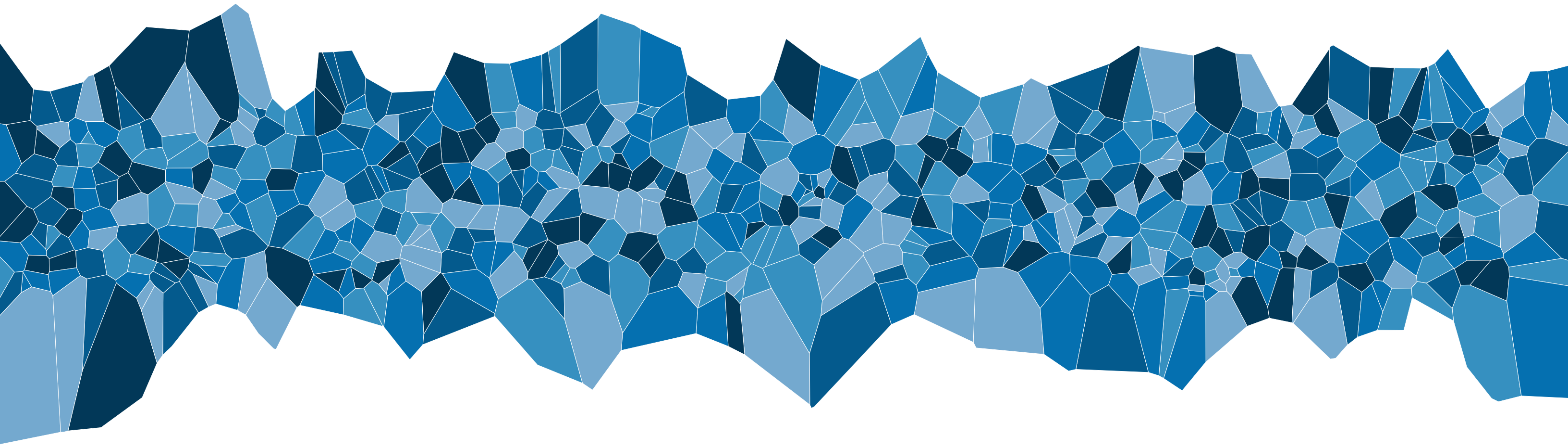
Data Science for Crime Analysis with Python

Andrew P. Wheeler



Data Science for Crime Analysis with Python

Andrew P. Wheeler

2024-06-23

Data Science for Crime Analysis with Python

ISBN 979-8-9903770-0-4 (ebook), ISBN 979-8-9903770-1-1 (paperback)

You can see my other work at https://crimede-coder.com/

CRIME
De-Coder

Table of Contents

Preface

Python is an open-source, free, computer programming language. It allows you to write simple (or complex) programs, to allow your computer to accomplish tasks. For a brief example, here is a python code snippet that tells you the difference in the number of days between two dates. The lines that start with # are comments in python, the other lines perform different operations in python code. The text in the grey portion below is the python code, and the blue section text is the output of the program.

```python
# importing library to calculate times
from datetime import datetime

# creating two datetime objects
begin = datetime(2022,1,16)
end = datetime(2023,1,16)

# calculating the difference
dif = end - begin

# printing the result
print(dif.days)
```

```
365
```

So this is a trivial program – you could figure out the number of days between the two dates using various tools. The power of being able to program in python is that you can write computer code to do (close to) any computation you want. A common example for a crime analyst may be querying a database and creating a table of crime counts year-to-date this year vs last. You can then run the code at will, and it will update the year-to-date stats on whatever frequency you want. Such a report in practice will just be chaining together short code examples like above into more complicated series of operations.

Who is this book for?

This book is aimed at individuals with no (or beginner) background in programming, but who are interested in using code to conduct quantitative analysis and automate tasks. The main intended audience of the book are crime analysts, but any individual looking to get started in coding and data analysis should find the content of the book useful. Besides crime analysts, those wishing to advance their career in a data science role or pursue graduate level research (who have a background in criminal justice), will find the book and its examples useful.

There are many current resources on using python on the internet – one can use a search engine to find various entirely free resources online. I commonly blog on technical computing at andrewpwheeler.com, which is free for anyone to read. These free resources are often haphazard though, and are very difficult for newcomers to understand and get started. Things like "How do I run a simple python script" or "How do I install a python library" are not typical topics covered in even introductory python materials online. This book is intended to make a singular resource for individuals in crime analysis to get started.

I intend this book to not only introduce code examples in python, but to also describe other necessary steps for beginners, such as setting up python environments and automating tasks using shell scripts. Don't worry if you do not understand what those are at the moment – they will be explained! I even spend time describing a typical project structure that is fairly standard across more professional software development. These are things not related to coding directly, but are necessary to be able to get started using python and use it effectively.

So this book fills a niche – an introduction to doing tasks in python of relevance to crime analysts. The book subsequently contains:

- installing and creating python environments
- an introduction to python programming
- working with tabular data using scientific libraries
- using SQL to query databases
- automating report creation and creating high quality tables and graphs

These are the necessary ingredients, both in terms of coding and doing more realistic projects, that allow one to become more productive in their regular tasks using python.

What this book is not

When approaching learning to write code and conduct data analysis, many books include *both* at the same time. This is often a mistake, as it can greatly increase the burden on those wishing to learn the material. This book *is not* meant as an introduction to crime analysis as a topic in general. For those who wish to learn basic statistics and analyses that crime analysts conduct, I would suggest to check out the course materials on my personal website, as well as the materials from the International Association of Crime Analysts (IACA). If demand is sufficient, I may create future books to cover intro statistics for crime analysts more thoroughly, so let me know if that is something you are interested in!

This book is aimed to get you started writing code and applying it to real tasks crime analysts need to conduct. I use realistic examples that a crime analyst may be interested in conducting, such as sending automated emails, making year-to-date tables, and creating line charts. But I do not discuss in detail things like the Poisson distribution for analyzing crime rates or why hotspot analyses is important.

While the material is definitely relevant to *everyone* who needs to conduct data analysis using python, I hope the use of more realistic crime analysis examples helps to better illustrate the utility of using python to conduct analyses in your day-to-day tasks as a crime analyst.

Why learn to code?

Crime analysts do much of their quantitative work in spreadsheets (e.g. Excel), and then a smaller number use additional tools, such as databases (e.g. Access, SQLServer), formatted documents (Word, Powerpoint, PDF), and GIS tools (such as ESRIs ArcMap). Why bother to learn python? I agree that Excel can be used to do amazing things with data, and many tasks are *exchangeable* between python and one (if not several) different tools.

The power of using programming, as opposed to tools that use a graphical user interface (e.g. point and click in the *GUI*), are:

- tasks can be fully automated
- tasks are fully documented

The first bullet point is an argument based on potential time savings for automating tasks. Say it takes you 30 minutes to do a task on a daily basis. If you take 100 hours to write python code to automate the task entirely, you have saved yourself time within 50 days from the automated process.

Many regular reports that crime analysts work on this time savings argument may not be compelling though. For example, when I worked as an analyst I had a monthly CompStat report with various graphs and maps. Using GUI tools it maybe took me 24 hours (three working days) to complete. Once I wrote code to automatically create the graphs, it was under a day task. But I maybe spent over 160 hours writing code to automate that task. It would take over a year to break even in terms of time savings.

Many regular reports crime analysts write will look like the latter; they will only be semi-regular, and so the time savings argument for automating via code is not as impressive. (Code automation makes more sense for time savings for things that need to be done more often.) Even in those cases of semi-regular reports though, I believe writing code to automate as much as possible is still worth it.

This is because of the second bullet point – tasks when written in code are by their nature fully documented. This provides the ability for an analyst to retrospectively say things like “this number looks weird, how did I calculate that?”, or when a new analyst comes in and takes over the job, you can say “just go look at the scripts in folder X”. Having standardized code provides a much more professional and transparent environment, which is helpful for you as an analyst as well as the organization as a whole.

