Practical Python Design Patterns

Pythonic Solutions to Common Problems

Wessel Badenhorst

Apress®

Practical Python Design Patterns

Pythonic Solutions to Common Problems

Wessel Badenhorst

Apress®

Practical Python Design Patterns: Pythonic Solutions to Common Problems

Wessel Badenhorst
Durbanville, Eastern Cape, South Africa

ISBN-13 (pbk): 978-1-4842-2679-7 ISBN-13 (electronic): 978-1-4842-2680-3
https://doi.org/10.1007/978-1-4842-2680-3

Library of Congress Control Number: 2017957538

Cover image by Shutterstock (www.shutterstock.com)

Managing Director: Welmoed Spahr
Editorial Director: Todd Green
Acquisitions Editor: Steve Anglin
Development Editor: Matthew Moodie
Technical Reviewer: Michael Thomas
Coordinating Editor: Mark Powers
Copy Editor: April Rondeau

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com or visit http://www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484226797. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

For Tanya, my love, always.

Table of Contents

About the Author

Wessel Badenhorst is deeply passionate about the process of attaining mastery, especially in the field of programming. The combination of a bachelor's degree in computer science and extensive experience in the real world gives him a balanced and practical perspective on programming and the road every programmer must walk.

About the Technical Reviewer

Michael Thomas has worked in software development for more than 20 years as an individual contributor, team lead, program manager, and vice president of engineering. Michael has more than ten years of experience working with mobile devices. His current focus is in the medical sector, which is using mobile devices to accelerate information transfer between patients and health care providers.

Acknowledgments

A very big thank you to Mark Powers and his team at Apress, without whom this book would not have seen the light of day. To Tanya, Lente, and Philip, for letting me do this when I could have spent time on other things, thank you. Thank you, Lord, for the grace to complete this project. To you, the reader, thank you for taking steps to make yourself a better programmer, thereby making the profession better.

CHAPTER 1

Before We Begin

Design patterns help you learn from others' successes instead of your own failures.

—Mark Johnson

The world is changing.

As you are reading this, people all over the world are learning to program, mostly poorly. There is a flood of "programmers" heading for the market, and there is nothing you can do to stop them. As with all things, you will soon see that as the supply of programmers increases, the cost of hiring programmers will decrease. Simply being able to write a few lines of code or change something in a piece of software will (like the role of spreadsheet or word-processing guru) become a basic skill.

If you plan on making a living as a programmer, this problem is made worse by the Internet. You have to worry not only about programmers in your own area, but also those from all over the world.

Do you know who does *not* worry about the millions of programmers produced by the hundreds of code boot camps that seem to spring up like mushrooms?

The Masters

While everyone can write, there are still the Hemingways of the world. While everyone can work Excel, there are still the financial modelers. And when everyone can code, there will still be the Peter Norvigs of the world.

Programming is a tool. For a time, simply knowing how to use this tool made you valuable. That time has come to an end. There is a new reality, the same one that a lot of industries have had to face over the years. When a tool is new, no one wants to use it. Then, some see the value, and it makes them twice as good as the ones who can't use the tool. Then, it becomes popular. Before you know it, everyone can use the Internet.

W. Badenhorst, *Practical Python Design Patterns*, https://doi.org/10.1007/978-1-4842-2680-3_1

Suddenly, being able create a website becomes a lot less valuable. All the businesses that were consulting, charging big bucks for the service, get marginalized. However, the people who spent time mastering the tools are able to thrive no matter what happens in the market. They can do this because they are able to outproduce the average person on every level—quality of work, speed of development, and the beauty of the final result.

Right now, we see the popular uptake of that which is easy. The next step will be automating what is easy. Easy tasks, in every area of life, are easy because they do not require creativity and deep understanding, and as a result of these characteristics they are the exact tasks that will be handed over to computers first.

The reason you picked up this book is because you want to become a better programmer. You want to progress beyond the thousands of introductions to this and that. You are ready to take the next step, to master the craft.

In the new world of programming, you want to be able to solve big, complex problems. To do that, you need to be able to think on a higher level. Just like the chess grand-masters, who can digest the game in larger chunks than mere masters, you too need to develop the skill to solve problems in larger chunks. When you are able to look at a brand-new problem and rapidly deconstruct it into high-level parts, parts you have solved before, you become an exponential programmer.

