sitepoint

Useful Python

Make your life easier with practical Python techniques

Useful Python

Copyright © 2023 SitePoint Pty. Ltd.

Ebook ISBN: 978-1-925836-57-8

- **Author:** Stuart Langridge
- **Technical Editor:** Cláudio Ribeiro
- **Product Manager:** Simon Mackie
- **English Editor:** Ralph Mason
- **Cover Designer:** Mark O'Neill

Notice of Rights

Notice of Liability

Trademark Notice

Published by SitePoint Pty. Ltd.

10-12 Gwynne St, Cremorne, VIC, 3121
Australia
Web: www.sitepoint.com
Email: books@sitepoint.com

About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit http://www.sitepoint.com/ to access our blogs, books, newsletters, articles, and community forums. You’ll find a stack of information on JavaScript, PHP, design, and more.

About the Author

Stuart is a consultant CTO, software architect, and developer to startups and small firms on strategy, custom development, and how to best work with the dev team. Code and writings are to be found at kryogenix.org and social networks; Stuart himself is mostly to be found playing D&D or looking for the best vodka Collins in town.

PREFACE

Who Should Read This Book?

In this series of tutorials, we're not looking at data science. That is, this isn't about doing heavy statistical or numerical calculations on data we've received. Python is one of the industry-standard tools for doing calculations like these—using libraries such as NumPy and pandas—and there are plenty of resources available for learning data science.

In this series, we'll looking at how to convert data from one form to another so that we can then go on to manipulate it.

We're also going to assume a little knowledge of Python and programming already—such as what a variable is, what a dictionary is, and how to import a module.

Conventions Used

Code Samples

Code in this book is displayed using a fixed-width font, like so:

```
<h1>A Perfect Summer's Day</h1>
<p>It was a lovely day for a walk in the park.
The birds were singing and the kids were all back at school.</p>
```

You'll notice that we've used certain layout styles throughout this book to signify different types of information. Look out for the following items.

Tips, Notes, and Warnings

Hey, You!

Tips provide helpful little pointers.

Ahem, Excuse Me ...

Notes are useful asides that are related—but not critical—to the topic at hand. Think of them as extra tidbits of information.

Make Sure You Always ...

... pay attention to these important points.

Watch Out!

Warnings highlight any gotchas that are likely to trip you up along the way.

Supplementary Materials

- https://www.sitepoint.com/community/ are SitePoint’s forums, for help on any tricky problems.
- **books@sitepoint.com** is our email address, should you need to contact us to report a problem, or for any other reason.

For Bruce.

CHAPTER 1: PYTHON AS GLUE

Python is a versatile and powerful language that can be used for a wide variety of tasks. One of the most common use cases for Python is as a “glue” language: it helps us combine skills and programs we already know how to use by allowing us to easily convert data from one format to another. This means that we can take data in one format that we don’t have tools to manipulate and change it into data for tools that we’re comfortable with. Whether we need to process a CSV, web page, or JSON file, Python can help us get the data into a format we can use.

For example, we might use Python to pull data from a web page and put it into Excel, where we already know how to manipulate it. We might also read a CSV file downloaded from a website, calculate the totals from it, and then output the data in JSON format.

Who This Series is For

In this series of tutorials, we’re not looking at data science. That is, this isn’t about doing heavy statistical or numerical calculations on data we’ve received. Python is one of the industry-standard tools for doing calculations like these—using libraries such as NumPy and pandas—and there are plenty of resources available for learning data science (such as Become a Python Data Scientist).

In this series, we’ll looking at how to convert data from one form to another so that we can then go on to manipulate it.

We’re also going to assume a little knowledge of Python and programming already—such as what a variable is, what a dictionary is, and how to import a module. To start learning Python (or any other programming language) from scratch, check out SitePoint’s programming tutorials. The Python wiki also has a list of Python programming tutorials for programmers.

Getting Started

We’ll start this tutorial by looking at how to read data and then how to write it to a different format.

Reading Data

Let’s try an example. Let’s imagine we’re an author on a tour of local libraries, talking to people about our books. We’ve been given `plymouth-libraries.json`—a JSON file of all the public libraries in the town of Plymouth in the UK—and we want to explore this dataset a little and convert it into something we can read in Excel or Google Sheets, because we know about Excel.

First, let’s read the contents of the JSON file into a Python data structure:

```
import json
with open("plymouth-libraries.json") as fp:
    library_data = json.load(fp)
```

Now let’s explore this data a little in Python code to see what it contains:

```
print(library_data.keys())
```

This will print `dict_keys(['type', 'name', 'crs', 'features'])`, which are the top-level keys from this file.

Similarly:

```
print(library_data["features"][0]["properties"]["LibraryName"])
```

This will print `Central Library`, which is the `LibraryName` value in `properties` for the first entry in the features list in the JSON file.

This is the most basic, and most common, use of Python's built-in json module: to load some existing JSON data into a Python data structure (usually a Python dictionary, or nested set of Python dictionaries).

Bear in mind that, to keep these examples simple, this code contains no error checking. (Check out A Guide to Python Exception Handling for more on that.) But handling errors is important. For example, what would happen if the `plymouth-libraries.json` file didn't exist? What we do in that situation depends on how we should react for errors. If we're running this script by hand, Python will display the exception that occurs—in this case, a `FileNotFoundError` exception. Simply seeing that exception may be enough; we may not want to "handle" this in code at all:

```
$ python load-json.py
Traceback (most recent call last):
  File "/home/aquarius/Scratch/fail.py", line 13, in <module>
    open("plymouth-libraries.json")
FileNotFoundError: [Errno 2] No such file or directory: 'plymouth
```

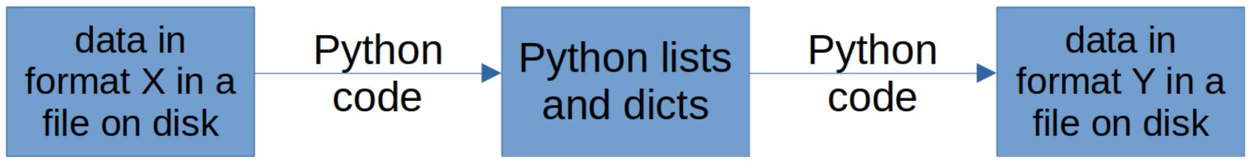
If we’d like to do something more than have our program terminate with an error, we can use Python’s try and except keywords (as the exception handling article above describes) to do something else of our choosing. In this case, we display a more friendly error message and then exit (because the rest of the program won’t run without the list of libraries!):

```
try:
    with open("plymouth-libraries.json") as fp:
        library_data = json.load(fp)
except FileNotFoundError:
    print("I couldn't find the plymouth-libraries.json file!")
    sys.exit(1)
```

Writing

Now we want to write that data from its Python dictionary into a different format on disk, so we can open it in Excel. For now, let’s use CSV format, which is a very simple file format that Excel understands. (If you’re thinking, “Hey, why don’t we make it a full Excel file!” … then read on. CSV is simpler, so we’ll do that first.) This process of taking Python data structures and writing them out as some file format is called **serialization**. So we’re going to serialize the data we read as JSON into CSV format.

The image below demonstrates the stages involved in serialization.


data in format X in a file on disk
Python code
Python lists and dicts
Python code
data in format Y in a file on disk

A **CSV file** is a text file of tabular data. Each row of the table is one line in the CSV file, with the entries in the row separated by commas. The first line of the file is a list of column headings.

Consider a set of data like this:

Animal	Leg count	Furry?
Cat	4	Yes
Cow	4	No
Snake	0	No
Tarantula	8	Yes

This data could look like this as a CSV file:

```
Animal,Leg count,"Furry?"
Cat,4,Yes
Cow,4,No
Snake,0,No
Tarantula,8,Yes
```

To write out a CSV file, we need a list of column header names. Fortunately, these will be the keys of the properties of the first entry in `"features"`, since all libraries have the same keys:

```
header_names = library_data["features"][0]["properties"].keys()
```

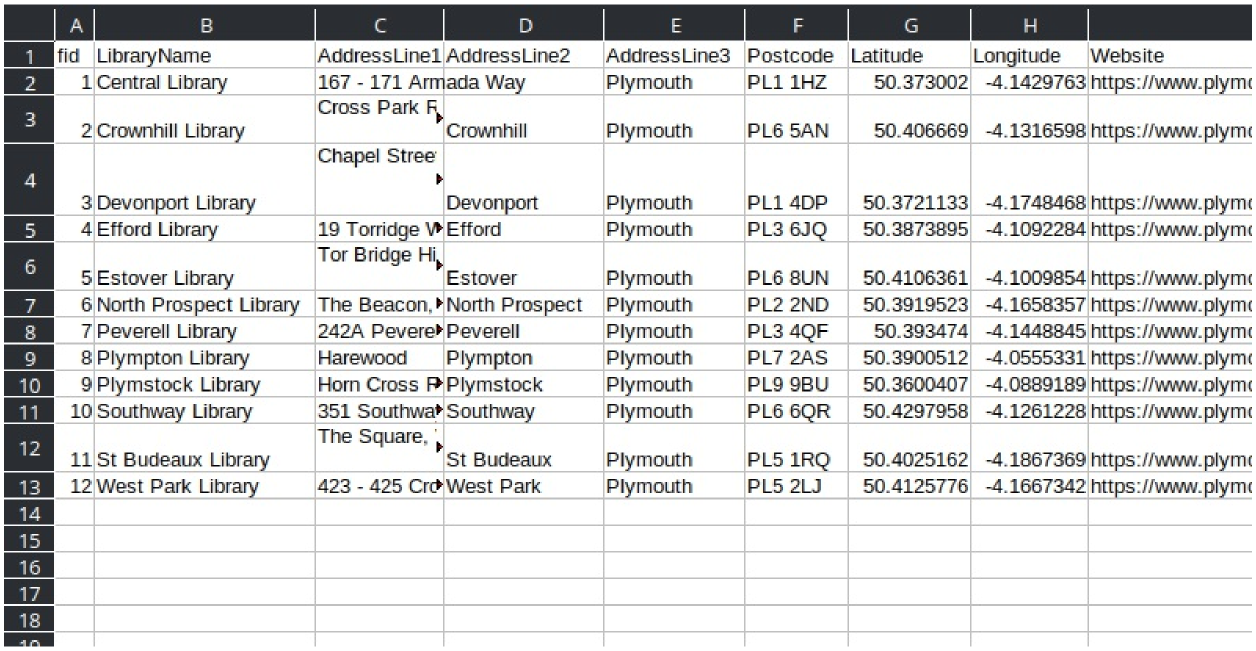
Given those names, we use the built-in `csv` module to write the header, and then write one row per library—to a file we open called `plymouth-libraries.csv`—like this:

```
with open("plymouth-libraries.csv", "w", newline="") as csvfile:
    writer = csv.DictWriter(csvfile, fieldnames=header_names)
    writer.writeheader()
    for library in library_data["features"]:
        writer.writerow(library["properties"])
```

This is the core principle behind using Python as a file format converter to glue together two things:

1. Read data in whatever format it's currently in, which gives us that data as Python dictionaries.
2. Then serialize those Python dictionaries into the file format we actually want.

That's all there is to it. Now we have a CSV file that we can open and look through as we choose.



	A	B	C	D	E	F	G	H	
1	fid	LibraryName	AddressLine1	AddressLine2	AddressLine3	Postcode	Latitude	Longitude	Website
2	1	Central Library	167 - 171 Armada Way		Plymouth	PL1 1HZ	50.373002	-4.1429763	https://www.plymo
3	2	Crownhill Library	Cross Park R	Crownhill	Plymouth	PL6 5AN	50.406669	-4.1316598	https://www.plymo
4	3	Devonport Library	Chapel Stree	Devonport	Plymouth	PL1 4DP	50.3721133	-4.1748468	https://www.plymo
5	4	Efford Library	19 Torridge W	Efford	Plymouth	PL3 6JQ	50.3873895	-4.1092284	https://www.plymo
6	5	Estover Library	Tor Bridge Hi	Estover	Plymouth	PL6 8UN	50.4106361	-4.1009854	https://www.plymo
7	6	North Prospect Library	The Beacon,	North Prospect	Plymouth	PL2 2ND	50.3919523	-4.1658357	https://www.plymo
8	7	Peverell Library	242A Pevere	Peverell	Plymouth	PL3 4QF	50.393474	-4.1448845	https://www.plymo
9	8	Plympton Library	Harewood	Plympton	Plymouth	PL7 2AS	50.3900512	-4.0555331	https://www.plymo
10	9	Plymstock Library	Horn Cross R	Plymstock	Plymouth	PL9 9BU	50.3600407	-4.0889189	https://www.plymo
11	10	Southway Library	351 Southwa	Southway	Plymouth	PL6 6QR	50.4297958	-4.1261228	https://www.plymo
12	11	St Budeaux Library	The Square,	St Budeaux	Plymouth	PL5 1RQ	50.4025162	-4.1867369	https://www.plymo
13	12	West Park Library	423 - 425 Cro	West Park	Plymouth	PL5 2LJ	50.4125776	-4.1667342	https://www.plymo
14									
15									
16									
17									
18									

The script and input files referenced above can be downloaded from GitHub.

Reading Data with Code

A second example will further demonstrate this principle of format conversion. Let’s resume our fictional library tour. The following week, we’re planning to continue our tour by visiting libraries in north Somerset, and the list of *those* libraries is in a format called XML. Opening the XML file and looking at its contents suggests that an individual library is listed in this file as something like this:

```
<Row>
  <PublisherLabel>North Somerset Council</PublisherLabel>
  <PublisherURI>http://opendatacommunities.org/id/unitary-authori
  <LibraryName>Clevedon Library</LibraryName>
  <Address>37 Old Church Road</Address>
  <Locality>Clevedon</Locality>
  <Postcode>BS21 6NN</Postcode>
  <TelephoneNo>(01934) 426020</TelephoneNo>
  <Website>http://www.n-somerset.gov.uk/Leisure/libraries/your_lo
</Row>
```

So we’re going to read that file with Python’s built-in XML module. Again, the goal here is to make a convenient Python dictionary and then serialize it. But this time, we’re also going to alter the data a little along the way: we only care about libraries where the postcode (the British version of a zip code) is BS40 or less. (So we want to keep BS21 6NN as a postcode, but ignore a postcode such as BS49 1AH.) This lets us use our knowledge of Python strings and numbers to discard some of the data.

First, let’s read in the XML file. The ElementTree module is traditionally imported with the name `ET`, for brevity, and can read a file with `.parse()`. We’ll also need the `json` module later to write out a JSON version of this data:

```
import xml.etree.ElementTree as ET
import json
tree = ET.parse('somerset-libraries.xml')
```

ElementTree reads in an XML file and forms a tree structure out of it. At the top of the tree is the `root`, which is the first element in the XML. An ET element has properties relating to its content. An XML element like `<MyElement>content here</MyElement>` would have a `tag` property of `"MyElement"`, the tag name, and a `text` property of `"content here"`, containing the text in the element. The element also presents as a list that can be iterated over with a Python `for` loop, which will yield all the element’s direct child elements.

