. What if we want to give a thread a separate global variable? That is where `threading.local` comes in: it gives you a context specifically for your current thread. This can be useful for database connections, for example; you probably want to give each thread its own database connection, but having to pass around the connection is inconvenient, so a global variable or connection manager is a much more convenient option.

This section does not apply to `multiprocessing`, since variables are not automatically shared between processes. A forked process can inherit the variables from the parent, however, so care must be taken to explicitly initialize non-shared resources.

Let's illustrate the usage of thread-local variables with a small example:

```
import threading
import concurrent.futures

context = threading.local()

def init_counter():
    context.counter = 10

def increment(name):
    current_value = context.counter
    print(f'{name} value before increment: {current_value}')
    context.counter = current_value + 1
    print(f'{name} value after increment: {context.counter}')

init_counter()
print(f'Before thread start: {context.counter}')

with concurrent.futures.ThreadPoolExecutor(
        initializer=init_counter) as executor:
    executor.map(increment, range(5))

print(f'After thread finish: {context.counter}')
```

This example is largely the same as the thread-safety example, but instead of having a global `counter` variable, we are now using `threading.local()` as a context to set the `counter` variable to. We are also using an extra feature of the `concurrent.futures.ThreadPoolExecutor` here, the `initializer` function. Since a thread-local variable only exists within that thread and is not automatically copied to other threads, all threads (including the main thread) need to have `counter` set separately. Without setting it, we would get an `AttributeError`.

When running the code, we can see that all threads are independently updating their variables instead of the completely mixed version we saw in the thread-safety example:

```
$ python3 T_15_thread_local.py
Before thread start: 10
0 value before increment: 10
0 value after increment: 11
1 value before increment: 10
2 value before increment: 11
1 value after increment: 11
3 value before increment: 10
3 value after increment: 11
2 value after increment: 12
4 value before increment: 10
4 value after increment: 11
After thread finish: 10
```

If possible, I would always recommend returning variables from a thread or appending them to a post-processing queue and never updating a global variable or global state because it is faster and less error-prone. Using thread-local variables can really help you in these cases to make sure you have only one instance of a connection or collection class.

Now that we know how to share (or stop sharing) variables, it is time to learn about the advantages and disadvantages of using threads as opposed to processes. We should have a basic grasp of memory management with threads and processes now. With all of these options, what should we choose and why?

Processes, threads, or a single thread?

Now that we know how to use `multiprocessing`, `threading` and `concurrent.futures`, which should you choose for your case?

Since `concurrent.futures` implements both `threading`, and `multiprocessing`, you can mentally exchange `threading` in this section with `concurrent.futures.ThreadPoolExecutor`. The same goes for `multiprocessing` and `concurrent.futures.ProcessPoolExecutor`, of course.

When we consider the choice between single-threaded, multithreaded, and multiprocess, there are multiple factors that we can consider.

The first and most important question you should ask yourself is whether you really need to use `threading` or `multiprocessing`. Often, code is fast enough and you should ask yourself if the cost of dealing with the potential side effects of memory sharing and such is worth it. Not only does writing code become more complicated when parallel processing is involved, but the complexity of debugging is multiplied as well.

Second, you should ask yourself what is limiting your performance. If the limitation is external I/O, then it could be useful to use `asyncio` or `threading` to handle that, but it is still no guarantee.

For example, if you are reading a bunch of files from a slow hard disk, threading might not even help you. If the hard disk is the limiting factor, it will not become faster no matter what you try. So before you rewrite your entire codebase to function with `threading`, make sure to test if your solution has any chance of working.

Assuming that your I/O bottleneck can be alleviated, then you still have the choice of `asyncio` versus `threading`. Since `asyncio` is the fastest of the available options, I would opt for that solution if it works with your codebase, but using `threading` is not a bad option either, of course.

If the GIL is your bottleneck due to heavy calculations from your Python code, then `multiprocessing` can help you a lot. But even in those cases, `multiprocessing` is not your only option; for many slow processes, it can also help to employ fast libraries such as `numpy`.

I am a great fan of the `multiprocessing` library and it is one of the easiest implementations of multiprocess code that I have seen so far, but it still comes with several caveats such as more difficult memory management and deadlocks, as we have seen. So always consider if you actually need the solution and if your problem is suitable for multiprocessing. If a large portion of code is written using functional programming it can be really easy to implement; if you need to interact with a lot of external resources, such as databases, it can be really difficult to implement.

threading versus concurrent.futures

When given the choice, should you use `threading` or `concurrent.futures`? In my opinion, it depends on what you are trying to do.

The advantages of `threading` over `concurrent.futures` are:

- We can specify the name of the thread explicitly, which can be seen in the task manager on many operating systems.
- We can explicitly create and start a long-running thread for a function instead of relying on the availability within a thread pool.

If your scenario allows you to choose, I believe you should use `concurrent.futures` instead of `threading` for the following reasons:

- With `concurrent.futures` you can switch between threads and processes by using `concurrent.futures.ProcessPoolExecutor` instead of `concurrent.futures.ThreadPoolExecutor`.
- With `concurrent.futures` you have the `map()` method to easily batch-process a list of items without having the (potential) overhead of setting up and shutting down the thread.
- The `concurrent.futures.Future` objects as returned by the `concurrent.futures` methods allow for fine-grained control of the results and the handling.

multiprocessing versus concurrent.futures

When it comes to multiprocessing, I think the `concurrent.futures` interface adds much less benefit than it does in the case of threading, especially since `multiprocessing.Pool` essentially offers you a nearly identical interface to `concurrent.futures.ProcessPoolExecutor`.

The advantages of `multiprocessing` over `concurrent.futures` are:

- Many advanced mapping methods such as `imap_unordered` and `starmap`.
- More control over the pool (i.e. `terminate()`, `close()`).
- It can be used across multiple machines.
- You can manually specify the startup method (`fork`, `spawn`, or `forkserver`), which gives you control over how variables are copied from the parent process.
- You can choose the Python interpreter. Using `multiprocessing.set_executable()`, you could run a Python 3.10 pool while running Python 3.9 for the main process.

The advantages of `concurrent.futures` over `multiprocessing` are:

- You can easily switch to the `concurrent.futures.ThreadPoolExecutor`.
- The returned `Future` objects allow for more fine-grained control over the result handling when compared to the `AsyncResult` objects `multiprocessing` uses.

Personally, I prefer `multiprocessing` if you have no need for compatibility with `threads` because of the advanced mapping methods.

Hyper-threading versus physical CPU cores

Hyper-threading is a technology that offers extra virtual CPU cores to your physical cores. The idea is that, because these virtual CPU cores have separate caches and other resources, you can more efficiently switch between multiple tasks. If you task-switch between two heavy processes, the CPU won't have to unload/reload all caches. When it comes to actual CPU instruction processing, however, it will not help you.

When you truly maximize CPU usage, it is generally better to only use the physical processor count. To demonstrate how this affects the performance, we will run a simple test with several process counts. Since my processor has `8` cores (`16` if you include hyper-threading), we will run it with `1`, `2`, `4`, `8`, `16`, and `32` processes to demonstrate how it affects the performance:

```
import timeit
import multiprocessing

def busy_wait(n):
    while n > 0:
        n -= 1

def benchmark(n, processes, tasks):
    with multiprocessing.Pool(processes=processes) as pool:
        # Execute the busy_wait function 'tasks' times with
        # parameter n
```

```
            pool.map(busy_wait, [n for _ in range(tasks)])
        # Create the executor

if __name__ == '__main__':
    n = 100000
    tasks = 128
    for exponent in range(6):
        processes = int(2 ** exponent)
        statement = f'benchmark({n}, {processes}, {tasks})'
        result = timeit.timeit(
            statement,
            number=5,
            setup='from __main__ import benchmark',
        )
        print(f'{statement}: {result:.3f}')
```

To keep the processor busy, we are using a `while` loop from `n` to `0` in the `busy_wait()` function. For the benchmarking, we are using a `multiprocessing.Pool()` instance with the given number of processes and running `busy_wait(100000)` 128 times:

```
$ python3 T_16_hyper_threading.py
benchmark(100000, 1): 3.400
benchmark(100000, 2): 1.894
benchmark(100000, 4): 1.208
benchmark(100000, 8): 0.998
benchmark(100000, 16): 1.124
benchmark(100000, 32): 1.787
```

As you can see, with my 8-core CPU with hyper-threading enabled, the version with `8` threads is obviously the fastest. Even though the operating system task manager shows `16` cores, it is not always faster to utilize more than the `8` physical cores. Additionally, due to the boosting behavior of modern processors, you can see that using `8` processors is only `3.4` times faster than the single-threaded variant, as opposed to the expected `8`-times speedup.

This illustrates the problem with hyper-threading when heavily loading the processor with instructions. As soon as the single processes actually use 100% of a CPU core, the task switching between the processes actually reduces performance. Since there are only `8` physical cores, the other processes have to fight to get something done on the processor cores. Don't forget that other processes on the system and the operating system itself will also consume a bit of processing power.

If you are truly pressed for performance with a CPU-bound problem then matching the physical CPU cores is often the best solution, but if locking is a bottleneck, then a single thread can be faster than any multithreaded solution due to CPU boosting behavior.

If you do not expect to maximize all cores all the time, then I recommend not passing the `processes` parameter to `multiprocessing.Pool()`, which causes it to default to `os.cpu_count()`, which returns all cores including hyper-threaded ones.

It all depends on your use case, however, and the only way to know for certain is to test for your specific scenario. As a rule of thumb, I recommend the following:

- Disk I/O bound? A single process is most likely your best bet.
- CPU bound? The number of physical CPU cores is your best bet.
- Network I/O bound? Start with the defaults and tune if needed. This is one of the few cases where 128 threads on an 8-core processor can still be useful.
- No obvious bound but many (hundreds of) parallel processes are needed? Perhaps you should try `asyncio` instead of `multiprocessing`.

Note that the creation of multiple processes is not free in terms of memory and open files; while you could have a nearly unlimited number of coroutines, this is not the case for processes. Depending on your operating system configuration, you could reach the maximum open files limit long before you even reach 100 processes, and even if you reach those numbers, CPU scheduling will be your bottleneck instead.

So what should we do if our CPU cores are not enough? Simple: use more CPU cores. Where do we get those? Multiple computers! It is time to graduate to distributed computing.

Remote processes

So far, we have only executed our scripts on multiple local processors, but we can actually expand this much further. Using the `multiprocessing` library, it's actually very easy to execute jobs on remote servers, but the documentation is currently still a bit cryptic. There are actually a few ways of executing processes in a distributed way, but the most obvious one isn't the easiest one. The `multiprocessing.connection` module has both the `Client` and `Listener` classes, which facilitate secure communication between the clients and servers in a simple way.

Communication is not the same as process management and queue management, however; those features require some extra effort. The `multiprocessing` library is still a bit bare in this regard, but it's most certainly possible given a few different processes.

Distributed processing using multiprocessing

We will start with a module containing a few constants that should be shared between all clients and the server, so the secret password and the hostname of the server are available to all. In addition to that, we will add our prime calculation functions, which we will be using later. The imports in the following modules will expect this file to be stored as `T_17_remote_multiprocessing/constants.py`, but feel free to call it anything you like as long as the imports and references keep working:

```
host = 'localhost'
port = 12345
```

```
password = b'some secret password'
```

Next up, we define the functions that need to be available to both the server and the client. We will store this as `T_17_remote_multiprocessing/functions.py`:

```
def primes(n):
    for i, prime in enumerate(prime_generator()):
        if i == n:
            return prime

def prime_generator():
    n = 2
    primes = set()
    while True:
        for p in primes:
            if n % p == 0:
                break
        else:
            primes.add(n)
            yield n
        n += 1
```

Now it's time to create the actual server that links the functions and the job queue. We will store this as `T_17_remote_multiprocessing/server.py`:

```
import multiprocessing
from multiprocessing import managers

import constants
import functions

queue = multiprocessing.Queue()
manager = managers.BaseManager(address=('', constants.port),
                               authkey=constants.password)

manager.register('queue', callable=lambda: queue)
manager.register('primes', callable=functions.primes)

server = manager.get_server()
server.serve_forever()
```

After creating the server, we need to have a client script that sends the jobs. You could use a single script for both sending and processing, but to keep things sensible we will use separate scripts.

The following script will add `0` to `999` to the queue for processing. We will store this as `T_17_remote_multiprocessing/submitter.py`:

```
from multiprocessing import managers

import constants

manager = managers.BaseManager(
    address=(constants.host, constants.port),
    authkey=constants.password)
manager.register('queue')
manager.connect()

queue = manager.queue()
for i in range(1000):
    queue.put(i)
```

Lastly, we need to create a client to actually process the queue. We will store this as `T_17_remote_multiprocessing/client.py`:

```
from multiprocessing import managers

import functions

manager = managers.BaseManager(
    address=(functions.host, functions.port),
    authkey=functions.password)
manager.register('queue')
manager.register('primes')
manager.connect()

queue = manager.queue()
while not queue.empty():
    print(manager.primes(queue.get()))
```

From the preceding code, you can see how we pass along functions; the manager allows the registering of functions and classes that can be called from the clients as well. With that, we pass along a queue from the multiprocessing class, which is safe for both multithreading and multiprocessing.

Now we need to start the processes themselves. First, the server that keeps on running:

```
$ python3 T_17_remote_multiprocessing/server.py
```

After that, run the producer to generate the prime generation requests:

```
$ python3 T_17_remote_multiprocessing/submitter.py
```

Now we can run multiple clients on multiple machines to get the first 1,000 primes. Since these clients now print the first 1,000 primes, the output is a bit too lengthy to show here, but you can simply run this in parallel multiple times or on multiple machines to generate your output:

```
$ python3 T_17_remote_multiprocessing/client.py
```

Instead of printing, you can use queues or pipes to send the output to a different process if you'd like. As you can see, though, it's still a bit of work to process things in parallel and it requires some code synchronization to work. There are a few alternatives available, such as **Redis**, **ØMQ**, **Celery**, **Dask**, and **IPython Parallel**. Which of these is the best and most suitable depends on your use case. If you are simply looking for processing tasks on multiple CPUs, then `multiprocessing`, Dask, and IPython Parallel are probably your best choices. If you are looking for background processing and/or easy offloading to multiple machines, then ØMQ and Celery are better choices.

Distributed processing using Dask

The Dask library is quickly becoming the standard for distributed Python execution. It has very tight integration with many scientific Python libraries such as NumPy and Pandas, making parallel execution in many cases completely transparent. These libraries are covered in detail in *Chapter 15*, *Scientific Python and Plotting*.

The Dask library provides an easy parallel interface that can execute single-threaded, use multiple threads, use multiple processes, and even use multiple machines. As long as you keep the data-sharing limitations of multiple threads, processes, and machines in mind, you can easily switch between them to see which performs best for your use case.

Installing Dask

The Dask library consists of multiple packages and you might not need all of them. Broadly speaking, the `Dask` package is only the core, and we can choose from several extras, which can be installed through `pip install dask[extra]`:

- `array`: Adds an array interface similar to `numpy.ndarray`. Internally, these structures consist of multiple `numpy.ndarray` instances spread across your Dask cluster for easy parallel processing.
- `dataframe`: Similar to the array interface, this is a collection of `pandas.DataFrame` objects.
- `diagnostics`: Adds profilers, progress bars, and even a fully interactive dashboard with live information about the currently running jobs.
- `distributed`: Packages needed for running Dask across multiple systems instead of locally only.
- `complete`: All of the above extras.

For the demonstrations in this chapter, we will need to install at least the `distributed` extra, so you need to run either:

```
$ pip3 install -U "dask[distributed]"
```

Or:

```
$ pip3 install -U "dask[complete]"
```

If you are playing around with Jupyter notebooks, the progress bars in the `diagnostics` extra also have Jupyter support, which can be useful.

Basic example

Let's start with a basic example of executing some code via Dask without explicitly setting up a cluster. To illustrate how this can help performance, we will be using a `busy-wait` loop to maximize CPU load. In this case, we will be using the `dask.distributed` submodule, which has an interface quite similar to `concurrent.futures`:

```
import sys
import datetime

from dask import distributed

def busy_wait(n):
    while n > 0:
        n -= 1

def benchmark_dask(client):
    start = datetime.datetime.now()

    # Run up to 1 million
    n = 1000000
    tasks = int(sys.argv[1])  # Get number of tasks from argv

    # Submit the tasks to Dask
    futures = client.map(busy_wait, [n] * tasks, pure=False)
    # Gather the results; this blocks until the results are ready
    client.gather(futures)

    duration = datetime.datetime.now() - start
    per_second = int(tasks / duration.total_seconds())
    print(f'{tasks} tasks at {per_second} per '
          f'second, total time: {duration}')

if __name__ == '__main__':
    benchmark_dask(distributed.Client())
```

The code is mostly straightforward, but there are a few small caveats to look at. First of all, when submitting the task to Dask, you need to tell Dask that it is an impure function.

If you recall from *Chapter 5*, *Functional Programming – Readability Versus Brevity*, a pure function in functional programming is one that has no side effects; its output is consistent and only depends on the input. A function returning a random value is impure because repeated calls return different results.

Dask will automatically cache the results in the case of pure functions. If you have two identical calls, Dask will only execute the function once.

To queue the tasks, we need to use a function such as `client.map()` or `client.submit()`. These work in a very similar way to `executor.submit()` in the case of `concurrent.futures`.

Lastly, we need to fetch the results from the futures. This can be done by calling `future.result()`, or in batch by using `client.gather(futures)`. Once again, very similar to `concurrent.futures`.

To make the code a bit more flexible, we made the number of tasks configurable so it runs in a reasonable amount of time on your system. If you have a much slower or much faster system, you will want to adjust this to get useful results.

When we execute the script, we get the following results:

```
$ python3 T_18_dask.py 128
128 tasks at 71 per second, total time: 0:00:01.781836
```

That is how easily you can execute some code across all of your CPU cores. Naturally, we can also test in single-threaded or distributed mode; the only part we need to vary is how we initialize `distributed.Client()`.

Running a single thread

Let's run the same code but in single-threaded mode:

```
if __name__ == '__main__':
    benchmark_dask(distributed.Client())
```

Now if we run it, we can see that Dask was definitely using multiple processes before:

```
$ python3 T_19_dask_single.py 128
128 tasks at 20 per second, total time: 0:00:06.142977
```

This can be useful for debugging thread-safety issues. If the issues still persist in single-threaded mode, thread safety is probably not your issue.

Distributed execution across multiple machines

For a much more impressive feat, let's run the code on multiple machines at the same time. To run Dask on multiple systems simultaneously, there are many deployment options available:

- Manual setup using the `dask-scheduler` and `dask-worker` commands
- Automatic deployment over SSH using the `dask-ssh` command

- Deployment straight to an existing compute cluster running Kubernetes, Hadoop, and others
- Deployment to cloud providers such as Amazon, Google, and Microsoft Azure

In this case, we are going to use `dask-scheduler` since it's the solution that you can run on pretty much any machine that can run Python.

Note that you can encounter errors if the Dask versions and dependencies are not in sync, so updating to the latest version before starting is a good idea.

First, we start the `dask-scheduler`:

```
$ dask-scheduler
[...]
distributed.scheduler - INFO - Scheduler at:  tcp://10.1.2.3:8786
distributed.scheduler - INFO - dashboard at:                :8787
```

Once you have the `dask-scheduler` running, it will also host the dashboard mentioned above, which shows the current status: `http://localhost:8787/status`.

Now we can run the `dask-worker` processes on all machines that need to participate:

```
$ dask-worker --nprocs auto tcp://10.1.2.3:8786
```

With the `--nprocs` parameter, you can set the number of processes to start. With `auto`, it is set to the number of CPU cores including hyper-threading. When set to a positive number, it will start that exact number of processes; when set to a negative number the number is added to the number of CPU cores.

Your dashboard screen and the console should show all of the connected clients now. It's time to run our script again, but distributed this time:

```
if __name__ == '__main__':
    benchmark_dask(distributed.Client('localhost:8786'))
```

That's the only thing we need to do: configure where the scheduler is running. Note that we could also connect from other machines using the IP or hostname instead.

Let's run it and see if it became any faster:

```
$ python3 T_20_dask_distributed.py 2048
[...]
2048 tasks at 405 per second, total time: 0:00:05.049570
```

Wow, that's quite a difference! Instead of the `20` per second we could do in single-threaded mode or the `71` per second we could do in multiple-process mode, we can now process `405` of these tasks per second. As you can see, it also took very little effort to set up.

The Dask library has many more options to increase efficiency, limit memory, prioritize work, and more. We didn't even cover the combining of tasks by chaining them or running a `reduce` on bundled results. I can strongly recommend considering Dask if your code could benefit from running on multiple systems at the same time.

Distributed processing using ipyparallel

The IPython Parallel module, similar to Dask, makes it possible to process code on multiple computers at the same time. It should be noted that you can run Dask on top of `ipyparallel`. The library supports more features than you are likely to need, but the basic usage is important to know just in case you need to do heavy calculations that can benefit from multiple computers.

First, let's start by installing the latest `ipyparallel` package and all the IPython components:

```
$ pip3 install -U "ipython[all]" ipyparallel
```

Especially on Windows, it might be easier to install IPython using Anaconda instead, as it includes binaries for many science, math, engineering, and data analysis packages. To get a consistent installation, the Anaconda installer is also available for OS X and Linux systems.

Secondly, we need a cluster configuration. Technically, this is optional, but since we are going to create a distributed IPython cluster, it is much more convenient to configure everything using a specific profile:

```
$ ipython profile create --parallel --profile=mastering_python
[ProfileCreate] Generating default config file: '~/.ipython/profile_mastering_
python/ipython_config.py'
[ProfileCreate] Generating default config file: '~/.ipython/profile_mastering_
python/ipython_kernel_config.py'
[ProfileCreate] Generating default config file: '~/.ipython/profile_mastering_
python/ipcontroller_config.py'
[ProfileCreate] Generating default config file: '~/.ipython/profile_mastering_
python/ipengine_config.py'
[ProfileCreate] Generating default config file: '~/.ipython/profile_mastering_
python/ipcluster_config.py'
```

These configuration files contain a huge number of options, so I recommend searching for a specific section instead of walking through them. A quick listing gave me about 2,500 lines of configuration in total for these five files. The filenames already provide hints about the purpose of the configuration files, but we'll walk through the files explaining their purpose and some of the most important settings.

ipython_config.py

This is the generic IPython configuration file; you can customize pretty much everything about your IPython shell here. It defines how your shell should look, which modules should be loaded by default, whether or not to load a GUI, and quite a bit more. For the purpose of this chapter, it's not all that important, but it's definitely worth a look if you're going to use IPython more often. One of the things you can configure here is the automatic loading of extensions, such as `line_profiler` and `memory_profiler` discussed in *Chapter 12*, *Performance – Tracking and Reducing Your Memory and CPU Usage*. For example:

```
c.InteractiveShellApp.extensions = [
    'line_profiler',
```

```
    'memory_profiler',
]
```

ipython_kernel_config.py

This file configures your IPython kernel and allows you to overwrite/extend `ipython_config.py`. To understand its purpose, it's important to know what an IPython kernel is. The kernel, in this context, is the program that runs and introspects the code. By default, this is `IPyKernel`, which is a regular Python interpreter, but there are also other options such as `IRuby` or `IJavascript` to run Ruby or JavaScript respectively.

One of the more useful options is the possibility of configuring the listening port(s) and IP addresses for the kernel. By default, the ports are all set to use a random number, but it is important to note that if someone else has access to the same machine while you are running your kernel, they will be able to connect to your IPython kernel, which can be dangerous on shared machines.

ipcontroller_config.py

`ipcontroller` is the master process of your IPython cluster. It controls the engines and the distribution of tasks and takes care of tasks such as logging.

The most important parameter in terms of performance is the `TaskScheduler` setting. By default, the `c.TaskScheduler.scheme_name` setting is set to use the Python LRU scheduler, but depending on your workload, others such as `leastload` and `weighted` might be better. If you have to process so many tasks on such a large cluster that the scheduler becomes the bottleneck, there is also the `plainrandom` scheduler, which works surprisingly well if all your machines have similar specs and the tasks have similar durations.

For the purpose of our test, we will set the IP of the controller to `*`, which means that *all* IP addresses will be accepted and that every network connection will be accepted. If you are in an unsafe environment/network and/or don't have any firewalls that allow you to selectively enable certain IP addresses, then this method is *not* recommended! In such cases, I recommend launching through more secure options, such as `SSHEngineSetLauncher` or `WindowsHPCEngineSetLauncher`, instead.

Assuming your network is indeed safe, set the factory IP to all the local addresses:

```
c.HubFactory.client_ip = '*'
c.RegistrationFactory.ip = '*'
```

Now start the controller:

```
$ ipcontroller --profile=mastering_python
[IPControllerApp] Hub listening on tcp://*:58412 for registration.
[IPControllerApp] Hub listening on tcp://127.0.0.1:58412 for registration.
...
 [IPControllerApp] writing connection info to ~/.ipython/profile_mastering_
python/security/ipcontroller-client.json
```

```
[IPControllerApp] writing connection info to ~/.ipython/profile_mastering_
python/security/ipcontroller-engine.json
...
```

Pay attention to the files that were written to the security directory of the profile directory. They contain the authentication information that is used by `ipengine` to find and connect to the `ipcontroller`, such as the encryption keys and port information.

ipengine_config.py

`ipengine` is the actual worker process. These processes run the actual calculations, so to speed up the processing you will need these on as many machines as you have available. You probably won't need to change this file, but it can be useful if you want to configure centralized logging or need to change the working directory. Generally, you don't want to start the `ipengine` process manually since you will most likely want to launch multiple processes per computer. That's where our next command comes in, the `ipcluster` command.

ipcluster_config.py

The `ipcluster` command is actually just an easy shorthand to start a combination of `ipcontroller` and `ipengine` at the same time. For a simple local processing cluster, I recommend using this, but when starting a distributed cluster, it can be useful to have the control that the separate use of `ipcontroller` and `ipengine` offers. In most cases the command offers enough options, so you might have no need for the separate commands.

The most important configuration option is `c.IPClusterEngines.engine_launcher_class`, as this controls the communication method between the engines and the controller. Along with that, it is also the most important component for secure communication between the processes. By default it's set to `ipyparallel.apps.launcher.LocalControllerLauncher`, which is designed for local processes, but `ipyparallel.apps.launcher.SSHEngineSetLauncher` is also an option if you want to use SSH to communicate with the clients. Alternatively, there is `ipyparallel.apps.launcher.WindowsHPCEngineSetLauncher` for Windows HPC.

Before we can create the cluster on all machines, we need to transfer the configuration files. Your options are to transfer all the files or to simply transfer the files in your IPython profile's `security` directory.

Now it's time to start the cluster. Since we already started the `ipcontroller` separately, we only need to start the engines. On the local machine, we simply need to start it, but the other machines don't have the configuration yet. One option is copying the entire IPython profile directory, but the only file that really needs copying is `security/ipcontroller-engine.json`; after creating the profile using the profile creation command, that is. So unless you are going to copy the entire IPython profile directory, you need to execute the profile creation command again:

```
$ ipython profile create --parallel --profile=mastering_python
```

After that, simply copy the `ipcontroller-engine.json` file and you're done. Now we can start the actual engines:

```
$ ipcluster engines --profile=mastering_python -n 4
[IPClusterEngines] IPython cluster: started
[IPClusterEngines] Starting engines with [daemon=False]
[IPClusterEngines] Starting 4 Engines with LocalEngineSetLauncher
```

Note that the `4` here was chosen for a quad-core processor, but any number would do. The default will use the number of logical processor cores, but depending on the workload it might be better to match the number of physical processor cores instead.

Now we can run some parallel code from our IPython shell. To demonstrate the performance difference, we will use a simple sum of all the numbers from 0 to 10,000,000. Not an extremely heavy task, but when performed 10 times in succession, a regular Python interpreter takes a while:

```
In [1]: %timeit for _ in range(10): sum(range(10000000))
1 loops, best of 3: 2.27 s per loop
```

This time however, to illustrate the difference, we will run it 100 times to demonstrate how fast a distributed cluster is. Note that this is with only a three-machine cluster, but it's still quite a bit faster:

```
In [1]: import ipyparallel

In [2]: client = ipyparallel.Client(profile='mastering_python')
In [3]: view = client.load_balanced_view()
In [4]: %timeit view.map(lambda _: sum(range(10000000)), range(100)).wait()
1 loop, best of 3: 909 ms per loop
```

More fun, however, is the definition of parallel functions in `ipyparallel`. With just a simple decorator, a function is marked as parallel:

```
In [1]: import ipyparallel

In [2]: client = ipyparallel.Client(profile='mastering_python')
In [3]: view = client.load_balanced_view()
In [4]: @view.parallel()
   ...: def loop():
   ...:     return sum(range(10000000))
   ...:
In [5]: loop.map(range(10))
Out[5]: <AsyncMapResult: loop>
```

The `ipyparallel` library offers many more useful features, but that is outside the scope of this book. Even though `ipyparallel` is a separate entity from the rest of Jupyter/IPython, it does integrate well, which makes combining them easy enough.

Exercises

While preparing for multiple threads and/or multiple processes is less invasive than preparing for `asyncio` is, it still requires a bit of thought if you have to pass or share variables. So, this is really a question of how difficult you want to make it for yourself.

See if you can make an echo server and client as separate processes.Even though we did not cover `multiprocessing.Pipe()`, I trust you can work with it regardless. It can be created through `a, b = multiprocessing.Pipe()` and you can use it with `[a/b].send()` and `[a/b].recv()`.

- Read all files in a directory and sum the size of the files by reading each file using `threading` and `multiprocessing`, or `concurrent.futures` if you want an easier exercise. If you want an extra challenge, walk through the directories recursively by letting the thread/process queue new items while running.
- Create a pool of workers that keeps waiting for items to be queued through `multiprocessing.Queue()`. Bonus points if you make it a safe RPC (remote procedure call) type operation.
- Apply your functional programming skills and calculate something in a parallel way. Perhaps parallel sorting?

All of these exercises are unfortunately still easy compared to what you can experience in the wild. If you really want a challenge, start applying these techniques (especially memory sharing) to your existing or new projects and hope (or not) that you run into a real challenge.

Example answers for these exercises can be found on GitHub: `https://github.com/mastering-python/exercises`. You are encouraged to submit your own solutions and learn about alternative solutions from others.

Summary

We have covered many different topics in this chapter, so let's summarize them:

- What the Python GIL is, why we need it, and how we can work around it
- When to use threads, when to use processes, and when to use `asyncio`
- Running code in parallel threads using `threading` and `concurrent.futures`
- Running code in parallel processes using `multiprocessing` and `concurrent.futures`
- Running code distributed across multiple machines
- Sharing data between threads and processes
- Thread safety
- Deadlocks

The most important lesson you can learn from this chapter is that the synchronization of data between threads and processes is *really slow*. Whenever possible, you should only send data to the function and return once it is done, with nothing in between. Even in that case, if you can send less data, send less data. If possible, keep your calculations and data local.

In the next chapter, we will learn about scientific Python libraries and plotting. These libraries can help you perform difficult calculations and data processing in record time. These libraries are mostly highly optimized for performance and go great together with multiprocessing or the Dask library.

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:
`https://discord.gg/QMzJenHuJf`

15

Scientific Python and Plotting

The Python programming language is quite suited for scientific work. This is due to it being really easy to program for while being powerful enough to do almost anything you need. This combination has spawned a whole bunch of (very large) Python projects, such as `numpy`, `scipy`, `matplotlib`, `pandas`, and so on, over the years. While these libraries are all large enough to warrant entire books for themselves, we can offer a little insight into where and when they can be useful so you have an idea of where to start.

The major topics and libraries covered in this chapter are split into three sections:

- **Arrays and matrices:** NumPy, Numba, SciPy, Pandas, statsmodels, and xarray
- **Mathematics and precise calculations:** gmpy2, Sage, mpmath, SymPy, and Patsy
- **Plotting, graphing, and charting:** Matplotlib, Seaborn, Yellowbrick, Plotly, Bokeh, and Datashader

It is very likely that not all libraries in this chapter are relevant to you, so don't feel bad for not reading through all of it. However, I would recommend you at least look at the NumPy and Pandas sections briefly, as they are used heavily in the next chapter on machine learning.

Additionally, I would also recommend taking a look at the Matplotlib and Plotly sections, since those could be very useful in a wide range of scenarios.

Installing the packages

As is always the case with Python libraries that are built on C and other non-Python code, installing is very platform-dependent. On most platforms, thanks to binary wheels, we can simply do:

```
$ pip3 install <package>
```

For this and the next chapter, however, I would recommend an alternative solution instead. While some of the libraries, such as `numpy`, are easy to install on most platforms, some of the other libraries are more challenging. For this reason, I would recommend the use of either the **Anaconda** distribution or one of the **Jupyter Docker Stacks**.

The Jupyter Docker Stacks require you to have Docker working on your system, but if you do, it can be extremely easy to launch very complicated systems that would be near impossible to set up otherwise. The list of available stacks can be found here: `https://jupyter-docker-stacks.readthedocs.io/en/latest/using/selecting.html#core-stacks`.

A good starting point for this chapter is the `jupyter/scipy-notebook` stack, which includes a huge list of packages such as `numpy`, `scipy`, `numba`, `matplotlib`, `cython`, and many more. Running this image (assuming you have Docker running) is as easy as:

```
$ docker run -p 8888:8888 jupyter/scipy-notebook
```

After running the command, it will give you some information on how to open Jupyter in your browser.

Arrays and matrices

Matrices are at the heart of most scientific Python and artificial intelligence libraries because they are very convenient for storing a lot of related data. They are also suitable for really fast bulk processing, and calculations on them can be performed much faster than you could achieve with many separate variables. In some cases, these calculations can even be offloaded to the GPU for even faster processing.

Note that a 0D matrix is effectively a single number, a 1D matrix is a regular array, and there is no real limit to the number of dimensions you can use. It should be noted that both size and processing time quickly increase with multiple dimensions, of course.

NumPy – Fast arrays and matrices

The `numpy` package spawned most of the scientific Python development and is still used at the core of many of the libraries covered in this chapter and the next. The library is largely (where it matters, at least) written in C, which makes it extremely fast; we will see a few benchmarks later, but depending on the operation, it can easily be 100 times faster than pure Python for the CPython interpreter.

Since `numpy` has numerous features, we can only cover a few of the basics. But these already demonstrate how incredibly powerful (and fast) it is and why it is the basis for many of the other scientific Python packages in this chapter.

The core feature of the `numpy` library is the `numpy.ndarray` object. The `numpy.ndarray` object is implemented in C and offers a very fast and memory-efficient array. It can be represented as a single-dimension array or a multi-dimensional matrix with very powerful slicing features. You can store any Python object in one of these arrays, but to take full benefit of the power of `numpy`, you will need to use numbers such as integers or floating point numbers.

