Learning Python has a dynamic and varied nature. It reads easily and lays a good foundation for those who are interested in digging deeper. It has a practical and example-oriented approach through which both introductory and advanced topics are explained.

The book begins by exploring the essentials of programming data structures and teaches you how to manipulate them. It then moves on to controlling the flow of a program and writing reusable and error-proof code. You will then explore different programming paradigms that will allow you to find the best approach to any situation, and also learn how to perform performance optimization as well as effective debugging. Throughout, the book steers you through the various types of application, and it concludes with a complete mini website built using all the concepts that you learned.

Who this book is written for

Python is the most popular introductory teaching language in top computer science universities, so if you are new to software development, or maybe you have some experience and would like to start off on the right foot, then this language and this book are what you need.

What you will learn from this book

- Get Python up and running on Windows, Mac, and Linux in no time
- Grasp the fundamental concepts of coding, along with the basics of data structures and control flow
- Write elegant, reusable, and efficient code in any situation
- Understand when to use the functional or the object-oriented programming approach
- Create bulletproof, reliable software by writing tests to support your code
- Explore examples of GUIs, scripting, data science, and web applications
- Learn to be independent, capable of fetching any resource you need

Fabrizio Romano

Learn to code like a professional with Python – an open source, versatile, and powerful programming language

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Introduction and First Steps – Take a Deep Breath'
- A synopsis of the book’s content
- More information on Learning Python
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.

Before Python, he has worked with several other languages, such as C/C++, Java, PHP, and C#.

In 2011, he moved to London and started working as a Python developer for Glasses Direct, one of Europe's leading online prescription glasses retailers.

He then worked as a senior Python developer for TBG (now Sprinklr), one of the world's leading companies in social media advertising. At TBG, he and his team collaborated with Facebook and Twitter. They were the first in the world to get access to the Twitter advertising API. He wrote the code that published the first geo-narrowcasted promoted tweet in the world using the API.

He currently works as a senior platform developer at Student.com, a company that is revolutionizing the way international students find their perfect home all around the world.

He has delivered talks on *Teaching Python* and *TDD with Python* at the last two editions of EuroPython and at Skillsmatter in London.
Shortly after I started writing, a friend asked me if there really was a need of another Learning Python book.

An excellent question that we could also express in another form: What has this book to offer? What makes this book different from the average introductory book on Python?

I think there are two main differences and many good reasons why you would want to read it.

Firstly, we start with introducing some important programming concepts. We build a solid foundation by covering the critical aspects of this wonderful language.

The pace gradually increases, along with the difficulty of the subjects presented. By the end of Chapter 7, Testing, Profiling, and Dealing with Exceptions, we will cover all the fundamentals.

From Chapter 8, The Edges – GUIs and Scripts, onward, the book takes a steep turn, which brings us to difference number two.

To consolidate the knowledge acquired, there is nothing like working on a small project. So, in the second part of the book, each chapter delivers a project on a different subject. We explore scripting, graphical interfaces, data science, and web programming.

Each project is small enough to fit within a chapter and yet big enough to be relevant. Each chapter is interesting, conveys a message, and teaches something valuable.

After a short section on debugging, the book ends with a complete example that wraps things up. I tried to craft it so that you will be able to expand it in several ways.
So, this is definitely not the usual Learning Python book. Its approach is much more "hands-on" and practical.

I wanted to empower you to help you become a true Python ninja. But I also did my best to entertain you and foster your logical thinking and creativity along the way.

Now, have I answered the question?

What this book covers

*Chapter 1, Introduction and First Steps – Take a Deep Breath*, introduces you to fundamental programming concepts. It guides you to 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, Advanced Concepts – 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, 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 8, The Edges – GUIs and Scripts*, guides you through an example from two different points of view. They are at the extremities of a spectrum: one implementation is a script and the other one a proper graphical user interface application.
Chapter 9, *Data Science*, introduces a few key concepts and a very special tool, the Jupyter Notebook.

Chapter 10, *Web Development Done Right*, introduces the fundamentals of web development and delivers a project using the Django web framework. The example will be based on regular expressions.

