Python Chases Monkey

An Introduction to Python Programming

Jason M. Pittman

Chapter 1

Getting Started

1.1 Introduction

Be forewarned, this will be the most *talkative* or read-intensive chapter in the book. There are some ground truths that mandate our attention and understanding. Take out a pen with some paper and jot down what catches your interest, what is familiar based on a parallel concept you know, and definitely what makes no sense (yet).

Right then, I think it is fair to say Python is a powerful language with a great many use cases. The barrier to entry is low but the language has a great deal of complexity if we allow it to bloom. I would argue the language exemplifies the best aspects of high level programming concepts while minimizing the drawbacks.

The consensus across industries and academies is Python is fast to pickup and powerful for prototyping. I agree. I've personally seen Python used in production software where portability, clarity, and scalability are important features of the codebase. Further, most critically for our endeavor, I would add that Python is great for learning because the language and tools don't get in our way. That is, we can immediately get started once we have our toolset ready.

1.2 Using this book

Before we dive into our learning objectives for this chapter, I want to point out a few things about how I think you should use this book.

Foremost, understand that the examples are just that: examples. The code I provide throughout is functional but not always complete. There is value in learning to read and analyze existing code. There is also quite a bit of value in imagining how you might finish the code examples when these are incomplete. That's where the chapter **Questions** come into play. You'll see.

I want to make a suggestion for us to seriously consider. I the value of copying examples character by character, statement by statement, is extremely low. I think there's likely even less value, probably zero, in performing a broad copy and paste. In contrast, I think you can learn a lot by trying something *slightly* different than what the example indicates. I say *slightly* because you don't want to introduce a difference so large that you cannot trace the output back to the change. I attempt to guide such efforts in the chapter **Exercises** sections.

Furthermore, I firmly think there is tremendous benefit in *emulating* an example. This is where the chapter **Projects** come into the picture. The intention for **Projects** is that you will develop a program to match the indicated technical requirements. While not identical to the chapter examples, certainly there are lessons to be extracted from the examples and exercises which will make developing the project code easier.

The last thing I'll offer you before we get started is a suggested order of operations for using this book. In fact, we can render the suggestion as an ordinal list:

1. Read through the chapter without paying attention to the example code.
2. Read the example code by itself.
3. Read through the exercises and make some notes as to what you think the outcome(s) may be.
4. Read back through the chapter and examples.
5. Complete the exercises.
6. Tackle the project.

With all that said, let's get into learning Python programming.

1.3 Learning objectives

- Summarize how to obtain, install, and verify a Python implementation
- Discover two ways to interact with Python
- Describe the underlying mechanics associated with running a Python program
- Create and execute a simple Python program

1.4 Getting started with Python

The current version of Python is 3.9 as of this writing. However, as long as you're running a version greater than or equal to 3.6 I don't think you'll run into any issues developing the programs in this text. Whatever you do, don't try to run any version of Python 2 as there are major feature differences.

With that said, there are two ways to get started with Python. First, if you are using a GNU/Linux operating system (e.g. Ubuntu), you already have Python. Otherwise, you are going to want to download and install the official stable version given your operating system.

For Windows, I would suggest installing **Windows Subsystem for Linux** with Ubuntu as the distribution. The experience will be that of authentic GNU/Linux. Further, doing so will give you at least Python 3.6.9. Alternativley, you could use the official Python release for Windows as found here: `https://www.python.org/downloads/windows/`

For Mac, you need to exercise caution. Python 2 is included by default. My recomendation would be to use the official Python 3 distribution for Mac which can be found here: `https://www.python.org/downloads/mac-osx/`

In all of the above cases, you can verify the implementation by launching a shell, terminal, or command prompt and typing `python3 --version`.

Interacting with Python

While we have that shell, terminal, or command prompt open, let's go ahead and learn how to interact with our Python implementation directly. This is great for a quick test, brief calculation, or experimenting with a module. Type `python3` and press `Enter`.

We should receive several lines of output which includes the version number along with a few suggestions on what to enter next. We should also see the prompt has changed to `>>>`.

Now, this wouldn't be a programming textbook if we didn't have a *hello world* example. This is the perfect place for it and I see no good reason to deviate from tradition. So, still in your shell or terminal with the Python interpreter running, type the following and press `Enter`:

Example 1

```
(1) print("Hello, world!")
```

That's awesome- our first program! Let's unpack this real quick.

Foremost, this is the formatting for all of the examples throughout this book. I'm a huge fan of line numbers because it makes referencing specific statements much more efficient, hence the `(1)`.

Furthermore, programmatically we (a) successfully called a **function**; (b) executed a statement; and (c) displayed output. The function is the `print()` and we executed a full statement by passing `print()` a string *type* parameter `"Hello, world!"` and pressing `Enter`.

Also, we may have learned what we don't know. More specifically, perhaps we don't know or understand why the `print()` function requires a string type as a parameter or even what a **function** is for that matter. That's good because we can take a note and either look for a definition or ask someone more experienced. If we go with the latter, I

always suggest a question format of, *here's what I think, where am I mistaken?*. Your mileage may vary.