We obtain a reference to the root element of the tree:

```
root = tree.getroot()
```

In this XML file in particular, this element is actually `<Root>`, but the name is a coincidence; it could be called anything. We can then iterate over all child elements of the root. These child elements are the `<Row>` elements in the XML, each of which defines one library. Our plan here is to turn the above

XML for a `<Row>` into a Python dictionary, with one entry per child of the `<Row>`. That is, the above XML should become this Python dictionary:

```
{
    "PublisherLabel": "North Somerset Council",
    "PublisherURI": "http://opendatacommunities.org/id/unitary-au
    "LibraryName": "Clevedon Library",
    "Address": "37 Old Church Road",
    "Locality": "Clevedon",
    "Postcode": "BS21 6NN",
    "TelephoneNo": "(01934) 426020",
    "Website": "http://www.n-somerset.gov.uk/Leisure/libraries/yo
}
```

To do this, we'll iterate over each of the `<Row>` elements in the root, and then, for each `<Row>`, iterate over each of *its* child elements and add them to a dictionary for that row. This will give us one dictionary per library, which we'll store in a list called `libraries`:

```
libraries = []

for row in root:
    this_library = {}
    for element in row:
        name = element.tag
        value = element.text
        this_library[name] = value
    libraries.append(this_library)
```

At this point, we’ve successfully parsed the input XML data into a list of Python dictionaries, meaning that we can discard the XML and work simply with the Python data structures. The first goal is to remove all the libraries with postcode `BS40` or greater, which can be done by removing items from the list that don’t match:

```
for library in libraries:
    # get the third and fourth characters of the postcode, as a n
    # note that real postcode parsing is more complex than this!
    postcode = library["Postcode"]
    postcode_number = int(postcode[2:4])
    if postcode_number >= 40:
        libraries.remove(library)
```

The final step is to serialize the Python data into the destination format we want, which in this case is JSON. This is done with the `json` module’s `dump` function:

```
with open("somerset-libraries.json", mode="w") as fp:
    json.dump(libraries, fp, indent=2) # the indent makes the JSO
```

The outputted JSON data will look something like this:

```
[
  {
    "PublisherLabel": "North Somerset Council",
    "PublisherURI": "http://opendatacommunities.org/id/unitary-au
    "LibraryName": "Clevedon Library",
    "Address": "37 Old Church Road",
    "Locality": "Clevedon",
    "Postcode": "BS21 6NN",
    "TelephoneNo": "(01934) 426020",
    "Website": "http://www.n-somerset.gov.uk/Leisure/libraries/yo
  },
... more libraries here ...
]
```

The script and input files can be downloaded from GitHub.

General Principles

The goal here is to take three steps:

1. Load the data we want into a set of Python data structures.
2. Manipulate the data however we want—which is convenient, because it's now a set of Python data structures.
3. Serialize the data into whatever output format we want.

Sometimes, the steps might require a little programming, as above. JSON data, as we saw in the first example, has a built-in Python library to load it (or "de-serialize" it) directly into a Python dictionary, ready for us to manipulate. XML data doesn't have such a library, so a little delving into the Python docs is required if we're to understand exactly how to read it. Many more esoteric formats also have Python libraries available to read or to write them, which can be installed from PyPI, Python's suite of third-party libraries.

Parsing Complicated Formats

A useful guide to parsing some more complicated formats using the Python `pandas` module is available in Using Python to Parse Spreadsheet Data.

Understanding All the Formats

One example of such a format is KML, which was originally developed for Google Earth to allow a number of points on the globe to be specified and then loaded onto a map. It might be useful to plot our list of Plymouth libraries on a map so we can see where they all are in relation to one another and plan the best route for our tour. Let’s write a script that reads in that data and outputs it as KML so we can do that!

A little searching reveals that Python doesn’t have a built-in module for reading KML, but there are quite a few available for download. The simplest seems to be simplekml (truth in advertising!), and this is available from PyPI. Installing modules from PyPI involves a little setup, which is documented in Virtual Environments in Python Made Easy and also in the Python Packaging Guide, but once that’s done (and it only needs to be done once and will work for ever more), we should be able to `pip install simplekml` to get the `simplekml` module.

Embracing All the Formats

At this point, the script we need to write follows our standard three steps: load the data from the format it’s in (in this case, we’ll use the CSV file we created earlier), manipulate it if we wish (we don’t need to this time), and then serialize it to a file in the new format (which the simplekml documentation shows us how to do for KML).

First, let’s read in our input data. The Python `csv` module knows how to read the CSV file of libraries that we created previously:

```
import simplekml
import csv

libraries = []
with open('plymouth-libraries.csv') as csvfile:
    reader = csv.DictReader(csvfile)
    for row in reader:
        libraries.append(row)
```

Now we have a Python list of dictionaries. One of those dictionaries looks like this:

```
{
    "fid": "1",
    "LibraryName": "Central Library",
    "AddressLine1": "167 - 171 Armada Way",
    "AddressLine2": "",
    "AddressLine3": "Plymouth",
    "Postcode": "PL1 1HZ",
    "Latitude": "50.373002",
    "Longitude": "-4.1429763",
    "Website": "https://www.plymouth.gov.uk/libraries/findlibrary
}
```

To create a KML file, we can follow the `simplekml` module’s documentation.

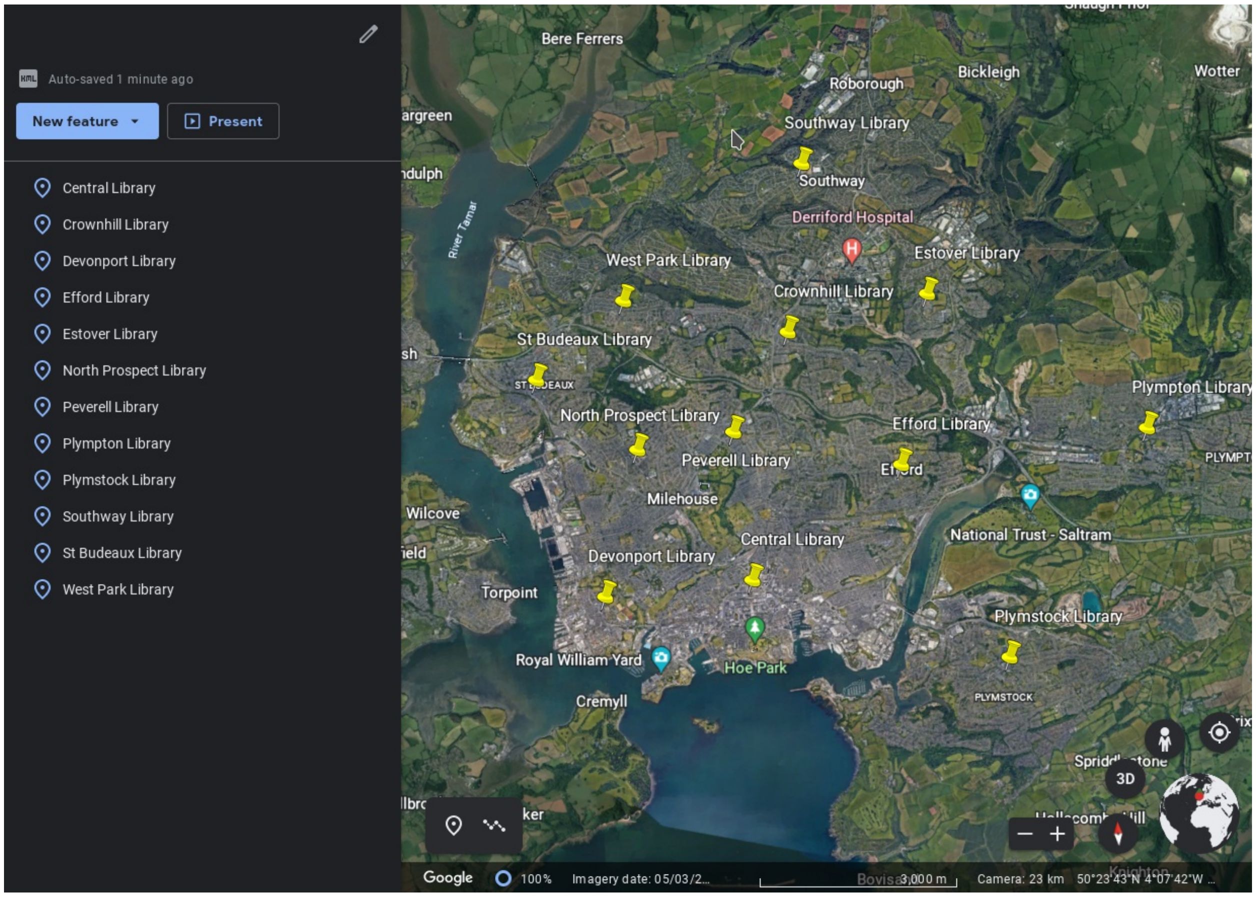
Make a `Kml()` object, then call its `newpoint()` function with name, latitude, and longitude values for each library:

```
kml = simplekml.Kml()
for library in libraries:
    # the coordinates need to be numeric, so convert
    # them to floating-point numbers
    lat = float(library["Latitude"])
    lon = float(library["Longitude"])
    # and add a new "point" to the KML file for this library
    kml.newpoint(name=library["LibraryName"], coords=[(lon, lat)]
```

Finally, save the file as KML:

```
kml.save("plymouth-libraries.kml")
```

Now we can import that KML file to Google Earth and … this is what we see.



The script and input files can be downloaded from GitHub.

Reading HTML

While we’re in Plymouth, we notice that the skyline is dominated by Beckley Point, the tallest building in the southwest of England, and that gets us wondering about tall buildings generally. Wikipedia, as usual, provides a page about the tallest buildings in the world, and we decide we’d like to explore that list a little more in Excel. Of course, we already know that we can copy a table from a web page and paste it into Excel and it does a reasonable job, but that’s not very exciting. We’re all Python all the time now! What if we wanted to look at, say, all the tall buildings without an E in their name? Or to get that list every day by running one command? So let’s break out the text editor again, with the intention of getting the data from that Wikipedia page and then putting it into Excel with a script.

Extracting data from an HTML page is the task of **screen scraping**, and much has been written about it. One useful guide is Web Scraping for Beginners. There are dedicated screen scraping tools to do the work, but Python can do it too, and then we have all the power of code at our disposal.

First, let’s look at how to fetch and read an HTML page with Python to extract data from it.

Requesting data from the Web can be done with the built-in `urllib.request` module, but the `requests` module is easier to use, so it’s worth installing that. We’ll also need Beautiful Soup (BS) for parsing HTML. (Again, this can be done with built-in modules, but BS is more pleasant.) So `pip install requests beautifulsoup4` should get those modules ready for use. After that, the content of a web page can be fetched with `requests.get()` and the response’s `content` property:

```
import requests
from bs4 import BeautifulSoup

# Use the Python requests module to fetch the web page
response = requests.get("https://en.wikipedia.org/wiki/List_of_ta
# and extract its HTML
html = response.content
```

At this point, `html` is a long string variable of HTML. Beautiful Soup knows how to parse that into a useful data structure that we can explore:

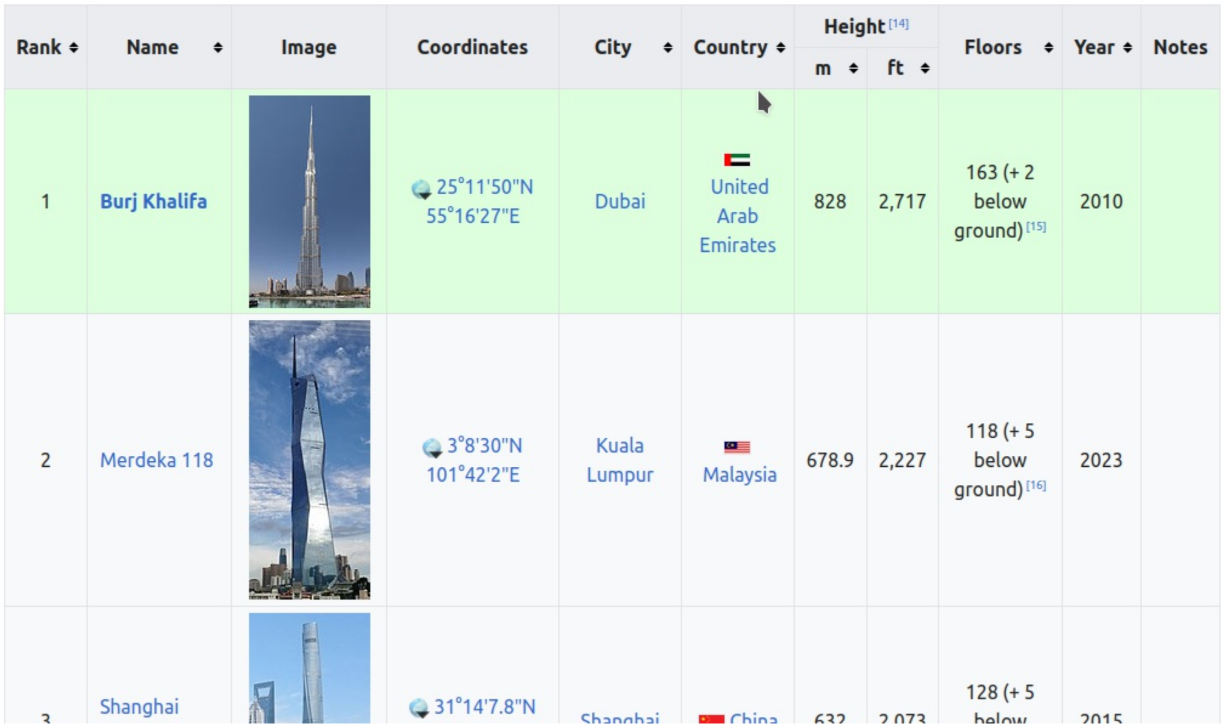
```
soup = BeautifulSoup(html, "html.parser")
```

Extracting data from HTML is one of the tasks that requires some programming—as well as some knowledge of the structure of the HTML in question—to make best use of the data. Beautiful Soup is the most-used Python module for this, and it has a deep well of functionality to help with advanced screen scraping.

For most examples, though, it will suffice to load the HTML into a “soup” of parsed data, and then use the `.select()` function to extract HTML elements by CSS selector, iterate through them, and read the `.text` within them. This is somewhat similar to using JavaScript methods to identify HTML elements in the DOM in a web page, which may be familiar to web programmers. The BS4 documentation explains some of Beautiful Soup’s more advanced features in detail, but much of the work can be done with `select()`, which takes a CSS selector as a parameter and returns a list of elements matching that selector, similarly to JavaScript’s `querySelectorAll()`.

The HTML table that we want—of the tallest buildings in the world by height to pinnacle—is the fifth `<table>` element on the page. If the Wikipedia HTML had an ID attribute on that table, we could use that to find it directly, but it doesn’t. So we’ll use the knowledge that it’s the fifth table to find it with the `.select()` function:

```
tallest_table = soup.select("table")[4]
```



Rank	Name	Image	Coordinates	City	Country	Height[14]		Floors	Year	Notes
						m	ft			
1	Burj Khalifa		25°11'50"N 55°16'27"E	Dubai	United Arab Emirates	828	2,717	163 (+ 2 below ground)[15]	2010	
2	Merdeka 118		3°8'30"N 101°42'2"E	Kuala Lumpur	Malaysia	678.9	2,227	118 (+ 5 below ground)[16]	2023	
3	Shanghai		31°14'7.8"N	Shanghai	China	632	2,073	128 (+ 5 below	2015	

As can be seen from the table, we can get the title of each column by reading the first row. The table actually has two header rows, because there are separate columns for both height in meters and height in feet, and so we handle this a little differently by adding two columns to the list of column titles:

```
# Get the title for each column, by reading the text of
# each <th> in the first row of this table.
# Note that the "Height" column is two columns, m and ft,
# so we handle this differently.
column_titles = []
first_row = tallest_table.select("tr")[0]
for th in first_row.select("th"):
    column_title = th.text.strip() # .strip() removes carriage re
    if column_title == "Height":
        # add two columns
        column_titles.append(column_title + " (m)")
        column_titles.append(column_title + " (ft)")
    else:
        # add this column title to the list
        column_titles.append(column_title)
```

Once the column titles have been extracted—so that they can be used to make a dictionary for each building—it's time to actually extract the building data. Each row in the table contains the values for a building. This means that, for each row, we need to combine the column titles and the column data to make a dictionary. Fortunately, Python provides the `zip()` function, which makes this easy.