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

Chapter 12, *Summing Up – A Complete Example*, presents a Django website that acts as an interface to an underlying slim API written with the Falcon web framework. This chapter takes all the concepts covered in the book to the next level and suggests where to go to dig deeper and take the next steps.
"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. So they need to be told everything. They need to be told 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 as well as 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 on them (kettle is plugged-in, mug is clean, 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 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).

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 on what are the things it has to do. Where does the action happen? In many places: the computer memory, hard drives, network cables, 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 e-mail, 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 height and weight 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 amongst 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 represents 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:

"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, Advanced Concepts – 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 identity 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 creature 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. The 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, I finally had 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, and in general, I very rarely worked with a language that defines scoping with indentation. 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 exposes 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 operating system (OS) specific 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, less skimming through the documentation, and less need for mapping 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 of 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 code is more of a collective effort than a solo experience. Another important aspect of Python is its intrinsic multi-paradigm nature. You can use it as 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, 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 why I would recommend Python to everyone for. 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 the 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. When it comes to number crunching though, one can switch to faster Python implementations, such as PyPy, which provides an average 7-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 is driving the backend of services such as Spotify and Instagram, where performance is a concern. Nonetheless, Python does 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, Pixar, NASA, NSA, Red Hat, Nokia, IBM, Netflix, Yelp, Intel, Cisco, HP, Qualcomm, and JPMorgan Chase, just to name a few.

Even games such as Battlefield 2, Civilization 4, 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.

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 – the great debate

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 on some aspects.

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 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, 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 was hindering 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, 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 (2 x 3) library, which helps introspecting and adapting the behavior according to the version used.

On my Linux box (Ubuntu 14.04), I have the following Python version:

```python
>>> import sys
>>> print(sys.version)
3.4.0 (default, Apr 11 2014, 13:05:11)
[GCC 4.8.2]
```

So you can see that my Python version is 3.4.0. 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 this Python version. Most of them will run also in Python 2 (I have version 2.7.6 installed as well), and those that won’t will just require some minor adjustments to cater for the small incompatibilities between the two versions. Another reason behind this choice is that 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 instruction 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 the way it 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.

If any of the URLs or resources I’ll point you to are no longer there by the time you read this book, just remember: 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 Mac, 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 happens to be the most user friendly OS for Python programmers, Windows on the other hand is the one that requires the biggest effort, Mac being somewhere in between. For this reason, if you can choose, I suggest you to use Linux. If you can't, and you have a Mac, then go for it anyway. If you use Windows, you'll be fine for the examples in this book, but in general working with Python will require you a bit more tweaking.

My OS is Ubuntu 14.04, and this is what I will use throughout the book, along with Python 3.4.0.

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.

Another excellent, resourceful website on Python and its ecosystem is http://docs.python-guide.org.

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 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 of this book, please make sure you are running the console with administrator rights.
On the Mac OS 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 Mac terminal. I will also indicate the command-line prompt with the Linux default format, like this:

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

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 how it should look on Windows 7:
And this is how it should look on Linux:

```
@ @ fab@xps:~

fab@xps:~$ python3
Python 3.4.0 (default, Apr 11 2014, 13:05:11)
[GCC 4.8.2] on linux
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 10, Web Development Done Right*. 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. He 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 use pip though, with the following command:

$ sudo pip install virtualenv # sudo may be optional

pip is a package management system used to install and manage software packages written in Python.

Python 3 has built-in support for virtual environments, but in practice, the external libraries are still the default on production systems. If you have trouble getting virtualenv up and running, please refer to the virtualenv official website: https://virtualenv.pypa.io.
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 a virtualenv, when you want to work with it, is to activate it. Activating a 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 both Linux and Windows. We will:

1. Create a folder named learning.python 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 learning.python folder, we will create a virtual environment called .lpvenv.

   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. This is a very common technique that I use as well. The dot in .venv is because in Linux/Mac prepending a name with a dot makes that file or folder invisible.

3. After creating the virtual environment, we will activate it (this is slightly different between Linux, Mac, and Windows).

4. Then, we’ll make sure that we are running the desired Python version (3.4.*) 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 like on Linux (commands that start with a # are comments):

