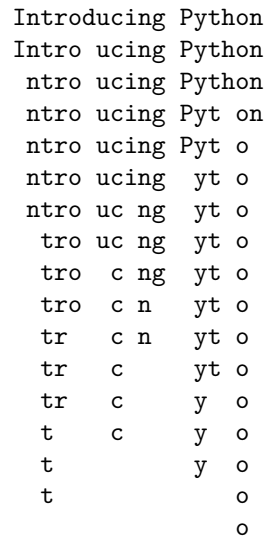
Introducing Python

Bernie Hogan

November 29, 2021


Introducing Python
Intro ucing Python
ntro ucing Python
ntro ucing Pyt on
ntro ucing Pyt o
ntro ucing yt o
ntro uc ng yt o
tro uc ng yt o
tro c ng yt o
tro c n yt o
tr c n yt o
tr c yt o
tr c y o
t c y o
t y o
t o
o

Contents

Prologue: Welcome to Python

0.1 Welcome!

Thanks for checking out this book.

The origins of this book started many, many years ago as a series of Python scripts for teaching social scientists how to code in Python. Over the years these grew into ever more extensive lecture notes. During the process of compiling lecture notes for my larger book "From Social Science to Data Science", I thought it might be a good idea to also take my introductory notes and compile them as a book.

It turned out to be a little more work than I expected! Part of the issue is that in reading my notes in a book form I became aware of certain assumptions I made or glossed over because I thought I could talk through such issues during a lecture. Yet, students appreciate resources. I adore using this book as a set of "living notebooks" in Jupyter. You can edit this book, run it, change it etc. But not much beats being able to have a printed or at least nicely formatted version of these notes to annotate, flip through and refer back to at a glance. So I hope you appreciate these notes. I further hope they encourage you to consider learning Python and developing your own way of working programmatically.

0.2 Python as a computer language

Python is a computer language. Most computer langauges have some resemblance to human language but they are very fussy! Every punctuation must be correct, every space has to be in place, every word has to have the correct capitalisation.

Language tends to have a notion of **nouns** and **verbs**. Nouns are often thought of as objects, like a *pizza*. Verbs are actions like *throw*. You should not throw a pizza. Python has a similar notion of **objects** and **functions/methods**. Objects contain things. Functions are the ways that we make use of things in Python. They are like the verbs. For example:

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

In this case: - `print` is the function; - `()` contains what gets done; - `"Hello world!"` is data to which we do something, in this case, we **print** the characters between the quotes. This is much safer than throwing pizza.

Most of programming will involve moving around data between these functions in increasing layers of complexity, starting first with step by step instructions, and then subsequently with increasing amounts of abstraction. That is how the programming in this book will proceed.

0.3 An outline

The first thing we will need to do is get you started with an environment for programming. Chapter 1 introduces some basics of Python and of an environment called Jupyter Lab. You might already be reading these chapters in Jupyter Lab! But if not, Chapter 1 has some tips on how to install and work with it.

In Chapter 2, we are going to start with a discussion of primitive data types. These are the basic building blocks of objects. They correspond to letters and numbers. Then in the second half of this chapter we will show **collections**. Collections include multiple data objects, either of the same type or a different type. That takes us to the end of this chapter just covering the logic of collections.

In Chapter 3, we ask, if we have a collection how can we repeat an action for each element in that collection? Thus, we will learn about **iteration**. Iterating leads quite naturally to the question: what if I want to do something some of the time (of for only some of the elements?) This means we are doing something under some *condition*. Thus, in Python we have an important notion of **conditionals**, the most common of these are `if` and `else` statements. As in:

```
if YEAR == 2020:
    buy("mask")
```

The means to assign a bunch of variables, have iterations, and do it under some conditions form the basis of programming in virtually any language. But this can also get very messy. So different languages have ways of organising code, often to *minimise redundancy* or *maximise reusability or robustness*. One essential concept for organising code in Python is to make use of **functions**, both functions that are pre-built and those that you create yourself. In 5 we look at how to build a function, what sort of inputs are possible (and useful), and how to bundle functions together as objects.

Functions can stand alone, or they can be dedicated functions for a specific class of **object**. These dedicated functions are called **methods**. We cover objects in more detail also in Chapter 5.

In Python, if it is a noun and it is not a primitive data type, then it is an object. So you can have a `tweet` object which contains data about a tweet, such as its author, time, URL, hashtags, etc. A simpler object is a `list`, which is just an ordered collection such as `["Abba", "Toto", "Gaga"]`. The type of object in Python is called its `class`. Classes are also covered in Chapter 5.

Chapter 6 looks at how to read and write files, both for reading and writing data, but also for running Python programs. This does not really involve more complex programming concepts but starts you along a path towards using Python in Jupyter and beyond.

The book concludes with some ideas and resources for further learning in Chapter 7.

This book has a series of appendcies. These are exercises that I have put together based on the material for the different chapters. These tend to work better as Jupyter notebooks, but I also wanted to compile them. I feel that some of the most interesting work I've done is not in telling people *what* to learn in Python, but inviting them to explore Python themselves. I hope the exercises encourage a level of playfulness with the code. The first, Chapter A, presents a few shorter exercises tied to each chapter of the book. The second, Chapter B, are some example answers for the prior appendix. I'm sure you won't just copy and paste the answers, but this way removes some of the temptation. Then afterwards in Chapter C I propose a few longer projects that might be a fun challenge.

0.4 Why this book?

Python is a vast and well supported language. But that can be a problem as well as an advantage. It's easy to get overwhelmed by resources, Stack Exchange pages, endless YouTube videos with ever so slightly different content. And indeed, hundreds of introductory books and blog pages.

This book is based on a decade of experience teaching graduate students Python, often graduate students from a humanities and social science background who never thought they could or would learn a programming language. It is certainly not the only way to learn, but it is the way that I have been able to teach. So in a way, this is a book mainly for me and my classes. But why limit knowledge? So for that reason, I have put this together as a .pdf and set of Jupyter notebooks so maybe it will be of use to you, too!

0.5 Dedication

This one is dedicated to the teachers who take risks to keep their learning fresh.

In my undergraduate degree 20 years ago Prof. Ron Byrne had been teaching a class on 'vocational languages' for years. The year I took it, he was using Python for the first time. It was relatively new language then. He figured it was going to replace the scripting language perl. Little did he realise that it was to become *the* pre-eminent language in data science and machine learning. But it was a gift to me to be engaged with a language so early on. Dozens of my papers have been touched by Python in some way or another.

At my current department, it's impossible to stand still. AI and machine learning are rapidly suffusing every domain of academic life in some way or another. I cannot keep teaching the same material or in the same way year on year. Who knows? In ten years I might not be teaching with Python at all! Even up until 2014 I was just teaching with script files and not Jupyter. It wasn't until around 2015 that I was using pandas! Sometimes these decisions seem a little late and sometimes they seem a little early. But they're always a risk. It's easier to teach what you know. But it's important to teach what students *need to know* and that's always a risk. More times than I can count I've been asked a question that I felt like I *should* have known the answer but didn't. Taking these risks is challenging and I've looked to some of my best teachers in the past to see how they navigated unknown and choppy waters. And in the end it's all the same - treat teaching as an opportunity to learn and not just an opportunity to teach.

I wish I could list all my great teachers here, but that list is more important to me than you. So why not take a couple seconds and reflect yourself on those teachers in your life who took a risk to teach you something new? I think these people are all around us if we know where to look and act like we want to learn. So this book goes out to them.

Chapter 1

Introducing Python and Jupyter

1.1 What is Python?

Python is a programming language. It is interpreted by the computer and transformed into low level code that can be interpreted by a processor. To make statements in Python, the most direct way is to type commands into a "Python console" (for example, by opening the terminal, or on windows the "anaconda prompt", then typing "python"). If you already have Python installed, on a Mac or Linux computer (and maybe someday Windows PowerShell) you can open up the terminal and type `python`. Then you will see a welcome message and a series of three chevrons, like so: (with what is likely to be some slight difference in the welcome message depending on your setup.)

```
Python 3.7.3 (default, Mar 27 2019, 16:54:48)
[Clang 4.0.1 (tags/RELEASE_401/final)] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

And in here you can enter commands. For more complex work, you will want to turn to Jupyter notebooks and Python script files (`*.py` files).

To exit the Python console, type `exit()` or press control-D. Then you will see the standard prompt, which will likely be either a single chevron (`>`) or a `\$`.

In this book, I am writing as if we are working in a Jupyter lab environment. You might be reading this on the screen or in a book form. But we will act as if the grey shaed areas are blocks of code that you can run, and the unshaded areas, including code, are just text to read.

Because this book began its life as a Jupyter notebook, it leverages certain features of Jupyter, such as *syntax highlighting*. So you will see different text in a different `font` or colour, such as:

```
print("Hello world")
```

```
Hello world
```

Instead of typing into a console, we will be typing Python into cells in a Jupyter notebook and running these. This makes it like halfway between running piecemeal in a console or running all-at-once in a single script.

1.2 Working with Anaconda and Jupyter

One of the best things about Python is that it now has an entire ecosystem for scientific computing. In this book we will be using the incredible Anaconda package for Python. This package includes a recent version

of the base Python language and a bunch of useful libraries for scientific computing. This book will not make use of most of these libraries, but they do feature in data science work generally.

In addition to installing Python, Anaconda comes with a few ways to code and develop Python programs. For standalone programs there's Spyder. It is a very involved development environment with features to help in developing code that spans multiple script files. Similer to Spyder is PyCharm, which is not presently included in Anaconda. But one program is especially useful, and that's Jupyter. Jupyter is a browser-based tool for viewing Python code alongside text, figures and results. It's like a *Microsoft Word* document where paragraphs can be run and results can be inserted directly into the document. Jupyter Lab is like a workspace where you can run multiple files in different tabs, like browser windows. It also has a nice and growing set of extensions, such as my personal favorite, the Table of Contents extension.

These documents are called *notebooks*. Their default extension is `.ipynb`, which stands for "ipython notebook". Before Jupyter there was a souped up Python console called ipython. You can still run it from the terminal by typing `ipython`. But Jupyter is like a souped up ipython.

Jupyter notebooks can be run on their own in a variety of ways.

1. The Jupyter notebook app, which you can run that from the "Anaconda Navigator" application. You can also run it from the terminal by typing `jupyter notebook` (that's the regular terminal `\$`, not the Python console `>>>`).
2. If you have a Google account, you can go to https://colab.google.com and upload your notebook or start a new one. If you store your notebooks on your Google drive you can open them in Google Colab by right clicking on them and selecting open with → Google Colab. There are similar open source and university run services like Binder. They tend not to be as polished as Google but might be right for you and do not require you to have a Google account.

3. You can read a Jupyter notebook if it has been uploaded to **GitHub repository**. Often these notebooks also have little badges in them you can click, like "Open in Google Colab". This book does not, yet. But it will render the file even though you cannot run the code there.
4. Last but definitely not least, Jupyter Lab is the most fully fledged way to run Jupyter notebooks. It is my current Python coding environment of choice. You can run it from the Terminal by typing `jupyter lab`.

1.2.1 How to get Python, Anaconda, and Jupyter?

To get the Anaconda package and all the bells and whistles, you'll need about 2-3 GB space on your computer. It's a big ask on smaller devices, but it's very thorough especially with packages like `pandas` (tables) and `matplotlib` (drawing) which can be a real pain on their own. I personally wish they threw in `geopandas` because as of this writing, it is still a separate install and it is really, really fussy.

When you download and install Anaconda, you will get an application called *Anaconda Navigator*. This is an application hub that allows you to open and run a variety of applications for scientific computing. The first application in the upper left corner is probably Jupyter Lab and the second is Jupyter. If you haven't used the terminal before you might find it easier to run Jupyter from the Navigator application, but I prefer to run it directly from the Terminal (or when using windows, running it from the `Anaconda PowerShell`).

1.2.2 Which version of Anaconda?

There are many versions of Anaconda. For your computer, you will want to select the one for the right **operating system** (MacOS, Windows, or Linux…but not ChromeOS or iOS, these mobile-oriented operating systems are not ready for data science).

Then you will want to choose your version of Python. Get the latest one. As of this writing that would be the Python 3.8 graphical installer for your preferred OS. It will be the "individual version", not enterprise or anything more fancy. You will almost certainly want a 64-bit version.

1.3 Getting started with Jupyter Lab

When you click on the Jupyter Lab icon in the Anaconda Navigator it should open up your default web browser and navigate to a page with the following URL: http://localhost:8888/lab . This URL is a little different from conventional URLs. Instead of a domain name like www.eff.org, it is just the word `localhost`. This is your machine. It turns out Jupyter lab is actually a small server running on your computer that is serving you the application through a browser. The `8888` is a port number. A server communicates using different ports, often for different services. Standard unencrypted web traffic runs through port 80, while email often runs through ports 25 and 587. When I say open Jupyter Lab, I mean navigate to that particular browser tab that is running Jupyter Lab. Remember, however, if Jupyter is not running in the background then `localhost:8888` will just display a blank page. You must launch the server first and then you can use it.

When you open Jupyter Lab for the first time, you'll be greeted with a navigation pane on the left and side and a single tab open labeled 'Launcher'. You can use the Launcher to create a new Jupyter notebook. Create one using the first button (labeled "Python 3"). This is the default Python Jupyter notebook. There are a few other types of notebooks that you can create at this point.

Here are some useful details about Jupyter Lab: 1. *The browser address bar.* This is where you would type a URL. At the moment, it probably shows localhost:8888/lab. This means that you are looking at a webpage that is run from your local computer. 'localhost' is a shorthand for the Internet Protocol (IP) address for one's own computer.

2. *The Jupyter file menu.* This is where you can click on commands for Jupyter such as **File→New Launcher** so you can create a new notebook. This is not the browser's file menu (which may be hidden if this window is in full screen). Notice one of the file menu items is called 'Kernel'. This is the term for the instance of Python that runs the code, stores data, and returns a result. Each lab notebook has its own kernel. Sometimes we will need to restart the kernel, for example if we accidentally run a command that has no end, such as "count every number". The other important thing to note in kernel is that we can clear output. Sometimes, we will want to start our Jupyter notebooks fresh. Clicking **kernel→Restart Kernel and Clear All Outputs...** will make sure that all the cells are treated as if they have never ran. It's good to do this and re-run all your code from start to finish before sharing with other people.

3. *The navigation sidebar.* On the left-hand side is where you can select a file or check to see which files are currently running. The top icon is the file icon, it points to a file browser. The navigation is pretty similar to what you would get with a file browser on your computer such as 'Finder' on Mac or 'Explorer' on Windows.

4. *The tabs panel.* With Jupyter Lab you can have multiple tabs available as different workspaces, each using a different Jupyter notebook or file. These tabs work like browser tabs. You can click on one to start working on that tab, drag the panels to change their order and check whether they have been recently saved by seeing whether there is a circle in the tab name on the right-hand side. You'll notice that there's a new file you just created and a second tab called 'launcher'.

5. *The actions panel.* This small panel has some important and common actions like 'save', 'run' and 'stop'. I tend to use keyboard shortcuts for these actions. You will definitely want to notice on this panel where it says the word 'code'. That's where we assign the 'type' of a cell. You can change the type of a cell by selecting a different type from the drop-down menu that appears when you click where it says 'Code'/'Markdown'. On the right hand side it says 'Python 3' which means that this particular notebook interprets code as Python 3 code. The right most dot is a status meter. When the computer is busy running code the circle is filled in and looks like a spot. When the computer is idle the circle is empty and looks like a ring.

6. *The main panel.* This is where the work gets done. In this panel you'll see that content is organised in cells. Each cell can be either 'code', 'Markdown' or 'raw'. **Raw** text is not highlighted and the computer just ignores special characters and code. **Code** means that the contents of a cell are treated as Python code. **Markdown** is text that has extra characters to denote formatting. For example, Markdown

uses two asterisks on either side of a string to indicate it should be bold: When I type **this** into a Markdown panel the text is rendered like **this**. Markdown is discussed more just below.

7. *A Python cell.* You can tell this is a Python cell on your computer because it has *syntax highlighting* that indicates Python-oriented words and variables. For example, the word 'print' will show up in green and comments will show up in blue. It also will say 'code' in the cell type in the main panel. To run the cell you can do any of the following:

- The file menu. Click "Run"→"Run Selected Cells".
- The right facing triangle in the actions panel.
- (My favourite) Shift-enter on the keyboard.

8. *Code numbers.* When you run a cell, it will report a number off to the left of the cell. That number represents the order in which cells were run. If there is no number that means the cell has not run yet. If you run a cell a second time, it will increment the number, so the number could actually go much higher than the number of cells in the notebook. You can also see a blue bar to the left of the number. Click that bar and it will collapse the output. This is handy if you have just printed out a lot of output but you want to hide it while you work on the code underneath.

1.3.1 How to add text to a Markdown cell

A markdown cell is one that has text in it. Markdown is a simple way to add features to text, like *italics*, headers, ~~strikethrough~~, and **bold**. In a Jupyter notebook that is rendered, you can click on a cell to see the Markdown that produced the text. Some of the more common things you will see in Markdown:

1. Use two tildes (the ~ character) for ~~strikethrough~~
2. Use two asterisks (the * character) for **bold**
3. Use underscores (the _ character) for *italics*.
4. Lists are auto generated by having several lines, where each starts with an asterisk and a space.

- Here is a list item
- A second item
- These are indented because we had a space before the asterisk

5. You can also embed code in a Markdown cell. It won't run but it will have syntax highlighting and a monospace font. This is using three tildes and then the name of the language like so:

   ~~~ python

   print("Hello World")

   ~~~

and it will be formatted on the screen like:

```
print("Hello World")
```

6. Use hash (the # symbol) at the beginning of a line to make it a heading. You can use two hashes to make it a subheading (or three for subsubheading, etc…).
7. Use the dollar symbol on either side of a formula to use math notation like $e^{\pi i} = -1$ (and I use it for *commands*).
8. Three or more dashes at the beginning of a line create a straight line across the page

---

1.3.2 How to create a new cell / navigate with the keyboard

At any given time only one cell might be in focus. To say 'in focus' means that the cell is editable. This is denoted at the bottom of the screen where it says "Mode: Edit". It also means keyboard shortcuts and the computer's list of undo actions refer to the text in that specific cell. When a cell is out of focus, it is still indicated with a blue strip on the left-hand side, but keyboard shortcuts and the undo actions refer to

the Jupyter file and cells rather than their contents. At the bottom it will say "Mode: Command". It is important to be able to navigate into and out of focus for a given cell with the keyboard if you want to be a fluent user of Jupyter. Being able to have cells of different types organized in your notebooks is where Jupyter shines. For example, you can have one cell of code, then graphical output, a well-formatted table of numbers, and your notes just below. Moving around these cells can help you navigate not just the code, but the overall analysis. It also helps you to think about chunking your code and organising your questions.

To change the focus from one cell to another, you can click on a new cell or 'run' the current cell. To run the current cell, press $shift - enter$.

To get out of focus ("Mode: Command") you can either:

- Run the current cell ($shift - enter$),
- Escape the cell ($escapekey$).
- $Single - click$ outside of the cell.

To get in focus ("Mode: Edit") you can either:

- Press $enter$,
- $Double - click$ with the mouse.

If the cell is not in focus you can tell because there is no cursor and pressing up or down will move the blue bar on the left hand side up and down. If it is in focus you can tell because pressing up and down will move the cursor within the cell.

To create a new cell, you have to be out of focus. You can do this by pressing a for above and b for below the current highlighted cell.

To delete a cell, you press dd, that's d twice. Remember that this only happens when you are not in focus, otherwise you would just be typing the letters dd into the cell.

To change a cell from 'code' to 'Markdown' you can either:

- Use the menu at the top,
- When the cell is not in focus you can press c for code, r for raw and m for Markdown.

1.3.3 How to write formulae in a cell

There are lots of times when it helps to show a formula in addition to some code snippets or claims. For example, here is a formula for getting the average (or specifically the 'arithmetic mean'):

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

This formula was not written with Markdown but with a special typesetting language called LaTeX. Technical papers are often drafted in LaTeX as are many books in STEM fields. It is less common in social sciences, but it is really handy. I wrote my dissertation in a combination of LaTeX and Markdown.

The formula was given its own line because it was enclosed with `$$` characters. Here is what the code looks like: `\ bar{x} = \frac{1}{n} \sum^{n}_{i=1}x_{i}`. If we enclose it with single `$` it will be a formula inline, like so: $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$. I use inline formulae for most numbers.

If you click on this cell, you can see the formatting underneath. Here we are just using MathJax, which is a subset of LaTeX used for formulae. StackExchange have a nice brisk tutorial of the syntax of MathJax. Basically, there is syntax for:

- Superscripts , `x^i` e^i, subscripts, `x_i` e_i,
- Fractions, with `\frac{NUMERATOR}{DENOMINATOR}`, as in $\frac{x^3}{y_i}$
- Parentheses (using `\left(` and `\right)` to scale properly) as in $\left(\frac{\sqrt{(\bar{x}-x_i)^2}}{n}\right)$

- Summation, product, and related symbols. `\sum` for $\sum$, and `\prod` for $\prod$.
- Greek symbols. Use their name for the symbol such as `\alpha` for α or `\omega` for ω.
- A host of diacritics, math symbols, and fonts. Check the tutorial above for clear examples.

1.3.4 The big Jupyter Gotcha

There are many advantages to running code in Jupyter but there are a few caveats. One in particular is really important to discuss right up front: You can run cells in any order even if you do not mean to. From this you can run into some pretty common issues.

1. Running code out of sequence. Imagine that in cell one I clean up some text (for example, I remove all the periods and commas). Then in cell two I run some code on that text (for example, make it ALL CAPS). Now imagine I then go back and do something else in cell one, such as change my code to remove apostrophes as well. So I run cell one again, but skip the second cell. Now my data is not in ALL CAPS and subsequent cells will not get the data they expected.

2. The "I've changed my variables to make them read better" issue. This one is my number one gotcha. As an example, I might change a variable name once I get my code working but want to make it more readable. But because Jupyter does not have a great 'find and replace' system, I might forget to change *all* of the instances of a variable. So if the variable was called `tl` but I want it to be `tweet_list`, then I replace the variable name. But what if I do not change it *everywhere*? There might still be a tl left in the code somewhere. Here's the gotcha: **Therefore, the program keeps running (since `tl` was already created)**. But the next time I restart the program or when you, the reader, try to run my Jupyter notebook, `tl` will not be created, `tweet_list` will. So the program will throw an error that any remaining `tl` is an unrecognised variable.

In general,

- Run your Jupyter cells in order, unless absolutely necessary.
- If you change a variable, be sure to change it **everywhere**.
- If you change a cell further up in your code, run every line afterwards.
- If you send notebooks to other people, then from the menu: "Kernel"→"Restart Kernel and Run All Cells…". If you get an error then debug it before you send the code to people.

Chapter 2

Primitive Data Types

2.1 Primitive Data Types in Python

There are two basic types of data in python, **primitive** data types and **object** data types. Primitives are the basic buiding blocks of more complex data structures, much like how letters are the building blocks of words and digits the building blocks of numbers. For example, each letter is a primitive data type in python called a **character**. An ordered list of characters is called a **string** object.

There are (as I understand it) five primitive types in Python:

- `int` for integer or whole numbers.
- `float` for floating point numbers. These are numbers with decimals in them.
- `char` for characters.
- `byte` for a kind of character interpreted by the computer, for example for storing image data.
- `bool` for Boolean, namely `True` or `False`

Usually you do not want to type out some primitive data every time. Instead you would use a label to represent them. This label is called a **variable**. You assign a value to a variable and then you can use that variable to 'represent' the value. See how this works below:

```
new_string = 'Hello world!'
print(new_string)
```

```
Hello world!
```

Variable names in Python start with an alphabetical character (`a-Z`) but can include numbers and underscores. There are some reserved words in Python. If you type them in Jupyter they usually show up in green. If you try to make a variable one of these (like saying `print = 4`), you might have some unexpected consequences.

Don't worry, you won't 'break Python', but you might mess up that specific instance of Python. In which case, you can always start again by restarting the kernel from up in the menu. So I encourage you to toy around with the code, get a feel for some errors or ways to tinker. Then if you feel it's messed up, you can always restart the kernel. Of course, as you get further into production or academic level code you will not so easily want to restart the kernel, but by then you will likely have developed other approaches to tinkering with your code.

2.2 Characters

The first primitive data type is the character. We don't really interact with characters directly but instead through their collection as a 'string' or `str` object. You saw above the string `Hello world!`.

Characters become a string when encased in quotes. There are three types of quotes: the single tick, the double quote, and the triple tick.

Quote 1. The single tick.