When I began programming, I could not read the manual yet. There was a picture of *Space Invaders* on the cover, and it promised to teach you how to program a game yourself. It was a bit of a racket because the game ended up being a `for` loop and a single `if` statement, not *Space Invaders*. But I was in love. Passion drove me to learn everything I could about programming, which was not a lot. After mastering the basics, I stagnated, even while attaining a BS in computer science (cum laud). I felt like what I was learning was a simple rehashing of the basics. I did learn, but it felt slow and frustrating, like everybody was waiting for something.

In the "real world," it did not look a lot different. I came to realize that if I wanted to become an exponential programmer, I had to do something different. At first, I thought about getting a master's degree or Ph.D., but in the end I decided to dig deeper on my own.

I reread the old theory of computing books, and they gained new meaning and new life. I started regularly taking part in coding challenges and studied more idiomatic ways of writing code. Then, I began exploring programming languages and paradigms foreign to my own. Before I knew it, I was completely transformed in my thinking about programming. Problems that used to be hard became trivial, and some that seemed impossible became merely challenging.

I am still learning, still digging.

You can do it too. Start with this very book you are reading right now. In it you will find a number of tools and mental models to help you think and code better. After you have mastered this set of patterns, actively seek out and master different tools and techniques. Play with different frameworks and languages and figure out what makes them great. Once you can understand what makes someone love a different language than the one you use regularly, you may find a couple of ideas in it that you can integrate into your way of thinking. Study recipes and patterns wherever you find them and translate them into an abstract solution that you can use over and over. Sometimes, you can simply analyze a solution you already use: can you find a more elegant way to implement the idea, or is the idea flawed in some way? Constantly ask yourself how you can improve your tools, your skills, or yourself.

If you are ready to dig deep and master the art, you will find ideas in this book that will guide you along the road to mastery. We are going to explore a set of fundamental design patterns in a real-world context. While you are doing this, you will begin to realize how these design patterns can be seen as building blocks you can use when you encounter a specific type of problem in the future.

My hope is that you will use the design patterns in this book as a blueprint; that it will help you kick-start your own collection of problem–solution chunks. After you have read this book, you should be well on your way to reaching the next level in your journey as a programmer. You should also see how this set of abstract tools can be translated into any programming language you may encounter, and how you can use ideas and solutions from other languages and incorporate them into your mental toolbox.

Becoming a Better Programmer

To become a better programmer, you need to decide to obsess about mastery. Every day is a new opportunity to become better than the day before. Every line of code is an opportunity to improve. How do you do that?

- When you must work on a piece of existing code, leave the code better than you found it.
- Try to complete a quick problem-solving challenge every day.
- Look for opportunities to work with people who are better programmers than you are (open source projects are great for this).

- Focus on deliberate practice.
- Practice systems thinking whenever you can find an excuse.
- Collect and analyze mental models.
- Master your tools and understand what they are good for and what they should not be used for.

Deliberate Practice

Dr. Anders Erickson studied people who reached a level of performance we call mastery. He found that these people had a couple of things in common. The first, as popularized in Malcolm Gladwell's book *Outliers*, was that these people all seemed to have spent a significant amount of time working on the skill in question. The actual numbers vary, but the time invested approached about 10,000 hours. That is a significant amount of time, but simply spending 10,000 hours, or ten years, practicing is not enough. There is a very specific type of practice that was shown to be required for master-level performance. The key to world-dominating skill is *deliberate* practice.

We will look at how this relates to programming in a moment, but let's first look at what deliberate practice is.

Deliberate practice is slow, halting, anti-flow. If you are practicing the violin, deliberate practice would entail playing the piece extremely slowly, making sure you hit every note perfectly. If you are deliberately practicing tennis, this might mean doing the same shot over and over and over again with a coach, making small adjustments until you are able to perfectly hit that shot in that location time and time again.

The problem with knowledge work is that the standard deliberate practice protocols seem foreign. A sports task or playing an instrument is fairly simple when compared to the process involved in programming. It is a lot harder to deconstruct the process of designing and developing a software solution. It is hard to figure out how to practice a way of thinking. One of the reasons for this is that a solved problem tends to stay solved. What I mean by that is that as a developer you will very rarely be asked to write a piece of software that has already been written. Finding the "shot" to practice over and over is hard.

In programming, you want to learn different ways to solve problems. You want to find different challenges that force you to attack the same type of problem with different constraints. Keep working at it until you understand the problem and the set of possible solutions inside out.

In essence, deliberate practice has these components:

- Each practice session has a single focus.
- The distance (time) between attempt and feedback is as short as possible.
- Work on something that you cannot yet do.
- Follow in the path of those who have gone before you.