```bash
$ mkdir learning.python
$ cd learning.python
$ python3 -m venv .lpvenv
$ source .lpvenv/bin/activate
$ python3 -c "import sys; print(sys.version)"
$ deactivate
```
Notice that I had to explicitly tell virtualenv to use the Python 3.4 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 3.4.

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

```
$ virtualenv -p $( which python3.4 ) .lpvenv
```

I preferred 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 its effects stick around when it's done running). 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). In Mac OS, the steps are the same so I won't repeat them here.
Now let's have a look at how we can achieve the same result in Windows. You will probably have to play around a bit, especially if you have a different Windows or Python version than I'm using here. This is all good experience though, so try and think positively at the initial struggle that every coder has to go through in order to get things going.

Here's how it should look on Windows (commands that start with :: are comments):

```
C:\Users\fab\srv>:: step 1 - create folder
C:\Users\fab\srv>mkdir learning.python

C:\Users\fab\srv>cd learning.python

C:\Users\fab\srv\learning.python>:: step 2 - create virtual environment
C:\Users\fab\srv\learning.python>virtualenv -p \Python34\python.exe .lpvenv
Already using interpreter C:\Python34\python.exe
Using base prefix 'C:\\Python34'
New python executable in .lpvenv\Scripts\python.exe
Installing setuptools, pip...done.

C:\Users\fab\srv\learning.python>:: step 3 - activate virtual environment
C:\Users\fab\srv\learning.python>.lpvenv\Scripts\activate
(.lpvenv) C:\Users\fab\srv\learning.python>:: step 4 - verify which python
(.lpvenv) C:\Users\fab\srv\learning.python>where python
C:\Python34\python.exe

(.lpvenv) C:\Users\fab\srv\learning.python>python
Python 3.4.3 (v3.4.3:9b75f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AM
OS4)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> exit()

(.lpvenv) C:\Users\fab\srv\learning.python>:: step 5 - deactivate virtual env
(.lpvenv) C:\Users\fab\srv\learning.python>deactivate
C:\Users\fab\srv\learning.python>
```

Notice there are a few small differences from the Linux version. Apart from the commands to create and navigate the folders, one important difference is how you activate your virtualenv. Also, in Windows there is no `which` command, so we used the `where` command.
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 to 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 it though, allow me to spend 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, you're losing time. Getting things done with the console, counter-intuitively 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 in doubt, 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. For example, I have a laptop that doesn't have a Fn key to toggle the touchpad on and off. I find this very annoying, and I don't want to go clicking about through several menus when I need to do it, so I wrote a small script that is smart enough to tell my system to toggle the touchpad active state, and now I can do it with one simple click from my launcher. Priceless.

We'll devote half of Chapter 8, The Edges – 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 (if you are on Linux):

```
Python 3.4.0 (default, Apr 11 2014, 13:05:11)
[GCC 4.8.2] on linux
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 maths:

>>> 2 + 4
6
>>> 10 / 4
2.5
>>> 2 ** 1024
179769313486231590772930519078902473361797976978942306572734300811577326758
055009631327084772240753602112011387987139335765878976881441662249284743
06394741243377767893424865485276302219601246094119453082952085005768838150
6823424628814739131105408272371633505106845862982399472459384797163048353
56329624224137216

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 IDLE (Integrated DeveLopment Environment). 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 Erik 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 proper software. 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 GUI (Graphical User Interface). There are several frameworks available, some of which are cross-platform and some others are platform-specific. In Chapter 8, The Edges – 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 graphical user interface toolkit that takes desktop application development to a higher level than the conventional approach. It is the standard GUI for Tcl (Tool Command Language), but also for many other dynamic languages and can produce rich native applications that run seamlessly under Windows, Linux, Mac OS 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 in the GUI Programming 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 paragraph, we'll start going down the rabbit hole a little bit more and introduce a bit 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 Mac, which typically hide file extensions to the user, please make sure you change the configuration so that you can see the complete name of the files. This is not strictly a requirement, but a hearty 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. 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 `__init__.py` 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 Linux 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 how a structure of a real simple application could look like:

```
example/
├── core.py
├── run.py
└── util
    ├── __init__.py
    ├── db.py
    ├── math.py
    └── network.py
```
Introduction and First Steps – Take a Deep Breath