```
print('One small tick for strings')
```

Quote 2. The double quote. This is not two ticks, but the 'double-quote' character, ". It looks like two vertical ticks, but closer together than two ticks (e.g.," vs '') . Be careful with this character. Some programs like Microsoft Word like to replace the generic " with stylized quote characters that are different at the beginning and end of a quote such as "these". Python doesn't like stylized quotes and prefers the generic ".

```
print("Python doesn't mind the tick here")
```

The advantage of using double-quotes is that you can write phrases like "don't bring me down" without the program being confused when the string stops. If you then wrote `"society", man`, inside of a string, as in:

```
print("We all live in a "society", man")
```

Then it will get assume the string ends when the next double quote appears, which is not what was the intention.

Quote 3. The triple tick. This is indeed three ticks in a row. Python will evaluate everything inside of the three ticks literally. So if you want to have a string break across a line you can just type that in between three ticks and it will not throw an error.

```
print('''I have no problems breaking
across the lines!''')
```

```
print('One small tick for strings')

print() # this just prints an empty line.

print("Python doesn't mind the tick here")

print()

print('''I have no problems breaking
across the lines!''')
```

```
One small tick for strings

Python doesn't mind the tick here

I have no problems breaking
across the lines!
```

2.2.1 When syntax issues arise in special characters

So what happens if you want to use a quotation mark in your text? If you just insert a quote then Python will think it is the end of the string. It will then `raise` an error. Here is what an error, specifically a `SyntaxError`, looks like:

```
print("We all live in a "society", man")
```

```
  File "<ipython-input-85-d713ff32c4e4>", line 1
    print("We all live in a "society", man")
                             ^
SyntaxError: invalid syntax
```

Decoding errors is a bit of an art that will develop over time. Here, we can see indeed, the syntax is invalid. But Python is not great at explaining why. This is where experience comes in. Over time you will get better at deducing errors and cleaning them up.

Some pointers to help with errors:

- The bottom part is closest to your code. The error might have been triggered at many different layers of abstraction and so the output looks long and intimidating. But it really refers to the *line number* of a line that was in the process of running the code at that point and an indication of where the code was when the error was raised;
- Try to `print()` out at many points to get feedback (then remove these print statements from working code);
- Break the problem down: make the smallest possible changes and see if it affects the code;
- Examine online forums that received a similar error. But be mindful of what you throw into a search engine. Your variable names might be noise or might be research data. Be cautious of online sources, know that the most popular is not always the best (often it is biased by age of comment). Often Stack Overflow comments are popular because of *how* they explain the content. Students sometimes rush to paste a code snippet without understanding it, but the snippet does not work because it is an example with slightly different details than in the students' code. Be patient and read the explanation rather than immediately copy-paste-hope.

In the case of the error above, the program used a caret character (`^`) to indicate that there should not be an `s` directly after a closing quote. But that's not actually a closing quote is it? It's those darned inverted commas used by skeptical academics everywhere. So in order to preserve those inverted commas we need to "escape" them.

We use the backslash character to escape, so we should see a string that looks more like:

```
print("We all live in a \"society\", man")
```

Notice that the text colour is also a hint of when things are amiss. Observe the correctly formatted print statements below:

```
print("We all live in a \"society\", man, but at least we can escape the quotes.")
```

```
We all live in a "society", man, but at least we can escape the quotes.
```

So there are a number of 'special characters' that are escaped:

- `'`: To escape the single tick, as in `print('say it ain\'t so?')`.
- `"`: To escape the double quote, as in `print("Okay then, \"It ain't so\"")`.
- `\`: To escape the backslash in case you want to literally print it, as in `print('scanning C:\\temp folder')`.

Not all characters are visible.

Sometimes we want to add some spaces to our code, maybe a tab or maybe a new line. Up until now we have just used `print()` to add an extra line. We can, however, do that right in the text. These sorts of characters are called 'whitespace' characters. There's two particularly relevant ones:

- `\n` is the new line character;

- `\t` prints a tab. This is nice when you're printing tables as it counts spaces from the left-hand side and moves in multiples of 4 or 8 (depending on settings). Below I demonstrate the use of tab characters in a Haiku.

See newline and tab in action below:

```
print("\"A \\n\" (A New Line)")
print("By Bernie Hogan\n")
print("Autumn students learn:")
print("\"A tab that provides \tstructure")
print("\t\t\tmay not provide space\"")
```

```
"A \n" (A New Line)
By Bernie Hogan

Autumn students learn:
"A tab that provides    structure
                        may not provide space"
```

We will continue to explore features of characters when we return to collections, since a string is a collection of characters.

2.3 `float` and `int` as the two basic types of numbers

For numbers, there are two basic primitive data types:

- **Integers**, which refer to whole numbers such as 1, 42 or 1812;
- **Floating point numbers**, which refer to real numbers that can be approximated by digits using a decimal point, such as 0.5, 12.345 and 0.333333333.

```
# An integer
x = 7

# A floating point number. Still a whole number, but the .0 makes it a float rather␣
 ↪than an integer.
y = 4.0

print ( type(x) )
print ( type(y) )

z = x + y

# See how z inherits the floating point number even though the value could be an␣
 ↪integer?
print (type(z), z)
```

```
<class 'int'>
<class 'float'>
<class 'float'> 11.0
```

2.3.1 Casting numbers

Above I printed `type(<some_object>)`. The result of `type(x)` was `<class 'int'>` and the result of `type(y)` was `<class 'float'>`. But when we add these two together, we have to have comparable primitive data types, so the integer gets recast as a float.

Data have a type, like `float` or `int` but we can transform data from one type to another by "casting" it. This does not always work as planned. But if it does not, then we have learned a valuable lesson and should seek a different way to convert the data if possible.

If you convert an integer `7` to a float, that's easy. It's going to look the same, except it will have a decimal point and be ready to receive floating point precision, such as `7.0`. If you try to convert the string `"7.0"` to a float it should also work. But if you try to convert `"seven"` to a float the program will thrown an error.

Below are some casting operations that work. Generally, it should be easy to find a way to turn a number into a string. It just ends up being that number, but as characters. Turning a string into a number on the other hand is really challenging given all the possible ways a number can be written in different languages, different spelling,etc. It would require some real understanding and is beyond the scope of simple Python operations. Below I will cast some ~~spells~~ data in different classes.

```
# Start and end with Int:
var_int = 7
print(var_int, type(var_int))

var_float = float(var_int)
print(var_float, type(var_float))

var_str = str(var_float)
print(var_str, type(var_str))
```

```
7 <class 'int'>
7.0 <class 'float'>
7.0 <class 'str'>
```

So far, so good. We went from an int `7`→float `7.0`→str `"7.0"`. But now we have a problem if we want to go back to int. It will throw a `ValueError`. If you run it, then it will state `ValueError: invalid literal for int() with base 10: '7.0'`. The problem here is that you might think that `.0` means nothing. But it's actually something out of nothing. Python is not fussy that the value after the decimal point is zero or some long string of digits. The matter is that it is something after the decimal and that particular conversion does not like it.

```
var_int = int(var_str)
print(var_int, type(var_int))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-3-3af91342de95> in <module>
----> 1 var_int = int(var_str)
      2 print(var_int, type(var_int))

ValueError: invalid literal for int() with base 10: '7.0'
```

It is often said that a language which requires you to specify the class of a variable ahead of time is a "strongly cast" language. For example, in Java you cannot write `x=5` unless you have previously defined `x` as an integer like `int x = 5`.

Python is a "weakly cast" language meaning that it does not check the data type before assigning an object to that variable. So in one line you could code `x=0`, making `x` an integer number. On the next line, you can code `x="fabulous"` and Python does not have a problem with that.

2.3.2 Basic number operations.

You will remember some basic number operations from arithmetic, such as addition and subtraction. Python implements these and a few others worth remembering. Let's have a look at several of these. You should pay attention to whether the result includes digits after the decimal point or not. We can do operations on both integers and floating points. When in doubt Python uses the type of number that gives more precision. So `1 + 2.5` will not round up or down, it will return `3.5`. Here is a list of common number operations:

- Addition: `X + Y`
- Subtraction: `X - Y`
- Multiplication: `X * Y`
- Exponent (i.e., raising X to the power of Y): `X ** Y`
- Floating point division: `X / Y`
- Integer division: `X // Y`
- Modulo (i.e., the remainder from integer division): `X % Y`

```
x = 9
y = 4
print("x = ",x)
print("y = ", y)
print("x + y = ", x + y)
print("x - y = ", x - y)
print("x * y = ", x * y)
print("x ** y = ", x ** y)
print("x / y = ", x / y)
print("x // y = ", x // y)
print("x % y = ", x % y)
```

```
x =  9
y =  4
x + y =  13
x - y =  5
x * y =  36
x ** y =  6561
x / y =  2.25
x // y =  2
x % y =  1
```

2.3.3 Floating point numbers mean floating point precision

Floating point numbers are finite. Some numbers are not. For example, if you divide 1 by 3 the answer will be approximated by `0.33333` but really we would say that it goes on infinitely. So there is a limited level of precision using these primitive data types. Astrophysicists and others who need obscenely large levels of precision might find that floating point precision is not enough. For them there are specialist methods. You might never encounter the limits of floating point precision. Or rather, not until now.

Observe this quirk below about what happens when you add some really large numbers together in Python.

```
# Example 1
x1 = 1/3
y1 = 1/3
z1 = 1/3

print("x1 is ", x1)
print("y1 is ", y1)
```

```
print("z1 is ", z1)
print()

# Shouldn't this add up to 0.99999999999?
print("Then why is x + y + z = ", x1 + y1 + z1)
print("and not 0.9999999999999999?")
```

```
x1 is  0.3333333333333333
y1 is  0.3333333333333333
z1 is  0.3333333333333333

Then why is x + y + z =  1.0
and not 0.9999999999999999?
```

```
# Example 2
x2  = 0.333333333333333  # <- Notice one digit short
x2a = 0.3333333333333333
y2  = 0.6666666666666666

print("If x2 + y2 is 0.9999999999999996:\n", x2 + y2)

print("Will x2a + y2 be 0.9999999999999999:\n", x2a + y2)
```

```
If x2 + y2 is 0.9999999999999996:
 0.9999999999999996
Will x2a + y2 be 0.9999999999999999:
 1.0
```

```
# Example 3
x3 = 16/9
y3 = 7/9
print("Won't 16/9 minus 7/9 equal 9/9, which is the same as 1?\n",x-y)
```

```
Won't 16/9 minus 7/9 equal 9/9, which is the same as 1?
 0.9999999999999999
```

This is why we say floating point numbers are approximations of real numbers. π is a real number, but it is also infinitely non-repeating. The computer then cannot load the full number of π (as it does not have infinite memory), but it loads in an approximation.

When we calculate things in Python, we are accepting a certain loss of precision. It does not do fractional math. In `Example 3`, the variable `x3` was first calculated as `1.7777777777777777` and stored as such.

We rarely encounter that level of precision, but I think its nice to get a sense of our limits.

2.4 From characters to strings

Strings, as we have now seen, are collections of characters. So we can now tell that:

`"This is a string"`

There is a huge amount of work in programming that is really just dealing with strings in some form or another. Strings become the basis of more complex file types, like `csv` for tables, or `xml` for data. Getting good at programming for computational social science will often mean being confident in transforming strings from one form to another and being able to select part of a string as having some sort of pattern or structure.

Whether it is text collected via communications, surveys, comments, or reviews, it will be a string and it will probably need formatting. We might want to detect the presence of words or remove certain characters because they do not print well.

2.4.1 Some example string methods

To use these you would have a string variable. I let `<var>` stand in for that here.

- **`<var>.upper()`**: This returns a version of the string in all upper case.
- **`<var>.lower()`**: This returns a version of the string string in all lower case.
- **`<var>.find(<substring>)`**: This returns the numerical index of the first complete mention of the substring. This returns `-1` if the substring is not found.
- **`<var>.isalpha()`**: This returns `True` if the string is all alphabetical characters and `False` otherwise.
- **`<var>.replace(<old_string>,<new_string>)`**: This takes additional two arguments, which is what you want to find and which is what you want to replace it with.
- **`<var>.strip()`**: This is a useful command to remove whitespace from the beginning and the end of a string. To remove only from the beginning use `<var>.lstrip()`. To remove only from the right, use `<var>.rstrip()`.

If you want to find out details about a method, you can check the help. There are several ways to do that in Jupyter. The first is to run **`help(<var>.<method>)`**. But don't include the `()` at the end of the method or it will first run the method and then query what was returned for help.

```
help("str".find)
```

```
Help on built-in function find:

find(…) method of builtins.str instance
    S.find(sub[, start[, end]]) -> int

    Return the lowest index in S where substring sub is found,
    such that sub is contained within S[start:end].  Optional
    arguments start and end are interpreted as in slice notation.

    Return -1 on failure.
```

The second way in Jupyter is to create a new tab (via the Launcher) and instead of selecting a notebook or Terminal, notice that in the lower right corner is a type of file called "Show contextual help". This will give realtime help on what command you are using. It's a second tab so drag it around Jupyter lab until you find a spot where it is visible but not obtrusive. Finally if you are in a code cell and you place your cursor inside a method and hit *shift→tab* it should bring up the help for that variable as a tooltip.

2.4.2 How to use a string method

Methods and functions are the 'verbs' of Python. A method is basically a function except you invoke a method on an object. For now, it's okay to use the terms interchangably, but the difference will be clear (and important) when we start building our own functions later.

So if we have a string object, we can attach methods directly to the string:

```
"This is an object"
```

And we can attach the `upper()` method like so:

```
"This is an object".upper()
```

It will then print: ~~~ python > THIS IS AN OBJECT ~~~

But more commonly we first assign a string to a variable then then use our method on the variable.

```
new_string = "This is a new string"
print ( new_string.upper() )
> THIS IS A NEW STRING
```

Try it below:

```
example_string = "The quick brown fox jumps over the lazy dog"

print("The original string:",example_string)

print("\nTo upper case:",example_string.upper())

print("\nTo lower case:",example_string.lower())

print("\nIs the string just alphabetic characters?",
      example_string.isalpha())

print("\nReplacing 'o' with 'you':",
      example_string.replace("o","you"))
```

```
The original string: The quick brown fox jumps over the lazy dog

To upper case: THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG

To lower case: the quick brown fox jumps over the lazy dog

Is the string just alphabetic characters? False

Replacing 'o' with 'you': The quick bryouwn fyoux jumps youver the lazy dyoug
```

2.4.3 Strings are really a special kind of a `list`.

We will learn more about lists in the next chapter. But strings are just a special kind of list - one with only characters in the same encoding. This means we can do things with strings like we can with a list, like sort the strings, or ask for elements 3 through 10. Above we acted on the string as a collection of characters, but we can convert it to a list in two ways. Observe the difference between them:

```
new_str = "An example string to use"

print("1. The string itself:\n",new_str)
print("2. The string split by spaces:\n",new_str.split())
print("3. The string recast as a list:\n",list(new_str))
```

```
1. The string itself:
 An example string to use
2. The string split by spaces:
 ['An', 'example', 'string', 'to', 'use']
3. The string recast as a list:
 ['A', 'n', ' ', 'e', 'x', 'a', 'm', 'p', 'l', 'e', ' ', 's', 't', 'r', 'i',
'n', 'g', ' ', 't', 'o', ' ', 'u', 's', 'e']
```

The method `new_str.split()` uses a single empty space character, , by default. So then the program split

there. You can split by a word or combination of characters. Also, while the program splits every instance of that pattern by default, you can ask it to split only once or n times. See below:

```
new_str2 = "Singing, dancing, grooving"

print("1. The string itself:\n", new_str2)
print("2. Splitting at ever 'ing' value:\n", new_str2.split("ing"))
print("3. Replacing only the first instance:\n", new_str2.split("ing",1))
```

```
1. The string itself:
 Singing, dancing, grooving
2. Splitting at ever 'ing' value:
 ['S', '', ', danc', ', groov', '']
3. Replacing only the first instance:
 ['S', 'ing, dancing, grooving']
```

The splitting on `ing` produced 5 elements: `'S'`, `''`, `', danc'`, `', groov'`, and `''`. Two of these are empty. The first one is empty because we used `Singing` and so it wanted to split in between the two `ing`s, even though there was nothing between them. Similarly at the end it wanted to split between the final `ing` (from `grooving`) and the end of the string, so it produced a second empty string. In the last example we only splint at the `S` so that we have two elements, `S`, and the text from the other side of the first `ing`: `'ing, dancing, grooving'`.

`replace`, like `split` has the same feature of being able to select multiple or single instances.