`zip()` takes two lists—`[a, b, c, d]` and `[w, x, y, z]`—and returns one list: `[(a,w), (b,x), (c,y), (d,z)]`. We can then iterate through this combined list.

This means that, for each table row (or `<tr>`), we can retrieve the cells (`<td>`) in that row with `tr.select("td")`, and then zip the column titles together with them. Each cell returned from the `<td>` is a soup object, so we must then use the `.text` property of each `<td>` to retrieve the text in it:

```
tallest_buildings = []

for tr in tallest_table.select("tr")[1:]:
    building = {}
    tds = tr.select("td")
    # combine the header list and the data list from this row wit
    named_data = zip(column_titles, tds)
    for title, td in named_data:
        building[title] = td.text.strip()
    # and store this building in the list
    tallest_buildings.append(building)
```

More Data Fiddling

Now that we have the data in a convenient Python data structure and no longer have to worry about HTML (or however else we obtained it), the next step is to manipulate the data. Here, it will be useful to convert the numeric heights of the buildings to be actual numbers. Since HTML data is all textual, this means that all the “height” values in our data are strings of digits rather than actual numbers. We can see this with a quick test, using Python’s built-in `pprint` module to “pretty-print” the first building in the list:

```
import pprint
pprint.pprint(tallest_buildings[0])
```

```
{'Building': 'Burj Khalifa†',
 'Built': '2010',
 'City': 'Dubai',
 'Country': 'United Arab Emirates',
 'Floors': '163',
 'Height (ft)': '2,722\xa0ft',
 'Height (m)': '829.8 m',
 'Rank': '1'}
```

Let’s convert the “Height (ft)”, “Height (m)” and “Floors” values from strings to numbers. These data may contain characters that aren’t numbers, so a simple call to `float()` may not work and we’ll need to be a little more clever.

The most basic way to convert string data like this to a floating point number or integer is to walk through each character in the string and only keep it if it’s a digit or a decimal point, and then to call `float()` on the kept characters only. This removes any commas, carriage returns, spaces, or other extraneous data in a string like `"2,345.6 m"`, and would look like this, for converting the “Height (m)” data for each building:

```
height_m_digits = ""
for character in building["Height (m)"]:
    if character.isdigit() or character == ".":
        height_m_digits += character
building["Height (m)"] = float(height_m_digits)
```

A second but slightly more advanced way is to use a list comprehension to extract these digits, which will convert `"2,345.6 m"` into `["2", "3", "4", "5", ".", "6"]`—which can then be combined back into a string with `.join()` and then to a number with `float()`:

```
height_ft_digits = [
    character for character in building["Height (ft)"]
    if character.isdigit() or character == "."]
height_ft_digits = "".join(height_ft_digits)
building["Height (ft)"] = float(height_ft_digits)
```

And a third way that’s shorter still is to use regular expressions, which are more powerful but also a more complex subject:

```
floors_digits = re.findall(r"[\d.]", building["Floors"])
building["Floors"] = int("".join(floors_digits)) # note floor cou
```

Learning Regular Expressions

If you’d like to learn more about using regular expressions, check out the article Understanding Python Regex Functions, with Examples.

Once the data is converted to numbers for each building, if we pretty-print the first building, the data is more properly numeric:

```
import pprint
pprint.pprint(tallest_buildings[0])
```

```
{'Building': 'Burj Khalifa†',
 'Built': '2010',
 'City': 'Dubai',
 'Country': 'United Arab Emirates',
 'Floors': 163,
 'Height (ft)': 2722.0,
 'Height (m)': 829.8,
 'Rank': 1}
```

Writing for Excel for Real

The final step is to convert this data to our preferred destination format.

It’s sometimes useful to be able to create nicer Excel output than CSV can provide. In particular, a CSV file looks very plain when loaded into a spreadsheet, and has no formatting at all. If we’re producing a sheet for our own data exploration, then it may not matter how pretty it is. But a little effort put into making data look more readable can help a lot, especially if it’s destined for someone else. For this, we’ll use Python’s XlsxWriter, which can be installed with `pip install Xlsxwriter`. We can then pay some attention to the examples given in XlsxWriter’s tutorial.

First, create a new Excel spreadsheet file, and add a worksheet to it:

```
import xlsxwriter


workbook = xlsxwriter.Workbook('tallest_buildings.xlsx')
worksheet = workbook.add_worksheet()
```

To add formatting to Excel cells using XlsxWriter, it’s easiest to define a particular format style, which can then be applied later. Let’s do that for some of the cell data up front:

```
# Add a bold red format to use to highlight cells
bold = workbook.add_format({
    'bold': True, 'font_color': 'red'
})

# Add a number format for cells with heights.
heights = workbook.add_format({'num_format': '#,##0.00'})
```

The first row of the spreadsheet will be the column titles. We can fetch these by using the dictionary keys from the first building in the list:

```
column_titles = tallest_buildings[0].keys()
row_number = 0 # we write the column titles in row 0
```

We now need to walk through this list of titles and write each into its corresponding column in row `0`. To do this, we use `worksheet.write` with a row number (always `0`), column number (starting at `0` and increasing by one for each column title), cell content (the column title string itself), and a cell format (`bold`, as defined above). Python provides the useful `enumerate()` function to iterate the list while getting the index position for each item. `enumerate([a, b, c, ...])` will return a list (`[(0, a), (1, b), (2, c), ...]`) which can be iterated over to write out our column titles at the top of each column:

```
for column_number, title in enumerate(column_titles):
    worksheet.write(row_number, column_number, title, bold)
```

Then, to write out all the data, we do exactly the same thing: iterate through the column titles list once for each building, and write out the value of that field in the building's dictionary into the appropriate row and column number:

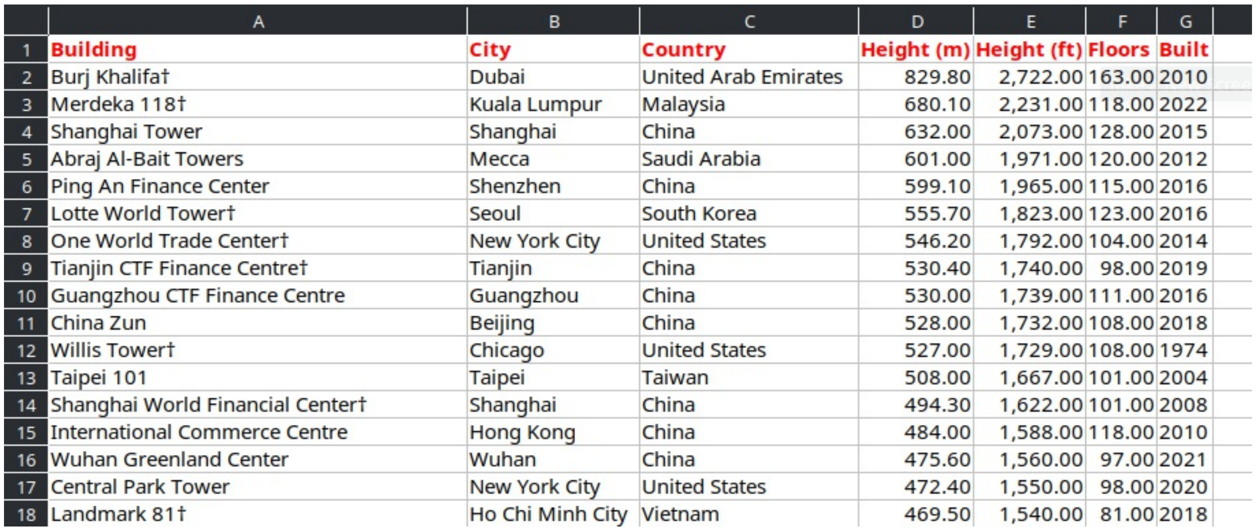
```
# Start from the first cell below the headers.
row_number = 1


# Iterate over the buildings and write them out row by row,
for building in tallest_buildings:
    for column_number, title in enumerate(column_titles):
        worksheet.write(row_number, column_number, building[title
    row_number += 1
```

Finally, a call to `worksheet.autofit()` adjusts all the cell widths in the spreadsheet so they fit the data contained with them, similar to double-clicking on a column border in Excel itself. Then, closing the worksheet ensures it's saved:

```
worksheet.autofit()
workbook.close()
```

The script and input files can be downloaded from GitHub.

And then we have some more nicely formatted Excel data to look at, as pictured below.



	A	B	C	D	E	F	G
1	**Building**	**City**	**Country**	**Height (m)**	**Height (ft)**	**Floors**	**Built**
2	Burj Khalifa†	Dubai	United Arab Emirates	829.80	2,722.00	163.00	2010
3	Merdeka 118†	Kuala Lumpur	Malaysia	680.10	2,231.00	118.00	2022
4	Shanghai Tower	Shanghai	China	632.00	2,073.00	128.00	2015
5	Abraj Al-Bait Towers	Mecca	Saudi Arabia	601.00	1,971.00	120.00	2012
6	Ping An Finance Center	Shenzhen	China	599.10	1,965.00	115.00	2016
7	Lotte World Tower†	Seoul	South Korea	555.70	1,823.00	123.00	2016
8	One World Trade Center†	New York City	United States	546.20	1,792.00	104.00	2014
9	Tianjin CTF Finance Centre†	Tianjin	China	530.40	1,740.00	98.00	2019
10	Guangzhou CTF Finance Centre	Guangzhou	China	530.00	1,739.00	111.00	2016
11	China Zun	Beijing	China	528.00	1,732.00	108.00	2018
12	Willis Tower†	Chicago	United States	527.00	1,729.00	108.00	1974
13	Taipei 101	Taipei	Taiwan	508.00	1,667.00	101.00	2004
14	Shanghai World Financial Center†	Shanghai	China	494.30	1,622.00	101.00	2008
15	International Commerce Centre	Hong Kong	China	484.00	1,588.00	118.00	2010
16	Wuhan Greenland Center	Wuhan	China	475.60	1,560.00	97.00	2021
17	Central Park Tower	New York City	United States	472.40	1,550.00	98.00	2020
18	Landmark 81†	Ho Chi Minh City	Vietnam	469.50	1,540.00	81.00	2018

Summary

And that’s all we need! Whether we’re using code libraries to find real libraries, or trying to extract sense from a mess of reports or published data, Python’s there for us.

For each file or set of data we want to work with, we can take these three steps:

1. Load the data we want into a set of Python data structures, using Python’s built-in modules or others that we find from PyPI which can read that format.
2. Manipulate the data however we want—which is convenient, because it’s now a set of Python data structures.
3. Serialize the data into whatever output format we want, again using built-in or third-party modules.

CHAPTER 2: PYTHON FOR STITCHING TOGETHER OTHER THINGS

Python is good at translating data from one format to another and altering it along the way, as we saw in the first tutorial. But a good proportion of what we may want to do involves controlling and working with other programs and other data sources that aren't files.

Sometimes we need to do more than process the data in an Excel file we've been sent. For example, we may want to fetch some pages from the Web, or work with an online API, or control our computer itself (such as renaming a batch of files, or changing how our operating system works).

Python is good at this stuff as well, so let's dive into how we can use Python as a Swiss Army knife for anything we might want to do with our computer or the Internet.

All the example code from this tutorial is available on GitHub.

Fetching Data from an HTTP API

In the first tutorial, we extracted data from Wikipedia's HTML, but this isn't the best way to retrieve data in our scripts. Most of the time, we'll be accessing an HTTP API. That is, we'll be making an HTTP call to a web page designed to be read by machines rather than by people. API data is normally in a machine-readable format—usually either JSON or XML. (If we come across data in another format, we can use the techniques described in the previous tutorial to convert it to JSON, of course!) Let's look at how to use an HTTP API from Python.

The general principles of using an HTTP API are simple:

1. Make an HTTP call to the URLs for the API, possibly including some authentication information (such as an API key) to show that we're authorized.
2. Get back the data.
3. Do something useful with it.

Python provides enough functionality in its standard library to do all this without any additional modules, but it will make our life a lot easier if we pick up a couple of third-party modules to smooth over the process. The first is the requests module. This is an HTTP library for Python that makes fetching HTTP data more pleasant than Python's built-in `urllib.request`, and it can be installed with `python -m pip install requests`.

To show how easy it is to use, we'll use Pixabay's API (documented here). Pixabay is a stock photo site where the images are all available for reuse, which makes it a very handy destination. What we'll focus on here is fruit. We'll use the fruit pictures we gather later on, when manipulating files, but for now we just want to find pictures of fruit, because it's tasty and good for us.

To start, we'll take a quick look at what pictures are available from Pixabay. We'll grab a hundred images, quickly look through them, and choose the ones we want. For this, we'll need a Pixabay API key, so we need to create an account and then grab the key shown in the API documentation under "Search Images".

The requests Module

The basic version of making an HTTP request to an API with the `requests` module involves constructing an HTTP URL, requesting it, and then reading the response. Here, that response is in JSON format. The `requests` module makes each of these steps easy. The API parameters are a Python dictionary, a `get()` function makes the call, and if the API returns JSON, `requests` makes that available as `.json` on the response. So a simple call will look like this:

```
import requests

PIXABAY_API_KEY = "11111111-7777777777777777777777777"

base_url = "https://pixabay.com/api/"
base_params = {
    "key": PIXABAY_API_KEY,
    "q": "fruit",
    "image_type": "photo",
    "category": "food",
    "safesearch": "true"
}

response = requests.get(base_url, params=base_params)
results = response.json()
```

This will return a Python object, as the API documentation suggests, and we can look at its parts:

```
>>> print(len(results["hits"]))
20
>>> print(results["hits"][0])
{'id': 2277, 'pageURL': 'https://pixabay.com/photos/berries-fruit
```

The API returns 20 hits per page, and we'd like a hundred results. To do this, we add a `page` parameter to our list of `params`. However, we don't want to alter our `base_params` every time, so the way to approach this is to create a loop and then make a *copy* of the `base_params` for each request. The built-in `copy` module does exactly this, so we can call the API five times in a loop:

```
for page in range(1, 6):
    this_params = copy.copy(base_params)
    this_params["page"] = page
    response = requests.get(base_url, params=params)
```

This will make five separate requests to the API, one with `page=1`, the next with `page=2`, and so on, getting different sets of image results with each call. This is a convenient way to walk through a large set of API results. Most APIs implement **pagination**, where a single call to the API only returns a limited set of results. We then ask for more pages of results—much like looking through query results from a search engine.

Since we want a hundred results, we could simply decide that this is five calls of 20 results each, but it would be more robust to keep requesting pages until we have the hundred results we need and then stop. This protects the calls in case Pixabay changes the default number of results to 15 or similar. It also lets us handle the situation where there *aren't* a hundred images for our search terms. So we have a `while` loop and increment the page number every time, and then, if we’ve reached 100 images, or if there are no images to retrieve, we break out of the loop:

```
images = []
page = 1
while len(images) < 100:
    this_params = copy.copy(base_params)
    this_params["page"] = page
    response = requests.get(base_url, params=this_params)
    if not response.json()["hits"]: break
    for result in response.json()["hits"]:
        images.append({
            "pageURL": result["pageURL"],
            "thumbnail": result["previewURL"],
            "tags": result["tags"],
        })
    page += 1
```

This way, when we finish, we’ll have 100 images, or we’ll have all the images if there are fewer than 100, stored in the `images` array. We can then go on to do something useful with them. But before we do that, let’s talk about caching.