Single Focus

The reason you only want to focus on one thing during a specific training session is that you want devote all your attention to the element that you wish to improve. You want to do whatever it takes to do it perfectly, even if it is only once. Then, you want to repeat the process to get another perfect response, and then another. Every practice run is a single reinforcement of the skill. You want to reinforce the patterns that lead to the perfect result rather than those that have less desirable outcomes.

In the context of this book, I want you to target a single design pattern at a time. Really seek to understand not only how the pattern works, but also why it is used in the first place and how it applies to the problem at hand. Also, think about other problems that you might be able to solve using this design pattern. Next, try to solve some of these problems using the pattern you are working on.

You want to create a mental box for the problem and the pattern that solves it. Ideally, you will develop a sense for when problems would fit into one of the boxes you have mastered, and then be able to quickly and easily solve the problem.

Rapid Feedback

One of the pieces of deliberate practice that is often overlooked is the rapid-feedback loop. The quicker the feedback, the stronger the connection, and the more easily you learn from it. That is why things like marketing and writing are so hard to master. The time between putting letters on a page and getting feedback from the market is simply too long to really be able to see the effects of experiments. With programming this is not the case; you can write a piece of code and then run it for instant feedback. This allows you to course correct and ultimately reach an acceptable solution. If you go one step further and write solid tests for your code, you get even more feedback, and you are able to arrive at the solution much more quickly than if you had to test the process manually each time you made a change.

Another trick to help you learn from the process more rapidly is to make a prediction as to what the outcome of the block of code you want to write will be. Make a note as to why you expect this specific outcome. Now write the code and check the result against the expected result. If it does not match up, try to explain why this is the case and how you would verify your explanation with another block of code. Then, test that. Keep doing it until you have attained mastery.

You will find yourself getting feedback on different levels, and each has its own merit.

Level one is simply whether the solution works or not. Next, you might begin considering questions such as "How easy was the solution to implement?" or "Is this solution a good fit for the problem?" Later, you might seek out external feedback, which can take the form of simple code review, working on projects, or discussing solutions with like-minded people.

Stretch Yourself

What are the things that you shy away from? These are the areas of programming that cause you slight discomfort. It might be reading from a file on disk or connecting to a remote API. It makes no difference if it is some graphical library or a machine-learning setup, we all have some part of the craft that just does not sit comfortably. These are usually the things that you most need to work on. Here are the areas that will stretch you and force you to face your own weaknesses and blind spots. The only way to get over this discomfort is to dive deep into it, to use that tool so many times and in so many ways that you begin to get a feel for it. You must get so comfortable with it that you no longer have to look up that file open protocol on stack overflow; in fact, you have written a better one. You jump rope with the GUI and suck data out of a database with one hand tied behind your back.

There is no shortcut to this level of mastery; the only way is through the mountain. That is why so few people become true masters. Getting there means spending a lot of time on the things that are not easy, that do not make you feel like you are invincible. You spend so much time in these ego-destroying zones that very few masters of any craft have a lot of arrogance left in them.

So, what should you work on first? That thing you thought of as you read the previous two paragraphs.

Working your way through the design patterns in this book is another good way of finding potential growth areas. Just start with the singleton pattern and work your way through.

Stand on the Shoulders of Giants

There are people who do amazing things in the programming space. These people often give talks at developer conferences and sometimes have a presence online. Look at what these people are talking about. Try to understand how they approach a novel problem. Type along with them as they demonstrate solutions. Get into their head and figure out what makes them tick. Try to solve a problem like you imagine one of them would do it; how does the solution you come up with in this way differ from the one you came up with on your own?

Really great developers are passionate about programming, and it should not take a lot of prodding to get them to talk about the finer details of the craft. Seek out the user groups where this type of developer hangs out and strike up a lot of conversations. Be open and keep learning.

Pick personal projects that force you to use design patterns you have yet to master. Have fun with it. Most of all, learn to love the process, and don't get caught up on some perceived outcome rather than spending time becoming a better programmer.

How Do You Do This?

Begin the same way Leonardo da Vinci began when he decided to make painting his vocation.

Copy!

That is right. Begin by identifying some interesting problem, one that is already solved, and then blatantly copy the solution. Don't copy/paste. Copy the solution by typing it out yourself. Get your copy to work. Once you do, delete it all. Now try to solve the problem from memory, only referring to the original solution when your memory fails you. Do this until you are able to reproduce the solution flawlessly without looking at the original solution. If you are looking for solutions to problems that you can copy and learn from, Github is a gold mine.