It additionally allows you to scale your work. If you need to loan out your time forever for a particular task, even if only a single day per month for a particular report, you can only expand the scope of the work you do a certain amount. Being able to automate the boring stuff is a necessary step to free up your time to pursue other projects. It even allows you to go on vacation, and reporting requirements can still be fulfilled. Ultimately learning to code will likely make you more productive when conducting ad-hoc data analysis, as well as make you more marketable in a wider array of jobs (such as private sector data science jobs).

Why python?

The section above only describes why one would want to write code to automate tasks, it does not detail why to use python specifically (over say R or another statistical program). In addition to python, I have used SPSS (a program you need to pay for), and R (another open source statistical program) fairly extensively over my career. I have an R package, ptools, for regular functions of interest to crime analysts for example.

I have almost entirely migrated my personal coding to python, and do not use these other tools very often anymore. Again, python is very exchangeable with R for many tasks, but I prefer python at this point in my career due to its ability to manage entire projects, not just do a single task. In addition to this, many private sector data science positions focus almost entirely on python (and less so on R). So I believe in terms of professional development, especially if you have a goal for expanding your skills to pursue private sector data science positions, python is a better choice than R.

There are situations when paid for tools are appropriate as well. Statistical programs like SPSS and SAS do not store their entire dataset in memory, so can be very convenient for some large data tasks. ESRI's GIS (*Geographic Information System*) tools can be more convenient for specific mapping tasks (such as calculating network distances or geocoding) than many of the open source solutions. (And ESRI's tools you can automate by using python code as well, so it is not mutually exclusive.) But that being said, I can leverage python for nearly 100% of my day to day tasks. This is especially important for public sector crime analysts, as you may not have a budget to purchase closed source programs. Python is 100% free and open source.

How to read this book

I believe the optimal way to consume the material in this book is via a two step process. Your level of experience with python (either some or zero), will alter what materials you focus on and what ones you can likely skip. For everyone, I would suggest *skimming* each chapter briefly from the start, and understanding the high level goals each chapter is trying to teach.

This is how I personally consume technical material. You need to understand the high level goals that any particular piece of code is trying to accomplish before you can understand the finer technical details. If you cannot understand the high level goals, it will be very difficult to understand the technical details. It is also useful to understand what is possible – you do not need to point and click in excel to regenerate that CompStat report every month, you can write code to automate that (see Chapter 10).

The second part, after the skimming, is where it depends on whether you are neophyte to python, or whether you have some background experience in programming. For those neophytes with no experience, I would suggest you study in detail the entry chapters 1 through 4 in the book. A large problem in learning to run code is the "getting started running a simple example" problem – downloading a program and running commands is challenging for those who have never done it before.

This part – figuring out how to install python and run a simple command can be the most challenging hurdle to get started. Part of the challenge, as the author, is that everyone's systems is slightly different and changing over time. Instructions to get started tend to be idiosyncratic to your personal computer. Part of the reason I am writing this book is that most beginner materials do not even try to discuss this issue, and use tricks (such as using online platforms) to help people get started.

To accomplish real tasks crime analysts need for their jobs though, you cannot use the online platforms. Many individuals who are taught python in university courses use said online platforms (e.g. if you only have experience using Jupyter notebooks or only experience with Google Collab notebooks). You need to know how to download python and run it locally on your personal computer to be able to use it for work related tasks. But do not despair if you are having problems getting started! One technique professional software engineers use is called pair-programming – grab a friend who knows how to run python code (which can be me, or someone else in your network), look over their shoulder, and then have them look over your shoulder. This will help you get started in Chapter 1.

Chapters 2 through 4 introduce basic objects (strings, numbers, lists, dictionaries), and actions (conditional statements, looping, string substitution). These are very boring python basics – similar to how learning the normal distribution is boring in your introductory stats class, or learning algebra is boring in mathematics. They are the necessary building blocks though for understanding how to effectively write python code. Those with more entry level experience may feel comfortable skimming chapters 2-4; I would suggest to examine them in a cursory fashion at least though – there are likely a few things you did not know that will be introduced.

Chapter 5 is a section that even those with introductory experience often are not exposed. Writing your own functions and understanding how to import those functions are an important step from writing hobby code to creating a professional environment to develop work projects over time. Again, many individuals who have had a course in college on python programming are not exposed to this.

Chapters 6 through 9 are oriented around showing specific examples of working and presenting data that will be of wide interest, not only to those in crime analysis but those in any data oriented role. Chapter 6 shows the two main libraries to work with tabular data – `numpy` and `pandas`. Understanding the pandas library in particular is an important skill for those using python to conduct data analysis.

Chapter 7 shows how to use python to generate SQL queries. For those who are not familiar with SQL, *Structured Query Language*, SQL is used to pull data from an external database and into a pandas dataframe. This chapter I will also introduce different SQL statements, as in some scenarios it is better to do certain data analysis tasks in the database *before* you load the data into an in-memory pandas dataframe.

Chapter 8 introduces the `matplotlib` graphing library in python. Creating professional looking graphics is an important skill for data analysts. Generating high quality graphs is a signal to consumers of the quality of the work (for crime analysts these might be police officers, command staff, or the general public). Generating such graphs via code in python is a good way to control the look and consistency of graphs you produce.

Chapter 9 introduces Jupyter notebooks – notebooks offer a different environment than the terminal to run code. Jupyter notebooks can intermix plain text description, executable cells for code, and the output from those code executions (e.g. graphs and tables). This book, under the hood, is compiled from a series of Jupyter notebooks. I introduce Jupyter, as it is a convenient way to create standardized reports that contain different elements of data analysis.

The final chapter 10, *project organization*, discusses aspects of project management and workflow automation – the final necessary components to be able to take simple projects and really leverage python to help you do your job as a crime analyst. Now that you know how to write code, what does a project look like? There are standard ways you should organize your project, so when either you need to re-run the code, or others need to, they can understand the necessary components. This involves things like creating a README that has information to replicate the necessary environment to run the code, having functions documented and stored in a specific location, and a clear entry point that runs the code in an automated fashion.

The overall contents of the book are intended to go beyond “how to write python code”, to giving individuals the end-to-end experience of creating realistic projects that can help crime analysts do their job. This involves more than just running a single script, but knowing how professionals do things like query a database, create reusable functions, and set up projects to automate different data analysis tasks over time.

These not writing code portions are what is severely lacking in current python programming how-tos, and is the main motivation to write the book.

My background

While getting my doctorate degree in criminal justice at SUNY Albany (between 2008 and 2015), I worked in several analyst roles. First, at the Division of Criminal Justice Services for New York state. That job mainly involved writing standardized reports based on the New York states criminal arrest history database. Then for several years I worked in-house at the Troy, NY police department as their lone crime analyst. Finally, I worked as a research analyst at the Finn Institute for Public Safety, a non-profit who collaborated in research projects with police departments in upstate New York.

I then was a professor of criminology for several years at the University of Texas at Dallas, from 2016 through 2019. During that time I wrote around 40 peer reviewed publications and collaborated on quantitative projects with police departments across the United States. I regularly presented this work at the IACA conference, and for a brief period was the head of the publications committee for IACA.

Currently (since late 2019), I have worked as a data scientist (in the private sector) for a healthcare company. My job now is to write software, focused on using predictive models in relation to healthcare claims data. While healthcare may seem quite different than crime analysis, many of the problems are fundamentally the same (working with health insurance claims is not all that different than working with crime reports). Examples of things I have built at my current private sector job are predictive models to identify when claims are overpaid, or when individuals are at high risk of a follow up heart attack.

The skills to build those predictive models are no different from work I have done forecasting crime in different areas, or identifying chronic offenders at high risk of committing future violence. So my personal experience as a crime analyst, a researcher, and then a private sector data scientist are what motivated me to write this book. I want to see some of the more advanced work in academia, as well as the software engineering practices in the private sector, trickle down more broadly into the crime analysis profession. An introductory book I believe is the best method to accomplish that goal.

Feedback on the book

For feedback on the contents of the book, you can send a message to me at https://crimede-coder.com/contact. Always feel free to send me feedback, suggest additional topics, or let me know of errors. If you are interested in more direct services, such as in person training for your analysts or direct consulting on projects you are working on, feel free to email me as well. Examples of past work I have conducted for various criminal justice agencies are program evaluation, redistricting, automating different processes, civil litigation consulting, and generating predictive models.