Caching HTTP Requests

It’s a good idea to avoid making the same request to an HTTP API more than once. Many APIs have usage limits in order to avoid them being overtaxed by requesters, and a request takes time and effort on their part and on ours. We should try to not make wasteful requests that we've done before. Fortunately, there's a useful way to do this when using Python's `requests` module: install requests-cache with `python -m pip install requests-cache`. This will seamlessly record any HTTP calls we make and save the results. Then, later, if we make the same call again, we'll get back the locally saved result without going to the API for it at all. This saves both time and bandwidth. To use `requests_cache`, import it and create a `CachedSession`, and then instead of `requests.get` use `session.get` to fetch URLs, and we'll get the benefit of caching with no extra effort:

```
import requests_cache
session = requests_cache.CachedSession('fruit_cache')
...
response = session.get(base_url, params=this_params)
```

Making Some Output

To see the results of our query, we need to display the images somewhere. A convenient way to do this is to create a simple HTML page that shows each of the images. Pixabay provides a small thumbnail of each image, which it calls `previewURL` in the API response, so we could put together an HTML page that shows all of these thumbnails and links them to the main Pixabay page—from which we could choose to download the images we want and credit the photographer. So each image in the page might look like this:

```
<li>
    <a href="https://pixabay.com/photos/berries-fruits-food-black
```

```
        <img src="https://cdn.pixabay.com/photo/2010/12/13/10/05/
    </a>
</li>
```

We can construct that from our `images` list using a list comprehension, and then join together all the results into one big string with `"\n".join()`:

```
html_image_list = [
    f"""<li>
            <a href="{image["pageURL"]}">
                <img src="{image['thumbnail']}" alt="{image["tags
            </a>
        </li>
    """
    for image in images
]
html_image_list = "\n".join(html_image_list)
```

F-strings

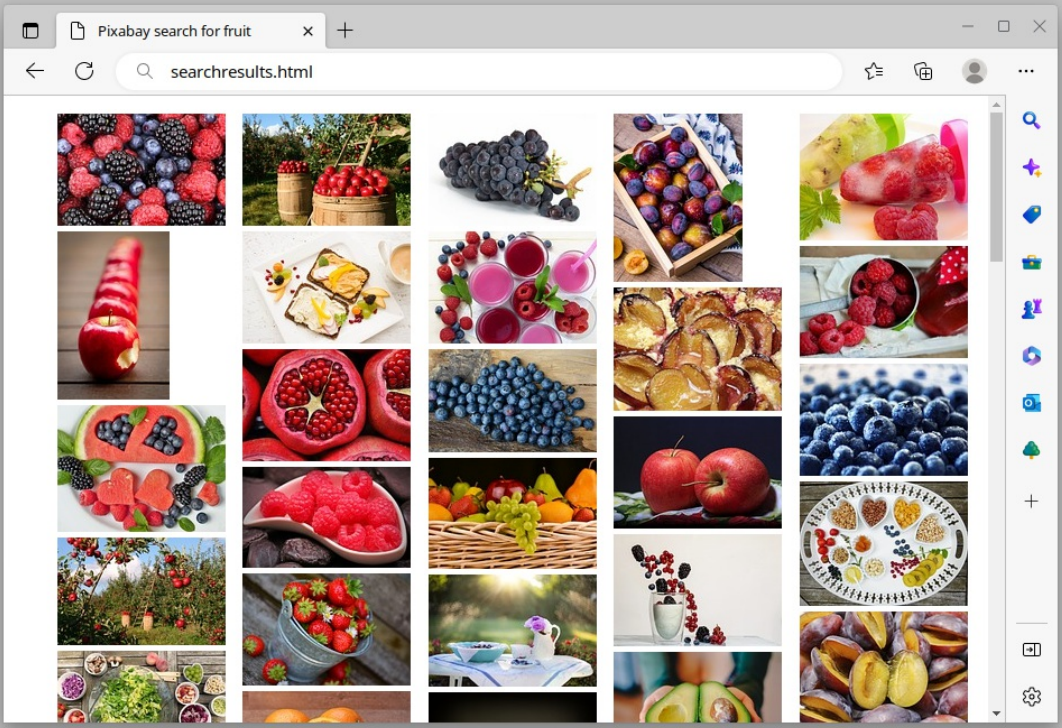
In the code above, we used an f-string, or “formatted string literal”. These were introduced in Python 3.6 as a convenient alternative to Python’s old string `format()` method or `%` interpolation syntaxes, used for embedding the value of a variable in a string. An f-string—prefixed with `f` before the quote marks—can have a variable’s value inserted into it (and optionally formatted in many ways) by using `{curly braces}`. So, a simple example might be:

```
>>> name = "Stuart"
>>> print(f"Hello, {name}!")
Hello, Stuart!
```

At that point, if we write out a very plain HTML page containing that list, it’s easy to open that in a web browser for a quick overview of all the search results we got from the API, and click any one of them to jump to the full Pixabay page for downloads:

```
html = f"""<!doctype html>
<html><head><meta charset="utf-8">
<title>Pixabay search for {base_params['q']}</title>
<style>
ul {{
    list-style: none;
    line-height: 0;
    column-count: 5;
    column-gap: 5px;
}}
li {{
    margin-bottom: 5px;
}}
</style>
</head>
<body>
<ul>
{html_image_list}
</ul>
</body></html>
```

```
"""
output_file = f"searchresults-{base_params['q']}.html"
with open(output_file, mode="w", encoding="utf-8") as fp:
    fp.write(html)
print(f"Search results summary written as {output_file}")
```



Controlling Windows

In this section, we’ll look at ways to control the Windows OS with Python.

The Windows Registry

Windows is entirely controllable from code, using the Win32 API, and

Microsoft provides extensive documentation at Microsoft Docs for everything that Windows can programmatically do. All of this is accessible from Python as well, although it can seem a little impenetrable if we're not already accustomed to the Win32 API's particular way of working. Fortunately, there are various wrappers for these low-level APIs to make code easier to write for Python programmers.

A simple example is to interact with the Windows Registry. Python actually includes the winreg module for doing this out of the box, so no extra installation is required. For an example, let's check where the `Program Files` folder actually lives:

```
>>> import winreg
>>> hive = winreg.ConnectRegistry(None, winreg.HKEY_LOCAL_MACHINE
>>> key = winreg.OpenKey(hive, r"SOFTWARE\Microsoft\Windows\Curre
>>> value, type = winreg.QueryValueEx(key, "ProgramFilesDir")
>>> value
'C:\\Program Files'
```

Raw Strings

In the code above, we're using "raw strings" to specify the key name:

```
r"SOFTWARE\Microsoft\Windows\CurrentVersion"
```

Strings passed to the Win32 API often include the backslash character (\), because Windows uses it in file paths and registry paths to divide one directory from the next.

However, Python uses a backslash as an escape character to allow adding special, untypeable characters to a string. For example, the Python string `"first line\nsecond line"` is a string with a newline character in it, so that the text is spread over two lines. This would conflict with the Windows path character: a file path such as `"C:\newdir\myfile.txt"` would have the `\n` interpreted as a newline.

Raw strings avert this: prefixing a Python string with `r` removes the special meaning of a backslash, so that `r"C:\newdir\myfile.txt"` is interpreted as intended. We can see that backslashes are treated specially by the value we get back for the folder location: it's printed as `'C:\\Program Files'`—with the backslash doubled to remove its special meaning—but this is how Python prints it rather than the actual value. Python could have printed that as `r'C:\Program Files'` instead.

The Windows API

Reading the registry (and even more so, writing to it) is the source of a thousand hacks on web pages (many of which are old, shouldn't be linked to, and use the ancient `REGEDT32.EXE`), but it's better to actually use the API for this. (Raymond Chen has written many long sad stories about why we should use the API and not the registry.) How would we use the Win32 API from Python to work this out?

The Win32 Python API is available in the PyWin32 module, which can be obtained with `python -m pip install pywin32`. The documentation for the module is rather sparse, but the core idea is that most of the Windows Shell API (that's concerned with how the Windows OS is set up) is available in the `win32com.shell` package. To find out the location of the `Program Files` folder, MSDN shows that we need the SHGetKnownFolderPath function, to which is passed a KNOWNFOLDERID constant and a flag set to `0`. Shell constants are available to Python in `win32com.shell.shellcon` (for "shell constants"), which means that finding the `Program Files` folder requires just one (admittedly complex) line:

```
>>> from win32com.shell import shell, shellcon
>>> shell.SHGetKnownFolderPath(shellcon.FOLDERID_ProgramFiles, 0)
"C:\\Program Files"
```

Digging around in the depths of the Win32 API gives us access to anything we may want to access in Windows (including windows!), but as we've seen, it can be quite complicated to find out how to do what we need to, and then to translate that need into Python. Fortunately, there are wrapper libraries for many of the functions commonly used. One good example is PyGetWindow, which allows us to enumerate and control on-screen windows. (It claims to be cross-platform, but it actually only works on Windows. But that's all we need here.)

We can install PyGetWindow with `python -m pip install pygetwindow`, and then list all the windows on screen and manipulate them:

```
>>> import pygetwindow as gw
>>> allMSEdgeWindows = gw.getWindowsWithTitle("edge")
>>> allMSEdgeWindows
```

```
[Win32Window(hWnd=197414), Win32Window(hWnd=524986)]
>>> allMSEdgeWindows[0].title
'pywin32 · PyPI - Microsoft Edge'
>>> allMSEdgeWindows[1].title
'Welcome to Python.org - Microsoft Edge'
```

Those windows can be controlled. A window object can be minimized and restored, or resized and moved around the screen, and focused and brought to the front:

```
>>> pythonEdgeWindow = allMSEdgeWindows[1]
>>> pythonEdgeWindow.minimize()
>>> pythonEdgeWindow.restore()
>>> pythonEdgeWindow.size
Size(width=1050, height=708)
>>> pythonEdgeWindow.topleft
Point(x=218, y=5)
>>> pythonEdgeWindow.resizeTo(800, 600)
```

It’s always worth looking on PyPI for wrapper modules that provide a more convenient API for whatever we’re trying to do with windows or with Windows. But if need be, we have access to the whole Win32 API from Python, and that will let us do anything we can think of.

Controlling macOS

Working on a Mac, we can control almost everything about the system using pyobjc, the Python-to-Objective-C bridge. Apple makes most of its OS controllable via the AppKit module, and `pyobjc` gives Python access to all of this. This will be most useful if we already know the AppKit way to do the thing we want, but with a little exploration it's possible to make our way through the operating system APIs.

Let's try an example. First, we'll need `pyobjc`, which can be installed with `pip install pyobjc`. This will install a whole list of operating system API bridges, allowing access to all sorts of aspects of macOS. For now, we'll consider AppKit, which is the tool used to build and control running apps on a Mac desktop.

We can list all the applications currently running using AppKit:

```
Python 3.9.6 (default, Oct 18 2022, 12:41:40)
[Clang 14.0.0 (clang-1400.0.29.202)] on darwin
Type "help", "copyright", "credits" or "license" for more informa
>>> from AppKit import NSWorkspace
>>> NSWorkspace.sharedWorkspace().runningApplications()
(
    "<NSRunningApplication: 0x60000145c000 (com.apple.loginwindow
    "<NSRunningApplication: 0x60000145c080 (com.apple.backgroundt
    "<NSRunningApplication: 0x60000145c100 (com.apple.WindowManag
    "<NSRunningApplication: 0x60000145c180 (com.apple.CoreLocatio
    "<NSRunningApplication: 0x60000145c980 (com.apple.Terminal - 
    "<NSRunningApplication: 0x60000145ca00 (com.apple.Safari - 13
    "<NSRunningApplication: 0x60000145cb80 (com.apple.Spotlight -
    "<NSRunningApplication: 0x60000145cc00 (com.apple.finder - 13
)
>>>
```

This will give a long list of `NSRunningApplication` objects. Each one corresponds to a specific application currently running on the desktop. Many are “invisible” applications (things that are running but aren’t necessarily showing a window), but others are things that we might think of as actual applications that we can see—such as Safari, Terminal, and so on. `NSRunningApplication` is documented at developer.apple.com, where its properties can be seen. For example, each application has a `localizedName` and a `bundleIdentifier`:

```
>>> for nsapp in NSWorkspace.sharedWorkspace().runningApplication
...   print(f"{nsapp.localizedName()} -> {nsapp.bundleIdentifier(
...
loginwindow -> com.apple.loginwindow
BackgroundTaskManagementAgent -> com.apple.backgroundtaskmanageme
WindowManager -> com.apple.WindowManager
CoreLocationAgent -> com.apple.CoreLocationAgent
Terminal -> com.apple.Terminal
Safari -> com.apple.Safari
Spotlight -> com.apple.Spotlight
Finder -> com.apple.finder
```

We can also see that a `NSRunningApplication` object has an activate function, which we can call to activate that app as though we had clicked its icon in the Dock. So, to find Safari and then activate it, we would use that activate function. The call to `activate` requires a value for `options`, as the documentation describes, and that also needs to be imported from AppKit:

```
>>> from AppKit import NSWorkspace, NSApplicationActivateIgnoring
>>> safari_list = [x for x in NSWorkspace.sharedWorkspace().runni
    if x.bundleIdentifier() == 'com.apple.Safari']
>>> safari = safari_list[0]
>>> safari.activateWithOptions_(NSApplicationActivateIgnoringOthe
```

Now Safari is activated.

Finding Python Versions of macOS APIs

Finding the name of something in Python that corresponds to the Objective-C name can be a little tricky. As shown in the code above, the Objective-C `activate` function is called `activateWithOptions_` in Python. There’s a set of rules for this name translation, which the pyobjc documentation explains, but it can sometimes be quicker to use Python’s own `dir()` function to show all the properties of an object and then pick out the one that looks most plausible:

```
>>> print(len(dir(safari)))
452
```

Ouch! Our `safari` instance of an `NSRunningApplication` has 452 properties! Well, the one we want is probably called something *like* “activate”, so:

```
>>> print([x for x in dir(safari) if "activate" in x.lower()])
['activateWithOptions_', 'activateWithOptions_']
```

Aha! So `activateWithOptions_` is the name of the function we need to call. Similarly, the name of the option we want to pass to that function is in

AppKit itself:

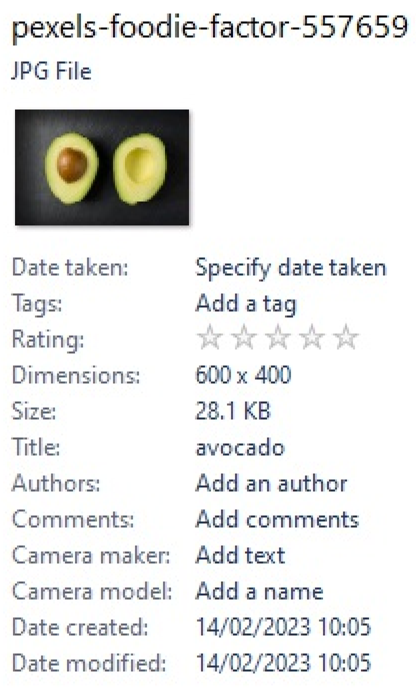
```
>>> [x for x in dir(AppKit) if "ignoringotherapps" in x.lower()]
['NSApplicationActivateIgnoringOtherApps']
```

This process can feel a little exploratory at times, but it’s possible to do anything that Objective-C can do from Python as well.