Once you are comfortable with the solution as you found it, try to improve on the original solution. You need to learn to think about the solutions you find out there—what makes them good? How can you make them more elegant, more idiomatic, more simple? Always look for opportunities to make code better in one of these ways.

Next, you want to take your new solution out into the wild. Find places to use the solution. Practicing in the real world forces you to deal with different constraints, the kinds you would never dream up for yourself. It will force you to bend your nice, clean solution in ways it was never intended to be. Sometimes, your code will break, and you will learn to appreciate the way the problem was solved originally. Other times, you may find that your solution works better than the original. Ask yourself why one solution outperforms another, and what that teaches you about the problem and the solution.

Try using languages with different paradigms to solve similar problems, taking from each and shaping a solution yourself. If you pay attention to the process as much as you do to actually solving the problem, no project will leave you untouched.

The beauty of working on open source projects is that there is usually just the right mix of people who will help you along, and others who will tell you exactly what is wrong with your code. Evaluate their feedback, learn what you can, and discard the rest.

The Ability to Course Correct

When exploring a problem, you have two options: go on trying this way, or scrap everything and start over with what I have learned. Daniel Kahnamann, in his book *Thinking, Fast and Slow*, explains the sunk cost fallacy. It is where you keep investing in a poor investment because you have already invested so much in it. This is a deadly trap for a developer. The quickest way to make a two-day project take several months is to try to power your way through a poor solution. Often it feels like it would be a massive loss if we were to scrap a day, a week, or a month's work and start from nothing.

The reality is that we are never starting from scratch, and sometimes the last 10,000 lines of code you are deleting are the 10,000 lines of code you needed to write to become the programmer you need to be to solve the problem in 100 lines of code with a startlingly elegant solution.

You need to develop the mental fortitude to say enough is enough and start over, building a new solution using what you have learned.

No amount of time spent on a solution means anything if it is the wrong solution. The sooner you realize that the better.

This self-awareness gives you the ability to know when to try one more thing and when to head in a different direction.

Systems Thinking

Everything is a system. By understanding what elements make up the system, how they are connected, and how they interact with one another, you can understand the system. By learning to deconstruct and understand systems, you inevitably teach yourself how to design and build systems. Every piece of software out there is an expression of these three core components: the elements that make up the solution, and a set of connections and interactions between them.

At a very basic level, you can begin by asking yourself: "What are the elements of the system I want to build?" Write these down. Then, write down how they are connected to each other. Lastly, list all the interactions between these elements and what connections come into play. Do this, and you will have designed the system.

All design patterns deal with these three fundamental questions: 1) What are the elements and how are they created? 2) What are the connections between elements, or what does the structure look like? 3) How do the elements interact, or what does their behavior look like?

There are other ways to classify design patterns, but for now use the classical three groupings to help you develop your systems thinking skills.

Mental Models

Mental models are internal representations of the external world. The more accurate your models of the world are, the more effective your thinking is. The map is not the territory, so having more-accurate mental models makes your view of the world more accurate, and the more versatile your set of mental models is, the more diverse the set of problems you will be able to solve. Throughout this book you will learn a set of mental tools that will help you solve specific programming problems you will regularly come across in your career as a programmer.

Another way to look at mental models is to see them as a grouping of concepts into single unit of thought. The study of design patterns will aid you in developing new mental models. The structure of a problem will suggest the kind of solution you will need to implement in order to solve the problem in question. That is why it is important that you gain complete clarity into exactly what the problem is you are trying to solve.

The better—and by *better* I mean more complete—the problem definition or description is, the more hints you have of what a possible solution would look like. So, ask yourself dumb questions about the problem until you have a clear and simple problem statement.

Design patterns help you go from A (the problem statement) to C (the solution) without having to go through B and however many other false starts.

The Right Tools for the Job

To break down a brick wall, you need a hammer, but not just any hammer—you need a big, heavy hammer with a long handle. Sure, you can break down the wall with the same hammer you would use to put nails in a music box, but it will take you a couple of lifetimes to do what an afternoon's worth of hammering with the right tool will do.

With the wrong tools, any job becomes a mess and takes way longer than it should. Selecting the right tool for the job is a matter of experience. If you had no idea that there were hammers other than the tiny craft hammer you were used to, you would have a hard time imagining that someone could come along and take down a whole wall in a couple of hours. You might even call such a person a 10x wall breaker. The point I am trying to make is that with the right tool you will be able to do many times more work than someone who is trying to make do with what they have. It is worth the time and effort to expand your toolbox, and to master different tools, so that when you encounter a novel problem you will know which to select.