One important thing to note about `numpy` arrays is that they have a **fixed** size and cannot be resized because they reserve a contiguous block of memory. If you need to make them smaller or larger, you will need to create a new array.

Let's look at a few basic examples of how this array can be used and why it is very convenient:

```
# A commonly used shorthand for numpy is np
>>> import numpy as np

# Generate a list of numbers from 0 up to 1 million
>>> a = np.arange(1000000)
>>> a
array([     0,      1,      2, ..., 999997, 999998, 999999])

# Change the shape (still references the same data) to a
# 2-dimensional 1000x1000 array
>>> b = a.reshape((1000, 1000))
>>> b
array([[     0,      1,      2, ...,    997,    998,    999],
       [  1000,   1001,   1002, ...,   1997,   1998,   1999],
       ...,
       [998000, 998001, 998002, ..., 998997, 998998, 998999],
       [999000, 999001, 999002, ..., 999997, 999998, 999999]])

# The first row of the matrix
>>> b[0]
array([  0,   1,   2,   3, ..., 995, 996, 997, 998, 999])

# The first column of the matrix
>>> b[:, 0]
array([     0,   1000,   2000,   ..., 997000, 998000, 999000])

# Row 10 up to 12, the even columns between 20 and 30
>>> b[10:12, 20:30:2]
array([[10020, 10022, 10024, 10026, 10028],
       [11020, 11022, 11024, 11026, 11028]])

# Row 10, columns 5 up to 10:
>>> b[10, 5:10]
array([10005, 10006, 10007, 10008, 10009])

# Alternative syntax for the last slice
>>> b[10][5:10]
array([10005, 10006, 10007, 10008, 10009])
```

As you can see, the slicing options of `numpy` are very powerful, but what is even more useful about these slices is that they are all references/views instead of copies.

This means that if you modify the data in a slice, the original array will be modified as well. To illustrate using the array we created in the earlier examples:

```
>>> b[0] *= 10
>>> b[:, 0] *= 20

>>> a
array([     0,     10,     20, ..., 999997, 999998, 999999])
>>> b[0:2]
array([[    0,    10,    20, ...,  9970,  9980,  9990],
       [20000,  1001,  1002, ...,  1997,  1998,  1999]])
```

As you can see, after modifying the first row and the first column for each row, we now see that `a`, `b`, and consequently all slices of `a` and `b` have been modified; and all of that in a single operation instead of having to loop.

Let's try to run a simple benchmark to see how fast `numpy` can be at certain operations. If you are familiar with linear algebra, you undoubtedly know what a dot product is. If not, the dot product is an algebraic operation on two equal-length arrays of numbers, which are multiplied pair-wise and summed after. In mathematical terms, it looks like this:

$$a \cdot b = \sum_{i=0}^{n} a_i b_i = a_0 * b_0 + a_1 * b_1 + \cdots + a_n * b_n$$

It is a rather simple procedure and not that computationally heavy, but still something that is much faster when executed through `numpy`.

The goal of the dot product is to apply the growth of the second vector (array) onto the first vector. When applied to matrices, this can be used to move/rotate/scale a point or even an *n*-dimensional object. Simply put, if you have a 3D model stored in `numpy`, you can run a full transform on it using `numpy.dot`. Some examples of these operations can be found in my `numpy-stl` package: `https://pypi.org/project/numpy-stl/`.

Within this example, we will keep to the standard dot product of two 1-dimensional arrays, however.

To easily time the results, we will execute this from an IPython shell:

```
In [1]: import numpy

In [2]: a = list(range(1000000))
In [3]: b = numpy.array(a)

In [4]: def dot(xs, ys):
   ...:     total = 0
   ...:     for x, y in zip(xs, ys):
```

```
    ...:         total += x * y
    ...:     return total
    ...:

In [5]: %timeit dot(a, a)
78.7 ms ± 1.03 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
In [6]: %timeit numpy.dot(b, b)
518 µs ± 27.6 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

In this basic example, we can see that the pure Python version takes 78.7 ms and the `numpy` version takes 518 µs. That means that the `numpy` version is 150 times faster. Depending on what you are trying to do and on the size of the array, the advantage can be far greater.

To create an array, there are several options available, but the following are the most useful in my experience:

- `numpy.array(source_array)` creates an array from a different array (as we saw in the previous example).
- `numpy.arange(n)` creates an array with the given range. Effectively identical to `numpy.array(range(n))`.
- `numpy.zeros(n)` creates an array of size `n`, filled with zeros. It also supports tuples to create matrices: `numpy.zeros((x, y, z))`.
- `numpy.fromfunction(function, (x, y, z))` creates an array with the given shape using the given function. It should be noted that this function will be passed the index/indices of the current item, so the `x`, `y`, and `z` indices in this case.

The `numpy` library has many more useful functions, but at the very least it offers an array with nearly unbeatable performance and a very easy-to-use interface.

Numba – Faster Python on CPU or GPU

We already covered the basics of `numba` in *Chapter 12*, *Performance – Tracking and Reducing Your Memory and CPU Usage*. Combined with `numpy`, `numba` gets even more powerful because it natively supports functions that broadcast over `numpy` arrays (`numpy` calls these `ufuncs` or **universal functions**), similar to how the built-in `numpy` functions work. The only important difference between a regular `numba` function and one that supports `numpy` per-element processing is which decorator function you use. Normally you would use `numba.jit()`; for `numpy` per-element processing you need to use the `numba.vectorize(...)` decorator with the input and output types as parameters:

```
>>> import numpy
>>> import numba

>>> numbers = numpy.arange(500, dtype=numpy.int64)

>>> @numba.vectorize([numba.int64(numba.int64)])
```

```
... def add_one(x):
...     return x + 1

>>> numbers
array([  0,   1,   2, ..., 498, 499])

>>> add_one(numbers)
array([  1,   2,   3, ..., 499, 500])
```

Adding 1 is a useless example of course, but you can do anything you want here, which makes it very useful. The real point is how easy it is; as long as your function is purely functional (in other words, does not mutate external variables), it can be made extremely fast with very little effort. That is also the reason why several of the other libraries in this chapter heavily depend on `numba` for their performance.

As we specified `numba.vectorize([numba.int64(numba.int64)])`, our function will only accept a 64-bit integer and will return a 64-bit integer. To create a function that takes two 32- or 64-bit floats and returns a 64-bit integer, we would use the following:

```
@numba.vectorize([
    numba.int64(numba.float32, numba.float32),
    numba.int64(numba.float64, numba.float64),
])
```

In addition to the `numba.vectorize()` decorator, we have several other options available, such as the `numba.jitclass()` decorator for JIT-compiling an entire class, or the `numba.jit_module()` function to enhance an entire module.

SciPy – Mathematical algorithms and NumPy utilities

The `scipy` (Scientific Python) package contains a collection of mathematical algorithms for many different problems. The functions vary from signal processing to spatial algorithms to statistical functions.

Here's a list of some of the current sub-packages available in the `scipy` library (according to the `scipy` manual):

- `cluster`: Clustering algorithms such as *k*-means
- `fftpack`: Fast Fourier Transform routines
- `integrate`: Integration and ordinary differential equation solvers
- `interpolate`: Interpolation and spline smoothing functions
- `linalg`: Linear algebra functions such as linear equation solving
- `ndimage`: *N*-dimensional image processing
- `odr`: Orthogonal distance regression
- `optimize`: Optimization and root-finding routines

- `signal`: Signal processing functions such as peak finding and spectral analysis
- `sparse`: Sparse matrices and associated routines to save memory
- `spatial`: Spatial data structures and algorithms for triangulation and plotting
- `stats`: Statistical distributions and functions

As you can see, `scipy` features algorithms for a large range of topics and many of the functions are really fast, so it is definitely worth taking a look at.

With most of these topics, you can already guess by their names whether or not they apply to your use case, but there are a few that warrant a small example. So, let's look at one.

Sparse matrices

One of the most useful features of `scipy` (in my opinion, at least) is `scipy.sparse`. This module allows you to create sparse arrays, which can save you a huge amount of memory. While a `numpy` array takes roughly the amount of memory you are reserving, the `sparse` arrays only store the non-zero values or the non-zero blocks/rows/columns, depending on the type you choose. In the case of `numpy`, storing 1 million 64-bit integers takes 64 million bits or 8 megabytes.

Naturally, the advantage of a sparse array comes with a bunch of downsides, such as slower processing for certain operations or directions. The `scipy.sparse.csc_matrix` method, for example, produces sparse matrices that are really fast to slice in the column direction, but slow when slicing rows. Meanwhile, `scipy.sparse.csr_matrix` is the opposite.

Usage of sparse arrays is roughly as straightforward as a `numpy` array, but care has to be taken when selecting the specific sparse matrix type. The options are:

- `bsr_matrix(arg1[, shape, dtype, copy, blocksize])`: Block Sparse Row matrix
- `coo_matrix(arg1[, shape, dtype, copy])`: A sparse matrix in COOrdinate format.
- `csc_matrix(arg1[, shape, dtype, copy])`: Compressed Sparse Column matrix
- `csr_matrix(arg1[, shape, dtype, copy])`: Compressed Sparse Row matrix
- `dia_matrix(arg1[, shape, dtype, copy])`: Sparse matrix with DIAgonal storage
- `dok_matrix(arg1[, shape, dtype, copy])`: Dictionary Of Keys-based sparse matrix.
- `lil_matrix(arg1[, shape, dtype, copy])`: Row-based List-Of-Lists sparse matrix

If you only need something like a large identity matrix, this can be extremely useful. It is easy to construct and takes very little memory. The following two matrices are identical in contents:

```
>>> import numpy
>>> from scipy import sparse

>>> x = numpy.identity(10000)
>>> y = sparse.identity(10000)

>>> x.data.nbytes
800000000
```

```
# Summing the memory usage of scipy.sparse objects requires the summing
# of all internal arrays. We can test for these arrays using the
# nbytes attribute.
>>> arrays = [a for a in vars(y).values() if hasattr(a, 'nbytes')]

# Sum the bytes from all arrays
>>> sum(a.nbytes for a in arrays)
80004
```

As you can see here, the non-sparse version of the identity matrix (`x`) took 10,000 times more memory. In this case, it is 800 megabytes versus 80 kilobytes, but if you have a much larger matrix this quickly becomes impossible. Since the matrix grows in size quadratically (`n^2`; the matrix above has size 10,000x10,000=100,000,000) this can make a very dramatic difference. The sparse matrix (in this case, at least) grows linearly (`n`).

For smaller non-sparse arrays (up to a billion numbers) the memory usage is still workable and it would take about 8 gigabytes of memory for a billion 64-bit numbers, but when you go beyond that, most systems will quickly run out of memory. As is often the case, these memory savings do come at the cost of increased CPU time for many operations, so I would not recommend replacing all of your `numpy` arrays with sparse arrays.

In conclusion, `scipy` is a versatile and very useful module that supports a wide variety of calculations and algorithms. If `scipy` has an algorithm available for your goal, it is likely one of the fastest options you are going to find within the Python ecosystem. Many of the functions are very domain-specific, however, so you can probably guess which are (and are not) useful for you.

Pandas — Real-world data analysis

While the focus of `numpy`, `scipy`, and `sympy` is mostly mathematical, Pandas is focused more on real-world data analysis. With Pandas, you are generally expected to load data from some external source such as databases or CSV files. Once you have the data loaded, you can easily calculate statistics, visualize the data, or combine the data with other datasets.

To store data, Pandas offers two different data structures. The `pandas.Series` is a 1-dimensional array and the `pandas.DataFrame` is a 2-dimensional matrix where the columns can be labeled if needed. Internally these objects wrap a `numpy.ndarray`, so all `numpy` operations are still possible on these objects as well.

Why do we need Pandas on top of `numpy`? It all comes down to convenience, and Pandas offers several features on top of `numpy` that are beneficial for doing real-world data analysis:

- It can gracefully handle missing data. Within a `numpy` floating point number, you can store `NaN` (not a number), but not all `numpy` methods will handle that nicely without custom filtering.
- As opposed to the fixed-size `numpy.ndarray`, columns can be added and removed to a `numpy.DataFrame` as desired.

- It provides bundled data management functions to easily group, aggregate, or transform data. While you can easily modify `numpy` data, grouping data is a lot harder out of the box.
- It also provides utility functions for data containing time series, allowing you to easily apply moving window statistics and compare newer to older data with very little effort.

Let's create a simple example that stores the release dates of major Python releases with their versions. The data is sourced from Wikipedia, which has a nice table that we can quickly use and copy: `https://en.wikipedia.org/wiki/History_of_Python#Table_of_versions`.

For brevity, we are showing a shortened version of the code here, but you can copy/paste the full table from Wikipedia or look in the GitHub project for this book.

First, let's read the data into a dataframe:

```
# A commonly used shorthand for pandas is pd
>>> import re
>>> import io

>>> import pandas as pd

>>> data = '''
... Version\tLatest micro version\tRelease date\tEnd of full support\tEnd ...
... 0.9\t0.9.9[2]\t1991-02-20[2]\t1993-07-29[a][2]
... ...
... 3.9\t3.9.5[60]\t2020-10-05[60]\t2022-05[61]\t2025-10[60][61]
... 3.10\t\t2021-10-04[62]\t2023-05[62]\t2026-10[62]
... '''.strip()

# Slightly clean up data by removing references
>>> data = re.sub(r'\[.+?\]', '', data)

# df is often used as a shorthand for pandas.DataFrame
>>> df = pd.read_table(io.StringIO(data))
```

In this case, we have the entire table stored in `data` as a tab-separated string. Since that includes the references that Wikipedia uses, we use a regular expression to clean up everything that looks like `[...]`. Lastly, we read the data into a `pandas.DataFrame` object using `pandas.read_table()`. The `read_table()` function supports either a filename or a file handle and, since we have the data as a string, we're using `io.StringIO()` to convert the string to a file handle.

Now that we have the data, let's see what we can do with it:

```
# List the columns
>>> df.columns
Index(['Version', ..., 'Release date', ...], dtype='object')
```

```
# List the versions:
>>> df['Version']
0     0.9
...
25    3.9
26    3.1
Name: Version, dtype: float64

# Oops... where did Python 3.10 go in the output above? The
# conversion to float trimmed the 0 so we need to disable that.
>>> df = pd.read_table(io.StringIO(data), dtype=dict(Version=str))

# Much better, we didn't lose the version info this time
>>> df['Version']
0      0.9
...
25     3.9
26     3.10
Name: Version, dtype: object
```

Now that we know how to read the data from the table, let's see how we can do something more useful with it. This time we are going to convert it into a time series so we can do analysis based on dates/times:

```
# The release date is read as a string by default, so we convert
# it to a datetime:
>>> df['Release date'] = pd.to_datetime(df['Release date'])

>>> df['Release date']
0      1991-02-20
...
26     2021-10-04
Name: Release date, dtype: datetime64[ns]

# Let's see which month is the most popular for Python releases.
# First we run groupby() on the release month and after that we
# run a count() on the version:
>>> df.groupby([df['Release date'].dt.month])['Version'].count()
Release date
1     2
2     2
3     1
4     2
```

```
6     3
7     1
9     4
10    8
11    1
12    3
Name: Version, dtype: int64
```

While you could do all of this with plain `numpy`, it is certainly much more convenient with `pandas`.

Input and output options

One huge advantage of Pandas is the huge amount of readily available input and output options. Let's start by saying that this list will never be complete because you can easily implement your own method, or install a library to handle other types for you. We will see an example of this later in this chapter when we cover `xarray`.

At the time of writing, the `pandas` library natively supports a huge list of input and/or output formats:

- Common formats such as Pickle, CSV, JSON, HTML, and XML
- Spreadsheets such as Excel files
- Data formats used by other statistical systems such as HDF5, Feather, Parquet, ORC, SAS, SPSS, and Stata
- Many types of databases using SQLAlchemy

If your preferred format is not on the list, the odds are that you can easily find a converter for it. Alternatively, it is fairly easy to write a converter yourself as you can implement them in plain Python.

Pivoting and grouping

One very useful feature of Pandas is the ability to **pivot** and **unpivot** a DataFrame. When pivoting, we can convert rows to columns based on their values, effectively grouping them. The `pandas` library has several options to pivot/unpivot your data:

- `pivot`: Returns a reshaped pivot table without aggregation (e.g. sum/count/etc.) support
- `pivot_table`: Returns a pivot table with aggregation support
- `melt`: Reverses the operation of `pivot`
- `wide_to_long`: A simpler version of `melt` that can be more convenient to use

What can we achieve by pivoting? Let's create a very simple example of some temperature measurements in a long list, and pivot them so we get the days as columns instead of rows:

```
>>> import pandas as pd
>>> import numpy as np

>>> df = pd.DataFrame(dict(
...     building=['x', 'x', 'y', 'x', 'x', 'y', 'z', 'z', 'z'],
```

```
...     rooms=['a', 'a', 'a', 'b', 'b', 'b', 'c', 'c', 'c'],
...     hours=[10, 11, 12, 10, 11, 12, 10, 11, 12],
...
...     temperatures=np.arange(0.0, 9.0),
... ))

>>> df
  building rooms  hours  temperatures
0        x     a     10           0.0
1        x     a     11           1.0
...
7        z     c     11           7.0
8        z     c     12           8.0
```

The way this data is set up is similar to how a data logging tool would usually return it, with a single row for a single measurement. However, this is often not the most convenient way to read or analyze the data, and that is where pivoting can really help.

Let's look at the mean room temperature per hour:

```
>>> pd.pivot_table(
...     df, values='temperatures', index=['rooms'],
...     columns=['hours'], aggfunc=np.mean)
hours   10   11   12
rooms
a      0.0  1.0  2.0
b      3.0  4.0  5.0
c      6.0  7.0  8.0
```

That shows a row for each room and a column for each hour, with the values generated through `numpy.mean()`.

We can also get the mean room temperature per building, per room, per hour:

```
>>> pd.pivot_table(
...     df, values='temperatures', index=['building', 'rooms'],
...     columns=['hours'], aggfunc=np.mean)
hours            10   11   12
building rooms
x        a      0.0  1.0  NaN
         b      3.0  4.0  NaN
y        a      NaN  NaN  2.0
         b      NaN  NaN  5.0
z        c      6.0  7.0  8.0
```

As you can see, `pandas` handles missing values by showing `NaN` for the missing data and gives us a very nice aggregate result.

In addition to these pivoting features, Pandas provides a huge list of grouping functions that also allow you to aggregate results. The big advantage of the grouping feature over pivoting is that you can group over arbitrary ranges and functions. For time-based results, for example, you could choose to group per second, minute, hour, 5 minutes, or any other interval that might be useful to you.

As a basic example with the data above:

```
>>> df.groupby(pd.Grouper(key='hours')).mean()
       temperatures
hours
10              3.0
11              4.0
12              5.0
```

This example already shows how the `groupby` feature can be used, but the real power comes when combining it with timestamps. For instance, you could do `pd.Grouper(freq='5min')`.

Merging

Another extremely useful feature of Pandas is that you can merge data, similar to how you would join tables in a database. As is the case with pivoting, the `pandas` library has several join methods:

- `pandas.merge`: The `merge` function is pretty much the straight equivalent of a database join. It can do inner, outer, left, right, and cross joins, similar to many databases. Additionally, it can validate if the relations between the columns are correct (i.e. one-to-one, one-to-many, many-to-one, and many-to-many), in a similar way to how referential integrity in a database functions.
- `pandas.merge_ordered`: Similar to `merge` but allows for optional filling/interpolation using a function.
- `pandas.merge_asof`: This function does a left join on the nearest key instead of requiring an exact match.

The ability to easily merge multiple `DataFrame` objects is a really powerful feature that is invaluable when processing real-world data.

Rolling or expanding windows

In Pandas, windows can help you to efficiently run calculations on rolling subsets of (expanding) data. Naively calculating is of course possible, but that can be highly inefficient and infeasible for larger datasets. With a **rolling window**, you can have a running mean, sum, or other function on a fixed window size in an efficient manner.

To illustrate, let's assume you have an array with 100 items and you want to get the mean value using a window size of 10. The naïve solution would be to sum the first 10 items and divide them by 10, then repeat that for items 1 to 11, and so on.

For each of these, you would have to walk through all 10 items in the window. If we take `n` as the length of the array and `w` as the size of the window, this takes `O(n*w)` time. We can do much better if we keep track of the intermediate sum, however; if we simply add the next number and simultaneously remove the first number from our running sum, we can do the same in `O(n)` instead.

Let's illustrate how `pandas` can take care of this for us:

```
>>> import pandas as pd
>>> import numpy as np

>>> pd_series = pd.Series(np.arange(100))  # [0, 1, 2, ... 99]

>>> # Create a rolling window with size 10
>>> window = pd_series.rolling(10)
>>> # Calculate the running mean and ignore the N/A values at the
>>> # beginning before the window is full
>>> window.mean().dropna()
9      4.5
10     5.5
       ...
99    94.5
Length: 91, dtype: float64
```

The rolling window as we have seen above supports functions for count, sum, mean, median, variance, standard deviation, quantiles, and several more. If you need something special, you can also provide your own function.

There are a few extra features to these windows. Instead of having all items calculated with the same weight, you can also use **weighted windows** to vary the weight of the items so recent data becomes more relevant than older data. In addition to regular weighted windows, you can also opt for **exponentially weighted windows** to increase the effect even further.

Lastly, we also have **expanding windows**. With these, you get the result from the beginning of the dataset up to your current point. If you were to sum a series with values `1, 2, 3, 4, 5`, it would return `1, 3, 6, 10, 15`, with each item being the total sum from the beginning of the series up to that point.

To conclude, the `pandas` library is extremely useful for analyzing data from varying sources. Since it was built on top of `numpy` it is also extremely fast, which makes it very convenient for in-depth analysis.

If you ever have a large amount of data to process, or data from several different sources, give `pandas` a try and see if it can help you to sort it out.

Statsmodels — Statistical models on top of Pandas

Similar to how `scipy` builds on top of `numpy`, we have `statsmodels` that builds on top of `pandas`. Initially, it was part of the `scipy` package, but later split off and greatly improved.

The `statsmodels` library offers a host of statistical methods and plotting tools and can be used to create regression models, choice models, analysis of variance (ANOVA), forecasting, and more.

A quick example of a weighted least squares regression, which attempts to fit a line to a set of data points, can be applied like this:

```
# The common shorthand for statsmodels is sm
>>> import statsmodels.api as sm
>>> import numpy as np

>>> Y = np.arange(8)
>>> X = np.ones(8)

# Create the weighted-least-squares model
>>> model = sm.WLS(Y, X)

# Fit the model and generate the regression results
>>> fit = model.fit()

# Show the estimated parameters and the t-values:
>>> fit.params
array([3.5])
>>> fit.tvalues
array([4.04145188])
```

While it still requires some background knowledge about statistics to be able to apply this properly, it does show how easily you can do a regression with `statsmodels`.

An abbreviated list of the models and analysis types that are currently supported by `statmodels` from the `statsmodels` manual follows.

Regression and linear models:

- Linear regression
- Generalized linear models
- Generalized estimating equations
- Generalized additive models (GAMs)
- Robust linear models
- Linear mixed effects models
- Regression with discrete dependent variable
- Generalized linear mixed effects models
- ANOVA

Time series analysis:

- Generic time series analysis such as univariate and vector autoregressive models (ARs/VARs)
- Time series analysis by state space methods
- Vector autoregressions

Other models:

- Methods for survival and duration analysis
- Nonparametric methods
- Generalized method of moments
- Multivariate statistics

The actual list of supported features is quite a bit longer, but this should give you a good indication as to whether it is a useful library for you. If you are familiar with statistical models, you should be able to get started with `statsmodels` rather quickly and the package is well documented with great examples.

xarray — Labeled arrays and datasets

The `xarray` library is very similar to `pandas` and is also built on top of `numpy`. The main differences are that `xarray` is multi-dimensional, whereas `pandas` supports one-dimensional and two-dimensional data only, and it was created with the **netCDF** (**Network Common Data Form**) formats in mind. The netCDF formats are commonly used for scientific research data, which (as opposed to CSV files, for example) contain both the data and metadata such as variable labels, data descriptions, and documentation, allowing for easy use in a multitude of software.

The `xarray` library can easily work together with `pandas`, so for this example, we will re-use the data from our earlier `pandas` example. The other way around is also easily possible using the `to_dataframe()` method on an `xarray.DataArray` object (the standard `xarray` matrix object). In this example, we will assume that you still have the `df` variable available from the `pandas` example earlier:

```
# The common shorthand for xarray is xr
>>> import xarray as xr

>>> ds = xr.Dataset.from_dataframe(df)

# For reference, the pandas version of the groupby
# df.groupby([df['Release date'].dt.month])['Version'].count()

>>> ds.groupby('Release date.month').count()['Version']
<xarray.DataArray 'Version' (month: 10)>
array([2, 2, 1, 2, 3, 1, 4, 8, 1, 3])
Coordinates:
  * month    (month) int64 1 2 3 4 6 7 9 10 11 12
```

The syntax for the `groupby()` is slightly different from `pandas`, and less Pythonic (if you ask me) due to the use of strings over variables, but it essentially comes down to the same operation.

In the `pandas` version, the order of the `count()` and the `['Version']` can be swapped to be even more similar. That is, the following is also valid and returns the same results:

```
df.groupby([df['Release date'].dt.month]).count()['Version']
```

Additionally, for this use case, I would argue that the output of `xarray` is not all that readable, but it certainly isn't bad either. Often, you will have so many data points that you won't be too interested in the raw data anyway.

The real advantage to `xarray` over `pandas` (in my opinion, at least) is the support for multi-dimensional data. You can add as much as you want to the `Dataset` object:

```
>>> import xarray as xr
>>> import numpy as np

>>> points = np.arange(27).reshape((3, 3, 3))
>>> triangles = np.arange(27).reshape((3, 3, 3))
>>> ds = xr.Dataset(dict(
...     triangles=(['p0', 'p1', 'p2'], triangles),
... ), coords=dict(
...     points=(['x', 'y', 'z'], points),
... ))

>>> ds
<xarray.Dataset>
Dimensions:    (p0: 3, p1: 3, p2: 3, x: 3, y: 3, z: 3)
Coordinates:
    points     (x, y, z) int64 0 1 2 3 4 5 ... 21 22 23 24 25 26
Dimensions without coordinates: p0, p1, p2, x, y, z
Data variables:
    triangles  (p0, p1, p2) int64 0 1 2 3 4 ... 21 22 23 24 25 26
```

In this case, we only added the `triangles` and the `points`, but you can add as much as you want and you can use `xarray` to combine these so you can reference multi-dimensional objects easily. Data combination can be achieved through several methods such as concatenation, merging to combine multiple datasets into one, combining based on field values, through per-row updates, and others.

When it comes down to `pandas` versus `xarray`, I would recommend simply giving them both a try and seeing which is more convenient for your use case. The libraries are very similar in features and usability and both have their own advantages. The multi-dimensionality of `xarray` is a huge advantage over `pandas` if you need it, however.

If it's all the same to you then I would currently recommend `pandas` over `xarray`, simply because it is currently the most used of the two, which results in more documentation/blog posts/books being available.

STUMPY — Finding patterns in time series

The `stumpy` library offers several tools to automatically detect patterns and anomalies in your time series matrices. It is built upon `numpy`, `scipy`, and `numba` to provide great performance and gives you the possibility of employing GPU (video card) power as well, to process the data even faster.

Using `stumpy` you could, for example, automatically detect if a website is getting an abnormal number of visitors. One of the nice features of `stumpy` in this scenario is that, in addition to static matrices, you can also add more data in a streaming way, which allows you to do real-time analysis without too much overhead.

As an example, let's assume we have a list of temperatures for a living room thermostat and see if we can find any repeating patterns:

```
>>> import numpy as np
>>> import stumpy

>>> temperatures = np.array([22., 21., 22., 21., 22., 23.])

>>> window_size = 3

# Calculate a Euclidean distance matrix between the windows
>>> stump = stumpy.stump(temperatures, window_size)

# Show the distance matrix. The row number is the index in the
# input array. The first column is the distance; the next columns
# are the indices of the nearest match, the left match, and the
# right match.
>>> stump
array([[0.0, 2, -1, 2],
       [2.449489742783178, 3, -1, 3],
       [0.0, 0, 0, -1],
       [2.449489742783178, 1, 1, -1]], dtype=object)

# As we can see in the matrix above, the first window has a
# distance of 0 to the window at index 2, meaning that they are
# identical. We can easily verify that by showing both windows:

# The first window:
>>> temperatures[0:window_size]
```

```
array([22., 21., 22.])

# The window at index 2:
>>> temperatures[2:2 + window_size]
array([22., 21., 22.])
```

The observant among you may have noticed that this distance matrix only has `4` rows for `6` values instead of the traditional `n*n` (`6*6` in this case) distance matrix. This is in part because we use a window size of `3` and we only look at the number of windows (which is `n-window_size+1=4`). A larger part is due to `stumpy` storing only the closest pairs, resulting in only requiring `O(n)` space instead of the normal `O(n*n)`.

While you can do these types of analysis with plain `numpy` as well, `stumpy` uses a very smart algorithm and relies heavily on `numba` for faster processing, so if you can use the library, I would recommend it.

Mathematics and precise calculations

Python has a decent number of mathematical functions and features built in, but there are cases where you need more advanced features or something faster. In this section, we will discuss a few libraries that help by introducing many extra mathematical functions and/or increase mathematical precision and/or performance quite a bit.

First, let's discuss the options in the Python core libraries to store numbers and perform calculations with varying precision:

- `int`: To store whole numbers (e.g. `1, 2, 3`), we have the `int` object in Python. The `int` is directly translated into a C `int64` on most systems as long as it can fit within 64-bit. Outside of that, it is internally cast to a Python `long` type (not to be confused with a C `long`), which can be arbitrarily large. This allows for infinite accuracy but only works as long as you use whole numbers.
- `fractions.Fraction`: The `Fraction` object makes it possible to store fractional numbers (for example, `1/2`, `1/3`, `2/3`) and they are infinitely precise since they rely on two `int` (or `long`) objects internally as the denominator and the numerator. However, these only work if the number you are trying to store can be represented as a fraction. Irrational numbers such as Pi cannot be represented this way.
- `float`: Floating point numbers make it really easy to store numbers that include decimals (for example `1.23`, `4.56`). These numbers are generally stored as a 64-bit floating point, which is a combination of a sign (1 bit positive or negative), exponent (11 bits), and a fraction (52 bits), resulting in the following equation:$(-1)^{exponent} * (2^{exponent}) * 1.fraction$. This means that a number such as 0.5 is stored using fraction 0 and exponent -1, resulting in: $2^{-1} * 1.0 = 0.5$. In the case of 0.5, this can be stored perfectly; in many other cases, this is problematic because not every number can be accurately described like this, which causes floating point inaccuracies.
- `decimal.Decimal`: The `Decimal` object allows for calculations with an arbitrary but specified precision. You can choose how many decimals you want, but it is not all that fast.

Several of the following libraries offer solutions to enhance the precision of your calculations.

gmpy2 – Fast and precise calculations

The `gmpy2` library uses libraries that are written in C for really fast high-precision calculations. On Linux/Unix systems it will rely on GMP (hence the name); on Windows it will use MPIR, which is based on GMP. Additionally, the MPFR and MPC libraries are used for correctly rounding floating point real and complex numbers respectively. Lastly, it uses `mpz_lucas` and `mpz_prp` for really fast primality testing.

Here's a tiny example on how to get Pi to 1000 places, which you can't easily do with the Python core library:

```
>>> import gmpy2

>>> gmpy2.const_pi(1000)
mpfr('3.14159265358979...33936072602491412736',1000)
```

This library is invaluable if you need fast and high-precision calculations.

For my personal use case, the `gmpy` library (`gmpy2` didn't exist yet at that time) has been extremely helpful when competing in the fun online math challenge project called Project Euler: `https://projecteuler.net/`.

Sage – An alternative to Mathematica/Maple/MATLAB

If you have ever taken an advanced math class in college or university, chances are that you have encountered software such as Mathematica, Maple, MATLAB, or Magma. Or perhaps you have used WolframAlpha, which is built on Mathematica. The Sage project is meant as a free and open source alternative to those really expensive software packages.

For reference, at the time of writing, the basic Mathematica Home edition, which can only run at 4 CPU cores at the same time, costs 413 euros (487 US dollars).

The Sage package can be used to solve equations both numerically and exactly, plot charts, and perform many other tasks from the Sage interpreter. Similar to IPython and Jupyter, Sage offers its own interpreter with a custom language so it feels closer to mathematical packages such as Mathematica. Naturally, you could import the Sage code from regular Python as well.

A small example of solving for a variable using Sage with the Sage interpreter:

```
sage: x, y, z = var('x, y, z')
sage: solve([x + y == 10, x - y == 5, x + y + z == 1], x, y, z)
[[x == (15/2), y == (5/2), z == -9]]
```

In this case, we asked Sage to solve an equation with three variables for us given the following constraints:

$$x + y = 10$$
$$x - y = 5$$
$$x + y + z = 1$$

According to Sage (correctly), this results in:

$$x = \frac{15}{2}, y = \frac{5}{2}, z = -9$$

If you are looking for a full-fledged mathematical software system (or some features of one), Sage is a good option.

mpmath — Convenient, precise calculations

The `mpmath` library is an all-round mathematical library offering functions for trigonometry, calculus, matrices, and many others while still maintaining a configurable precision.

Installing `mpmath` is really easy since it is pure Python and has no required dependencies, but it does offer speedups using Sage and `gmpy2` if they are available. This combines the benefits of the Sage and `gmpy2` libraries for speed with the convenience of a pure Python installation if those are not available.

Let's illustrate the advantages of configurable precision versus regular floating point numbers in Python:

```
>>> N = 10
>>> x = 0.1

# Regular addition
>>> a = 0.0
>>> for _ in range(N):
...     a += x

>>> a
0.9999999999999999

# Using sum, the same result as addition
>>> sum(x for _ in range(N))
0.9999999999999999
```

As you can see, both regular addition and `sum()` are both inaccurate. Python does have a better method available for this specific problem:

```
# Sum using Python's optimized fsum:
>>> import math
```

```
>>> math.fsum(x for _ in range(N))
1.0
```

When it comes to the general case, however, floating point math will always be inaccurate and at times that can be problematic. So, if your calculations do require floating point math but you want more accuracy, `mpmath` can help you:

```
>>> import mpmath

# Increase the mpmath precision to 100 decimal places
>>> mpmath.mp.dps = 100
>>> y = mpmath.mpf('0.1')

# Using mpmath with addition:
>>> b = mpmath.mpf('0.0')
>>> for _ in range(N):
...     b += y

>>> b
mpf('1.000000000000000000000000000...0000000000000000000000014')

# Or a regular sum with mpmath:
>>> sum(y for _ in range(N))
mpf('1.000000000000000000000000000...0000000000000000000000014')
```

While these results are obviously still not perfect (you would assume the result to be 1.0, like what `math.fsum()` produced), it can help to reduce floating point errors a lot more. Make sure to feed `mpmath` strings or integers, otherwise your variable can already introduce floating point errors. If we had used `x` in the sum instead of `y`, it would have resulted in floating point inaccuracy similar to regular Python math.

Naturally, `fpmath` can do a lot more than simply reduce your floating point errors, such as plotting and calculus, but I will leave that for you to explore. If you are looking for solutions to mathematical problems, this library should be on your list.

SymPy — Symbolic mathematics

The `sympy` module is a library that you might never need, but it is such a great library that it should be covered. The goal of `sympy` is to be a fully featured **Computer Algebra System** (**CAS**) so you can manipulate mathematical expressions similar to how you would do so on paper.

Let's start with a little demo on how we can express and solve an integral using `sympy`:

```
>>> from sympy import *

>>> init_printing(use_unicode=True)
```

```
>>> x, y, z = symbols('x y z')

>>> integral = Integral(x * cos(x), x)
>>> integral
⌠
⎮ x cos(x) dx
⌡
>>> integral.doit()
x sin(x) + cos(x)
```

Apologies if this gave you horrible flashbacks to some calculus exam, but I think it is amazing to be able to do this. This code first imports `sympy` using a wildcard because the equations would quickly become unreadable if all functions needed to be prefixed by `sympy`.

After that, we use the `init_printing()` function with the Unicode flag enabled to tell `sympy` that our shell supports Unicode characters. This allows for pretty rendering of many mathematical formulas, but certainly not all of them. The alternatives to this are basic ASCII rendering (as you can imagine, this does not look too pretty for an integral), and LaTeX output, which can render as images (for example, when using Jupyter). There are actually several other rendering modes available, but they greatly depend on your environment so we will not be getting into those.

Because you can use any variable name in an equation, we need to specifically declare `x`, `y`, and `z` as variables. Even though we only use `x` in this case, you will often need the others as well, so why not declare them in advance?

Now we use the `Integral` function to declare the integral. Due to font limitations, the example above is not perfect, but the rendered integral should look like this in your shell or browser:

$$\int x \cos x \, dx$$

Lastly, we tell `sympy` to solve the integral using the `doit()` method. This correctly results in the equation:

$$x \sin x + \cos x$$

The only nitpick I have here is that `sympy` omits the integration constant by default. Ideally, it would include the `+ C`.

If you're looking to represent (and solve) an equation, `sympy` can certainly help. I personally think it is a really great library even though I have very little use for it.

Patsy — Describing statistical models

Similar to how `sympy` can describe mathematical formulas in Python, `patsy` can describe statistical models, which makes it go hand in hand with the `statsmodels` package. It can also use regular Python functions or directly apply `numpy`:

```
>>> import patsy
>>> import numpy as np
```

```
>>> array = np.arange(2, 6)

>>> data = dict(a=array, b=array, c=array)
>>> patsy.dmatrix('a + np.square(b) + np.power(c, 3)', data)
DesignMatrix with shape (4, 4)
  Intercept  a  np.square(b)  np.power(c, 3)
          1  2             4               8
          1  3             9              27
          1  4            16              64
          1  5            25             125
  Terms:
    'Intercept' (column 0)
    'a' (column 1)
    'np.square(b)' (column 2)
    'np.power(c, 3)' (column 3)
```

In this example, we created a `numpy` array with a range from 2 to 6 and passed this to the `patsy.dmatrix()` function under the names `a`, `b`, and `c`, since duplicate names will be ignored. After that, we created the matrix using `patsy`; as you can see, the `+` in the `patsy` language tells it to add a new column. Those columns can be plain columns such as `a`, but they can also call functions such as `np.square(b)`.

If you are familiar with the mathematics behind vectors and matrices, this library might feel very natural to you. At the very least, it can be a slightly more obvious way to declare how your data interacts.

Plotting, graphing, and charting

Being able to read, process, and write data is important, of course, but to understand the meaning of data it is often far more convenient to create a plot, graph, or chart. As the old adage goes: "A picture is worth a thousand words."

If you have experience with any of the libraries mentioned earlier in this chapter, you may know that many of them have options for graphical output. In (almost?) all cases, however, this is not really a built-in feature but a convenient shortcut to an external library such as `matplotlib`.

As is the case with several of the libraries mentioned in this chapter, there are multiple libraries with similar features and possibilities, so this is certainly not an exhaustive list. To make visual plotting easier, for these examples we will mostly rely on `jupyter-notebook` with the use of the `ipywidgets` to create interactive samples. As always, the code (in these cases, the `jupyter-notebooks`) can be found on GitHub at `https://github.com/mastering-python/code_2`.

Matplotlib

The `matplotlib` library is the reliable standard for plotting and is supported by many of the scientific libraries in this chapter.

Most of the libraries mentioned earlier in this chapter either explain how `matplotlib` can be used with the library, or even have utility functions to facilitate plotting with `matplotlib`.

Does this mean that the `matplotlib` library is the gold standard for plotting? As usual, it depends. While `matplotlib` is certainly the most used scientific plotting Python library with a huge array of features, it is not always the most beautiful option. That doesn't mean you cannot configure it to be pretty, but out of the box, the library focuses on easy-to-read, consistent results and works for everyone and all scenarios. Some of the prettier libraries might look fantastic on a web page and have very useful interactive features but are not that suited for publishing and printing.

The basic example is trivially easy:

```
# The common shorthand for pyplot is plt
import matplotlib.pyplot as plt
import numpy as np

# Enable in-line rendering for the Jupyter notebook
%matplotlib inline

a = np.arange(100) ** 2
plt.plot(a)
```

Effectively, we only need `plt.plot()` to plot a basic chart:

```
Out[1]: [<matplotlib.lines.Line2D at 0x112a51700>]
```

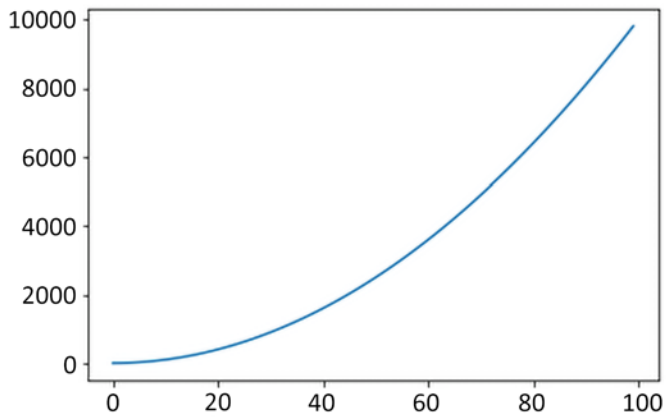


Figure 15.1: Matplotlib plot

This simple example was very easy to plot, but `matplotlib` can do so much more. Let's take a look at how we can combine a few graphs and make the plot interactive using `ipywidgets`:

```
%matplotlib notebook

import matplotlib.pyplot as plt
```

```
import numpy as np
import ipywidgets as widgets

# Using interact, we create 2 sliders here for size and step.
# In this case we have size which goes from 1 to 25 with increments
# of 1, and step, which goes from 0.1 to 1 with increments of 0.1
@widgets.interact(size=(1, 25, 1), step=(0.1, 1, 0.1))
def plot(size, step):
    # Create a matplotlib figure
    # We will render everything onto this figure
    fig = plt.figure()

    # Add a subplot. You could add multiple subplots but only one will
    # be shown when using '%matplotlib notebook'
    ax = fig.add_subplot(projection='3d')

    # We want X and Y to be the same, so generate a single range
    XY = np.arange(-size, size, step)

    # Convert the vectors into a matrix
    X, Y = np.meshgrid(XY, XY)

    R = np.sqrt(X**2 + Y**2)

    # Plot using sine
    Z = np.sin(R)
    ax.plot_surface(X, Y, Z)

    # Plot using cosine with a Z-offset of 10 to plot above each other
    Z = np.cos(R)
    ax.plot_surface(X, Y, Z + 10)
```

This function generates the following figure:

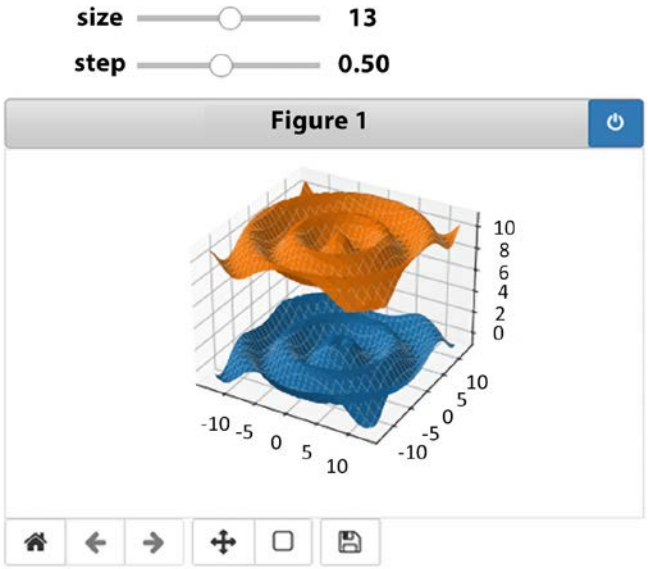


Figure 15.2: Matplotlib in Jupyter Notebook with adjustable sliders

With a combination of `jupyter-notebook` and `matplotlib`, we can create interactive plots. If you run this in your own browser, not only can you drag the 3D plot around and view it from all sides, but you can also modify the `size` and `step` parameters by dragging the sliders.

With regard to actual plot types supported by `matplotlib`, there are really too many options and variations to list here, but if you are looking for any type of chart, graph, or plot, you are likely to find a solution using `matplotlib`. Additionally, many of the scientific Python libraries natively support it, which makes it an easy choice. This short section really does not do justice to the depth and features of `matplotlib`, but fear not – we are far from done with it as it, is the basis of a few other plotting libraries in this chapter.

Seaborn

The `seaborn` library is related to `matplotlib` in a similar way to how `statsmodels` works on top of `pandas`. It provides an interface for `matplotlib` with a strong focus on statistical data. The major feature of `seaborn` is that it makes it really easy to automatically generate an entire grid of plots.

Additionally, which is very convenient for our examples, `seaborn` comes with some test data so we can show fully fledged demonstrations based on real data. To illustrate, let's look at how easily we can create a very elaborate set of plots:

```
%matplotlib notebook

import seaborn as sns

sns.pairplot(
    # Load the bundled Penguin dataset
    sns.load_dataset('penguins'),
    # Show a different "color" for each species
    hue='species',
    # Specify the markers (matplotlib.markers)
    markers=['o', 's', 'v'],
    # Gray was chosen due to the book being printed in black and white
    palette='Greys',
    # Specify which rows and columns to show. The default is to show all
    y_vars=['body_mass_g', 'flipper_length_mm'],
    x_vars=['body_mass_g', 'flipper_length_mm', 'bill_length_mm'])
```

This produces the following set of plots:

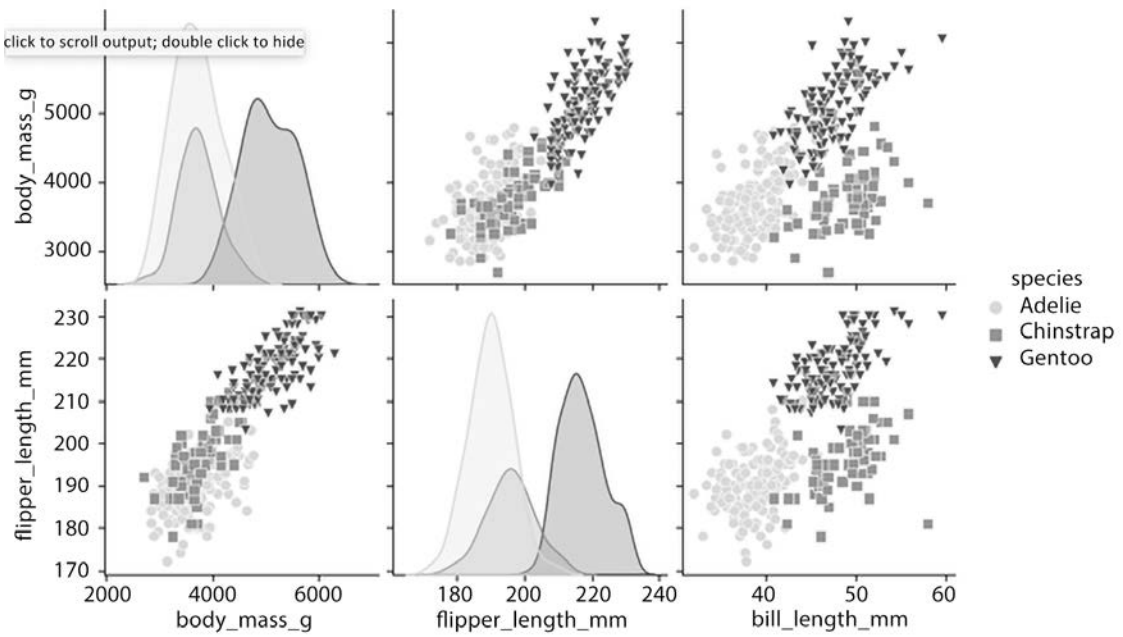


Figure 15.3: Seaborn pairplot render

While this still seems like a very elaborate call, you could actually get away with just using `sns.pairplot(df)` to get great results. Without the `hue=...` parameter, the results will not be split by species, however.

The `seaborn` library has support for many types of plots:

- Relational plots such as line plots and scatter plots
- Distribution plots such as histograms
- Categorical plots such as box plots
- Matrix plots such as heatmaps

The `seaborn` library also has many shortcuts for creating sets of plots or automatically processing the data using algorithms such as kernel density estimation.

If you are looking for a nice-looking plotting library, `seaborn` is a very good option, especially due to the multi-plot grid features. The list of plots above are all specific plots, but as we saw with `pairplot`, `seaborn` can generate an entire grid of plots in just a single line of code, which is extremely useful. You could do the same with `matplotlib` directly, but it would probably take you a few dozen lines of code.

Yellowbrick

As is the case with `seaborn`, `yellowbrick` is also built on top of `matplotlib`. The difference is that `yellowbrick` is focused on visualizing machine learning results and depends on the scikit-learn (`sklearn`) machine learning library. The scikit-learn integration is also what makes this library very powerful in those scenarios; it natively understands the scikit-learn data structures so it can easily plot them for you with almost no configuration. In the next chapter, we will see more on scikit-learn.

This example, straight from the `yellowbrick` manual, shows how you can visualize a regression in effectively a single line of code:

```
%matplotlib notebook

from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split as tts

from yellowbrick.datasets import load_concrete
from yellowbrick.regressor import residuals_plot

# Load the dataset and split into train/test (pandas.DataFrame) splits
X, y = load_concrete()

X_train, X_test, y_train, y_test = tts(X, y, test_size=0.2, shuffle=True)

# Create the visualizer, fit, score, and show it
viz = residuals_plot(RandomForestRegressor(), X_train, y_train, X_test, y_test)
```

This generates the following scatter plot:

Figure 15.4: Yellowbrick regression plot

These kinds of shortcut functions make it really easy to generate usable output and work on your regression instead of having to worry about how to properly plot the data. In addition to plotting regressions, `yellowbrick` has many more visualizers organized by analysis type. Similar to `seaborn`, `yellowbrick` can take care of not only the plotting but also the calculations and analysis for you.

The `yellowbrick` library has functions for many types of analysis such as:

- **Feature visualization:** Displaying features as scatter plots, detecting relationships and ranking them, creating a circular plot of related features, and so on
- **Classification visualization:** Displaying the thresholds, precision, and the error prediction for classifications as line, area, or matrix plots
- **Regression visualization:** Displaying a scatter or a combination of scatter plots and histograms
- **Cluster visualization:** Displaying maps to visualize the distance between the clusters
- **Model selection visualization:** Displaying the learning curve through a combination of lines and area or showing the feature importance as a bar chart

The `yellowbrick` library is currently the most convenient option for visualizing scikit-learn output, but most of the charting options also apply to other data types such as `pandas.DataFrame` objects, so it's worth taking a look if `seaborn` does not suit your needs.

Plotly

The `plotly` library supports a lot of different types of plots and even has native support for controls such as sliders, so you can change parameters when viewing from a web browser. Additionally, similar to how `seaborn` makes usage of `matplotlib` much easier in some cases, `plotly` also includes Plotly Express (often denoted as `px`), which makes usage trivially easy.

To illustrate how easy Plotly Express can be, let's try to replicate the plots we made with `seaborn`:

```
import seaborn as sns
import plotly.express as px

fig = px.scatter_matrix(
    # Load the Penguin dataset from seaborn
    sns.load_dataset('penguins'),
    # Show a different "color" for each species
    color='species',
    # Specify that the symbols/markers are species-dependent
    symbol='species',
    # Specify which rows and columns to show. The default is to show all
    dimensions=['body_mass_g', 'flipper_length_mm', 'bill_length_mm'],
)
fig.show()
```

Here is the result:

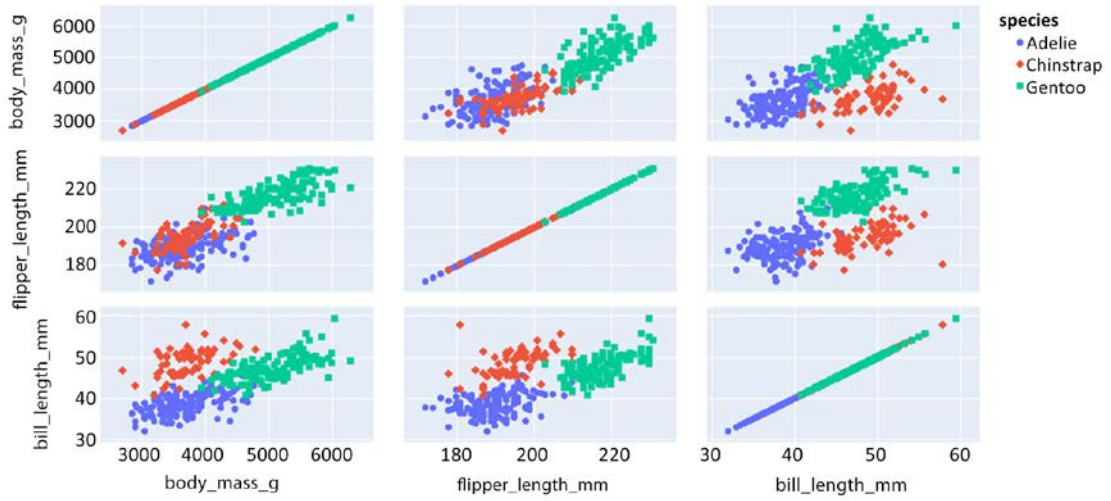


Figure 15.5: Plotly Express example output

While I would argue that the `seaborn` output is slightly prettier in this specific case, it does show just how easy it is to create useful plots using Plotly Express.

You might be wondering how easy or difficult it is to use the regular `plotly` API, as opposed to Plotly Express. For that, let's see if we can replicate the 3D `matplotlib` render:

```
import plotly
import numpy as np
import ipywidgets as widgets
import plotly.graph_objects as go

# Using interact, we create 2 sliders here for size and step.
# In this case we have size which goes from 1 to 25 with increments
# of 1, and step, which goes from 0.1 to 1 with increments of 0.1
@widgets.interact(size=(1, 25, 1), step=(0.1, 1, 0.1))
def plot(size, step):
    # Create a plotly figure, we will render everything onto this figure
    fig = go.Figure()

    # We want X and Y to be the same, so generate a single range
    XY = np.arange(-size, size, step)

    # Convert the vectors into a matrix
    X, Y = np.meshgrid(XY, XY)

    R = np.sqrt(X**2 + Y**2)

    # Plot using sine
    Z = np.sin(R)
    fig.add_trace(go.Surface(x=X, y=Y, z=Z))

    # Plot using cosine with a Z-offset of 10 to plot above each other
    Z = np.cos(R)
    fig.add_trace(go.Surface(x=X, y=Y, z=Z + 10))
    fig.show()
```

Here's the final result with the two cosines plotted in 3D:

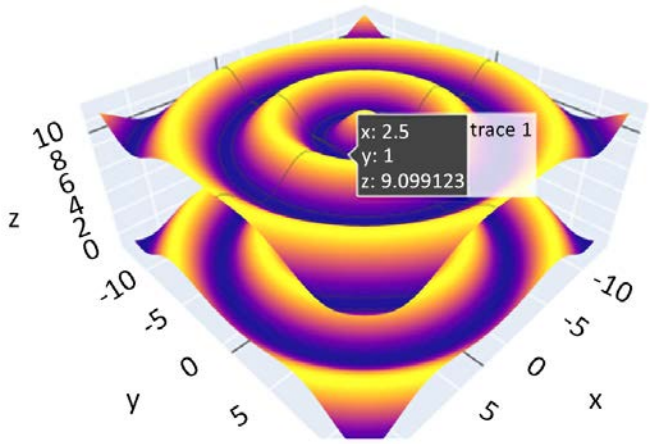


Figure 15.6: 3D plot using plotly

This is pretty much identical to `matplotlib`, and I would argue that it's even slightly better due to being even more interactive (which the book cannot effectively show, unfortunately). By default, `plotly` features a very useful display of the values when you hover with the mouse and allows for really easy zooming and filtering interactively.

When it comes to the choice between `matplotlib` and `plotly`, I would recommend looking at your specific use case. I think `plotly` is slightly easier and more convenient to use, but `matplotlib` is deeply integrated with many scientific Python libraries, which makes it a very convenient option. As always, opinions vary, so make sure to take a look at both.

Bokeh

The `bokeh` library is a beautiful and powerful visualization library with a strong focus on interactive visualizations in web browsers. Being able to make plots interactive can be extremely useful for analyzing the results. Instead of having to create multiple plots in a grid as we saw with `seaborn`, you can use a single grid and filter interactively. As this is a book, however, we cannot really demonstrate the full power of `bokeh`.

Before we get started with some examples, we need to talk about the two ways you can use `bokeh`. Effectively it comes down to **static** versus **dynamic**, where the static version uses a static snapshot of all data shown and the dynamic version loads data on demand.

The static version is similar to how `matplotlib` and most plotting libraries work: all data is contained in a single image or on a single web page without loading external resources. This works great for many cases, but not all.

What if you have a *lot* of data? A nice example of a visualization like this is Google Earth. You could never realistically download all of the data from Google Earth onto your computer (according to some estimates, currently over 100 petabytes of data), so you need to load it as you move around the map. For this purpose, `bokeh` has a server built in so the visualization can dynamically load the results as you filter. For the purpose of this book that makes little sense because it will be static in all cases, but we can show examples of both.

First, let's create a very basic plot:

```
import numpy as np

from bokeh.plotting import figure, show
from bokeh.io import output_notebook

# Load all javascript/css for bokeh
output_notebook()

# Create a numpy array of length 100 from 0 to 4 pi
x = np.linspace(0, 4*np.pi, 100)

# Create a bokeh figure to draw on
p = figure()
# Draw both a sine and a cosine
p.line(x, np.sin(x), legend_label='sin(x)', line_dash='dotted')
p.line(x, np.cos(x), legend_label='cos(x)')

# Render the output
show(p)
```

From this, we get the sine and cosine rendered as lines:

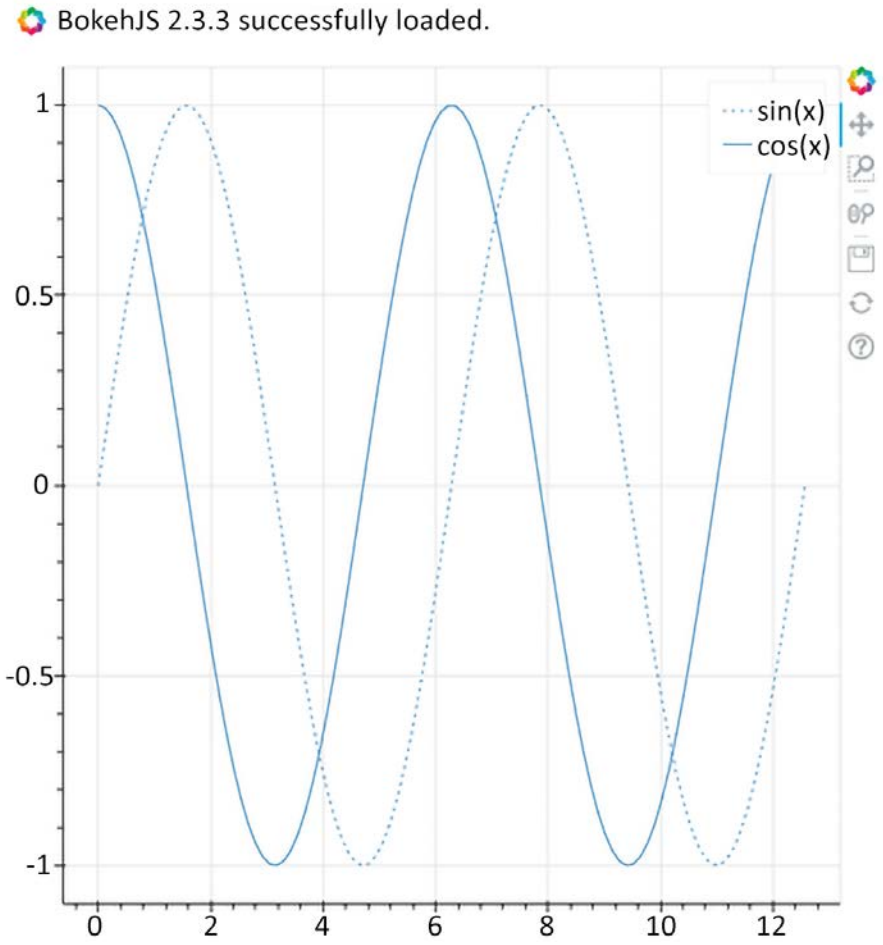


Figure 15.7: Bokeh basic render

As you can see, rendering basic *x*/*y* data as a line is really easy and does not look too different from the `matplotlib` output. If you look carefully, however, you might also notice the buttons on the right. These are what `bokeh` calls **tools**, and they can be used for zooming by either scrolling or by drawing a rectangle around what you wish to see. Panning can be done by dragging the image. It is also possible to save the render as an image file. If desired, you can create tooltips that respond to mouse clicks or mouse hovers.

Now let's see if we can recreate a more advanced plot like the one we made with `seaborn`:

```
import numpy as np
import seaborn as sns

from bokeh.plotting import figure, show
from bokeh.io import output_notebook
from bokeh.layouts import gridplot
from bokeh.transform import factor_cmap, factor_mark

output_notebook()

# Load the seaborn penguin dataset (pandas.DataFrame)
penguins = sns.load_dataset('penguins')
# Get the unique list of species for the marker and color mapping
species = penguins['species'].unique()
# Specify the marker list which will be mapped to the 3 species
markers = ['circle', 'square', 'triangle']
# Create a list of rows so we can build the grid of plots
rows = []

for y in ('body_mass_g', 'flipper_length_mm'):
    row = []
    rows.append(row)

    for x in ('body_mass_g', 'flipper_length_mm', 'bill_length_mm'):
        # Create a figure with a fixed size and pass along the labels
```

```
            p = figure(width=250, height=250,
                x_axis_label=x, y_axis_label=y)
            row.append(p)

            if x == y:
                # Calculate the histogram using numpy and make sure to drop
                # the NaN values
                hist, edges = np.histogram(penguins[x].dropna(), bins=250)
                # Draw the histograms as quadrilaterals (rectangles)
                p.quad(top=hist, bottom=0, left=edges[:-1], right=edges[1:])
            else:
                # Create a scatter-plot
                p.scatter(
                    # Specify the columns of the dataframe to show on the
                    # x and y axis
                    x, y,
                    # Specify the datasource, the pandas.DataFrame is
                    # natively supported by bokeh
                    source=penguins,
                    # Specify the column that contains the legend data
                    legend_field='species',
                    # Map the species onto our list of markers
                    marker=factor_mark('species', markers, species),
                    # Map the species to the Greys4 color palette
                    color=factor_cmap('species', 'Greys4', factors=species),
                    # Add transparency to the markers to make them easier
                    # to see
                    fill_alpha=0.2,
                )

    # Show a grid of plots. Expects a 2D array
    show(gridplot(rows))
```

This results in a collection of scatter plots and histograms:

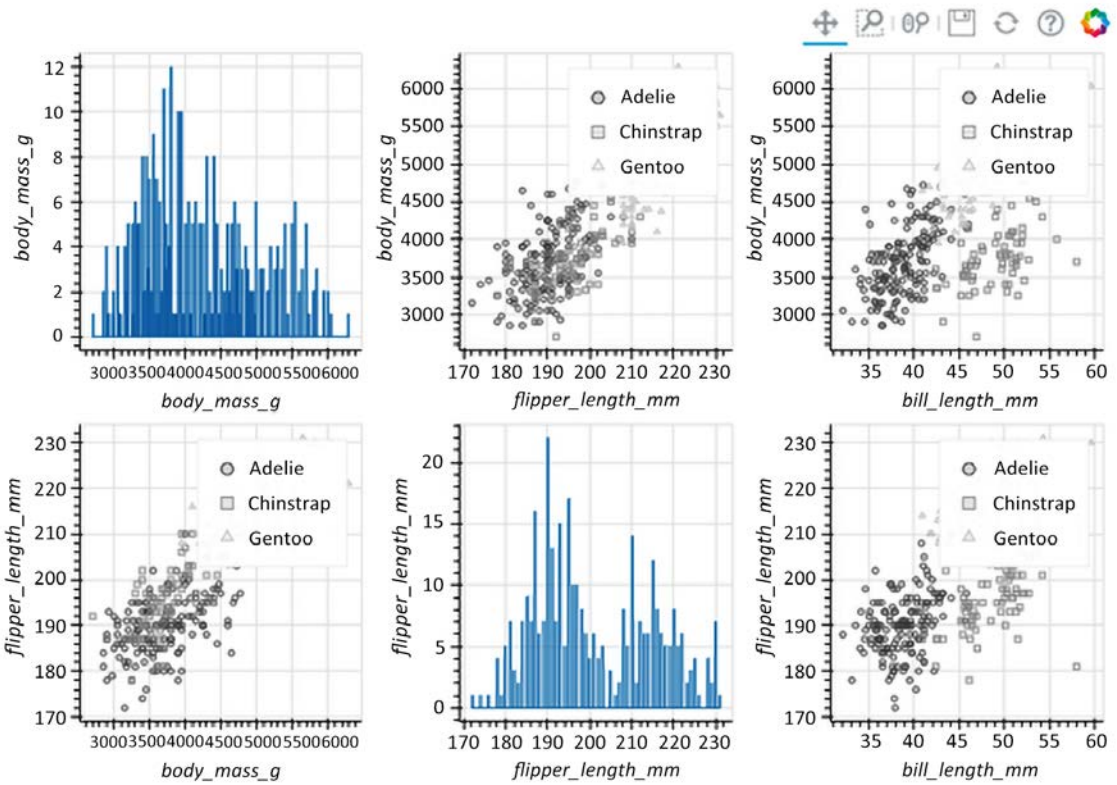


Figure 15.8: Seaborn-like plots using Bokeh

This somewhat resembles what we created with `seaborn`, but it still took quite a bit of effort to do. It does show how we can fairly easily combine several plots (and types of plots) together even when using a `pandas.DataFrame` as a source.

Should you use `bokeh`? I think `bokeh` is a nicely documented plotting library with a lot of merits, but so are many of the others. In my opinion, the main feature of `bokeh` is the support for dynamic data loading through the `bokeh` server, which can be a really useful feature in some cases. As opposed to `plotly`, the `bokeh` server has more features for maintaining its own state, so chart changes can be made easily without recalculation.

Datashader

The `datashader` library is a special case but I believe it deserves a mention. The `datashader` plotting library can be used for regular plotting, but it is specially optimized for high performance and large datasets. As a little example, this plot with 10 million data points only takes about a second to render:

```
import numpy as np, pandas as pd, datashader as ds
from datashader import transfer_functions as tf
from datashader.colors import inferno, viridis
from numba import jit
from math import sin, cos, sqrt, fabs

# Set the number of points to calculate, takes about a second with
# 10 million
n=10000000

# The Clifford attractor code, JIT-compiled using numba
@jit(nopython=True)
def Clifford(x, y, a, b, c, d, *o):
    return sin(a * y) + c * cos(a * x), \
           sin(b * x) + d * cos(b * y)

# Coordinate calculation, also JIT-compiled
@jit(nopython=True)
def trajectory_coords(fn, x0, y0, a, b=0, c=0, d=0, e=0, f=0, n=n):
    x, y = np.zeros(n), np.zeros(n)
    x[0], y[0] = x0, y0
    for i in np.arange(n-1):
        x[i+1], y[i+1] = fn(x[i], y[i], a, b, c, d, e, f)
    return x,y

def trajectory(fn, x0, y0, a, b=0, c=0, d=0, e=0, f=0, n=n):
    x, y = trajectory_coords(fn, x0, y0, a, b, c, d, e, f, n)
    return pd.DataFrame(dict(x=x,y=y))