I have future plans to generate more advanced python content. These include books on:

- more advanced programming, data, and package management
- regression modeling
- machine learning applications in crime analysis
- GIS analysis with python

Feedback on content and letting me know what you are interested in helps me prioritize on this future work. And more sales of this individual book also give me motivation to write more – so tell your friends if you like it.

Thank you

Thanks are in order for several friends for reviewing early drafts of the book and providing critical feedback. Renee Mitchell gave me early feedback on drafts, and was one of the major impetuses to start this idea and finish it. Dae-Young Kim has been diligent in giving me very detailed feedback on each chapter, most importantly when I need to explain the code in more detail or my narrative is not consistent with the code examples. And I thank my wife for early feedback on the beginning chapters writing python code. (Easily the hardest part of coding is understanding how to run the code, not writing the code itself. A good test to see if your documentation is sufficient is to have someone who is *not* a programmer see if they can understand.)

Thank you all for the support, and for all of those who purchased pre-versions of the book.

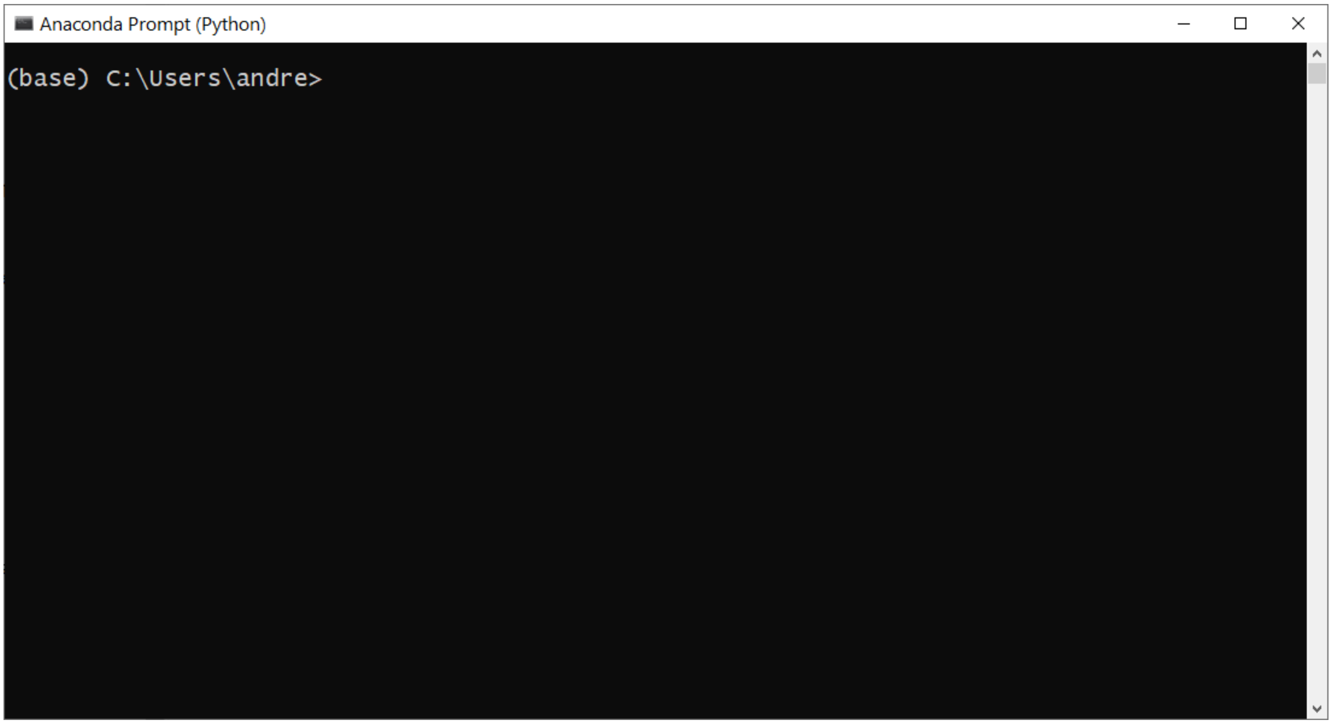
This book is only possible based on various open source contributions. I am using the Quarto engine to render this book to different environments (PDF and EPUB), which itself uses $\LaTeX$ and Pandoc under the hood. Python itself is open source, and I extensively use tools by Anaconda. My thanks to all those individuals who help make the world go round behind the scenes.

1 Setting up python

First, before we can get started *writing code*, we need to set up python on your local machine. My suggestion for crime analysts is to install the Anaconda version of python, which can be downloaded for your machine at https://www.anaconda.com/download. I am writing this book on Windows, but most of my advice should also extend to Mac and Unix users (you can download Anaconda and run python on any of those operating systems). Where it might make a difference, I will provide a callout note. For example:

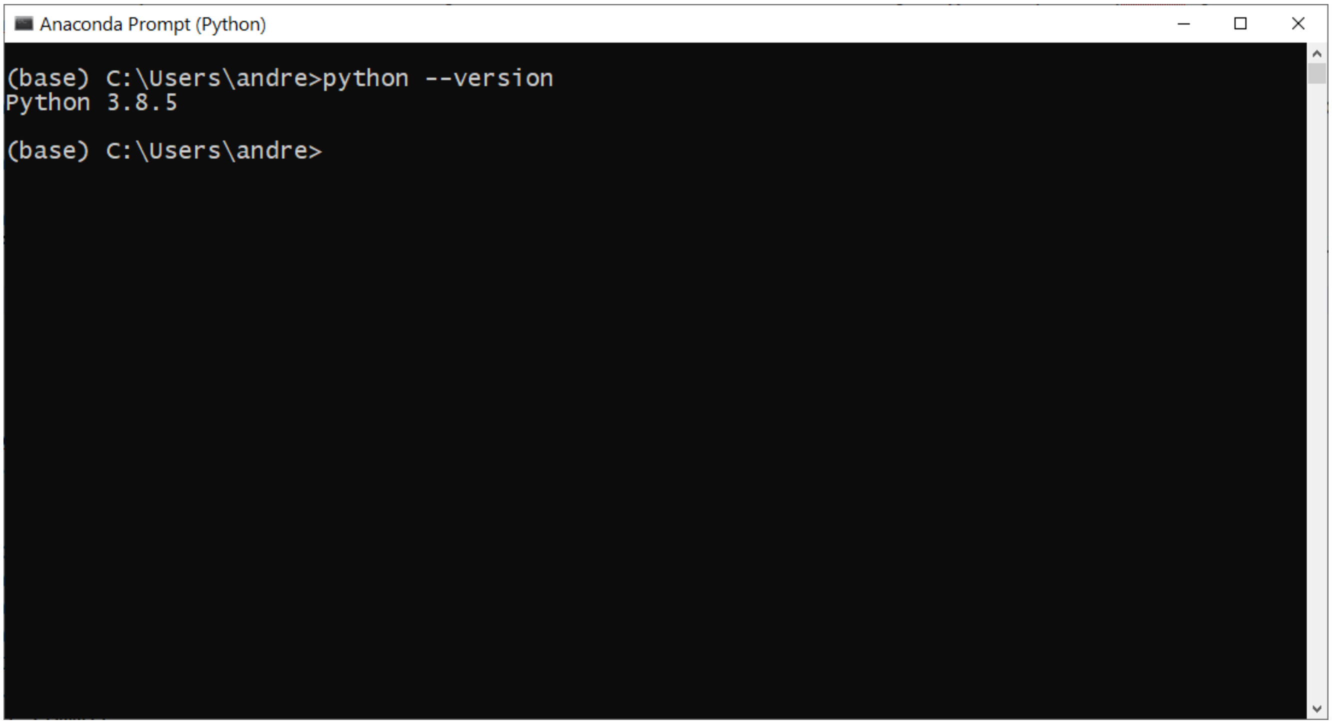
> Note
>
> When using different paths to file locations, Windows machines use backslashes, e.g. `C:\Users\andre`, whereas Mac and Unix machines use forward slashes, `/users/andre`.

