Learn Python Programming

Second Edition

The no-nonsense, beginner's guide to programming, data science, and web development with Python 3.7

Packt®

www.packt.com

Fabrizio Romano

Learn Python Programming *Second Edition*

The no-nonsense, beginner's guide to programming, data science, and web development with Python 3.7

Fabrizio Romano

Packt

BIRMINGHAM - MUMBAI

Learn Python Programming
Second Edition

Commissioning Editor: Richa Tripathi
Acquisition Editor: Karan Sadawana
Content Development Editor: Rohit Singh
Technical Editor: Romy Dias
Copy Editor: Safis Editing
Project Coordinator: Vaidehi Sawant
Proofreader: Safis Editing
Indexer: Mariammal Chettiyar
Graphics: Jason Monteiro
Production Coordinator: Shantanu Zagade

First published: December 2015
Second edition: June 2018

Production reference: 2241018

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

ISBN 978-1-78899-666-2

`www.packtpub.com`

To my dear dear friend and mentor, Torsten Alexander Lange.
Thank you for all the love and support.

Foreword

I first got to know Fabrizio when he became our lead developer a few years ago. It was quickly apparent that he was one of those rare people who combine rigorous technical expertise with a genuine care for the people around him and a true passion to mentor and teach. Whether it was designing a system, pairing to write code, doing code reviews, or even organizing team card games at lunch, Fab was always thinking not only about the best way to do the job, but also about how to make sure that the entire team had the skills and motivation to do their best.

You'll meet the same wise and caring guide in this book. Every chapter, every example, every explanation has been carefully thought out, driven by a desire to impart the best and most accurate understanding of the technology, and to do it with kindness. Fab takes you under his wing to teach you both Python's syntax and its best practices.

I'm also impressed with the scope of this book. Python has grown and evolved over the years, and it now spans an enormous ecosystem, being used for web development, routine data handling, and ETL, and increasingly for data science. If you are new to the Python ecosystem, it's often hard to know what to study to achieve your goals. In this book, you will find useful examples exposing you to many different uses of Python, which will help guide you as you move through the breadth that Python offers.

I hope you will enjoy learning Python and become a member of our global community. I'm proud to have been asked to write this, but above all, I'm pleased that Fab will be your guide.

Naomi Ceder

Python Software Foundation Fellow

Contributors

About the author

Fabrizio Romano was born in Italy in 1975. He holds a master's degree in computer science engineering from the University of Padova. He is also a certified scrum master, Reiki master and teacher, and a member of CNHC.

He moved to London in 2011 to work for companies such as Glasses Direct, TBG/Sprinklr, and student.com. He now works at Sohonet as a Principal Engineer/Team Lead.

He has given talks on Teaching Python and TDD at two editions of EuroPython, and at Skillsmatter and ProgSCon, in London.

> *I'm grateful to all those who helped me create this book. Special thanks to Dr. Naomi Ceder for writing the foreword to this edition, and to Heinrich Kruger and Julio Trigo for reviewing this volume. To my friends and family, who love me and support me every day, thank you. And to Petra Lange, for always being so lovely to me, thank you.*

About the reviewers

Heinrich Kruger was born in South Africa in 1981. He obtained a bachelor's degree with honors from the University of the Witwatersrand in South Africa in 2005 and a master's degree in computer science from Utrecht University in the Netherlands in 2008.

He worked as a research assistant at Utrecht University from 2009 until 2013 and has been working as a professional software developer developer since 2014. He has been using Python for personal and projects and in his studies since 2004, and professionally since 2014.

Julio Vicente Trigo Guijarro is a computer science engineer with over a decade of experience in software development. He completed his studies at the University of Alicante, Spain, in 2007 and moved to London in 2010.

He has been using Python since 2012 and currently works as a senior software developer and team lead at Sohonet, developing real-time collaboration applications for the media industry.

He is also a certified ScrumMaster and was one of the technical reviewers of the first edition of this book.

I would like to thank my parents for their love, good advice, and continuous support. I would also like to thank all the friends I have met along the way, who enriched my life, for keeping up my motivation, and make me progress.

Table of Contents

Preface

When I started writing the first edition of this book, I knew very little about what was expected. Gradually, I learned how to convert each topic into a story. I wanted to talk about Python by offering useful, simple, easy-to-grasp examples, but, at the same time, I wanted to pour my own experience into the pages, anything I've learned over the years that I thought would be valuable for the reader—something to think about, reflect upon, and hopefully assimilate. Readers may disagree and come up with a different way of doing things, but hopefully a better way.

I wanted this book to not just be about the language but about programming. The art of programming, in fact, comprises many aspects, and language is just one of them.

Another crucial aspect of programming is independence. The ability to unblock yourself when you hit a wall and don't know what to do to solve the problem you're facing. There is no book that can teach it, so I thought, instead of trying to teach that aspect, I will try and train the reader in it. Therefore, I left comments, questions, and remarks scattered throughout the whole book, hoping to inspire the reader. I hoped that they would take the time to browse the Web or the official documentation, to dig deeper, learn more, and discover the pleasure of finding things out by themselves.

Finally, I wanted to write a book that, even in its presentation, would be slightly different. So, I decided, with my editor, to write the first part in a theoretical way, presenting topics that would describe the characteristics of Python, and to have a second part made up of various real-life projects, to show the reader how much can be achieved with this language.

With all these goals in mind, I then had to face the hardest challenge: take all the content I wanted to write and make it fit in the amount of pages that were allowed. It has been tough, and sacrifices were made.

My efforts have been rewarded though: to this day, after almost 3 years, I still receive lovely messages from readers, every now and then, who thank me and tell me things like *your book has empowered me*. To me, it is the most beautiful compliment. I know that the language might change and pass, but I have managed to share some of my knowledge with the reader, and that piece of knowledge will stick with them.

And now, I have written the second edition of this book, and this time, I had a little more space. So I decided to add a chapter about IO, which was desperately needed, and I even had the opportunity to add two more chapters, one about secrets and one about concurrent execution. The latter is definitely the most challenging chapter in the whole book, and its purpose is that of stimulating the reader to reach a level where they will be able to easily digest the code in it and understand its concepts.

I have kept all the original chapters, except for the last one that was slightly redundant. They have all been refreshed and updated to the latest version of Python, which is 3.7 at the time of writing.

When I look at this book, I see a much more mature product. There are more chapters, and the content has been reorganized to better fit the narrative, but the soul of the book is still there. The main and most important point, empowering the reader, is still very much intact.