To become a master programmer, you need to consistently add new tools to your arsenal, not just familiarizing yourself with them, but mastering them. We already looked at the process for attaining mastery of the tools you decide on, but in the context of Python, let me make some specific suggestions.

Some of the beauties of the Python ecosystem are the packages available right out of the gate. There are a lot of packages, but more often than not you have one or two clear leaders for every type of problem you might encounter. These packages are valuable tools, and you should spend a couple of hours every week exploring them. Once you have mastered the patterns in this book, grab Numpy or Scipy and master them. Then, head off in any direction your imagination carries you. Download the package you are interested in, learn the basics, and then begin experimenting with it using the frameworks touched on already. Where do they shine, and what is missing? What kinds of problems are they especially good at solving? How can you use them in the future? What side project can you do that will allow you to try the package in question in a real-world scenario?

Design Patterns as a Concept

The Gang of Four's book on design patterns seems to be the place where it all started. They set forth a framework for describing design patterns (specifically for C++), but the description was focused on the solution in general, and as a result many of the patterns were translated into a number of languages. The goal of the 23 design patterns set forth in that book was to codify best-practice solutions to common problems encountered in object-oriented programming. As such, the solutions focus on classes and their methods and attributes.

These design patterns represent a single complete solution idea each, and they keep the things that change separate from the things that do not.

There are many people who are critical of the original design patterns. One of these critics, Peter Norvig, showed how 16 of the patterns could be replaced by language constructs in Lisp. Many of these replacements are possible in Python too.

In this book, we are going to look at several of the original design patterns and how they fit into real-world projects. We will also consider arguments about them in the context of the Python language, sometimes discarding the pattern for a solution that comes stock standard with the language, sometimes altering the GoF solution to take advantage of the power and expressiveness of Python, and other times simply implementing the original solution in a pythonic way.

What Makes a Design Pattern?

Design patterns can be a lot of things, but they all contain the following elements (credit: Peter Norvig, `http://norvig.com/design-patterns/ppframe.htm`):

- Pattern name
- Intent/purpose
- Aliases
- Motivation/context
- Problem
- Solution
- Structure
- Participants

- Collaborations
- Consequences/constraints
- Implementation
- Sample code
- Known uses
- Related patterns

In Appendix A, you can find all the design patterns we discuss in this book structured according to these elements. For the sake of readability and the learning process, the chapters on the design patterns themselves will not all follow this structure.

Classification

Design patterns are classified into different groupings to help us as programmers talk about categories of solutions with one another and to give us a common language when discussing these solutions. This allows us to communicate clearly and to be expressive in our discussions around the subject.

As mentioned earlier in this chapter, we are going to classify design patterns according to the original groupings of creational, structural, and behavioral patterns. This is done not only to stick to the general way things are done, but also to aid you, the reader, in looking at the patterns in the context of the systems they are found in.

Creational

The first category deals with the elements in the system—specifically, how they are created. As we are dealing with object-oriented programming, the creation of objects happens through class instantiation. As you will soon see, there are different characteristics that are desirable when it comes to solving specific problems, and the way in which an object is created has a significant effect on these characteristics.

Structural

The structural patterns deal with how classes and objects are composed. Objects can be composed using inheritance to obtain new functionality.

Behavioral

These design patterns are focused on the interaction between objects.

The Tools We Will Be Using

The world is shifting toward Python 3. This shift is slow and deliberate and, like a glacier, cannot be stopped. For the sake of this book, we will release Python 2 and embrace the future. That said, you should be able to make most of the code work with Python 2 without much trouble (this is less due to the code in the book and more a result of the brilliant work done by the Python core developers).

With Python, CPython (the default one) specifically, you get to use Pip, the Python Package Installer. Pip integrates with PyPI, the Python Package Index, and lets you download and install a package from the package index without manually downloading the package, uncompressing it, running python setup.py install and so on. Pip makes installing libraries for your environment a joy. Pip also deals with all the dependencies for you, so no need to run around after required packages you have not installed yet.

By the time you begin working on your third project, you are going to need a lot of packages. Not all of these packages are needed for all of your projects. You will want to keep every project's packages nicely contained. Enter VirtualEnv, a virtual Python interpreter that isolates the packages installed for that interpreter from others on the system. You get to keep each project in its own minimalist space, only installing the packages it needs in order to work, without interfering with the other projects you may be working on.