```
new_str2 = "Singing, dancing, grooving"

print("1. The string itself:\n", new_str2)
print("2. Replacing all instances with ***:\n", new_str2.replace("ing","***"))
print("3. Replacing only the first instance with ***:\n", new_str2.
 ↪replace("ing","***",1))
```

```
1. The string itself:
 Singing, dancing, grooving
2. Replacing all instances with ***:
 S******, danc***, groov***
3. Replacing only the first instance with ***:
 S***ing, dancing, grooving
```

The second way we turned the string into a list was to take every character and make it its own element in a list. So that's why it displayed as `['A', 'n', ' ', 'e', ...`. In a sense, this is similar to a string in that it has just as many elements. But when you print the list versus the string, you can see that a list is geared towards thinking of the characters as elements in a collection, whereas the string is geared towards thinking of the characters as constituting a single "string" object.

2.5 Combining strings

String formatting is complicated enough that it warrants its own section in this chapter. This will only be a cursory look at string formatting, but it will demonstrate the power of this approach. Usually we want to format strings because we have some variable that we want to include. So for example, if we want to print a greeting based on the day, we would make a generic greeting and then have a place to insert the day.

Below we will insert some numbers into a string. You will notice that there are ways of formatting these numbers. They get a bit tricky and so here I'm just giving the basic syntax.

2.5.1 String concatenation

There are many ways to format a string but they tend to refer to either string concatenation or string insertion. To concatenate is to bring together. So if you have `"blue"` and `"berry"` you can concatenate them with `"blue" + "berry"`. This means that the plus symbol is **overloaded**. It means different things in different contexts. Watch us use the plus symbol to concatenate a string as well as add some numbers below:

```
print("blue" + "berry")
print(7 + 9)
print("7" + "9")
```

```
blueberry
16
79
```

So if you have a variable, `name` and you want to insert it into a greeting, you can concatenate with `print("Hello " + name)`.

2.5.2 Combining strings with f-insertions

One of the challenges with concatenation is how to deal with variables of different kinds. Like imagine you calculate a test score and then want to print the score and a string. Since the `+` symbol is overloaded it will not like trying to determine if it should add some numbers or concatenate some strings. See the error below:

```
score = 17

print("Your score was " + score + "out of 30, or " + score/30 + "percent")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-184-3c2f3578b68f> in <module>
      2
      3
----> 4 print("Your score was " + score + "out of 30, or " + score/30 + "percent")

TypeError: can only concatenate str (not "int") to str
```

So we can format the number as a string ahead of time, but that would be unnecessary. Instead we can format the string itself. There's the classic way of doing this using the `format()` method. Then there's the new way (which I adore) using f-insertions. Let's see them both since you will encounter both in the wild.

In both cases they use `{}` inside of string quotes to create a marker for where the variable should go in the string. So for the example above it would be like: `"Your score was {} out of 30, or {}."`. See below (you will see why I left out "percent" in a bit):

```
score = 17

print("Your score was {} out of 30 or {}.".format(score, score/30))
print(f"Your score was {score} out of 30 or {score/30}.")
```

```
Your score was 17 out of 30 or 0.5666666666666667.
Your score was 17 out of 30 or 0.5666666666666667.
```

Did you see how with the second approach, we inserted the variable right inside where it was supposed to be. It also made the colour of the variable stand out. In the second approach we had a different kind of string, called an f-insertion. This is like a regular string except we put `f` before the quote. It then knows that `{}` inside of a string means a variable is coming.

Tips for f-insertions

Tip 1. What if you want to print `{` literally inside of an f-insertion? You escape it like with other escape codes, by using \.

Tip 2. If you are using a dictionary inside your f-insertion, like `f"I like eating {food['sweet']}."` the dictionary should use different quotes to the statement. Here I used a single tick inside double quotes.

2.5.3 Formatting strings nicely

You might have seen how the result of 17/30 was very long and a number between 0 and 1 rather than a percentage? We can fix these things. After the varaible name inside te `{}` we can put some formatting codes. I rarely remember them all. Instead, I tend to look to pyformat.info which has clear examples of these. But here are two important ones:

```
print(f"Out of 30 with two significant digits: {score/30:0.2f}")
```

```
The score out of 30 with two significant digits: 0.57
```

```
print(f"Out of 30 as percent with one significant digit: {score/30:0.1%}")
```

```
Out of 30 as percent with one significant digit: 56.7%
```

In the first instance we used the code `0.2f` to mean `0` padding at the front, two significant digits after the decimal, and the number is a `float`. The second instance `0.1%` changed it to one significant digit, but understood the number as a percent so it was `56.7` rather than `0.57`. When printing the results of analysis, Python will report as much precision as it has, but that might make the number look noisy or hard to grasp. Reporting a meaningful level of significant digits thus makes it easier for people to see the number for what it means.

2.6 Conclusion

In this chapter we went from the most primitive data types (`int`, `float`, `char`) towards more meaningful data types. I showed how to convert data types, how to consider errors, and how to do some string manipulation. These get more interesting when we have many strings, numbers, or calculations. Then we can put these in a collection and start to ask questions of the collection itself. That is where we are headed next.

Chapter 3

Collections

3.1 Collections

Python has many ways in which to organise a collection of data. Computer scientists often search for interesting ways to collect and structure data. Some reasons are to make its storage and retrieval more robust and efficient. Later on you will find some structures work better than others for a task at hand.

Here we are going to focus on three main kinds of collections which are extremely common and impressively versatile: `list`, `set`, and dictionary (or `dict`). Getting used to the logic of these three will enable you to manage a huge amount situations in programming. Other, fancier, and more abstract packages will come along for specific tasks, but these three are the most important.

The thing about collections that matters most is how they associate data. Each type of collection may associate data in a different way. You might have guessed from the discussion so far that lists place data in some ordered sequence. We would say that a list orders `by position`. A set, by contrast, does not really have a notion of position. There's not element guaranteed to be first or last when you print a set. Instead, a set is ordered `by inclusion`. This means that some object or data is either in a set or not in a set. Finally a dictionary is ordered `by key`. Keys are like an index for more data. We will show what keys mean after covering lists and sets more fully first.

Virtually all collections in Python are iterable. This means that you can ask for one item from the set and then keep asking until you get all the items. Sometimes the order might be different depending on the type of collection, but you will definitely get through all of them eventually. We will iterate through collections in the next chapter.

3.2 Ordered by position: The `list`

Lists are ordered collections of objects. They, like most British buildings, start at 0. The next element is 1 and so forth.

Lists use square brackets as ends. So a list looks like:

```
list_example = ["apples","bananas","cucumbers","durians"]
print(list_example)
```

```
['apples', 'bananas', 'cucumbers', 'durians']
```

And to return a single element, we can count its position and use that as an index, like so:

```
print(list_example[1])
```

```
bananas
```

We can also count backwards through the list using negative numbers, like so:

```
print(list_example[-1])
```

```
durians
```

Lists have a **range** that goes from 0 to $n-1$ where n is the number of elements. If you try to index an element that's out of the range, python will throw an error. To get the range, we can use a function called `len` which is short for **length**.

Notice what we do below. We get the length, then use that as a variable. It will give us an error, but length minus one won't.

```
list_length = len(list_example)

print("The list has",list_length,"elements")
```

```
The list has 4 elements
```

```
# This should give an InderError
print(list_example[list_length])
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-142-325d5d69b80d> in <module>
      1 # This should give an InderError
----> 2 print(list_example[list_length])

IndexError: list index out of range
```

```
# This should work just fine
print(list_example[list_length -1])
```

```
durians
```

3.2.1 List indexing and slicing

We do not simply index just one element of a list at a time. We can actually index entire chunks of elements at once using the `:` inside of the `[]`. For example, with a four element list we can print elements zero through two:

```
list_example = ["apples","bananas","cucumbers","durians"]
print( list_example[0:2] )
print( list_example[2:])
```

```
['apples', 'bananas']
['cucumbers', 'durians']
```

The first number is the list index for the element we start. The second number is the index that we get up to,**but not including**. This way, `list_example[0:2]` and `list_example[2:4]` will not return overlapping lists.

Yet, we can also slice lists **from the end** rather than the front of the list by using negative numbers. -1 is the final element, and this happy one `:-1` will give us everything up to the final element.

```
list_example = ["apples","bananas","cucumbers","durians"]
print( list_example[:-1] )

print( list_example[1:-1] )
print( list_example[:])
```

```
['apples', 'bananas', 'cucumbers']
['bananas', 'cucumbers']
['apples', 'bananas', 'cucumbers', 'durians']
```

3.2.2 Adding data to a list (and adding two lists together)

Lists are **mutable** meaning that they can be changed by the program. Other data structures we will encounter later are **immutable** which means that we can only create or destroy them but not change them. To change a list with methods, we can:

- Add things:
  - One at a time: **Append** an item to the end of the list with `list.append(item)`
  - Adding any collection: **extend** a list with the items from any **iterable** collection with `list1.extend(list2)`.
- Remove things:
  - Remove everything with **clear**: Want to keep the name but empty the list? `list1.clear()`
  - Removing one item by index: **pop** a value from the list. By default it is the final element, but you can pass an index to pop an element anywhere in the list. `list.pop()` or `list.pop(4)`. This is the same as **del** before the list, like `__del list1[-1]` or `del list[4]` except that deleting doesn't return the things you deleted whereas pop does.
  - Remove an item by its value: **remove** the first instance of a value in the list with `list.remove("item")`.
- Sort things:
  - **sort** the list in place with `list.sort()`.

Returning versus changing data

Most of the time when we call a method we expect it to return something. A method called `upper()` returns the string in upper case, for example. It might be confusing (it sure has been to me at times), but sometimes methods just alter the state of an object rather than returning a new object. The list methods mentioned above all change the list in place. So you would not say:

`list1 = list1.extend(list2)`

Because while `extend()` changes `list1` it does not *return* `list1`. You can test this below by printing what is returned from the list.

```
# Attempt 1. Extending a list - it returns none.
list1 = [1,4,9]
list2 = [4,5,6]

print(list1.extend(list2))

# Attempt 2. Extending a list - print the old, freshly extended list.
list1 = [1,4,9]
list2 = [4,5,6]
```

```
list1.extend(list2)
print(list1)
```

```
None
[1, 4, 9, 4, 5, 6]
```

Here are the other list methods in action. Each time I start with a fresh `list1 = [1,2,3]`.

```
# Append
list1 = [1,4,9]
new_val = 70
print("Original:\t",list1)
list1.append(new_val)
print("Appended:\t",list1)
```

```
Original:       [1, 4, 9]
Appended:       [1, 4, 9, 70]
```

```
# Extend
list1 = [1,4,9]
list2 = [4,5,6]
print("Original:\t",list1)
list1.extend(list2)
print("Extended:\t",list1)
```

```
Original:       [1, 4, 9]
Extended:       [1, 4, 9, 4, 5, 6]
```

```
# Clear
list1 = [1,4,9]
print("Original:\t",list1)
list1.clear()
print("Cleared:\t",list1)
```

```
Original:       [1, 4, 9]
Cleared:        []
```

```
# Pop. Notice that we are popping by position not value.
# The value in 1 position is the number 4.
list1 = [1,4,9]
print("Original:\t\t",list1)
x = list1.pop(1)
print("Popped (index 1):\t",list1)
print("Popped value:\t\t",x)
```

```
Original:               [1, 4, 9]
Popped (index 1):       [1, 9]
Popped value:           4
```

```
# Del
list1 = [0,54,31,5,77,-3]
print("Original:\t\t",list1)
del list1[-2:]
print("Deleted last two:\t",list1)
```

```
Original:                [0, 54, 31, 5, 77, -3]
Deleted last two:        [0, 54, 31, 5]
```

```
# Remove
list1 = [10,20,30]
print("Original:\t",list1)
list1.remove(20)
print("Removed '20':\t",list1)
```

```
Original:        [10, 20, 30]
Removed '20':    [10, 30]
```

```
# Sort
list3 = [7,3,1,2,3,4]
print("Original:\t",list3)
list3.sort()
print("Sorted:\t\t",list3)
```

```
Original:        [7, 3, 1, 2, 3, 4]
Sorted:          [1, 2, 3, 3, 4, 7]
```

3.2.3 A list versus a tuple

If you have some elements inside of square brakets `[]` this is usually a list or a list-like object. But sometimes you'll see elements inside of a slightly different structure with parentheses `()`. This is called a tuple. It tends to work the same as a list, whereby you can index by position, but unlike a list you cannot change the contents of a tuple, you can only create a tuple or destroy it.

One neat thing about a tuple is that you can split it up when you are working with it. So watch below how I create a tuple with two elements, and then I assign them each to a variable.

```
xy_tup = (-1,3)
x,y = xy_tup

print(xy_tup)
print(x)
print(y)
```

```
(-1, 3)
-1
3
```

This will be handy later when we get sent tuples and we really want one part of the tuple or another.

3.3 Ordered by inclusion: the `set`

A `set` is a data structure that contains only unique values. So if you have a list like so: `ex1 = ["Spain","France","Spain","Italy","Italy"]` and you convert it to a set `set(ex1)`, it will only be the following: `{"Spain","France","Italy"}`. If you start with an empty set you can add more elements to it. But if you add an element that was already in the set nothing changes.

```
s1 = set()
s1.add("Cherry")
s1.add("Lemon")
```

```
print(s1)
s1.add("Cherry")
print(s1)
```

```
{'Cherry', 'Lemon'}
{'Cherry', 'Lemon'}
```

```
s2 = [1,2,2,3,4,5,5,5,5,5,6]
print("As a list:\t",s2)
print("As a set:\t",set(s2))
```

```
As a list:       [1, 2, 2, 3, 4, 5, 5, 5, 5, 5, 6]
As a set:        {1, 2, 3, 4, 5, 6}
```

3.3.1 Set inclusion and efficiency in code

With a set, we can and usually do as about set inclusion. For that we just as if an element is `in` a set. So in our set of flavours above, we included Lemon but not Pineapple. See below how we would do that in Python:

```
print("Lemon" in s1)
print("Pineapple" in s1)
```

```
True
False
```

Granted you can often ask if `<element> in <collection>` for lots of collections. But it turns out that there are a multitude of ways of organising data in the computer. A set is faster than a list because of the way it maps data to memory. A `frozenset` is faster still. It is like a set but cannot be altered (i.e. it's frozen).

```
speed_list = [1,2,3,4,5,6,7,8,9]
speed_set = set(speed_list)

%timeit -n 10000 1 in speed_list
%timeit -n 10000 9 in speed_list

%timeit -n 10000 1 in speed_set
%timeit -n 10000 9 in speed_set
```

```
83.8 ns ± 27.1 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
205 ns ± 13.7 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
54 ns ± 0.518 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
53.5 ns ± 0.344 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

We used a "magic command" here for the first time. This is `%timeit`. It's called "magic" because it does something in Jupyter but not necessarily in Python generally. You will see a few of these peppered throughout code. In this case we said "do what's on this line 10000 times and report the average time". Why 10,000? Because I used the `-n` flag for number of times and set it to `10000`. You can remove that but the program will probably do it a few million times and be a bit slower.

The first two are using the list. It took ~100 nanoseconds on my machine to look for the first element. Then to get the last item in the list it took almost double the time (and that's with only nine elements!). That's because a list looks one by one through the data structure. With a set, it has a way of checking if there is a value at the memory address for `1` and it is either there or it is not. So that's why it took about 50 nanosecond for the first element or the last element (or any element regardless of set size).

This huge difference makes a difference to our programming. Sometimes we want to search through a list to find the element, but most of the time we just want the element in the fastest way possible.

3.3.2 Set logic: Union and Intersection

Since sets contain at most one copy of any object, we can compare them and do set operations. These might be familiar to you through 'Venn diagrams'.

Below are three key set operations that are often characterised using Venn diagrams:

- `<set1>.union(<set2>)`: This looks for all elements from both sets.
- `<set1>.intersection(<set2>)`: This looks for all elements in common.
- `<set1> - <set2>`: This removes all elements from `<set1>` that were in `<set2>`. If there were additional elements in `<set2>` these are not considered.

```
setCount = {1,2,3,4,5} # the first five numbers

setOdd = {1,3,5,7,9} # the first five odd numbers

print(f"setCount:\t{setCount}")
print(f"setOdd:  \t{setOdd}")
print()

print("Union (all of the elements from both):\t\t", setOdd.union(setCount))
print("Intersection (all elements in common):\t\t",
      setOdd.intersection(setCount))

print("Set subtraction (setCount minus setOdd):\t", setCount - setOdd)
print("Set subtraction (setOdd minus setCount):\t", setOdd - setCount)
```

```
setCount:       {1, 2, 3, 4, 5}
setOdd:         {1, 3, 5, 7, 9}

Union (all of the elements from both):          {1, 2, 3, 4, 5, 7, 9}
Intersection (all elements in common):          {1, 3, 5}
Set subtraction (setCount minus setOdd):        {2, 4}
Set subtraction (setOdd minus setCount):        {9, 7}
```

If you have a list that you want to include in a set, you can do that with `<set1>.update(<list>)`. Then all the elements in the list will be considered. If they were not in the set before, they are now.

```
set3 = {"Cherry","Lemon","Orange","Grape"}
new_flavor_list = ["Orange","Pineapple","Sarsaparilla"]
set3.update(new_flavor_list)
print(set3)
```

```
{'Sarsaparilla', 'Grape', 'Lemon', 'Orange', 'Cherry', 'Pineapple'}
```

Notice that the order in the new set is not necessarily the order in which we added these elements. The order in a set comes some underlying Python operations about memory management. But recall, we are not using a set because of its order. We are using it to determine whether something is 'included' or not.

3.4 Ordered by key: the dictionary or `dict`

A set only has one copy of each element such as `{"breakfast","lunch","dinner"}`. What's advantageous about a set is that you can quickly check for whether an element is included. This is pretty handy if we

want to store some large and complex objects. We should be able to create some sort of key and then ask the computer "if I give you the key will you give me the large and complex object?"

In Python an example of this is a **dictionary**. Like a real dictionary, each word is only present once, but that word can refer to many different definitions (for example, the word `run` has hundreds of definitions).

In a Python dictionary we might have `{"breakfast":"porridge"}`. Then instead of calling `"breakfast"` one element of a set, we call it one of the **keys** of the dictionary. Then in this example, `"porridge"` is a **value**. But a value could be any object. It could be a list: `"breakfast":["eggs","toast","porridge"]` or even another dictionary! For example:

```
{"breakfast":
     {"eggs":2,
      "bread":"toasted",
      "porridge":"classic"
     }
}
```

See below how I create and then query a dictionary.

```
food_dict = {"breakfast":"porridge",
             "lunch":"pizza",
             "dinner":"stir fry"}

print(food_dict["lunch"])
```

```
pizza
```

3.4.1 Checking in on syntax with indexers and sets

So at this point you might be curious about some syntax issues. If it's a dictionary then why would we say `new_dict['lunch']` and not `new_dict{'lunch'}`? The reason is that the square brackets are what's called an 'indexer'. We use an indexer to look up some value. In a list we look up that value by position, e.g., `list1[3]`. In a dictionary we look up that value *by key*. But we still use `[]` to look it up. You will see more complex indexers for getting rows and columns in tables if you continue on towards data science in Python and use the `pandas` package, but they still are denoted by brackets (`[]`).

On the other hand, did you notice above that when we printed a set, it printed it with curly braces? And when we define a dictionary we also did it with curly braces? E.g.,

```
print(setCount)
> {1, 2, 3, 4, 5}
```

That's because both dictionaries and sets have unique elements. For the set it's the elements, for the dictionary it's the keys. You cannot have two keys in a dictionary with the same value. Watch what happens below when we create a dictionary with two keys that are the same:

```
error_dict = {"breakfast": "porridge",
              "breakfast": "fruit"}

print(error_dict)
print(error_dict["breakfast"])
```

As a small programming aside, you will notice by now that not all Python commands need to be on the same line. There are a few places where you can 'naturally' break a line and not confuse the Python interpreter. One is after a comma. I tend to want my code to by shorter than 80 characters wide, so I use the comma to create breaks and keep the code readable.

3.4.2 Accessing a dictionary's components.

With a dictionary, you might not simply want to query the key and get the value. You might want all the values, all the keys, or all the items (i.e. key:value pairs). These are all available through methods. Here are the methods:

- **keys**: `<dict>.keys()`;
- **values**: `<dict>.values()`;
- **items** (key-value pairs): `<dict>.items()`.

Below I will demonstrate each of these. Pay attention not just to what is printed, but the *type* of what is printed. You'll see that it looks like a list, but behaves slightly differently.

```
ex1_dict = {"fish":"salmon",
            "mushroom":"enoki"}

ex1_dict["fruit"] = "apple"

print(ex1_dict)

keys = ex1_dict.keys()
vals = ex1_dict.values()
items = ex1_dict.items()

print(type(keys), keys, sep="\n")
print(type(vals), vals, sep="\n")
print(type(items), items, sep="\n")
```

```
{'fish': 'salmon', 'mushroom': 'enoki', 'fruit': 'apple'}
<class 'dict_keys'>
dict_keys(['fish', 'mushroom', 'fruit'])
<class 'dict_values'>
dict_values(['salmon', 'enoki', 'apple'])
<class 'dict_items'>
dict_items([('fish', 'salmon'), ('mushroom', 'enoki'), ('fruit', 'apple')])
```

It turns out that these resulting objects look like a list, they are a sequence of elements encased in `[]`, afterall. But they are then wrapped inside some label, like (`dict_keys()`). If you need to, you can turn them into a list by simply casting them as one. For example, these objects are not "scriptable", which translates to "you cannot use an indexer on them". Watch below how I try to query for `items[0]` and it throws an error. but when I cast it as a list, it works no problem.

```
items[0]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-23-5dbb89d7bbf9> in <module>
----> 1 items[0]

TypeError: 'dict_items' object is not subscriptable
```

```
list(items)[0]
```

```
('fish', 'salmon')
```

Also notice that in the `items` collection, the keys and values are represented like `('fish','salmon')`. This is the infamous tuple I mentioned above. You can't change this value, but you can iterate through these. Iterating will be featured in the next chapter.

3.4.3 Dictionary gotchas

The dictionary is an essential data structure in Python, but it is not without its quirks. Recall above that you can only have one key in a dictionary. So if you have `{"fish":"salmon"}` and want to add `"cod"`, you have to decide what to do with the salmon first. If you want to *replace* the value, then this will work:

```
food_dict = {"fish":"salmon"}
food_dict["fish"] = "cod"
food_dict
```

```
{'fish': 'cod'}
```

However, if you want to add cod, then clear that strategy would not work. You also cannot do `{"fish":"salmon", "fish":"cod"}` as we established above with our breakfast dictionary. So what can you do? You can create a new list and make that the value. Here is one approach":

```
food_dict = {"fish":["salmon"]}
food_dict["fish"].append("cod")
food_dict
```

```
{'fish': ['salmon', 'cod']}
```

But in case you did not have `"salmon"` in a list in the first place, you can always just insert it into a list:

```
food_dict = {"fish":"salmon"}
food_dict["fish"] = [food_dict["fish"],"cod"]
food_dict
```

```
{'fish': ['salmon', 'cod']}
```

This last one might have tripped you up. But remember that `food_dict["fish"]` at the beginning of line 2 *was* `"salmon"`, so that way we inserted it as the first entry in a list, with `"cod"` as the second entry. Then we assigned this list (`["salmon","cod"]`) to the value of `"fish"` in the dictionary.

3.5 Conclusion

Herein we have seen a number of collections. We focused on three main ones, `list`, `set`, and `dict`. Yet, in the course of working through these, we also encountered tuples, `dict_items`, `dict_values`, and `dict_keys`. Yet, these other collections tend to work in pretty similar ways. Ultimately, if we have a collection we need a way to access the data from that collection. That's why I chose `list`, `set`, and `dict`. They exemplify three key ways of ordering (and thus querying) for data: by position, by inclusion, and by key.

In the next chapter we will make use of collections through iteration. We will be able to do something for each element in a collection, or only under certain conditions.

The exercises in Appendix A enable some practice with dictionaries, and particularly some objects that are a messy of nested lists and dictionaries. This will start to resemble some of the messy data structures we see in the real world, like JSON (which really resembles combinations of lists and dictionaries).

Chapter 4

Controlling Flow: Conditionals, Loops, and Errors

4.1 Using conditionals to change the flow of a program

One way to manage the flow of data through an algorithm is to use conditionals. This means that we will evaluate some data. If the evaluation is true then we can do something. If the evaluation is false we can do something else. Now in life there are many sides to every store. But in Python we tend to work with strict conditions that are either fulfilled or not. For example we would have a set of things, `characters = {"Kermit","Piggy","Fozzie"}` and if we ask `"Kermit" in characters` the program will return `True`.

```
characters = {"Kermit","Piggy","Fozzie"}
"Kermit" in characters
```

```
True
```

You might have noticed that Kermit, Piggy, and Fozzie are all names of characters from The Muppet Show. So in a sense you might say muppet in `characters` is true, but Python doesn't know that. Python is not giving a meaning to the text, it is simply evaluating the string of characters against another string.