I hope that this edition will be even more successful than the previous one, and that it will help the readers become great programmers. I hope to help them develop critical thinking, great skills, and the ability to adapt over time, thanks to the solid foundation they have acquired from the book.

Who this book is for

Python is the most popular introductory teaching language in the top computer science universities in the US, so if you are new to software development, or if you have little experience and would like to start off on the right foot, then this language and this book are what you need. Its amazing design and portability will help you to become productive regardless of the environment you choose to work with.

If you have already worked with Python or any other language, this book can still be useful to you, both as a reference to Python's fundamentals, and for providing a wide range of considerations and suggestions collected over two decades of experience.

What this book covers

`Chapter 1`, *A Gentle Introduction to Python*, introduces you to fundamental programming concepts. It guides you through getting Python up and running on your computer and introduces you to some of its constructs.

`Chapter 2`, *Built-in Data Types*, introduces you to Python built-in data types. Python has a very rich set of native data types, and this chapter will give you a description and a short example for each of them.

`Chapter 3`, *Iterating and Making Decisions*, teaches you how to control the flow of your code by inspecting conditions, applying logic, and performing loops.

`Chapter 4`, *Functions, the Building Blocks of Code*, teaches you how to write functions. Functions are the keys to reusing code, to reducing debugging time, and, in general, to writing better code.

`Chapter 5`, *Saving Time and Memory*, introduces you to the functional aspects of Python programming. This chapter teaches you how to write comprehensions and generators, which are powerful tools that you can use to speed up your code and save memory.

`Chapter 6`, *OOP, Decorators, and Iterators*, teaches you the basics of object-oriented programming with Python. It shows you the key concepts and all the potentials of this paradigm. It also shows you one of the most beloved characteristics of Python: decorators. Finally, it also covers the concept of iterators.

`Chapter 7`, *Files and Data Persistence*, teaches you how to deal with files, streams, data interchange formats, and databases, among other things.

`Chapter 8`, *Testing, Profiling, and Dealing with Exceptions*, teaches you how to make your code more robust, fast, and stable using techniques such as testing and profiling. It also formally defines the concept of exceptions.

`Chapter 9`, *Cryptography and Tokens*, touches upon the concepts of security, hashes, encryption, and tokens, which are part of day-to-day programming at present.

`Chapter 10`, *Concurrent Execution*, is a challenging chapter that describes how to do many things at the same time. It provides an introduction to the theoretical aspects of this subject and then presents three nice exercises that are developed with different techniques, thereby enabling the reader to understand the differences between the paradigms presented.

`Chapter 11`, *Debugging and Troubleshooting*, shows you the main methods for debugging your code and some examples on how to apply them.

`Chapter 12`, *GUIs and Scripts*, guides you through an example from two different points of view. They are at opposite ends of the spectrum: one implementation is a script, and another one is a proper graphical user interface application.

`Chapter 13`, *Data Science*, introduces a few key concepts and a very special tool, the Jupyter Notebook.

`Chapter 14`, *Web Development*, introduces the fundamentals of web development and delivers a project using the Django web framework. The example will be based on regular expressions.

To get the most out of this book

You are encouraged to follow the examples in this book. In order to do so, you will need a computer, an internet connection, and a browser. The book is written in Python 3.7, but it should also work, for the most part, with any recent Python 3.* version. I have given guidelines on how to install Python on your operating system. The procedures to do that change all the time, so you will need to refer to the most up-to-date guide on the Web to find precise setup instructions. I have also explained how to install all the extra libraries used in the various examples and provided suggestions if the reader finds any issues during the installation of any of them. No particular editor is required to type the code; however, I suggest that those who are interested in following the examples should consider adopting a proper coding environment. I have given suggestions on this matter in the first chapter.

Download the example code files

You can download the example code files for this book from your account at `www.packtpub.com`. If you purchased this book elsewhere, you can visit `www.packtpub.com/support` and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at `www.packtpub.com`.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Learn-Python-Programming-Second-Edition`. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Conventions used

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

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Within the `learn.pp` folder, we will create a virtual environment called `learnpp`."

A block of code is set as follows:

```
# we define a function, called local
def local():
    m = 7
    print(m)
```

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

```
# key.points.mutable.assignment.py
x = [1, 2, 3]
def func(x):
    x[1] = 42  # this changes the caller!
    x = 'something else'  # this points x to a new string object
```