Once Anaconda is installed, go ahead and open up the *Anaconda command prompt* (treat this just the same as any program you have installed on your machine, so on Windows can navigate the programs that pop up when clicking the Windows icon on the lower left). Once the Anaconda command prompt is open, it should look like something below.



```
(base) C:\Users\andre>
```

Go ahead and type `python --version` into the command prompt, hit enter, and see what the results are. This will tell you if you have installed python correctly! My python version is `3.8.5`. Your version may be different than the one I wrote this book with, but the content I will be covering in this book it will not make a difference.

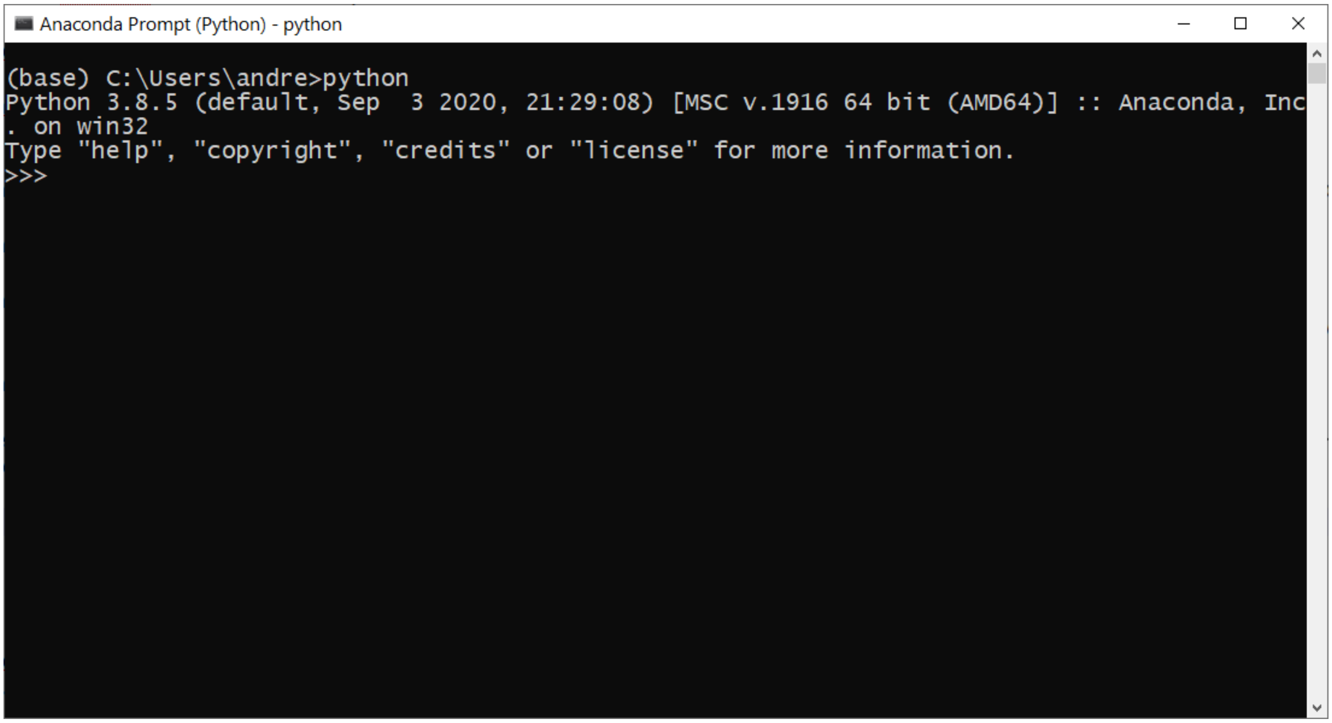


```
(base) C:\Users\andre>python --version
Python 3.8.5

(base) C:\Users\andre>
```

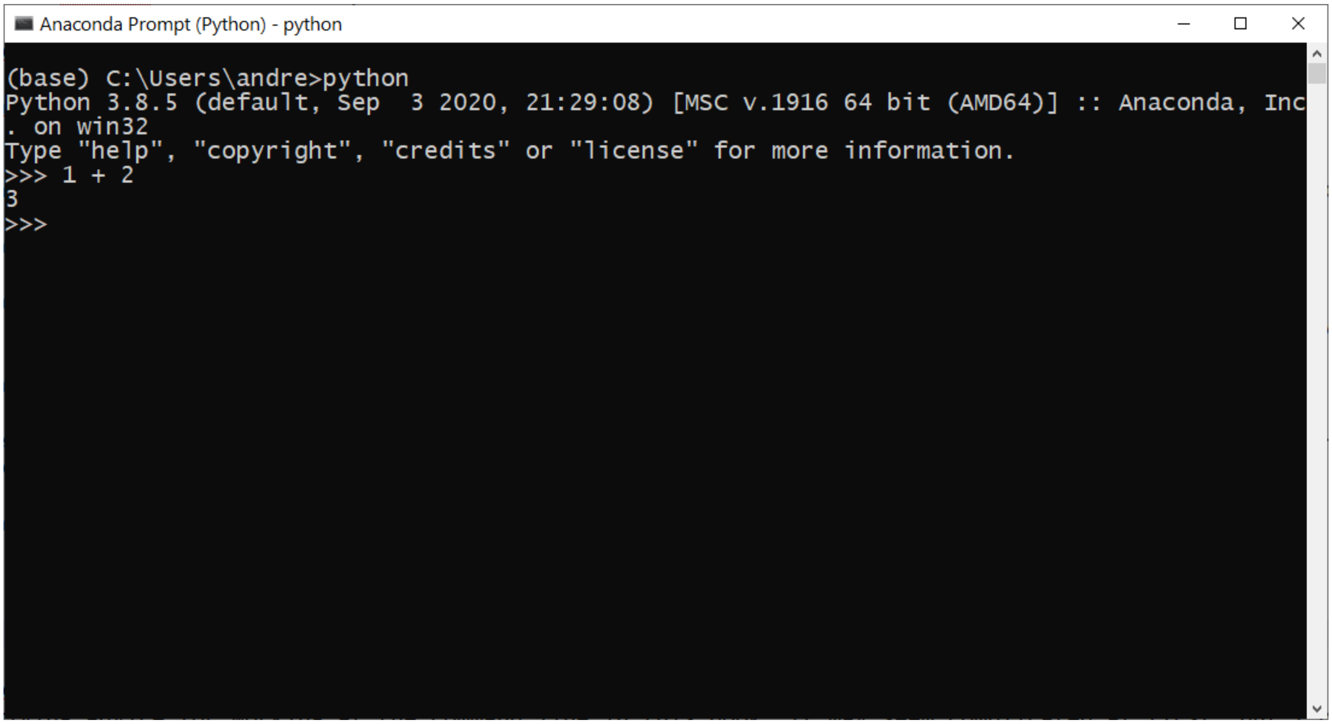
1.1 Running in the REPL

Now, we are going to run an interactive python session, sometimes people call this the *REPL*, read-eval-print-loop. Simply type `python` in the command prompt and hit enter. You will then be greeted with this screen, and you will be inside of a python session.



```
(base) C:\Users\andre>python
Python 3.8.5 (default, Sep  3 2020, 21:29:08) [MSC v.1916 64 bit (AMD64)] :: Anaconda, Inc
. on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The cursor in the terminal should be located at the `>>>` part of the screen. Now at the prompt simply type `1 + 2`, hit enter, and it will output the answer:



```
(base) C:\Users\andre>python
Python 3.8.5 (default, Sep  3 2020, 21:29:08) [MSC v.1916 64 bit (AMD64)] :: Anaconda, Inc
. on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 1 + 2
3
>>>
```

Congrats, you have now written python code.

Note

The location of the cursor at the terminal at this point should be on the `>>>` line. You cannot go up to a prior line in the terminal, like you can in a word editor, but you can edit items on a single line in the terminal. So you can type `1 + 2`, and then before hitting enter hit backspace, and edit the line to be `1 + 3`.

1.2 Running a python script

Now lets make a simple python script, and call that script from the command line. First, navigate to any folder on your machine that you can add a file (see the note below for navigating to different locations via the command line). Here I navigated to the folder on my machine `B:\code_examples`, but your folder location will likely be different. Make a simple file, call it `hello.py`, in that same folder. You can initialize the file by simply writing `echo "" > hello.py` at the command prompt, or `touch hello.py` is easier if on a Mac/Unix machine. Or in the Windows OS make a text file and then rename the extension to `.py` instead of `.txt`.

Note

I will be giving advice for working at the command line in this book. It may seem complicated at first, but I only have a handful of commands memorized. It is important for project management to know exactly *where* you run commands from. Here are a few notes on using `cd` to navigate to different folders:

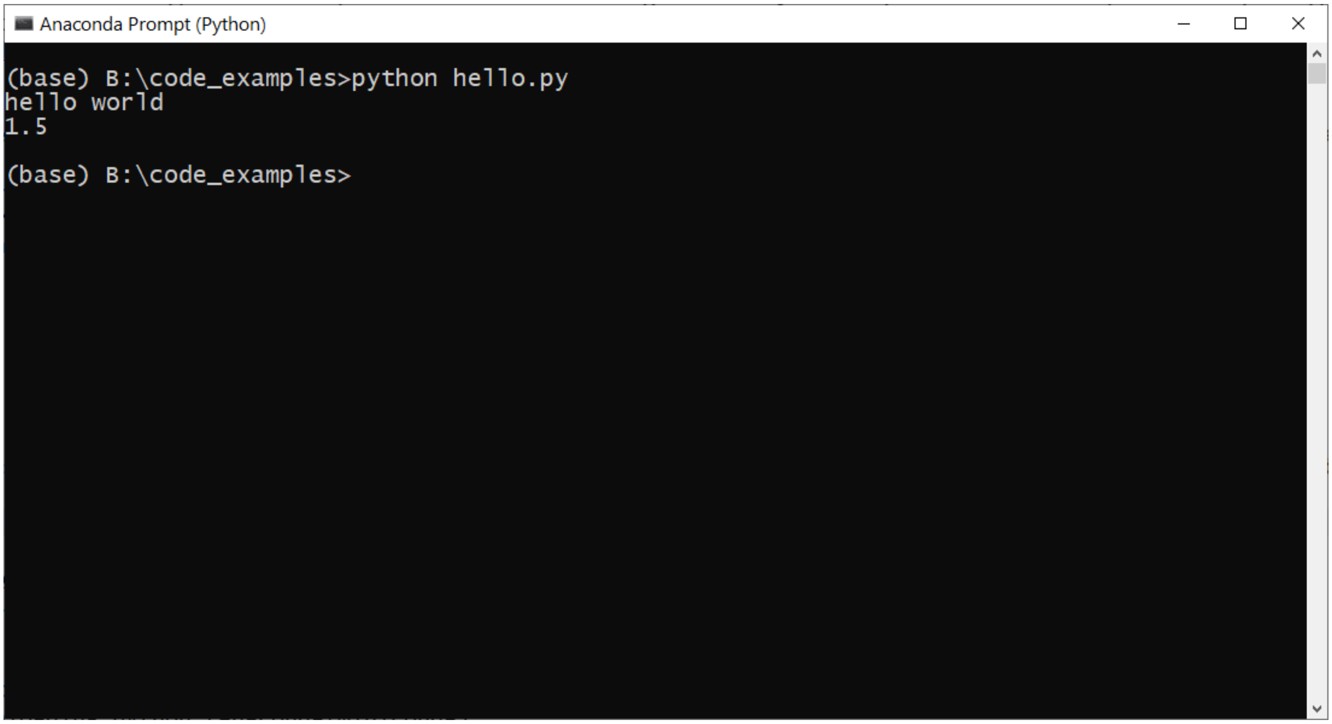
- use `cd YourPath\YourFolder` to move to different folders, e.g. on Windows it might be `cd D:\Dropbox\Project`, whereas on Unix it might be `cd /Project/sub_folder`. On windows, to switch to a different drive, you can just type that drive letter (no `cd` at the front, e.g. if you just type `D:`, the command prompt will switch the directory to the `D:` drive.
- If your path has a space in it, it needs to be quoted when using `cd`, for example `cd "C:\OneDrive\OneDrive - Uni\Folder"`. Without the quotes, the `cd` command will give an error.
- use `cd ..\` (or `cd ../` on Unix/Mac) to move up a folder, e.g. if you are in `D:\Project\sub_project` and you type `cd ../` you will now be in `D:\Project`.
- You do not need to type the full path to move down a single folder, so if you are in `D:\Project` and you type `cd .\sub_project` you will move down into the `D:\Project\sub_project` directory.
- use the `pwd` command to print out the current directory.

In that `hello.py` file (which is just a text file), open it using whatever text editor you want. Then type these lines of code into that file, and save the file:

```
# This is a comment line
x = 'hello world'
print(x)

y = 3/2
print(y)
```

Now back at the command prompt, type in `python hello.py` and hit enter. You should see that it ran your python file, and printed out the results.



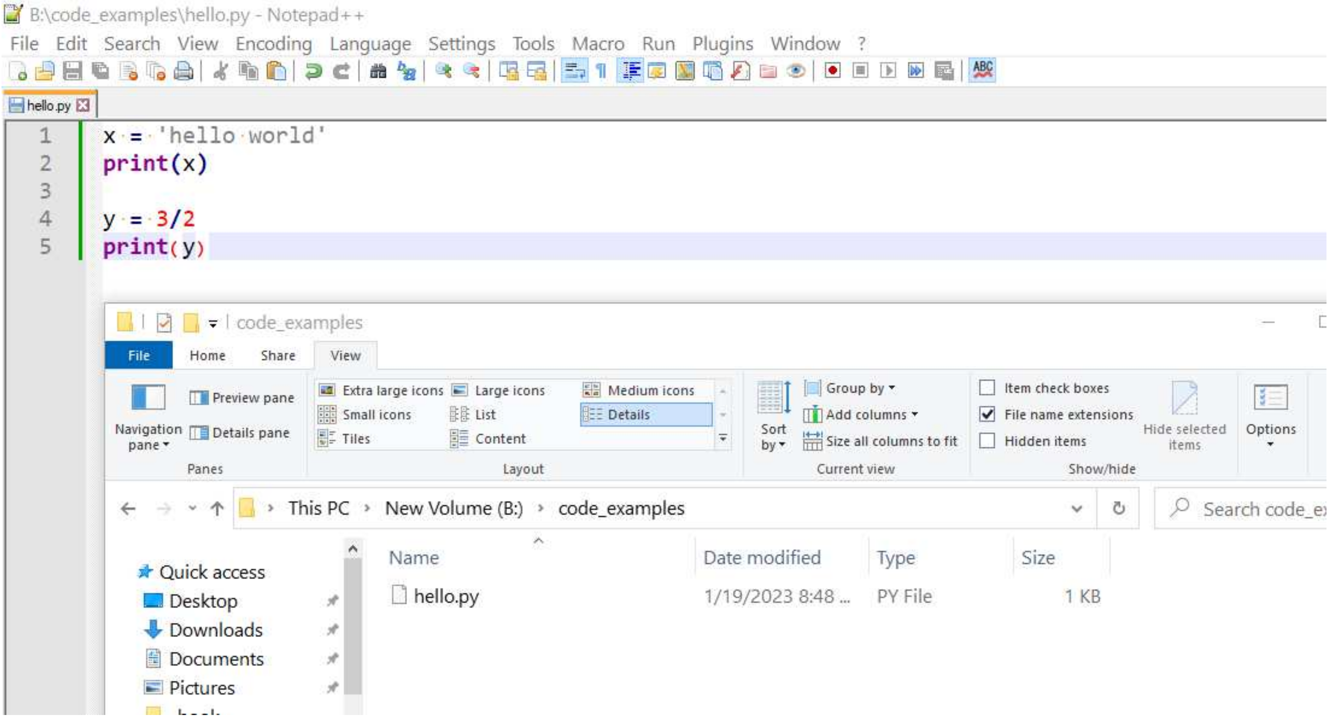
```
(base) B:\code_examples>python hello.py
hello world
1.5

(base) B:\code_examples>
```

Certain scenarios will dictate whether you are writing code in an interactive REPL session, or running code via scripts. Most of my work, I debug initial code using the REPL, and then save the finalized code and run the entire set of procedures through a script.

1.3 Some Extra Notes

I use Notepad++ to write much of my code. Here is what the prior set of code looks like, showing Notepad++ and the file itself:



```
x = 'hello world'
print(x)

y = 3/2
print(y)
```

Name	Date modified	Type	Size
hello.py	1/19/2023 8:48 ...	PY File	1 KB

Notepad++ knows that a `.py` file is a python file, and so gives nice code formatting. It also allows you to set an option to see whitespace, which will become more important later when writing conditional code blocks in python. (You should not write code in a program that formats your text, such as Microsoft Word, as its auto-formatting can cause errors in your code.)

But there are other options to help you write python code. On occasion at my work I use Visual Studio (VS) Code, which is an entire IDE (integrated development environment). This just means it has extra stuff (like an embedded command prompt and github support). VS Code has extensions for python as well as several other languages. Another popular IDE for python is PyCharm. These again are mostly exchangeable, I like Notepad++ for its simplicity.

Note

Here are a few additional command line commands I use on a regular basis:

- On Windows, to clear the terminal you can use `cls`, on Unix/MAC you can use `clear`
- you can use `cat file.py` to print out the contents of a file to the terminal
- you can use `mkdir` to make a new folder
- you can pipe output to a file, e.g. `python hello.py > log.txt` would save the output of the command above to the `log.txt` file instead of printing to the terminal.
- if you use `python script.py >> log.txt` this *appends* the output to log text file. Which is useful for repeated tasks that update over time.

Another option to write python code (especially for individuals who do scientific computing) is to use Jupyter notebooks. Many beginner python tutorials will suggest this. I will have a later chapter showing how to make a standardized report using Jupyter Notebooks, but I do not suggest this tool for beginners when first getting started. Many of the pieces of information I will discuss in this book about project management are difficult to control with Jupyter.

Now that you have the background to run python code, next chapter will cover more of the basics of how to write python code.

2 Getting started writing python code

While the prior chapter showed you how to run code, either in the REPL or via scripts, this chapter will get you started writing python code itself. The code examples shown via the text boxes I suggest typing into a REPL session to follow along, but you could also save them into `.py` scripts and run them directly as well. The grey portions are what you would type into the terminal, and the blue boxes are what should be printed out.

This chapter will walk through an introduction to working with numbers, strings, booleans, lists, and dictionaries. These *objects* are the building blocks of pretty much all code in python.

> Note
>
> On first read, this chapter (maybe all of them), will seem like a lot of information, and will likely be boring. I suggest to follow along in the REPL, but to *skim* the chapters fairly fast (especially Chapters 2, 3 & 4 on the basics of objects, strings, and looping). I have intentionally tried to be fairly comprehensive for the basics. When working on actual projects, you may need to revisit chapters to understand and re-acquaint yourself with the topics. You do not become an expert by reading a chapter one time, but by repeatedly writing code for your own projects over time.

2.1 Numeric Values

To get started, you can think about python code as *objects* and *operations* applied to those objects. So for example, if you run the python code:

```python
x = 1
y = x + 1
print(y)
```

```
2
```

Here we first create an object, `x` that is *assigned* a value of 1 via the `=` symbol. We then create a second object, `y`, that is assigned the value `x + 1`. And then finally, we `print` the value of the `y` object. `print` is a function, whose only purpose is to output the value (or more specifically the string representation of that object), to the terminal (or whatever location you tell python to output its results to).

Note

In the REPL, it is also possible to just type a single item on a line and it will print the output to the terminal. So instead of `print(y)`, in a REPL session you could just type `y` and it will accomplish the same thing. In a script though, only `print(y)` would send the output to the terminal.

In the example above the objects were integer values. You can also have float numeric objects as well. Python, unlike some programming languages, does not force you to define the type of object before hand.

```
x = -1
y = 3.2
z = x/y
print(z)
```

```
-0.3125
```

When dealing with numeric values, python is smart and coerces `z` here to be a float value, even though it uses one integer as input (even with two integers, e.g. `z = 1/2`, in python `z` will be a float). You can see that here I did division, most of the math operations are similar to what you would type into any calculator, with the exception that powers use `**` instead of `^`:

```
# Showing off the different
# math operations
x = 2
print(x - 1)
print(x + 1)
print(x/2)
print(x*2)
print(x**3)       # x to the third power
print(x**(1/2))   # x to the 1/2 power (square root)
```

```
1
3
1.0
4
8
1.4142135623730951
```

One special operator in python is to modify an objects numeric value. So for example, to increment an objects value by one, you could do:

```
x = 1
x = x + 1
print(x)
```

```
2
```

But it is easier to use the special notation of `x += 1`:

```
x = 1
x += 1
print(x)
```

```
2
```

Note you can also do other mathematical operations, such as subtracting `x -= 1`, multiplication `x *= 2`, division `x /= 2`, or exponentiation `x **= 2`.

The final basic numeric examples to give are `//`, integer division, and `%`, the modulus operator. Integer division only gives the whole numbers in a division problem, and modulus is the remainder of the division problem.

```
x = 5
print(x//2)
print(x % 2)
```

```
2
1
```

Later chapters discussing libraries intended to work with tabular data (numpy and pandas), I will discuss numeric data processing in further detail. As you often don't want to do math on a single object, but a vector of multiple objects.

2.2 Strings

Another basic object you will often be using in python are strings. If you enclose a set of characters inside of quotes, that results in a string object. Here I even do addition of string objects, which concatenates the two strings together.

```
x = "ABC"
y = "de"
```

```
z = x + y
print(z)
```

```
ABCde
```

Strings can hold any alpha-numeric characters, so strings can hold text representations of numbers. Note that adding two strings is not the same as adding two numeric values! It concatenates the two strings together.

```
x = "1"
y = "3"
z = x + y
print(z)
```

```
13
```

You can coerce strings to numeric values via the `int` and `float` functions.

```
x = "1"
xi = int(x)
xf = float(x)
print(xi,",",xf) # can print multiple objects at once
```

```
1 , 1.0
```

And vice-versa you can coerce numeric values to be strings via the `str` function.

```
xi = 1
xf = 1.0 # if you use decimal, will be float

xsi = str(xi)
xsf = str(xf)

print(xsi + "|" + xsf) #concat the strings
```

```
1|1.0
```

Note

I have introduced three functions so far, `print`, `int`, `float`, and `str`. Functions in python take the form `function(input)`, so a particular name followed by two parentheses.

Strings can be enclosed either with single or double quotes. Note however that special characters in strings can cause issues. Backward slashes in windows path variables is one common example:

```
project_path = "C:\Project\SubFolder"
project_path # Note no print statement
```

```
'C:\\Project\\SubFolder'
```

Note

What gets printed to the console is not necessarily the same text representation of the object itself. For example, try `x = "Line1\nLine2"` and then type just `x` at the terminal and see what is printed out, vs typing `print(x)`. In this example, print interprets the line breaks in the string, whereas just typing `x` in a REPL session does not.

You can see that python inserted extra slashes! If you want the string exactly as you typed it, you can use the `r""` string option, which stands for *real* string.

```
project_path = r"C:\NoExtra\BackSlashes"
print(project_path)
```

```
C:\NoExtra\BackSlashes
```

If you want a multi-line string in python, you can use triple quotes. Note that this string has line breaks in the string object.

```
long_note = '''
This is a long
string note
over multiple lines
'''

print(long_note)
```

```
This is a long
string note
over multiple lines
```

If you have a very long string, such as a url, that you don't want to insert line breaks into, you can wrap the string in parenthesis. The python interpreter just turns it into a string in the end:

```
long_url = ("Pretend I am a super "
            "long url on a single line")

print(long_url)
```

```
Pretend I am a super long url on a single line
```

I have an entire chapter, Chapter 3, devoted to more advanced usage of strings.

2.3 Booleans

Boolean objects can only have two values, `True` or `False`.