Lastly, you may be wondering how you exit the interactive interpreter... try `exit()` or `quit()`.

Python mechanics

Python, in contrast to `C` or `C++`, is an interpreted language. This is why we have to install the language instead of just ensuring we have a compiler available. I believe the difference is material which obligates me to a short explanation. As well, there is a key similarity to languages such as `C#` and `Java` that also bears mentioning.

To begin, mechanically we write all of our program code in plaintext using the symbols and syntax defined by the Python language reference. You can view the reference for Python 3 here. Then, we run the interpreter and pass our source as an argument. Compare that to compilation where we run a program (technically, two because of linking and compiling) against the source to produce a new program.

For comparison, Python is similar to `Java` or `C#` in that the interpreter renders our source down to bytecode. Bytecode is the `.pyc` files you'll see in your working directory. These bytecode files, along with the interpreter, are why Python code is portable with little concern for anything beyond the interpreter. We can use that to our advantage. To do so, we need to build up our understanding of what goes into making a Python program.

Anatomy of a simple Python program

Example 2

```
(1) #!/usr/bin/env python3
(2)
(3) print("What is the sum of 2 and 3?")
(4) sum = 2 + 2
(5) print(sum)
```

The strangest part of our simple example program might be line one if you are unfamiliar with interpreted programming languages. This is called a **shebang** and is what the program loader reads to call out to the proper runtime binary. The program loader acts as an intermediary and is not something we control. That's okay, less to think about.

The shebang is something to think about because we control our runtime environment through it. If we consider the shebang to be an automated way to call the indicated program, the Python 3 interpreter in this case, it should make more sense.

The loader then passes the location of our source file to the interpreter we indicated in the shebang. Yes, that's circular. Thankfully, as long as the binary we write in the

shebang exists, the rest is invisible to us. That gives us a clue as to what advantage the shebang system lends us.

Alternatively, we can omit the shebang and pass our source file to the Python interpreter directly. For example: `python3 simple.py`

Now, what might confuse you is the inclusion of the shebang statement while running the interpreter directly. There's nothing wrong strictly speaking with including a shebang and running the interpreter. We're effectively doubling down on running a specific interpreter binary.

For what it is worth, I say always include a shebang. Worse case, you're telling the user what interpreter binary you developed against.

Style and standards

There is a definitive *pythonic* way of doing things, programmatically speaking. There are a handful of style standards that functionally matter (we'll get errors if we don't follow them) and a host that conceptually matter.

Two functional standards you will need to get up to speed fast on are *indents* and *colons*. Whereas other languages (pretty much all of them) use curly braces `{ }` to fence or scope blocks of code, Python uses simple identation in conjunction with the colon `:`. For instance:

Example 3

```
(1) if python is great:
(2)     print("Python is great!")
```

How clean is that code my friends? Pretty clean if I do say so and I do say so. The first line is our **conditional** followed by line two which is a familiar function call statement. We'll examine **conditionals** in chapter 6. For now, focus on the colon `:` followed by the four space indentation.

However, there is a tremendous amount of danger lurking here for us when we are first getting started. Look at the next example.

Example 4

```
(1) if python is great:
(2)     print("Python is great!")
(3)
(4) if python is not great:
(5)   print("Python is not great?")
```

Do you see the difference between lines two in **Example 3** and five in **Example 4**? We should create two programs mirroring these examples and apply a bit of ye olde scientific method. In other words, let's do the experiment and see what happens.

Meanwhile, conceptual stuff includes principles such as variable naming, using comments versus self-describing code, what to name files, and so forth. Here, what matters most is for us to pick a style, pick a standard, and stick with them. Don't change between projects and for the love of everything holy don't change within a project. My suggestion is to defer to what makes your code more readable and eases future maintenance.

Third and last, read and write a lot of code. The more the better and I definitely recommend at least thirty minutes a day at a minimum. As much as possible, I've attempted to organize this book with that in mind. Speaking of which, look at what is next! That's right, chapter **Exercises** and **Questions**.

Exercises

1. Develop a Python program that asks the user to input two numbers, n_1 and n_2. Assign the input to two different variables. Compute and output the following:

- Sum
- Difference
- Product
- Quotient
- Modulus

2. Modify your program from excercise one above to output each value in two forms: **int()** and **float()**.
3. Using the interactive Python interpreter, compute the average time per mile if we completed a 10k race (6.2 miles) in 68 minutes 40 seconds.

Questions

1. Are `exit()` and `quit()` functions: yes or no?
2. What do you think would happen if we remove line 1 from **Example 2** and attempt to run it by just typing the file name?
3. What exactly would you type in an interactive Python interpreter session, if you wanted to run the same calculation and print the output from **Example 2**?
4. What happens when we run **Example 4** exactly as indicated?
5. How would fix **Example 4** so that it can run (assume we have the value of the `python` variable set elsewhere)?