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

```
>>> import sys
>>> print(sys.version)
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "To open the console in Windows, go to the **Start** menu, choose **Run**, and type `cmd`."

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 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 `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 `authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packtpub.com`.

1
A Gentle Introduction to Python

"Give a man a fish and you feed him for a day. Teach a man to fish and you feed him for a lifetime."

– Chinese proverb

According to Wikipedia, **computer programming** is:

"...a process that leads from an original formulation of a computing problem to executable computer programs. Programming involves activities such as analysis, developing understanding, generating algorithms, verification of requirements of algorithms including their correctness and resources consumption, and implementation (commonly referred to as coding) of algorithms in a target programming language."

In a nutshell, coding is telling a computer to do something using a language it understands.

Computers are very powerful tools, but unfortunately, they can't think for themselves. They need to be told everything: how to perform a task, how to evaluate a condition to decide which path to follow, how to handle data that comes from a device, such as the network or a disk, and how to react when something unforeseen happens, say, something is broken or missing.

You can code in many different styles and languages. Is it hard? I would say *yes* and *no*. It's a bit like writing. Everybody can learn how to write, and you can too. But, what if you wanted to become a poet? Then writing alone is not enough. You have to acquire a whole other set of skills and this will take a longer and greater effort.

In the end, it all comes down to how far you want to go down the road. Coding is not just putting together some instructions that work. It is so much more!

Good code is short, fast, elegant, easy to read and understand, simple, easy to modify and extend, easy to scale and refactor, and easy to test. It takes time to be able to write code that has all these qualities at the same time, but the good news is that you're taking the first step towards it at this very moment by reading this book. And I have no doubt you can do it. Anyone can; in fact, we all program all the time, only we aren't aware of it.

Would you like an example?

Say you want to make instant coffee. You have to get a mug, the instant coffee jar, a teaspoon, water, and the kettle. Even if you're not aware of it, you're evaluating a lot of data. You're making sure that there is water in the kettle and that the kettle is plugged in, that the mug is clean, and that there is enough coffee in the jar. Then, you boil the water and maybe, in the meantime, you put some coffee in the mug. When the water is ready, you pour it into the cup, and stir.

So, how is this programming?

Well, we gathered resources (the kettle, coffee, water, teaspoon, and mug) and we verified some conditions concerning them (the kettle is plugged in, the mug is clean, and there is enough coffee). Then we started two actions (boiling the water and putting coffee in the mug), and when both of them were completed, we finally ended the procedure by pouring water in to the mug and stirring.

Can you see it? I have just described the high-level functionality of a coffee program. It wasn't that hard because this is what the brain does all day long: evaluate conditions, decide to take actions, carry out tasks, repeat some of them, and stop at some point. Clean objects, put them back, and so on.

All you need now is to learn how to deconstruct all those actions you do automatically in real life so that a computer can actually make some sense of them. And you need to learn a language as well, to instruct it.

So this is what this book is for. I'll tell you how to do it and I'll try to do that by means of many simple but focused examples (my favorite kind).

In this chapter, we are going to cover the following:

- Python's characteristics and ecosystem
- Guidelines on how to get up and running with Python and virtual environments
- How to run Python programs
- How to organize Python code and Python's execution model

A proper introduction

I love to make references to the real world when I teach coding; I believe they help people retain the concepts better. However, now is the time to be a bit more rigorous and see what coding is from a more technical perspective.

When we write code, we're instructing a computer about the things it has to do. Where does the action happen? In many places: the computer memory, hard drives, network cables, the CPU, and so on. It's a whole *world*, which most of the time is the representation of a subset of the real world.

If you write a piece of software that allows people to buy clothes online, you will have to represent real people, real clothes, real brands, sizes, and so on and so forth, within the boundaries of a program.

In order to do so, you will need to create and handle objects in the program you're writing. A person can be an object. A car is an object. A pair of socks is an object. Luckily, Python understands objects very well.

The two main features any object has are properties and methods. Let's take a person object as an example. Typically in a computer program, you'll represent people as customers or employees. The properties that you store against them are things like the name, the SSN, the age, if they have a driving license, their email, gender, and so on. In a computer program, you store all the data you need in order to use an object for the purpose you're serving. If you are coding a website to sell clothes, you probably want to store the heights and weights as well as other measures of your customers so that you can suggest the appropriate clothes for them. So, properties are characteristics of an object. We use them all the time: *Could you pass me that pen?—Which one?—The black one*. Here, we used the *black* property of a pen to identify it (most likely among a blue and a red one).

Methods are things that an object can do. As a person, I have methods such as *speak*, *walk*, *sleep*, *wake up*, *eat*, *dream*, *write*, *read*, and so on. All the things that I can do could be seen as methods of the objects that represent me.

So, now that you know what objects are and that they expose methods that you can run and properties that you can inspect, you're ready to start coding. Coding in fact is simply about managing those objects that live in the subset of the world that we're reproducing in our software. You can create, use, reuse, and delete objects as you please.

According to the *Data Model* chapter on the official Python documentation (`https://docs.python.org/3/reference/datamodel.html`):

> *"Objects are Python's abstraction for data. All data in a Python program is represented by objects or by relations between objects."*

We'll take a closer look at Python objects in `Chapter 6`, *OOP, Decorators, and Iterators*. For now, all we need to know is that every object in Python has an ID (or identity), a type, and a value.

Once created, the ID of an object is never changed. It's a unique identifier for it, and it's used behind the scenes by Python to retrieve the object when we want to use it.

The type, as well, never changes. The type tells what operations are supported by the object and the possible values that can be assigned to it.

We'll see Python's most important data types in `Chapter 2`, *Built-in Data Types*.

The value can either change or not. If it can, the object is said to be **mutable**, while when it cannot, the object is said to be **immutable**.

How do we use an object? We give it a name, of course! When you give an object a name, then you can use the name to retrieve the object and use it.

In a more generic sense, objects such as numbers, strings (text), collections, and so on are associated with a name. Usually, we say that this name is the name of a variable. You can see the variable as being like a box, which you can use to hold data.

So, you have all the objects you need; what now? Well, we need to use them, right? We may want to send them over a network connection or store them in a database. Maybe display them on a web page or write them into a file. In order to do so, we need to react to a user filling in a form, or pressing a button, or opening a web page and performing a search. We react by running our code, evaluating conditions to choose which parts to execute, how many times, and under which circumstances.

And to do all this, basically we need a language. That's what Python is for. Python is the language we'll use together throughout this book to instruct the computer to do something for us.

Now, enough of this theoretical stuff; let's get started.

Enter the Python

Python is the marvelous creation of Guido Van Rossum, a Dutch computer scientist and mathematician who decided to gift the world with a project he was playing around with over Christmas 1989. The language appeared to the public somewhere around 1991, and since then has evolved to be one of the leading programming languages used worldwide today.

I started programming when I was 7 years old, on a Commodore VIC-20, which was later replaced by its bigger brother, the Commodore 64. Its language was *BASIC*. Later on, I landed on Pascal, Assembly, C, C++, Java, JavaScript, Visual Basic, PHP, ASP, ASP .NET, C#, and other minor languages I cannot even remember, but only when I landed on Python did I finally have that feeling that you have when you find the right couch in the shop. When all of your body parts are yelling, *Buy this one! This one is perfect for us!*

It took me about a day to get used to it. Its syntax is a bit different from what I was used to, but after getting past that initial feeling of discomfort (like having new shoes), I just fell in love with it. Deeply. Let's see why.

About Python

Before we get into the gory details, let's get a sense of why someone would want to use Python (I would recommend you to read the Python page on Wikipedia to get a more detailed introduction).

To my mind, Python epitomizes the following qualities.

Portability

Python runs everywhere, and porting a program from Linux to Windows or Mac is usually just a matter of fixing paths and settings. Python is designed for portability and it takes care of specific **operating system** (**OS**) quirks behind interfaces that shield you from the pain of having to write code tailored to a specific platform.

Coherence

Python is extremely logical and coherent. You can see it was designed by a brilliant computer scientist. Most of the time, you can just guess how a method is called, if you don't know it.

You may not realize how important this is right now, especially if you are at the beginning, but this is a major feature. It means less cluttering in your head, as well as less skimming through the documentation, and less need for mappings in your brain when you code.

Developer productivity

According to Mark Lutz (*Learning Python, 5th Edition*, *O'Reilly Media*), a Python program is typically one-fifth to one-third the size of equivalent Java or C++ code. This means the job gets done faster. And faster is good. Faster means a faster response on the market. Less code not only means less code to write, but also less code to read (and professional coders read much more than they write), less code to maintain, to debug, and to refactor.

Another important aspect is that Python runs without the need for lengthy and time-consuming compilation and linkage steps, so you don't have to wait to see the results of your work.

An extensive library

Python has an incredibly wide standard library (it's said to come with *batteries included*). If that wasn't enough, the Python community all over the world maintains a body of third-party libraries, tailored to specific needs, which you can access freely at the **Python Package Index** (**PyPI**). When you code Python and you realize that you need a certain feature, in most cases, there is at least one library where that feature has already been implemented for you.

Software quality

Python is heavily focused on readability, coherence, and quality. The language uniformity allows for high readability and this is crucial nowadays where coding is more of a collective effort than a solo endeavor. Another important aspect of Python is its intrinsic multiparadigm nature. You can use it as a scripting language, but you also can exploit object-oriented, imperative, and functional programming styles. It is versatile.

Software integration

Another important aspect is that Python can be extended and integrated with many other languages, which means that even when a company is using a different language as their mainstream tool, Python can come in and act as a glue agent between complex applications that need to talk to each other in some way. This is kind of an advanced topic, but in the real world, this feature is very important.

Satisfaction and enjoyment

Last, but not least, there is the fun of it! Working with Python is fun. I can code for 8 hours and leave the office happy and satisfied, alien to the struggle other coders have to endure because they use languages that don't provide them with the same amount of well-designed data structures and constructs. Python makes coding fun, no doubt about it. And fun promotes motivation and productivity.

These are the major aspects of why I would recommend Python to everyone. Of course, there are many other technical and advanced features that I could have talked about, but they don't really pertain to an introductory section like this one. They will come up naturally, chapter after chapter, in this book.

What are the drawbacks?

Probably, the only drawback that one could find in Python, which is not due to personal preferences, is its *execution speed*. Typically, Python is slower than its compiled brothers. The standard implementation of Python produces, when you run an application, a compiled version of the source code called byte code (with the extension `.pyc`), which is then run by the Python interpreter. The advantage of this approach is portability, which we pay for with a slowdown due to the fact that Python is not compiled down to machine level as are other languages.

However, Python speed is rarely a problem today, hence its wide use regardless of this suboptimal feature. What happens is that, in real life, hardware cost is no longer a problem, and usually it's easy enough to gain speed by parallelizing tasks. Moreover, many programs spend a great proportion of the time waiting for IO operations to complete; therefore, the raw execution speed is often a secondary factor to the overall performance. When it comes to number crunching though, one can switch to faster Python implementations, such as PyPy, which provides an average five-fold speedup by implementing advanced compilation techniques (check `http://pypy.org/` for reference).

When doing data science, you'll most likely find that the libraries that you use with Python, such as **Pandas** and **NumPy**, achieve native speed due to the way they are implemented.

If that wasn't a good-enough argument, you can always consider that Python has been used to drive the backend of services such as Spotify and Instagram, where performance is a concern. Nonetheless, Python has done its job perfectly adequately.

Who is using Python today?

Not yet convinced? Let's take a very brief look at the companies that are using Python today: Google, YouTube, Dropbox, Yahoo!, Zope Corporation, Industrial Light & Magic, Walt Disney Feature Animation, Blender 3D, Pixar, NASA, the NSA, Red Hat, Nokia, IBM, Netflix, Yelp, Intel, Cisco, HP, Qualcomm, and JPMorgan Chase, to name just a few.

Even games such as *Battlefield 2*, *Civilization IV*, and *QuArK* are implemented using Python.

Python is used in many different contexts, such as system programming, web programming, GUI applications, gaming and robotics, rapid prototyping, system integration, data science, database applications, and much more. Several prestigious universities have also adopted Python as their main language in computer science courses.

Setting up the environment

Before we talk about installing Python on your system, let me tell you about which Python version I'll be using in this book.

Python 2 versus Python 3

Python comes in two main versions: Python 2, which is the past, and Python 3, which is the present. The two versions, though very similar, are incompatible in some respects.

In the real world, Python 2 is actually quite far from being the past. In short, even though Python 3 has been out since 2008, the transition phase from Version 2 is still far from being over. This is mostly due to the fact that Python 2 is widely used in the industry, and of course, companies aren't so keen on updating their systems just for the sake of updating them, following the *if it ain't broke, don't fix it* philosophy. You can read all about the transition between the two versions on the web.

Another issue that has hindered the transition is the availability of third-party libraries. Usually, a Python project relies on tens of external libraries, and of course, when you start a new project, you need to be sure that there is already a Version-3-compatible library for any business requirement that may come up. If that's not the case, starting a brand-new project in Python 3 means introducing a potential risk, which many companies are not happy to take.

At the time of writing, though, the majority of the most widely used libraries have been ported to Python 3, and it's quite safe to start a project in Python 3 for most cases. Many of the libraries have been rewritten so that they are compatible with both versions, mostly harnessing the power of the `six` library (the name comes from the multiplication 2 x 3, due to the porting from Version 2 to 3), which helps introspecting and adapting the behavior according to the version used. According to PEP 373 (`https://legacy.python.org/dev/peps/pep-0373/`), the **end of life** (**EOL**) of Python 2.7 has been set to 2020, and there won't be a Python 2.8, so this is the time when companies that have projects running in Python 2 need to start devising an upgrade strategy to move to Python 3 before it's too late.

On my box (MacBook Pro), this is the latest Python version I have:

```
>>> import sys
>>> print(sys.version)
3.7.0a3 (default, Jan 27 2018, 00:46:45)
[Clang 9.0.0 (clang-900.0.39.2)]
```

So you can see that the version is an alpha release of Python 3.7, which will be released in June 2018. The preceding text is a little bit of Python code that I typed into my console. We'll talk about it in a moment.

All the examples in this book will be run using Python 3.7. Even though at the moment the final version might still be slightly different than what I have, I will make sure that all the code and examples are up to date with 3.7 by the time the book is published.

Some of the code can also run in Python 2.7, either as it is or with minor tweaks, but at this point in time, I think it's better to learn Python 3, and then, if you need to, learn the differences it has with Python 2, rather than going the other way around.

Don't worry about this version thing though; it's not that big an issue in practice.

Installing Python

I never really got the point of having a *setup* section in a book, regardless of what it is that you have to set up. Most of the time, between the time the author writes the instructions and the time you actually try them out, months have passed. That is, if you're lucky. One version change and things may not work in the way that is described in the book. Luckily, we have the web now, so in order to help you get up and running, I'll just give you pointers and objectives.

I am conscious that the majority of readers would probably have preferred to have guidelines in the book. I doubt it would have made their life much easier, as I strongly believe that if you want to get started with Python you have to put in that initial effort in order to get familiar with the ecosystem. It is very important, and it will boost your confidence to face the material in the chapters ahead. If you get stuck, remember that Google is your friend.

Setting up the Python interpreter

First of all, let's talk about your OS. Python is fully integrated and most likely already installed in basically almost every Linux distribution. If you have a macOS, it's likely that Python is already there as well (however, possibly only Python 2.7), whereas if you're using Windows, you probably need to install it.

Getting Python and the libraries you need up and running requires a bit of handiwork. Linux and macOS seem to be the most user-friendly OSes for Python programmers; Windows, on the other hand, is the one that requires the biggest effort.

My current system is a MacBook Pro, and this is what I will use throughout the book, along with Python 3.7.

The place you want to start is the official Python website: `https://www.python.org`. This website hosts the official Python documentation and many other resources that you will find very useful. Take the time to explore it.

TIP

Another excellent, resourceful website on Python and its ecosystem is `http://docs.python-guide.org`. You can find instructions to set up Python on different operating systems, using different methods.

Find the download section and choose the installer for your OS. If you are on Windows, make sure that when you run the installer, you check the option `install pip` (actually, I would suggest to make a complete installation, just to be safe, of all the components the installer holds). We'll talk about `pip` later.

Now that Python is installed in your system, the objective is to be able to open a console and run the Python interactive shell by typing `python`.

Please note that I usually refer to the **Python interactive shell** simply as the **Python console**.

To open the console in Windows, go to the **Start** menu, choose **Run**, and type `cmd`. If you encounter anything that looks like a permission problem while working on the examples in this book, please make sure you are running the console with administrator rights.

On the macOS X, you can start a Terminal by going to **Applications** | **Utilities** | **Terminal**.

If you are on Linux, you know all that there is to know about the console.

I will use the term *console* interchangeably to indicate the Linux console, the Windows Command Prompt, and the Macintosh Terminal. I will also indicate the command-line prompt with the Linux default format, like this:

```
$ sudo apt-get update
```

If you're not familiar with that, please take some time to learn the basics on how a console works. In a nutshell, after the `$` sign, you normally find an instruction that you have to type. Pay attention to capitalization and spaces, as they are very important.

Whatever console you open, type `python` at the prompt, and make sure the Python interactive shell shows up. Type `exit()` to quit. Keep in mind that you may have to specify `python3` if your OS comes with Python 2.* preinstalled.

This is roughly what you should see when you run Python (it will change in some details according to the version and OS):

```
$ python3.7
Python 3.7.0a3 (default, Jan 27 2018, 00:46:45)
[Clang 9.0.0 (clang-900.0.39.2)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Now that Python is set up and you can run it, it's time to make sure you have the other tool that will be indispensable to follow the examples in the book: virtualenv.

About virtualenv

As you probably have guessed by its name, **virtualenv** is all about virtual environments. Let me explain what they are and why we need them and let me do it by means of a simple example.

You install Python on your system and you start working on a website for Client X. You create a project folder and start coding. Along the way, you also install some libraries; for example, the Django framework, which we'll see in depth in `Chapter 14`, *Web Development*. Let's say the Django version you install for Project X is 1.7.1.

Now, your website is so good that you get another client, Y. She wants you to build another website, so you start Project Y and, along the way, you need to install Django again. The only issue is that now the Django version is 1.8 and you cannot install it on your system because this would replace the version you installed for Project X. You don't want to risk introducing incompatibility issues, so you have two choices: either you stick with the version you have currently on your machine, or you upgrade it and make sure the first project is still fully working correctly with the new version.

Let's be honest, neither of these options is very appealing, right? Definitely not. So, here's the solution: virtualenv!

virtualenv is a tool that allows you to create a virtual environment. In other words, it is a tool to create isolated Python environments, each of which is a folder that contains all the necessary executables to use the packages that a Python project would need (think of packages as libraries for the time being).

So you create a virtual environment for Project X, install all the dependencies, and then you create a virtual environment for Project Y, installing all its dependencies without the slightest worry because every library you install ends up within the boundaries of the appropriate virtual environment. In our example, Project X will hold Django 1.7.1, while Project Y will hold Django 1.8.

It is of vital importance that you never install libraries directly at the system level. Linux, for example, relies on Python for many different tasks and operations, and if you fiddle with the system installation of Python, you risk compromising the integrity of the whole system (guess to whom this happened...). So take this as a rule, such as brushing your teeth before going to bed: *always, always create a virtual environment when you start a new project*.

To install virtualenv on your system, there are a few different ways. On a Debian-based distribution of Linux, for example, you can install it with the following command:

```
$ sudo apt-get install python-virtualenv
```

Probably, the easiest way is to follow the instructions you can find on the virtualenv official website: `https://virtualenv.pypa.io`.

You will find that one of the most common ways to install virtualenv is by using `pip`, a package management system used to install and manage software packages written in Python.

As of Python 3.5, the suggested way to create a virtual environment is to use the `venv` module. Please see the `official documentation` for further information. However, at the time of writing, virtualenv is still by far the tool most used for creating virtual environments.

Your first virtual environment

It is very easy to create a virtual environment, but according to how your system is configured and which Python version you want the virtual environment to run, you need to run the command properly. Another thing you will need to do with virtualenv, when you want to work with it, is to activate it. Activating virtualenv basically produces some path juggling behind the scenes so that when you call the Python interpreter, you're actually calling the active virtual environment one, instead of the mere system one.

I'll show you a full example on my Macintosh console. We will:

1. Create a folder named `learn.pp` under your project root (which in my case is a folder called `srv`, in my home folder). Please adapt the paths according to the setup you fancy on your box.
2. Within the `learn.pp` folder, we will create a virtual environment called `learnpp`.

Some developers prefer to call all virtual environments using the same name (for example, `.venv`). This way they can run scripts against any virtualenv by just knowing the name of the project they dwell in. The dot in `.venv` is there because in Linux/macOS prepending a name with a dot makes that file or folder invisible.

3. After creating the virtual environment, we will activate it. The methods are slightly different between Linux, macOS, and Windows.
4. Then, we'll make sure that we are running the desired Python version (3.7.*) by running the Python interactive shell.
5. Finally, we will deactivate the virtual environment using the `deactivate` command.

These five simple steps will show you all you have to do to start and use a project.