How to Read This Book

There are many ways to read a book, especially a programming book. Most people start a book like this hoping to read it from cover to cover and end up just jumping from code example to code example. We have all done it. With that in mind, there are ways you can read this book and gain optimal value from the time spent on it.

The first group of readers just wants a quick, robust reference of pythonic versions of the GoF design patterns. If this is you, skip ahead to Appendix A to see the formal definition of each of the design patterns. When you have time, you can then return to the relevant chapter and follow the exploration of the design pattern you are interested in.

The second group wants to be able to find specific solutions to specific problems. For those readers, each design pattern chapter starts with a description of the problem addressed by the pattern in question. This should help you decide if the pattern will be helpful in solving the problem you are faced with.

The last group wants to use this book to master their craft. For these readers, I suggest you begin at the beginning and code your way through the book. Type out every example. Tinker with every solution. Do all the exercises. See what happens when you alter the code. What breaks, and why? Make the solutions better. Tackle one pattern at a time and master it. Then, find other real-world contexts where you can apply your new knowledge.

Setting Up Your Python Environment

Let's begin by getting a working Python 3 environment running on your machine. In this section, we will look at installing Python 3 on Linux, Mac, and Windows.

On Linux

The commands we use are for *Ubuntu* with the *apt package manager.* For other distributions that do not work with apt, you can look at the process for installing Python 3 and `pip` using an alternative package manager, like yum, or installing the relevant packages from source.

The whole installation process will make use of the terminal, so you can go ahead and open it up now.

Let's begin by checking if you have a version of Python already installed on your system.

Just a note: For the duration of this book, I will indicate terminal commands with the leading `$`; you do not type this character or the subsequent space when entering the comment in your terminal.

```
$ python --version
```

If you have Python 3 (as is the case with Ubuntu 16.04 and up) already installed, you can skip ahead to the section on installing pip.

If you have either Python 2 or no Python installed on your system, you can go ahead and install Python 3 using the following command:

```
$ sudo apt-get install python3-dev
```

This will install Python 3. You can check the version of the Python install, as before, to verify that the right version was installed:

```
$ python3 --version
```

Python 3 should now be available on your system.

Next, we install build essentials and Python `pip`:

```
$ sudo apt-get install python-pip build-essential
```

Now, check that `pip` is working:

```
$ pip --version
```

You should see a version of `pip` installed on your system, and now you are ready to install the `virtualenv` package; please skip past the Mac and Windows installation instructions.

On Mac

macOS comes with a version of Python 2 installed by default, but we will not be needing this.

To install Python 3, we are going to use Homebrew, a command-line package manager for macOS. For this to work, we will need Xcode, which you can get for free from the Mac AppStore.

Once you have installed Xcode, open the Terminal app and install the command-line tools for Xcode:

```
$ xcode-select --install
```

Simply follow the prompt in the window that pops up to install the command-line tools for Xcode. Once that is done, you can install Homebrew:

```
$ ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/
master/install)"
```

If you do not have XQuartz already installed, your macOS might encounter some errors. If this happens, you can download the XQuartz `.dmg` here: `https://www.xquartz.org/`. Then, check that you have successfully installed Homebrew and that it is working:

```
$ brew doctor
```

To run `brew` commands from any folder in the Terminal, you need to add the Homebrew path to your `PATH` environment variable. Open or create `~/.bash_profile` and add the following line at the end of the file:

```
export PATH=/usr/local/bin:$PATH
```

Close and reopen Terminal. Upon reopening, the new `PATH` variable will be included in the environment, and now you can call `brew` from anywhere. Use `brew` to find the available packages for Python:

```
brew search python
```

You will now see all the Python-related packages, `python3` being one of them. Now, install Python 3 using the following `brew` command:

```
$ brew install python3
```

Finally, you can check that Python 3 is installed and working:

```
python3 --version
```

When you install Python with Homebrew, you also install the corresponding package manager (`pip`), Setuptools, and pyvenv (an alternative to `virtualenv`, but for this book you will only need `pip`).

Check that pip is working:

```
$ pip --version
```

If you see the version information, it means that `pip` was successfully installed on your system, and you can skip past the Windows installation section to the section on using `pip` to install VirtualEnv.

On Windows

Begin by downloading the Python 3 Windows installer. You can download the installer here: `https://www.python.org/`.

When your download is done, run the installer and select the Customize option. Make sure that pip is selected to be installed. Also, select the option to add Python to your environment variables.