# Calculate the pandas.DataFrame
df = trajectory(Clifford, 0, 0, -1.7, 1.5, -0.5, 0.7)

# Create a canvas and render
cvs = ds.Canvas()
agg = cvs.points(df, 'x', 'y')
tf.shade(agg, cmap=["white", "black"])
```

Here is the plot generated by calculating the 10 million points:

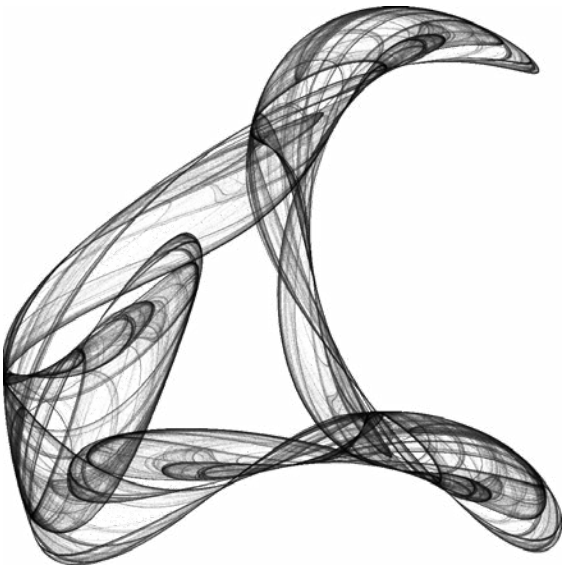


Figure 15.9: Datashader attractor render

Exercises

Due to the nature of this chapter, we have only covered the absolute basics of the mentioned libraries and they really do deserve much more. In this case, as an exercise, I recommend that you try and use some (or all) of the mentioned libraries and see if you can do something useful with them, using the variety of examples we have introduced already as inspiration.

Some suggestions:

- Create your own beautiful datashader plots
- Render the lines of code per project of your personal workspace
- Continuing from the lines of code per project, see if you can cluster the projects by programming language

Example answers for these exercises can be found on GitHub: `https://github.com/mastering-python/exercises`. You are encouraged to submit your own solutions and learn about alternative solutions from others.

Summary

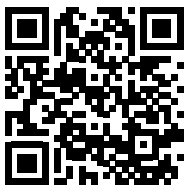
This chapter has shown us a sample of the most commonly used and generic scientific Python libraries. While it covered a lot of libraries, there are many more available, especially when you start looking for domain-specific libraries. With regard to plotting alone, there are at least several other very big libraries that could be useful for your use cases but would be superfluous for this chapter.

To recap, we have covered the basics of working with NumPy matrices and Pandas data objects, both of which are important for the next chapter. We have also seen a few libraries that focus on mathematics and really precise calculations. Lastly, we have covered several plotting libraries, some of which will be used in the next chapter as well.

Next up is the chapter about artificial intelligence and machine learning in Python. As is the case with this chapter, we cannot go into too much depth, but we can cover the most important technologies and libraries so you know where to look.

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:
`https://discord.gg/QMzJenHuJf`



16

Artificial Intelligence

In the last chapter, we saw a collection of scientific Python libraries that allow for really fast and easy processing of large data files. In this chapter, we will use some of these and a few others for machine learning.

Machine learning is a complex subject, and many completely distinct subjects within it are entire branches of research by themselves. This should not discourage you from diving in, however; many of the libraries mentioned in this chapter are really powerful and allow you to get started with a very reasonable amount of effort.

It should be noted that there is a huge difference between applying a pre-trained model and generating your own. Applying a model is usually possible in a few lines of code and barely requires any processing power; building your own model usually takes many lines of code and hours or more to process. This makes the training of models outside of the scope of this book in all but the most trivial cases. In these cases, you will get an overview of what the library can do with some explanation of where this would be useful, without explicit examples.

Artificial intelligence is the branch of computer science relating to the study of all types of machine learning, which includes neural networks and deep learning, Bayesian networks, evolutionary algorithms, computer vision, **natural language processing** (**NLP**), and **support-vector machines** (**SVM**s), among others.

In this chapter, we will cover the following topics:

- Introduction to artificial intelligence
- Libraries for image processing
- Libraries for NLP
- Libraries for neural networks and deep learning
- Generic AI libraries and utilities

Introduction to artificial intelligence

Before we continue with this chapter, we need to establish a few definitions. Because **artificial intelligence** (**AI**) is such a broad subject, the lines tend to blur a bit, so we need to make sure that we are all talking about the same thing.

First of all, we define AI as *any algorithm with a human-like ability to solve problems*. While I admit that this statement is very broad, any narrower definition would exclude valid AI strategies. What is and is not AI is more a philosophical question than a technical one. While (almost) anyone would consider a neural network to be AI, once you get to algorithms such as (Bayesian) decision trees, not everyone agrees anymore.

With that broad definition in mind, here is a list of technologies and terms we are going to cover, with a short explanation of what they are and what they can do.

Types of AI

Within the broad scope of AI, we have two major branches, **machine learning** (**ML**) and the rest. Machine learning covers any method that can learn by itself. You might wonder, is it even AI if it does not involve learning? This is a bit of a philosophical question, but I personally think that there are several non-learning algorithms that can still be considered AI because they can produce human-like decisions.

Within self-learning systems, we have further distinctions with their own goals and applications:

- Supervised learning
- Reinforcement learning
- Unsupervised learning

The use of one of these does not exclude the others from being used too, so many practical implementations use combinations of multiple methods.

Non-machine learning systems are quite a bit more diverse because they can mean just about anything, so here are a few examples of non-learning algorithms that can rival humans in some ways:

- **NLP**: It should be noted that NLP by itself does not use ML. Many NLP algorithms are still written by hand, because it is far easier for a human to explain to a machine how and why certain grammar and semantics work than to have a computer figure out the oddities and complexities of human languages. That field is changing very rapidly, however, and this might not be the case for much longer.
- **Expert systems**: This is the first type of AI that was actually successful in practice. The first expert systems were created in 1970 and they have been used ever since. These systems work by asking you a string of questions and narrowing down a list of potential solutions/answers based on those. You have certainly encountered many of these when going through problem-solving wizards at some point, perhaps in the FAQ on websites or when calling a helpdesk. These systems allow the capturing of expert information and compress it down into a simple system that can make decisions. Many of these have been used (and are still used today) in diagnosing medical issues.

Before we continue with actual AI implementations, it is a good idea to look at a few image processing libraries that are used as a basis in many of the AI examples.

Installing the packages

As was the case with installing the scientific Python libraries in *Chapter 15*, installing the packages in this chapter directly using `pip` can be troublesome in some cases. Using one of the Jupyter Docker Stacks or `conda` can be more convenient. Additionally, most of these projects have very well-documented installation instructions for many scenarios.

For the neural networks portion of this chapter, it would be best to get a notebook stack that has most libraries available. I would recommend giving the `jupyter/tensorflow-notebook` stack a test: `https://jupyter-docker-stacks.readthedocs.io/en/latest/using/selecting.html#jupyter-tensorflow-notebook`.

Image processing

Image processing is an essential part of many types of machine learning, such as **computer vision** (**CV**), so it is essential that we show you a few of the options and their possibilities here. These range from image-only libraries to libraries that have full machine learning capabilities while also supporting image inputs.

scikit-image

The scikit-image (`skimage`) library is part of the scikit project with the main project being scikit-learn (`sklearn`), covered later in this chapter. It offers a range of functions for reading, processing, transforming, and generating images. The library builds on `scipy.ndimage`, which provides several image processing options as well.

We need to talk about what an image is in terms of these Python libraries first. In the case of `scipy` (and consequently, `skimage`), an **image** is a `numpy.ndarray` object with 2 or more dimensions. The conventions are:

- 2D grayscale: Row, column
- 2D color (for example, RGB): Row, column, color channel
- 3D grayscale: Plane, row, column
- 3D color: Plane, row, column, color channel

All of these are just conventions, however; you can shape your arrays in other ways as well. A multi-channel image could also mean **CMYK** (**cyan, magenta, yellow, and key/black**) colors instead of **RGB** (**red, green, and blue**), or something completely different.

Naturally you could have more dimensions as well, such as a dimension for time (in other words, video). Since the arrays are regular `numpy` arrays, you can manipulate them by slicing as usual.

Often you will not use the scikit-image library for machine learning directly, but rather for *pre-processing* image data before you feed it to your machine learning algorithms. In many types of detections, for example, color is not that relevant, which means you can make your machine learning system three times as fast by going from RGB to grayscale. Additionally, there are often fast algorithms available to pre-process the data so your machine learning system only needs to look at the relevant sections of the image.

Installing scikit-image

The package is easily installable through `pip` for many platforms; I would suggest installing not just the base package but the `optional` extras as well, which add extra capabilities to scikit-image, such as parallel processing:

```
$ pip3 install -U 'scikit-image[optional]'
```

Edge detection

Let's look at how we can display one of the built-in images and do some basic processing on it:

```
%matplotlib inline
from skimage import io, data

coins = data.coins()
io.imshow(coins)
```

In this case, we are using the `coins` dataset that is bundled with `skimage`. It contains a few coins and we can use it to display some of the nice features of `skimage`. First, let's look at the results:

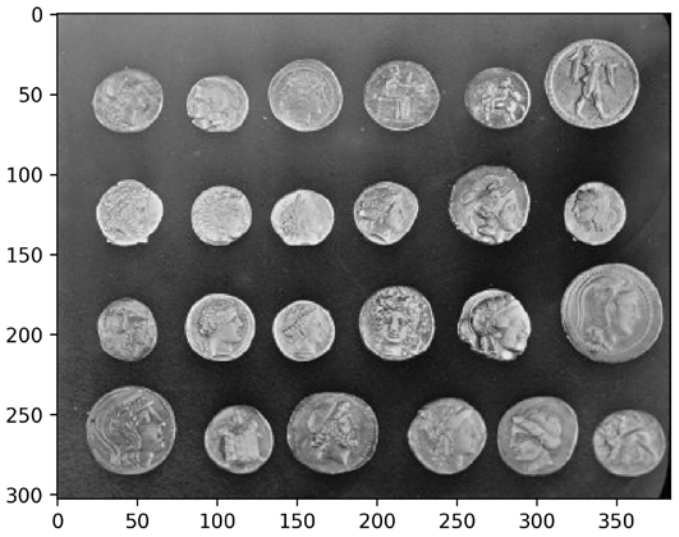


Figure 16.1: scikit-image coins

As an example of what kind of processing we can do, let's do some edge detection using the Canny edge detection algorithm. This is a prime example of a non-ML algorithm that can be really useful for pre-processing your data before you feed it to your ML system. To display the results a bit better, first we will slice the image so only the top-right three coins are visible. In *Figure 16.1*, the numbers indicate the actual pixel indices for the *x* and *y* axes, which can be used to estimate where to slice. After that, we will apply the `canny()` function to detect the edges:

```
%matplotlib inline
from matplotlib import pyplot as plt
from skimage import feature

# Get pixels 180 to the end in the X direction
x0, x1 = 180, -1
# Get pixels 0 to 90 in the Y direction
y0, y1 = 0, 90
# Slice the image so only the top-right three coins are visible
three_coins = coins[y0:y1, x0:x1]
# Apply the canny algorithm
plt.imshow(feature.canny(three_coins), cmap='gray')
```

The results are shown in the following image, where you can see the auto-detected edges of coins we have selected:

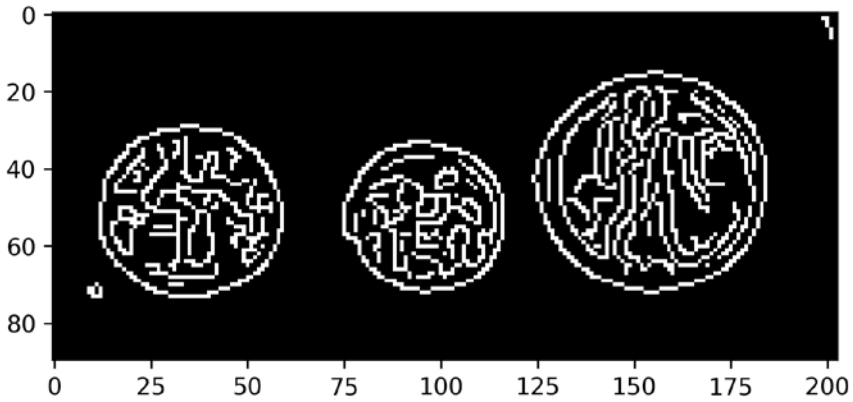


Figure 16.2: Coins after edge detection

scikit-image can do much more, but this is a nice and basic example of how you can do edge detection in a single line of code, which can make your data much more useful for ML systems.

Face detection

We will now use one of the examples from the fantastic scikit-image documentation: `https://scikit-image.org/docs/dev/auto_examples/applications/plot_face_detection.html`.

This is a machine learning example that uses a pre-trained model to automatically detect faces. The specific model uses a multi-block **local binary pattern** (**LBP**). An LBP looks at points surrounding a center point and indicates whether these points are greater (lighter) or smaller (darker) than the center point. The multi-block part is an optional extension to this method and performs the LBP algorithm across multiple block sizes of 9 identically sized rectangles. The first iteration might look at a 3x3 pixel square; the second iteration could look at 6x6; the third 9x9; and so on.

The model was trained using the OpenCV cascade classifier training, which can train your model, generate samples, and run the detection. A cascade classifier concatenates the results of multiple classifiers to reach a combined model that is expected to perform better than the separate classifiers by themselves.

To test the face detection, we will apply it to a photo of the NASA astronaut Eileen Collins. First, we will import the libraries, load the image, and tell `matplotlib` to draw it:

```
%matplotlib inline
from skimage import data
from skimage.feature import Cascade

# We are using matplotlib directly so we can
# draw on the rendered output
import matplotlib.pyplot as plt
from matplotlib import patches

dpi = 300
color = 'white'
thickness = 1
step_ratio = 1
scale_factor = 1.2
min_object_size = 60, 60
max_object_size = 123, 123

# A photo of Astronaut Eileen Collins
img = data.astronaut()

# Plot the image as high resolution in grayscale
plt.figure(dpi=dpi)
plt.imshow(img.mean(axis=2), cmap='gray')
```

Looking at the code above, you might notice a few magic numbers such as the `scale_factor`, `step_ratio`, `min_object_size`, and `max_object_size`. These parameters are ones that you will have to tune to your input image. These specific numbers are straight from the OpenCV documentation, but depending on your input you will need to experiment with these values until they suit your scenario.

Since these parameters are somewhat arbitrary and dependent on your input, it can be a good idea to apply a bit of automation to find them. An evolutionary algorithm could be useful in helping you find effective parameters.

Now we are ready to start the detection and illustrate what we found:

```
# Load the trained file and initialize the detector cascade
detector = Cascade(data.lbp_frontal_face_cascade_filename())

# Apply the detector to find faces of varying sizes
out = detector.detect_multi_scale(
    img=img, step_ratio=step_ratio, scale_factor=scale_factor,
    min_size=min_object_size, max_size=max_object_size)

img_desc = plt.gca()
for box in out:
    # Draw a rectangle for every detected face
    img_desc.add_patch(patches.Rectangle(
        # Col and row as X and Y respectively
        (box['c'], box['r']), box['width'], box['height'],
        fill=False, color=color, linewidth=thickness))
```

After loading the cascade, we run the model using the `detect_multi_scale` method. This method searches for matching objects (faces) with sizes varying from `min_size` to `max_size`, which is needed because we don't know how large the subject (face) is. Once we have the matches, we draw a rectangle around them to indicate where they are:

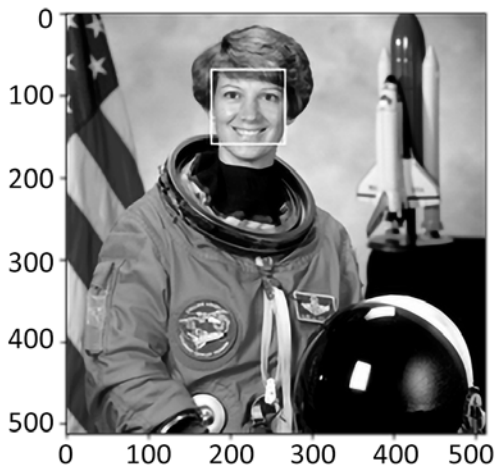


Figure 16.3: Face detected by scikit-image

By itself, scikit-image does not have many machine learning features available, but the coupling with other libraries is what makes this library very useful for machine learning. In addition to the frontal face dataset we loaded above, you can also use pre-trained cascades from OpenCV.

Several pre-trained models are available in the OpenCV Git repository: `https://github.com/opencv/opencv/tree/master/data/lbpcascades`.

scikit-image overview

The scikit-image library can do much more than we have covered. Here's a quick overview of a few of the available submodules:

- `exposure`: Functions for analyzing and fixing photo exposure levels, which can be essential for cleaning data before you feed it to your AI system.
- `feature`: Feature detection such as the `canny()` edge detection function we used earlier. This allows for detecting objects, blobs of content, and more to pre-filter your input so you can reduce the processing time needed by your AI system.
- `filters`: Image filtering functions, such as thresholding to automatically filter noise, and many others. Similar to the `exposure` functions, these can be very useful for cleanup.
- `morphology`: Many functions to sharpen edges, fill sections, find minima/maxima, and so on.
- `registration`: Functions for calculating the optical flow in an image. With these functions, you can estimate what part of the image is moving, and how fast objects are moving. Given two images, this can help to calculate the intermediate image.
- `segmentation`: Functions for segmenting images. In the case of the coins above, the separate coins can be extracted and/or labeled.

As you can see, the scikit-image library offers an extensive list of image manipulation and processing functions. Additionally, it is well integrated into the scientific Python ecosystem.

OpenCV

The big "competitor" to scikit-image is **OpenCV** (**Open Source Computer Vision library**). The OpenCV library is written in C/C++ but has bindings for several languages such as Python and Java. The reason I put "competitor" between quotes is that these libraries don't have to compete; you can easily combine the strengths of both if you wish to do so, and it is something I have done myself in several projects.

We will first look at installing the Python OpenCV package.

Installing OpenCV for Python

The `opencv-python` package comes in several variants depending on your needs. Besides the main OpenCV package, OpenCV also has many "contrib" and "extra" packages, which can be very useful. The contrib packages are mainly for following tutorials and trying examples, and the extra modules contain many useful additional algorithms.

The list of extra modules can be found in the documentation: `https://docs.opencv.org/5.x/`.

I strongly recommend installing the extra modules as well, since many very useful modules are part of the extra package.

You have the following options if you are installing the package on a desktop machine where you will be using a GUI:

- `opencv-python`: The main modules, the bare minimum
- `opencv-contrib-python`: The full package including the main modules from the `opencv-python` package, but also the contrib and extra modules

For servers that are not running a GUI, you have these options:

- `opencv-python-headless`: Beyond not including any GUI output functions such as `cv2.imshow()`, this is identical to `opencv-python`
- `opencv-contrib-python-headless`: As above, this is the headless version of `opencv-contrib-python`

Now that we have OpenCV installed, let's see if we can replicate the Canny edge detection from scikit-image using OpenCV.

Edge detection

Let's look at how we can perform the Canny algorithm using OpenCV, similar to what we did in the scikit-image example earlier. The Canny algorithm is not part of the OpenCV core, so you need to install the `opencv-contrib-python` package for this:

```
$ pip3 install opencv-contrib-python
```

We will use the same coins image as before:

```
%matplotlib inline
import cv2
from matplotlib import pyplot as plt
from skimage import data

# Use the coins image from scikit-image
coins = data.coins()

# Get pixels 180 to the end in the X direction
x0, x1 = 180, -1
# Get pixels 0 to 90 in the Y direction
y0, y1 = 0, 90
# Slice the image so only the top-right three coins are visible
three_coins = coins[y0:y1, x0:x1]
# scikit-image automatically guesses the thresholds, OpenCV does not
```

```
threshold_1, threshold_2 = 100, 200
# Apply the canny algorithm
output = cv2.Canny(three_coins, threshold_1, threshold_2)

# OpenCV's imshow() function does not work well with Jupyter so
# we use matplotlib to render to grayscale
plt.imshow(output, cmap='gray')
```

At a first glance the code looks quite similar, but there are a few differences.

First, the `cv2.Canny()` function requires two extra parameters: `threshold_1` and `threshold_2`, or the lower and upper bounds. These parameters decide what should be considered noise and what parts are relevant for the edges. By increasing or decreasing these values, you can get finer details in the resulting edges, but doing so means the algorithm can also start wrongly detecting the background gradient as edges, which is already happening at the top right of the output image (*Figure 16.4*).

While you can pass these along to scikit-image if you wish, by default scikit-image automatically guesses some suitable parameters for you. With OpenCV you could easily do the same, but this is not included by default. The algorithm that scikit-image uses for this estimation can be seen in the source: `https://github.com/scikit-image/scikit-image/blob/main/skimage/feature/_canny.py` Second, OpenCV has no native support for Jupyter, so we are using `matplotlib` to render the output. Alternatively, we could also use the `IPython.display` module to display the image.

The generated output is similar, however:

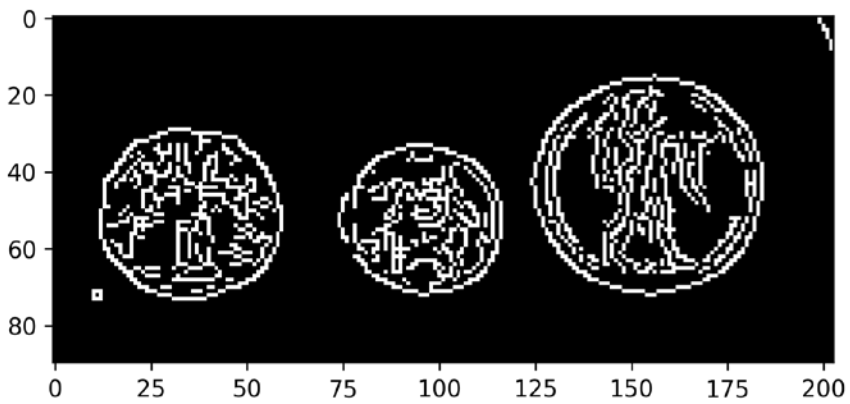


Figure 16.4: OpenCV Canny

For more similar output, you could even use scikit-image to render the output from OpenCV. Since they both operate on `numpy` arrays, you can easily mix and match the functions if needed.

Object detection

In the scikit-image face detection example, we were actually using an OpenCV-generated model, so we could easily use that model with `opencv-python` directly, with a few small changes:

- Instead of `skimage.feature.Cascade(filename)`, you need to use `cv2.CascadeClassifier(filename)`
- Instead of `cascade.detect_multi_scale()` the function is called `cascade.detectMultiScale()`

This immediately illustrates one of the differences between scikit-image and `python-opencv`. Where scikit-image uses the Python convention of underscores between words in a function name, `opencv-python` uses the camelCase function names directly from the OpenCV source.

With OpenCV we can easily go beyond the simple cascades we used for face detection; this time we will use a **DNN** (**deep neural network**) instead.

The network we will be using is called **YOLOv3** (**You Only Look Once, version 3**) and is able to detect many types of objects such as cars, animals, fruit, and many more. Naturally this model is far larger as well. The face detection model was only about 50 KiB, while the YOLOv3 network is nearly 5000 times larger, at 237 MiB.

Before we can start, we need to download a few files for the YOLO network to be fully functional:

- The model (237 MiB): `https://pjreddie.com/media/files/yolov3.weights`
- The YOLO configuration file: `https://raw.githubusercontent.com/pjreddie/darknet/master/cfg/yolov3.cfg`
- The names for the objects: `https://raw.githubusercontent.com/pjreddie/darknet/master/data/coco.names`

Once you have those files, we can demonstrate the YOLO network. First, we set up a few imports and variables, and then load the image:

```
%matplotlib inline
import cv2 as cv
import numpy as np
from matplotlib import pyplot as plt
from skimage import data

color = 0xFF, 0xFF, 0xFF  # White
dpi = 300
font = cv.FONT_HERSHEY_SIMPLEX
font_size = 3
image_dims = 320, 320
label_offset = 10, 70
min_score = 0.9
```

```
thickness = 2

# Load the astronaut image from scikit-image as before
img = data.astronaut()
# Convert the image into a 4-dimensional blob
# by subtracting the mean and rescaling
blob = cv.dnn.blobFromImage(img, 1 / 255, size=image_dims)
```

Now that we have the imports ready and the image converted to a blob that's suitable for the model, we can load the model and show the results:

```
# Load names of classes so we know what was detected
classes = open('coco.names').read().splitlines()
# Load the deep neural network model and configuration
net = cv.dnn.readNetFromDarknet('yolov3.cfg', 'yolov3.weights')
# Determine the output layer
ln = net.getLayerNames()
ln = [ln[i - 1] for i in net.getUnconnectedOutLayers()]

# Pass the blob to the net and calculate the output blobs
net.setInput(blob)
out = net.forward(ln)
# Loop through all outputs after stacking because
# the net attempts to match multiple sizes
for result in np.vstack(out):
    # x, y, w and h are numbers between 0 and 1 and need to be
    # scaled by the width and height
    result[:4] *= img.shape[1::-1] * 2
    x, y, w, h, *scores = result
    # Search the net for the best match
    match_index = np.argmax(scores)
    # Skip questionable matches
    if scores[match_index] < min_score:
        continue

    # Calculate the top left and bottom right points
    tl = np.array([x - w / 2, y - h / 2], dtype=int)
    br = np.array([x + w / 2, y + h / 2], dtype=int)
    cv.rectangle(img, tl, br, color, thickness)
    # Calculate the point to place the text
    cv.putText(img, classes[match_index], tl + label_offset,
               font, font_size, color, thickness)
```

```
        # Stop after the first match to prevent overlapping results
        break

plt.figure(dpi=dpi)
plt.imshow(img.mean(axis=2), cmap='gray')
```

For brevity this example is very condensed, but it shows you how you can do something as advanced as object detection in just a few lines. If we look at the output, the deep neural network has correctly identified the astronaut as being a person:

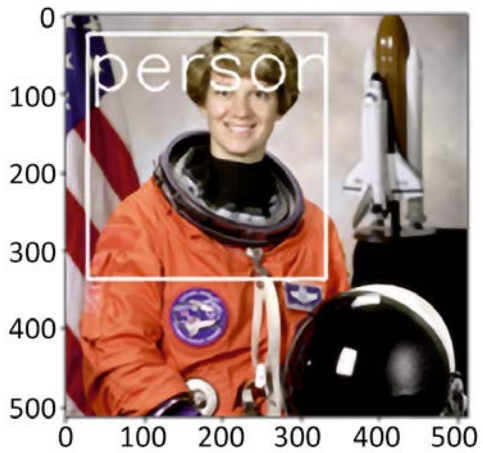


Figure 16.5: Object detection on astronaut

I highly encourage you to try the YOLOv3 network yourself with different images. For an old image of a street, I got the following results:

Figure 16.6: Applying YOLOv3 on an image of a street with cars

Isn't it amazing how easy it is to do object detection these days and how well it works? If you take a good look at the image, you might notice that it's even detecting cars and people that are partially obstructed. Training a new deep neural network and doing the research for it is a completely different question, of course, but at least applying these networks has become child's play and they execute well within a second, including the loading of the network.

The possibilities certainly don't end here, and you could even use techniques like these to do real-time analysis of a video stream if you wanted. The OpenCV library really is an impressive bit of software.

OpenCV versus scikit-image

Both scikit-image and OpenCV have their own advantages over the other. This is one of the cases where you don't really have to choose, however; you can easily use both simultaneously.

In my opinion, OpenCV has three major advantages over scikit-image:

- OpenCV has native support for processing using your GPU
- Since it is implemented in C++, you can do parallel processing in threads without having to worry about the GIL
- OpenCV has even more features than scikit-image

Naturally, scikit-image has a few advantages as well:

- scikit-image is written in Python so it is very easy to view (or modify) the algorithms right from your editor.
- scikit-image is focused toward Python, so the naming conventions feel more natural.
- As scikit-image is only for Python, all documentation is immediately relevant. With OpenCV, many of the examples you will find on the web (and in the documentation) are about the C++ interface, which is slightly different.

If you need high performance for the live processing of video streams, then OpenCV would be my top recommendation because it has several methods built in to make that task a bit easier. If you simply need to read and modify some images and you can get away with scikit-image, then that would be my top recommendation.

In either case, both libraries are great and I can confidently recommend both. If your needs span across both, use both.

Now it is finally time to start discussing the artificial intelligence libraries themselves.

Natural language processing

NLP is the process of parsing text and understanding its meaning. This can be used to extract knowledge from pieces of text, understand the differences between texts, and more.

There are several well-developed libraries available for this purpose that work quite well. Additionally, there are also hosted pre-trained networks available such as the **GPT-3** network, which can be accessed through the OpenAI API.

This network can generate text of such high quality that it is often indistinguishable from human-generated text.

NLTK – Natural Language Toolkit

NLTK is not really a machine learning library by itself like most of the other libraries here, but it's the basis for many natural language processing libraries. The NLTK project started in 2001 with the purpose of understanding natural languages, and definitely deserves a place in this list.

The project comes bundled with a large collection of corpora and pre-trained models for many different languages.

Corpora are large collections of structured texts that can be used for training and testing models.

Using these corpora and models, it can do sentiment analysis, tokenize the text to find the relevant keywords, and more.

First, we need to install `nltk`:

```
$ pip3 install nltk
```

As a basic example, let's use the pre-trained sentiment analysis capability to see how positive or negative a sentence is:

```
>>> import nltk
>>> from nltk import sentiment

>>> nltk.download('vader_lexicon')
True

>>> sentences = [
...     'Python is a wonderful programming language',
...     'Weak-typed languages are prone to errors',
...     'I love programming in Python and I hate YAML',
... ]

>>> si = sentiment.SentimentIntensityAnalyzer()
>>> for sentence in sentences:
...     scores = si.polarity_scores(sentence)
...     print(sentence)
...     print('negative: {neg}, positive: {pos}'.format(**scores))
Python is a wonderful programming language
```

```
negative: 0.0, positive: 0.481
Weak-typed languages are prone to errors
negative: 0.324, positive: 0.0
I love programming in Python and I hate YAML
negative: 0.287, positive: 0.326
```

We start by downloading the pre-trained model for sentiment analysis. After that, we can use the `SentimentIntensityAnalyzer` to detect if a sentence is negative, neutral, positive, or a combination.

The library can do much more, but this already gives you a nice indication of how easy it is to get started. If you need any basic human input parsing, make sure to give it a try as it offers very impressive results.

spaCy – Natural language processing with Cython

The spaCy library is a very impressive and extremely fast NLP library. It comes with many pre-trained neural network models for over 60 languages and does a very good job at text classification and named entity recognition.

The documentation is amazing and, while being fully open-source, it is developed by the company Explosion, which is doing a really good job of keeping up with the latest developments in NLP. If you want a high-level understanding of text, this library is one of your best options. If you only need basic text tokenization, then I would still recommend `NLTK` because it is faster and more effective.

Before we continue with the example, we need to install spaCy and download the models:

```
$ pip3 install spacy
$ python3 -m spacy download en_core_web_sm
```

The `en_core_web_sm` dataset is a small and fast English dataset. If you need a more thorough dataset, you can download `en_core_web_trf` instead.

To install a different language, I recommend you visit the spaCy website: `https://spacy.io/usage#quickstart`. For example, the Dutch dataset is called `nl_core_news_sm`, as opposed to `nl_core_web_sm`, which you might have been expecting.

Now that we have that taken care of, let's try to extract some information from a sentence:

```
>>> import spacy
>>> import en_core_web_sm

>>> nlp = en_core_web_sm.load()
>>> _ = nlp.add_pipe("merge_entities")

>>> sentence = ('Python was introduced in 1989 by Guido van '
... 'Rossum at Stichting Mathematisch Centrum in Amsterdam.')
```