Here's an example of how those steps might look (note that you might get a slightly different result, according to your OS, Python version, and so on) on the macOS (commands that start with a # are comments, spaces have been introduced for readability, and ⇢ indicates where the line has wrapped around due to lack of space):

```
fabmp:srv fab$ # step 1 - create folder
fabmp:srv fab$ mkdir learn.pp
fabmp:srv fab$ cd learn.pp

fabmp:learn.pp fab$ # step 2 - create virtual environment
fabmp:learn.pp fab$ which python3.7
/Users/fab/.pyenv/shims/python3.7
fabmp:learn.pp fab$ virtualenv -p
⇢ /Users/fab/.pyenv/shims/python3.7 learnpp
Running virtualenv with interpreter /Users/fab/.pyenv/shims/python3.7
Using base prefix '/Users/fab/.pyenv/versions/3.7.0a3'
New python executable in /Users/fab/srv/learn.pp/learnpp/bin/python3.7
Also creating executable in /Users/fab/srv/learn.pp/learnpp/bin/python
Installing setuptools, pip, wheel...done.

fabmp:learn.pp fab$ # step 3 - activate virtual environment
```

```
fabmp:learn.pp fab$ source learnpp/bin/activate

(learnpp) fabmp:learn.pp fab$ # step 4 - verify which python
(learnpp) fabmp:learn.pp fab$ which python
/Users/fab/srv/learn.pp/learnpp/bin/python

(learnpp) fabmp:learn.pp fab$ python
Python 3.7.0a3 (default, Jan 27 2018, 00:46:45)
[Clang 9.0.0 (clang-900.0.39.2)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> exit()

(learnpp) fabmp:learn.pp fab$ # step 5 - deactivate
(learnpp) fabmp:learn.pp fab$ deactivate
fabmp:learn.pp fab$
```

Notice that I had to tell virtualenv explicitly to use the Python 3.7 interpreter because on my box Python 2.7 is the default one. Had I not done that, I would have had a virtual environment with Python 2.7 instead of Python 3.7.

You can combine the two instructions for step 2 in one single command like this:

```
$ virtualenv -p $( which python3.7 ) learnpp
```

I chose to be explicitly verbose in this instance, to help you understand each bit of the procedure.

Another thing to notice is that in order to activate a virtual environment, we need to run the `/bin/activate` script, which needs to be sourced. When a script is **sourced**, it means that it is executed in the current shell, and therefore its effects last after the execution. This is very important. Also notice how the prompt changes after we activate the virtual environment, showing its name on the left (and how it disappears when we deactivate it). On Linux, the steps are the same so I won't repeat them here. On Windows, things change slightly, but the concepts are the same. Please refer to the official virtualenv website for guidance.

At this point, you should be able to create and activate a virtual environment. Please try and create another one without me guiding you. Get acquainted with this procedure because it's something that you will always be doing: **we never work system-wide with Python**, remember? It's extremely important.

So, with the scaffolding out of the way, we're ready to talk a bit more about Python and how you can use it. Before we do that though, allow me to speak a few words about the console.

Your friend, the console

In this era of GUIs and touchscreen devices, it seems a little ridiculous to have to resort to a tool such as the console, when everything is just about one click away.

But the truth is every time you remove your right hand from the keyboard (or the left one, if you're a lefty) to grab your mouse and move the cursor over to the spot you want to click on, you're losing time. Getting things done with the console, counter-intuitive as it may be, results in higher productivity and speed. I know, you have to trust me on this.

Speed and productivity are important and, personally, I have nothing against the mouse, but there is another very good reason for which you may want to get well-acquainted with the console: when you develop code that ends up on some server, the console might be the only available tool. If you make friends with it, I promise you, you will never get lost when it's of utmost importance that you don't (typically, when the website is down and you have to investigate very quickly what's going on).

So it's really up to you. If you're undecided, please grant me the benefit of the doubt and give it a try. It's easier than you think, and you'll never regret it. There is nothing more pitiful than a good developer who gets lost within an SSH connection to a server because they are used to their own custom set of tools, and only to that.

Now, let's get back to Python.

How you can run a Python program

There are a few different ways in which you can run a Python program.

Running Python scripts

Python can be used as a scripting language. In fact, it always proves itself very useful. Scripts are files (usually of small dimensions) that you normally execute to do something like a task. Many developers end up having their own arsenal of tools that they fire when they need to perform a task. For example, you can have scripts to parse data in a format and render it into another different format. Or you can use a script to work with files and folders. You can create or modify configuration files, and much more. Technically, there is not much that cannot be done in a script.

It's quite common to have scripts running at a precise time on a server. For example, if your website database needs cleaning every 24 hours (for example, the table that stores the user sessions, which expire pretty quickly but aren't cleaned automatically), you could set up a Cron job that fires your script at 3:00 A.M. every day.

According to Wikipedia, the software utility Cron is a time-based job scheduler in Unix-like computer operating systems. People who set up and maintain software environments use Cron to schedule jobs (commands or shell scripts) to run periodically at fixed times, dates, or intervals.

I have Python scripts to do all the menial tasks that would take me minutes or more to do manually, and at some point, I decided to automate. We'll devote half of `Chapter 12`, *GUIs and Scripts,* on scripting with Python.

Running the Python interactive shell

Another way of running Python is by calling the interactive shell. This is something we already saw when we typed `python` on the command line of our console.

So, open a console, activate your virtual environment (which by now should be second nature to you, right?), and type `python`. You will be presented with a couple of lines that should look like this:

```
$ python
Python 3.7.0a3 (default, Jan 27 2018, 00:46:45)
[Clang 9.0.0 (clang-900.0.39.2)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Those >>> are the prompt of the shell. They tell you that Python is waiting for you to type something. If you type a simple instruction, something that fits in one line, that's all you'll see. However, if you type something that requires more than one line of code, the shell will change the prompt to `...`, giving you a visual clue that you're typing a multiline statement (or anything that would require more than one line of code).

Go on, try it out; let's do some basic math:

```
>>> 2 + 4
6
>>> 10 / 4
2.5
>>> 2 ** 1024
179769313486231590772930519078902473361797697894230657273430081157732675805
```