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 called by the type 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 much 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
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 very 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 very likely that it will be needed in more than one place. For example in a poll application, where the user is asked many question, 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 DRY (Don't Repeat Yourself) 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 (got the irony?).

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 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 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 which 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 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:

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

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:

```python
>>> 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 had 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 paragraph, 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):

```python
>>> n = 3  # integer number
>>> address = "221b Baker Street, NW1 6XE, London" # S. Holmes
>>> employee = {
...     'age': 45,
...     'role': 'CTO',
...     'SSN': 'AB1234567',
... }
>>> # let's print them
>>> n
3
>>> address
'221b Baker Street, NW1 6XE, London'
>>> employee
```

---

[25]
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 \( \text{address} \) (type: `str`, value: Sherlock Holmes' address)
- A dictionary \( \text{employee} \) (type: `dict`, value: a dictionary which holds three key/value pairs)

Don't worry, I know you're not supposed to know what a dictionary is. We'll see in the next chapter that it's the king of Python data structures.

So, what are \( n \), \( \text{address} \) and \( \text{employee} \)? 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 for free 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:

```python
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`. 
Chapter 1

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 to 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 to alter that during runtime. There are four different scopes that Python makes accessible (not necessarily all of them 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 a 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 continue 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).

The order in which the namespaces are scanned when looking for a name is therefore: **local, enclosing, global, built-in** (LEGBo).
This is all very theoretical, so let's see an example. In order to show you Local and Enclosing namespaces, I will have to define a few functions. Don't worry if you are not familiar with their syntax for the moment, we'll study functions in Chapter 4, Functions, the Building Blocks of Code. Just remember that in the following code, when you see `def`, it means I'm defining a function.

```python
scopes1.py
# Local versus Global

def local():
    m = 7
    print(m)

m = 5
print(m)

# we call, or `execute` the function local
local()
```

In the preceding example, we define the same name `m`, both in the global scope and in the local one (the one defined by the function `local`). When we execute this program with the following command (have you activated your virtualenv?):

```bash
$ python scopes1.py
```

We see two numbers printed on the console: 5 and 7.

What happens is that the Python interpreter parses the file, top to bottom. First, it finds a couple of comment lines, which are skipped, then it parses the definition of the function `local`. When called, this function does two things: it sets up a name to an object representing number 7 and prints it. The Python interpreter keeps going and it finds another name binding. This time the binding happens in the global scope and the value is 5. The next line is a call to the `print` function, which is executed (and so we get the first value printed on the console: 5).

After this, there is a call to the function `local`. At this point, Python executes the function, so at this time, the binding `m = 7` happens and it's printed.

One very important thing to notice is that the part of the code that belongs to the definition of the function `local` is indented by four spaces on the right. Python in fact defines scopes by indenting the code. You walk into a scope by indenting and walk out of it by unindenting. Some coders use two spaces, others three, but the suggested number of spaces to use is four. It's a good measure to maximize readability. We'll talk more about all the conventions you should embrace when writing Python code later.
What would happen if we removed that $m = 7$ line? Remember the LEGB rule. Python would start looking for $m$ in the local scope (function `local`), and, not finding it, it would go to the next enclosing scope. The next one in this case is the global one because there is no enclosing function wrapped around `local`. Therefore, we would see two number $5$ printed on the console. Let’s actually see how the code would look like:

```python
count = 0

while count < 10:
    count += 1
    print(count)
```

Running `scopes2.py` will print this:

```
(1pvenv) fab@xps:ch1$ python scopes2.py
5 printing from the global scope
5 printing from the local scope
```

As expected, Python prints $m$ the first time, then when the function `local` is called, $m$ isn’t found in its scope, so Python looks for it following the LEGB chain until $m$ is found in the global scope.