```
>>> for token in nlp(sentence):
...     if token.ent_type_:
...         print(f'{token.ent_type_}: {token.text}')
DATE: 1989
PERSON: Guido van Rossum
ORG: Stichting Mathematisch Centrum
GPE: Amsterdam
```

After loading `spacy` and the `en_core_web_sm` model, we added the `merge_entities` pipe. This pipe automatically merges the tokens together so we get `"Guido van Rossum"` instead of `"Guido"`, `"van"`, and `"Rossum"` as separate tokens.

Isn't this an amazing result? It automatically understands that `"Guido van Rossum"` is a person, `"Stichting Mathematisch Centrum"` is an organisation, and `"Amsterdam"` is a geopolitical entity.

Gensim — Topic modeling for humans

The Gensim library (`https://radimrehurek.com/gensim/`) takes care of NLP for you. It is similar to NLTK but more focused on the modern machine learning libraries. It is well documented and easy to use and can be used to calculate similarities between texts, analyze the topic of a piece of text, and more. While there is a large overlap between NLTK and Gensim, I would argue that Gensim is a bit of a higher-level library and slightly easier to get started with. NLTK, on the other hand, has existed for over 20 years and has a huge amount of documentation available in the wild because of that.

Machine learning

Machine learning is the branch of artificial intelligence that can learn by itself. This can be fully autonomous learning, learning based on pre-labeled data, or a combination of these.

We need a little bit of background information before we can dive into the libraries and the examples for this subject. Feel free to gloss over this section and jump straight to the libraries if you are already familiar with the types of machine learning.

Types of machine learning

As we have briefly covered in the introduction, machine learning roughly splits up into three different methodologies, but often uses a combination of several. To recap, we have the following three major branches:

- Supervised learning
- Reinforcement learning
- Unsupervised learning

Naturally, there are many combinations of these, so we will discuss a few important distinct types of learning that are based on the branches above. The names themselves should already give you a hint about how they function, but we will dive deeper.

Supervised learning

In the case of supervised learning, we provide the system with a lot of labeled data so the machine can learn the relationship between the input data and the labels. Once it has been trained on that data, we can test using new data to see if it works. If the results are not as expected, parameters or intermediate training steps are tuned until the results improve.

Examples of these are:

- Classification models where the models are trained on a large number of photos to recognize the objects in the photo. Or to answer a question such as: "Are we looking at a bird?"
- Sentiment analysis of text. Is the person writing the message happy, sad, hostile, and so on?
- Weather prediction. Since we have a huge amount of historical weather data available, this is a perfect case for supervised learning.

If you have the data available, this will probably be your best option. In many cases, however, you either don't have the data or you have data without high-quality labels. That is where the other learning methods come in.

Reinforcement learning

Reinforcement learning is similar to supervised learning, but instead of using labeled input/output pairs it uses a **scoring** or **reward function** to provide feedback. The parameter that has to be tuned with reinforcement learning is whether to re-use existing knowledge or to investigate a new solution. Leaning too heavily toward re-use will result in a "local optimum," where you will never get the best (or even a good) result because you get stuck on your previously found solution. Leaning too much toward investigation/exploration of new solutions, however, results in never reaching an optimal solution.

Examples of these are:

- Creating solvers/players for games. For a game such as Go or Chess, you could use win/lose as a scoring function. For a game such as Pong or Tetris, you could use the score as the reward.
- Robot navigation systems. As a scoring system, you could use "distance moved from origin" combined with "not hitting a wall."
- Swarm intelligence. These are systems with many (a swarm) of independent, self-organizing systems that need to reach a common goal. As an example, some online supermarkets use swarms of robots to automatically fetch and package groceries with this method. The swarm intelligence takes care of collision avoidance and automatically replacing defective robots.

Reinforcement learning is the next best option after supervised learning, because it doesn't require a large amount of high-quality data. You can combine these methods quite well, though. Creating a good scoring function can be difficult, and you can easily verify your function by testing it on known good data.

Unsupervised learning

By the name alone, you might be confused by unsupervised learning.

After all, how would an unsupervised system work if it has no idea when it has reached a useful solution? The point is that with unsupervised learning you don't know what the solution will look like in the end, but you can declare how a solution *could* look.

Since the explanation of unsupervised learning is a bit vague, I hope some examples help:

- Clustering algorithms. With clustering, you feed the algorithm data with a lot of variables (for example, in the case of people, weight, height, gender, and so on) and tell the algorithm to find clusters.
- Anomaly detection. This is also an area where unsupervised learning can really shine. With anomaly detection, you never know what you are really looking for, but any patterns that are out of the ordinary could be important.

Unsupervised learning is quite a different type of method from the other two machine learning methods we covered earlier because there is often no known target. However, that does not make it useless by any means. Finding patterns in seemingly random data can be really useful in uptime/stability monitoring or visitor analysis for e-commerce websites, among other things.

Now it's time to look at combinations of the previous methods.

Combinations of learning methods

AI development is as active as it has ever been and I expect the field to keep growing in the foreseeable future. That is why more and more variants of algorithms are being used, which causes these clear-cut definitions to become more flexible all the time.

In some cases, for example, you can get much better results by combining supervised and reinforcement learning together than you could by using either of these methods alone. That is why the lines between all of these methods can be extremely blurry, and if a method works for your goal, it is not wrong to combine them.

Deep learning

One of the most effective examples of machine learning is deep learning. This type of machine learning has become extremely popular over the last few years because it has proven to be one of the most effective types of neural networks in practical applications, in some cases even outperforming human experts.

This type of network is called **deep** because the neural network has multiple (often many) hidden internal layers, while traditional neural networks usually only have a single or a few hidden layers.

Beyond that, it is just a regular neural network and can be supervised, unsupervised, reinforcement, or anything in between.

Artificial neural networks and deep learning

When thinking about AI, most people will immediately think of **artificial neural networks** (**ANNs**). These networks are an attempt to mimic the workings of animal brains by having artificial neurons and connections between them similar to synapses.

There are a few key differences, however. In an animal brain, a neuron can function both as input and output, whereas with an ANN there are usually a set of input neurons in an input layer, a set of neurons as an output layer, and the middle layer(s) that handles the processing.

Currently (in 2021; it was launched in June 2020) by far the most impressive ANN is the GPT-3 network, which has been trained for NLP. It has an incredible 175 billion machine learning parameters and in some cases the text it generates is indistinguishable from human-written text.

This text is likely to be outdated quite soon, however. The GPT-3 network is already 100 times bigger than GPT-2, which was released in 2019. GPT-4 has already been announced and is supposed to be about 500 times larger than GPT-3.

It should be noted that while ANNs (and especially deep learning) networks are very powerful and can be self-learning, many of them are static. After they have been trained once, they do not improve or update anymore.

The libraries in this section are made to build neural networks and to enable deep learning. Since this is an entirely distinct field in AI, it really deserves its own section. Note that you can still mix and match AI strategies if needed, of course.

Within Python, there are multiple large libraries for creating neural networks, but the biggest ones by far are **PyTorch** and **TensorFlow/Keras**. Until a few years ago, there was another large library with similar features called Theano. That library has since been discontinued and forked under a new name, Aesara. Neither of these is used very often these days, but Theano is considered to be the original Python neural network library. The TensorFlow library was actually created to replace Theano within Google.

Tensors

The basis of an ANN is the tensor. Tensors are a mathematical representation for your data with descriptions of valid transformations that can be applied to this data. The actual story is much more complicated, of course, but for the purposes of the discussion here you can think of a tensor as a multi-dimensional array very similar to the `numpy.ndarray` object we have seen in the previous chapter.

When people talk about a 0-dimensional or 0D Tensor, they are effectively talking about a single number. Going up from that, a 1D tensor is an array or vector, and a 2D tensor is a matrix.

The big takeaway for now is that the difference between a regular number/array/matrix and a tensor is that the tensors specify what transformations are valid on them as well. It is basically the difference between a `list()` and a custom `class` that contains the data for the `list()` but has additional properties as well.

PyTorch – Fast (deep) neural networks

PyTorch is a library developed by Facebook and focuses on building neural networks, such as deep learning networks, using tensors.

The tensors in PyTorch use a custom data structure (instead of `numpy.ndarray`) for performance reasons. The PyTorch library is heavily optimized for performance and it has built-in support for GPU acceleration for further speedups.

In many cases you can use `torch.Tensor` as a drop-in replacement for `numpy.ndarray` to enable GPU acceleration. The `torch.Tensor` API is largely identical to the `numpy.ndarray` API.

The real strength of PyTorch (besides the performance) is the number of utility libraries included for different kinds of inputs. You can easily use it to process images, video, audio, and text using these APIs, and most processes can easily be run in parallel in a distributed fashion.

Here is a little overview of the most useful modules:

- `torch.distributed`: For parallel training across multiple GPUs in a single system or across multiple systems.
- `torchaudio`: For processing audio, either from pre-recorded files or straight from (multiple) microphones.
- `torchtext`: For processing text; you can also combine this with NLP libraries such as NLTK.
- `torchvision`: For processing images and sequences of images (videos).
- `torchserve`: For setting up a server that hosts your models so you can build a service that runs your calculations. This is useful because starting a process and loading the model can be a slow and heavy task.
- `torch.utils`: Contains many useful utility functions, but above all, TensorBoard. With TensorBoard, you can interactively (through a web interface) inspect your models and make changes to your model parameters.

It's time for a small example, but before we can get started we need to install both `pytorch` and `torchvision`:

```
$ pip3 install torch torchvision
```

We will use the pre-trained **Mask R-CNN** model to do object recognition. This is a **region-based convolutional neural network** (**R-CNN**) that has been trained using a combination of images and labeled image masks (object outlines).

CNNs are well suited for visual applications such as image classification and image segmentation. They can also be applied to other types of problems such as NLP as well.

The R-CNN is a specialized version of the CNN specifically for computer vision tasks such as object detection. R-CNN tasks are trained by specifying the **region of interest** (**ROI**) in a set of images. The Mask R-CNN is a specialization that specifies the ROI not as a rectangle but as a mask that only highlights the specific object.

Now we'll do some object recognition using PyTorch. First, we load the photo and imports and convert the photo into a tensor:

```
%matplotlib inline
from PIL import Image
from matplotlib import pyplot as plt, patches
from torchvision import transforms
from torchvision.models import detection

dpi = 300
font_size = 14
color = 'white'
min_score = 0.8
min_size = 100
label_offset = 25, -25

# Load the img and convert it to a PyTorch Tensor
img = Image.open('amsterdam-street.jpg')
img_t = transforms.ToTensor()(img)
```

The conversion to a tensor can be done using the `ToTensor` transform operation. The `torchvision.transforms` module has many more operations available, such as resizing, cropping, and color normalization, to pre-filter the images before we send them to the model.

Next up is the loading of the model and the labels:

```
# Read the labels from coco_labels. The entire COCO
# (Common Objects in COntext) dataset is available at:
# https://cocodataset.org/#download
labels = open('coco_labels.txt').read().splitlines()

# Load the R-CNN model and set it to eval mode for execution
model = detection.fasterrcnn_resnet50_fpn(pretrained=True)
model.eval()
# Apply the model to the img as a list and unpack after applying
out, = model([img_t])
```

The label file is available on this book's GitHub page.

As you can see, the model itself is bundled with PyTorch. After loading the model and setting it to `eval` mode (as opposed to training), we can quickly apply the model to our image. The labels are unfortunately not bundled, so we need to fetch those ourselves. Now we need to display the results:

```
results = zip(out['boxes'].detach(), out['labels'], out['scores'])

# Increase the DPI to get a larger output image
plt.figure(dpi=dpi)
img_desc = plt.subplot()
# Walk through the list of detections and print the results
for (t, l, b, r), label_idx, score in results:
    # Skip objects that are questionable matches
    if score < min_score:
        continue

    # Skip tiny matches
    h, w = b - t, r - l,
    if w < min_size or h < min_size:
        continue

    # Draw the bounding box and label
    img_desc.add_patch(patches.Rectangle(
        (t, l), h, w, fill=False, color=color))
    label = f'{labels[label_idx]} {score * 100:.0f}%'
    img_desc.text(
        t + label_offset[0], r + label_offset[1], label,
        fontsize=font_size, color=color)

# Output the img as grayscale for print purposes
plt.imshow(img.convert('L'), cmap='gray')
plt.show()
```

We can display the matches and their bounding boxes to get the following result:

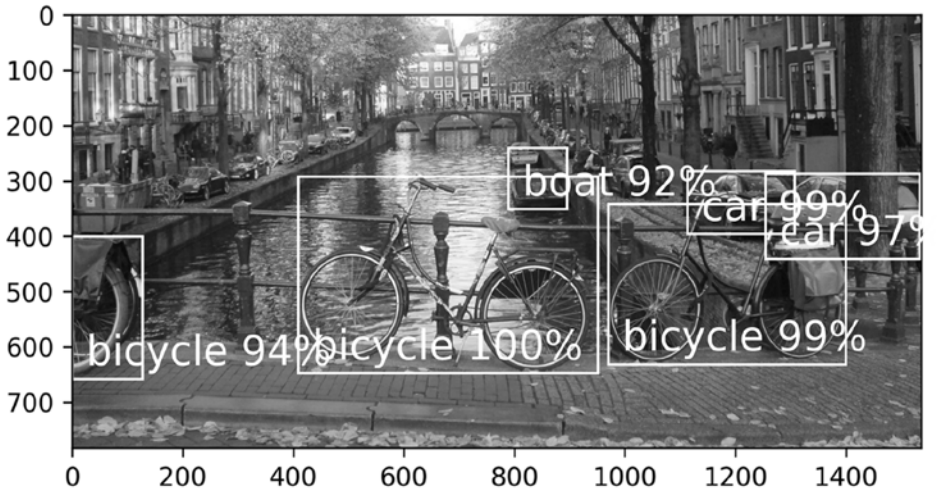


Figure 16.7: Street in Amsterdam with objects labeled by PyTorch

With just a few lines of code, we managed to create an object recognizer that correctly identified a few cars, bicycles, and a boat.

In practice, the model actually recognized far more objects in the image, but we filtered out small matches so the image is not too busy. It actually recognized seven more cars, four people, and two boats.

PyTorch Lightning and PyTorch Ignite — High-level PyTorch APIs

The PyTorch Lightning and PyTorch Ignite libraries are convenient shortcuts for getting your network up and running with fewer steps and several useful features built in. You can do the same with PyTorch directly, but using the utility functions you can run several PyTorch steps at once, meaning less repetition while working.

These libraries were created independently, but serve roughly the same goal and are comparable in features. It depends on your personal preference as to which is the best for you. I would initially recommend you start with PyTorch directly, however. While these libraries are really great, it is important to understand the underlying principles before you start using shortcuts that you might not completely understand. The PyTorch documentation is quite easy to follow and largely identical in workings to PyTorch Ignite and PyTorch Lightning, besides being a bit more verbose.

Skorch — Mixing PyTorch and scikit-learn

As was briefly mentioned, scikit-learn natively supports neural networks, but its performance is not good enough for large-scale networks. The Skorch library takes care of that; you can still use the scikit-learn API if you are familiar with that, but it runs on PyTorch internally to achieve great performance.

TensorFlow/Keras – Fast (deep) neural networks

The TensorFlow library is developed by Google and focuses on building deep neural networks very similar to PyTorch. The library is well documented and has a large number of pre-trained models available to use; you may never have to train your own models, which can be a big advantage.

Similar to PyTorch, TensorFlow is also based on tensors for the actual calculations, and it is highly optimized for performance on many platforms, including mobile phones for deployment and dedicated **tensor processing units** (**TPU**s) or GPU hardware for training the models.

As an example, we will run the Mask R-CNN we used with PyTorch earlier again. Since this model is not bundled with `tensorflow`, we need to install `tensorflow-hub` in addition to `tensorflow`:

```
$ pip3 install tensorflow tensorflow-hub
```

This will automatically install `tensorflow` with GPU support if available for your platform. Currently, that means either Windows or Ubuntu Linux. Now we can test some TensorFlow/Keras code. First, we import what we need, set some variables, and load the image:

```
%matplotlib inline
import numpy as np
import tensorflow_hub as hub
from keras.preprocessing import image
from matplotlib import pyplot as plt, patches

dpi = 300
font_size = 14
color = 'white'
min_score = 0.8
min_size = 100
label_offset = 25, -25

# Load the img and convert it to a numpy array
img = image.load_img('amsterdam-street.jpg')
img_t = image.img_to_array(img)
img_w, img_h = img.size
```

Now that the image is loaded, let's load the model using `tensorflow_hub` and apply it on our image:

```
labels = open('coco_labels.txt').read().splitlines()
model = hub.load(
    'https://tfhub.dev/tensorflow/mask_rcnn/inception_resnet_v2_1024x1024/1')
out = model(np.array([img_t]))
```

```
# The box coordinates are normalized to [0, 1]
img_dim = np.array([img.size[1], img.size[0]] * 2)
result = zip(
    out['detection_boxes'][0] * img_dim,
    out['detection_classes'][0],
    out['detection_scores'][0],
)
```

Once again, we don't have the labels available, so we read that from our `coco_labels.txt` file. However, once we load the model we can easily apply it to our image.

Now we need to prepare the results for easy processing and display them:

```
# Increase the DPI to get a larger output image
plt.figure(dpi=dpi)
img_desc = plt.subplot()

# Walk through the list of detections and print the results
for (l, t, r, b), label_idx, score in result:
    label_idx = int(label_idx)
    # Skip objects that are questionable matches
    if score < min_score:
        continue

    # Skip tiny matches
    h, w = b - t, r - l,
    if w < min_size or h < min_size:
        continue

    # Draw the bounding box and label
    img_desc.add_patch(patches.Rectangle(
        (t, l), h, w, fill=False, color=color))
    label = f'{labels[label_idx]} {score * 100:.0f}%'
    img_desc.text(
        t + label_offset[0], r + label_offset[1], label,
        fontsize=font_size, color=color)

# Output the img as a large grayscale for print purposes
plt.imshow(img.convert('L'), cmap='gray')
```

The code is largely similar to the PyTorch code because it uses the same pre-trained model. The notable differences are:

- We loaded the model using `tensorflow_hub`. This automatically downloads and executes pre-trained models from `https://tfhub.dev/`.
- The box points are from 0 to 1 instead of being relative to the image size. So, coordinate `10x5` in a `20x20` image results in `0.5x0.25`.
- The output variable names are different. It should be noted that these are dependent on the model and can be found on TensorFlow Hub for this model: `https://tfhub.dev/tensorflow/mask_rcnn/inception_resnet_v2_1024x1024/1`.
- The box points use the left, top, right, bottom order instead of top, left, bottom, right, as was the case with PyTorch.

Beyond those small changes, the code is effectively identical.

NumPy compatibility

The actual tensor objects in TensorFlow are slightly different from the PyTorch tensors. While the `pytorch.Tensor` API can be used as a `numpy.ndarray` alternative, with `tensorflow.Tensor` the API is a bit different.

There is a `tensorflow.Tensor.numpy()` method, which returns a `numpy.ndarray` of the data. It is important to note that this is *not* a reference, however; modifying the `numpy` array will *not* update the original tensor, so you will need to convert it back after your changes.

As an alternative, TensorFlow does offer an experimental `numpy` API if you prefer that API. It can be enabled like this:

```
>>> import tensorflow.experimental.numpy as tnp

>>> tnp.experimental_enable_numpy_behavior()
```

Usage is fairly straightforward, but it is by no means fully `numpy.ndarray`-compatible:

```
>>> x = tnp.random.random([5])
>>> x[:5] += 10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tensorflow.python.framework.ops.EagerTensor' object does not
support item assignment
```

Keras

The Keras submodule of TensorFlow is similar to what PyTorch Lightning and PyTorch Ignite are for PyTorch. It offers a high-level interface for TensorFlow, making it easier to use and get started with. As opposed to the aforementioned PyTorch libraries, Keras is quite suitable as a starting point as well. Knowing the underlying TensorFlow functions can be useful, but it is not a requirement for being able to use Keras effectively.

Keras might be for you if you are just getting started with TensorFlow and want to apply some machine learning to your project without going down the rabbit hole.

TensorFlow versus PyTorch

There are a few advantages and disadvantages to TensorFlow compared to PyTorch, so let's list those before we continue.

Here are some reasons you might choose TensorFlow over PyTorch:

- TensorFlow supports the execution of the pre-trained model in a web browser. While PyTorch does have a few libraries available to do this as well, they are either stale or far behind TensorFlow in terms of features and stability.
- TensorFlow is largely language-agnostic. This means that it has bindings for multiple languages, whereas PyTorch is largely Python-only.
- TensorFlow, or more specifically, Keras, is a very high-level API that allows you to get started quickly. When comparing Keras to PyTorch Lightning/PyTorch Ignite, I personally feel that you can get a working result more quickly with TensorFlow. Keras has many utility functions and classes bundled that can save you some work while creating a model. Another big help is TensorFlow Hub, which offers many pre-trained models with example code for your convenience.
- TensorFlow has a slightly bigger community and slightly more tutorials available.

Conversely:

- PyTorch was written around Python and has a much more Pythonic API.
- PyTorch offers more fine-grained control and easily gives you many parameters to tune.
- While this is a personal opinion, I find that debugging PyTorch is much nicer (from Python, at least) than TensorFlow or Keras because the codebase has fewer layers and seems less complicated. Stepping through the execution of your model with a regular Python debugger works great and is easy to follow in the case of PyTorch. In my experience, regular Python debuggers do not work at all with TensorFlow.
- PyTorch is a bit faster than TensorFlow. This can be a huge help while developing and debugging.

Which of the libraries you should use depends on personal preference and factors such as pre-existing experience for you and/or the rest of your team. I can certainly recommend both of them.

Evolutionary algorithms

Evolutionary algorithms are a technique based on evolution in nature that improve by using a fitness function to determine quality, and by evolving the solution.

The most common implementation is the **genetic algorithm**, which commonly encodes the solution, or chromosome, into a string or an array that can be tested by the fitness function. This chromosome could be a list of functions to apply, a list of parameters to a function, or something else entirely. How you wish to encode the chromosome is up to you, as long as the fitness function can use it to calculate a fitness score.

The genetic algorithm will employ one of the following operations to try and improve the fitness score:

- **Mutation**: This could be a bit flip from 0 to 1, or a more complex mutation of replacing multiple bits. For example, if we have bit-string `0101`, then a mutation could result in `0111`.

- **Selection**: Given a set of different tested chromosomes using the fitness function, only keep the best few.
- **Crossover**: Given a few different chromosomes, combine parts of them to try new solutions. For example, if we have two strings, AABB and DEFG, the crossover can split them and combine them; for instance, you could get AAFG, which combines the first two characters from the first string and the last two from the second string.

The genetic algorithm takes a few parameters to control which strategy is employed at a given run. The **mutation rate** sets the probability of a mutation occurring; the **elitism parameter** decides how many results to keep in the selection process; and the **crossover rate** sets the probability of crossovers occurring. The difficult part is tuning these parameters to return a good and stable solution (in other words, one that does not change too much between runs), but not getting stuck in a local optimum where your solution appears the best but could be far better by attempting more genetic diversity.

There are many applications where genetic algorithms (or more generally, genetic programming) are the most feasible option to get a good solution to your problem. One of the prime examples of where genetic algorithms shine is the **traveling salesman problem** (**TSP**). With the TSP, you have a list of cities that you want to visit, and you want to find the shortest route that covers all of them. The standard brute-force solution has a time complexity of `O(n!)`, which means that for 10 cities you need about 3,628,800 steps to calculate. That is a lot, but still easily manageable. For 20 cities, however, the number grows to 2,432,902,008,176,640,000, or, 2 quintillion (2 billion billion), and that growth continues very rapidly. With genetic programming, the fitness problem will almost immediately eliminate parts of the solution space that are completely infeasible and gives you a good (but possibly not the best) solution relatively fast.

Even though evolutionary algorithms offer a lot of power, implementing them is relatively easy to do and often highly specific to your specific use case. This makes it a scenario where applications and libraries usually opt for writing their own implementation instead of using a library for this goal.

There is at least one notable Python library for genetic algorithms however. The PyGAD library can make it easily possible for you to use genetic algorithms in your project. It also comes with built-in support for Keras and PyTorch to save you some work.

Let's start by installing PyGAD:

```
$ pip3 install pygad
```

Now we will attempt to solve a problem you might encounter in the real world. Let's say that you need a new floor and you want wooden floorboards. Due to bulk discounts, it can be cheaper to buy a large stack of boards instead of just a few separate boards, so let's assume we have a few different bulk quantities and make our algorithm optimize for cost. First, we need to define our list of bulk sizes with the prices. We will also define the number of boards we are looking for. Lastly, we will define the fitness function to tell PyGAD how good (or bad) the solution is:

```
import numpy as np
import pygad
```

```
# Combination of number of boards with the prices per board
stack_prices = np.array(
    [
        [1, 10],  # $10 per board
        [5, 5 * 9],  # $9 per board
        [10, 10 * 8],  # $8 per board
        [25, 25 * 7],  # $7 per board
    ]
)

# The minimum number of boards to buy
desired_boards = 67

def fitness_function(solution: numpy.ndarray, solution_index):
    # We can't have a negative number of boards
    if (solution < 0).any():
        return float('-inf')

    # Make sure we have the minimum number of boards required
    total_area = stack_prices[:, 0] * solution
    if total_area.sum() < desired_boards:
        return float('-inf')

    # Calculate the price of the solution
    price = stack_prices[:, 1] * solution
    # The fitness function maximizes so invert the price
    return - price.sum()
```

The fitness functions for PyGAD optimize for the highest number; since we are looking for the lowest price, we can simply invert the price. Additionally, we can return minus infinity when we want to rule out "bad" solutions.

To get some intermediate results, we can optionally add a function that will show us the state at every generation:

```
def print_status(instance):
    # Only print the status every 100 iterations
    if instance.generations_completed % 100:
        return

    total = 0
    solution = instance.best_solution()[0]
```

```
    # Print the generation, bulk size, and the total price
    print(f'Generation {instance.generations_completed}', end=' ')
    for mp, (boards, price) in zip(solution, stack_prices):
        print(f'{mp:2d}x{boards},', end='')
        total += mp * price
    print(f' price: ${total}')
```

Now it's time to run the algorithm and show the output while doing so:

```
ga_instance = pygad.GA(
    num_generations=1000,
    num_parents_mating=10,
    # Every generation will have 100 solutions
    sol_per_pop=100,
    # We use 1 gene per stack size
    num_genes=stack_prices.shape[0],
    fitness_func=fitness_function,
    on_generation=print_status,
    # We can't buy half a board, so use integers
    gene_type=int,
    # Limit the solution space to our maximum number of boards
    gene_space=numpy.arange(desired_boards),
    # Limit how large the change in a mutation can be
    random_mutation_min_val=-2,
    random_mutation_max_val=2,
    # Disable crossover since it does not make sense in this case
    crossover_probability=0,
    # Set the number of genes that are allowed to mutate at once
    mutation_num_genes=stack_prices.shape[0] // 2,
)

ga_instance.run()
ga_instance.plot_fitness()
```

This will run 1,000 generations for us with 100 solutions per generation. A single solution contains the number of stacks of wood to buy for each stack size. When we run this code, we should get something similar to this:

```
$ python3 T_02_pygad.py
Generation 100  3x1, 1x5, 6x10, 0x25, price: $555
Generation 200  3x1, 0x5, 4x10, 1x25, price: $525
...
```

```
Generation 900  2x1, 1x5, 1x10, 2x25, price: $495
Generation 1000  2x1, 1x5, 1x10, 2x25, price: $495
```

The plot of our results:

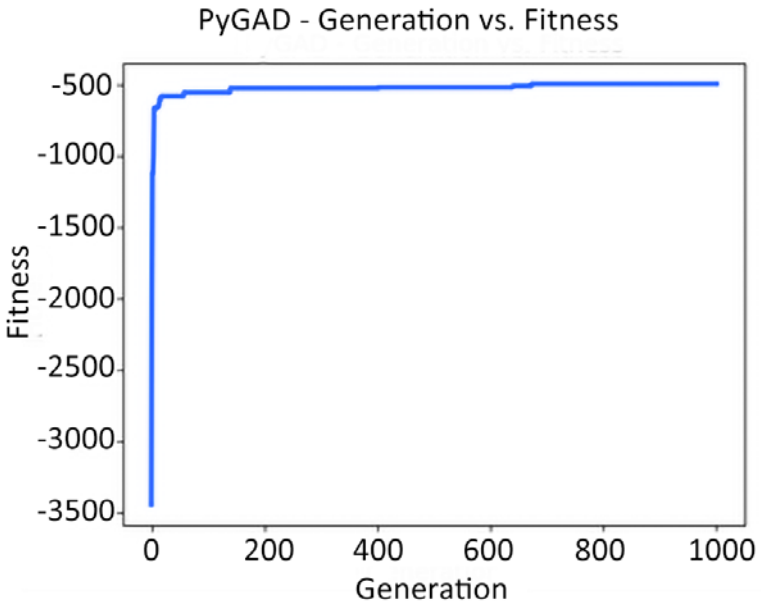


Figure 16.8: Genetic algorithm fitness result plot

In this case, 495 is actually the optimal result; in most cases, though, you don't know if you have reached the optimal result. This essentially means that you could keep your code running forever, which is why you should either configure a fixed number of generations, or tell PyGAD to stop once it has reached a steady state for a certain number of generations.

More importantly, however, after about 50 generations we already had a great and very usable solution for our problem, whereas the optimal solution took roughly 700 generations this run. In many of the other runs I did, it never even found the optimal solution. This shows you how quickly the genetic algorithm can give you a useful result.

Support-vector machines

Support-vector machines (**SVMs**) or support-vector networks are common models for supervised learning. Since it is a supervised learning method, it expects a dataset that is already labeled (for example, a list of photos with correct labels) to train on. Once the model has been trained, it can be used for classification and regression analysis.

In statistics, **regression analysis** is a way to show the relationship between variables. These can be used to fit lines, create predictors, detect outliers, and more. We have seen several examples of regression analysis in *Chapter 15, Scientific Python and Plotting*, as well.

Classification refers to statistical classification and is a method of splitting data. For example, the question as to whether an email is spam or not is a form of binary classification.

Bayesian networks

Bayesian networks are based on the idea that we have probabilities of an event occurring. This is usually expressed as `P(event)`, where `P(event)=1` is 100% probability of `event` occurring and `P(event)=0` is no probability at all.

These can be used for all sorts of applications and are particularly useful for expert systems, which can make recommendations based on your given data. For example, given that there is a thunderstorm outside, we know that there is a larger probability of rain than if it is sunny outside. In Bayesian terms, we would describe it like this:

```
P(rain) = The probability of rain
P(thunderstorm) = The probability of a thunderstorm
P(rain | thunderstorm) = The probability of rain given that there is a
thunderstorm
```

Bayesian networks are often used for spam filters that look for certain keywords and calculate the odds of an email being spam. Another possible use case for Bayesian networks is text prediction when typing. If you train your network with many sentences, you can calculate the next most likely word to occur given the previous word or words.

As you have seen, there are many types of different machine learning models, and many more submodels that all have their own strengths and weaknesses. This list of examples is an extremely condensed and simplified list of available models, but it should give you at least some idea of the scenarios in which these different algorithms can do their magic.

Versatile AI libraries and utilities

Python is by far the most popular language when it comes to developing AI systems. The result of this popularity is that there are a huge number of libraries available for every branch of AI you can think of. There is at least a single good library for nearly every AI technique, and often dozens.

In this section of this chapter, you will find a curated (and incomplete) list of useful AI libraries split into segments. There are many more that are not mentioned due to being too specific, too new, or simply because I have omitted them owing to the great number of libraries that are out there.

scikit-learn – Machine learning in Python

The scikit-learn library is an extremely versatile machine learning library that covers many AI topics; for many of them, this should be your starting point. We have already seen the scikit-image library earlier, which is a part of the scikit-learn project, but there are many more options.

The complete list of possibilities is huge, so I will try to give you a very small list based on the scikit-learn modules that I have personally found useful. Many more methods are available, so make sure to read through the scikit-learn documentation if you are interested in anything specific.

This section is split between supervised and unsupervised options, since your dataset is the most important factor in deciding on an algorithm for your use case.

Supervised learning

Starting with supervised learning, scikit-learn offers a host of different options in many different categories.

Linear models

First of all, scikit-learn offers dozens of different linear models for performing many types of regressions. It has functions for many specific use cases, such as:

- **Ordinary least squares** regression, as we have seen several times in the previous chapter as well.
- **Ridge regression and classifier**, a function similar to the ordinary least squares method but more resistant to collinearity.
- The **LASSO (least absolute shrinkage and selection operator) model**, which can be seen as the successor to the Ridge model for specific use cases. One of the advantages of the lasso model is that, in the case of machine learning, it can help filter out (usually irrelevant) features with very little data.
- **Polynomial regression**: Methods such as the ordinary least squares method perform regression by creating a single straight line. In some cases, however, a straight line will never properly fit your data. In these cases, polynomial regression can help a lot since it can generate curved lines.

There are many more methods in this module, so make sure to take a look at the documentation: `https://scikit-learn.org/stable/modules/linear_model.html`.

Support-vector machines

Next up are support-vector machines. We already discussed SVMs briefly, but in short these can be used for classification, regression, and outlier detection. As opposed to the linear (2D) models above, these methods also function for higher-dimensional data.

Currently, scikit-learn supports these types of SVMs:

- `SVC/SVR`: Support-vector classification and regression based on the C `libsvm` library. For smaller datasets (a few thousand samples), this is the most useful and flexible SVM implementation in scikit-learn. This method can also handle support vectors, which can increase the precision of classifiers.
- `NuSVC/NuSVR`: A modified version of SVC/SVR that introduces a parameter ν (the Greek letter Nu) to approximate the fraction of training errors and support vectors.

- `LinearSVC`/`LinearSVR`: A fast (faster than `SVC`/`SVR`) linear support vector classification and regression system. For large datasets (over 10,000 samples) this is the better alternative to `SVC`/`SVR`, but it does not handle separate support vectors.

SVMs are very robust prediction methods for higher-dimensional data while still maintaining decent execution speeds.

Decision trees

Decision trees (**DTs**) also deserve special attention. While most of the machine learning models are still relatively expensive to use after training, with DTs you build a tree based on the training data to use in your classification or regression. If you are familiar with tree structures, you know that many lookups only take `O(log(n))` time to do. In addition to being really fast to calculate, it can also make it much easier to visualize your data, because `scikit-learn` can export the evaluated results to Graphviz, a tool for rendering graph structures.

To supercharge the DTs, you can also combine a collection of them into a forest using a `RandomForestClassifier` or `RandomForestRegressor`, which results in reduced variance. To take this a step further, you can also use the **extremely randomized trees** methods `ExtraTreesClassifier` or `ExtraTreesRegressor`, which also randomize the specific thresholds between the trees, for further reduced variance over the normal forest methods.

Feature selection

Using feature selection, you can input a large number of input parameters without specifying what they are for, and let the model figure out the most important features.

For example, let's say that you have collected a large set of weather and geographical data, such as temperature, humidity, air pressure, altitude, and coordinates, and you want to know which of these play a role in answering the question of whether it will snow. The coordinates and air pressure are probably less important factors than temperature is in this case.

The scikit-learn library has several different options available for feature selection:

- `sklearn.feature_selection.VarianceThreshold`: Excludes items with a small variance by satisfying the equation Var[X]=p(1-p)
- `sklearn.feature_selection.SelectKBest`: Selects the k highest scoring features
- `sklearn.feature_selection.SelectPercentile`: Selects the top nth percentile scoring features
- `sklearn.feature_selection.SelectFromModel`: A special and very useful feature selector that can use previously generated models (such as an SVM) to filter features

There are several other feature selection and feature filtering methods available, so make sure to check the documentation to see if there is a better method available for your specific use case.

Other models

In addition to these methods, there are many other methods supported by scikit-learn, such as:

- **Bayesian networks**: Gaussian, multinomial, complement, Bernoulli, categorical, and out-of-core.
- **Linear and quadratic discriminant analysis**: These are similar to the linear models but also offer quadratic solutions.
- **Kernel ridge regression**: A combination of ridge regression and classification. This can be a faster alternative to SVR.
- **Stochastic gradient descent**: A very fast regression/classifier alternative to SVM for specific use cases.
- **Nearest neighbor**: These methods are useful for a range of different purposes and are at the core of many of the other functions in this library. At the very least, take a look at this section, because structures such as KD-trees have many applications outside of machine learning as well.

While there are several other options as well, these are probably the ones that are most useful to you. Note that even though scikit-learn does support neural networks such as multi-layer perceptrons, I would not recommend you use scikit-learn for this purpose. While the implementation works well, it does not have support for GPU (video card) acceleration, which makes a huge performance difference. For neural networks I recommend using TensorFlow, as discussed earlier in this chapter.

Unsupervised learning

Due to the nature of unsupervised learning, it is a lot less versatile than supervised learning, but there are a few scenarios where unsupervised learning absolutely makes sense and is an easy solution. While the unsupervised learning portion of scikit-learn is smaller than the supervised portion, there are still several really useful functions available.

Clustering is the prime example of where unsupervised learning shines. This comes down to giving the algorithm a whole bunch of data and telling it to cluster (split) it into useful sections wherever it can find a pattern. To facilitate this, scikit-learn has a range of different algorithms. The documentation explains this very well: `https://scikit-learn.org/stable/modules/clustering.html#overview-of-clustering-methods`.

A subsection of this documentation is given below:

Method name	Scalability	Use case
K-Means	Very large `n_samples`, medium `n_clusters` with MiniBatch code	General-purpose, even cluster size, flat geometry, not too many clusters, inductive
Affinity propagation	Not scalable with `n_samples`	Many clusters, uneven cluster size, non-flat geometry, inductive
Mean-shift	Not scalable with `n_samples`	Many clusters, uneven cluster size, non-flat geometry, inductive

Spectral clustering	Medium `n_samples`, small `n_clusters`	Few clusters, even cluster size, non-flat geometry, transductive
Ward hierarchical clustering	Large `n_samples` and `n_clusters`	Many clusters, possibly connectivity constraints, transductive
Agglomerative clustering	Large `n_samples` and `n_clusters`	Many clusters, possibly connectivity constraints, non-Euclidean distances, transductive
DBSCAN	Very large `n_samples`, medium `n_clusters`	Non-flat geometry, uneven cluster sizes, transductive
OPTICS	Very large `n_samples`, large `n_clusters`	Non-flat geometry, uneven cluster sizes, variable cluster density, transductive
Gaussian mixtures	Not scalable	Flat geometry, good for density estimation, inductive
BIRCH	Large `n_clusters` and `n_samples`	Large dataset, outlier removal, data reduction, inductive

All of these methods have their own use cases and the scikit-learn documentation explains this much better than I could. In general, however, the K-Means algorithm, which we have also used in the previous chapter, is a very good starting point.

Note that the clusters can also be used for learning features and the relationship between them. Once you have learned the features, you could use the feature selection in supervised learning to filter them for subselections.

To summarize, for the general machine learning cases, scikit-learn is probably your best bet. For special cases, there are often better libraries available; many of these are built on top of scikit-learn, however, so it is recommended that you familiarize yourself with the library if you plan to employ machine learning.

auto-sklearn — Automatic scikit-learn

The scikit-learn library can do so many things that it is often overwhelming to use. At the time of writing, there are 34 distinct regression functions and 25 different classifiers, which can make it quite a challenge to select the right one for you.

This is where `auto-sklearn` can help. It can automatically select a classification function for you and fill in the parameters needed for it to work. If you're just looking for something that just works, this is your best bet.

mlxtend — Machine learning extensions

`mlxtend` is a library with a range of relatively easy and well-documented machine learning examples.

It uses `scikit-learn`, pandas, and `matplotlib` internally to provide a more user-friendly interface for machine learning compared to scikit-learn. If you are starting out with machine learning (or scikit-learn), this can be a nice introduction, since it's a bit less complicated than using scikit-learn directly.

scikit-lego — scikit-learn utilities

Even though scikit-learn already has a huge catalog of functions and features built in, there are still many things that it does not provide an easy interface for. This is where the scikit-lego library can help, it has many convenient functions for scikit-learn and pandas so you don't need to repeat yourself too often.

In the previous chapter, we used the Penguins dataset a few times. Loading that dataset and plotting the distribution can be done in just a few lines:

```
import collections

from matplotlib import pyplot as plt
from sklego import datasets

X, y = datasets.load_penguins(return_X_y=True)
counter = collections.Counter(y)
plt.bar(counter.keys(), counter.values())
plt.show()
```

This results in:

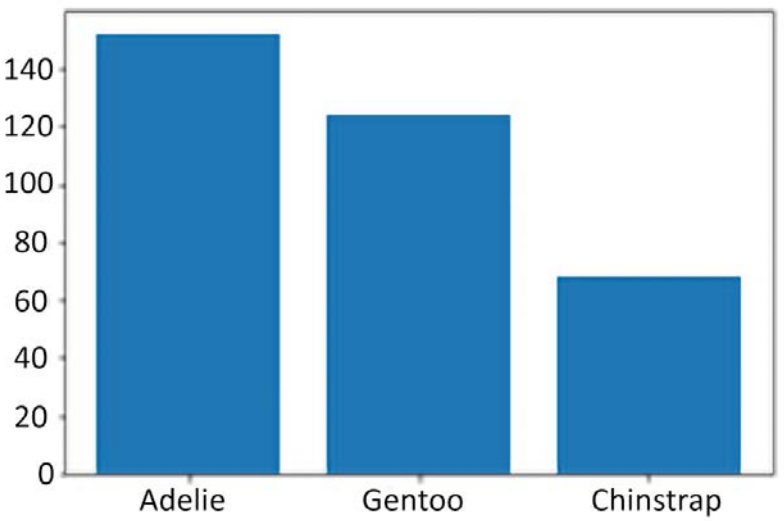


Figure 16.9: Penguin distribution

scikit-lego can automatically perform some conversions for us (the `return_X_y` parameter here) so we can easily plot the results. There are many more of these functions available, which make it really easy to play around with Scikit-learn.

XGBoost — eXtreme Gradient Boosting

XGBoost is a fast and efficient library for gradient boosting, a regression/classification technique that produces forests of decision trees. The main advantage of this technique compared to many other regression/classification algorithms is the **scalability**. With XGBoost, you can easily spread your workload along clusters of many computers, and it happily scales to billions of data points.

If you have very large datasets, XGBoost might be one of your best options.

Featuretools — Feature detection and prediction

The Featuretools library makes it really easy to transform your datasets into aggregated feature matrices based on either time-based datasets or relational ones. Once the feature matrix is constructed, the library can be used for predictions about these features.

You could, for example, predict trip durations based on a collection of multiple trips, or predict when a customer will purchase from you again.

Snorkel — Improving your ML data automatically

Snorkel is a library that attempts to make the training of your ML models much easier. Getting enough training data to properly train your models can be really difficult, and this library has several clever methods to make this easier.

The library has three core operations to help you build your datasets:

- First, to help with labeling, the Snorkel library features several heuristic methods. While these labels will not be perfect, manually labeling all data can be a prohibitive task.
- The second core operation is the transforming and augmenting of datasets. Once again, these use heuristic methods to (hopefully) improve your data quality.
- The last core operation is the slicing of data so you only get data that is relevant for your use case. This operation is also heuristics-based.

You will not need this if you already have good-quality data available, but it is certainly worth looking at if your data could use some improvement. As is always the case with machine learning, care must be taken to avoid overfitting or underfitting data. Applying the Snorkel methods can quickly exacerbate problems in your dataset, since it uses the dataset as a source.

TPOT — Optimizing ML models using genetic programming

TPOT (tea-pot) is a library that optimizes your learning pipelines through genetic programming. We already covered evolutionary algorithms earlier, but to remind you, they are algorithms that improve by changing themselves or their parameters through evolution.

While genetic algorithms are relatively easy to implement by themselves, the complexity comes from the encoding of the solution so it is compatible with the genetic algorithm. This is what is very nice about the TPOT library; it makes it really easy to encode your features, cache parts of the pipeline, and even run the attempts in parallel using Dask.

To illustrate, here is the code needed to tell TPOT to automatically optimize a scikit-learn classifier with its parameters:

```
import tpot
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    some_data, some_target, train_size=0.75, test_size=0.25)

tpot = tpot.TPOTClassifier(verbosity=2, max_time_mins=10)
tpot.fit(X_train, y_train)
tpot.export('optimized_classifier.py')
```

Once it is done trying multiple classifiers, it will write the optimized function calls to `optimized_classifier.py`. It is important to note that the classifier returned is also dependent on the optimizer results; it could be `sklearn.neighbors.KNeighborsClassifier`, but you could also get `sklearn.ensemble.RandomForestClassifier` or something else.

Do not assume that `TPOT` is a fast solution for finding your parameters, though; getting a good solution using genetic algorithms can take a long time, and it can be beneficial to reduce your test set before you apply this algorithm.

That was the last library, and it's time to try things out for yourself in the *Exercises* section.

Exercises

Due to the nature of this chapter, all topics only cover the absolute basics of the mentioned libraries and they really do deserve much more. In this case, as an exercise, I recommend that you try and use some (or all) of the mentioned libraries and see if you can do something useful with them.

Some suggestions:

- Browse through TensorFlow Hub and apply some models to your own data. Perhaps you can apply object detection to your holiday photos.
- After applying a model to your photos, try and improve the model by adding some new objects and finetuning it.
- Try to extract some data or information from this chapter's summary by applying one of the NLP algorithms.

AI is a complicated subject, and even simple example guides are often quite elaborate. Luckily, these days we can often immediately try examples online through Google Colab or by running a Jupyter Notebook. Dive in and don't get discouraged; there is an incredible amount of high-quality information available online from field experts.

Example answers for these exercises can be found on GitHub: `https://github.com/mastering-python/exercises`. You are encouraged to submit your own solutions and learn about alternative solutions from others.

Summary

This chapter gave you a sample of some of the largest and most popular Python AI libraries, but there are many more (large) libraries around that could be useful for your particular use case. There are, for example, also many libraries available for specific topics such as astronomy, geographical information systems (GISes), protein folding, and neurological imaging.

After this chapter, you should have some idea of where to start searching for particular types of AI libraries. Additionally, you should know a little bit about when to apply a particular type of AI. For many use cases, you will need a combination of these methods to solve the problem in an efficient manner. A supervised ML system, for example, is a fantastic option if you have a vast amount of good-quality, labeled data. Often this is not the case, which is where the other algorithms come in.

Surprisingly enough, many of the current "AI" start-up companies don't actually use AI for their recommendation systems but humans instead, hoping to upgrade to an effective AI somewhere in the future when they have gathered enough training data. Effectively, they are trying to solve the data requirement for supervised ML systems with brute force. Similarly, algorithms are only part of the reason that voice recognition systems such as Alexa, Google Assistant, or Siri have become possible. Another large part is the availability of training data over the last several years. Naturally, these systems are not built on one algorithm specifically but use a combination of multiple algorithms; the system not only tries to convert your voice to words, but also attempts to understand what you are likely to say by constantly cross-validating those results with what would be a logical sentence structure.

The field of AI is improving and changing more rapidly with each year. With the increased processing power we have now, there are many more options than we had in the past. The currently used deep learning AI models were completely infeasible to build only 20 years ago, and in 10 years' time the models will have far surpassed what is possible now. If there is no solution available for the issue you are facing today, the situation might be completely different a year from now.

It is also perfectly reasonable to skip this part of Python entirely. While AI is becoming a larger and larger portion of what is being done with Python, a big part of that is in academic settings and might not be interesting for your field of work. AI can be a great help, but it is often a much more complicated solution than is actually needed.

In the next chapter, we will learn about creating extensions in C/C++ to increase performance and allow low-level access to memory and other hardware resources. While this can greatly help with performance, performance rarely comes free, as we will see.

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:
`https://discord.gg/QMzJenHuJf`

17

Extensions in C/C++, System Calls, and C/C++ Libraries

The last few chapters have shown us many machine learning and scientific computing libraries. Many of these libraries are not written in pure Python because of code reuse from existing libraries, or for performance reasons. In this chapter, we will learn how we can do some of this ourselves by creating C/C++ extensions.

In *Chapter 12*, *Performance – Tracking and Reducing Your Memory and CPU Usage*, we saw that the `cProfile` module is about 10 times faster than the `profile` module, which indicates that at least some C extensions are faster than their pure Python equivalents. This chapter will not focus on performance that much, however. The goal here is interaction with non-Python libraries. To paraphrase Linus Torvalds, any performance improvement will just be a completely unintentional side effect.

If performance is your main goal, you really should not be looking at writing a C/C++ extension manually. For the Python core modules, that was done, of course, but in most practical applications you are far better off using `numba` or `cython`. Or, if the use case allows, use pre-existing libraries such as `numpy` or `jax`. The main reason for using the tools in this chapter should be to reuse existing libraries so you don't have to reinvent the wheel.

We will discuss the following topics in this chapter:

- `ctypes` for handling foreign (C/C++) functions and data from Python
- **C Foreign Function Interface** (**CFFI**), similar to `ctypes`, but with a slightly different approach
- Writing native C/C++ to extend Python

Setting up tooling

Before we begin, it is important to note that this chapter will require a working compiler that plays nicely with your Python interpreter. Unfortunately, these vary from platform to platform. For Linux distributions, this can usually be achieved with one or two commands without much hassle.

For OS X, the experience is often very similar, mostly because the heavy lifting can be offloaded to package management systems such as Homebrew. For Windows, it can be slightly trickier, but that process has been streamlined over the last few years as well.

A good and up-to-date starting point to get the required tooling is the Python Developer's Guide: `https://devguide.python.org/setup/`.

For building the actual extensions, the Python manual can be useful: `https://docs.python.org/3/extending/building.html`.

Do you need C/C++ modules?

In almost all cases, I'm inclined to say that you don't need C/C++ modules. If you are really strapped for best performance, then there are almost always highly optimized Python libraries available that use C/C++/Fortran/etc. internally and fit your purpose. There are some cases where native C/C++ (or just "not Python") is a requirement. If you need to communicate directly with hardware that has specific timings, then Python might not do the trick. Generally, however, that kind of communication should be left to an operating system kernel-level driver that takes care of the specific timings. Regardless, even if you will never write one of these modules yourself, you might still need to know how they work when you are debugging a project.

Windows

For Windows, the general recommendation is Visual Studio. The specific version depends on your Python version:

- **Python 3.4**: Microsoft Visual Studio 2010
- **Python 3.5 and 3.6**: Microsoft Visual Studio 2015 or Visual Studio 2017
- **Python 3.7–3.10**: Microsoft Visual Studio 2017

Visual Studio 2019 is also supported, but the official builds of Python 3.7 to Python 3.10 still use Visual Studio 2017, making that the recommended solution.

The specifics of installing Visual Studio and compiling Python modules fall somewhat outside the scope of this book. Luckily, the Python documentation has some documentation available to get you started: `https://devguide.python.org/setup/#windows`.

If you are looking for a more Linux/Unix-like solution, you can also choose to use the GCC compiler through MinGW.

OS X

For a Mac, the process is mostly straightforward, but there are a few tips specific to OS X. First, install Xcode through the Mac App Store.

Once you have done that, you should be able to run the following command:

```
$ xcode-select --install
```

Next up is the fun part. Because OS X comes with a bundled Python version (which is generally out of date), I would recommend installing a new Python version through Homebrew instead. The most up-to-date instructions for installing Homebrew can be found on the Homebrew home page (`http://brew.sh/`), but the gist of installing Homebrew is this command:

```
$ /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/
install/HEAD/install.sh)"
```

After that, make sure you check whether everything is set up correctly using the `doctor` command:

```
$ brew doctor
```

When all of this is done, simply install Python through Homebrew and make sure you use that Python release when executing your scripts:

```
$ brew install python3
$ python3 --version
Python 3.9.7
which python3
/usr/local/bin/python3
```

Also ensure that the Python process is in `/usr/local/bin`, that is, the Homebrewed version. The regular OS X version would be in `/usr/bin/` instead.

Linux/Unix

The installation for Linux/Unix systems greatly depends on the distribution, but it is generally simple to do.

For Fedora, Red Hat, CentOS, and other systems that use `yum` as the package manager, use these lines:

```
$ sudo yum install yum-utils
$ sudo yum-builddep python3
```

For Debian, Ubuntu, and other systems that use `apt` as the package manager, use the following line:

```
$ sudo apt-get build-dep python3.10
```

Note that Python 3.10 is not available everywhere yet, so you might need Python 3.9 or even Python 3.8 instead.

For most systems, to get help with the installation, a web search along the lines of `<operating system> python.h` should do the trick.

Calling C/C++ with ctypes

The `ctypes` library makes it easily possible to call functions from C libraries, but you do need to be careful with memory access and data types. Python is generally very lenient in memory allocation and type-casting; C is, most definitely, not that forgiving.

Platform-specific libraries

Even though all platforms will have a standard C library available somewhere, the location and the method of calling it differs per platform. For the purpose of having a simple environment that is easily accessible to most people, I will assume the use of an Ubuntu (virtual) machine. If you don't have a native Ubuntu machine available, you can easily run it through VirtualBox on Windows, Linux, and OS X.

Since you will often want to run examples on your native system instead, we will first show the basics of loading `printf` from the standard C library.

Windows

One problem of calling C functions from Python is that the default libraries are platform-specific. While the following example will work just fine on Windows systems, it won't run on other platforms:

```
>>> import ctypes

>>> ctypes.cdll
<ctypes.LibraryLoader object at 0x...>
>>> libc = ctypes.cdll.msvcrt
>>> libc
<CDLL 'msvcrt', handle ... at ...>
>>> libc.printf
<_FuncPtr object at 0x...>
```

The `ctypes` library exposes the functions and attributes of the C/C++ library (`MSVCRT.DLL` in this case) to your Python installation. Since the `ms` part of `msvcrt` stands for Microsoft, this is one library you generally won't find on non-Windows systems.

There is a difference between Linux/Unix and Windows in loading as well; on Windows, the modules will generally be auto-loaded, while on Linux/Unix systems, you will need to load them manually, because these systems will often have multiple versions of the same library available.

Linux/Unix

Calling standard system libraries from Linux/Unix does require manual loading, but it's luckily nothing too involved. Fetching the `printf` function from the standard C library is quite simple:

```
>>> import ctypes

>>> ctypes.cdll
<ctypes.LibraryLoader object at 0x...>
>>> libc = ctypes.cdll.LoadLibrary('libc.so.6')
```

```
>>> libc
<CDLL 'libc.so.6', handle ... at ...>
>>> libc.printf
<_FuncPtr object at 0x...>
```

OS X

For OS X, explicit loading is also required, but beyond that, it is quite similar to how everything works on regular Linux/Unix systems:

```
>>> import ctypes

>>> libc = ctypes.cdll.LoadLibrary('libc.dylib')
>>> libc
<CDLL 'libc.dylib', handle ... at 0x...>
>>> libc.printf
<_FuncPtr object at 0x...>
```

Making it easy

Besides the way libraries are loaded, there are more differences, unfortunately, but the earlier examples at least give you the standard C library, which allows you to call functions such as `printf` straight from your C implementation. If, for some reason, you have trouble loading the right library, there is always the `ctypes.util.find_library` function.

As always, I would recommend explicit over implicit declarations, but things can be made easier in some cases using this function. To illustrate a run on an OS X system:

```
# OS X
>>> from ctypes import util
>>> from ctypes import cdll

>>> library = util.find_library('libc')
>>> library
'/usr/lib/libc.dylib'

# Load the library
>>> libc = cdll.LoadLibrary(library)
>>> libc
<CDLL '/usr/lib/libc.dylib', handle ... at 0x...>
```

Calling functions and native types

Calling a function through `ctypes` is nearly as simple as calling native Python functions. The notable difference is the arguments and `return` statements. These should be converted to native C variables.

These examples assume that you have `libc` in your scope from one of the examples in the previous paragraphs.

We will now create a C string that is effectively a memory block, with the characters as ASCII characters and terminated with a null character. After creating the C string, we will run `printf` on the string:

```
>>> c_string = ctypes.create_string_buffer(b'some bytes')
>>> ctypes.sizeof(c_string)
11
>>> c_string.raw
b'some bytes\x00'
>>> c_string.value
b'some bytes'
>>> libc.printf(c_string)
10
some bytes>>>
```

This output might look a bit confusing initially, so let's analyze it. When we call `libc.printf` on `c_string`, it will write the string to `stdout` directly. Because of this, you can see that the output is interleaved (`some bytes>>>`) with the Python output, as this circumvents the Python output buffering and Python does not know this is happening. Additionally, you can see that `libc.printf` returned `10`, which is the number of bytes written to `stdout`.

To call the `printf` function, you *must*—and I cannot stress this enough—convert your values from Python to C explicitly. While it might appear to work without this initially, it really doesn't:

```
>>> libc.printf(123)
segmentation fault (core dumped)  python3
```

Remember to use the `faulthandler` module from *Chapter 11*, *Debugging – Solving the Bugs*, to debug `segfaults`.

Another thing to note from the example is that `ctypes.sizeof(c_string)` returns `11` instead of `10`. This is caused by the trailing null character that C strings require, which is visible in the raw property of the C string.

Without it, the string functions in C such as `printf` won't know where the string will end, since a C string is just a block of bytes in memory and C only knows at what memory address the string starts; the end is indicated by the null character. This is why memory management in C requires paying a lot of attention.

If you allocate a string of size 5 and write 10 bytes to it, you will be writing into the memory outside of your variable, which could be another function, another variable, or outside of your program's memory. This would result in a segmentation fault.

Python will generally protect you from silly mistakes; C and C++ most certainly won't. To quote Bjarne Stroustrup (the creator of C++):

"C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do, it blows away your whole leg."

As opposed to C, C++ does have a string type to protect you in these cases. However, it is still a language where you have easy access to memory addresses and mistakes are easily made.

To pass along other types (such as integers) toward `libc` functions, we have to use some conversion as well. In some cases, it is optional:

```
>>> format_string = b'Number: %d\n'
>>> libc.printf(format_string, 123)
Number: 123
12

>>> x = ctypes.c_int(123)
>>> libc.printf(format_string, x)
Number: 123
12
```

But not in all cases, so caution is advised, and explicitly converting is the safer option:

```
>>> format_string = b'Number: %.3f\n'
>>> libc.printf(format_string, 123.45)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ctypes.ArgumentError: argument 2: <class 'TypeError'>: Don't know how to
convert parameter 2

>>> x = ctypes.c_double(123.45)
>>> libc.printf(format_string, x)
Number: 123.450
16
```

It's important to note that even though these values are usable as native C types, they are still mutable through the `value` attribute:

```
>>> x = ctypes.c_double(123.45)
>>> x.value
123.45
```

```
>>> x.value = 456
>>> x
c_double(456.0)
```

This is the case unless the original object was immutable, which is a very important distinction to make. The `create_string_buffer` object creates a mutable string object, whereas `c_wchar_p`, `c_char_p`, and `c_void_p` create references to the actual Python string. Since strings are immutable in Python, these values are also immutable. You can still change the `value` property, but it will only assign a new string. Passing one of these immutable variables to a C function that mutates the internal value will result in unpredictable behavior and/or crashes.

The only values that should convert to C without any issues are integers, strings, and bytes, but I personally recommend that you always convert all of your values so that you are certain of which type you will get and how to treat it.

Complex data structures

We have already seen that we can't just pass along Python values to C, but what if we need more complex objects such as classes or tuples? Luckily, we can easily create (and access) C structures using `ctypes`:

```
>>> from _libc import libc
>>> import ctypes

>>> class ComplexStructure(ctypes.Structure):
...     _fields_ = [
...         ('some_int', ctypes.c_int),
...         ('some_double', ctypes.c_double),
...         ('some_char', ctypes.c_char),
...         ('some_string', ctypes.c_char_p),
...     ]
...
>>> structure = ComplexStructure(123, 456.789, b'x', b'abc')
>>> structure.some_int
123
>>> structure.some_double
456.789
>>> structure.some_char
b'x'
>>> structure.some_string
b'abc'
```

This supports any of the fundamental data types such as integers, floating-point numbers, and strings. Nesting is also supported; for instance, other structures could use `ComplexStructure` instead of `ctypes.c_int` in this example.

Arrays

Within Python, we generally use a `list` to represent a collection of objects. These are very convenient in that you can easily add and remove values. Within C, the default collection object is the **array**, which is just a block of memory with a fixed size.

The size of the block in bytes is decided by multiplying the number of items by the size of the type. In the case of a `char`, this is `8` bits, so if you wish to store `100` chars, you would have `100 * 8 bits = 800 bits = 100 bytes`.

This is literally all it is—a block of memory—and the only reference you receive from C is a pointer to the memory address where the block of memory begins. Since the pointer does have a type, `char*` in this case, C will know how many bytes to jump ahead when trying to access a different item. Effectively, when trying to access item 25 in a `char` array, you simply need to do `array_pointer + 24 * sizeof(char)`. This has a convenient shortcut: `array_pointer[24]`. Note that we need to access index 24 because we start counting at 0, just like with Python collections such as lists and strings.

Note that C does not store the number of items in the array, so even though our array has only 100 items, it won't block us from doing `array_pointer[1000]` and reading other (random) memory. At some point, however, you will go outside of the reserved memory of your application and your operating system will punish you with a segmentation fault.

If you take all of these limitations into account, C arrays are definitely usable but mistakes are quickly made and C is unforgiving. No warnings; just crashes and strangely behaving code. Beyond that, let's see how easily we can declare an array with `ctypes`:

```
>>> TenNumbers = 10 * ctypes.c_double
>>> numbers = TenNumbers()
>>> numbers[0]
0.0
```

As you can see, because of the fixed sizes and the requirement of declaring the type before using it, its usage is slightly awkward. However, it does function as you would expect. Additionally, as opposed to regular C, the values are initialized to zero by default and it will protect you from out-of-bound errors when accessing from Python. Naturally, this can be combined with our previously created custom structures as well:

```
>>> GrossComplexStructures = 144 * ComplexStructure
>>> complex_structures = GrossComplexStructures()

>>> complex_structures[10].some_double = 123
>>> complex_structures[10]
<__main__.ComplexStructure object at ...>
>>> complex_structures
<__main__.ComplexStructure_Array_144 object at ...>
```

Even though you cannot simply append to these arrays to resize them, they are actually resizable with a few constraints. Firstly, the new array needs to be larger than the original array. Secondly, the size needs to be specified in bytes, not items. To illustrate, we have this example:

```
>>> TenNumbers = 10 * ctypes.c_double
>>> numbers = TenNumbers()

>>> ctypes.resize(numbers, 11 * ctypes.sizeof(ctypes.c_double))
>>> ctypes.resize(numbers, 10 * ctypes.sizeof(ctypes.c_double))
>>> ctypes.resize(numbers, 9 * ctypes.sizeof(ctypes.c_double))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: minimum size is 80

>>> numbers[:5] = range(5)
>>> numbers[:]
[0.0, 1.0, 2.0, 3.0, 4.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

As a starting point, the `TenNumbers` array has 10 items. Next up, we try to resize the array to 11, which works because it's more than the original 10. Resizing back to 10 is also allowed, but resizing to 9 items is not allowed because that is fewer than the 10 items we had originally.

Lastly, we mutate a slice of items simultaneously, which works as you would expect.

Gotchas with memory management

Besides the obvious memory allocation issues and mixing mutable and immutable objects, there is one more non-obvious memory mutability issue.

In regular Python, we can do something like `a, b = b, a` and it will work as you would expect because Python uses internal temporary variables. With regular C, you do not have that luxury, unfortunately; with `ctypes`, you do have the benefit of Python taking care of the temporary variable for you, but sometimes that can still go wrong:

```
>>> import ctypes

>>> class Point(ctypes.Structure):
...     _fields_ = ('x', ctypes.c_int), ('y', ctypes.c_int)

>>> class Vertex(ctypes.Structure):
...     _fields_ = ('c', Point), ('d', Point)

>>> a = Point(0, 1)
>>> b = Point(2, 3)
>>> a.x, a.y, b.x, b.y
(0, 1, 2, 3)
```

```
# Swap points a and b
>>> a, b = b, a
>>> a.x, a.y, b.x, b.y
(2, 3, 0, 1)

>>> v = Vertex()
>>> v.c = Point(0, 1)
>>> v.d = Point(2, 3)
>>> v.c.x, v.c.y, v.d.x, v.d.y
(0, 1, 2, 3)

# Swap points c and d
>>> v.c, v.d = v.d, v.c
>>> v.c.x, v.c.y, v.d.x, v.d.y
(2, 3, 2, 3)
```

With the first example, we get `2, 3, 0, 1` when swapping `a` and `b`, as expected. With the second example, we get `2, 3, 2, 3` instead. The problem is that these objects are copied to a temporary buffer variable, but the objects themselves are being changed in the meantime.

Let's elaborate for a bit more clarity. With Python, when you do `a, b = b, a`, it will effectively run `temp = a; a = b; b = temp`. That way, the replacement works as expected and you will receive the correct values in `a` and `b`.

When you execute `a, b = b, a` in C, you effectively get `a = b; b = a`. By the time the `b = a` statement is executed, the value for `a` has already been changed by the `a = b` statement, so both `a` and `b` will have the original value of `b` at that point.

CFFI

The `CFFI` (C Foreign Function Interface) library offers options very similar to `ctypes`, but it's a bit more direct. Unlike the `ctypes` library, a C compiler is really a necessity for CFFI. With it comes the opportunity to directly call your C compiler from Python in an easy way. We illustrate by calling `printf`:

```
>>> import cffi

>>> ffi = cffi.FFI()
>>> ffi.cdef('int printf(const char* format, ...);')
>>> libc = ffi.dlopen(None)
>>> arg = ffi.new('char[]', b'Printing using CFFI\n')
>>> libc.printf(arg)
20
Printing using CFFI
```

Okay... so that looks a bit weird, right? We had to define how the `printf` function looks and specify the arguments to `printf` with a valid C function header. Additionally, we had to specify the C-string as a `char[]` array manually. With `ctypes`, that would not be required, but there are several advantages to `CFFI` as opposed to `ctypes`.

With CFFI, we can directly control what is sent to the C compiler, which gives us much more control over what is happening internally compared to `ctypes`. This means you can exactly control what types you feed the functions and what types you are returning, and you can use C macros.

Additionally, CFFI allows for easy reuse of existing C code. If the C code you are using has several `struct` definitions, you don't have to manually map them to a `ctypes.Structure` class; you can use the `struct` definition straightaway. You can even write C code directly in your Python code and CFFI will take care of calling the compiler and building the library for you.

Getting back to the declarations, you may notice that we called `ffi.dlopen` with a `None` parameter. When you pass `None` to this function, it will automatically load the entire C namespace; on non-Windows systems, at least. On Windows systems, you will need to explicitly tell CFFI which library to load.

If you remember the `ctypes.util.find_library` function, you can use that again in this case, depending on your operating system:

```
>>> from ctypes import util
>>> import cffi

# Initialize the FFI builder
>>> ffi = cffi.FFI()

# Find the libc library on OS X. Look back at the ctypes examples
# for other platforms.
>>> library = util.find_library('libc.dylib')
>>> library
'/usr/lib/libc.dylib'

# Load the library
>>> libc = ffi.dlopen(library)
>>> libc
<cffi.api._make_ffi_library.<locals>.FFILibrary object at ...>

# We do have printf available, but CFFI requires a signature
>>> libc.printf
Traceback (most recent call last):
  ...
AttributeError: printf
```

```
# Define the printf signature and call printf
>>> ffi.cdef('int printf(const char* format, ...);')
>>> libc.printf
<cdata 'int(*)(char*, ...)' ...>
```

we can see here, the workings are initially quite comparable to `ctypes` and loading the library is just as easy. The big difference is when actually calling functions and using library attributes; those need to be explicitly defined.

Luckily, the function signatures are almost always available in a C header file for your convenience so you don't need to write those yourself. And that is one of the advantages of CFFI: it allows you to reuse existing C code.

Complex data structures

The `CFFI` definitions are somewhat similar to the `ctypes` definitions, but instead of having Python emulating C, it's just plain C that is accessible from Python. In reality, it's only a small syntactical difference. While `ctypes` is a library for accessing C from Python while remaining as close to the Python syntax as possible, CFFI uses plain C syntax to access C systems, which actually removes some confusion for people experienced with C. I personally find CFFI easier to use because I have experience with C and know what is actually happening, whereas I am not always 100% certain with `ctypes`.

Let's repeat the `Vertex` and `Point` example with CFFI:

```
>>> import cffi

>>> ffi = cffi.FFI()

# Create the structures as C structs
>>> ffi.cdef('''
... typedef struct {
...     int x;
...     int y;
... } point;
...
... typedef struct {
...     point a;
...     point b;
... } vertex;
... ''')

# Create a vertex and return the pointer
>>> v = ffi.new('vertex*')

# Set the data
```

```
>>> v.a.x, v.a.y, v.b.x, v.b.y = (0, 1, 2, 3)

# Print before change
>>> v.a.x, v.a.y, v.b.x, v.b.y
(0, 1, 2, 3)

>>> v.a, v.b = v.b, v.a

# Print after change
>>> v.a.x, v.a.y, v.b.x, v.b.y
(2, 3, 2, 3)
```

As you can see, the mutable variable issues remain, but the code is just as usable. Since the struct can be copied from your C headers, the only thing that remains for you is to allocate the memory for the vertex.

In C, a regular `int` type variable `x` looks like `int x;`. A pointer to a memory address with size `int` looks like this: `int *x;`. The `int` part of the pointer tells the compiler how much memory to fetch when using the variable. To illustrate:

```
int a = 123; // Variable a contains integer 123
int* b = &a; // Variable b contains the memory address of a
int c = *b;  // Variable c contains 123, the value at memory
address c
```

The **&** operator returns the memory address for a variable and the * operator returns the value at the pointer's address.

The special workings of CFFI allow you to shortcut these operations. Normally in C, using `vertex*` would only allocate the memory for the pointer, not the `vertex` itself. In the case of CFFI, that is taken care of automatically.

Arrays

Allocation memory for new variables is almost trivial with CFFI. The previous section showed you an example of a single `struct` allocation. Let's now see how we can allocate an array of structs:

```
>>> import cffi

>>> ffi = cffi.FFI()

# Create arrays of size 10:
>>> x = ffi.new('int[10]')
>>> y = ffi.new('int[]', 10)
```

```
>>> x
<cdata 'int[10]' owning 40 bytes>
>>> y
<cdata 'int[]' owning 40 bytes>

>>> x[0:10] = range(10)
>>> list(x)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> x[:] = range(10)
Traceback (most recent call last):
    ...
IndexError: slice start must be specified

>>> x[0:100] = range(100)
Traceback (most recent call last):
    ...
IndexError: index too large (expected 100 <= 10)
```

In this case, you might wonder why the slice includes both the start and the stop. This is a requirement for CFFI. Not problematic, but a tad annoying nonetheless. Luckily, as you can see in the example above, CFFI does protect us from allocating outside of the bounds of the array.

ABI or API?

As always, there are some caveats. The examples so far have partially used the **ABI** (**application binary interface**), which loads the binary structures from the libraries. With the standard C library, this is generally safe; with other libraries, it generally isn't. The difference between the **API** (**application programming interface**) and the ABI is that the latter calls the functions at a binary level, directly addressing memory, calling memory locations directly, and expecting them to be functions.

To be able to do this, all sizes need to be consistent as well. When compiled as a 32-bit binary, a pointer will be 32 bits; when compiled as a 64-bit binary, a pointer will be 64 bits. That means that the offsets are not guaranteed to be consistent and you could be calling a random block of memory as a function.

Within CFFI, it's the difference between `ffi.dlopen` and `ffi.set_source`. Here, `dlopen` is not always safe, but `set_source` is, because it passes a compiler instead of just guessing how to call a method. The downside of using `set_source` is that you need the actual source for the library you are planning to use. Let's look at a quick example of using `ffi.set_source` to call a function we defined ourselves:

```
>>> import cffi

>>> ffi = cffi.FFI()
```

```
# In API mode, we can in-line the actual C code
>>> ffi.set_source('_sum', '''
... int sum(int* input, int n){
...     int result = 0;
...     while(n--)result += input[n];
...     return result;
... }
... ''')
>>> ffi.cdef('int sum(int*, int);')

>>> library = ffi.compile()
```

The initialization of CFFI is as normal, but instead of using `ffi.dlopen()` we are now using `ffi.set_source()` to directly pass the C code to CFFI. By doing this, CFFI can compile the library specifically for our system so we know that we will not run into ABI issues because we are creating the ABI ourselves with the call to `ffi.compile()`.