File Processing

It’s common to want to manipulate a collection of files—such as find their names and rename them, move them around into different directories, or copy and delete them. Let’s see how Python’s built-in `os`, `glob`, and `shutil` modules give us the tools we need to do that.

Imagine we have a set of photographs we’d like to classify and name according to what’s in them. There are online APIs that can actually give you a description of the contents of an image, but they all require signups and mostly require payment, so for now we’ll take a simpler example. JPEG images can have embedded metadata describing when a picture was taken, which camera it was taken on, and a whole host of other data, including an image title. So let’s imagine that we have a set of images with title data, and that we’d like to rename the images *after* that title data—so that a picture named `IMG_1234.JPG` but titled “Grandma” becomes `Grandma.jpg`.


pexels-foodie-factor-557659
JPG File

Date taken:	Specify date taken
Tags:	Add a tag
Rating:	☆☆☆☆☆
Dimensions:	600 x 400
Size:	28.1 KB
Title:	avocado
Authors:	Add an author
Comments:	Add comments
Camera maker:	Add text
Camera model:	Add a name
Date created:	14/02/2023 10:05
Date modified:	14/02/2023 10:05

For this, we'll need to read the metadata from Python, and the best way to do any image manipulation is with Pillow, the Python Imaging Library. The metadata we want here is called IPTC, defined in extreme detail by the International Press Telecommunications Council in the IPTC Photo Metadata Standard. In particular, we want the image's title tag, with code `2:120`. We can define a convenient variable for that:

```
IPTC_Title = (2, 120)
```

Next, let's loop over each of our images. Python's `glob` module is useful for this. A **glob** (a term from Unix prehistory, short for "global") is a pattern using a wildcard character and file specification to gather a list of all files matching that pattern. (For example, `*.txt` will match a list of files such as `myfile.txt`, `1.txt`, `everything.txt`, and so on.) In Python, `glob.glob("*.txt")` does the same thing. It returns a list of matching filenames, which we can then iterate over. We can also open each one with the Python Imaging Library:

```
import glob, os
from PIL import Image

for image_filename in glob.glob("*.jpg"):
    im = Image.open(image_filename)
```

Here, we have a reference to each image. Pillow can do all sorts of image manipulations, such as resizing, changing the content, saving in different formats, and so on. But for now, all we want to do is read the metadata. For that, we use Pillow's `IptcImagePlugin`, and we can have it print out the relevant image title:

```
from PIL import Image, IptcImagePlugin
for image_filename in glob.glob("*.jpg"):
    im = Image.open(image_filename)
    iptc = IptcImagePlugin.getiptcinfo(im)
    im.close()
    title = iptc[IPTC_Title].decode("utf-8")
```

Now that we know the title for a given image, we can use it to construct a new filename, and then use `os.rename` to rename the file to this new name. The `os.rename` function does more than simply renaming; it can also move a file to a different directory or different disk. But in this case, we're only renaming the file in place:

```
destination_filename = f"{title}.jpg"
print(f"Renaming {image_filename} -> {destination_filename}")
if os.path.exists(destination_filename):
    print(f"{destination_filename} already exists; skipping")
else:
    os.rename(image_filename, destination_filename)
```

We're being a little careful here. If the `destination_filename` already exists, then `os.rename` can—depending on our OS—either overwrite it without warning or throw an exception. So we check beforehand whether that file exists, and if it does, we assume that we've already handled this image and therefore skip it.

Now, a list of the directory shows the files all renamed according to their titles. We could look in the file manager for this, but we're in Python, so let's use Python for it:

```
>>> import os
>>> os.listdir()
['avocado.jpg', 'kiwifruit.jpg', 'lemon.jpg', 'apple.jpg',
'strawberry.jpg', 'rename-images.py', 'orange.jpg']
```

Sending Email via Gmail

When writing scripts for our own use, it's useful to be able to send emails from them. For example, if we build a script that's run as a scheduled task on a regular basis, having it email a summary report to us after it's run can be a good way to check that the script did what it was supposed to, and also to see what it found. Something that regularly downloads data from an HTTP API and then processes it can email us a description of what it found, so that we can go and read its results.

Setting Up Scheduled Tasks

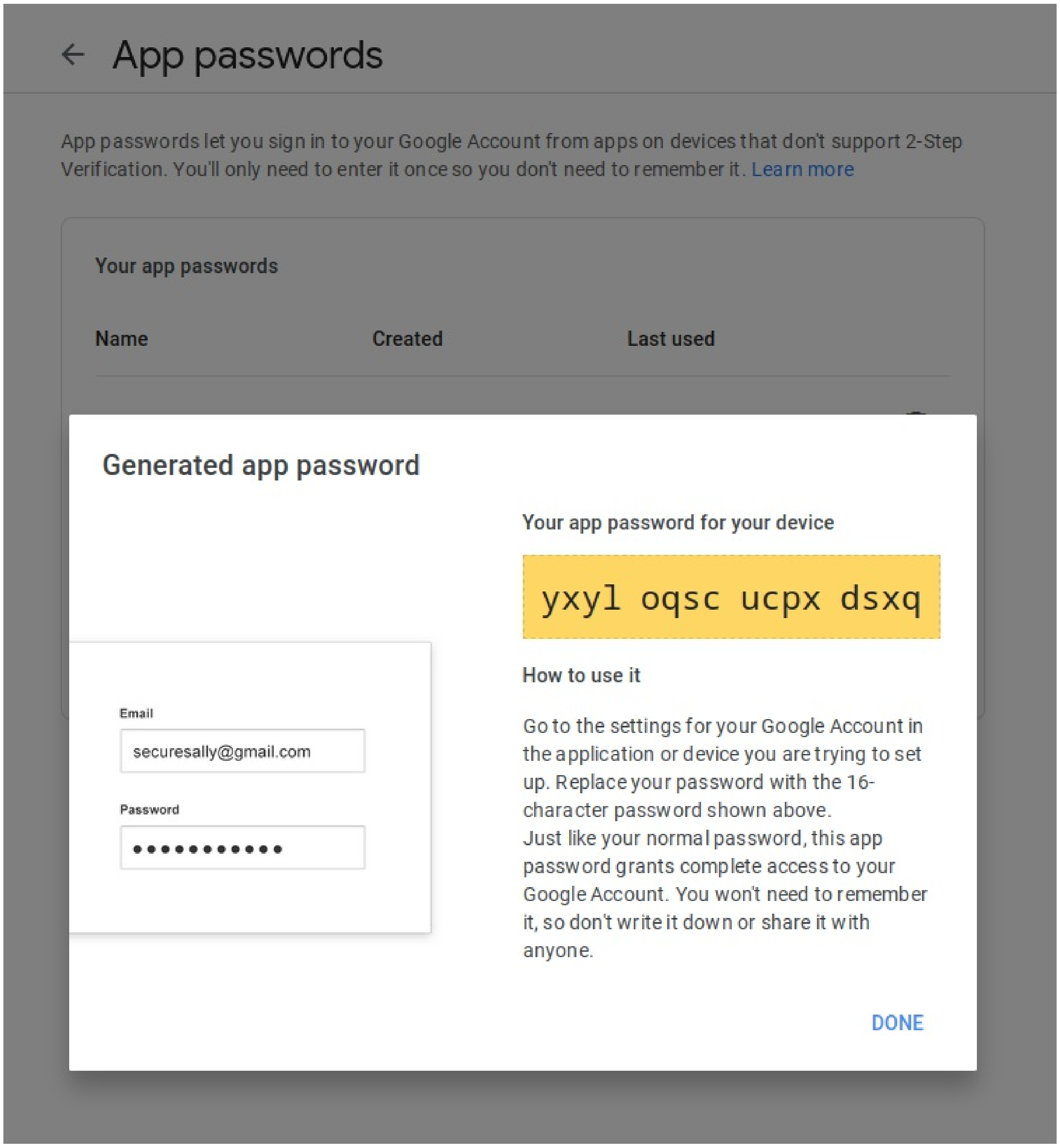
For details on setting up scheduled tasks, see the Windows, macOS, or Linux cron documentation.

There are many programmatic email services, such as SendGrid, Mandrill, and Mailgun. These are useful if we're generating a lot of email. They have official APIs and paid plans, and if we're setting up a large-scale operation, it's certainly worth looking into signing up to one of these services and using their Python libraries to send email. But for something small or personal, this can seem like a lot of effort, and there's an alternative if we have a Gmail account (as many people do).

There's an official Google Gmail Python API and module, but it's quite annoying to set up and use. Python comes with the smtplib and email modules as part of the built-in library, and these are perfectly capable of sending email via Gmail after a little setup. We can even use them to send email from ourself *to* ourself. We can't send too many emails this way, though. If we want to send tens or hundreds of emails to many different recipients, it's best to investigate the programmatic email services mentioned above. But as an email notification after a scheduled task, using Gmail from Python can be the ideal personal solution.

To use our Gmail account to send email this way, we first have to set up an *app password* for our Python script to use. Go to the App passwords of your

Google account, and under **Select app**, choose **Mail**, and under **Select device** choose **Other (custom name)** and fill in a name (such as “My Python Script”). We’ll be shown a screen that lists our new app password. Make a note of this password somewhere.



App passwords

App passwords let you sign in to your Google Account from apps on devices that don't support 2-Step Verification. You'll only need to enter it once so you don't need to remember it. Learn more

Your app passwords

Name	Created	Last used

Generated app password

Email: securesally@gmail.com

Password: ●●●●●●●●●●●

Your app password for your device

yxyl oqsc ucpx dsxq

How to use it

Go to the settings for your Google Account in the application or device you are trying to set up. Replace your password with the 16-character password shown above.
Just like your normal password, this app password grants complete access to your Google Account. You won't need to remember it, so don't write it down or share it with anyone.

To send an email, we’ll use the `smtplib` module. First, we need to define the content of our email. This part is our job. It’s a Python string, so we can substitute values in, use a templating language, or build it up from a list; whatever’s convenient. Here, we’ll use a simple example:

```
email_text = f"""
Hi! This is the report from our script.


We have added 1 + 2 and gotten the answer {1+2}.


Bye!
"""
```

We’ll also define two variables to hold our Gmail account details: the account name (which is the part of our Gmail address before `@gmail.com`) and the app password we just created:

```
GMAIL_USERNAME = "mygmailaccount12345"
GMAIL_APP_PASSWORD = "yxyloqscucpxdsxq"
```

Next, we’ll create the message as an object using the `email` module. An email can have many different properties, but the important ones here (in addition to the text of the body) are `To`, `From`, and `Subject`. `From` will be set to our Gmail address, which we’ve already defined, and `To` should be a string containing the email address the email is being sent to. This can be our own address, and if the email is going to more than one person, we need to separate the addresses by commas. We’ll define these in a list, because we’ll need the list later:

```
recipients = ["sil@kryogenix.org"]
msg = MIMEText(email_text)
msg["Subject"] = "Email report: a simple sum"
msg["To"] = ", ".join(recipients)
msg["From"] = f"{GMAIL_USERNAME}@gmail.com"
```

Finally, we'll use `smtplib` to connect to Gmail's mail server, log in with our provided details, and send the email. SMTP is the underlying protocol that's used to send email. When we send an email from our normal email app, SMTP is how the app actually does that. Here, we're doing it directly from our own code:

```
smtp_server = smtplib.SMTP_SSL('smtp.gmail.com', 465)
smtp_server.login(GMAIL_USERNAME, GMAIL_APP_PASSWORD)
smtp_server.sendmail(msg["From"], recipients, msg.as_string())
smtp_server.quit()
```

Our email should now be sent. Be aware, of course, that this is an introduction, so we've done no error handling or input validation. As mentioned, if we're sending lots of email, we should consider using a dedicated service, and we should think about the need to handle errors, failures to send, and the like. But sending one alert email to ourself, for example, can be done usefully with simple code like this.

Summary

Python gives us the ability to combine and control other things. Whether that's retrieving data from other people's APIs ready for processing, altering our own desktop and the settings and windows on it, or creating and sending emails, any time we plan to automate a task there's likely to be a Python library to help us. We've seen that Python is able to glue together data in different formats. Now we can also see that it's able to glue together different programs and systems as well.

CHAPTER 3: WORDS AND NUMBERS

Python is a powerful tool that can be used to solve a wide variety of problems, both big and small. In this tutorial, we'll explore how to use Python to answer questions, solve puzzles, and simulate various scenarios. Some are too large to solve without help, but small enough that they can be reasonably solved on a computer. For others, we'll use code to do some of the heavy lifting, to verify the answers we might have obtained some other way, or just to have a little fun. When we're playing word puzzles or checking out fun recreational math tricks, a little code can go a long way.

Word Ladder

There's a common word game where we're given two words (such as "SALT" and "MEAL") and we have to build a "ladder" from one word to the other, by changing one letter at a time so that each change still gives a legitimate word. In the example above, we might start with SALT and then change the first letter to M, giving MALT, then the second to give MELT, then to MEAT and finally to the destination of MEAL, in four steps.

This is quite a good way to sharpen our language skills—including our Python language skills, because we're going to cheat!

If we're going to do anything with words, we need a list of valid words. A good one for English is words_alpha.txt from the english-words repository.

Word Lists

There are similar lists available for other languages as well. Users on macOS or Linux are likely to already have one built in to the OS, in `/usr/share/dict/words`, which may save a download! This is a text file of words, one per line. It starts with “a”, then “aa”, and carries on until “zygotes” or “zwitterionic” or “Zyzzogeton”.

There are other, more comprehensive lists of words around. Scrabble famously has a list of officially sanctioned words (which we can even buy in hardcopy!), but one of the lists above will do for now.

The first building block towards solving this problem will be to find all the legitimate words we can make by changing one letter of another word. There are various ways to do this (and it’s worth considering how to do this *most efficiently* later), but for now we’ll keep things simple. Given a word, and a list of all words, find all the new words we can make by changing one letter.

First, we need the word list. We’ll load this into a Python set. A **set** is like a list, but it doesn’t have an order, and it can’t contain duplicates. This saves us from having to worry about the same word appearing in the set twice. Sets also have some convenient methods taken from mathematics. The **intersection** of two sets is all the entries that are in both sets, and the **difference** is all the entries that are in one or the other but not both:

```
>>> mylist = ["able", "echo", "site", "site", "salt", "type", "zo
>>> myset = set(mylist)
>>> myset
{'able', 'zoos', 'salt', 'echo', 'type', 'site'}
>>> myset2 = set(["code", "echo", "site", "walk", "webs"])
>>> myset2
```

```
{'echo', 'webs', 'code', 'site', 'walk'}
>>> myset.intersection(myset2)
{'site', 'echo'}
>>> myset.difference(myset2)
{'type', 'able', 'salt', 'zoos'}
```

Note that the word “site” appears in `mylist` twice but only once in `myset`.

For the purposes of the word ladder game, we only care about four-letter words, and we want to eliminate words with capital letters such as “Abby”, “AWOL”, or “YMCA”. So, load the words file from above (whether downloaded or provided by the OS) and extract each line as a word, as long as the word is four characters long and it’s all lowercase:

```
with open("/usr/share/dict/words") as fp:
    words = set([x.strip() for x in fp.readlines()
        if len(x.strip()) == 4 and x == x.lower()])
```

Now we need a function that can find all the possible ways to change one letter in a given word to get another word. One simple way to do this is to make a complete copy of the word list, and then successively discard items from it that do or don’t match.