Let’s see an example with an extra layer, the enclosing scope:

```python
count = 0

while count < 10:
    count += 1
    print(count)
```

Running `scopes2.py` will print this:

```
(1pvenv) fab@xps:ch1$ python scopes2.py
5 printing from the global scope
5 printing from the local scope
```

As expected, Python prints $m$ the first time, then when the function `local` is called, $m$ isn’t found in its scope, so Python looks for it following the LEGB chain until $m$ is found in the global scope.

Let’s see an example with an extra layer, the enclosing scope:

```python
count = 0

while count < 10:
    count += 1
    print(count)
```

Running `scopes2.py` will print this:

```
(1pvenv) fab@xps:ch1$ python scopes2.py
5 printing from the global scope
5 printing from the local scope
```

As expected, Python prints $m$ the first time, then when the function `local` is called, $m$ isn’t found in its scope, so Python looks for it following the LEGB chain until $m$ is found in the global scope.
m = 5
print(m, 'printing from the global scope')

enclosing_func()

Running scopes3.py will print on the console:

(.lpvenv)fab@xps:ch1$ python scopes3.py
5 printing from the global scope
13 printing from the local scope

As you can see, the print instruction from the function local is referring to m as before. m is still not defined within the function itself, so Python starts walking scopes following the LEGB order. This time m is found in the enclosing scope.

Don't worry if this is still not perfectly clear for now. It will come to you as we go through the examples in the book. The Classes section of the Python tutorial (official documentation) has an interesting paragraph about scopes and namespaces. Make sure you read it at some point if you wish for a deeper understanding of the subject.

Before we finish off this chapter, I would like to talk a bit more about objects. After all, basically everything in Python is an object, so I think they deserve a bit more attention.

Object and classes

When I introduced objects in the A proper introduction section, I said that we use them to represent real-life objects. For example, we sell goods of any kind on the Web nowadays and we need to be able to handle, store, and represent them properly. But objects are actually so much more than that. Most of what you will ever do, in Python, has to do with manipulating objects.

So, without going too much into detail (we'll do that in Chapter 6, Advanced Concepts – OOP, Decorators, and Iterators), I want to give you the in a nutshell kind of explanation about classes and objects.
We’ve already seen that objects are Python’s abstraction for data. In fact, everything in Python is an object. Numbers, strings (data structures that hold text), containers, collections, even functions. You can think of them as if they were boxes with at least three features: an ID (unique), a type, and a value.

But how do they come to life? How do we create them? How to we write our own custom objects? The answer lies in one simple word: classes.

Objects are, in fact, instances of classes. The beauty of Python is that classes are objects themselves, but let’s not go down this road. It leads to one of the most advanced concepts of this language: metaclasses. We’ll talk very briefly about them in Chapter 6, Advanced Concepts – OOP, Decorators, and Iterators. For now, the best way for you to get the difference between classes and objects, is by means of an example.

Say a friend tells you "I bought a new bike!" You immediately understand what she’s talking about. Have you seen the bike? No. Do you know what color it is? Nope. The brand? Nope. Do you know anything about it? Nope. But at the same time, you know everything you need in order to understand what your friend meant when she told you she bought a new bike. You know that a bike has two wheels attached to a frame, a saddle, pedals, handlebars, brakes, and so on. In other words, even if you haven’t seen the bike itself, you know the concept of bike. An abstract set of features and characteristics that together form something called bike.

In computer programming, that is called a class. It’s that simple. Classes are used to create objects. In fact, objects are said to be instances of classes.

In other words, we all know what a bike is, we know the class. But then I have my own bike, which is an instance of the class bike. And my bike is an object with its own characteristics and methods. You have your own bike. Same class, but different instance. Every bike ever created in the world is an instance of the bike class.