```
print(1 == 1)
print(2 == 3)
```

```
True
False
```

You can see I use `==` to do an equality comparison. Remember a single `=` is for assignment. To use does not equal, the symbol is `!=`:

```
print('a' != 'a')
print('b' != 'c')
```

```
False
True
```

And then one can also use less than and greater than logic as well:

```
print(1 < 1)
print(1 <= 1)
print(2 > 1)
print(2 >= 1)
```

```
False
True
True
True
```

Note one can also do less than checks with strings, e.g. `'a' < 'b'` is `True`, but I don't use this very frequently. The example below shows a case that is probably not intended, as it accidentally compares string representations of numbers instead of numbers directly.

```
print('10' < '2')
```

```
True
```

Often you use a boolean to do conditional logic in python code. So you can have an `if` statement like below:

```
a = 1

if a == 1:
  print(a + 1)
```

```
2
```

A special part of python syntax is that white space is special. To denote that the `print` line is inside of the if statement, we append several spaces to the front. The number of spaces does not matter, it needs to be consistent though. (And you can technically also use tabs instead of spaces, but I highly recommend against that, as it can make editing the files a pain.)

Python uses `if`, `elif`, `else` for conditional statements. Here is an if and an else example:

```
a = 1

if a != 1:
  print('a does not equal 1')
  print(a + 1)
else:
  print('I am in the else part')
```

```
I am in the else part
```

The way these statements work, is that it checks if the first `if` statement is true. If that statement is true, it executes the code nested in the if part, *and then exits the code block*. If the `if` statement evaluates to False, it then executes the `else` statement block for all other cases.

Sometimes you want to chain together multiple checks, e.g. “if A, do Y, else if B, do X”. To do that in python code, you would use `if` and then `elif`.

```
a = 1

if a != 1:
  print('a does not equal 1')
  print(a + 1)
elif a == 1:
  print('I am in the elif part')
else:
  print('I am in the else part')
```

```
I am in the elif part
```

If an `elif` is true, it executes that block and then exits, same as the `if` statement. So in the above code snippet, because the `elif` statement is true, it never gets to the `else` part of the conditional logic.

There is nothing that ties the different conditional statements together, so not all sections need to reference the same item:

```
a = 1
b = 'X'

if a != 1:
  print('a does not equal 1')
elif a == 2:
  print('a equals 2')
elif b == 'X':
  print('I am in the b check elif part')
else:
  print('I am in the else part')
```

```
I am in the b check elif part
```

And you can write subsequent further conditional logic inside of a conditional.

```
a = 1
b = 'X'
```

```
if a != 1:
  print('a does not equal 1')
  if b == 'X':
    print('b check in first if')
else:
  print('I am in the else part')
  if b == 'X':
    print('b check in elif')
```

```
I am in the else part
b check in elif
```

Sometimes in part of the condition, you want to do nothing. In those cases, you can use `pass` inside the condition.

```
a = 1

if a == 1:
  pass # This snippet will print nothing
else:
  print('I am in the else part')
```

You typically want the more common conditions first in a series of if statements, and less common conditions further down. So if the most common condition you do nothing, and only in rarer conditions you perform some action, this is a perfectly reasonable way to write your if statements.

These examples compare just two objects, but you can compose multiple conditional statements using `and` and `or`.

```
# and example
a = (1 == 1) and (2 == 2)
print(a)

b = (1 == 1) and (2 == 3)
print(b)

# or example
c = (1 == 1) or (2 == 3)
print(c)
```

```
True
False
True
```

There are short hand operators though, ampersand for `and` and the pipe for `or`:

```
# ampersand for and
a = (1 == 1) & (2 == 2)
print(a)

b = (1 == 1) & (2 == 3)
print(b)

# pipe for or
c = (1 == 1) | (2 == 3)
print(c)
```

```
True
False
True
```

You technically don't need the parentheses in the above examples, but I find it much easier to read the code that way and keep different terms together:

```
#       This is false        But this is true
a = ((1 == 2) & (2 == 2)) | (4 == 4)
print(a)
```

```
True
```

But most of the time, if possible, I just break it down in code and make the ultimate if statement line as simple as possible.

```
check1 = (1 == 2) & (2 == 2) # This is false
check2 = (4 == 4)            # This is true

if check1 & check2:
  print('Both check1 and check2 are true')
elif check1 | check2:
  print('At least one of check1 or check2 are true')
else:
  print('Neither check1 or check2 are true')
```

```
At least one of check1 or check2 are true
```

On occasion one does not use the math operators to generate the boolean statements, but `is` or `is not`. Perhaps the most common use of this is with the `None` object, what can be considered missing data in python.

```
x = None

if x is None:
  print('x has no value')
else:
  print('x has some value')
```

```
x has no value
```

Technically when you write `a is b`, it not only checks the objects values, but also that it references the *exact same object* in memory. Technically `x == None` will work above, but it is common practice to write it the `x is None` way. You can also check the opposite condition via `is not None`:

```
x = None

if x is not None:
  print('x has some value')
else:
  print('x is None')
```

```
x is None
```

For a final example, one can drop the comparison operators or if statements entirely. If you pass an "empty" object to an if statement, python checks to see if the object has any elements at all, and returns `True` if it does. So if you pass in an empty string, the if statement returns `False` here:

```
x = ''

if x:
  print('x has some value')
else:
  print('x is an empty string')
```

```
x is an empty string
```

This works for other python objects, such as empty lists and dictionaries, which will be illustrated in the subsequent section.

2.4 Lists

It is important for beginners to have an understanding of different objects that are containers for other objects. The first is a *list* object. A list contains multiple other objects, and you create a list by placing items inside brackets, and separating the items via commas. It is easier to show than to explain in words!

```
x = [1,2,3]
print(x)
```

```
[1, 2, 3]
```

Lists can hold different object types, it can contain both strings and numeric values in the same list for example. Here I also show that lists can contain other defined python objects.

```
a = 1
b = 'z'
x = [a,b,3]
print(x)
```

```
[1, 'z', 3]
```

You can split items in a list onto multiple lines in the python interpreter, it knows to look for the end bracket to know the input to the list is finished. So the resulting list below is exactly the same as `x = [1,2,3]`, just a different way to write it (it can be nice to split very long lists onto multiple lines for readability in your code).

```
# it is ok to define lists
# over multiple lines
x = [1,
     2,
     3]

# it is the same as earlier
# on a single line
print(x)
```

```
[1, 2, 3]
```

A common error when working with lists is to forget the comma. With numeric values this will often result in an error, but with strings it can sometimes not result in an error, since python will just implicitly concatenate the strings together. For an example:

```
# This is probably not what was intended
x = ['a' 'b', 'c']
print(x)
```

```
['ab', 'c']
```

So here the comma between the first two strings was omitted, so the resulting list is just two elements, with the first being `'ab'`.

> Note
>
> One benefit to writing lists on multiple lines, is you can use *column* editing in various text editors. In Notepad++, you can hold down the alt key and select multiple rows to edit at once. Most advanced text editors have a similar feature.

You can access individual items in a list via its index. Note in python that list indices start at 0, not at 1, so the first element of a list will be 0, the second element will be 1, etc.

```
y = ['a','b','c']
print(y[0])
print(y[1])
print(y[2])
```

```
a
b
c
```

You can also use *negative* indices to access items in reverse order in a list. So -1 accesses the last item in a list, -2 the second to last item, etc.

```
y = ['a','b','c']
print(y[-1])
print(y[-2])
```

```
c
b
```

Finally in terms of accessing multiple items in a list, you can use slice notation. So `x[1:3]` would access the 2nd and 3rd items in the list (it is equivalent to `[,)` notation is mathematics, so the left end of the slice is closed, and the right end of the slice is open.

```
y = ['a','b','c']
print(y[1:3]) # note it returns a list!
```

```
['b', 'c']
```

You can also use `x[1:]` to indicate, “give me the 2nd item in the list until the end”, or `x[:3]` to “give me the first 3 items in the list.

One can create an empty list, simply by assigning `[]` to a value. Another useful trick to know is that you can do a boolean check for an empty list.

```
x = []

if x:
  print('The x list is not empty')
else:
  print('x is empty')
```

```
x is empty
```

An empty list has no objects, so if you try to access an object it will return an error. Above if you try to use `x[0]`, you will get an error `index out of range`.

Another boolean operation on lists is to check if an element is contained in that list.