After the `ffi.compile()` step has completed, CFFI has created a `_sum.dll`, `sum.so`, or `_sum.cpython-...-os.so` file, which can be imported as a regular Python library. Now we will use the generated library:

```
# Now we can import the library
>>> import _sum

# Or use 'ffi.dlopen()' with the results from the compile step
>>> _sum_lib = ffi.dlopen(library)

# Create an array with 5 items
>>> N = 5
>>> array = ffi.new('int[]', N)
>>> array[0:N] = range(N)

# Call our C function from either the import or the dlopen
>>> _sum.lib.sum(array, N)
10

>>> _sum_lib.sum(array, N)
10
```

As you can see, both `import _sum` and `ffi.dlopen(library)` work in this case. For use in production applications, I would recommend the `import _sum` method, but the `ffi.dlopen()` method can be very convenient to use from long-running applications such as Jupyter Notebooks. If you were to use `import _sum` and make a change in the library, it would not show your changes without you first calling `reload(_sum)`.

Since this is a C function, we need to pass a C array for complex types, which is why we are using `ffi.new()` here. After that, the function call is straightforward, but since a C array does not have a notion of size, we need to pass the array size for this to work.

You can easily go out of bounds here and put in some arbitrary number instead of `N`, and the function will most likely work without crashes, but it will return very strange results because it will be summing random data in your memory.

CFFI or ctypes?

This really depends on what you are looking for. If you have a C library that you simply need to call and you don't need anything special, then `ctypes` is most likely the easier choice. If you're actually writing your own C library and trying to link to the library from Python, CFFI is probably a more convenient option.

In C/C++, linking a library means using an external pre-compiled library without requiring the source. You do need to have the header files, which contain details such as the function arguments and return types. This is exactly what we are doing when we use CFFI in ABI mode.

If you're not familiar with the C programming language, then I would definitely recommend `ctypes` or perhaps `cython`.

Native C/C++ extensions

The libraries that we have used so far only showed us how to access a C/C++ library within our Python code. Now we are going to look at the other side of the story: how C/C++ functions/modules within Python are actually written and how modules such as `cPickle` and `cProfile` are created.

A basic example

Before we can actually start with writing and using native C/C++ extensions, we have a few prerequisites. First of all, we need the compiler and Python headers; the instructions at the beginning of this chapter should have taken care of this for us. After that, we need to tell Python what to compile. The `setuptools` package mostly takes care of this, but we do need to create a `setup.py` file:

```
import pathlib
import setuptools

# Get the current directory
PROJECT_PATH = pathlib.Path(__file__).parent

sum_of_squares = setuptools.Extension('sum_of_squares', sources=[
    # Get the relative path to sum_of_squares.c
    str(PROJECT_PATH / 'sum_of_squares.c'),
])
```

```
if __name__ == '__main__':
    setuptools.setup(
        name='SumOfSquares',
        version='1.0',
        ext_modules=[sum_of_squares],
    )
```

This tells Python that we have an `Extension` object named `sum_of_squares` that will be based on `sum_of_squares.c`.

Now, let's write a function in C that sums all perfect squares (`2*2`, `3*3`, and so on) up to a given number. The Python code will be stored in `sum_of_squares_python.py` and looks like this:

```
def sum_of_squares(n):
    total = 0
    for i in range(n):
        if i * i < n:
            total += i * i
        else:
            break

    return total
```

The raw C version of this code would look something like this:

```
long sum_of_squares(long n){
    long total = 0;
    /* The actual summing code */
    for(int i=0; i<n; i++){
        if((i * i) < n){
            total += i * i;
        }else{
            break;
        }
    }

    return total;
}
```

Now that we know how the C code looks, we will create the actual C Python version that we will be using.

As we have seen with `ctypes` and `CFFI`, Python and C have different data types and some conversion needs to be done. Since the CPython interpreter is written in C, it has definitions specifically to take care of this translation step.

To load these definitions, we need to include `Python.h`, which are the CPython header files that should have everything you need.

If you look carefully, you will see that the actual summing code is identical to the C version, but we need quite a few conversion steps to make Python understand what we are doing:

```
#include <Python.h>

static PyObject* sum_of_squares(PyObject *self, PyObject
        *args){
    /* Declare the variables */
    int n;
    int total = 0;

    /* Parse the arguments */
    if(!PyArg_ParseTuple(args, "i", &n)){
        return NULL;
    }

    /* The actual summing code */
    for(int i=0; i<n; i++){
        if((i * i) < n){
            total += i * i;
        }else{
            break;
        }
    }

    /* Return the number but convert it to a Python object first */
    return PyLong_FromLong(total);
}

static PyMethodDef methods[] = {
    /* Register the function */
    {"sum_of_squares", sum_of_squares, METH_VARARGS,
     "Sum the perfect squares below n"},
    /* Indicate the end of the list */
    {NULL, NULL, 0, NULL},
};

static struct PyModuleDef module = {
    PyModuleDef_HEAD_INIT,
    "sum_of_squares", /* Module name */
```

```
    NULL, /* Module documentation */
    -1, /* Module state, -1 means global. This parameter is
           for sub-interpreters */
    methods,
};

/* Initialize the module */
PyMODINIT_FUNC PyInit_sum_of_squares(void){
    return PyModule_Create(&module);
}
```

It looks quite complicated, but it's really not that hard. There is just a lot of overhead in this case because we only have a single function. Generally, you would have several functions, in which case you only need to expand the `methods` array and create the functions. We will explain the code in more detail shortly, but first, let's look at how to run our first example. We need to build and install the module:

```
$ python3 T_09_native/setup.py build install
running build
running build_ext
building 'sum_of_squares' extension ...
...
Processing dependencies for SumOfSquares==1.0
Finished processing dependencies for SumOfSquares==1.0
```

Now, let's create a little test script to time the difference between the Python version and the C version. First, some imports and setup:

```
import sys
import timeit
import argparse
import functools

from sum_of_squares_py import sum_of_squares as sum_py

try:
    from sum_of_squares import sum_of_squares as sum_c
except ImportError:
    print('Please run "python setup.py build install" first')
    sys.exit(1)
```

Now that we have the modules imported (or got an error if you hadn't run the build step yet), we can start benchmarking:

```
if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('repetitions', type=int)
    parser.add_argument('maximum', type=int)
    args = parser.parse_args()

    timer = functools.partial(
        timeit.timeit, number=args.repetitions, globals=globals())

    print(f'Testing {args.repetitions} repetitions with maximum: '
          f'{args.maximum}')

    result = sum_c(args.maximum)
    duration_c = timer('sum_c(args.maximum)')
    print(f'C: {result} took {duration_c:.3f} seconds')

    result = sum_py(args.maximum)
    duration_py = timer('sum_py(args.maximum)')
    print(f'Py: {result} took {duration_py:.3f} seconds')

    print(f'C was {duration_py / duration_c:.1f} times faster')
```

In essence, we have a basic benchmarking script where we compare the C version to the Python version here, with a configurable number of repetitions and a maximum number to test for. Now, let's execute it:

```
$ python3 T_09_native/test.py 10000 1000000
Testing 10000 repetitions with maximum: 1000000
C: 332833500 took 0.009 seconds
Py: 332833500 took 1.264 seconds
C was 148.2 times faster
```

Perfect! Exactly the same results but much faster.

If your goal is speed alone, you should give `numba` a try instead. Adding the `@numba.njit` decorator to `sum_of_squares_python` is much easier and probably even faster.

The main advantage of writing C modules is the reuse of existing C code, however. For speedups, you are often better off with `cython`, `numba`, or converting your code to use libraries such as `numpy` or `jax`.

C is not Python — Size matters

The Python language makes programming so easy that you might forget about the underlying data structures at times; with C and C++, you can't afford to do that. Just take our example from the previous section but with different parameters:

```
$ python3 T_09_native/test.py 1000 10000000
Testing 1000 repetitions with maximum: 10000000
C sum of squares: 1953214233 took 0.003 seconds
Python sum of squares: 10543148825 took 0.407 seconds
C was 145.6 times faster
```

It's still very fast, but what happened to the numbers? The Python and C versions give different results, `1953214233` versus `10543148825`. This is caused by integer overflows in C. While Python numbers can essentially have any size, with C, a regular number has a fixed size. How much you get depends on the type you use (`int`, `long`, and so on) and your architecture (32-bit, 64-bit, and so on), but it's definitely something to be careful with. It might be hundreds of times faster in some cases, but that is meaningless if the results are incorrect.

We can increase the size a bit, of course. This makes it better:

```
typedef unsigned long long int bigint;

static PyObject* sum_of_large_squares(PyObject *self, PyObject *args){
    /* Declare the variables */
    bigint n;
    bigint total = 0;

    /* Parse the arguments */
    if(!PyArg_ParseTuple(args, "K", &n)){
        return NULL;
    }

    /* The actual summing code */
    for(bigint i=0; i<n; i++){
        if((i * i) < n){
            total += i * i;
        }else{
            break;
        }
    }

    /* Return the number but convert it to a Python object first */
    return PyLong_FromUnsignedLongLong(total);
}
```

We used `typedef` to create a `bigint` alias for `unsigned long long int`.

If we test it now, we realize that it works great:

```
$ python3 T_10_size_matters/test.py 1000 10000000
Testing 1000 repetitions with maximum: 10000000
C: 10543148825 took 0.001 seconds
Py: 10543148825 took 0.405 seconds
C was 270.3 times faster
```

And with the increased size, the difference in performance increases as well.

Making the number even larger breaks things again since even an `unsigned long long int` still has its limits:

```
$ python3 T_10_size_matters/test.py 1 100000000000000
Testing 1 repetitions with maximum: 100000000000000
C: 1291890006563070912 took 0.004 seconds
Py: 333332833333335000000 took 1.270 seconds
C was 293.7 times faster
```

So, how can you fix this? The simple answer is that you can't, and Python hasn't really fixed it either. The complex answer is that you can if you use a different data type to store your data. The C language by itself doesn't have the "big number support" that Python has.

Python supports infinitely large numbers by combining several regular numbers in the actual memory and automatically switches to those types of numbers when needed. With Python 2, that was much more obvious with the distinction between the `int` and `long` types. With Python 3, the `long` and `int` types have been merged into the `int` type. You will not notice the switchover to the `long` type; it will automatically happen in the background.

Within C, there are no commonly available provisions for this, so there is simply no easy way to get this working. But we can check for errors instead:

```
static unsigned long long int get_number_from_object(int* overflow,
        PyObject* some_very_large_number){
    return PyLong_AsLongLongAndOverflow(sum, overflow);
}
```

Note that this only works for `PyObject*`, which means it doesn't work for internal C overflows. However, you can, of course, just keep the original Python long around and perform operations on that instead. So, you do have big number support in C without too much effort.

The example explained

We have seen the results from our example, but if you're not familiar with the Python C API, you might be confused as to why the function parameters look the way they do.

The basic calculations within `sum_of_squares` are identical to the regular C `sum_of_squares` function, but there are a few small differences. Firstly, the type definition for a function using the Python C API should look something like this:

```
static PyObject* sum_of_squares(PyObject *self, PyObject *args);
```

Let's break this down.

static

This means that the function is **static**. A function that's static can be called only from the same translation unit within the compiler. This effectively results in a function that cannot be linked (imported/used) from other modules, which allows the compiler to optimize a bit further. Since functions in C are global by default, this can be very useful in preventing naming collisions. Just to be sure, however, you could prefix your function names with the name of the module if you use a name that is less likely to be unique.

Be careful not to confuse the word `static` here with the `static` before a variable. They are completely different beasts. A `static` variable means that the variable will exist for the entire runtime of the program instead of the runtime of just the function.

PyObject*

The `PyObject` type is the basic type for Python data types, which means that all Python objects can be cast to `PyObject*` (the `PyObject` pointer). Effectively, it only tells the compiler what kind of properties to expect, which can be used later for type identification and memory management. Instead of direct access to `PyObject*`, it is generally a better idea to use the available macros, such as `Py_TYPE(some_object)`. Internally, this expands to `(((PyObject*)(o))->ob_type)`, which is why the macro is generally a better idea. Besides being unreadable, a typo can easily happen.

The list of properties is long and depends greatly on the type of object. For those, you can refer to the Python documentation: `https://docs.python.org/3/c-api/typeobj.html`.

The entire Python C API could fill a book of its own, but it is luckily well documented within the Python manual. Its usage, on the other hand, might be less obvious.

Parsing arguments

With regular C and Python, you specify the arguments explicitly, since variable-sized arguments are a bit tricky with C. This is because they need to be parsed separately. `PyObject* args` is the reference to objects containing the actual values. To parse these, you need to know how many and which type of variables to expect. In the example, we used the `PyArg_ParseTuple` function, which parses the arguments as positional arguments only, but it is quite easily possible to parse named arguments as well using `PyArg_ParseTupleAndKeywords` or `PyArg_VaParseTupleAndKeywords`. The difference between these is that the first one uses a variable number of arguments to specify the destination and the latter uses a `va_list` to set the values to.

Let's analyze the code from the actual example:

```
if(!PyArg_ParseTuple(args, "i", &n)){
    return NULL;
}
```

We know that `args` is the object containing the reference to the actual arguments. The `"i"` is a format string, which in this case will try to parse a single integer. `&n` tells the function to store the value at the memory address of the `n` variable.

The format string is the important part here. Depending on the character, you get a different data type, but there are many; `i` specifies a regular integer, and `s` converts your variable to a C-string (actually a `char*`, which is a null-terminated character array). It should be noted that this function is, luckily, smart enough to take overflows into consideration as well.

Parsing multiple arguments is quite similar; you need to add multiple characters to the format string and multiple destination variables:

```
PyObject* callback;
int n;

/* Parse the arguments */
if(!PyArg_ParseTuple(args, "Oi", &callback, &n)){
    return NULL;
}
```

The version with keyword arguments is similar, but requires a few more code changes as the list of methods needs to be informed that the function takes keyword arguments. Otherwise, the `kwargs` parameter would never arrive:

```
static PyObject* function(
        PyObject *self,
        PyObject *args,
        PyObject *kwargs){
    /* Declare the variables */
    PyObject* callback;
    int n;

    static char* keywords[] = {"callback", "n", NULL};

    /* Parse the arguments */
    if(!PyArg_ParseTupleAndKeywords(args, kwargs, "Oi", keywords,
                &callback, &n)){
        return NULL;
    }
```

```
    Py_RETURN_NONE;
}

static PyMethodDef methods[] = {
    /* Register the function with kwargs */
    {"function", function, METH_VARARGS | METH_KEYWORDS,
     "Some kwargs function"},
    /* Indicate the end of the list */
    {NULL, NULL, 0, NULL},
};
```

Let's look at the differences from the version that only supported `*args`:

1. Similar to pure Python, our function header now includes `PyObject *kwargs`.
2. Because we need to pre-allocate strings in C, we have an array of words called `keywords` with all of the `kwargs` we plan to parse.
3. Instead of `PyArg_ParseTuple` we now have to use `PyArg_ParseTupleAndKeywords`. This function overlaps the `PyArg_ParseTuple` function and adds keyword parsing by walking through the previously defined `keywords` array.
4. At the function registry, we need to specify that the function supports keyword arguments by adding the `METH_KEYWORDS` flag in addition to the `METH_VARARGS` flag.

Note that this still supports normal arguments, but keyword arguments are also supported now.

C is not Python — Errors are silent or lethal

As we saw in a previous example, integer overflows are not something you will generally notice, and unfortunately, there's no good cross-platform way to catch them. However, those are actually the easier errors to handle; the worst one is generally memory management. With Python, if you get an error, you will get an exception that you can catch. With C, you can't really handle it gracefully. Take a division by zero, for example:

```
$ python3 -c '1/0'
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ZeroDivisionError: division by zero
```

This is simple enough to catch with `try: ... except ZeroDivisionError: ...`. With C, on the other hand, if you get a bad error, it will kill your entire process. But debugging C code is what C compilers have debuggers for, and to find the cause of the error, you can use the `faulthandler` module discussed in *Chapter 11*, *Debugging – Solving the Bugs*. Right now, let's see how we can properly throw errors from C:

```
static PyObject* count_eggs(PyObject *self, PyObject *args){
    PyErr_SetString(PyExc_RuntimeError, "Too many eggs!");
```

```
    return NULL;
}
```

When executing this, it will effectively run `raise RuntimeError('Too many eggs!')`. The syntax is slightly different—`PyErr_SetString` instead of `raise`—but it's the same basic principle.

Calling Python from C — Handling complex types

We have seen how to call C functions from Python, but now let's try Python from C and back. Instead of using the readily available `sum` function, we will build one of our own with a callback and handling of any type of iterable. While this sounds simple enough, it does actually require a bit of type meddling, as you can only expect `PyObject*` as arguments. This is contrary to the simple types, such as integers, chars, and strings, which are immediately converted to the native Python version.

For clarity, this is just a single function that is broken up into multiple parts.

First, we start with the `include` function signature, and the declaration of the variables we need. Note that the values for `total` and `callback` are defaults in the event that these arguments are not specified:

```
#include <Python.h>

static PyObject* custom_sum(PyObject* self, PyObject* args){
    long long int total = 0;
    int overflow = 0;
    PyObject* iterator;
    PyObject* iterable;
    PyObject* callback = NULL;
    PyObject* value;
    PyObject* item;
```

Now we parse a `PyObject*` followed by, optionally (the | character), a `PyObject*` and a `long long int`. This is specified by the `O|OL` argument. The results will be stored in the memory addresses (the `&` sends the memory address of a variable) of `iterable`, `callback`, and `total`:

```
    if(!PyArg_ParseTuple(args, "O|OL", &iterable, &callback, &total)){
        return NULL;
    }
```

We see if we can create an iterator from the iterable. This is effectively the same as doing `iter(iterable)` in Python:

```
    iterator = PyObject_GetIter(iterable);
    if(iterator == NULL){
        PyErr_SetString(PyExc_TypeError,
                "Argument is not iterable");
        return NULL;
    }
```

Next, we check whether the callback exists or wasn't specified. If it was specified, check whether it's callable or not:

```
    if(callback != NULL && !PyCallable_Check(callback)){
        PyErr_SetString(PyExc_TypeError, "Callback is not callable");
        return NULL;
    }
```

Looping through the iterable, if we have a callback available, we call it. Otherwise, we just use the `item` as the `value`:

```
    while((item = PyIter_Next(iterator))){
        if(callback == NULL){
            value = item;
        }else{
            value = PyObject_CallFunction(callback, "O", item);
        }
```

We add the value to `total` and check for overflows:

```
        total += PyLong_AsLongLongAndOverflow(value, &overflow);
        if(overflow > 0){
            PyErr_SetString(PyExc_RuntimeError, "Integer overflow");
            return NULL;
        }else if(overflow < 0){
            PyErr_SetString(PyExc_RuntimeError, "Integer underflow");
            return NULL;
        }
```

If we were indeed using the callback, we decrease the reference count to the value because it is a separate object now.

We also need to dereference `item` and the iterator. Forgetting to do this results in memory leaks because it decreases the reference count for the Python garbage collector.

So, always make sure you call the `PyDECREF` function after using `PyObject*` types:

```
            if(callback != NULL){
                Py_DECREF(value);
            }
            Py_DECREF(item);
        }
        Py_DECREF(iterator);
```

Lastly, we need to convert `total` to the correct return type and return it:

```
        return PyLong_FromLongLong(total);
    }
```

This function is callable in three different ways. When given only an iterable, it will sum the iterable and return the value. Optionally, we can pass a callback function, which will be applied to each value in the iterable before summing. As a second optional parameter, we can specify the initial value to start with:

```
>>> x = range(10)
>>> custom_sum(x)
45
>>> custom_sum(x, lambda y: y + 5)
95
>>> custom_sum(x, lambda y: y + 5, 5)
100
```

Another important issue is that even though we catch overflow errors when converting to `long long int`, this code is still not safe. If we sum even two very large numbers (close to the `long long int` limit), we will still have an overflow:

```
>>> import spam

>>> n = (2 ** 63) - 1
>>> x = n,
>>> spam.sum(x)
9223372036854775807
>>> x = n, n
>>> spam.sum(x)
-2
```

In this case, you could test for this by doing something like `if(value > INT_MAX - total)`, but that solution does not always apply, so it is most important to be conscious of overflows and underflows when using C.

Exercises

The possibilities with external libraries are endless, so perhaps you already have some ideas about what to implement. If not, here's some inspiration:

- Try to sort a list of numbers using `ctypes`, `CFFI`, and with a native extension. You can use the `qsort` function in `stdlib`.
- Try to make the `custom_sum` function we created safer by adding proper errors for overflow/underflow issues. Additionally, catch the errors when summing multiple numbers that only overflow or underflow in summation.

These exercises should be a nice starting point for doing something useful with your newly acquired knowledge. If you are looking for more of the native C/C++ examples, I would recommend looking through the CPython source. There are many examples available: `https://github.com/python/cpython/tree/main/Modules`. I would suggest starting with a relatively simple one such as the `bisect` module.

Example answers for these exercises can be found on GitHub: `https://github.com/mastering-python/exercises`. You are encouraged to submit your own solutions and learn about alternative solutions from others.

Summary

In this chapter, you learned about writing and using extensions in C/C++. As a quick recap, we covered:

- Loading external (system) libraries such as `stdlib` using `ctypes`
- Creating and handling complex data structures using `ctypes` and `CFFI`
- Handling arrays using `ctypes` and `CFFI`
- Combining C and Python functions
- Important caveats regarding numeric types, arrays, overflows, and other error handling

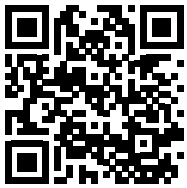
Even though you can now create C/C++ extensions, I still recommend that you avoid them, if possible, because it is so easy to end up with bugs. Even the code examples in this chapter don't handle many of the possible error scenarios and, as opposed to errors in Python, if these errors happen in C, they can kill your interpreter or application entirely.

If your goal is better performance, then I would recommend trying `numba` or `cython` instead. If you really need interoperability with non-Python libraries, however, these libraries are good options. A few examples of universal libraries such as these are TensorFlow and OpenCV, which are available in many languages and have Python wrappers for convenience.

While building the examples in this chapter, you may have noticed that we used a `setup.py` file and imported from the `setuptools` library. This is what the next chapter will cover: packaging your code into an installable Python library and distributing it on the Python Package Index.

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:
`https://discord.gg/QMzJenHuJf`



18

Packaging – Creating Your Own Libraries or Applications

The chapters thus far have covered how to write, test, and debug Python code. With all of that, there is only one thing that remains: packaging and distributing your Python libraries and applications. To create installable packages, we will use the `setuptools` package, which is bundled with Python these days. If you have created packages before, you might remember `distribute` and `distutils2`, but it is very important to remember that these have been replaced by `setuptools` and `distutils` and you shouldn't use them anymore!

We have several types of packages and packaging methods to cover:

- Building new-style packages using the PEP 517/518 `pyproject.toml` file
- Advanced package building using the `setup.py` file
- Package types: wheels, eggs, source packages, and others
- Installing executables and custom `setuptools` commands
- Packages containing C/C++ extensions
- Running tests on the package

Introduction to packages

Python has a very messy history when it comes to packaging. Over the decades that Python has existed, we have (had) libraries such as `distutils`, `distutils2`, `distribute`, `buildout`, `setuptools`, `packaging`, `distlib`, `poetry`, and several others. All of these projects were started with the best intentions to improve upon the status quo, unfortunately with varying degrees of success. And that is not to mention all the different package types such as wheels, source packages, and binary packages such as eggs, Red Hat `.rpm` files, and Windows `.exe`/`.msi` files.

The good news is that even though packaging has had a complicated history, things have started to settle over the last few years and the situation has improved greatly. Building packages has become much easier, and maintaining a stable project dependency state is now easily possible.

Types of packages

Python has (had) a whole bunch of package types, but there are only two that really matter these days:

- **Wheels**: These are small, ready-to-install `.zip` files with a `.whl` extension that only need extraction as opposed to the building a source package would need. Additionally, these can be either source or binary, depending on the package.
- **Source packages**: These can have many extensions such as `.zip`, `.tar`, `.tar.gz`, `.tar.bz2`, `.tar.xz`, and `.tar.Z`. They contain the Python/C/etc. source and data files needed to install and build the package.

Now we'll go into a bit more detail about the formats.

Wheels – The new eggs

For pure Python packages, the source packages have always been enough. For binary C/C++ packages, however, it is a much less convenient option. The problem with C/C++ source packages is that compilation is needed, which requires not only a compiler but often headers and libraries on the system as well. With binary packages, you usually don't need a compiler or any other libraries installed because the required libraries are bundled with the package; Python itself is enough.

Traditionally, Python used the `.egg` format for binary packages. The `.egg` format is, in essence, just a renamed `.zip` file containing the source code and metadata, and in the case of binary `.egg` files also the compiled binaries. While the idea was great, `.egg` files never really solved the problem quite right; an `.egg` file could match multiple environments, but that was no guarantee of it actually running on those systems.

That is why the `wheel` format was introduced (`PEP-427`), a package format that can contain both source and binary files and can be installed on Windows, Linux, macOS X, and other systems without requiring a compiler.

As an added bonus, wheels can install both pure Python and binary packages more quickly because there are no build, install, or post-processing steps and because they are smaller. Installing a wheel only requires extracting the `.whl` file to the `site-packages` directory of your Python environment and you are done.

The biggest issue binary wheels solve over eggs is the naming of files. With wheels, this is done in a simple and consistent way so that checking for the existence of a compatible wheel can be done by filename alone. The files use the following format:

```
{distribution}-{version}[-{build tag}]-{python tag}-{abi tag}-{platform tag}.
whl
```

Let's dive into these fields and look at what their possible values are. But first, the syntax. The names between the { and } parentheses are the fields, and the [and] parentheses indicate an optional field:

- `distribution`: The name of the package, e.g. `numpy`, `scipy`, etc.
- `version`: The version of the package, e.g. `1.2.3`.
- `build tag`: An optional build number as a tie-breaker if multiple wheels match.

- `python tag`: The Python version and platform. In the case of CPython 3.10, this would be `cp310`. For PyPy 3.10, this would be `pp310`. You can read more about this in `PEP-425`. For pure Python packages, this is either `py3` for Python 3 support or `py2.py3` for universal packages that support both Python 2 and Python 3.
- `abi tag`: The ABI (application binary interface) tag indicates the required Python ABI. `cp310d` would be CPython 3.10 with debugging enabled, for example. More details can be found in `PEP-3149`. In the case of a pure Python package, this is usually `none`.
- `platform tag`: The platform tag tells you the operating systems it will run on. This can be `win32` or `win_amd64` for 32-bit or 64-bit Windows respectively. For macOS X, this could be something like `macosx_11_0_arm64`. For pure Python packages, this is usually `any`.

With all of these different options, you can probably guess that to support many platforms you will need many wheels. This is actually a good thing, as it solves one of the big issues of egg files, namely that installable files did not always work. If you can find a matching wheel for your system, you can expect it to work without any issues.

The downside is the build time required for all of these wheels. The `numpy` package, for example, has 29 different wheels at the time of writing. Each of these wheels takes between 15 and 45 minutes to build, so if we take an average of 30 minutes per wheel, we end up with 15 hours of build time for each `numpy` release. Naturally, they can be built in parallel, but it is something to take into consideration.

Even though we have 29 different wheels available for `numpy`, there is still no support for many platforms, such as FreeBSD, so the need for a source package also remains.

Source packages

Source packages are the most versatile out of all the types of Python packages. They contain the source, build scripts, and potentially many other files such as documentation and tests. These allow you to build and/or compile the package for your system. Source packages can have many different extensions, such as `.zip` and `.tar.bz2`, but are basically a slightly stripped-down version of the entire project directory and related files.

Since these packages often contain not only the straight-up source files but also tests and documentation, they take up more space and are slower to install than wheels. Looking at the source package for `numpy`, for example, I currently see 1941 files, whereas the wheel only contains 710 files. This difference can actually be useful as well because you might have a use for the test files or the documentation. If you wish to skip the binary files because you wish to have the original sources or if you want an optimized build for your specific system, you can opt for installing the source files by telling `pip` to skip the binary files.

Installing the package from the source instead of the binaries can result in a smaller and/or faster binary because it will only link to the libraries available on your system instead of being universal.

The `psycopg` package for connecting to PostgreSQL databases is a good example of this. It offers three possible installation options to install through `pip` in descending order of preference:

- `psycopg[c]`: Both the Python and the C source files for building and compiling locally
- `psycopg[binary]`: The Python source and precompiled binaries
- `psycopg`: The Python source only; in this case, you need to have the `libpq` library installed on your system, which is accessed through `ctypes`

To install without any pre-compiled binaries:

```
$ pip3 install --no-binary ...
```

Since a source package comes with a build script, the installation alone can already be dangerous. While a wheel will only unpack and not run anything, a source package will execute the build scripts during installation. At one point, there was even a Russian Roulette package on PyPI that would have a 1/6 chance of deleting files on your system during installation to illustrate the dangers of this approach.

I personally think the security risk of executing build scripts during installion is of much less importance than vetting a package before you even plan to install it. Installing potentially malicious packages on your system is a bad idea, whether or not you actually execute the code.

Package tools

So, what tools do we still need and use for installation these days?

The `distribute`, `distutils`, and `distutils2` packages have largely been replaced by `setuptools`. To install a `setup.py` based source package, you usually need `setuptools`, and `setuptools` comes bundled with `pip`, so that is a requirement you should already have available. When it comes to installing wheels, you need the `wheel` package; this is also conveniently bundled with `pip`. On most systems, this means that you should have everything you need to install extra packages once Python is installed.

The Ubuntu Linux distribution is unfortunately a notable exception which ships with a mutilated Python installation that lacks both the `pip` and `ensurepip` commands. This can be fixed by installing `pip` separately using:

```
$ apt install python3-pip
```

If that does not work, you can always install `pip` by running the `get-pip.py` script: `https://bootstrap.pypa.io/get-pip.py`

Since `setuptools` and `pip` have seen quite a lot of development over the last few years, it might be a good idea to upgrade these packages in any case:

```
$ pip3 install --upgrade pip setuptools wheel
```

Now that we have all the prerequisites installed, we can continue with building our own packages.

Package versioning

While there are many versioning schemes available, many Python packages, and Python itself, use PEP-440 for the version specifications.

Some people adhere to the slightly stricter version called **semantic versioning** (**SemVer**), but the two are largely compatible.

The short and simplified explanation is that version numbers such as `1.2` or `1.2.3` are used. For instance, looking at version `1.2.3`:

- `1` is the major version and indicates API-breaking incompatible changes
- `2` is the minor version and indicates backward-compatible functionality addition
- `3` is the patch version, which is used for backward-compatible bugfixes

In the case of major versions, some libraries opt for making the versions non-contiguous and use dates for the versions, such as `2022.5`.

Pre-releases such as alphas and betas can be specified through the minor version with letters. The options are `a` for alpha, `b` for beta, and `rc` for release candidate. This results in `1.2a3` for `1.2 alpha 3`, for instance.

In the case of semantic versioning, this is handled by adding a pre-release identifier to the end, such as `1.2.3-beta` or `1.2.3-beta.1` for multiple betas.

Lastly, PEP-440 allows the use of post-releases using `1.2.post3` instead of `1.2.3` for minor bugfixes, and similarly `1.2.dev2` for development releases.

Whichever versioning system you use, think about it carefully before starting your project. Not taking the future into account can certainly cause problems in the long run. An example of this is Windows. Some applications had trouble supporting Windows 10 because an alphabetical sort of version number puts Windows 10 below Window 8 (after all, 1 is smaller than 8).

Building packages

Python packages were traditionally built using a `setup.py` file that contained (part of) the build script. This method usually depends on `setuptools` and is still the standard for most packages, but we have easier methods available these days. If your project is not too demanding, you can use a small `pyproject.toml` file instead, which can be much easier to maintain.

Let's give both methods a try and see how easy it is to build a basic Python package.

Packaging using pyproject.toml

The `pyproject.toml` file allows for really easy packaging depending on the tooling used. It was introduced in 2015 through `PEP-517` and `PEP-518`. This method was created to improve upon the `setup.py` file by introducing build-time dependencies, automatic configuration, and making it easier to work in a DRY (Don't Repeat Yourself) manner.

TOML stands for "Tom's Obvious, Minimal Language" and is somewhat comparable to YAML and INI files, but a bit simpler. Since it is such a simple language, it can easily be included in packages such as `pip` with little overhead. This makes it perfect for scenarios where you need a flat structure and have no need for complicated features such as inheritance and includes.

Before we continue, we need to clarify a few things. When we talk about the `setup.py` file, we are often actually talking about the `setuptools` library instead. The `distutils` library, which is bundled with Python, can be used as well, but since `pip` depends on `setuptools` it is often the better option; it has more features, and updates together with `pip` instead of with your Python installation.

Similar to how `setup.py` usually means `setuptools`, with `pyproject.toml` we also have multiple libraries available for building and managing `PEP-517` style packages. This approach to creating a standard and relying on community projects for the implementations has worked quite well for Python in the past, which makes it a sound choice. An example of this approach is the Python Web Server Gateway Interface (WSGI), which was introduced as `PEP-333` and currently has several great implementations available.

The reference solution for `PEP-517` is the `pep517` library, which works but is rather limited. Another option is the `build` library, which is maintained by the Python Package Authority (PyPA), which also maintains the Python Package Index (PyPI). While that library works, it is also really limited in terms of features and not an option I would recommend either.

The best option by far, in my opinion, is the `poetry` tool. The `poetry` tool not only handles the building of packages for you but also takes care of:

- Fast installing of dependencies in parallel
- Creating virtual environments
- Creating easy access points for runnable scripts
- Managing dependencies by specifying smart version constraints (for example, major and minor versions, covered in detail later in this chapter)
- Building packages
- Publishing to PyPI
- Handling multiple Python versions using `pyenv`

For most cases, `pyproject.toml` can replace the traditional `setup.py` files completely, but there are a few cases where you will need some extra tools.

In the case of building C/C++ extensions and others, you either need a `setup.py` file or to specify how to build the extensions some other way. One option for this is to use the `poetry` tool and add a build script to the `pyproject.toml` tool. We will discuss this more later on, in the section about C/C++ extensions.

Editable installs (i.e. `pip install -e ...`) were not possible until 2021, but that has been remedied by PEP-660.

Creating a basic package

Let's start by using `poetry` to create a basic `pyproject.toml` file in our current directory:

```
$ poetry new .
Created package t_00_basic_pyproject in .
```

Since our parent directory is called `t_00_basic_pyproject`, `poetry` automatically makes that the new project name. Alternatively, you can also do `poetry new some_project_name` and it will create a directory for you.