Let’s see an example. We will write a class that defines a bike and then we’ll create two bikes, one red and one blue. I’ll keep the code very simple, but don’t fret if you don’t understand everything about it; all you need to care about at this moment is to understand the difference between class and object (or instance of a class):

```python
bike.py
# let's define the class Bike

class Bike:
    def __init__(self, colour, frame_material):
        self.colour = colour
        self.frame_material = frame_material

    def brake(self):
        print("Braking!")
```
Introduction and First Steps – Take a Deep Breath

```python
# let's create a couple of instances
red_bike = Bike('Red', 'Carbon fiber')
blue_bike = Bike('Blue', 'Steel')

# let's inspect the objects we have, instances of the Bike class.
print(red_bike.colour)  # prints: Red
print(red_bike.frame_material)  # prints: Carbon fiber
print(blue_bike.colour)  # prints: Blue
print(blue_bike.frame_material)  # prints: Steel

# let's brake!
red_bike.brake()  # prints: Braking!
```

I hope by now I don't need to tell you to run the file every time, right? The filename is indicated in the first line of the code block. Just run $ python filename, and you'll be fine.

So many interesting things to notice here. First things first; the definition of a class happens with the `class` statement (highlighted in the code). Whatever code comes after the `class` statement, and is indented, is called the body of the class. In our case, the last line that belongs to the class definition is the `print("Braking!")` one.

After having defined the class we're ready to create instances. You can see that the class body hosts the definition of two methods. A method is basically (and simplistically) a function that belongs to a class.

The first method, `__init__` is an **initializer**. It uses some Python magic to set up the objects with the values we pass when we create it.

Every method that has leading and trailing double underscore, in Python, is called **magic method**. Magic methods are used by Python for a multitude of different purposes, hence it's never a good idea to name a custom method using two leading and trailing underscores. This naming convention is best left to Python.

The other method we defined, `brake`, is just an example of an additional method that we could call if we wanted to brake the bike. It contains just a `print` statement, of course, it's an example.

We created two bikes then. One has red color and a carbon fiber frame, and the other one has blue color and steel frame. We pass those values upon creation. After creation, we print out the color property and frame type of the red bike, and the frame type of the blue one just as an example. We also call the `brake` method of the `red_bike`.
One last thing to notice. You remember I told you that the set of attributes of an object is considered to be a namespace? I hope it's clearer now, what I meant. You see that by getting to the `frame_type` property through different namespaces (`red_bike`, `blue_bike`) we obtain different values. No overlapping, no confusion.

The dot (.) operator is of course the means we use to walk into a namespace, in the case of objects as well.

**Guidelines on how to write good code**

Writing good code is not as easy as it seems. As I already said before, good code exposes a long list of qualities that is quite hard to put together. Writing good code is, to some extent, an art. Regardless of where on the path you will be happy to settle, there is something that you can embrace which will make your code instantly better: **PEP8**.

According to Wikipedia:

"Python's development is conducted largely through the Python Enhancement Proposal (PEP) process. The PEP process is the primary mechanism for proposing major new features, for collecting community input on an issue, and for documenting the design decisions that have gone into Python."

Among all the PEPs, probably the most famous one is PEP8. It lays out a simple but effective set of guidelines to define Python aesthetic so that we write beautiful Python code. If you take one suggestion out of this chapter, please let it be this: use it. Embrace it. You will thank me later.

Coding today is no longer a check-in/check-out business. Rather, it's more of a social effort. Several developers collaborate to a piece of code through tools like git and mercurial, and the result is code that is fathered by many different hands.

Git and Mercurial are probably the most used distributed revision control systems today. They are essential tools designed to help teams of developers collaborate on the same software.

These days, more than ever, we need to have a consistent way of writing code, so that readability is maximized. When all developers of a company abide with PEP8, it's not uncommon for any of them landing on a piece of code to think they wrote it themselves. It actually happens to me all the time (I always forget the code I write).
This has a tremendous advantage: when you read code that you could have written
yourself, you read it easily. Without a convention, every coder would structure the
code the way they like most, or simply the way they were taught or are used to, and
this would mean having to interpret every line according to someone else's style. It
would mean having to lose much more time just trying to understand it. Thanks to
PEP8, we can avoid this. I'm such a fan of it that I won't sign off a code review if the
code doesn't respect it. So please take the time to study it, it's very important.