The most common conditional in Python is probably the `==` comparator. This is two `=` characters in a row. One equals character is what we use for variable assignment. Two characters is for comparisons. See below for an example of comparisons using `==`.

```
a = 42
print(a == 42)

b = 55
print(a == b)
```

```
True
False
```

```
print(a = 42)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-3-6dee9f19c588> in <module>
```

```
----> 1 print(a = 42)

TypeError: 'a' is an invalid keyword argument for print()
```

4.1.1 Boolean operators

The primitive data type we spent virtually no time on in the prior chapter is the Boolean variable. This is a variable that is either `True` or `False`. To get Python to return a Boolean, all you need to do is make a comparison using a Boolean operator.

- `==` is used for comparison. Does X equal Y? `x == y`
- `and` is used to ask if two things are both true. `x and y`
- `or` is used to ask if either thing is true. `x or y`
- `not` as well as `!` are used for not. `not x`
- `>` is used for left side greater than right side. `x > y`
- `<` is used for left side less than right side. `x < y`
- `in` is used for membership of element in set. `x in y`
- `is` is used to check if two labels point to the same variable `x is y`

```
x = "yes"
print("x:",x)

x_list = ["yes","no","maybe"]
print("x_list:",x_list)

print("x in x_list:",x in x_list)
print()
print('x_list == ["yes","no","maybe"]:',x_list == ["yes","no","maybe"])
print('x_list is ["yes","no","maybe"]:',x_list is ["yes","no","maybe"])
print()

y_list = x_list
print("y_list is x_list:", y_list is x_list)
```

```
x: yes
x_list: ['yes', 'no', 'maybe']
x in x_list: True

x_list == ["yes","no","maybe"]: True
x_list is ["yes","no","maybe"]: False

y_list is x_list: True
```

4.1.2 Flow control using `if` statements and Boolean operators

Inside a loop or inside a program generally you will want to direct the flow of the program. That is to say, you will want the program to do different things under different conditions. So with the operators above we can direct the flow of a program using `if` statements.

```
x = 5
if x == 5:
    print("Yep, x did equal 5, thank golly.")
```

```
Yep, x did equal 5, thank golly.
```

With an if statement also comes an else statement. Below I am going to simulate a coin toss using the `random` module. That module has a method called choice. You can probably follow all the rest by reading the code and running it a few times.

```
import random

coin_sides = ["heads","tails"]

if random.choice(coin_sides) == "heads":
    print("You win!")
else:
    print("Sorry, better luck next time.")
```

```
Sorry, better luck next time.
```

As it happens, there might be more than two conditions you want to cover. One way to include multiple conditions is to use an `elif` statement, which is combination of `else` and `if`.

```
coin_sides = ["heads","tails","rim"]
side = random.choice(coin_sides)

if side == "heads":
    print("You win!")
elif side == "tails":
    print("Sorry, better luck next time.")
elif side == "rim":
    print("It landed on its side, what are the odds?")
```

```
You win!
```

4.1.3 The walrus operator (:=)

So the first example I placed `random.choice` inside the `if` statement. Yet, in the second example I didn't, but instead went `side = random.choice(coin_sides)` and then compared `side` in the `if` statement. Why do you think that was? It was because if I went:

```
if random.choice(coin_sides) == "heads":
    print("You win!")
elif random.choice(coin_sides) == "tails:
    print("Sorry, better luck next time.")
```

Then in the `elif` statement I would run an entirely different `random.choice`, which was not my intention. I intended to run `random.choice` once and ask about that under a few conditions. So I created that `side` variable first and then I ran the program. But if I was able somehow turn that first `random.choice` into a variable `side` *inside* the if statement then I could skip a step. This is the new walrus operator, which looks like `:=` or two eyes and tusks. It stores the result of whatever comes after it. Watch it below:

```
if (side := random.choice(coin_sides)) == "heads":
    print("You win!")
elif side == "tails":
    print("Sorry, better luck next time.")
elif side == "rim":
    print("It landed on its side, what are the odds?")
else:
    print(side)
```

```
You win!
```

Notice I had to put the walrus in parenthesis. That's because I wanted to assign `random.choice(coin_sides)` to `side` and not the full `random.choice(coin_sides) == "heads"` to `side`. In the former case `side` would be one of our three choices. In the latter case `side` would end up being `True` or `False`.

4.1.4 Comparing things *other* than Booleans

There are some tedious issues with comparisons when you are not Boolean operators. Below I show some issues with comparing numbers, strings, and empty variables.

You can compare strings.

String encodings have code points. These are used to evaluate whether one string is greater than another. So you can ask if `a > b`. The behavior can be a bit unexpected so I would only use this with caution. For example, what's greater: `A, a,` or `B`?

```
# String comparisons
print("Is a > b?")
print('a' > 'b')

print("\nIs A > b?")
print('A' > 'b')

print("\nIs a > A?")
print('a' > 'A')
```

```
Is a > b?
False

Is A > b?
False

Is a > A?
True
```

Comparing numbers: zero is false, one is true, the rest are trouble

When comparing numbers, you can of course assess whether numbers are greater or less than each other with ease. But what if you want to assess whether a number is true or false? In Python, the number 0 equals `False`, the number 1 equals `True`, and the rest are neither `True` nor `False`. This is quite strange to say they are neither true nor false. But follow the code below to see this in action:

```
print("What evalues to 'True'?")
print("-1\t", -1 == True)
print("0\t",   0 == True)
print("1\t",   1 == True)
print("2\t",   2 == True)
```

```
What evalues to 'True'?
-1      False
0       False
1       True
2       False
```

```
print("\nWhat evalues to 'False'?")

print("-1\t", -1 == False)
print("0\t",   0 == False)
print("1\t",   1 == False)
print("2\t",   2 == False)
```

```
What evalues to 'False'?
-1      False
0       True
1       False
2       False
```

Not everything that is empty...is False.

There are a number of ways of expressing *nothing* in Python. There's the notion of a variable being `None` or empty. There's a numeric variable that isn't actually a number (`nan`, for *Not A Number* for things that don't compute or are missing), there's the empty string `""` and certainly more. Be extra careful when evaluating these as `True` or `False`.

```
import numpy as np # The Python numeric package 'numpy'.

print(np.nan == True)
print(np.nan == False)
```

```
False
False
```

Ok so `nan` like the numbers 2 and -1 is neither equal to True or False. Yet observe below:

```
if np.nan:
    print("Nan evaluates as True")
else:
    print("Nan evaluates as False")
```

```
Nan evaluates as True
```

Curiouser and curiouser. Below are some other empty strings. Again neither of them are equal to the Boolean `True` or `False` values but yet they still somehow can be used in `if` statements.

```
print(None == True)
print(None == False)

if None:
    print("None evaluates as True")
else:
    print("None evaluates as False")
```

```
False
False
None evaluates as False
```

```
print("" == True)
print("" == False)
```

```
if "":
    print("Empty quotes evaluate as True")
else:
    print("Empty quotes evaluate as False")
```

```
False
False
Empty quotes evaluate as False
```

4.2 Iteration to simplify repetitive tasks

Often times we can think of something to do with a program but doing it once might be easier by hand. For example, imagine having a file with the name `"John_Smith_131313.pdf"`, and having some other detail linked to ID 131313 rather the name. If we want to find application 131313 we will have to look through the whole list one by one. So you get the idea that it would be easier to rename the file `131313_John_Smith.pdf`. Doing that by hand is not too hard. Now imagine you have dozens or hundreds of files. At some point it will become easier find a way to automate this task.

If the files were named `"Jane_1412223_Doe.pdf"`,`"John_Smith_131313.pdf"`,`"Bernie_13Ho151gan.pdf"`, etc., then this would be a harder (but not necessarily impossible) task. To that end, we tend to want collections to have some consistency to them. Then we can make use of that consistency in order to define an operation once and do it for all the items. Let's look at one part of this particular challenge first and then generalise to loops second.

```
records = ["John_Smith_131313.pdf",
           "Bernie_Hogan_113113.pdf",
           "Richard_D_James_112358.pdf"]

for record in records:
    print(record.split("_")[-1])
```

```
131313.pdf
113113.pdf
112358.pdf
```

So we didn't get all the way to changing the names here, but that is unnecessary to make the main point. Notice that we did the same thing to each of the records iteratively. We took each `record` one after the other, split the record at the underscores and printed the last element. We used the Python syntax of `for <element> in <collection>: <do something>`. This syntax is called a `for loop`.

As a verb we would *loop* over a collection, but also as a noun, would refer to 'the loop' as the set of iterations. So we could *loop* over a collection of items but also if we stop halfway through because of an error we have just broke the *loop*.

4.2.1 Iterating using a `for` loop.

As I mentioned in Chapter 3, virtually all collections are iterable. That means they will start with one element in the collection and keep giving you elements until you exhaust all the elements. In the loop example above, every time we went through the loop we used a different element from the list. That element became our `record`. Notice that the final time we do not clear the variable, so you can ask for `record` below and it will give you the last one from the above result.

```
print(record)
```

`Richard_D_James_112358.pdf`

Sometimes the collection is ordered, such as a `list`, and sometimes the collection is unordered, such as a `set` or a `dictionary`. Regardless, you are able to perform some action with each element of the collection and know that once you've finished you've iterated through every element of the collection. The process of going through all the items is called **iterating**.

The way to iterate through a collection in Python is to use a `for` loop. It is called a `for` loop because we use it to do something *for* each element in a collection. In fact, the way it is written is meant to be similar to written English.

Since we iterate through the collection, we need a temporary variable to represent the next item each time. That was the `record` variable above. We could have named it anything, but `record` was useful to help us understand what kind of item was in the collection. Often times people will use `i` for the iterator, like with the following:

```
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

Sometimes you will encounter `_` instead of i. That's meant to signify that while you are iterating through a collection you are not interested in the variable. For example, imagine you wanted to print a counter for each element in a list but only print the counter and not the value.

```
counter = 0
for _ in ["apple","banana","cherry"]:
    print(counter)
    counter += 1
```

```
0
1
2
```

4.2.2 The `else` condition

When a loop ends you can do something specifically at the end of the loop. This you can define as the `else` condition.

```
for num in range(5):
    print(num)
else:
    print("Done!")
```

```
0
1
2
3
4
Done!
```

Now to be clear, this is not `if` and then `else`. This is `for` and then `else`. It's a bit of a strange name. Perhaps it might be better if it was called `after` or `once_finished`, but they called it `else`.

4.2.3 Enumerate

Sometimes when you are iterating through a for loop it is useful to have a counter. Python does not provide one by default. So you might end up doing the following:

```
list_of_fruit = ["apple","banana","cherry","date"]

counter = 0
for fruit in list_of_fruit:
    print(str(counter),fruit,sep="\t")
    counter += 1
```

```
0       apple
1       banana
2       cherry
3       date
```

A more pythonic way of doing this would be to use a special function called `enumerate()`. This function takes in your collection and then during the for loop returns **both** a counter and one element at a time from your collection. See `enumerate` in action below:

```
list_of_fruit = ["apple","banana","cherry","date"]

for counter, fruit in enumerate(list_of_fruit):
    print(counter,fruit,sep=". ")
```

```
0. apple
1. banana
2. cherry
3. date
```

The enumerate function has an additional parameter `start`. The default value is `0` which is why above we printed `0. apple` on the first line. If we say `start=1` inside the enumerate (or just `enumerate(<collection>, 1)` then we will get a list starting from 1 instead.

```
list_of_fruit = ["apple","banana","cherry","date"]

for counter, fruit in enumerate(list_of_fruit,1):
    print(counter,fruit,sep=". ")
```

```
1. apple
2. banana
3. cherry
4. date
```

One very common use of enumerate is to report something every n^{th} iteration. The trick here is to use 'the remainder' from integer division. Since we have a counter `c`, when we divide that counter by n using integer division we will get a remainder. That remainder will be 0 every n times. See below where we will print every fifth number:

```
for c, num in enumerate(range(30)):
    if c%5==0:
        print(num)
```

```
0
5
```

```
10
15
20
25
```

4.2.4 Managing a loop with `continue` and `break`

You might want to stop a loop under some condition, or stop a loop partway through and then start again for the element in the collection. To do this, we can apply `break` and `continue`.

- **`break`** stops the loop completely. It does not stop a program so it continues running the code that comes *after* the loop.
- **`continue`** will stop that particular iteration of a loop but the loop will continue if there are more elements.

You can see examples of both of these below:

```
fruits = ["apple","banana","cherry","durian","eggplant"]

for fruit in fruits:
    if 'r' in fruit:
        continue
    print(f"The word {fruit} is {len(fruit)} characters long")
```

```
The word apple is 5 characters long
The word banana is 6 characters long
The word eggplant is 8 characters long
```

So notice that it skipped over both `cherry` and `durian` because they had an `r` in there? That's how continue works. On the other hand, if we wanted to stop when we come across a word with `r` in it, we would have used `break` like so:

```
fruits = ["apple","banana","cherry","durian","eggplant"]

for fruit in fruits:
    if 'r' in fruit:
        break
    print(f"The word {fruit} is {len(fruit)} characters long")
```

```
The word apple is 5 characters long
The word banana is 6 characters long
```

In this case, the program stopped once it got to cherry, since that triggered the `break` statement.

4.2.5 Double loops

You can loop while inside of a loop. This is useful for thinking about data in a table form. Imagine you have a table of records such as person and their test results:

Person	Test1	Test2	Test3	Test4
Alan	9	6	8	9
Barb	7	7	7	6
Carl	7	6	4	9
Dana	8	9	9	8
Ella	5	7	8	7

Now imagine you want to calculate an average. You could either calculate the average *per-test*, which would move down the columns or you could calculate the average *per-student*, which would move across the rows. Both of these can be done with a double `for` loop. But the way the loop works will depend on the data structure. It would be different if we have a list of test results (e.g., `test1 = [9,7,7,8,5]`) and we knew that Alan was always the first result versus if we have a list of students and new that Alan was the first student (e.g., `Alan = [9,6,8,9]`).

Below I will create a list-of-lists. In this example, the inner lists will be the scores per person. That means we received a list of scores for each person one at a time. Then if we combine them we will have a table like above.

```
results = [[9,6,8,9],
           [7,7,7,6],
           [7,6,4,9],
           [8,9,9,8],
           [5,7,8,7]]
```

We can print this table using a double loop. Let's do this first before embarking on some more complicated algorithms. First let's just print it using a single loop to see what each object in `results` looks like:

```
for row in results:
    print(row)
```

```
[9, 6, 8, 9]
[7, 7, 7, 6]
[7, 6, 4, 9]
[8, 9, 9, 8]
[5, 7, 8, 7]
```

So this printed each row as a list, complete with the `[]` on either side. But we want to print each element in these lists.

```
for row in results:
    for score in row:
        print(score,end=",")
    else:
        print()
```

```
9,6,8,9,
7,7,7,6,
7,6,4,9,
8,9,9,8,
5,7,8,7,
```

I don't like having that trailing comma in there, so I'm going to complicate this just a little bit. Watch how the inner loop is for every element in the row *except* the last one with `row[:-1]`. Then I do something different for the last one `row[-1]`.

```
for row in results:
    for score in row[:-1]:
        print(score,end=",")
    else:
        print(row[-1])
```

```
9,6,8,9
7,7,7,6
```

```
7,6,4,9
8,9,9,8
5,7,8,7
```

Now let's work within this double loop in some interesting ways. IF we want to calculate the average within row, that's not too hard. For each person we can add up that person's scores and divide by the number of scores. So we will then have a new list called `student_averages`, with one score for each person.

Just to reassure you, however, this is not where we stop with Python if we want to get an average. Later we will find ways that are less cumbersome and look more like a one-stop `<collection>.average()`. But it helps to understand how to build your own algorithms like this so that when we get to these other ones you will know what to look for and how to circumvent it in case it does not do precisely what you expected. Also, there will still be lots of times where you will have a collection, and another collection inside there.

```
student_averages = []

for student_scores in results:
    total = 0
    for test in student_scores:
        total += test
    else:
        student_averages.append(total/len(student_scores))

print(student_averages)
```

```
[8.0, 6.75, 6.5, 8.5, 6.75]
```

From here we can see that Alan's score was `8.0`. Carl had the lowest score with `6.5` and Dana the highest with `8.5`.

So *that* double loop was relatively straightforward. You had an object (the `results` list) and inside was another collection (the test scores) and we did something for that. What if we wanted to calculate the average per test with the results data? This might be a little more complicated. Later, we might find a more amenable structure like a `pandas` table or `numpy` array. But that's for another day. Today, let's see how one might go about such a challenge with the tools on hand.

```
num_students = len(results)
num_tests = len(results[0])
test_totals = [0]*num_tests
test_averages = []

for student_scores in results:
    for c,test in enumerate(student_scores):
        test_totals[c] += test
else:
    for c,total in enumerate(test_totals):
        test_averages.append(total/num_students)

print(test_averages)
```

```
[7.2, 7.0, 7.2, 7.8]
```

That one is probably a bit tricky just starting out. But if you do not understand what I did in that algorithm, try to break it down. One thing I like to do when learning what's going on in a loop is to print some parts of it and then insert a break statement so that something is only done once. For example if I forget what exactly are `student_scores` I would insert a `print(student_scores)` inside the outer for loop and then insert a `break`.

4.2.6 Iterating dictionaries

Like lists and sets, dictionaries are collections. But recall that they are collections of `key:value` pairs? So what happens when you iterate a dictionary by default?

```
ingredients = {"salt":"parmesean",
               "fat":"olive oil",
               "acid":"vinegar",
               "heat":"toast"}

for test in ingredients:
    print(test)
```

```
salt
fat
acid
heat
```

Notice that it just printed the keys. This makes sense in a way for a dictionary; it's why you can ask `'salt' in ingredients` and it will return `True`, but `'parmesean' in ingredients` will return `False` even though we can see above that it is a value. The dictionary is focusing on the keys.

```
print('salt' in ingredients)
print('parmesean' in ingredients)
```

```
True
False
```

To get the dictionary to print the values you would want to use `ingredients.values()` which will return the values as a list (or specifically a `dict_list` which is close enough:

```
for i in ingredients.values():
    print(i)
```

```
parmesean
olive oil
vinegar
toast
```

We can ask for both the key and the value at the same time with `<dict>.items()`. So in a for loop we can do the following:

```
for category, food in ingredients.items():
    print(f"{food.capitalize()} helps with the {category}.")
```

```
Parmesean helps with the salt.
Olive oil helps with the fat.
Vinegar helps with the acid.
Toast helps with the heat.
```

4.2.7 List comprehensions

The list comprehension is literally my favorite syntactic sugar in Python. Often times you will have a list and you will want to create a new list from the old one. For example, imagine having a list of words and you want to turn them all to upper case or to return to the beginning, a list of file names that you want to

process. Below I will first show the code pattern that does this by creating a list and appending things to that list, much like what we have seen so far. Then I will show the list comprehension.

```
my_list = ["allspice","basil","cumin"]

new_list = []
for i in my_list:
    i = i.upper()
    new_list.append(i)

print(new_list)
```

```
['ALLSPICE', 'BASIL', 'CUMIN']
```

```
new_list = [i.upper() for i in my_list]
print(new_list)
```

```
['ALLSPICE', 'BASIL', 'CUMIN']
```

The list comprehension was able to reduce the for loop down considerably. List comps can be pretty complex as well. I will leave out much of that complexity for now, but one thing worth introducing is how you can include a conditional in the comprehension. Imagine you only want the new list to keep *some* of the items. In such cases, just place a conditional after the for statement, like so:

```
new_list = [i.upper() for i in my_list if len(i) == 5]
print(new_list)
```

```
['BASIL', 'CUMIN']
```

4.2.8 Dictionary Comprehensions

Very similar to list comprehensions, sometimes you might want to create a dictionary comprehension. In this case, the syntax is pretty familiar. You simply need to stipulate what is the key and what is the value. See for example how to construct a dictionary comprehension with our foods.

```
my_dict = {"allspice":"1 tsp","basil":"1/2 tsp","cumin":"2 tsp"}

new_dict = {key.title():val.split()[0] for key,val in my_dict.items()}

print("The new dict comp (with measures in teaspoons):\n",new_dict)
```

```
The new dict comp (with measures in teaspoons):
 {'Allspice': '1', 'Basil': '1/2', 'Cumin': '2'}
```

4.3 While loops

In the case of `for` loops and list comprehensions, we had a sense that we first had a finite collection of something and then we were going to iterate through that collection doing something every time until we exhaust elements of a collection (or until we threw a `break` statement). But what if we do not know how long to wait for something? For example, if I generate a set of random numbers, how long before I get the number 7? That's uncertain. But we can establish that a set of random numbers should return a 7 eventually.

See below how I use a while loop to keep running until we get a 7.

```
x = True

while x:
    random_number = random.randint(0,10)
    print(random_number)

    if random_number == 7: # If you comment this out, it will run indefinitely
        x = False
```

```
6
5
2
6
3
0
5
8
7
```

If you leave out the `x = False` above, you will get an infinite loop since there is no stopping condition. You can try this for fun but you will eventually run out of memory or your processor will overheat. Either one is not good. You can stop an infinite loop by selecting `Kernel -> Restart Kernel` from the menu above.

One common code pattern for a `while` loop is when asking for user input. Imagine you have a series of choices and a text prompt. You can loop through this prompt until you get a valid choice. First let's observe a simple user input. Then let's put that in a `while` loop.

```
value = input("Please type some text and hit enter")
print(f"\nIt appears you typed:\n{value}")
```

```
Please type some text and hit enter Hello!