Imagine that we want to find all words that can be produced from CODE by changing the third letter (which should give us COME, COKE, COPE, and so on). First, make a copy of the word list. Then cut the word list down to all those words that match the first letter of our start word, CODE—that is, all words that begin with C. The word list will now look like `{cabs, cads,` `cage, cagy, ...}`.

Now, let’s move on to the second letter, cutting the word list down again to all those where the second letter (O) matches. The word list now looks like `{coal, coat, coax, cobs, coda, code, cods, cogs, ...}`.

The third letter is the one we want to change! So now we cut the word list down again, but this time we only keep words that *differ* from our start word. That is, we keep only words from the list *without* a D in the third position. Now the list looks like `{coal, coat, coax, cobs, cogs, coif, coil, coin, ...}`.

Finally, let’s deal with the fourth letter. Here again, we keep words that match by having the E from code in the fourth position. This gives our final word list: `{coke, come, cone, cope, core, cote, cove}`. This list leaves us with all the words that can be formed by changing the third letter of CODE. In Python, it looks like this:

```
def change(word, charidx, words):
    matches = copy.copy(words)
    for i in range(len(word)):
        if i == charidx:
            matches = [x for x in matches if x[i] != word[i]]
        else:
            matches = [x for x in matches if x[i] == word[i]]
    return set(matches)
```

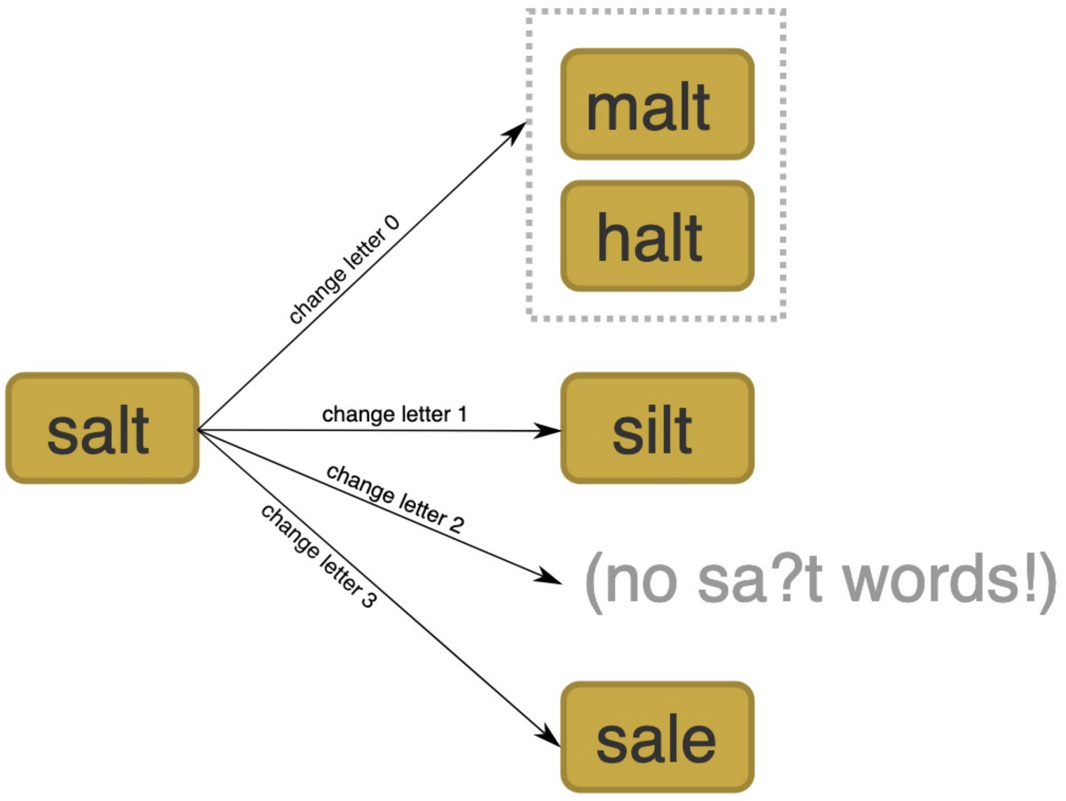
We can call this code to confirm what we’ve just established:

```
>>> print(wordladder.change('code', 2, wordladder.words))
{'cote', 'coke', 'cove', 'core', 'cope', 'cone', 'come'}
```

Note that we pass `2` as `charidx`, because it’s zero-based (so 0 for first character, 1 for second, etc.). Note also that the set returned by the function isn’t displayed in alphabetical order, because sets have no ordering.

With this, we can create a list of all possible words that can be found by changing one letter—by calling the `change` function four times, once for each letter. Let’s try that for SALT:

```
>>> word = "salt"
>>> changes = (list(change(word, 0, words)) + list(change(word, 1
               list(change(word, 2, words)) + list(change(word, 3
>>> changes
['halt', 'malt', 'silt', 'sale']
```



And with that, the steps to making a ladder are clear: find all the words that the start word can become, and find all the words that the end word can become, and look for crossovers. If there's a word in both these new sets, then that's our ladder. If there isn't, then do the step again: take each of the words we've made and evolve each of them into all *its* possible next words, and look again for crossovers.

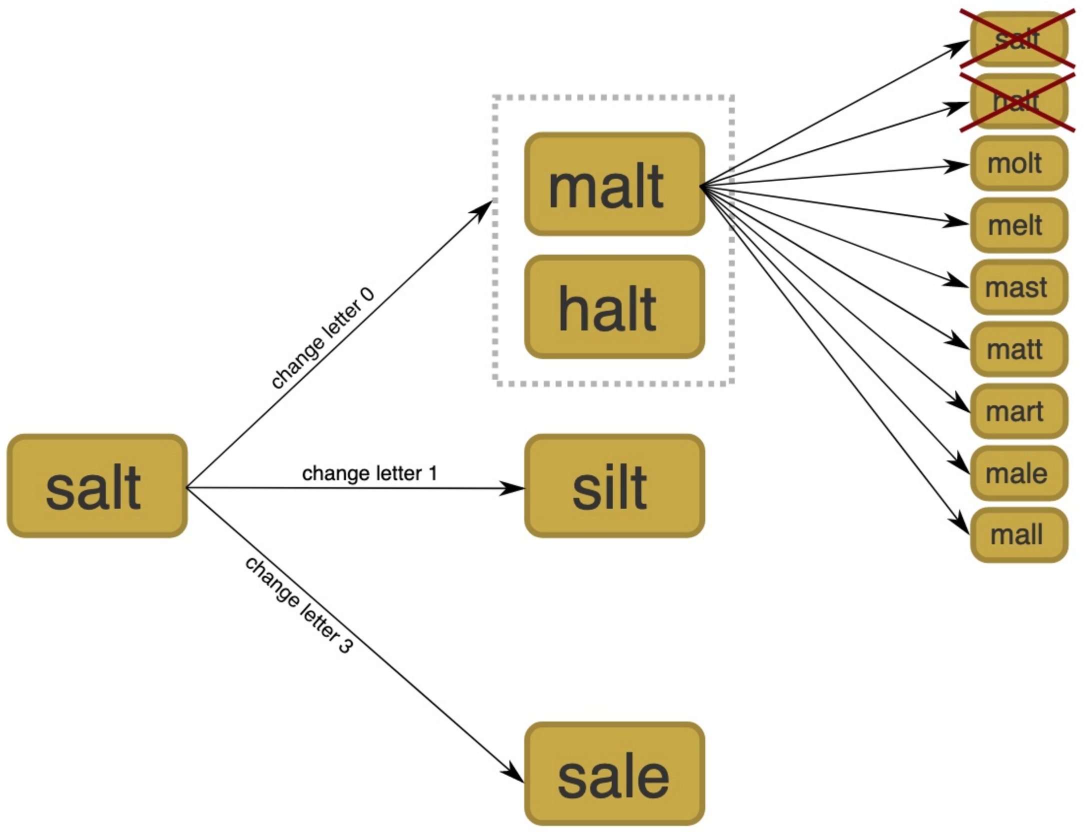
An example may help. To make a ladder from SALT to MATH, we take each of the words and evolve them one step:

- SALT with one change gives MALT, HALT, SILT, SALE.

- MATH with one change gives LATH, HATH, BATH, OATH, PATH, MYTH, MOTH, MASH, MATS, MATE, MATT.

There’s no word in common in those two lists, so we do it again. For each of those words, make another list:

- HALT gives SALT, MALT, HILT, HART, HAFT, HALL, HALO, HALE, HALF. We’ve already seen SALT and MALT in the lists, so discard those.
- MALT gives HALT, SALT, MELT, MOLT, MAST, MART, MATT, MALE, MALL. Again discard HALT and SALT … but now we have MATT, which is in the list from MATH, so we’ve found a word ladder! SALT to MALT to MATT to MATH, and that’s our answer.



To implement this, it will be useful to keep a few things: a **queue** (an ordered collection) of words we need to evolve, a set of words we've already seen, and a record of the chain taken to reach each word. The algorithm then looks something like this, in pseudocode:

```
Add the start word to the "from" queue
Add the end word to the "end" queue
Start the loop
For each word in the "from" queue:
    Get all possible words this word can evolve into
    For each of these evolutions:
        Add the evolution to the end of the "from" queue
        Set the chain for the evolution to be the chain of the fr
        Add the evolution to the "from" set of seen words
Do the same for each word in the "to" queue
Is there an intersection between the "from" set of seen words and
If there is, then we have a word ladder!
    Get the intersecting word and look up its chain in both chain
    Add the two chains together
    Print this combined chain as our word ladder
If there is no intersection:
    Go back to the loop and do it all again
```

Taking one step in this process—evolving the next word in the queue and then adding it to the appropriate lists—looks like this:

```
def step(queue, chain, used, words):
    nxt = queue.pop(0)
    changes = (list(change(nxt, 0, words)) + list(change(nxt, 1,
               list(change(nxt, 2, words)) + list(change(nxt, 3,
    changes = [c for c in changes if c not in used]
    for c in changes:
        chain[c] = chain[nxt] + [nxt]
        used.add(c)
        queue.append(c)
```

To check for intersections between the “from” set of seen words and the “to” set, we need a function that uses set intersections, and then combines the chains if an intersection is found:

```
def win(wto, wfrom, chainfrom, chainto):
    matches = wto.intersection(wfrom)
    if matches:
        m = list(matches)[0]
        return chainfrom[m] + [m] + list(reversed(chainto[m]))
```

Finally, the controlling loop sets up the lists, and repeatedly calls `step` and `win` until a ladder is found:

```
def ladder(start_word, end_word):
    wfrom = set()
    wto = set()
    qfrom = [start_word]
    qto = [end_word]
    chainfrom = {start_word: []}
    chainto = {end_word: []}

    while True:
        try:
            step(qfrom, chainfrom, wfrom, words)
            res = win(wto, wfrom, chainfrom, chainto)
            if res: return res
            step(qto, chainto, wto, words)
            res = win(wto, wfrom, chainfrom, chainto)
            if res: return res
        except IndexError:
            return ["No ladder found"]
```

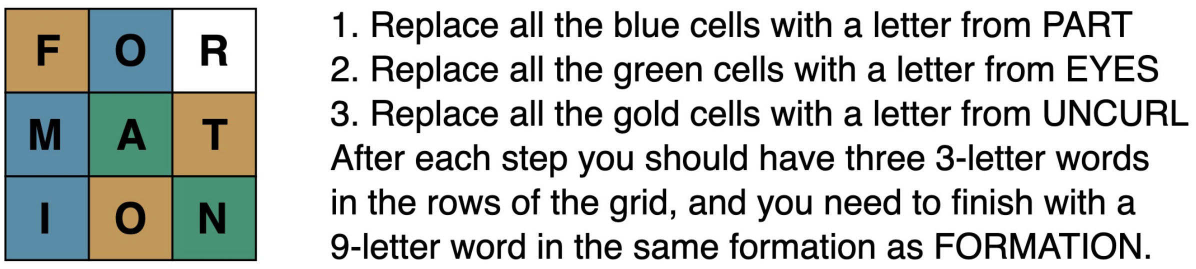
Now, calling the `ladder` function returns a word ladder as expected!

```
>>> ladder("salt", "meat")
['salt', 'malt', 'melt', 'meat']
```

Formation

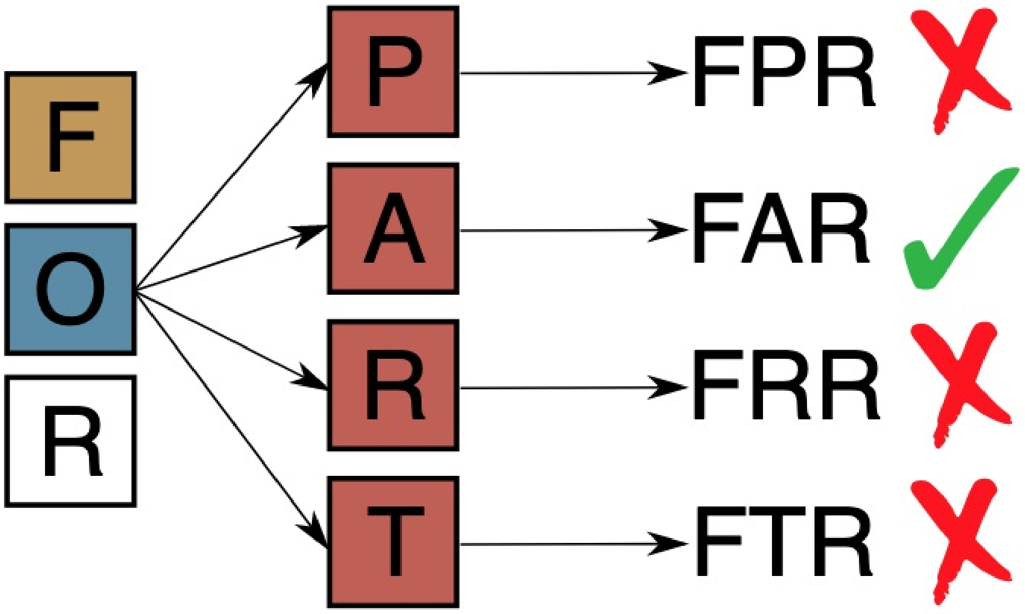
The techniques for solving word puzzles outlined above—by juggling letters and checking a word list—are applicable quite generally.

Let’s take another example. At Christmas time each year, the UK communications and spy agency GCHQ releases a set of puzzles to the public. In 2022, one of these puzzles (in the “Coding” category) was called Formation.



1. Replace all the blue cells with a letter from PART
2. Replace all the green cells with a letter from EYES
3. Replace all the gold cells with a letter from UNCURL
After each step you should have three 3-letter words in the rows of the grid, and you need to finish with a 9-letter word in the same formation as FORMATION.

So, what we’d really like here is a way to take a three letter word (such as FOR, the first word of the three), and get back a list of all legitimate words that result from replacing one letter (O, in blue) with each of the letters of another word (PART, the blue word).



We’ll need to do this for each of the three words, and for different letters depending on the color, so factoring this out into a function will be useful.