In the examples of this book, I will try to respect it as much as I can. Unfortunately,
I don't have the luxury of 79 characters (which is the maximum line length suggested
by PEP8), and I will have to cut down on blank lines and other things, but I promise
you I'll try to layout my code so that it's as readable as possible.

The Python culture
Python has been adopted widely in all coding industries. It's used by many different
companies for many different purposes, and it's also used in education (it's an
good language for that purpose, because of its many qualities and the fact that it's
easy to learn).

One of the reasons Python is so popular today is that the community around it is
vast, vibrant, and full of brilliant people. Many events are organized all over the
world, mostly either around Python or its main web framework, Django.

Python is open, and very often so are the minds of those who embrace it. Check out
the community page on the Python website for more information and get involved!

There is another aspect to Python which revolves around the notion of being
Pythonic. It has to do with the fact that Python allows you to use some idioms that
aren't found elsewhere, at least not in the same form or easiness of use (I feel quite
claustrophobic when I have to code in a language which is not Python now).

Anyway, over the years, this concept of being Pythonic has emerged and, the way I
understand it, is something along the lines of doing things the way they are supposed to
be done in Python.

To help you understand a little bit more about Python's culture and about being
Pythonic, I will show you the Zen of Python. A lovely Easter egg that is very popular.
Open up a Python console and type import this. What follows is the result of
this line:

```python
>>> import this
The Zen of Python, by Tim Peters
```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren’t special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one— and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you’re Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it’s a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let’s do more of those!

There are two levels of reading here. One is to consider it as a set of guidelines that have been put down in a fun way. The other one is to keep it in mind, and maybe read it once in a while, trying to understand how it refers to something deeper. Some Python characteristics that you will have to understand deeply in order to write Python the way it’s supposed to be written. Start with the fun level, and then dig deeper. Always dig deeper.

**A note on the IDEs**

Just a few words about Integrated Development Environments (IDEs). To follow the examples in this book you don't need one, any text editor will do fine. If you want to have more advanced features such as syntax coloring and auto completion, you will have to fetch yourself an IDE. You can find a comprehensive list of open source IDEs (just Google "python ides") on the Python website. I personally use Sublime Text editor. It's free to try out and it costs just a few dollars. I have tried many IDEs in my life, but this is the one that makes me most productive.
Two extremely important pieces of advice:

- Whatever IDE you will chose to use, try to learn it well so that you can exploit its strengths, but don’t depend on it. Exercise yourself to work with VIM (or any other text editor) once in a while, learn to be able to do some work on any platform, with any set of tools.

- Whatever text editor/IDE you will use, when it comes to writing Python, indentation is four spaces. Don't use tabs, don't mix them with spaces. Use four spaces, not two, not three, not five. Just use four. The whole world works like that, and you don't want to become an outcast because you were fond of the three-space layout.

**Summary**

In this chapter, we started to explore the world of programming and that of Python. We've barely scratched the surface, just a little, touching concepts that will be discussed later on in the book in greater detail.

We talked about Python's main features, who is using it and for what, and what are the different ways in which we can write a Python program.

In the last part of the chapter, we flew over the fundamental notions of namespace, scope, class, and object. We also saw how Python code can be organized using modules and packages.

On a practical level, we learned how to install Python on our system, how to make sure we have the tools we need, pip and virtualenv, and we also created and activated our first virtual environment. This will allow us to work in a self-contained environment without the risk of compromising the Python system installation.

Now you're ready to start this journey with me. All you need is enthusiasm, an activated virtual environment, this book, your fingers, and some coffee.

Try to follow the examples, I'll keep them simple and short. If you put them under your fingertips, you will retain them much better than if you just read them.

In the next chapter, we will explore Python's rich set of built-in data types. There's much to cover and much to learn!
Where to buy this book
You can buy Learning Python from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.