It appears you typed:
Hello!
```

```
input_continue = True

while input_continue:
    value = input("Type Y to leave the loop. Type something else to stay:")
    if value.lower() == 'y':
        input_continue = False
    else:
        print("I'm sorry, that was not a valid choice")
else:
    print("You have now exited the loop")
```

```
Type Y to leave the loop. Type something else to stay: I'm staying

I'm sorry, that was not a valid choice

Type Y to leave the loop. Type something else to stay: Y

You have now exited the loop
```

4.4 Errors to be handled and sometimes raised

The ways above to control the flow of the program tend to let us continue the program under some conditions or for some elements. We might use a for loop to iterate or an if statement for some conditions, but we still want to keep the program running.

Yet, sometimes we want to signal to the user that this program is not behaving as expected. In this case, we can `raise` an error. On the other hand, there might be times when the program would typically raise an error on its own, but we anticipate that and catch that error so our program does not stop running. In Chapter 2 we introduced errors and I provided some tips on how to test your code to deal with them. There when I said deal with them, I implicitly meant make the error go away. However, here instead, I am now thinking about how we can use errors to manage the flow of the program.

This will be especially pertinent with work on the web, as errors with connections happen all the time, but we normally want to then just pause the program for a second or two before trying again.

Here is an example of an error:

```
1/0
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-48-9e1622b385b6> in <module>
----> 1 1/0

ZeroDivisionError: division by zero
```

The error is a zero division error. As we know from maths, we cannot divide by zero for zero means nothing (sparing us all the complex proofs of this, which are really interesting, but not for here). The important part is that we can catch the error and move on if we anticipate it and think it's not going to affect future code. We do this using `try` and `except` statements. See the example below:

```
import numpy as np

for i in range(-2,2):
    try:
        print(f"1/{i} is {1/i}")
    except ZeroDivisionError:
        print(f"Caught an error! 1/{i} is {np.nan}")
```

```
1/-2 is -0.5
1/-1 is -1.0
Caught an error! 1/0 is nan
1/1 is 1.0
```

If you wanted to create your own exception, we would say you 'raise' an error. So for example, if your code should throw an error when you receive a variable of the wrong type, you can `raise` a `TypeError`. This is much less common in data analysis than simply catching errors.

```
x = 8

if int(x):
    raise TypeError("The program did not expect an integer.")
```

```
---------------------------------------------------------------------------
```

```
TypeError                                 Traceback (most recent call last)
<ipython-input-50-bbb026ca7ced> in <module>
      2
      3 if int(x):
----> 4     raise TypeError("The program did not expect an integer.")

TypeError: The program did not expect an integer.
```

Then by combining raising and catching errors, you can find ways to ensure that your code is more robust, particularly when importing and processing raw data.

When we `raise` an error, that means we send an error object to the program. This is a special kind of object that will usually end the program unless we `except` it.

In the `1/0` case above we just said `except` and left out the `ZeroDivisionError` it would catch all possible errors. This is not necessarily what we want, since some of those errors might be legitimately concerning while others are things we anticipate as a matter of course. But sometimes you might want to catch all errors and then find out which one it was. Since the error itself is an object, it has properties we can work with. See below:

```
try:
    1/0
except Exception as err:
    print(err)
    print(type(err).__name__)
```

```
division by zero
ZeroDivisionError
```

Normally properties of an object do not start with `__` or "the dunder" for double underscore. When a variable or method starts with `__` it normally means it is only meant to be used by the system and not in code. But Python still exposes these methods in case you want to do any advanced tinkering, like printing the name of an error.

4.5 Conclusion

There are many ways to direct the flow of a program. Here we first saw the use of Boolean operators to evaluate something as `True` or `False`. We then saw how we could use this in `if` statements in order to do something under some conditions. Then we looked at loops, first the for loop, followed by list comprehensions and while loops. In each case, we were able to direct the flow of a program by doing something for each element of a collection. Finally, we explored errors, which are ways in which we can halt a program entirely (as well as how to catch these errors so our program does not halt if we can anticipate that error).

These provide a very sound basis for a program. Yet they are not the last topic on flow within a program. In fact they are not even the last word on loops. But now it would be worth thinking about how we can collect many of these operations in a single place if we want to use them together repeatedly. This single place with be a function. Then with a function we can really start to create programs with a variety of structures depending on our needs. You can see this in the next chapter.

Chapter 5

Functions and classes

5.1 Functions and object-oriented programming

Functions allow you to group together related operations in such a way that you can abstract away details in your program. Two main use cases of functions come to mind: 1. Avoiding repetition and the bugs that can come from inconsistent code; 2. Grouping together operations used elsewhere (like in list comprehensions and equality comparisons).

5.1.1 Defining a function

We have already seen a few functions such as `print()` and `len()`. Building your own functions is a crucial part of coding. Without user-defined functions, you are left with code that is literally just one command after another. With functions you can abstract away the common parts, code them once inside the function, and then send the unique or novel parts to the function as **arguments**.

Below is an example of some repetitive code followed by an example of a function that factors out the parts of the repetitive code. A function starts with `def` (for 'define'), followed by a **name**, and an **argument** on the same line. Then "inside" the function (which is also denoted visually with indentation) is the code that is run each time the function is called. This code will use the values sent in (those are the arguments) and then typically `return` some object as output.

```
# Try to identify the repetition
list_result = []

x = 5
if x %2 == 1:     list_result.append(x * 2)
else:             list_result.append(x)

x = 7
if x %2 == 1:     list_result.append(x * 2)
else:             list_result.append(x)

x = 12
if x %2 == 1:     list_result.append(x * 2)
else:             list_result.append(x)

print(list_result)
```

```
[10, 14, 12]
```

```
def doubleIfOdd(num):
    if num % 2 == 1:
        return num * 2
    else:
        return num

print([doubleIfOdd(x) for x in [5,7,12]])
```

```
[10, 14, 12]
```

To build that function we need to do the four things specified above:

- Name
- Inputs
- Calculations
- Outputs

The name was `doubleIfOdd`, the inputs in this case referred to a single variable called `num`, the calculations referred to the `if-else` statement, and the output referred to what we returned, namely `num` or `num*2`. Below we can see a similar function, except it doubles numbers if they are even.

```
def doubleTheNumberIfEven (input_number):
    if input_number%2==0:
        return input_number * 2
    else:
        return input_number

numbers = [1,4,6,7,9,14,17]

new_numbers = [doubleTheNumberIfEven(i) for i in numbers]

print(new_numbers)
```

```
[1, 8, 12, 7, 9, 28, 17]
```

5.1.2 Variables have a 'scope'

A variable that is created inside of a function is not the same as the one created outside of that function even if they have the same name. This is because the variable inside the function is a **local** variable. Variables created in Jupyter are typically treated as **global** variables if they are created in a cell but not if they are created inside a function. To be global means that they can be used anywhere in the code. Local variables are created and destroyed within their local context. You can watch this behavior with a code snippet.

```
# Local / Global scope example 1: Variable in the function stays in there.

def multiplyTheValue(input_number):
    x = input_number * 2
    print("Value of x inside the function",x)
    return x

x = 4
output_number = multiplyTheValue(x)
print("Result from the function:",output_number)
print("Value of x after the function:",x)
```

```
Value of x inside the function 8
Result from the function: 8
Value of x after the function: 4
```

But `x` wasn't the argument, `input_number` was. So what if we change `input_number` inside the function?

```
# Local / Global scope example 2: Argument sent to function doesn't escape the␣
↪function.

def multiplyTheValue(input_number):
    print("Inside the function",input_number)
    return input_number

x = 4
output_number = multiplyTheValue(x)
print("After the function",input_number)
print("Value of X after the function:",x)
```

```
Inside the function 4
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-5-f113304a7fcc> in <module>
      7 x = 4
      8 output_number = multiplyTheValue(x)
----> 9 print("After the function",input_number)
     10 print("Value of X after the function:",x)

NameError: name 'input_number' is not defined
```

We sent `x` to the function, at which point it became the value for the `input_number` parameter. So we could use `input_number` inside the function, but then when we try to call it outside the function it throws an error. To make it available outside the function is not an advised code pattern, but it is possible by using the `global` flag.

```
# Local / Global scope example 3: Casting a variable as global makes it available␣
↪outside the function.

def multiplyTheValue(input_number):
    global x
    x = input_number * 2
    print("Value of x inside the function",x,id(x))
    return x

x = 4
print("Value of x before the function",x,id(x))
output_number = multiplyTheValue(x)
print("Value of x after the function",x,id(x))
print("After the function",output_number)
```

```
Value of x before the function 4 4336056768
Value of x inside the function 8 4336056896
Value of x after the function 8 4336056896
After the function 8
```

In this third example, we can see that when we declare x is a global variable inside the function, that value then becomes the value outside of the function. We double `x` inside the function and then later when we print x it is no longer 4, it retains the value it had inside the function.

5.1.3 There are all kinds of ways of passing data to a function.

A function usually has some *parameters*. Parameters are like another word for options or settings. When you define a function, it is parameters that you write between the parentheses. But when you are coding you are more interested in the values of these parameters. These are *arguments*. So in the function:

```
def tinyexample(word):
    print("Tiny examples!", word)

tinyexample("Big ideas!")
```

```
Tiny examples! Big ideas!
```

`word` is the parameter, `"Big Ideas"` is the argument. That said, most people use these terms interchangably.

There are a number of different kinds of parameters. Some of these allow a function to take in a flexible number of arguments, others define the type of argument that the parameter will permit. Parameters can take default values. If the parameter has a default value, then one does not need to send an argument when running the function.

Note that since a function can have a combination of different parameter types, the ones without defaults come first. Let's see how some different functions take multiple arguments below:

```
# Example 1. Just a single positional argument
def example1(just_name):
    print(just_name)

example1("example 1 argument")
```

```
example 1 argument
```

```
# Example 2. A positional argument with a default value
def example2(arg_name, setting1 = True, setting2 = True ):
    if setting1:
        print(arg_name)
        return
    elif setting2:
        print(arg_name.upper())
        return
    else:
        print(f"{arg_name} You have disabled the settings")
```

```
example2("Example 2. Take 1.",False)

example2("Example 2. Take 2a.", setting2 = False)

example2("Example 2. Take 2b.", True, False)

example2("Example 2. Take 3.", False, False)
```

```
EXAMPLE 2. TAKE 1.
Example 2. Take 2a.
Example 2. Take 2b.
Example 2. Take 3. You have disabled the settings
```

```python
# Example 3. Postional arguments passed but not defined ahead of time
def example3(just_name, *args):
    if len(args) > 0:
        for i in args: print(i)

example3("some data","Maybe","more data")

# Below, why does it not print 'some data'?
```

```
Maybe
more data
```

```python
# Example 4. Keyword arguments passed but not defined ahead of time
def example4(just_name,**kwargs):
    if len(kwargs) > 0:
        for i,j in kwargs.items():
            print("var name:",i,"\tvalue:",j)

example4("example",
         var1="some data from v1",
         var3="Maybe it's v3?",
         var2="v2's valuedata")
```

```
var name: var1  value: some data from v1
var name: var3  value: Maybe it's v3?
var name: var2  value: v2's valuedata
```

```python
# Example 5. Showing the possibilities (and dangers) of fragile code and weakly cast␣
 ↪variables.

def MakeDouble(value):
    try:
        output = value*2
    except TypeError:
        output = None

    return output

print( MakeDouble(2)  )
print( MakeDouble("Double")  )
print( MakeDouble(["2"]))
print( MakeDouble({1:4}))
```

```
4
DoubleDouble
['2', '2']
None
```

5.1.4 A function always returns, but it might be nothing at all.

Your function always stops at the return statement. You can have multiple return statements for different conditions (like saying if…return one thing and else…return another). After the return statement, the remaining code will not be evaluated by the program. But if your function does not have a return statement, Python will still return `None` (which if you remember from above evaluates to `False`). Just try it for yourself.

```
def noReturn():
    pass

print(noReturn())

if noReturn():
    print("Did it work?")
else:
    print("Oh right, None evaluates to false.")
```

```
None
Oh right, None evaluates to false.
```

5.2 Classes and Objects

Classes are a means of grouping together relevant variables and methods into a single class. Then a class becomes the template for some kind of object. Stated differently, every object is an object of some type of *class*. Object-oriented programming is one of many paradigms of programming. And not all programs need to be object oriented. Regardless, Python is predominantly object-oriented (as is Java, C++, swift, Objective-C, and Ruby, for example).

To say that a program is object-oriented means that it uses `objects` as a part of its processing. An `object` is the generic term for any data structure that can be created by a program. A nice feature of an object is that it can contain other objects unless it is a 'primitive'. So a character is a primitive but a string is a collection of characters. But we can also have a collection of strings (like a list of strings or a dictionary of key:value pairs). Objects have specialised methods. For example, the string object has `.upper()` or `.lower()` methods.

We would say that objects of the same type are *instances* of the same *class*. So, above, when I said "the string object has a…", that was short hand. More specifically, I could have said "any instantiated objects of the string class can use the…"

You can create a class from scratch or extend and existing class.

5.2.1 Creating classes using `__init__`

To create an object, you need to first define the class name and the provide an internal method called `__init__`. This method will automatically run every time you create (or "initialise") a new object of that type. So if you had a class called `Pizza` which you know creates pizza objects, then you would probably initialise it with a few relevant variables such as `toppings = []`, `sauce='tomato'` , and `base = 'classic'`. You can then modify the pizza object. This would be a basic Pizza class:

```
class Pizza:
    def __init__ (self):
        self.toppings = []
        self.base = 'classic'
        self.sauce = 'tomato'
```

```
p = Pizza()
z = Pizza()
```

Now we can consider the pizza object as a combination of multiple other objects that all work together. A shopping cart, for example, might be a class that includes a list of items, a discount code, and an identifier for the customer that owns the shopping cart. Admittedly, for something like pizza or a shopping cart we can also get away with just using a dictionary. That is, we could have simply written:

```
pizza = {toppings:[], base:"classic",sauce:"tomato"}

pizza[toppings].append("red peppers")
pizza["base"] = "thin and crispy"
```

So what is the advantage of using a class rather than this structure? It depends on the purpose. For simple data transfer, actually it is nice to just keep it as dictionaries and lists. Later when we look at JSON files from the web we will see how they are essentially just collections of lists and dictionaries. But when programming, it is useful to be able to have a *structure* to the various objects that are related to each other. This structure can give some sense to the objects as well as ensure that they all work in sync. For example, what if we want to manage two pizza orders? Will we create another variable called `pizza2`?

Below I will show two approaches to printing off a receipt. Compare how I would do it for a dictionary like above, and then for a class:

```
cart = {"items":[], "code":None, "customer":None}

cart["items"] = ["Turntable","Microphone","Keyboard"]
cart["code"] = "HAPPY2020"
cart["customer"] = "Tom"

print(f'Welcome {cart["customer"]}\n\nYour items:\n{" ".join(cart["items"])}\nDiscount␣
 ↪code:{cart["code"]}')
```

```
Welcome Tom

Your items:
Turntable Microphone Keyboard
Discount code:HAPPY2020
```

```
class Cart:
    def __init__(self):
        self.items = []
        self.code = None
        self.customer = None

    def receipt(self):
        message = f"Welcome {self.customer}\n\nYour items:\n"
        message += "\n".join(self.items) + "\n"

        if self.code:
            message += f"Discount code:{self.code} applied"
        return message
```

So above is just the class file. From here we can see a couple differences. The first is that the `receipt` function is inside the `Cart` class. The second is that when we are referring to objects that belong to the `Cart`

class inside of the class definition we refer to them as `self.<object>`. So `__init__` is never really called directly, you never say `x = Cart.__init__()`, instead you initialise by saying `x = Cart()`, which then will automatically run the `__init__` method. In this case, it will create three internal variables, `self.items`, `self.code`, and `self.customer`, and give them some values. Although this seems a little overkill compared to the nested dictionary, it creates more of a structure to work with. Then we can create multiple cart instances, as can be seen below.

```
x = Cart()

x.items = ["Turntable","Microphone","Mixer"]
x.code = "HAPPYSPINNING"
x.customer = "Chuck"

print(x.receipt())
```

```
Welcome Chuck

Your items:
Turntable
Microphone
Mixer
Discount code:HAPPYSPINNING applied
```

Compare how the receipt was printed this time with the code above. We abstracted away the details of printing to the `receipt()` method of the `Cart` class, which we defined elsewhere. We were still able to access the objects in the `Cart` class, but instead of `self.items`, we first instantiated an object called `x`, and then used `x.items`. Some classes can be fussy and expect you to use a dedicated method to get these objects, like `x.get_items()`. Other times classes allow you to access the objects directly. It's a bit of trial and error as well as checking in on the docs for a particular package.

Below I will create a second object just to demonstrate how we can have separate `Cart` objects and use them together in a `print()` statement.

```
y = Cart()
y.items = ["808 Drum Machine", "Keyboard", "Laptop"]
y.customer = "Caterina"

print(y.receipt(),x.receipt(),sep="\n###########\n")
```

```
Welcome Caterina

Your items:
808 Drum Machine
Keyboard
Laptop

###########
Welcome Chuck

Your items:
Turntable
Microphone
Mixer
Discount code:HAPPYSPINNING applied
```

5.2.2 Extending classes and inheriting values

There are instances in both data access and machine learning where the task will have a class that's almost fit for purpose but typically there will be a few key functions missing. You could then 'extend' this class with your own version of these methods.

One example concerns Twitter data. So there is a way to get the Python to listen for new tweets based on some criteria, such as when someone tweets `#BLM`. In the `twitter` library this is done using the `StreamListener` class. When you instantiate a stream listener, it will handle many of the details automatically, like connecting to Twitter and receiving data according to your search parameters. However, it simply listens and does not do anything with the data it receives. For you to do something with the data, you need to *extend* the `StreamListener` class. This extension, perhaps called `CustomStreamListener` will *inherit* all the methods and objects in the `StreamListener` class, but you can add your own additional methods. One method that it will look for is called `on_data`. This method will be called anytime there is a tweet that appears according to your search terms, and then you get to define what to do with that data. For example, in the `on_data()` method, you could fill it with instructions such as "check for hate speech" or "store in a database" or "reply automatically with a messsage".

Here is a simple example building on the `Cart` above. Notice that in the `Trolley` class (which is what you would often call a cart in the UK), we do not say `self.items` or `self.customer` since they were *inherited* from the `Cart` class. But now we will add a user `post_code`, since in the UK addresses have post codes.

```
class Trolley(Cart):
    def __init__(self):
        Cart.__init__(self) # observe what happens if you remove this!
        self.post_code = "OX1 3JS"

    def delivery(self):
        message = "Your basket currently includes:\n"
        message += "\n".join(self.items) + "\n"
        message += "It will be delivered to " + self.post_code

        return message
```

```
z = Trolley()
z.items = ["Cables","Cassette Player"]

z.placename = "OII"

print(z.receipt())

print(z.delivery())
```

```
Welcome None

Your items:
Cables
Cassette Player

Your basket currently includes:
Cables
Cassette Player
It will be delivered to OX1 3JS
```

5.2.3 Reasons to use a class

Part of the reason for showing a class is that it helps us understand the basis of objects, as each object is necessarily an **instance** of some *class* of object. Later when we will be working with data, we will be using DataFrames, which are tables that contain data. These `DataFrame` objects have their own methods, but also *inherit* methods. We will want to know how to create an instance of a `DataFrame` object, what it means to send different arguments, and to query for parts of the `DataFrame`. Observing the code below:

```
import pandas as pd

df = pd.DataFrame(columns=["name","age"])
```

You can already notice that `pandas` is a library. In this library, which we have imported under the name `pd` for short, is a class called a DataFrame. By calling `df = pd.DataFrame()` we are creating an **instance** of the DataFrame class called `df`. By using `cols=["name","age"]` we are sending these two values to the `DataFrame.__init__` method. Thus, when it initialises the DataFrame object `df`, we will have a table with two columns, `name` and `age`. See below (notice that it will possibly run slow the first time you `import pandas`).