```
x = ['a','b','c']

if 'd' in x:
  print('The list has a d element')
elif 'c' in x:
  print('The list has a c element')
else:
  print('x has neither d or c')
```

```
The list has a c element
```

One special piece of information you need to know about lists is that they are *mutable*. What does that mean exactly? It means that we can modify the contents of a list. So for example, we can replace a single item in a list.

```
y = ['a','b','c']
print(y)

y[1] = 'Z'
print(y)
```

```
['a', 'b', 'c']
['a', 'Z', 'c']
```

For a more complicated example, lists can point to other lists. Because lists are mutable, you can modify the inner list here, and the outer list reflects this change.

```
x = ['a','b','c']
y = [1, x]
print(y)

# if we alter x
# y points to
# the altered x list
x[1] = 'Z'
print(y)
```

```
[1, ['a', 'b', 'c']]
[1, ['a', 'Z', 'c']]
```

The way to think about this, the list `y` here does not actually contain the contents of `x`, it simply *points to* the `x` object. So if the `x` object gets changed, it points to that new `x` object.

This may seem quite in the weeds, but it is an important feature of the python programming language. It allows one to write many different algorithms in a simpler fashion when one can alter lists in place.

You can concatenate two lists together by adding them:

```
x = ['a','b','c']
y = [1,2]
z = x + y
print(z)
```

```
['a', 'b', 'c', 1, 2]
```

You can also make a repeated list via multiplication:

```
x = ['a','b','c']
y = [1]
print(y*3 + x*2)
```

```
[1, 1, 1, 'a', 'b', 'c', 'a', 'b', 'c']
```

Lists have several *methods* to modify their contents; two commons one used are sorting and reversing:

```
x = [3,1,2]

x.sort() # sorting the list
print(x)

x.reverse() # reverse ordering
print(x)
```

```
[1, 2, 3]
[3, 2, 1]
```

Note

When you sort or reverse a list, it does this operation and modifies the list *in-place*. So the code `y = x.sort()` is probably incorrect, as `x.sort()` returns nothing. So `y` does not equal the sorted list, but is `None`.

Methods are special functions that are tied to particular objects. So look like `object.method(input)`. They will always be demarcated from the base object by a period – so this means you cannot use a period in a variable name. For example if you type `q.i = 1` into the terminal it will return an error that `q is not defined`. These methods can take additional arguments (they are just like functions), but these examples just use the default. For example, you can sort in descending order:

```
x = [3,1,2]

x.sort(reverse=True) # passing arg
print(x)
```

```
[3, 2, 1]
```

Note

This is the first example I have shown for a function that has a *keyword* argument, so instead of `function(input)` it is `function(keyword=input)`. Functions can take multiple arguments, such as `function(input1,input2)`. In this scenario the order of the arguments matter, and you may use keyword arguments to distinguish between the inputs. I will go into more details on this in a subsequent chapter on defining your own functions.

You can also append or remove items from lists:

```
x = [3,1,2]

x.append('a') # appending an item to end of list
print(x)

x.remove(1) # removing an item
print(x)
```

```
[3, 1, 2, 'a']
[3, 2, 'a']
```

To find the specific location of an item in a list, you can use the index method:

```
x = ['a','b','c']

# will be 1, the 2nd item in a list
bindex = x.index('b')
print(bindex)
```

```
1
```

There are other methods to lists I have not shown here, if you run the `dir` command on an object, it will print out all of its potential methods. I encourage you to experiment yourself and see how the other methods, like `count` or `pop`, work.

```
x = ['a','b','c']

# you can look at the methods
# for a object using dir(object)
me = dir(x)

# there are many more! only
# printing a few to save space
```

```
print(me[-6:])
```

```
['index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

Sometimes you want a list like object, but you do not want it to be mutable (so you cannot do operations like change a single value, append, or sort the list). This may occur if you have a set of constants in your script, and you know they should never be altered. Placing them in a *tuple* is one way to ensure they don't accidentally get modified. Tuple's look mostly the same as a list, but use parentheses instead of square brackets:

```
y = (1,2,3)
y[1] = 5 # this will give an error
```

```
TypeError: 'tuple' object does not support item assignment
```

You can convert a list to a tuple via the `tuple` command:

```
y = [1,2,3]
z = tuple(y)
y[1] = 5 # this is ok
print(y) # can see y list is updated
z[1] = 5 # this is not, again cannot modify tuple
```

```
[1, 5, 3]
```

```
TypeError: 'tuple' object does not support item assignment
```

Or vice versa convert a tuple to a list via the `list` command:

```
z = (1,2,3)
y = list(z)
y[1] = 5 # this is ok
print(y)
z[1] = 5 # this is not, again cannot modify tuple
```

```
[1, 5, 3]
```

```
TypeError: 'tuple' object does not support item assignment
```

One last note about tuples, you can assign multiple objects at the same time. So you can do:

```
x, y = 1, 2
print(x,y)
```

```
1 2
```

This is called *tuple unpacking*. The reason it is called this is that when you *don't* unpack the multiple values separated by a comma, it returns a tuple.

```
t = 1, 2
print(t)
```

```
(1, 2)
```

And you can assign more than two values:

```
a,b,c,d = (1,'a',6,-1.2)
print(a,b,c,d)
```

```
1 a 6 -1.2
```

This can be convenient in various for loop examples (shown in Chapter 4), and when functions return multiple values (shown in Chapter 5). Besides this though, tuples are not as commonly used as lists in my experience. But one example use they have is illustrated in the next section, where one has a need to use immutable tuples for dictionaries.

2.5 Dictionaries

The second major container of items in python is a dictionary. So to access items in a list, it is just a set order, the first element is `mylist[0]`, the second element is `mylist[1]`, etc. Sometimes you want to be able to access the elements by simpler names, e.g. imagine you had a list to contain a persons information:

```
# using a list to hold data
x = ['Andy Wheeler','Data Scientist','2019']
```

To access the name item, you need to know it is in the 0 index, the title item is in the 1 index, etc. It is probably easier to refer to this data via a dictionary.

```
# using a list to hold data
d = {'name': 'Andy Wheeler',
     'title':'Data Scientist',
     'start_year': 2019}

print(d)
```

```
{'name': 'Andy Wheeler', 'title': 'Data Scientist', 'start_year':
2019}
```

Now, it is easier to access an individual item via `dict[key]`, so if I only want the name, I just reference that explicitly:

```
# Grabbing the specific name element
print(d['name'])
```

```
Andy Wheeler
```

The terminology for dictionaries is that they have *keys* that reference *values*. Values can be anything: numeric values, strings, lists, other dictionaries, etc. Keys however need to be *immutable*, even though dictionaries themselves are mutable (so you cannot use a list as a key, but you can use a tuple). Here we can modify the values of the original `d` dictionary I created. You can also add in a new element once the dictionary is created.

```
# Modifying the start_year element
d['start_year'] = 2020

# Adding in a new element tenure
d['tenure'] = 3

print(d['name'])
print(d['title'])
print(d['start_year'])
print(d['tenure'])
```

```
Andy Wheeler
Data Scientist
2020
3
```

Most often people use strings for keys, since the main benefit of dictionaries over lists is to have a name for referencing specific objects. But it can be a number as well. So say you had numeric ids in another database referencing specific locations, it may make sense to write your dictionary using those same numeric identifiers.

```python
# You can have numeric values
# as a dictionary key
d = {} # can init a dict as empty
d[101] = {'address': 'Penny Lane', 'tot_crimes': 1}
d[202] = {'address': 'Outer Space', 'tot_crimes': 42}

# These show a dictionary inside of a dictionary
print(d[101])
print(d[202])
```

```
{'address': 'Penny Lane', 'tot_crimes': 1}
{'address': 'Outer Space', 'tot_crimes': 42}
```

And similar to empty lists, an empty dictionary will return `False` in a boolean if statement:

```python
d = {} # can init a dict as empty

if d:
  print('The dictionary d is not empty')
else:
  print('The dictionary d is empty')
```

```
The dictionary d is empty
```

There are *many* types of more complicated objects in python, the very first example in the preface showed an example working with *datetime* objects. But under the hood, they are often just a container to conduct different operations on the objects I listed above: numeric values, strings, lists, and dictionaries.