First, load the word list again in the same way as above. This time, we need a set of valid three-letter words:

```
with open("/usr/share/dict/words") as fp:
    words3 = set([x.strip() for x in fp.readlines()
        if len(x.strip()) == 3 and x == x.lower()])
```

Now we need the function to do replacements. Python has a convenient method for taking a substring of an existing string, called string slicing. For this, use square brackets and a zero-based “from” and “to” index. `"abcdefgh"`

`[2:5]` takes a substring starting at position 2 and ending at position 5, which is `"cde"`. We can leave out the “from” index, which makes the substring begin at the start of the string (so `"abcdefgh"[:5]` is `"abcde"`) and/or the “to” index (so the substring ends at the end, and thus `"abcdefgh"[4:]` is `"efgh"`). This means that if, for example, we want to replace the fifth letter of a string with an “X”, we can do so like this, where we get all the characters up to (but not including) the fifth letter (so the “to” index is 4), then the X for replacement, then all the characters after the fifth letter:

```
>>> mystr = "abcdefgh"
>>> mystr[:4] + "X" + mystr[5:]
'abcdXfgh'
```

With that, the replacement function might look like this:

```
def get_replacements(word, to_replace, replace_with):
    replacements = [word[:to_replace] + x + word[to_replace+1:]
        for x in replace_with]
    replacements = [w for w in replacements if w in words3 and w
    return replacements
```

We can now call this with a word, a character index to replace, and specify which letters to replace it with:

```
>>> get_replacements('for', 1, set('part'))
['far']
```

The function tries all the possible replacements of the O (the character at position 1—that is, the second character) with each of the letters of PART, giving FPR, FAR, FRR, and FTR. Only FAR is a word in the `words3` list, so that's all that's returned.

Given this function, solving the puzzle is now easy:

```
>>> get_replacements('mat', 0, set('part'))
['pat', 'tat', 'rat']
>>> get_replacements("ion", to_replace=0, replace_with=set("part"
['ton']
```

After the blue replacements, there are now three possibilities for the second word, and only one for each of the first and third. Similar calls to the function take care of the green and the gold replacements (skipping word 1 for green replacements because there aren't any green letters in it!):

```
# first the green
>>> get_replacements("pat", to_replace=1, replace_with=set("eyes"
['pet']
>>> get_replacements("tat", to_replace=1, replace_with=set("eyes"
[]
>>> get_replacements("rat", to_replace=1, replace_with=set("eyes"
[]
>>> get_replacements("ton", to_replace=1, replace_with=set("eyes"
['toe', 'toy']
# then the gold
>>> get_replacements("far", to_replace=0, replace_with=set("uncur
['car']
>>> get_replacements("pet", to_replace=2, replace_with=set("uncur
['per', 'pen']
>>> get_replacements("toe", to_replace=1, replace_with=set("uncur
[]
>>> get_replacements("toy", to_replace=1, replace_with=set("uncur
['try']
```

So the three words are CAR, PER or PEN, and TRY. Putting those three together yields only one valid word—CARPENTRY—which is the answer.

Rolling Dice

Tabletop roleplaying games have long used dice of many different shapes and sizes as random number generators, and it can be useful to treat this as something usefully handleable with a Python script. Many (many, many) dice-roller apps exist, but it's also easy enough to write our own dice-rolling code as well. Doing so provides a nice example of using randomness, number manipulation, and string parsing in Python.

In dice notation, rolling three six-sided dice is known as "3d6", and rolling two ten-sided dice is as "2d10". This will come in handy below.

Rolling dice is an application of randomness. Python's `random` module provides a handful of useful functions, two of which are `randint` (which gives a random number between two bounds), and `choice` (which chooses a random entry from a provided list). So to roll a d6 (a six-sided dice), we can use `random.randint(1, 6)`. This chooses one random number between and including the endpoints, so it will return a 1, a 2, a 3, a 4, a 5, or a 6. We can wrap this up in a convenient function that uses a list comprehension:

```
def roll_dice(count, die):
    return [random.randint(1, die) for i in range(count)]
```

Calling this will return a list of rolled numbers, so a roll of 3d6 will look like this:

```
>>> roll_dice(3, 6)
[4, 2, 1]
```

What range() Does

Note that we don't say `3 * [random.randint(1, die)]`. This would roll a random number only once and then return three copies of it, so the return value would be `[1, 1, 1]` or `[5, 5, 5]`. Instead, we roll separate dice the number of times specified by `count`. It can seem wasteful to generate a list of numbers `[1, 2, 3]`, which is what `range()` does, but Python is very efficient at this, so it's the standard way to do something a number of times.

The act of interpreting a string such as "3d6" or "2d10" is called **parsing**. For simple parsing, it's often common to use regular expressions—as we did in an earlier tutorial for extracting data from web pages. In this case, the regular expression is relatively simple: a "dice description" is the number of dice to roll, the letter "d", and the number of sides on the dice. The regular expression for this looks like `[0-9]+d[0-9]+`. It will be convenient to mark each of the sets of numbers as a group by putting them in brackets so they can be extracted later—which looks like `([0-9]+)d([0-9]+)`. Finally, it will also be convenient to give each group a name, so that we can extract the numbers later by name. Python offers an extension to regular expression syntax that does this: our regex will then look like `(?P<count>[0-9]+)d(?P<die>[0-9]+)`. Regular expressions can look rather like gibberish filled with strange characters, but if we build them up slowly they can be understood.

To use a regular expression with named groups, call `re.match`. This will return either `None` (if the regex doesn't match the passed string), or a `match` object with a `groupdict` method:

```
>>> import re
>>> match = re.match(r"(?P<count>[0-9]+)d(?P<die>[0-9]+)", "2d6")
>>> match
<re.Match object; span=(0, 3), match='2d6'>
>>> match.groupdict()
{'count': '2', 'die': '6'}
>>> re.match(r"(?P<count>[0-9]+)d(?P<die>[0-9]+)", "this is not a
>>>
```

Note that the last call returns `None` because "this is not a dice description" doesn't match the regular expression at all. Using `re.match` always insists that the whole string matches the regex. If we want to find a regex match somewhere within the string but without having to match the whole string, we should use `re.search`:

```
>>> re.search(r"(?P<count>[0-9]+)d(?P<die>[0-9]+)", "please roll
<re.Match object; span=(12, 15), match='2d6'>
```

Wrapping that up in a function also gives us a dice roller that makes use of our earlier `roll_dice` function:

```
def roll_string(dice_description):
    match = re.match(r"(?P<count>[0-9]+)d(?P<die>[0-9]+)", dice_d
    if not match:
        raise Exception(f"Invalid dice description {dice_descript
    return roll_dice(int(match.groupdict()["count"]),
                     int(match.groupdict()["die"]))
```

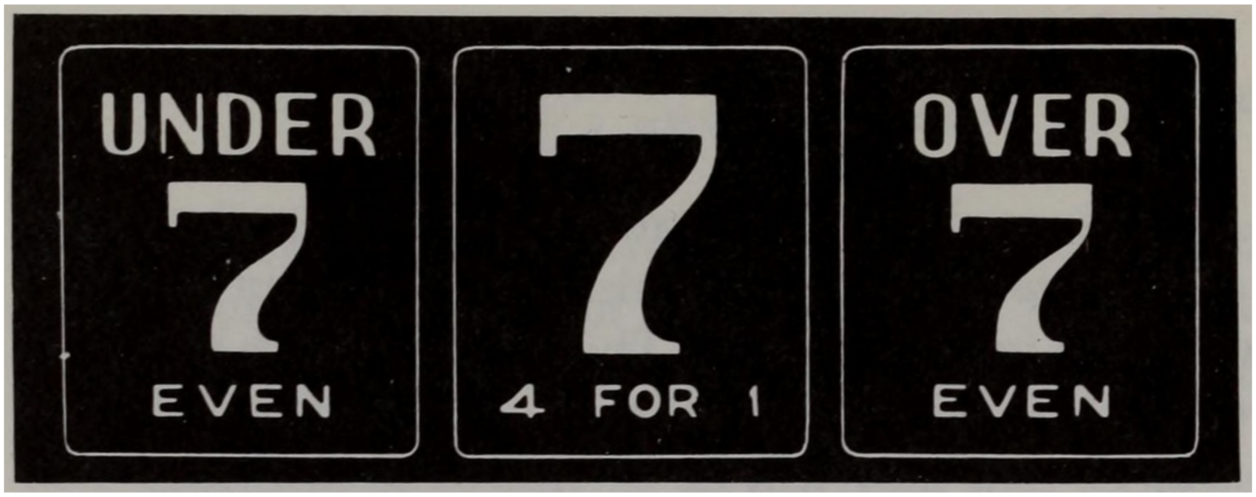
```
>>> roll_string("3d6")
[6, 4, 2]
>>> roll_string("3d6")
[5, 3, 6]
```

Probability

Puzzles and games—especially those with dice or other mathematics involved—can often come down to an assessment of probability. There are various mathematical techniques involved in understanding how probabilities combine. They help with problems like how to choose four unique items from a set of ten; how to work out the chances of rolling snake-eyes twice in a row; or how to deal with a hand of all red cards. But probability can be difficult to do, and computers are fast. So, in keeping with our theme here … let’s cheat! We can simply play the game a million times and see who mostly wins!

In fairness, this isn't just a technique for cheating. It's often useful to check an answer that's been arrived at by more rigorous mathematical means. If we've carefully calculated that the chance of dealing someone a poker hand containing two pairs is 4.7539%, it's a handy check to simulate that in a Python program by dealing ten million poker hands and confirming that around 500,000 of them are two pair. But if we don't need an exact answer, solving a puzzle by writing a Python script to do it many times over can be just as useful as working out the math, and sometimes quite a bit easier.

Casinos sometimes have a side game called "Under and Over 7". The rules are simple.



We bet on "under 7", which pays off at even odds (if we wager one chip, we get back our one chip and one more); "over 7" (also at even odds), or "7" (which pays off at "4 for 1": we get back three chips plus the one we bet, for four in total). Then we roll two dice. If the total of the two dice matches our bet—if we roll a 3 and a 2, for example, and we bet on "under 7"—then our bet pays out. If not, we lose our stake.

That’s Under and Over 7. These seem like pretty good odds, right? Four for one for rolling a seven, the most common dice roll on two dice? And even money on anything else? It almost seems like it’s hard to lose money at this game.

Working out the exact probabilities here is relatively simple for math whizzes, but getting to grips with that sort of thing can be awkward. Let’s simulate it with a bit of Python instead, which will give us the answer we need (should we play this game?) without needing to do a probability class first.

One game of Under and Over 7 looks something like this in Python:

```
def game():
    bet = random.choice(["under 7", "7", "over 7"])
    roll = sum(roll_string("2d6"))
    if roll < 7 and bet == "under 7":
        return 1
    elif roll > 7 and bet == "over 7":
        return 1
    elif roll == 7 and bet == "7":
        return 3
    else:
        return -1
```

We imagine that our computer player plays randomly, so it places a single bet of a single chip on the layout, choosing randomly which bet to place. This is what `random.choice` does: it chooses one item from a list, at random. Then we roll the dice with the `roll_string` dice roller, and get the result by summing the response (which looks like `[3, 5]`). If we make an “under 7” or “over 7” bet and also roll that, then we return 1 for our winnings (that is: we get back the chip we wagered, plus one more, for a total of 1 more than we started the game with). Successfully making a bet of “7” gives us winnings of 3. Missing the bet equates to a win of -1, because we lose the chip we bet.

We can test a single game out a few times to see what happens:

```
>>> game()
-1
>>> game()
-1
>>> game()
1
>>> game()
-1
>>> game()
3
>>> game()
-1
```

OK, that’s not looking too promising. But we can obviously test many more games with another loop:

```
def session(trials=10000):
    winnings = 0
    for i in range(trials):
        winnings += game()
    print(f"Total winnings after {trials} games: {winnings}")
```

```
>>> session()
Total winnings after 10000 games: -2244
>>> session()
Total winnings after 10000 games: -2210
>>> session(1000000)
Total winnings after 1000000 games: -223854
```

With that, we can see the truth: if we play ten thousand games of Under and Over 7, we'll be somewhere over two thousand chips worse off than when we started. The house does, in fact, always win.

Summary

It's fun to occasionally cheat at games, but Python can perform as the most *multi*- of multi-tools to explore wherever our mind takes us. Word puzzles and math explorations can be entertaining, and being able to throw a little code together to test out an exploration—whether that's following up on a Numberphile video, finding the longest palindrome, or solving coding challenges such as Advent of Code every December—can unlock a whole new range of questions that we can now answer.

Check out Peter Norvig's pytudes for some more detailed and in-depth approaches to using Python to solve problems of this kind. And if we now solve Wordle every day without fail … well, that'll just be our little secret.

CHAPTER 4: RUNNING SOMEONE ELSE'S PYTHON CODE

In the previous chapters of this book, we used existing Python modules to bring in new functionality to our scripts. Python (unlike many other languages) actually comes with modules for all sorts of useful functions. The developers of Python refer to this as a "batteries included" philosophy, where the language comes ready-supplied with many modules. The collection that comes with Python is called the **standard library** (a term often abbreviated to "stdlib"), and it's all documented as part of the Standard Library Reference.

In this stdlib, there are modules for doing things like this:

- accessing the filesystem and network and other system-level features
- working with compressed files
- doing simple cryptography
- manipulating dates and times
- using various internet-oriented systems such as HTTP and email
- providing implementations of many useful programming features such as queues, threads, asynchronous functions, command-line parsing, and functional programming

It's always best to look at the stdlib first when trying to solve a problem. We'll find help there good proportion of the time.

The Python Package Index

But help isn't *always* in the standard library. The stdlib is designed for containing modules that are useful to many Python programmers and for solving very general use cases. For anything specific, we'll want modules designed to deal with our specific problem. This is where PyPI comes in. PyPI, the Python Package Index, is the official repository for third-party Python packages. It's a vast collection of open-source software libraries, frameworks and applications that can be easily installed and used in Python projects. PyPI serves as a central hub for the distribution and discovery of Python packages. Developers can upload their packages to PyPI, which then makes them available to other developers who can easily download and install them. This makes PyPI the backbone of the Python ecosystem.

Packages from PyPI are installed with pip. We saw this in use when we installed the `requests` module to fetch web pages in the "Python for Stitching Together Other Things" tutorial, and this is a good example of a more specific Python module. The stdlib contains the `urllib.request` module for working with HTTP, so `requests` isn't strictly necessary, but it's an improvement over what the stdlib contains and is more pleasant to use.

We can install the requests module from PyPI with `python -m pip install requests`. (See below for the meaning of the `-m` flag.) Once that's done, the `requests` module is available to our code, and can be imported just like any module in the stdlib:

```
>>> import requests
>>> print(requests.get("https://sitepoint.com"))
<Response [200]>
```

This applies to any module we might need. If we're looking for something that has already solved a part of the problem we're working on, try searching for it. There's a search available on the PyPI website, but that's mostly useful for finding the exact details of a module that we already know exists; the search isn't very helpful for discovering *what* exists. (For example, if we're looking for something to write Excel spreadsheets, as we did in the first tutorial in this series, the thing to use is XlsxWriter. But a search on PyPI for "excel" doesn't bring it up until a later page.) It's better to use our favorite search engine; a search for "create excel file python" will likely bring up XlsxWriter as the top hit. Regardless of how we find out about a module, almost every published Python library is available in PyPI and therefore available to install via pip.

Virtualenv

It's useful to install Python modules required for a particular project in a way that makes them available to only that project. This is the purpose of a **virtualenv** (shorthand for "virtual environment", sometimes shortened still further to "venv"). Programmers of other languages may be familiar with the concept. For example, Node.js has its `node_modules` folder, and PHP has `composer` and its `vendor` folder.

Python requires a little setup for this; the virtual environment has to be created and then activated. Once a venv has been activated, any modules installed with pip will be installed into this venv rather than made available to the whole system. All this is accomplished via some complicated juggling with Python's system and site paths, and the Python docs explain the process in detail if we want to dig into the details, but fortunately we don't need to know how it works to use it.