```
import pandas as pd

# An empty table with two columns and five rows
df = pd.DataFrame(columns=["name","age"],index=range(5))
df
```

```
  name  age
0  NaN  NaN
1  NaN  NaN
2  NaN  NaN
3  NaN  NaN
4  NaN  NaN
```

The DataFrame in this case is now an empty table. To create a table with data or to manipuate data is outside the scope of this book. Rather it is where we start off in the book "From Social Science to Data Science".

5.3 Conclusion

Now we can see how programming can become pretty complicated, with objects referring to other objects and other functions or methods all over the place. Often times, when I'm trying something new with programming I often have check the documentation or print a lot to get a sense of what methods an object has available or simply to determine what type of object was returned from some method or function. Being able to understand how to query an object or manipulate it will be an important skill moving forward in Python and in giving your scripts some structure. This structure is not merely for its own sake. It helps to create code that is more reusable and robust. By structuring our code we are structuring our ideas about data. That is good if we want to do something repeatedly or consistently across many cases.

The last chapter does not expand our basic programming knowledge much. Instead, the next chapter will focus on how to get out of Jupyter by writing Python scripts as well as by learning the basics of how to read and write files.

Chapter 6

The File System, Path, and Running Scripts

6.1 Learning Python: The file system

Up until now we have only used Jupyter notebooks and stayed pretty closely in this environment. But we will need to branch out eventually. This involves learning a number of features about the operating system and how to interface with it. We will also learn how to create a file, write data to it and read the data from it.

Windows, MacOS, Linux, Chrome or iOS, all computers have an **operating system**. The operating system runs programs, manages memory, and allocates computer tasks so that multiple programs can run at the same time. An operating system is what tends to differentiate modern computing from mechanical tasks. The OS is meant to be a general purpose means for accomplishing tasks.

Virtually all operating systems operate on **Von Neumann architecture**. This means they will have an input device (typically a keyboard but could be any sensor), a processor that does calculations, memory (typically fast memory such as RAM and slow memory such as a Hard Drive for more long term storage) and an output device (often, but not necessarily a screen). Other architectures based on state machines and quantum computing exist but are often experimental and not quite relevant at the moment.

Most operating systems these days are either Windows-based or *nix-based (i.e., Unix or Linux). Macintosh used to have its own operating system kernel, but has switched roughly 20 years ago to Unix for Mac OS X. There are many differences under the hood between Mac and Linux but herein, we will focus on the many similarities related to file storage. These operating systems store files on a hard drive which can be accessed using the **file path**.

6.1.1 Navigating the file path with Python and in the terminal

Files, or more properly speaking **file indices**, are stored in hierarchical structures. The topmost structure would be considered **the root directory**. Under the root directory would be child directories. Even though it can be characterised as a 'tree', we tend to use the terms children and parent. Perhaps think of it more like a family tree, a very peculiar family tree.

Note: File systems are different on *Nix and Windows.

Windows was a small operating system back in the day. It wanted to preserve backwards compatibility with earlier DOS systems and IBM systems, but the `/` was already used in systems as a 'switch' meaning it would not be easy (so it seemed) to use it to denote directory structures. Thus, `\` was used instead since it looks similar. It didn't seem like a big deal at a time, but it has certainly become a nuisance for programmers

ever since. Along with some other quirks, such as the drives as having letter names, means that there are a number of little differences between Windows and *Nix users.

In the code herein, I want to have notes that are as generic as possible across systems as well as notes that are 'robust'. So I will be using some Python libraries to ensure the code stays consistent. In particular we will be using the `os` module, as you will likely see a lot of code snippets that use this module. I will also be using the `pathlib` library, which has been available since Python 3.4. I think it is more tidy and coherent than the earlier `os` module, but it is less commonly used still.

In the `os` library is a variable called `os.sep`. If you print it on Windows it will print a `\` and if you print on a *Nix system it will print `/`. You can use this to build a string representing path that are operating system independent. However, a more robust way is to use `pathlib` to create a Path object, for example:

```
from pathlib import Path
p = Path(".")
open(p / "file.txt")
```

Notice that if we initialise an object called `p` with a path of `"."` that means we are initialising it with a location of the current working directory. Then we can write in Python strings and `/` for folder dividers, yet the `/` here works regardless of windows or mac because it is not literally a `"/"` character. It is outside the string and represents *folder divider* more generally.

What is a root directory and how do I get there?

If we have a hierarchical structure, we can say that there is something at the top and other things below (or vice versa). So our root directory contains other directories, which can contain more directories themselves. On *Nix, the topmost directory for the OS is usually simply `/` or *"root"*. For windows there is no single topmost directory, but rather a set of drives. The operating system lives on the `C` drive. So you might have `C:\Program Files\Mozilla\Firefox.exe` as a path to a browser. Whereas on Mac it would be `/Applications/Firefox.app`.

Notably, in Windows, if you use the *PowerShell* which is a souped up terminal, you can use the *Nix based commands and backslashes. I recommend using the PowerShell for Python or using the 'Ubuntu shell', which is like running Linux on your Windows computer.

To navigate there you can also use a terminal inside Jupyter Lab. If you select a new Launcher from the side (either the + button, or 'new Launcher' from the 'File' menu), then you can select a new Terminal session. Do that and marvel at how plain the terminal looks. It should look something like this:

```
(base) work@MacBookAir ~ %
```

Then to the right of the % would be a blinking cursor, which is where you navigate the terminal by typing commands. First let's find out where we are.

> As a tip for this section. If you are working on Jupyter Lab, you can drag the tabs at the top so that you can have multiple windows open, so that you can have these notes and a terminal window side-by-side.

The first thing we are going to want to do is discover where we are in the directory structure. This is the 'current working directory'. For a Mac you would type in `pwd` or 'Print Working Directory'. For me, since I named this account 'work' my current working directory is: ~~~ zsh (base) work@Bernies-Air ~ % pwd /Users/work ~~~ This might be considered 'the default directory'. I can use `~` to substitute for this directory, as in `~/Documents` to refer to `/Users/work/Documents`.

In this notebook, I have a different working directory, which is the directory that this notebook was in when I run it. Watch below how to get the current working directory of the notebook:

```
import os
```

```
print( os.getcwd() )
print("The separator on this computer is: ", os.sep)
```

```
/Users/work/Documents/GitHub/FSSDS/introducing_python/chapters
The separator on this computer is:  /
```

This means that if we were to do something like create a file in Jupyter then the file will, by default, be written to this specific directory. Watch that happen below:

```
fileout = open("example_file.txt", 'w')
fileout.write("Here is the example file!\n")
fileout.write("You can open it in a text editor, too.")
fileout.close()
```

You should now see a new file named 'example_file.txt' in the same directory as these notes. To write it to a different directory, you can specify the *absolute path* to that directory. So in my case, since I know that I have an account called `work`, then I can create a file under that directory using the following:

```
fileout = open("/Users/work/example_file2.txt", 'w')
fileout.write("Here is another example file!")
fileout.close()
```

That is a pretty bad form, for two reasons. 1. You probably do not have a `/Users/work/` path on your computer. So when you run the above you would get a `FileNotFoundError`. 2. Writing example files all over your computer will create a mess.

To solve the second problem, I will first clean up my file with the help of `os.remove`, which will delete a file given the path name, or throw an error if the file is not found.

```
try:
    os.remove("/Users/work/example_file2.txt")
except FileNotFoundError:
    pass
```

To solve the first problem is a little tricker, since I want a solution that will work for both my computer and yours. The first thing we can consider, instead of *absolute* paths are *relative* paths. The simplest relative path is `.`, which means "here". So if you see `./file.txt`, that's the same as the file underneath this directory. If you see `..` that means the *parent* directory. In my case, since this is in a folder called `Python`, under `2021MT`, if I wanted to create a new file in the parent directory, `/Users/work/OneDrive - Nexus365/Teaching/2021MT/`, I could write:

```
fileout = open("../example_file2.txt", 'w')
fileout.write("Here is another example file!")
fileout.close()
```

Where this comes in handy is in having a folder for data separate from your notes. For my data courses, I recommend having a folder structure like the following:

```
<course_name>
  |- notebooks
  |- data
  |- output
  |- other
```

It's always a challenging moment when I have to help a student who has their entire course (and most others) running as notebooks from their downloads folder. It might seem like a good idea for the first file, but after a

dozen or so files things will get messy and lost. But importantly, then even if you are working from multiple different files, you always know where to put your data and your output (like charts and tables).

This StackOverflow conversation discusses many of the tricky aspects of getting the path of a specific Python file.

6.1.2 How to navigate the file system through a terminal. (*Nix edition)

This section will be done in a terminal window, so you'll have to switch back and forth. If I write a command it will be preceded by `$` and if I write the expected output, it will be preceded by `>`. So we will want to open a terminal and type `ls`. As in don't type the $, just:

```
$ ls
```

That should list all the files in the current working directory *for the shell*. Notice that it is probably not the same directory as the one you saw above. But let's navigate to our current Python working directory. To do this we use `cd <desired directory>`. In my case (since I have learned the location above) it is:

```
$ cd /Users/work/OneDrive - Nexus365/Teaching/2021MT/Python
```

To shorten the prompt

The prompt might be showing you the very long path to your directory, making it hard to type commands. To shorten it down to just the `$` type the following at the prompt:

```
$ PS1='<label>$'
```

You can then confirm you are still in the directory you expected with `pwd`.

Making a new file

You can make a new empty file here by typing:

```
$ touch temp_file.txt
```

Removing a file.

You can delete a file with `rm` command.

```
$ rm temp_file.txt
```

You can use * to pattern match in the shell. Thus, you can delete multiple files that match a pattern with

```
$ rm *.txt
```

Making a new directory

You can make a new directory by typing `mkdir` followed by the directory name.

```
$ mkdir <directory name>
```

Removing a directory

Directories have files in them. This means that on Windows they cannot be removed in the shell without also removing the files.

```
$ rm -r <directory_name>
```

will only work on an empty directory. If it has files you will need to add the -R (or recursive) argument after the file name.

```
$ rmdir -r <directory_name>
```

The trouble with spaces

You might notice that my file names do not have spaces. This is because when you are entering commands in a terminal, space is considered a break. So a file named `FSDS Week1 Lecture1.txt` would be considered `FSDS` as the file name by the terminal. In order to operate on that file you have to encase it in quotes, which is a nuisance. Such as:

```
$ rm "FSDS Week1 Lecture1.txt"
```

6.1.3 How to navigate the file system through PowerShell (Windows edition)

This is a rewritten section to reflect the fact that the Windows PowerShell should now include Python with the Anaconda install and even launch Jupyter Lab. Where possible, I would stringly encourage you to use the PowerShell over the Anaconda Prompt or the standard `cmd` command line. You can even run Jupyter directly in the PowerShell by typing `jupyter lab` directly in the `Anaconda Power Shell`.

If I write a command it will be preceded by `>` and if I write the expected output, it will be preceded by `|`. So we will want to open a console or 'command line window' and type `dir`. As in don't type the `>`:
~~~ > dir ~~~

or on PowerShell only ~~~ > ls ~~~

That should list all the files in the current working directory *for the shell*. Notice that it is probably not the same directory as the one you saw above. But let's navigate to our current Python working directory. To do this we use `cd <desired directory>`.

To shorten the prompt

The prompt might be showing you the very long path to your directory, making it hard to type commands. Please note that unlike in the Command Prompt, shortening this name, to the best of my knowledge, requires editing the PowerShell profile. If you wish to do this, this StackExchange conversation is useful.

Making a new file

You can make a new file here (just the file name) by typing:

```
> $null > temp_file.txt
```

In this case, $null is the empty character (meaning send 'nothing'). Normally it would send it to standad out (i.e. to the terminal screen). But by using `>` in the terminal we are saying send it to a file.

Removing a file.

You can delete a file with `Remove-Item` or `rm` command. ~~~ > Remove-Item temp_file.txt ~~~

You can use * to pattern match in the shell. Thus, you can delete multiple files that match a pattern with ~~~ > Remove-Item *.txt ~~~

This Microsoft help page goes into greater detail on how to remove items under a variate of conditions.

Making a new directory

You can make a new directory by typing `mkdir` followed by the directory name. ~~~ > mkdir ~~~

Removing a directory

Directories have files in them. This means that on windows they cannot be removed in the shell without also removing the files.

```
rmdir <directory_name>
```
~~~

will only work on an empty directory. If it has files you will need to add the /s argument after the file name.

```
> rmdir <directory_name> /s
| <directory_name>, Are you sure (Y/N)? Y
```

The trouble with spaces

You might notice that my file names do not have spaces. This is because when you are entering commands in a terminal, space is considered a break. So a file named `FSDS Week1 Lecture1.txt` would be considered `FSDS` as the file name by windows. In order to operate on that file you have to encase it in quotes, which is a nuisance. Such as: ~~~ > rm "FSDS Week1 Lecture1.txt" ~~~

However on PowerShell if you type the first character it will then automatically encase it with quotes for you.

6.2 Writing and reading files with Python

6.2.1 Creating files by creating a file 'opener'

In Python, you typically interact with a file by creating a **file opener**. You do this by calling the `open()` command. This command has a number of arguments that determine why you're opening the file: to read? to write? to append? If you forget the argument the default is to open the file for reading. Below we will: - Open a file for writing; - Check if we wrote to it correctly by opening it for reading; - Open it for appending; - Check if we did this correctly by reading it again; - Mercilessly write over our work with more text.

Pay attention to the argument in the `open()` function. For writing it is `'w'`, for reading it is `'r'` and for appending it is `'a'`. There are others as well, but we won't be using them today. They are primarily for **bytestrings** which is relevant if you are writing image data or other streaming data rather than characters. You can review those in the doc strings for the open command

```
# Here is the first line from the Tao Te Ching (trans. Stephen Mitchell)
# It reminds us that in life we can only give guidance but not specific instructions.

str_to_be_written = '''The tao that can be told
is not the eternal Tao.
The name that can be named
is not the eternal Name.'''

# Writing:
fileout = open("example_tao.txt",'w')
fileout.write(str_to_be_written)
fileout.close()

# Did it work? Let's open up the file and find out:
filein = open("example_tao.txt", 'r')
print( filein.read() )
filein.close()
```

```
The tao that can be told
is not the eternal Tao.
The name that can be named
is not the eternal Name.
```

```
# So far so good, but the second line is also very useful. It reminds us
# that operationalisations are always an approximation.
```

```
# The real world is unnamed; it simply is. We create names for our use.

str_to_be_appended = '''
The unnamable is the eternally real.
Naming is the origin
of all particular things.'''

# Appending:
fileout = open("example_tao.txt",'a')
fileout.write(str_to_be_appended)
fileout.flush()
fileout.close()

# Did it work? Let's open up the file and find out:
filein = open("example_tao.txt",'r')
print( filein.read() )
filein.close()
```

```
The tao that can be told
is not the eternal Tao.
The name that can be named
is not the eternal Name.
The unnamable is the eternally real.
Naming is the origin
of all particular things.
```

Note: Remember to flush!

Python might send things off to be written to disk and act as if its job is complete. The operating system might be busy writing other things to disk, however. This means that the operating system could, in some instances, not have written the charaters to disk when you assume it has. By writing `fileout.close()` we ensure that everything has flushed before we move on. But if you are writing a very big file and are worried you can also periodically use `fileout.flush()`.

Note: Be careful. You can actually damage things with Python.

The `os` package is pretty dangerous as you can literally delete files without question. Python, like most programming languages, will **clobber** a file name. To clobber means that it will overwrite a file without asking you first. In actuality, the file hasn't disappeared, but the pointer to that file is lost to the operating system. But in practice, especially on encrypted compuuters, that means it's lost once you overwrite it. **Also...encrypt your computer!** That means FileVault on Mac and BitLocker on Windows.

6.2.2 Reading files in a loop

Often times you will want to read a file line by line rather than use **read()** which dumps the whole file into memory. To do this, you can use the file opener in a loop, as files, like lists and dictionaries, are **iterable**. Try the following:

```
filein = open("example_tao.txt",'r')

for i in filein:
    print (i)
```

```
The tao that can be told
```

```
is not the eternal Tao.

The name that can be named

is not the eternal Name.

The unnamable is the eternally real.

Naming is the origin

of all particular things.
```

That seemed to print every line and then a space, unlike what happened above. Why is that? It's because it prints the *entire line including the new line character at the end*. Remember from day 1 that we can remove characters from a string using `[:-1]`. We can use this to remove the last character. However, sometimes that doesn't work as intended (if there's a `\r\n` for example, which is often the case with excel documents). Luckily, there's a string method called **strip** for removing whitespace characters from the ends of a string. As with most methods (outside of those pesky lists), it *returns* the cleaned string rather than altering the variable in place.

To remove whitespace from both sides:

```
newvar = strvar.strip()
```

To remove it only from the left:

```
newvar = strvar.lstrip()
```

But what we *really* want is to remove the new lines on the right:

```
newvar = strvar.rstrip()
```

```
filein = open("example_tao.txt",'r')

for i in filein:
    print (i.rstrip())

print(filein.closed)
```

```
The tao that can be told
is not the eternal Tao.
The name that can be named
is not the eternal Name.
The unnamable is the eternally real.
Naming is the origin
of all particular things.
False
```

Now while this works, it is not necessarily the most robust or Pythonic way to open a file. For example, we have created a file opener, but we haven't closed it when we finished. Generally, you'll want to close the file when you're done with it using `<filein>.close()`. However, if you are doing something where you are reading the file line by line, you can condense this by using a `with` statement, such as the following. The with statement will automatically close the file when you exit that block of code.

```
with open("example_tao.txt", 'r') as filein:
    for line in filein:
        print(line.rstrip())
```

```
print(filein.closed)
```

```
The tao that can be told
is not the eternal Tao.
The name that can be named
is not the eternal Name.
The unnamable is the eternally real.
Naming is the origin
of all particular things.
True
```

6.3 Running Python in the shell (and Python programs)

6.3.1 Running Python in the console.

Python consoles are available for you on your computer right now. There are at least two! One of them is in JupyterLab and the other is in the shell (sometimes a shell is called a prompt or a CLI for command line interface).

To open the Python console in Windows you should be running the PowerShell. On Mac you would run it through the terminal. You should be able to simply type `python` into either shell and receive a header that looks similar to the following:

```
> python
Python 3.8.3 (default, Jul  2 2020, 11:26:31)
[Clang 10.0.0 ] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

However, it is a little bit slicker to run ipython. It gives you extra colours, tips, etc…

```
>ipython

Python 3.8.3 (default, Jul  2 2020, 11:26:31)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.16.1 -- An enhanced Interactive Python. Type '?' for help.

In [1]:
```

Fun fact: ipython is the ancestor of JupyterLab.

6.3.2 Interacting with the Python shell.

Regardless of which Python console you use, you can now interact with it by tying Python commands one by one and seeing their output

Sometimes this is really nice if you just want to test a single line of code or two. Try typing

```
for i in range(10):
```

And see what happens. Notice that it doesn't execute, because it knows that there's going to be another line after the colon.

You an also use the up and down cursors to navigate through previous statements.

6.3.3 Creating a Python program to run.

If you give `python` a Python filename as an argument then it will execute that file rather than go to the console. So, since we know how to create files, let's create an incredibly simple one "test.py" then run that file and see what happens. First run the code in the block below then copy then type the following in the console (without the '>' of course).

```
> python test.py
```

Of course you will have to navigate to the directory of this file. Remember, you can use `os.getcwd()` as a quick way to determine this directory.

```
# filetext is a string that we write to a program.
# Normally you would edit a program in a text editor, such as SublimeText or Notepad++

import os

filetext = '''print("Hello, yet another world.\\nLet's count to ten!\\n")
for i in range(10): print(i)
'''

fileout = open("test.py",'w')
fileout.write(filetext)
fileout.close()
print("The file can be found by navigating with the following command:\n\ncd \"{}\" ".
 ↪format(os.getcwd()))
```

```
The file can be found by navigating with the following command:

cd "/Users/work/Documents/GitHub/FSSDS/introducing_python/chapters"
```

6.3.4 Running the Python program with file arguments.

A file that you run from the command line might have some arguments that you want to send it. Imagine that it cleans up mp3 tags, you might want to send it the folder of mp3s. In this case we will just give it a word and then print that word, because we're keeping it simple here.

Now the thing about the arguments is that Python can only read them if you use a special command from the `sys` module.

Below I'm going to write a file that takes arguments, then lets switch over to the console and run it.

```
# filetext is a string that we write to a program.
# Normally you would edit a program in a text editor, such as SublimeText or Notepad++

filetext = '''import sys

print('Number of arguments:', len(sys.argv), 'arguments.')
print('Argument List:', str(sys.argv))'''

fileout = open("example_argv.py",'w')
fileout.write(filetext)
fileout.close()
print("The file can be found by navigating with the following command:\n\ncd \"{}\"".
 ↪format(os.getcwd()))
```

The file can be found by navigating with the following command:

```
cd "/Users/work/Documents/GitHub/FSSDS/introducing_python/chapters"
```

Now go to the console and navigate to that folder. Then you can run the command below (omitting the '>'). It will work if there's a `example_argv.py` file in that folder.

```
> python example_argv.py arg1 arg34 YetAnotherArgument "is this also an argument?"
```

You can see that if you encase words in quotes it is counted as one argument. Then the program will be able to access these as a list of arguments in the `sys.argv` object. The first one (`sys.argv[0]`) will be the file name. The next few will be the argument strings written in the commend.

6.3.5 The 'main' statement

A python program can be imported into other programs. It can have a series of functions or just a line of data. For example, if you type a program with `x = "potatoes"` and save it as `veggies.py`, then you can use the following: ~~~ import veggies

print veggies.x > potatoes ~~~

However, sometimes you will want to ensure that the program that you pass to python is an executable program, and in that case, if so, you should do some things in particular. Thus, you will see many Python scripts with the following syntax in them:

```
if __name__ == "__main__":
        <additional code>
```

For example, the `veggies.py` might have a series of useful methods about slicing and dicing veggies. You might want to import them into your food processing program. However, if you run veggies.py as a standalone program from the terminal you would expect it not only to load those modules but to run whatever is included under `"__main__"`.

Try yourself to rename the files en masse using the `os.rename()` method. Be careful only to rename the ones you want to!

6.4 Navigating in Python effectively with `pathlib`

`pathlib` is a library meant to streamline the way paths are managed in Python. To remind, paths are the names for a file location. Paths become 'objects' in Python. Below are some features of `pathlib` and some examples of how to navigate with this library. The most important to remember is that we import a `Path` object. Then we give that object a location. This location may or may not exist on your directory, but it is still a Path object. You could instatiate `Path('dfasdfjlsfgg/fsadgag')` but it would not work if you try to search there for a file. It might either return `None` or `FileNotFoundError` or other error. This is helpful because we often want to check whether a directory exists and if not, then to create it.

Fortunately, the method `Path().exists()` is helpful as it returns `True` or `False`. So if you were on Windows, almost certainly `Path('C:\').exists()` would return `True` and on Mac `Path('/Applications').exists()` would similarly return `True`. `/Applications` is a string that refers to the directory. Path(`/Applications`) is an object.

Path(`.`) is a special object that refers to the current directory.

```
from pathlib import Path

print(Path('/Applications').exists())
print(Path('dfasdfjlsfgg/fsadgag').exists())
```

```
True
False
```

6.4.1 The current directory

The 'current' or current working directory is the directory from which commands are run and where files are stored. So if you type `open('fileout.txt','w')` then it will create a file in this working directory with the name `fileout.txt`. You can change the working directory or navigate to other directories, but by default we act as if the directory that runs the code is the working directory.

To display the working directory in the terminal or PowerShell, you can type `pwd` ("print working directory"). To capture it in Python using the `os` module you would write `os.get_cwd()`. To get it using a Path object, you would write `Path.cwd()`. Let's see all of these in action one after the other:

Using the `os` module

```
# This may or may not work on windows. If it does not, please continue.
import os

print(os.popen('pwd').read())
```

```
/Users/work/Documents/GitHub/FSSDS/introducing_python/chapters
```

Notice that we did not use `os.system` to run the command in the terminal. If you run a temrinal command in `os.system` it will run the command but it will only return a `0` for successful or `-1` for unsuccessful. Since we wanted the result from the terminal we can use `os.popen` which opens a pipe from the terminal to here, so the result of the terminal gets piped in to Python. Then we read that result. It's a bit overcomplicated, which makes sense that Python would find ever simpler ways of doing it.