The `poetry` command created the following files for us:

```
README.rst
pyproject.toml
t_00_basic_pyproject
t_00_basic_pyproject/__init__.py
tests
tests/__init__.py
tests/test_t_00_basic_pyproject.py
```

This is very simple boilerplate that contains enough to get your project started. The `t_00_basic_pyproject/__init__.py` file contains the version (which defaults to `0.1.0`) and the `tests/test_t_00_basic_pyproject.py` file tests for this version as an example test. The more interesting part is the `pyproject.toml` file, however, so let's look at that now:

```
[tool.poetry]
name = "T_00_basic_pyproject"
version = "0.1.0"
description = ""
authors = ["Rick van Hattem <Wolph@wol.ph>"]

[tool.poetry.dependencies]
python = "^3.10"

[tool.poetry.dev-dependencies]
pytest = "^5.2"

[build-system]
requires = ["poetry-core>=1.0.0"]
build-backend = "poetry.core.masonry.api"
```

As you can see, `poetry` has automatically configured the name and version. It also added me as an author by looking at the `git` config on my system. You can easily configure this yourself by running these commands:

```
$ git config --global user.email 'your@email.tld'
$ git config --global user.name 'Your Name'
```

Next up, we can see that it automatically set Python 3.10 as a requirement and added `pytest` 5.2 as a development dependency. For building the package, it added `poetry-core` as a dependency, which is the `poetry` equivalent of `setuptools`.

Installing packages for development

For development purposes, we usually install a package in **editable mode**. In editable mode, the package is not copied to your `site-packages` directory, but a link is made to your source directory so all changes to your source directory apply immediately. Without editable mode, you would have to do a `pip install` every time you made a change to your package, which is very inconvenient for development.

With `pip`, you can install in editable mode with the command:

```
$ pip3 install --editable <package-directory>
```

For installing in the current directory, you can use `.` as the directory, resulting in:

```
$ pip3 install --editable .
```

With the `poetry` command, installing in editable mode (or something similar for older versions of `poetry`) happens automatically. It also handles the creation of a virtual environment for us while making use of `pyenv` for the Python version specified in the `pyproject.toml` file. To install the package and all of its requirements, you only need to run:

```
$ poetry install
```

If you wish to have direct access to all of the commands in the virtual environment created, you can use:

```
$ poetry shell
(name-of-your-project) $
```

The `poetry shell` command spawns a new shell with the name of the current project added to your command-line prefix and with the virtual environment scripts directory added to your `PATH` environment variable. This results in commands such as `python` and `pip` executing within your virtual environment.

Adding code and data

In the basic example, we didn't have anything specifying which directory contains the source or that the `t_00_basic_pyproject` directory has to be included in the directory. By default, that is handled implicitly, but we can modify the `pyproject.toml` file to explicitly include directories or file patterns as the Python source:

```
[tool.poetry]
...
packages = [
    {include="T_00_basic_pyproject"},
    {include="some_directory/**/*.py"},
]
```

Note that adding the `packages` argument disables the automatic detection of packages, so you will need to specify all included packages at this point.

To include other data files such as documentation, we can use the `include` and `exclude` parameters. The `exclude` parameter overrides the files included by the `packages` argument:

```
[tool.poetry]
...
include = ["CHANGELOG.rst"]
exclude = ["T_00_basic_pyproject/local.py"]
```

For a basic project, you might not need to look at this. But, as always, explicit is better than implicit, so I recommend that you do take a quick look to prevent unwanted surprises where the wrong files are accidentally included in your package.

Adding executable commands

Some packages such as `numpy` are libraries only, meaning they are imported but have no runnable commands. Other packages such as `pip` and `poetry` contain runnable scripts which are installed as new commands during installion. After all, when the `poetry` package is installed, you can use the `poetry` command from your shell.

To create our own commands, we need to specify the name of the new command, the module, and the corresponding function, so `poetry` will know what to run. For example:

```
[tool.poetry.scripts]
our_command = 'T_00_basic_pyproject.main:run'
```

This would execute the `run()` function in a file called `T_00_basic_pyproject/main.py`. After installing the package, you could execute `our_command` from your shell to run the script. During development with `poetry`, you can use `poetry run our_command`, which automatically runs the command in the `poetry`-created virtual environment.

Managing dependencies

The `pyproject.toml` file we created already had a few requirements for both development and building, but you might also want to add other dependencies to your project. For example, if we wanted to add a progress bar we could run the following:

```
$ poetry add progressbar2
Using version ^4.0.0 for progressbar2
...
```

This automatically installs the `progressbar2` package for us and adds it to the `pyproject.toml` file like this:

```
[tool.poetry.dependencies]
...
progressbar2 = "^4.0.0"
```

Additionally, `poetry` will create or update a `poetry.lock` file, which contains the exact package versions that are installed, so an installation in a new environment can easily be reproduced. In the case above, we simply told `poetry` to install any version of `progressbar2` which resulted in `poetry` setting the version requirement to `^4.0.0`, but we can relax those requirements so `poetry` will automatically install the latest patch, minor, or major version of the package.

By default, `poetry` will add the dependencies to the `[tool.poetry.dependencies]` section, but you can also add them as development dependencies using the `--dev` or `-D` command-line arguments. If you want to add other types of dependencies such as the `build-system` dependencies or test dependencies, you will need to manually edit the `pyproject.toml` file, however.

The version specifiers expect SemVer-compatible versions and work as follows. For allowing newer non-major versions, you can use the caret (^). This looks at the first non-zero number, so the behavior of `^1.2.3` is different from `^0.1.2`, as follows:

- `^1.2.3` means `>=1.2.3` and `<2.0.0`
- `^1.2` means `>=1.2.0` and `<2.0.0`
- `^1` means `>=1.0.0` and `<2.0.0`
- `^0.1.2` means `>=0.1.2` and `<0.2.0`

Next up are the tilde (~) requirements, which specify the minimal version but allow for minor updates. They are a bit simpler than the caret versions and effectively specify what the number should start with:

- `~1.2.3` means `>=1.2.3` and `<1.3.0`.
- `~1.2` means `>=1.2.0` and `<1.3.0`.
- `~1` means `>=1.0.0` and `<2.0.0`. Note that the two options above both allow minor version updates, and this is the only one that allows major version updates.

Wildcard requirements are also possible using an asterisk (*):

- `1.2.*` means `>=1.2.0` and `<1.3.0`
- `1.*` means `>=1.0.0` and `<2.0.0`

The versioning system is compatible with the format used by `requirements.txt` as well, which allows for versions such as:

- `>= 1.2.3`
- `>= 1.2.3, <1.4.0`
- `>= 1.2.3, <1.4.0, != 1.3.0`
- `!= 1.5.0`

I personally prefer this last syntax since it's clear and doesn't require much prior knowledge, but you are free to use whichever you prefer. By default, `poetry` will use a `^1.2.3` format when adding dependencies.

Now, let's say we have a requirement like `progressbar2 = "^3.5"` and we have version `3.5.0` in our `poetry.lock` file. If we run `poetry install`, it will install exactly version `3.5.0` because we know that version to be good.

As a developer, you might want to update that dependency to a newer version so you can test if newer versions also work. This is also something we can ask of `poetry`:

```
$ poetry update
Updating dependencies
...
Package operations: 0 installs, 1 update, 0 removals
  • Updating progressbar2 (3.5.0 -> 3.55.0)
```

Now `poetry` will automatically upgrade the package and update the `poetry.lock` file within the constraints of `pyproject.toml`.

Building the package

Now that we have our `pyproject.toml` file configured and the dependencies we want, we can build the package. This is trivially easy using `poetry`, luckily. Building the package takes a single command:

```
$ poetry build
Building T_00_basic_pyproject (0.1.0)
  - Building sdist
  - Built T_00_basic_pyproject-0.1.0.tar.gz
  - Building wheel
  - Built T_00_basic_pyproject-0.1.0-py3-none-any.whl
```

With just that single command, `poetry` created a source package and a wheel for us. So, if you have been keeping track, you'll realize we can essentially create and build a package with just two commands: `poetry new` and `poetry build`.

Building C/C++ extensions

Before we start with this section, I need to provide a little disclaimer. Building C/C++ extensions is at the time of writing (the end of 2021) not a stable and supported feature by `poetry`, which means it could be replaced by a different mechanism in the future. For the time being, however, there is a working solution available for building C/C++ extensions and future versions are likely to work in a similar fashion.

If you are looking for a stable and well-supported solution right now, I would suggest going with a `setup.py` based project instead as covered later in this chapter.

We need to start by modifying our `pyproject.toml` file and adding the following line to the `[tool.poetry]` section:

```
build = "build_extension.py"
```

Make sure you don't name the file `build.py` if you ever wish to use the PyPA build command.

Once this is done, `poetry` will execute the `build_extension.py` file when we run `poetry build`, so now we need to create the `build_extension.py` file so `setuptools` can build the extension for us:

```
import pathlib
import setuptools

# Get the current directory
PROJECT_PATH = pathlib.Path(__file__).parent

# Create the extension object with the references to the C source
sum_of_squares = setuptools.Extension('sum_of_squares', sources=[
    # Get the relative path to sum_of_squares.c
    str(PROJECT_PATH / 'sum_of_squares.c'),
])

def build(setup_kwargs):
    setup_kwargs['ext_modules'] = [sum_of_squares]
```

This script is largely the same as what you would put in the `setup.py` file. The reason for this is that it's actually injecting into the same function call. If you look carefully at the `build()` function, you will see that it updates `setup_kwargs` and sets the `ext_modules` item within that. That argument is fed to the `setuptools.setup()` function verbatim. Essentially, we are just emulating the use of a `setup.py` file.

Note that for our C file we used the `sum_of_squares.c` file from *Chapter 17, Extensions in C/C++, System Calls, and C/C++ Libaries*. You will see that the rest of the code largely resembles the `setup.py` file we used in *Chapter 17*.

When we execute the `poetry build` command, `poetry` will automatically call `setuptools` internally and build the binary wheel:

```
$ poetry build
Building T_01_pyproject_extensions (0.1.0)
  - Building sdist
  - Built T_01_pyproject_extensions-0.1.0.tar.gz
  - Building wheel
```

```
running build
running build_py
creating build
...
running build_ext
building 'sum_of_squares' extension
...
```

With that, we are done. We now have a wheel file containing the built C extension.

Packaging using setuptools with setup.py or setup.cfg

The `setup.py` file is the traditional method of building Python packages, but is still used quite extensively and is a very flexible method of creating packages.

Chapter 17 has already shown us a couple of examples when building extensions, but let's reiterate and review what the most important parts actually do. The core function you will be using in this entire chapter is `setuptools.setup()`.

The `distutils` package bundled with Python will be sufficient as well in most cases, but I recommend `setuptools` regardless. The `setuptools` package has many great features that `distutils` lacks and nearly all Python environments will have `setuptools` available as it is included with `pip`.

Before we continue, it is always a good idea to make sure you have the latest version of `pip`, `wheel`, and `setuptools`:

```
$ pip3 install -U pip wheel setuptools
```

The `setuptools` and `distutils` packages have changed significantly over the last few years and the documentation/examples written before 2014 are most likely out of date. Be careful not to implement deprecated examples, and I would recommend skipping any documentation/examples using `distutils`.

As an alternative or addition to the `setup.py` file, you can also configure all metadata using a `setup.cfg` file. This uses the INI format and can be a bit more convenient for simple metadata where you do not need (or want) the overhead of the Python syntax.

You can even choose to use `setup.cfg` alone and skip `setup.py`; however, if you did, you would need a separate building utility. For those cases, I would recommend installing PyPA's `build` library:

```
$ pip3 install build
...
```

Creating a basic package

Now that we have all the prerequisites, let's create a package using a `setup.py` file. While the most basic `setuptools.setup()` call technically doesn't require any parameters, you should really include at least the `name`, `version`, `packages`, `url`, `author`, and `author_email` fields if you plan to publish the package to PyPI. Here's a really basic example containing these fields:

```
import setuptools

if __name__ == '__main__':
    setuptools.setup(
        name='T_02_basic_setup_py',
        version='0.1.0',
        packages=setuptools.find_packages(),
        url='https://wol.ph/',
        author='Rick van Hattem',
        author_email='wolph@wol.ph',
    )
```

As an alternative to configuring these as `setup()` parameters, you can also use a `setup.cfg` file, which uses the INI format but works in effectively the same way:

```
[metadata]
name = T_03_basic_setup_cfg
version = 0.1.0
url='https://wol.ph/',
author='Rick van Hattem',
author_email='wolph@wol.ph',

[options]
packages = find:
```

The main advantage of `setup.cfg` is that it is a more concise and simpler file format than the `setup.py` file is. Take a look at the `packages` section, for example; `setuptools.find_packages()` is quite a bit more verbose than `find:`.

The downside is that you need to pair a `setup.cfg` file with either a `setup.py` or `pyproject.toml` file to be able to build it. `setup.cfg` alone is not enough for a package, which makes `setup.cfg` a nice and clean way to separate your metadata from your setup code. Additionally, many libraries such as `pytest` and `tox` have native support for the `setup.cfg` file, so these can be configured through the file as well.

To pair `setup.cfg` and/or `setup.py` with a `pyproject.toml` file, we need to add these lines to the `pyproject.toml` file:

```
[build-system]
requires = ["setuptools", "wheel"]
build-backend = "setuptools.build_meta"
```

Note that a `pyproject.toml` file by itself won't give you `poetry` support; for `poetry` support, you need to add a `[tool.poetry]` section.

Installing the package for development

To install the package for local development, we can once again use the `-e` or `--editable` flag, as explained in the `poetry` section of this chapter. This installs a link from your source directory to the `site-packages` directory so the actual source is used, instead of having `setuptools` copy all of the source files to the `site-packages` directory.

In short, from the project directory you can either use the `setup.py` file:

```
$ python3 setup.py develop
```

Or `pip`:

```
$ pip3 install -e .
```

Adding packages

In the basic example, you could see that we used `find_packages()` as the argument for packages. This automatically detects all source directories and is usually fine as a default, but sometimes you need more control. The `find_packages()` function also allows you to add an `include` or `exclude` parameter if you wish to exclude tests and other files from the package, like this:

```
setuptools.find_packages(
    include=['a', 'b', 'c.*'],
    exclude=['a.excluded'],
)
```

The arguments to `find_packages()` can also be translated to a `setup.cfg` file with a slightly different syntax:

```
[options]
packages = find:

[options.packages.find]
include =
    a
    b
    c.*
exclude = a.excluded
```

Adding package data

In most scenarios, you probably won't have to include the package data such as test data or documentation files, but there are cases where you need extra files. Web applications, for example, might come bundled with `html`, `javascript`, and `css` files.

There are a few different options for including extra files with your package. First, it is important to know which files are included in your source package by default:

- Python source files in the package directories and all their subdirectories
- The `setup.py`, `setup.cfg`, and `pyproject.toml` files
- Readme files if available, such as `README.rst`, `README.txt`, and `README.md`
- Metadata files containing the package name, version, entry points, file hashes, and so on

For Python wheels the list is even shorter, and only the Python source and the metadata files will be packaged by default.

This means that if we want to include other files, we need to specify that those need to be added. We have two different options for adding other types of data to our package.

First of all, we can enable the `include_package_data` flag as an argument to `setup()`:

```
    setuptools.setup(
        ...
        include_package_data=True,
    )
```

Once that flag is enabled, we can specify what file patterns we want in a `MANIFEST.in` file. This file contains patterns to include, exclude, and more. The `include` and `exclude` commands use patterns to match. These patterns are glob-style patterns (see the `glob` module for documentation: `https://docs.python.org/3/library/glob.html`) and have three variants for both the `include` and `exclude` commands:

- `include`/`exclude`: These commands only work for the given path and nothing else
- `recursive-include`/`recursive-exclude`: These commands are similar to the `include`/`exclude` commands, but process the given paths recursively
- `global-include`/`global-exclude`: Be very careful with these, as they will include or exclude these files anywhere within the source tree

Besides the `include`/`exclude` commands, there are also two others: the `graft` and `prune` commands, which include or exclude directories including all the files under a given directory. This can be useful for tests and documentation, since they can include non-standard files. Beyond those examples, it's almost always better to explicitly include the files you need and ignore all the others. Here's an example `MANIFEST.in` file:

```
# Include all documentation files
include-recursive *.rst
include LICENSE
```

```
# Include docs and tests
graft tests
graft docs

# Skip compiled python files
global-exclude *.py[co]

# Remove all build directories
prune docs/_build
prune build
prune dist
```

Alternatively, we can use the `package_data` and `exclude_package_data` arguments and add them to `setup.py`:

```
    setuptools.setup(
        ...
        package_data={
            # Include all documentation files
            '': ['*.rst'],

            # Include docs and tests
            'tests': ['*'],
            'docs': ['*'],
        },
        exclude_package_data={
            '': ['*.pyc', '*.pyo'],
            'dist': ['*'],
            'build': ['*'],
        },
    )
```

Naturally, these also have an equivalent `setup.cfg` format:

```
[options]
...
include_package_data=True,

[options.package_data]
# Include all documentation files
* = *.rst
```

```
# Include docs and tests
tests = *
docs = *

[options.exclude_package_data]
* = *.pyc, *.pyo
dist = *
build = *
```

Note that these parameters use `package_data` instead of `data` for a reason. All of these require you to use a package. That means that data will only be included if it's inside a proper Python package (in other words, if it contains an `__init__.py`).

You can choose whichever format and method you prefer.

Managing dependencies

When you are using a `setup.py` or `setup.cfg` file, you don't get the easy dependency management that `poetry` provides. Adding new dependencies is not much harder, except that you need to add the requirement and install the package yourself instead of doing it all in a single command.

As is the case with `pyproject.toml`, there are multiple types of dependencies that you can declare:

- `[build-system] requires`: These are the requirements to build the project. These are usually `setuptools` and `wheel` for `setuptools`-based packages; for `poetry` this would be `poetry-core`.
- `[options] install_requires`: These are the requirements to be able to run the package. A project such as `pandas` will have a requirement for `numpy`, for example.
- `[options.extras_require] NAME_OF_EXTRA`: If your project has optional dependencies for specific circumstances, the extras can help. For example, to install `portalocker` with `redis` support, you can run this command:

```
$ pip3 install "portalocker[redis]"
```

If you have experience with creating packages, you might wonder why `tests_require` is not shown here. The reason is that there is no real need for it anymore since `extras_require` was added. You can simply add an extra requirement for `tests` and `docs` instead.

Here's an example of adding a few requirements to a `setup.py` file:

```
setuptools.setup(
    ...
    setup_requires=['pytest-runner'],
    install_requires=['portalocker'],
    extras_require={
        'docs': ['sphinx'],
```

```
        'tests': ['pytest'],
    },
)
```

Here is the equivalent in a `setup.cfg` file:

```
[build-system]
requires =
    setuptools
    wheel

[options]
install_requires =
    portalocker

[options.extras_require]
docs = sphinx
tests = pytest
```

Adding executable commands

As is the case with a `pyproject.toml`-based project, we can specify executable commands using the `setup.py` or `setup.cfg` files as well. To add a basic executable command similar to how we can run the `pip` or `ipython` commands, we can add `entry_points` to our `setup.py` file:

```
setuptools.setup(
...
    entry_points={
        'console_scripts': [
            'our_command = T_02_basic_setup_py.main:run',
        ],
    },
```

Or the `setup.cfg` equivalent:

```
[options.entry_points]
console_scripts =
    our_command = T_03_basic_setup_cfg.main:run
```

Once you have installed this package you can run `our_command` from your shell, similar to how you would run a command like `pip` or `ipython`.

From the examples above, you might wonder if we have other options besides `console_scripts`, and the answer is yes. One example is `distutils.commands`, which can be used to add extra commands to `setup.py`. By adding a command in that namespace, you can do:

```
$ python3 setup.py our_command
```

The most prominent example of this behavior, however, is the `pytest` library. The `pytest` library uses these entry points to automatically detect plugins that are compatible with `pytest`. We could easily create our own equivalent:

```
[options.entry_points]
our.custom.plugins =
    some_plugin = T_03_basic_setup_cfg.some_plugin:run
```

Once you have packages like these installed, you can query them through `importlib` like so:

```
>>> from importlib import metadata

>>> metadata.entry_points()['our.custom.plugins']
[EntryPoint(name='some_plugin', value='...some_plugin:run', ...]
```

This is a very useful feature for automatically registering plugins across libraries.

Building the package

To actually build the package, we have a few options. I personally use the `setup.py` file if it is available:

```
$ python3 setup.py build sdist bdist_wheel
running build
...
creating 'dist/T_02_basic_setup-0.1.0-py3-none-any.whl' and adding ...
```

If you only have a `setup.cfg` and `pyproject.toml` available, you will need to install a package to invoke the builder. In addition to `poetry`, PyPA provides a tool called `build` for this, which creates an isolated environment for building the package:

```
$ python3 -m build
* Creating venv isolated environment...
...
Successfully built T_02_basic_setup-0.1.0.tar.gz and T_02_basic_setup-0.1.0-
py3-none-any.whl
```

Both the wheel and the source package are written to the `dist` directory and they are ready for publishing.

Publishing packages

Now that we have the packages built, we need to actually publish them to PyPI. There are several different options we can use, but let's discuss some optional package metadata first.

Adding URLs

Our `setup.py` and `setup.cfg` files already contained a `url` parameter that will be used as the package homepage on PyPI. However, we can add more relevant URLs by configuring the `project_urls` setting, which is an arbitrary map of name/URL pairs. For `settings.py`:

```
    setuptools.setup(
        ...
        project_urls=dict(
            docs='https://progressbar-2.readthedocs.io/',
        ),
    )
```

Or for `settings.cfg`:

```
[options]
project_urls=
    docs=https://progressbar-2.readthedocs.io/
```

Similarly, for `pyproject.toml` using `poetry`:

```
[tool.poetry.urls]
docs='https://progressbar-2.readthedocs.io/'
```

PyPI trove classifiers

To increase the exposure of your package on PyPI, it can be useful to add a few classifiers. Some classifiers such as the Python version and the license are automatically added for you, but it can be useful to specify what kind of library or application you are writing.

There are many examples of useful classifiers for people interested in your packages:

- **Development status**: This can vary from "planning" to "mature" and tells your users whether an application is ready for production. People's definitions of what is stable or beta differs, of course, so this is usually considered a hint at most.
- **Framework**: The framework(s) you are using or extending. This could be Jupyter, IPython, Django, Flask, and so on.
- **Topic**: Whether this is a software development package, scientific, a game, and so on.

A full list of classifiers can be found on the PyPI website: `https://pypi.org/classifiers/`

Uploading to PyPI

Uploading and publishing your package to PyPI is really easy. Perhaps too easy, as we will see in the case of `twine`.

Before we get started, to prevent you from accidentally publishing your package to PyPI, you should be aware of the PyPI test server: `https://packaging.python.org/en/latest/guides/using-testpypi/`

In the case of `poetry`, we can configure the test repository like this:

```
$ poetry config repositories.testpypi https://test.pypi.org/simple/
$ poetry config pypi-token.testpypi <token>
```

First of all, if you are using `poetry` it is as simple as:

```
$ poetry publish --repository=testpypi
```

If you're not using `poetry` and don't want to use a `poetry`-compatible `pyproject.toml`, you'll need a different solution. The official solution from PyPA is to use the `twine` tool, maintained by the PyPA. After you have used `python3 -m build` to build the package, you can use `twine` for uploading:

Warning! This command will immediately register and upload the package to `pypi.org` if you are already authenticated. That's why `--repository testpypi` was added to upload to the test PyPI server instead. If you drop that argument, you will immediately publish your package to PyPI.

```
$ twine upload --repository testpypi dist/*
```

Before you start publishing your packages to PyPI, you should ask yourself a couple of questions:

- Is the package in a working state?
- Do you plan to support the package?

The PyPI repository is unfortunately full of empty packages from people that are claiming usable package names for no apparent reason.

C/C++ extensions

The previous chapter and earlier sections in this chapter have already covered the compilation of C/C++ components lightly, but this topic is complicated enough to warrant its own section with more in-depth explanations.

For convenience, we will start with a basic `setup.py` file that compiles a C extension:

```
import setuptools

sum_of_squares = setuptools.Extension('sum_of_squares', sources=[
    # Get the relative path to sum_of_squares.c
    str(PROJECT_PATH / 'sum_of_squares.c'),
])

setuptools.setup(
    name='T_04_C_extensions',
    version='0.1.0',
    ext_modules=[sum_of_squares],
)
```

Before you start with these extensions, you should learn the following `setup.py` commands:

- `build_ext`: This command builds the C/C++ extension so it can be used when the package is installed in development/editable mode.
- `clean`: This cleans the results from the `build` command. This is generally not needed, but sometimes the detection of files that need to be recompiled to work is incorrect. If you encounter strange or unexpected issues, try cleaning the project first.

Instead of using `python3 setup.py build_ext`, you can also choose to use the PyPA `build` command, but that is not a convenient option for development. If you use `python3 setup.py build` you can reuse your `build` directory and selectively build your C/C++ extensions, which saves you a lot of time for larger C/C++ modules. The PyPA `build` command is meant to produce clean, production-ready packages, which is strongly recommended for deploying and publishing, but not for development.

Regular C/C++ extensions

The `setuptools.Extension` class tells `setuptools` that a module named `sum_of_squares` uses the source file `sum_of_squares.c`. This is just the simplest version of an extension – a name and a list of sources – but often you are going to need not just the C file but also some headers from other libraries.

A prime example is the `pillow` library for image manipulation. When the library is building, it automatically detects the libraries available on the system and adds extensions based on that. For `.jpeg` support you need to have `libjpeg` installed; for `.tiff` images you need `libtiff`; and so on. As these extensions include binary libraries, some extra compilation flags and C header files are required. The basic PIL module itself doesn't appear too involved, but the `setup.py` file is filled with auto-detection code to detect which `libs` (libraries) are available, with the matching C macro definitions to enable these libraries.

Macros in C are preprocessor directives. These directives are executed before the actual compilation step occurs, which makes them ideal for conditional code. You could have a conditional block of debug code dependent on a `DEBUG` flag, for example:

```
#ifdef DEBUG
/* your debug code here */
#endif
```

If `DEBUG` is set, the code will be part of the compiled binary. If the flag is not set, the block of code will never end up in the resulting binary. This results in smaller and faster binaries because these conditionals happen at compile time as opposed to runtime.

Here's a partial example `Extension` from an older version of the `pillow` `setup.py` file:

```
exts = [(Extension("PIL._imaging", files, libraries=libs,
    define_macros=defs))]
```

The newer versions are quite different and the `setup.py` file for the `pillow` project is currently over 1,000 lines. The `freetype` extension has something similar:

```
if feature.freetype:
    exts.append(Extension(
        "PIL._imagingft", ["_imagingft.c"], libraries=["freetype"]))
```

Adding and compiling C/C++ extensions can certainly be challenging, so I would recommend taking inspiration from projects such as `pillow` and `numpy` if you need to take care of this. They are perhaps a bit too complicated, but should provide you with a nice starting point that covers nearly all scenarios.

Cython extensions

The `setuptools` library is a bit smarter than the regular `distutils` library when it comes to extensions: it actually adds a little trick to the `Extension` class. Remember the brief introduction to `cython` in *Chapter 12*, about performance? The `setuptools` library makes it a bit more convenient to compile Cython extensions. The Cython manual recommends that you use something similar to the following code:

```
from setuptools import setup
from Cython.Build import cythonize

setup(
    ext_modules = cythonize("src/*.pyx")
)
```

The problem with this approach is that `setup.py` will break with an `ImportError` unless you have Cython installed:

```
$ python3 setup.py build
Traceback (most recent call last):
  File "setup.py", line 2, in <module>
    import Cython
ImportError: No module named 'Cython'
```

To prevent that issue, we are just going to let `setuptools` handle the Cython compilation:

```
import setuptools

setuptools.setup(
    name='T_05_cython',
    version='0.1.0',
    ext_modules=[
        setuptools.Extension(
            'sum_of_squares',
            sources=['T_05_cython/sum_of_squares.pyx'],
        ),
```

```
    ],
    setup_requires=['cython'],
)
```

Now Cython will automatically be installed if needed and the code will work just fine:

```
$ python3 setup.py build
running build
running build_ext
cythoning T_05_cython/sum_of_squares.pyx to T_05_cython/sum_of_squares.c
building 'sum_of_squares' extension
...
```

For development purposes, however, Cython also offers a simpler method that doesn't require manual building, `pyximport`:

```
$ python3
>>> import pyximport

>>> pyximport.install()
(None, <pyximport.pyximport.PyxImporter object at ...>)

>>> from T_05_cython import sum_of_squares

>>> sum_of_squares.sum_of_squares(10)
14
```

That's how easy it is to run the `pyx` files without explicit compiling.

Testing

In *Chapter 10, Testing and Logging – Preparing for Bugs*, we saw a few of the many testing systems for Python. As you might suspect, at least some of these have `setup.py` integration. It should be noted that `setuptools` even has a dedicated `test` command (at the time of writing), but this command has been deprecated and the `setuptools` documentation now recommends using `tox`. While I am a huge fan of `tox`, for immediate local development it often incurs quite a bit of overhead. I find that executing `py.test` directly is faster, because you can really quickly test only the bits of the code that you changed.

unittest

Before we start, we should create a test script for our package. For actual tests please look at *Chapter 10*; in this case, we will just use a no-op test, `test.py`:

```
import unittest

class Test(unittest.TestCase):
```

```
    def test(self):
        pass
```

The standard `python setup.py test` command has been deprecated, so we will run `unittest` directly:

```
$ python3 -m unittest -v test
running test
...
```

The `unittest` library is still rather limited, however, so I recommend skipping straight to `py.test` instead.

py.test

The `py.test` package currently automatically registers as an extra command in `setuptools`, so after installing you can run `python3 setup.py pytest`. However, since `setuptools` is actively trying to reduce all interaction with `setup.py`, I would recommend using a `py.test` or `tox` call directly instead.

As mentioned earlier, it is recommended to use `tox` for bootstrapping your environment and fully testing the project. For fast local development, however, I would suggest installing the `pytest` module and running the tests directly.

Note that there might still be old documentation floating around suggesting the use of `pytest-runner`, `python setup.py test` with an alias or custom command, or the generation of a `runtests.py` file, but all of these solutions have been deprecated and should not be used anymore.

To configure `py.test` we have several options depending on your preferences. All of the following files will work:

- `pytest.ini`
- `pyproject.toml`
- `tox.ini`
- `setup.cfg`

For the projects I maintain, I have the test requirements defined as an extra so these can be installed using (for example) `pip3 install -e "./progressbar2[tests]"`. After that, you can easily run `py.test` to run the tests identically to how `tox` would run them. Naturally, `tox` can also install the requirements using the same extras, which ensures you are using the same test environment.

To enable this in your `setup.cfg` (or the equivalent for `setup.py` / `pyproject.toml`):

```
[options.extras_require]
tests = pytest
```

For local development, we can now install the package and the extras in editable mode for quick testing:

```
$ pip3 install -e '.[tests]'
```

That should be enough to be able to test using `py.test` directly:

```
$ py.test
```

To test using `tox`, you will need to create a `tox.ini` file, but for that, I suggest you take a look at *Chapter 10*.

Exercises

Now that you have reached the end of the book, there are many things to try, of course. You can build and publish your own applications and libraries, or extend existing libraries and applications.

While trying out the examples in this chapter, be careful not to accidentally publish packages to PyPI if that was not your intention. It just takes a single `twine` command to accidentally register and upload a package, and PyPI is already too crowded with packages that do nothing useful.

For some practical exercises:

- Create a `setuptools` command to bump the version in your package
- Extend the version bumping command by interactively asking for a major, minor, or patch upgrade
- Try and convert existing projects from `setup.py` to a `pyproject.toml` structure

Example answers for these exercises can be found on GitHub: `https://github.com/mastering-python/exercises`. You are encouraged to submit your own solutions and learn about alternative solutions from others.

Summary

After reading this chapter, you should be able to create Python packages containing not only pure-Python files but also extra data, compiled C/C++ extensions, documentation, and tests. With all these tools at your disposal, you are now able to make high-quality Python packages that can easily be reused in other projects and packages.

The Python infrastructure makes it really quite easy to create new packages and split your project into multiple subprojects. This allows you to create simple and reusable packages with fewer bugs because everything is easily testable. While you shouldn't go overboard with splitting up the packages, if a script or module has a purpose of its own then it's a candidate for packaging separately.

*

With this chapter, we have come to the end of the book. I sincerely hope you enjoyed reading it and have learned about some new and interesting topics. Any and all feedback is greatly appreciated, so feel free to contact me through my website at `https://wol.ph/`.

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:
`https://discord.gg/QMzJenHuJf`

packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

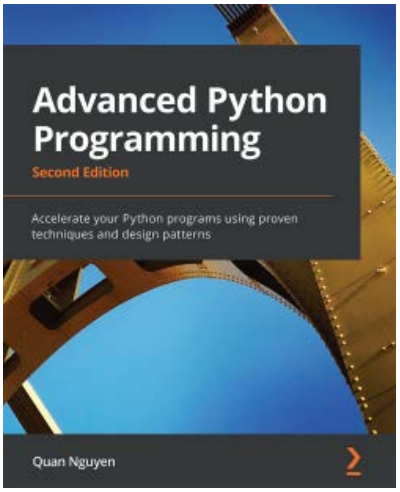
Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

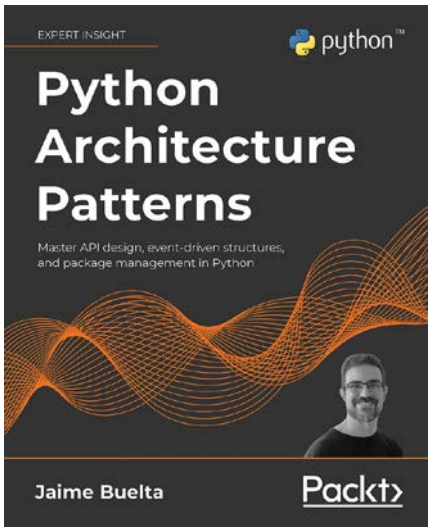


Advanced Python Programming - Second edition

Quan Nguyen

ISBN: 978-1-80181-401-0

- Write efficient numerical code with NumPy, pandas, and Xarray
- Use Cython and Numba to achieve native performance
- Find bottlenecks in your Python code using profilers
- Optimize your machine learning models with JAX
- Implement multithreaded, multiprocessing, and asynchronous programs
- Solve common problems in concurrent programming, such as deadlocks
- Tackle architecture challenges with design patterns



Python Architecture Patterns

Jaime Buelta

ISBN: 978-1-80181-999-2

- Think like an architect, analyzing software architecture patterns
- Explore API design, data storage, and data representation methods
- Investigate the nuances of common architectural structures
- Utilize and interoperate elements of patterns such as microservices
- Implement test-driven development to perform quality code testing
- Recognize chunks of code that can be restructured as packages
- Maintain backward compatibility and deploy iterative changes

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share your thoughts

Now you've finished *Mastering Python, Second Edition*, we'd love to hear your thoughts! If you purchased the book from Amazon, please `click here to go straight to the Amazon review page` for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y