To create a venv for a project, use Python's `venv` module. We'll need to specify a directory path for the virtual environment to live in; this will create the folder of our choice. It's conventional to create this in a folder named `venv` in the top-level directory of our project, but that's not actually required.

In a command prompt window, `cd` to the directory of our project and run `python -m venv ./venv`. This will create (but not activate) a new virtual environment in the `venv` folder in our current directory. This only needs to be done once. After this, whenever we want to work on the project, we can activate this virtual environment by running the activate command. Let's look at how.

Activation Commands

How to activate our virtual environment depends on which operating system and which command prompt we're using. In our command prompt window, `cd` to the directory of our project and do one of the following:

- Windows command prompt: `.\venv\Scripts\activate.bat`
- Windows PowerShell prompt: `.\venv\Scripts\Activate.ps1`
- macOS/Linux bash/zsh shell: `source ./venv/bin/activate`

(There are other options for less common shells; see the venv documentation for those.)

Also note that we'll need to run this every time we want to work on this project.

Seting Up a venv with an IDE

Setting up venvs, as described above, is a command-line process. However, if we’re using an IDE, that IDE may provide a convenient way to set up a venv for our project without us having to run the command-line parts directly. For example, if we’re using PyCharm, it’s worth reading the PyCharm virtual environment documentation.

As always, check the documentation. Any editor dedicated to Python will likely have a way to create virtual environments, because they’re so important to managing a project.

python -m

Using `python -m` is a way to run an installed module as a script. We can run any Python file as a script directly—by providing the path to it, of course. This is how we run any Python script we’ve written, with `python myscript.py`. Using `-m` first imports the named module and then runs it. This means that the thing being run can be anywhere on the Python import path, including in a venv. So `python -m myscript` will first do the equivalent of Python’s `import myscript` and *then* run it, meaning that we don’t have to provide a path to it.

There are many other options (sometimes called “flags” or “switches”) that can be passed to Python when running it in addition to `-m`, and as usual the Python documentation explains them all in detail, but the main other useful one is `-c`. This allows us to pass a quoted string of actual Python code on the command line rather than load it from a file. `python -c "print('Hello, world')"` will run that Python code directly—no file necessary. This can be useful for short Python tricks.

What to Do after Activation

Once we've activated the venv, any pip install commands will install our newly downloaded packages into this venv, and Python's `import` command will import things from the venv rather than from our system's list of Python packages. This means that, if we have a Python module installed at system level (outside a venv), we need to install it again once the venv is activated, because it's not available to this project specifically; the venv is a new, clean environment without extra modules installed. This is a useful feature. It means that different projects can use different modules—and different versions of the same module—without them colliding, and it helps us keep a handle on which modules are used by a particular project.

After creating and activating a venv for our project, use pip as normal to install new modules: `python -m pip install requests`. Now the `requests` module is available to our project and can be imported as normal.

Some command prompt windows will indicate that we've activated a virtualenv by adding it to the prompt, like this: `(venv) $`.

To deactivate a venv (that is, to exit it, so Python is no longer using it in our shell), type `deactivate` into our command prompt.

Installing Dependencies

Commonly, a complicated project may have a whole pile of modules that it needs to be installed, called **dependencies** or **requirements**. (Again, those familiar with other programming languages may have used `composer.json` or `package.json` files for this.) Python's equivalent is `requirements.txt`. If a project we're running (downloaded from GitHub or something similar) has a `requirements.txt` file, that file will contain a list of Python modules this project needs to be installed.

Fortunately, we don't need to manually install everything listed. Instead, we can run this:

```
python -m pip install -r requirements.txt
```

Once we've done so, pip will download and install each of the dependencies as if we'd commanded it to do so one by one. (For pip, the `-r` option means "load the list of things to install from a specified file", so that they don't have to be named in the command prompt. `python -m pip install help` will show all the many other pip options for installation.)

This is a good time to have (and to have activated) a venv for the project. Routinely creating a venv when installing packages is a good habit to get into when trying existing projects.

As an example (and since we looked at word puzzles in the previous tutorial in this series), let's take a simple word puzzle from GitHub and try it out. It's described as a "small Wordle clone written in Python", which seems ideal.

First, let's get the code with Git, via the command line. (We could also grab the code with GitHub Desktop or by using the tools in our code editor, such as Visual Studio Code.)

Learning Git

For any readers who are new to Git, a great way to get started with it is to check out *Jump Start Git*.

Here’s the command-line way to get the code for our Wordle clone:

```
git clone https://github.com/LauKr/wordle.git
Cloning into 'wordle'...
remote: Enumerating objects: 37, done.
remote: Counting objects: 100% (37/37), done.
remote: Compressing objects: 100% (31/31), done.
remote: Total 37 (delta 14), reused 15 (delta 4), pack-reused 0
Receiving objects: 100% (37/37), 43.19 KiB | 2.27 MiB/s, done.
Resolving deltas: 100% (14/14), done.
```

Next, open a command prompt and change to the directory of the project:

```
cd wordle/
```

Then create a new venv for this project:

```
python -m venv ./venv
```

Now that this is done, it won’t need to be done again.

Each time we want to try this project in a new command prompt, we’ll need to activate the venv. (This doesn’t need to be done before every command, but just at the beginning of a session or in a new window. A venv stays activated in the command prompt window we’re in until that command prompt window is closed.) Activate it using the appropriate command from above:

```
.\venv\Scripts\activate.bat
```

Since this is the first time we’ve opened the project, its dependencies aren’t installed. We can see this by attempting to run it:

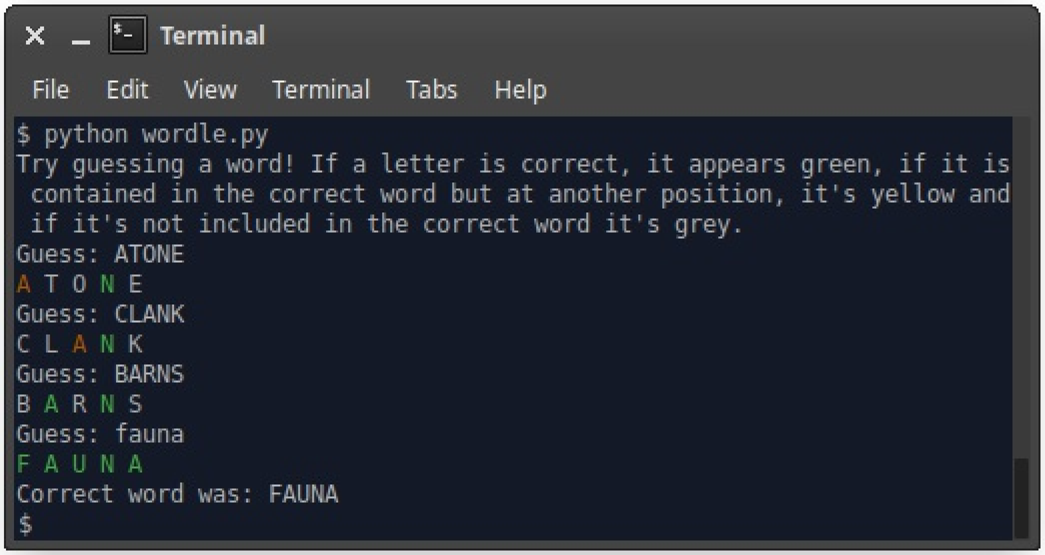
```
(venv) $ python wordle.py
Traceback (most recent call last):
  File ".../wordle.py", line 2, in <module>
    from termcolor import colored
ModuleNotFoundError: No module named 'termcolor'
```

It expects to have the `termcolor` module available, but we haven’t installed it. The project's author has provided a `requirements.txt`, so, as above, use it to install the dependencies:

```
(venv) $ python -m pip install -r requirements.txt
Collecting numpy==1.21.2
  Downloading numpy-1.21.2-cp310-cp310-manylinux_2_17_x86_64.many
Collecting termcolor==1.1.0
  Downloading termcolor-1.1.0.tar.gz (3.9 kB)
  Preparing metadata (setup.py) ... done
Using legacy 'setup.py install' for termcolor, since package 'whe
```

```
Installing collected packages: termcolor, numpy
  Running setup.py install for termcolor ... done
Successfully installed numpy-1.21.2 termcolor-1.1.0
```

Now the game will run successfully!



```
$ python wordle.py
Try guessing a word! If a letter is correct, it appears green, if it is
 contained in the correct word but at another position, it's yellow and
 if it's not included in the correct word it's grey.
Guess: ATONE
A T O N E
Guess: CLANK
C L A N K
Guess: BARNS
B A R N S
Guess: fauna
F A U N A
Correct word was: FAUNA
$
```

Writing Our Own Modules

Finally, of course, we can write our own modules. This can be useful for a number of purposes: code organization, listing our own project’s dependencies if we plan to release it, and running a module as a program.

Code Organization

The simplest form of organizing a slightly more complex Python script is to break it up into separate files, each with a singular purpose. The Python `import` statement works just as well for modules we’ve written as it does for dependencies and stdlib modules that were written by someone else.

Consider a simple script for converting a number to Roman numerals. Let’s imagine that we work at the BBC (British Broadcasting Corporation), who famously write the current year number in Roman numerals in the copyright notice at the end of programs. We’re having trouble remembering how to write a year’s number in Roman numerals, so we’ve written a useful Python script to work it out for us:

```
import sys


def int_to_roman(num):
    """ Convert an integer to a Roman numeral. """


    ints = (1000, 900,  500, 400, 100,  90, 50,  40, 10,  9,   5,
    nums = ('M',  'CM', 'D', 'CD','C', 'XC','L','XL','X','IX','V'
    result = []
    for i in range(len(ints)):
        count = int(num / ints[i])
        result.append(nums[i] * count)
        num -= ints[i] * count
    return ''.join(result)


if __name__ == "__main__":
    try:
        number = int(sys.argv[1])
    except (IndexError, ValueError):
        print(f"Syntax: {sys.argv[0]} <number>")
        sys.exit(1)
    print(f"{number} in Roman numerals is {int_to_roman(number)}"
```

Run as `python make-roman.py 2023` to get `2023 in Roman numerals is MMXXIII`. The script does a little checking of our input, and then uses a simple algorithm to convert the number.

Next, a colleague asks if there are any numbers in Roman numerals that are also words in the dictionary. We respond by saying that the very first one, I, is a word, but they're curious if there are any more. So we write another little script which checks all the numbers up to 3999 (because that's how high Roman numerals go) against the dictionary of all valid Scrabble words, `twl06`:

```
def int_to_roman(num):
    """ Convert an integer to a Roman numeral. """

    ints = (1000, 900,  500, 400, 100,  90, 50,  40, 10,  9,   5,
    nums = ('M',  'CM', 'D', 'CD','C', 'XC','L','XL','X','IX','V'
    result = []
    for i in range(len(ints)):
        count = int(num / ints[i])
        result.append(nums[i] * count)
        num -= ints[i] * count
    return ''.join(result)

def main():
    with open("twl06.txt") as fp:
        words = set([x.strip().upper() for x in fp.readlines()])
    roman_numerals = set([int_to_roman(x) for x in range(1, 4000)
    print("Roman numbers which are also words:")
    print(roman_numerals.intersection(words))

if __name__ == "__main__":
    main()
```

From this, we find the following valid words that are also valid Roman numbers: MM, XI, LI, MI, and MIX. (Scrabble players have a very expansive dictionary!)

But both of these scripts use the same `int_to_roman` function. It's wasteful to have the same code in two files, especially since it means that, if we ever make an improvement to the function, we'll have to change it in both places. So instead, we decide to pull that function out into its own file, `roman.py`, and then import it from there.

`roman.py` simply contains the function, like this:

```
def int_to_roman(num):
    """ Convert an integer to a Roman numeral. """

    ints = (1000, 900,  500, 400, 100,  90, 50,  40, 10,  9,   5,
    nums = ('M',  'CM', 'D', 'CD','C', 'XC','L','XL','X','IX','V'
    result = []
    for i in range(len(ints)):
        count = int(num / ints[i])
        result.append(nums[i] * count)
        num -= ints[i] * count
    return ''.join(result)
```

Now the improved, shorter version of `make-roman.py` can look like this:

```
import sys


from roman import int_to_roman


if __name__ == "__main__":
    try:
        number = int(sys.argv[1])
    except (IndexError, ValueError):
        print(f"Syntax: {sys.argv[0]} <number>")
        sys.exit(1)
    print(f"{number} in Roman numerals is {int_to_roman(number)}"
```

This uses `import` to bring in the specific function by name from `roman.py`.

The improved shorter version of `roman-words.py` looks like this:

```
import roman


def main():
    with open("twl06.txt") as fp:
        words = set([x.strip().upper() for x in fp.readlines()])
    roman_numerals = set([roman.int_to_roman(x) for x in range(1,
    print("Roman numbers which are also words:")
    print(roman_numerals.intersection(words))


if __name__ == "__main__":
    main()
```

This imports the whole `roman.py` module, and then calls a function within it as `roman.int_to_roman()`. (In this particular case, of course, there's only one function in `roman.py`, but a more complicated module would have many.)

This is how all the modules in the stdlib or in dependencies work, too. When we say `import random` and then call `random.randint()` to simulate a dice throw, we're not using a special technique that's only available to the stdlib. There's a `random.py` file shipped as part of Python, and we're importing a function defined in it named `randint`. And we can open that `random.py` file from our Python system installation and read the `randint` function, which is written in Python and begins `def randint(self, a, b):`—just as it would if we'd written it.

Listing Our Dependencies

If our code requires dependencies, it should come with a `requirements.txt` file so that others can use it. Fortunately, this is easy to do: `python -m pip freeze` will output the list of dependencies that are installed with pip. We should definitely be in a venv for this, of course; if we aren't, it will list all the Python modules that we've installed on the system, which isn't useful. All that's required is that we run the command, and redirect its output to a file called `requirements.txt`:

```
python -m pip freeze > requirements.txt
```

In practice, `requirements.txt` files aren't handwritten; they're created with `pip freeze`. But if we need to make or look at one, they consist of a textual list of required module names, optionally with version numbers. The pip documentation explains the format in detail, but as a simple example, the `requirements.txt` file for the Wordle project above looks like this:

```
numpy==1.21.2
termcolor==1.1.0
```

In the text above, `requirements.txt` is requiring the `numpy` and `termcolor` modules, each with a specific version number. We could install the modules (at their latest version) manually with `python -m pip install numpy termcolor`, or `python -m pip install numpy==1.21.2 termcolor==1.1.0` to install the particular versions specified. A `requirements.txt` file simply makes this easier.

Summary

Many scripts in Python don't need any external modules at all, requiring only what's available in Python's quite comprehensive standard library. But it's likely that, as our scripts get more complicated and detailed, we'll need resources beyond that. The Python Package Index is an extremely comprehensive list of Python modules written by others that we can freely install and use from our own code, via pip, and install into project-specific, separated virtual environments.

External programs, rather than libraries, will also need dependencies installed via pip, and a `requirements.txt` file is the standard way to indicate that. And those more complicated scripts will eventually benefit from having code organized into separate files to aid in reuse, which means we'll be using Python's import system.

What we've seen here only scratches the surface of what's available, and the Python documentation goes into much more detail about how import and virtual environments work for those wanting to dig into this further.

But we should only reach for complexity when we need it. Most of our scripts might not need anything at all. As we've seen from the tutorials making up this series, we can meet a lot of our needs simply by using Python's built-in standard library. Downloading and analyzing web pages, reading and converting data, and manipulating words and numbers, can all be done with the stdlib and maybe a library or two from PyPI. But it's good to know that, if we ever need it, we've got all the power in the world waiting for us. That's Python, by example.