The first is to use the `os` module directly. Notice that it is now cwd (for current working directory) and not pwd (for print working directory). Because this is an old, old command dating back to Python 1.0, it doesn't use underscores like more recent commands.

```
print(os.getcwd())
```

```
/Users/work/Documents/GitHub/FSSDS/introducing_python/chapters
```

This works well, but there is one problem with this approach. What gets returned is not a path, per se, as in a thing that you can navigate, but a string that represents the *address* to that path. Just watch:

```
result = os.getcwd()
print(type(result))
print(result.split(os.sep))
```

```
<class 'str'>
['', 'Users', 'work', 'Documents', 'GitHub', 'FSSDS', 'introducing_python',
'chapters']
```

We used `os.sep` since that will be the correct separator, whether it is Windows or Mac, but it split the string.

Using `pathlib`

You see you can transform this path just like a string. What might help us is to have a path as an object where we do things like navigate the folder structure. Then you can ask that object for the `stem` (meaning the part of the path with the filename) or add to it using the directory separator `\`. You can check what is

the directory 'above' this one or navigate to a directory below. Notice that this directory separator works the same on Windows and Linux. Let's see that below:

```
from pathlib import Path

print(Path.cwd())
```

```
/Users/work/Documents/GitHub/FSSDS/introducing_python/chapters
```

```
Path.cwd()[:5]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-18-360689f1baf3> in <module>
----> 1 Path.cwd()[:5]

TypeError: 'PosixPath' object is not subscriptable
```

6.4.2 Features of Path

The working directory can now be an object now with it's own methods. In fact, it's a good place to explore what an object does. For that you can use the 'directory' command or `dir`. It will return both system methods (that begin with _ and regular methods. Below I'll use a list comprehension to filter to the non-system methods and display them.

```
curdir = Path.cwd()
print([mthd for mthd in dir(curdir) if mthd.startswith("_") == False])
```

```
['absolute', 'anchor', 'as_posix', 'as_uri', 'chmod', 'cwd', 'drive', 'exists',
'expanduser', 'glob', 'group', 'home', 'is_absolute', 'is_block_device',
'is_char_device', 'is_dir', 'is_fifo', 'is_file', 'is_mount', 'is_reserved',
'is_socket', 'is_symlink', 'iterdir', 'joinpath', 'lchmod', 'link_to', 'lstat',
'match', 'mkdir', 'name', 'open', 'owner', 'parent', 'parents', 'parts',
'read_bytes', 'read_text', 'relative_to', 'rename', 'replace', 'resolve',
'rglob', 'rmdir', 'root', 'samefile', 'stat', 'stem', 'suffix', 'suffixes',
'symlink_to', 'touch', 'unlink', 'with_name', 'with_suffix', 'write_bytes',
'write_text']
```

Listing path objects

A path can refer to either a directory (or a file. A directory contains files and other directories. If directory `A` contains directory `B`, we would say `A` is the parent directory and `B` the child directory. - `<pathobject>.is_dir()` will return `True` if the path is a directory and false otherwise. - `<pathobject>.is_file()` will return `True` if the path is a file and false otherwise.

If the file refers to a directory then you can list the files in the directory with the command `iterdir()`. Let's observe these below:

```
if curdir.is_dir():
    for c,pp in enumerate(curdir.iterdir()):
        if c > 5: break
        print(pp.name)
```

```
Ch.A03.LongerIdeas.ipynb
MASTER_Ch.A.ShortQuestions.ipynb
Ch.04.ConditionalsLoopsErrors.ipynb
test.py
Ch.A01.ShortQuestions.ipynb
Ch.02.DataTypes.ipynb
```

If you want to check for a specific kind of file, you can use the `glob` command, which refers to `global`. There is also the full `glob` module, but nowadays I actually recommend switching to `<pathobject>.glob()` instead. So I strongly suspect that unless you changed your working directory, this notebook is in there and will end in `.ipynb`. Do you have other notebooks in the same folder? Let's inspect with a wildcard search:

```
for i in curdir.glob("Ch.0*.ipynb"):
    print(i.name)
```

```
Ch.04.ConditionalsLoopsErrors.ipynb
Ch.02.DataTypes.ipynb
Ch.03.Collections.ipynb
Ch.07.WhereToNext.ipynb
Ch.06.TheFileSystem.ipynb
Ch.05.FunctionsClasses.ipynb
Ch.01.IntroducingPython.ipynb
Ch.00.Prologue.ipynb
```

Parts of a path

What do you call the specific parts of a path? You can query for the `parent`, `stem`, `suffix`, and the `name` (unshown, but it's `stem` + `suffix`).

```
for i in curdir.glob("Ch.0*.ipynb"):
    print(i.stem, i.suffix, sep=" >>> ")
else:
    print(f"These are in:{i.parent}")
```

```
Ch.04.ConditionalsLoopsErrors >>> .ipynb
Ch.02.DataTypes >>> .ipynb
Ch.03.Collections >>> .ipynb
Ch.07.WhereToNext >>> .ipynb
Ch.06.TheFileSystem >>> .ipynb
Ch.05.FunctionsClasses >>> .ipynb
Ch.01.IntroducingPython >>> .ipynb
Ch.00.Prologue >>> .ipynb
These are in:/Users/work/Documents/GitHub/FSSDS/introducing_python/chapters
```

The parent then is both the directory above and a way to get the part of the path address other than the file name. Thus, you can navigate to the folder above this one and discover what files are in there, too.

```
for i in curdir.parent.iterdir():
    print(i.name);
else:
    print(f"These are in: {i.parent}")
```

```
tex
.DS_Store
chapters
```

```
Untitled.ipynb
build_book.py
.ipynb_checkpoints
example_file2.txt
These are in: /Users/work/Documents/GitHub/FSSDS/introducing_python
```

6.4.3 Recursion: A thorough wildcard search

A wildcard search in one directory will just list the files and directories in that main directory. But what about the files *underneath* those directories. There are a few ways to approach this. One uses a technique called recursion. We can see this in the exercises for next week once we learn how to write functions.

For now, we can skip that with a very clever wildcard: `**/*`

```
for i in curdir.glob("**/*"):
    if i.is_dir():
        print(i)
    else:
        print("FILE_REDACTED", i.suffix,sep=None)
```

```
FILE_REDACTED .ipynb
FILE_REDACTED .ipynb
FILE_REDACTED .ipynb
FILE_REDACTED .py
FILE_REDACTED .ipynb
FILE_REDACTED .ipynb
FILE_REDACTED .txt
FILE_REDACTED .ipynb
FILE_REDACTED .ipynb
FILE_REDACTED .txt
FILE_REDACTED .ipynb
FILE_REDACTED .ipynb
FILE_REDACTED .py
/Users/work/Documents/GitHub/FSSDS/introducing_python/chapters/.ipynb_checkpoint
s
FILE_REDACTED .ipynb
FILE_REDACTED .ipynb
FILE_REDACTED .ipynb
FILE_REDACTED .ipynb
FILE_REDACTED .ipynb
FILE_REDACTED .ipynb
FILE_REDACTED .ipynb
FILE_REDACTED .ipynb
FILE_REDACTED .ipynb
FILE_REDACTED .ipynb
FILE_REDACTED .ipynb
FILE_REDACTED .ipynb
FILE_REDACTED .ipynb
FILE_REDACTED .ipynb
FILE_REDACTED .ipynb
FILE_REDACTED .ipynb
```

The way that operation worked was to use 'recursion'. This meant that it would do the same operation within itself. So for each pathobject, if it was a file it would just return it, but if it was a directory it would then start again, list a file if it was a file, but if it was a directory it would start again, and so forth. One it ran out of directories to list it finishes.

6.4.4 Changing files and directories with `pathlib`

It is a common practice to create a directory in which you store files, or to check if a directory exists and only try to create it if it does not exist already. You can do this with `mkdir(<directory_name>)`. Let's create a directory below this one. And then we will destroy it.

```
(curdir / "temp").exists()
```

```
False
```

```
if (curdir / "temp").exists():
    (curdir / "temp").rmdir()
    print("I removed a directory called temp")
else:
    (curdir / "temp").mkdir()
    print("I made a directory called temp")
```

```
I removed a directory called temp
```

6.5 Conclusion

So you might be a little terrified at just how much you can get away with in Python. You can read and write files all over your computer. Your operating system will sandbox some of this but you really are out in the wild here with the ability to read and alter files on your computer. Hopefully, this skill will help you think of small Python projects you might do on your own.

This is pretty much it for the start of the journey. With these skills you can set out to learn in a variety of directions. The next chapter signposts some places to go. And to remind as usual, in the appendices are exercises. There are no specific exercises for this chapter, but this chapter will really benefit you for the longer exercises which do ask you to think about writing files as well as running scripts.

Chapter 7

Where to next?

7.1 Continuning your Python learning

There's a real challenge for people after just learning the basics of a language. And it's endemic to books teaching Python as well. After the basics they tend to fray outwards in a variety of directions, primarily reflecting the interest of the author. One of the books that I enjoyed in this regard was the now a little dated Python Crash Course by Eric Matthes. In that book after the basics (which frankly are pretty similar to what's included here), he presents projects which are considerably different in scope and technique, such as a video game and a data science project. I think that's the right approach. From here I hope you'll consider these skills as helping you consider a programmatic way of thinking. I don't want to say a new way, as we employ programmatic thinking all the time: sorting the laundry, organising a bookshelf, or preparing dinner step-by-step all have vestiges of algorithmic reasoning.

Now as it happens, algorithmic reasoning gets applied to many areas of human behavior, sometimes to the detriment of those whose behavior it shapes. We hear of cold bureacracies with endless forms, unfair job hiring, or police profiling of certain communities based on 'data'. These are very real and very consequential, down to the price of the food you buy and the quality of the air you breathe. It's hard to fully participate in a world governed so heavily be algorithms without a sense of how to create one yourself. In that sense, I see this sort of work as partially a journey of empowerment. You can ask questions at a different scale with programming. But imagination is also part motivation and part inspiration.

The first thing you'll need on this journey will be the standard tools. Often people sidestep these for searching on Stack Exchange or Google first. But I say start with some basics: *read*, read the documentation particularly. This is not just the specific instructions for a particular module or form of syntax. It includes tutorials and installation guides as well. It's worth being careful, however, that you're mindful of the version for the docs. Python is currently running version 3.10 as of December 2021. Anaconda is running 3.9. It's okay for the version to be a little behind, but since the language does evolve, there are always bound to be some new things. I find that some of the newer things to be really clever solutions to repetitive code issues (like the walrus operator).

One place that's especially useful for understanding how Python evolves are the PEP or Python Enhancement Proposals. One in particular is worth reading whenever you get the chance, PEP-8, which is the recommended style guide. As it happens I do not 100% follow the guide. Or maybe I do? As it says in the style guide: "A Foolish Consistency is the Hobgoblin of Little Minds". It's okay to be a little out of sync with any specific Python style, but it is preferable to be as consistent as possible, particularly within project. In fact, it's interesting to me to read other people's code and understand when they have a 'style' to their programming. To me it really does seem like a voice for some people.

Ok, but these are pretty dry. Read the docs? Seriously? No, seriously. It's kind of like the vegetables of your learning diet. But they're cooked well and pretty seasoned. Beyond that, I've grouped some resources

into different categories. And this is a very partial list.

7.1.1 Websites

Some websites are especially thorough when teaching Python, with pages dedicated to single topics. If you find that the coverage of a topic here (which is often just a paragraph and a single example) is not enough, it's worth having a look at the following sites:

- **w3schools** (https://www.w3schools.com): They have very extensive examples for many specific Python concepts, but many other web technologies as well. The examples are very sparse though. With Python you are often getting a clear view on a single concept rather than a sense of how this concept fits into a larger whole.

- **realpython** (https://realpython.com/): This one is similar to w3schools. It locks away some advanced content for paid subscibers, which is a shame. This tends to be the case for a lot of sites.
- **Towards Data Science** (https://towardsdatascience.com/): This is presently a blog on Medium, so it suffers from the same paid content issues. But I think that the blog posts on this site tend to be the right size for me. Digestable and often well-contextualised. It's not painfully slow and repetitive, but is still slow enough to be clear (at least for topics that are just at the cusp of where my own learning is).
- **Data Carpentry** (https://datacarpentry.org/): This non-profit community-driven site has some excellent tutorials on all skills levels in Python. It think it's both a great resource for it's content but also of how it's produced. That said, it is not as thorough as w3schools or realpython. It's more practice based and for specific concepts. That being said it is a great place to learn about some things just beyond this book.
- **Jake Van Der Plas' Python Repositories**. Jake was early out of the gate with a book that doubles as a Jupyter notebook. He's the author of a Whirlwind Tour of Python, which is very close in spirit and style as this book, and the Python Data Science Handbook, both from O'Reilly. I really want to cheer on these two books especially. Similarly written in Jupyter Notebooks and available on GitHub, they are a great complement, though perhaps with a little less social science flair.

7.1.2 Communities and forums

- **StackExchange**: I suspect this will end up being one of your more regularly visited sites when trying to solve problems in Python. StackOverflow for coding is a vast and useful resources. But be cautious on the site. Read the answer rather than cut-paste-and-hope. Tinker with the example a little before trusting it. It often pays off to know what you're getting into with someone else's code. Notice in the threads that there's sometimes a discussion - this can be worth engaging. People often point out how it works differently on later versions or how it could be simplified.
- **Reddit**: Reddit has a variety of places for learning new programming skills. I find the LearnPython community to be pretty civil and welcoming. Sort the content by top or by top this year to find a lot of walkthroughs, guides, and cheat sheets. The larger /r/Python and /r/DataScience subreddits can show some interesting projects as well, but are often more geared to those with programming experience. See what they are up to as well.

- **Twitter**: Twitter can often surface some real gems that might be hidden otherwise. Follow those practicing the sort of skills you want to learn. If there's an author of a package out there you find useful, check out their feed. Academics are particularly keen to share code and notebooks on Twitter.

Beyond this, tons of social media sites will have resources, news, and discussion about programming.

7.1.3 Online courses

There are a ton of online courses available. I would recommend any courses that are somewhat goal focused or practice based to get you exploring with some hands on experience. But they all have their place to

reinforce your learning. Just know that strictly following someone else's tutorial will not build the creative spark needed to advance your work. Codeacademy, Udemy, and Coursera are all examples of these sorts of sites. Personally, I think the tutorials and help pages for many projects themselves offer a solid foundation for learning a new skill, but there is still some merit to having it structured.

What is most useful to me, however, is having some sort of feedback or results from your work. Thus finding coding communities, hackathons, or user groups might enliven your experience considerably. Not all places are equally welcoming, unfortunately. It pays to shop around if you find a user group or community to be unwelcoming. But don't give up, people of all kinds of attitudes, identities, and ideologies are picking up programming and making it their own.

7.2 Ideas for directions next

Because Python is now so widely diffused, once you get the basics you can go in a huge number of directions. There's full fledged ways in python to make maps (geopandas and pysal), detect objects in images (opencv), do social network analysis (networkx), process text (nltk and spacy), produce a blog (django), and more. Some of these approaches emhpasise different parts of Python. For some, it is about creating a listener that will receive input (like mouse movement), for others it is about managing large streams of data, or interacting with complex objects.

Basically, try attaching pretty much any programming task or domain with "Python" these days and you are likely to find somebody working on it. Try searching around the internet, and especially PyPi, which is a huge repository of Python modules. You'd be surprised at not only how far Python can take you, but how far you can get with the skills herein.

Safe travels and many happy `return` statements.

Appendix A

Short Questions

Below are some short exercises to check your knowledge of the topics introduced in each of the chapters. Each section corresponds to one of the prior chapters in the book. After these are two more appendices. Appendix 2 is just this appendix but with some example code for answers for the questions below. Finally, in Appendix 3 is is a series of longer creative exercises that you might want to attempt with skills from this book. They are marked by which skills you would reasonably need to try your hand at the exercise.

In many cases I have provided some starter code and you should finish that code. You'll see where you should finish with an `...` or a similar sort of marker.

A.1 Chapter 1. Introducing Python

Just one question here: Is this running in Jupyter lab?

A.2 Chapter 2. Data Types

A.2.1 Practicing making strings

The first few exercises are just to warm you up to working with strings.

Debug one

```
print "So this is how we start, eh?"
```

Debug two

```
print("Well, "so far so good", as I like to say")
```

Debug three

```
print("This should be line 1."\n"This should be line 2.")
```

A.2.2 Making a greeting

With this exercise, you should learn about string insertions. We will do them three ways:

1. Using a + to concatenate the strings

2. Using `"<str>{<VAR>}".format()`
3. Using `f"{<var>}"`

All three different approaches should print:

```
<greeting>! My name is <name> and I'm from <origin>. Someday I hope to get to
<destination>, got any suggestions?
```

Remember you can check on https://pyformat.info/

```
greeting = ''
name = ''
origin = ''
destination = ''

# First using a +, as in print(var+var+var...)
st1 = ...

# Second using .format, as in  print("{}".format(vars))

st2 = ...
# Third using f insertions, as in print(f"{var}{var}")

st3 = ...

print(st1 == st2 == st3)
print(st1)
```

A.3 Chapter 3. Collections

Below are some exercises for the chapter on collections.

A.3.1 Building an algorithm to reproduce concrete poetry

There's not much you can do in Python exclusively by printing strings. However, I thought this would be a nice opportunity to produce some concrete poetry. Concrete poetry means the visual arrangement of the words has meaning as do the words. Ian Hamilton Finlay is a Scottish concrete poet. Below I have pasted a version of his poem "acrobats" as excerpted from Cockburn K. and Finlay, A (2001) *The Order of Things*. Edinburgh, UK: Pocketbooks.

```
a   a   a   a   a
  c   c   c   c
r   r   r   r   r
  o   o   o   o
b   b   b   b   b
  a   a   a   a
t   t   t   t   t
  s   s   s   s
```

Using only a variable `word = "acrobats"`, string insertions, spaces, and lists, try to print a reproduction of the poem.

In this version, the answer below should be done *without* `for` loops or `if` statements, which means it will likely have some repetition. Your goal is to minimise that repetition even if you can't eliminate it.

```
# Complete the answer: (I wrote some code to get you started)

word = "acrobats"
print((word[0] + "   ")*5)
...
...;
```

A.3.2 A Table of Muppets

The following questions use a table of values with some details from key Muppet characters in the show "The Muppet Show". We have the data in one form which is just raw text, which we will clean up so we can ask questions about it.

Splitting strings into lists

This data is presented as a string. It is structured as what we call tab-separated values (`.tsv`). Let's change it so that it is a list of lists. Below you can do this without a `for` loop as that is featured in the subsequent chapter. It will be important to think of how you deploy the `.split()` method since you will need to split both by line and then within line.

Doing this without a for loop might involve using nine repetitions of basically the same code. So I will show that and I will also include a `for` loop version.

Your final data structure should have nine elements. Each one will be a row with data. If you have 11 check that you have not included lines for the empty top and bottom lines of the text.

```
muppet_text = '''
name        gender        species        first_appearance
Fozzie        Male        Bear        1976
Kermit        Male        Frog        1955
Piggy        Female        Pig        1974
Gonzo        Male        Unknown        1970
Rowlf        Male        Dog        1962
Beaker        Male        Muppet        1977
Janice        Female        Muppet        1975
Hilda        Female        Muppet        1976
'''

# Complete this answer:

muppet_list = [...]*9 #Replace [...]*9 with your answer

muppet_list[0] = ...
muppet_list[1] = ...


print(muppet_list)
```

Separating the header from the rest.

Take muppet_list and then slice it so that you have two lists: `muppet_header` is only of length one, it's the header. `muppet_data` is the other list and contains the remianing elements. Check that the length of `muppet_data` is 8.

```
# Answer
muppet_header = ...
muppet_data = '...'

print(f"It is {len(muppet_data) == 8} that the Muppet Data has 8 rows")
```

Transforming the `muppet_data` into a `muppet_dict`

At this point, we should have 8 lists in a data structure called `muppet_data`. The first element in this list is the name, followed by three data points (`gender`, `species`, `first_appearance`).

Transform each line into a dictionary entry so that the whole dictionary will look something like this:

```
muppet_dict = {"Fozzie":["Male","Bear",1976],
               "Kermit": ...,
               ...}'''
```

To create this dictionary you might need to repeat lines of code while only changing the indices.

This will be the last repetitive code example to complete. I will give fewer instructions here. If you know loops, you can try them here, but I am assuming you have not skipped to chapter 3. In case you have, know that I provide two answers to this in the next appendix. One with and one without loops.

```
# Answer

m_dict = {}

m_dict[muppet_data[0][0]] = ...
m_dict[muppet_data[1][0]] = ...
...

print(m_dict)
print(f"It is {len(m_dict.keys())==8} that the muppet_dict has 8 keys.")
```

Query the muppet data (Tougher bonus challenge)

Use the following code pattern:

```
user_input = input("Which muppet do you want to profile:")
```

Then take the data from `user_input` and print a profile of the muppet in the following form: Character: Fozzie Profile: Gender: Male Species: Bear First Appearance: 1976 Consider printing a list of all muppets (i.e. all keys from the dictionary) before asking for user input so the user can get the correct spelling. You might find other ways to make this robust, especially after reading in the later chapters.

```
user_input = input("Which muppet do you want to profile:")

print(...)
```

A.4 Flow control

A.4.1 Fozzie Bear!

This is based on the classic coding challenge, "Fizz Buzz". To cheat or compare answers see: https://wiki.c2.com/?FizzBuzzTest. I, like many instructors, like to use FizzBuzz because you cannot simply make it work with one loop and one if statement. Here goes:

Make a program that spits out numbers and the words Fozzie Bear.

- If the line is a multiple of 3 print the line number + Fozzie, like '6. Fozzie
- If the line is a multiple of 5 print the line number + Bear, like `10. Bear`
- If the line is a multiple of both, print a line number + both words, like `15. Fozzie Bear`
- Otherwise do not print anything.