```
    500963132708477322407536021120113879871393357658789768814416622492847430639
    474124377767893424865485276302219601246094119453082952085005768838150682342
    462881473913110540827237163350510684586298239947245938479716304835356329624
    224137216
```

The last operation is showing you something incredible. We raise `2` to the power of `1024`, and Python is handling this task with no trouble at all. Try to do it in Java, C++, or C#. It won't work, unless you use special libraries to handle such big numbers.

I use the interactive shell every day. It's extremely useful to debug very quickly, for example, to check if a data structure supports an operation. Or maybe to inspect or run a piece of code.

When you use Django (a web framework), the interactive shell is coupled with it and allows you to work your way through the framework tools, to inspect the data in the database, and many more things. You will find that the interactive shell will soon become one of your dearest friends on the journey you are embarking on.

Another solution, which comes in a much nicer graphic layout, is to use **Integrated DeveLopment Environment** (**IDLE**). It's quite a simple IDE, which is intended mostly for beginners. It has a slightly larger set of capabilities than the naked interactive shell you get in the console, so you may want to explore it. It comes for free in the Windows Python installer and you can easily install it in any other system. You can find information about it on the Python website.

Guido Van Rossum named Python after the British comedy group, Monty Python, so it's rumored that the name IDLE has been chosen in honor of Eric Idle, one of Monty Python's founding members.

Running Python as a service

Apart from being run as a script, and within the boundaries of a shell, Python can be coded and run as an application. We'll see many examples throughout the book about this mode. And we'll understand more about it in a moment, when we'll talk about how Python code is organized and run.

Running Python as a GUI application

Python can also be run as a **graphical user interface** (**GUI**). There are several frameworks available, some of which are cross-platform and some others are platform-specific. In `Chapter 12`, *GUIs and Scripts*, we'll see an example of a GUI application created using Tkinter, which is an object-oriented layer that lives on top of **Tk** (Tkinter means Tk interface).

Tk is a GUI toolkit that takes desktop application development to a higher level than the conventional approach. It is the standard GUI for **Tool Command Language** (**Tcl**), but also for many other dynamic languages, and it can produce rich native applications that run seamlessly under Windows, Linux, macOS X, and more.

Tkinter comes bundled with Python; therefore, it gives the programmer easy access to the GUI world, and for these reasons, I have chosen it to be the framework for the GUI examples that I'll present in this book.

Among the other GUI frameworks, we find that the following are the most widely used:

- PyQt
- wxPython
- PyGTK

Describing them in detail is outside the scope of this book, but you can find all the information you need on the Python website (`https://docs.python.org/3/faq/gui.html`) in the *What platform-independent GUI toolkits exist for Python?* section. If GUIs are what you're looking for, remember to choose the one you want according to some principles. Make sure they:

- Offer all the features you may need to develop your project
- Run on all the platforms you may need to support
- Rely on a community that is as wide and active as possible
- Wrap graphic drivers/tools that you can easily install/access

How is Python code organized?

Let's talk a little bit about how Python code is organized. In this section, we'll start going down the rabbit hole a little bit more and introduce more technical names and concepts.

Starting with the basics, how is Python code organized? Of course, you write your code into files. When you save a file with the extension `.py`, that file is said to be a Python module.

If you're on Windows or macOS that typically hide file extensions from the user, please make sure you change the configuration so that you can see the complete names of the files. This is not strictly a requirement, but a suggestion.

It would be impractical to save all the code that it is required for software to work within one single file. That solution works for scripts, which are usually not longer than a few hundred lines (and often they are quite shorter than that).

A complete Python application can be made of hundreds of thousands of lines of code, so you will have to scatter it through different modules, which is better, but not nearly good enough. It turns out that even like this, it would still be impractical to work with the code. So Python gives you another structure, called **package**, which allows you to group modules together. A package is nothing more than a folder, which must contain a special file, `__init__.py`, that doesn't need to hold any code but whose presence is required to tell Python that the folder is not just some folder, but it's actually a package (note that as of Python 3.3, the `__init__.py` module is not strictly required any more).

As always, an example will make all of this much clearer. I have created an example structure in my book project, and when I type in my console:

```
$ tree -v example
```

I get a tree representation of the contents of the `ch1/example` folder, which holds the code for the examples of this chapter. Here's what the structure of a really simple application could look like:

```
example
├── core.py
├── run.py
└── util
    ├── __init__.py
    ├── db.py
    ├── math.py
    └── network.py
```

You can see that within the root of this example, we have two modules, `core.py` and `run.py`, and one package: `util`. Within `core.py`, there may be the core logic of our application. On the other hand, within the `run.py` module, we can probably find the logic to start the application. Within the `util` package, I expect to find various utility tools, and in fact, we can guess that the modules there are named based on the types of tools they hold: `db.py` would hold tools to work with databases, `math.py` would, of course, hold mathematical tools (maybe our application deals with financial data), and `network.py` would probably hold tools to send/receive data on networks.

As explained before, the `__init__.py` file is there just to tell Python that `util` is a package and not just a mere folder.

Had this software been organized within modules only, it would have been harder to infer its structure. I put a *module only* example under the `ch1/files_only` folder; see it for yourself:

```
$ tree -v files_only
```

This shows us a completely different picture:

```
files_only/
├── core.py
├── db.py
├── math.py
├── network.py
└── run.py
```

It is a little harder to guess what each module does, right? Now, consider that this is just a simple example, so you can guess how much harder it would be to understand a real application if we couldn't organize the code in packages and modules.

How do we use modules and packages?

When a developer is writing an application, it is likely that they will need to apply the same piece of logic in different parts of it. For example, when writing a parser for the data that comes from a form that a user can fill in a web page, the application will have to validate whether a certain field is holding a number or not. Regardless of how the logic for this kind of validation is written, it's likely that it will be needed in more than one place.

For example, in a poll application, where the user is asked many questions, it's likely that several of them will require a numeric answer. For example:

- What is your age?
- How many pets do you own?
- How many children do you have?
- How many times have you been married?

It would be very bad practice to copy/paste (or, more properly said: duplicate) the validation logic in every place where we expect a numeric answer. This would violate the **don't repeat yourself** (**DRY**) principle, which states that you should never repeat the same piece of code more than once in your application. I feel the need to stress the importance of this principle: *you should never repeat the same piece of code more than once in your application* (pun intended).

There are several reasons why repeating the same piece of logic can be very bad, the most important ones being:

- There could be a bug in the logic, and therefore, you would have to correct it in every place that the logic is applied.
- You may want to amend the way you carry out the validation, and again you would have to change it in every place it is applied.
- You may forget to fix/amend a piece of logic because you missed it when searching for all its occurrences. This would leave wrong/inconsistent behavior in your application.
- Your code would be longer than needed, for no good reason.

Python is a wonderful language and provides you with all the tools you need to apply all the coding best practices. For this particular example, we need to be able to reuse a piece of code. To be able to reuse a piece of code, we need to have a construct that will hold the code for us so that we can call that construct every time we need to repeat the logic inside it. That construct exists, and it's called a **function**.

I'm not going too deep into the specifics here, so please just remember that a function is a block of organized, reusable code that is used to perform a task. Functions can assume many forms and names, according to what kind of environment they belong to, but for now this is not important. We'll see the details when we are able to appreciate them, later on, in the book. Functions are the building blocks of modularity in your application, and they are almost indispensable. Unless you're writing a super-simple script, you'll use functions all the time. We'll explore functions in `Chapter 4`, *Functions, the Building Blocks of Code*.

Python comes with a very extensive library, as I have already said a few pages ago. Now, maybe it's a good time to define what a library is: a **library** is a collection of functions and objects that provide functionalities that enrich the abilities of a language.

For example, within Python's `math` library, we can find a plethora of functions, one of which is the `factorial` function, which of course calculates the factorial of a number.

> In mathematics, the **factorial** of a non-negative integer number *N*, denoted as *N!*, is defined as the product of all positive integers less than or equal to *N*. For example, the factorial of `5` is calculated as:
> `5! = 5 * 4 * 3 * 2 * 1 = 120`
> The factorial of `0` is `0! = 1`, to respect the convention for an empty product.

So, if you wanted to use this function in your code, all you would have to do is to import it and call it with the right input values. Don't worry too much if input values and the concept of calling is not very clear for now; please just concentrate on the import part. We use a library by importing what we need from it, and then we use it.

In Python, to calculate the factorial of number `5`, we just need the following code:

```
>>> from math import factorial
>>> factorial(5)
120
```

> Whatever we type in the shell, if it has a printable representation, will be printed on the console for us (in this case, the result of the function call: `120`).

So, let's go back to our example, the one with `core.py`, `run.py`, `util`, and so on.

In our example, the package `util` is our utility library. Our custom utility belt that holds all those reusable tools (that is, functions), which we need in our application. Some of them will deal with databases (`db.py`), some with the network (`network.py`), and some will perform mathematical calculations (`math.py`) that are outside the scope of Python's standard `math` library and, therefore, we have to code them for ourselves.

We will see in detail how to import functions and use them in their dedicated chapter. Let's now talk about another very important concept: *Python's execution model.*

Python's execution model

In this section, I would like to introduce you to a few very important concepts, such as scope, names, and namespaces. You can read all about Python's execution model in the official language reference, of course, but I would argue that it is quite technical and abstract, so let me give you a less formal explanation first.

Names and namespaces

Say you are looking for a book, so you go to the library and ask someone for the book you want to fetch. They tell you something like *Second Floor, Section X, Row Three*. So you go up the stairs, look for Section X, and so on.

It would be very different to enter a library where all the books are piled together in random order in one big room. No floors, no sections, no rows, no order. Fetching a book would be extremely hard.

When we write code, we have the same issue: we have to try and organize it so that it will be easy for someone who has no prior knowledge about it to find what they're looking for. When software is structured correctly, it also promotes code reuse. On the other hand, disorganized software is more likely to expose scattered pieces of duplicated logic.

First of all, let's start with the book. We refer to a book by its title and in Python lingo, that would be a name. Python names are the closest abstraction to what other languages call variables. Names basically refer to objects and are introduced by name-binding operations. Let's make a quick example (notice that anything that follows a # is a comment):

```
>>> n = 3  # integer number
>>> address = "221b Baker Street, NW1 6XE, London"  # Sherlock Holmes'
address
>>> employee = {
...     'age': 45,
...     'role': 'CTO',
...     'SSN': 'AB1234567',
... }
>>> # let's print them
>>> n
3
>>> address
'221b Baker Street, NW1 6XE, London'
>>> employee
{'age': 45, 'role': 'CTO', 'SSN': 'AB1234567'}
>>> other_name
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
NameError: name 'other_name' is not defined
```

We defined three objects in the preceding code (do you remember what are the three features every Python object has?):

- An integer number `n` (type: `int`, value: `3`)
- A string `address` (type: `str`, value: Sherlock Holmes' address)
- A dictionary `employee` (type: `dict`, value: a dictionary that holds three key/value pairs)

Don't worry, I know you're not supposed to know what a dictionary is. We'll see in `Chapter 2`, *Built-in Data Types*, that it's the king of Python data structures.

Have you noticed that the prompt changed from `>>>` to `...` when I typed in the definition of employee? That's because the definition spans over multiple lines.

So, what are `n`, `address`, and `employee`? They are **names**. Names that we can use to retrieve data within our code. They need to be kept somewhere so that whenever we need to retrieve those objects, we can use their names to fetch them. We need some space to hold them, hence: namespaces!

A **namespace** is therefore a mapping from names to objects. Examples are the set of built-in names (containing functions that are always accessible in any Python program), the global names in a module, and the local names in a function. Even the set of attributes of an object can be considered a namespace.

The beauty of namespaces is that they allow you to define and organize your names with clarity, without overlapping or interference. For example, the namespace associated with that book we were looking for in the library can be used to import the book itself, like this:

```
from library.second_floor.section_x.row_three import book
```

We start from the `library` namespace, and by means of the dot (`.`) operator, we walk into that namespace. Within this namespace, we look for `second_floor`, and again we walk into it with the `.` operator. We then walk into `section_x`, and finally within the last namespace, `row_three`, we find the name we were looking for: `book`.

Walking through a namespace will be clearer when we'll be dealing with real code examples. For now, just keep in mind that namespaces are places where names are associated with objects.

There is another concept, which is closely related to that of a namespace, which I'd like to briefly talk about: the **scope**.

Scopes

According to Python's documentation:

> *" A scope is a textual region of a Python program, where a namespace is directly accessible."*

Directly accessible means that when you're looking for an unqualified reference to a name, Python tries to find it in the namespace.

Scopes are determined statically, but actually, during runtime, they are used dynamically. This means that by inspecting the source code, you can tell what the scope of an object is, but this doesn't prevent the software from altering that during runtime. There are four different scopes that Python makes accessible (not necessarily all of them are present at the same time, of course):

- The **local** scope, which is the innermost one and contains the local names.
- The **enclosing** scope, that is, the scope of any enclosing function. It contains non-local names and also non-global names.
- The **global** scope contains the global names.
- The **built-in** scope contains the built-in names. Python comes with a set of functions that you can use in an off-the-shelf fashion, such as `print`, `all`, `abs`, and so on. They live in the built-in scope.

The rule is the following: when we refer to a name, Python starts looking for it in the current namespace. If the name is not found, Python continues the search to the enclosing scope and this continues until the built-in scope is searched. If a name hasn't been found after searching the built-in scope, then Python raises a `NameError` **exception**, which basically means that the name hasn't been defined (you saw this in the preceding example).