Have the program run in the range 1 to 30 inclusive (so I should read `30. Fozzie Bear` as the final line.

```
# Answer below here:
```

A.4.2 List (and dictionary) comprehension practice

```
# 3a. Loop 1. The simplest example.

ex_list = []
for i in range(1,10):
    ex_list.append(i)

# List comprehension

lc_ex_list = ...


# Check yout answer: (should be True)
print(lc_ex_list == ex_list)
```

```
# 3b. Loop 2. An example with an if statement (i.e. a 'conditional')

every_second_list = []
for i in range(1,10):
    if i%2 == 0:
        every_second_list.append(i)

# List comprehension

lc_every_second_list = ...

# Check your answer: (should be True)
print(lc_every_second_list == every_second_list)
```

```
# 3c. Loop 3. An example with calculation

powers_of_two_list = []
for i in range(10):
    powers_of_two_list.append(i**2)

# List comprehension

lc_powers_of_two_list = ...


# Check your answer: (Should be True)
```

```
print(lc_powers_of_two_list == powers_of_two_list)
```

```
# 3d. Loop 4. A Dictionary Comprehension

old_list = ["zeroith","first","what's zeroith?","Am I third or fourth?"]
new_dict = {}

for c,i in enumerate(old_list):
        new_dict[c] = i

# Dictionary comprehension


dc_new_dict = ...

# Check your answer: (Should be True)
print(new_dict == dc_new_dict)
```

A.4.3 Code refactoring I

In addition to these, just a reminder that all of the exercises in the previous section (for the chapter on collections) that have repetitive code can benefit from loops. Have a look at the answers for these and see if you can refactor them to use loops. The answers with loops are provided below here

Concrete poetry with a for loop

```
word = "acrobats"

...
```

The Muppets data cleaning with for loops

Recall this one had several steps. Try doing them all in a single cell to get from `muppet_text` to `muppet_dict`.

```
muppet_text = '''
name        gender        species        first_appearance
Fozzie        Male        Bear        1976
Kermit        Male        Frog        1955
Piggy        Female        Pig        1974
Gonzo        Male        Unknown        1970
Rowlf        Male        Dog        1962
Beaker        Male        Muppet        1977
Janice        Female        Muppet        1975
Hilda        Female        Muppet        1976'''
```

Making the profile display more robust

Try then to do the profiling code with a `while` statement for user input. Here you can now use elif statements to do somethings in different cases, such as check for valid input and keep going until the user types `quit` or `x`, etc.

```
while ...:
    user_input = input("Which muppet do you want to profile:(x to quit)")

    ...

    break
```

A.5 Chapter 5. Functions and classes

A.5.1 Who said programming was better than flipping burgers?

Flipping burgers can be fast-paced and stressful. But poor order tickets can make it harder. Let's build a function to produce clear order tickets.

All hamburgers will have a bun and a patty. - The default bun = "white" - The default patty = "beef" - Some hamburgers have additional toppings, they will be sent as a list e.g., toppings = ["cheese", "lettuce"]

HINT: Comment out parts of the code below END ANSWER while you are building your function. Start with the simple default burger and work your way towards to other burgers.

```
# Answer Below here.

def burger_order():
    ...
    return
```

```
# Testing code. Check the output of this code with the strings provided.

default_burger = burger_order()
print(default_burger)
# output should be:
'''
***Burger Order***

Bun: white
Patty: beef
'''

cheese_burger = burger_order(toppings = ["chesse"])
print(cheese_burger)
# output should be:
'''
***Burger Order***

Bun: white
Patty: beef
Extras:
- cheese
'''

super_burger = burger_order(bun="whole wheat",toppings␣
↪=["cheese","lettuce","tomato","pickle"])
```

```
print(super_burger)
# output should be:
'''
***Burger Order***

Bun: whole wheat
Patty: beef
Extras:
- cheese
- lettuce
- tomato
- pickle
'''

chicken_burger = burger_order(toppings=["lettuce","tomato"],patty = "chicken")
print(chicken_burger)
# output should be:
'''
***Burger Order***

Bun: white
Patty: chicken
Extras:
- lettuce
- tomato
'''

health_burger = burger_order("gluten-free","veggie",["lettuce","tomato","pickle"])
print(health_burger)
# output should be:
'''
***Burger Order***

Bun: gluten-free
Patty: veggie
extras:
- lettuce
- tomato
- pickle
''';
```

Some extensions to this include:

- Give each order a number. Try to remember the previous order number.
- What about using wildcard `kwargs` arguments in order to allow for any topping?
- What about set burger types? How might these be best expressed?

Appendix B

Short Questions with Example Answers

B.1 Chapter 1. Introducing Python

B.2 Chapter 2. Data Types

B.2.1 Practicing making strings

The first few exercises are just to warm you up to working with strings.

Debug one

```
print("So this is how we start, eh?")
```

Debug two

```
print("Well, \"so far so good\", as I like to say")
```

Debug three

```
print("This should be line 1.\nThis should be line 2.")
```

B.2.2 Making a greeting

With this exercise, you should learn about string insertions. We will do them three ways: 1. Using a + to concatenate the strings 2. Using `"<str>{<VAR>}".format()` 3. Using `f"{<var}"`

All three different approaches should print:

```
<greeting>! My name is <name> and I'm from <origin>. Someday I hope to get to
<destination>, got any suggestions?
```

Remember you can check on https://pyformat.info/

```
# Answer

```

```
greeting = 'Greetings Earthlings'
name = 'Lrrr'
origin = 'Omacron Persei 8'
destination = 'Earth'

# First using a +, as in print(var+var+var...)

st1 = greeting + "! My name is "+ name + " and I'm from " + origin + ". Someday I hope␣
 ↪to get to " + destination + ", got any suggestions?"

# Second using .format, as in  print("{}".format(vars))

st2 = "{}! My name is {} and I'm from {}. Someday I hope to get to {}, got any␣
 ↪suggestions?".format(greeting, name, origin, destination)

# Third using f insertions, as in print(f"{var}{var}")

st3 = f"{greeting}! My name is {name} and I'm from {origin}. Someday I hope to get to␣
 ↪{destination}, got any suggestions?"

print(st1 == st2 == st3)
print(st1)
```

B.3 Chapter 3. Collections

Below are some exercises for the chapter on collections.

```
a   a   a   a   a
  c   c   c   c
r   r   r   r   r
  o   o   o   o
b   b   b   b   b
  a   a   a   a
t   t   t   t   t
  s   s   s   s
```

```
# Answer using only list lookups

word = "acrobats"
print((word[0] + "   ")*5)
print("  " + (word[1] + "   ")*4)
print((word[2] + "   ")*5)
print("  " + (word[3] + "   ")*4)
print((word[4] + "   ")*5)
print("  " + (word[5] + "   ")*4)
print((word[6] + "   ")*5)
print("  " + (word[7] + "   ")*4)
```

B.3.1 A Table of Muppets

The following questions use a table of values with some details from key Muppet characters in the show "The Muppet Show". We have the data in one form which is just raw text, which we will clean up so we can

ask questions about it.

```
# Example answer (without for loop)
muppet_list = muppet_text.strip().split("\n")

muppet_list[0] = muppet_list[0].split("\t")
muppet_list[1] = muppet_list[1].split("\t")
muppet_list[2] = muppet_list[2].split("\t")
muppet_list[3] = muppet_list[3].split("\t")
muppet_list[4] = muppet_list[4].split("\t")
muppet_list[5] = muppet_list[5].split("\t")
muppet_list[6] = muppet_list[6].split("\t")
muppet_list[7] = muppet_list[7].split("\t")
muppet_list[8] = muppet_list[8].split("\t")
print(muppet_list)
```

Separating the header from the rest.

Take muppet_list and then slice it so that you have two lists: `muppet_header` is only of length one, it's the header. `muppet_data` is the other list and contains the remianing elements. Check that the length of `muppet_data` is 8.

```
# 2.3 Separating the header from the rest.

# Answer
muppet_header = muppet_list[0]
muppet_data = muppet_list[1:]

print(f"It is {len(muppet_data) == 8} that the Muppet Data has 8 rows")
```

Transforming the `muppet_data` into a `muppet_dict`

```
# Answer

m_dict = {}

m_dict[muppet_data[0][0]] = muppet_data[0][1:]
m_dict[muppet_data[1][0]] = muppet_data[1][1:]
m_dict[muppet_data[2][0]] = muppet_data[2][1:]
m_dict[muppet_data[3][0]] = muppet_data[3][1:]
m_dict[muppet_data[4][0]] = muppet_data[4][1:]
m_dict[muppet_data[5][0]] = muppet_data[5][1:]
m_dict[muppet_data[6][0]] = muppet_data[6][1:]
m_dict[muppet_data[7][0]] = muppet_data[7][1:]

print(f"It is {len(m_dict.keys())==8} that the muppet_dict has 8 keys.")
```

Query the muppet data (Tougher bonus challenge)

```
# Example answer

user_input = input("Which muppet do you want to profile:")
```

```
gen = m_dict[user_input][0]
sp = m_dict[user_input][1]
fa = m_dict[user_input][2]
print(f"Character:\n\t{user_input}\nProfle:\n\tGender: {gen}\n\tSpecies: {sp}\n\tFirst␣
 ↪Appearance: {fa}")
```

B.4 Flow control

B.4.1 Fozzie Bear!

```
# Answer below here:
for i in range(1, 31):
    if not i % 3:
        if not i % 5 :
            print(f'{i}. Fozzie Bear')
        else:
            print(f'{i}. Fozzie')
    elif i % 5 == 0:
        print(f'{i}. Bear')
```

B.4.2 List (and dictionary) comprehension practice

```
# 3a. Loop 1. The simplest example.

ex_list = []
for i in range(1,10):
    ex_list.append(i)

# List comprehension

lc_ex_list = [i for i in range(1,10)]


# Check yout answer: (should be True)
print(lc_ex_list == ex_list)
```

```
# 3b. Loop 2. An example with an if statement (i.e. a 'conditional')

every_second_list = []
for i in range(1,10):
    if i%2 == 0:
        every_second_list.append(i)

# List comprehension
lc_every_second_list = [i for i in range(1,10) if i%2 == 0]

# Check your answer: (should be True)
print(lc_every_second_list == every_second_list)
```

```
# 3c. Loop 3. An example with calculation

powers_of_two_list = []
for i in range(10):
    powers_of_two_list.append(i**2)

# List comprehension

lc_powers_of_two_list = [i**2 for i in range(10)]


# Check your answer: (Should be True)
print(lc_powers_of_two_list == powers_of_two_list)
```

```
# 3d. Loop 4. A Dictionary Comprehension

old_list = ["zeroith","first","what's zeroith?","Am I third or fourth?"]
new_dict = {}

for c,i in enumerate(old_list):
        new_dict[c] = i

# Dictionary comprehension


dc_new_dict = {c:i for c,i in enumerate(old_list)}

# Check your answer: (Should be True)
print(new_dict == dc_new_dict)
```

B.4.3 Code refactoring I

In addition to these, just a reminder that all of the exercises in the previous section (for the chapter on collections) that have repetitive code can benefit from loops. Have a look at the answers for these and see if you can refactor them to use loops. The answers with loops are provided below here

Concrete poetry with a for loop

```
# Answer using for loops

for c,w in enumerate(word):
    if c%2==1:
        print("  " + (w + "   ")*4)
    else:
        print((w + "   ")*5)
```

```
# Here's the densest I can make it.
for c,i in enumerate("acrobats"):
    print(f"{i}   "*5 if c%2 == 0 else "  " + f"{i}   "*4)
```

The Muppets data cleaning with for loops

Recall this one had several steps. Try doing them all in a single cell to get from `muppet_text` to `muppet_dict`.

```
muppet_text = '''
name        gender        species        first_appearance
Fozzie        Male        Bear        1976
Kermit        Male        Frog        1955
Piggy        Female        Pig        1974
Gonzo        Male        Unknown        1970
Rowlf        Male        Dog        1962
Beaker        Male        Muppet        1977
Janice        Female        Muppet        1975
Hilda        Female        Muppet        1976'''
```

```
# Example answer with some for loops

m_dict = {}
header_row = True

for row in muppet_text.strip().split("\n"):
    if header_row:
        muppet_header = row.split("\t")
        header_row = False
        continue
    row = row.split("\t")
    m_dict[row[0]] = row[1:]

m_dict
```

```
# Example answer (with for dictionary comprehension)
# Notice with this one you have to do the header row separately.
# I just excluded that from here.

m_dict = {i.split("\t")[0]:i.split("\t")[1:]
              for i in muppet_text.strip().split("\n")[1:]}

print(m_dict)
```

Making the profile display more robust

Try then to do the profiling code with a `while` statement for user input. Here you can now use elif statements to do somethings in different cases, such as check for valid input and keep going until the user types `quit` or `x`, etc.

```
while True:
    user_input = input("Which muppet do you want to profile:(x to quit)")

    if user_input.lower() == "l":
        print ("\n".join(m_dict.keys()))
    elif user_input.lower() == "x":
        break
    elif user_input in m_dict.keys():
```

```
        gen = m_dict[user_input][0]
        sp = m_dict[user_input][1]
        fa = m_dict[user_input][2]
        print(f"Character:\n\t{user_input}\nProfle:\n\tGender: {gen}\n\tSpecies␣
↪{sp}\n\tFirst Appearance: {fa}")
    else:
        print("That was not a valid name. Type L to see names of muppets.")
```

B.5 Chapter 5. Functions and classes

B.5.1 Who said programming was better than flipping burgers?

```
# Answer Below here.

def burger_order(bun = "white", patty = "beef", toppings = []):
    receipt = "\n***Burger Order***\n\n\n"

    receipt += "Bun: %s\n" % bun
    receipt += "Patty: %s\n" % patty
    if len(toppings) > 0:
        receipt += "Extras:\n"
        for i in toppings:
            receipt += "- %s\n" % i

    return receipt


#********** END ANSWER ************
```

```
# Testing code. Check the output of this code with the strings provided.

default_burger = burger_order()
print(default_burger)
# output should be:
'''
***Burger Order***

Bun: white
Patty: beef
'''

cheese_burger = burger_order(toppings = ["chesse"])
print(cheese_burger)
# output should be:
'''
***Burger Order***

Bun: white
Patty: beef
Extras:
- cheese
'''
```

```
super_burger = burger_order(bun="whole wheat",toppings␣
↪=["cheese","lettuce","tomato","pickle"])
print(super_burger)
# output should be:
'''
***Burger Order***

Bun: whole wheat
Patty: beef
Extras:
- cheese
- lettuce
- tomato
- pickle
'''

chicken_burger = burger_order(toppings=["lettuce","tomato"],patty = "chicken")
print(chicken_burger)
# output should be:
'''
***Burger Order***

Bun: white
Patty: chicken
Extras:
- lettuce
- tomato
'''

health_burger = burger_order("gluten-free","veggie",["lettuce","tomato","pickle"])
print(health_burger)
# output should be:
'''
***Burger Order***

Bun: gluten-free
Patty: veggie
extras:
- lettuce
- tomato
- pickle
''';
```

Appendix C

Longer Project Ideas

These are some of the longer creative questions that I have used in classes in the past. These normally involve combining multiple ideas from the book in creative ways as well as perhaps drawing upon some knowledge beyond the book. I don't have specific answers to these questions available, they are for you to explore on your own.

C.1 Mad Libs

The game Mad Libs involves first coming up with random words based on parts-of-speech tagging (verbs, nouns, pronouns, adverbs, etc…) and then putting them into a pre-given sentence. Such as:

```
{Proper noun} was so surprised when they saw the {noun} {verb continuous} in town square,
they immediately packed their bags full of {plural noun} and left for {Geographic location}.
```

Your goal is to make a mad libs game. You should use loops to ask for user input five times, each representing a different class of word. It need not be formal grammar; it could be "restaurant", "Movie", etc. but it certainly is more fun when you expand beyond nouns. Then print the sentence for the user. Try to make the order that you ask for the five words is *not* the same as the order in which they are presented.

C.1.1 Comparing Pseudocode to the real thing

For this one, here is a pseudocode version of my answer to get you started:

```
get a dictionary for the "libs".
The key is the type of word and the value will come from the user.
We will set the value as none for now.
Since dictionary keys need to be unique and we might want two nouns or two places:
label them noun1 and noun2

Figure out a way to iterate through a shuffled dictionary.
Since we cannot shuffle a dictionary directly:
 First get the keys from the dictionary,
 Shuffle the keys, and then iterate through those keys.

On each iteration, assign a value to the dictionary by asking for user input.
The user input should use the key in the prompt like
 "Please suggest a noun for the story"

Then use f-insertions to paste the answers in the story.
```

```
# Example answer

import random

answer_dict = {"plural noun1": None,
               "verb continuous1": None,
               "noun1": None,
               "place1": None,
               "place2": None,
               "proper noun1": None
              }

libs = list(answer_dict.keys())
random.shuffle(libs)

for i in libs:
    answer_dict[i] = input(f"Please suggest a {i[:-1]} for the story:")

ad = answer_dict

story = f'{ad["proper noun1"]} was so surprised when they saw the {ad["noun1"]}␣
 ↪{ad["verb continuous1"]} in {ad["place1"]}, they immediately packed their bags full␣
 ↪of {ad["plural noun1"]} and left for {ad["place2"]}.'

print(story)
```

C.1.2 Making it more robust or more general

Ok so that was pretty simple. And you didn't have to do much other than run my code. But what about the following extensions: - Creating a general way to insert some text and then create the madlib. Here we created a dictionary and then linked that to the string. What about a way to do it so the dictionary is not hard coded somehow? - Try feeding it text from a text file that you have marked up with the madlibs in a form like: `Hello ##Proper noun##, did I see you at ##Place name##.` And then auto-generating the madlib from this. - Learning about parts-of-speech taggers in programs like `nltk` and spacy. Then feed in any paragraph of text and automatically remove some of the words and generate a madlib that way.

C.2 Creating a word waterfall

Just past the title page of this book is a word waterfall (my own idea) with "Introducing Python". In this idea, each iteration a letter is replaced with a space.

Below I have written two different ways to create a word waterfall algorithmically.

Read them both, edit and play with them as you like.

Then below write the following: 1. A pseudocode explanation of algorithm 1. 2. A pseudocode explanation of algorothm 2. 3. An evaluation of algorithms 1 and 2: What is common about them and what is different?

```
import random

WORD = "waterfall"
```

```
word = list(WORD)

wordmap = list(range(len(word)))
random.shuffle(wordmap)
for i in wordmap:
    print("".join(word))
    word[i] = " "
```

```
waterfall
 aterfall
 aterfal
 a erfal
 a er al
 a er  l
   er  l
   er
    r
```

```
word = list(WORD)

while word != (len(word) * [" "]):
    to_del = random.randint(0,len(word)-1)
    if word[to_del].isalpha():
        print("".join(word))
        word[to_del] = " "

print()
```

```
waterfall
waterfal
wat rfal
wat rf l
wat rf
 at rf
  t rf
  t r
    r
```

C.2.1 Pseudocode, similarities, and differences

For the `for` loop waterfall:

`insert pseduocode here`

For the `while` loop waterfall

`insert pseudocode here`

Discuss the differences and advantages below:

C.2.2 Create a `waterfall.py` script

Create a script in python called `waterfall.py` which can be run from the terminal.

The script will take an argument and then print the argument as a waterfall:

```
% python waterfall.py avalanche
```

Should print something like:

```
avalanche
avala che
a ala che
a  la che
a  l  che
   l  che
      che
      ch
       h
```

C.2.3 Create your own waterfalls in a script.

Extend the script above with 'interactive mode'. This means that if your argument is `-i` instead of a word then it will ask a series of questions:

1. `What word or phrase would you like to waterfall?`:

- This will then print the waterfall from `input()`

2. `Would you like to save the waterfall as a file? (y/n)`:

- If the user enters `y` this will then save `<WORD>.txt` as a file.
- This means you have to save the waterfall as a string just in case you need it here to be written to file.
- `print(f"{<WORD>} has been written")`

C.3 Your very own restaurant on YummyNet

After the fall of Deliveroo, Uber Eats, and JustEats for questionable labour standards in the near future, YummyNet emerged as a more considerate albeit more expensive platform for restaurants. Having had limited success with your chatbots for pets app, you've decided to switch careers and open a restaurant.

For this exercise, you have to make a standalone Python program (a `*.py` script) that will simulate the store front for this resutaurant. But times are tough and goods are scarce, so this is going to be a bit of an unpridctable menu.

We expect the following steps:

1. Create a welcome message and ask for the user's post code.
2. List the items on the menu for delivery and their prices.

3. Have the user select an order for delivery from the items on the menu.

- There should be at least five mains and three sides. The mains should be made of multiple ingredients.
- when the program starts, randomly select one ingredient to be 'sold out due to those panic buyers' (or a similarly plausible reason for an ingredient to be sold out). If the user selects an item with that ingredient mention that it is sold out and offer an alternative.

4. When finished, print the order, an order number, the total price, and the delivery time.
5. Ask them to confirm by giving their mobile phone number.

Note:

- Consider what would be the maximum stock available. We will test by making a ridiculously large order.

- Items contain multiple (often overlapping) ingredients. So if cheese is sold out, then cheeseburgers and lasagna would not be available, but meatballs would still be available).
- Ensure that you can recommend at least one meal to the customer.
- You can make a larger menu but remember to test it for legibility and usability.

Challenge Store the order in a file before the program exits and give the user an 'order number'. Then if they run the program again and present the correct order number (and the correct telephone number) it will print a duplicate receipt.

Some things you'll want to consider about your data structure:

- How are you storing the order? As a list? As an object of the Order class?
- What will you do with bad input and how will you make the text input as easy as possible?

- How will you manage the inventory so that you will know which goods include which ingredients?
- What if you had to expand the list of menu items. How hard would that be with your program?
- Does the program end gracefully?
- Will your printing of output be attractive and easy to read?