Practical 3: Foundations (Part 2)

Getting to grips with Dictionaries, LOLs and DOLs, Packages and Functions

Table of contents

1 From Lists to Data (Little Steps) 2

2 Dictionaries 5

3 Tackling Programming Problems 10

4 Why 'Obvious' is Not Always 'Right' 19

Complete	Part 1: Foundations	Part 2: Data	Part 3: Analysis	
20%	▓▓▓▓▓▓▓▓	▓▓▓▓▓▓	▓▓▓▓▓▓	3/10

In this notebook we are exploring basic (in the sense of fundamental) data structures so that you understand both how to manage more complex types of data and are prepared for what we will encounter when we start using pandas to perform data analysis. To achieve that, you will need to be 'fluent' in "nested" lists and dictionaries; we will focus primarily on lists-of-lists and dictionaries-of-lists, but note that file formats like JSON can be understood as dictionaries-of-dictionaries-of-lists-of-... so this is just a *taster* of real-world data structures.

> **Tip**
>
> You should download this notebook from GitHub and then save it to your own copy of the repository. I'd suggest adding it (`git add ...`) right away and then committing (e.g. `git commit -m "Just added."`). Do this again at the end of the class (`git add ...`, `git commit -m ...`) and you'll have a record of everything you did. Then you can `git push` everything to GitHub.

> **Group Sign-Up**
>
> While there's no benefit to signing up early, if you've already formed a group then you can now Sign Up!

1 From Lists to Data (Little Steps)

We're going to start off using lists and dictionaries that *we* define right at the start of the 'program', but the *real* value of these data structures comes when we build a list or dictionary *from* data such as a file or a web page... and that's what we're going to do below!

First, here's a reminder of some useful methods (*i.e.* functions) that apply to lists which we covered in the lecture and practical in Week 2:

Method	Action
list.count(x)	Return the number of times x appears in the list
list.insert(i, x)	Insert value x at a given position i
list.pop([i])	Remove and return the value at position i (i is optional)
list.remove(x)	Remove the first element from the list whose value is x
list.reverse()	Reverse the elements of the list in place
list.sort()	Sort the items of the list in place
list.index(x)	Find the first occurence of x in the list
list[x:y]	Slice the list from index x to y-1

This should all be revision... because it's how we finished things up *last week*. But I want to go over it *briefly* again because we're going to build on it this week.

> **Hint**
>
> As before, ?? will highlight where one or more bit of code are missing and need to be filled in...

1.1 List Refresher

> Difficulty: Low.

To complete these tasks, all of the methods that you need are listed above, so this is about testing yourself on your understanding *both* of how to read the help *and* how to index elements in a list.

The next line creates a list of city names (each element is a string):

```
cities = ["Bristol", "London", "Manchester", "Edinburgh", "Belfast", "York"]
```

1.1.1 List Arithmetic

Replace the ?? so that it prints Belfast.

1.1.1.1 Answer

```
print(cities[2 + 2])
```

```
Belfast
```

1.1.2 Negative List Arithmetic

Use a **negative** index to print Belfast:

1.1.2.1 Answer

```
print(cities[-2])
```

```
Belfast
```

1.1.3 Finding a Position in a List

Replace the ?? so that it prints the *index* for Manchester in the list.

1.1.3.1 Answer

```
print("The position of Manchester in the list is: " + str( cities.index('Manchester') ))
```

```
The position of Manchester in the list is: 2
```

1.2 Looking Across Lists

> **Connections**
>
> This section draws on the LOLs lecture and you will also find Code Camp's Loops session useful here.

> **Difficulty: Medium.**

Notice that the list of temperatures below is the same length as the list of cities, that's because these are (roughly) the average temperatures for each city.

```
cities = ["Bristol", "London", "Manchester", "Edinburgh", "Belfast", "York"]
temperatures = [15.6, 16.5, 13.4, 14.0, 15.2, 14.8]
```

1.2.1 Lateral Thinking

Given what you know about cities and temperatures, how do you print:

"> The average temperature in Manchester is 13.4 degrees"

But you have to do this *without* doing any of the following:

1. Using a list index directly (*i.e.* cities[2] and temperatures[2]) or
2. Hard-coding the name of the city?

To put it another way, **neither** of these solutions is the answer:

```
print("The average temperature in Manchester is " + str(temperatures[2]) + " degrees.")
# ...OR...
city=2
print("The average temperature in " + cities[city] + " is " + str(temperatures[city]) + " degrees.")
```

> **Tip**
>
> You will need to combine some of the ideas above and also think about the fact that the list index is that we need is the same in both lists… Also, remember that you'll need to wrap a str(...) around your temperature to make it into a string.

1.2.1.1 Answer

```
city="Manchester" # Use this to get the solution...

# This way is perfectly fine
print("The average temperature in " + city + " is " + str(temperatures[cities.index(city)]))

# This way is more Python 3 and a bit easier to read
print(f"The average temperature in {city} is {temperatures[cities.index(city)]}")
```

```
The average temperature in Manchester is 13.4
The average temperature in Manchester is 13.4
```

1.2.2 Double-Checking Your Solution

You'll know that you got the 'right' answer to the question above if you can copy+paste your code and change only **one** thing in order to print out: "The average temperature in Belfast is 15.2 degrees"

1.2.2.1 Answer

```
city="Belfast"
print(f"The average temperature in {city} is {temperatures[cities.index(city)]}")
```

```
The average temperature in Belfast is 15.2
```

1.2.3 A Functional Solution

Can you convert this to a simple function called avg_tmp with two inputs and then use this to print the average temperature for Edinburgh?

1.2.3.1 Answer

```
def avg_tmp(city, temp):
    print(f"The average temperature in {city} is {temp}")

city="Edinburgh"
avg_tmp(city, temperatures[cities.index(city)])
```

```
The average temperature in Edinburgh is 14.0
```

1.2.4 Loops and Functions

Now use a for loop over the cities to print out the average temperature in each city using the function that we just created:

1.2.4.1 Answer

```
for c in cities:
    avg_tmp(c, temperatures[cities.index(c)])
```

```
The average temperature in Bristol is 15.6
The average temperature in London is 16.5
The average temperature in Manchester is 13.4
The average temperature in Edinburgh is 14.0
The average temperature in Belfast is 15.2
The average temperature in York is 14.8
```

2 Dictionaries

> **Connections**
>
> This section draws on the Dictionaries lecture and Code Camp Dictonaries session.

> **Difficulty: Low.**

Remember that dictionaries (a.k.a. dicts) are like lists in that they are data structures containing multiple elements. A key difference between dictionaries and lists is that while elements in lists are ordered, dicts (in most programming languages, though not Python) are unordered. This means that whereas for lists we use integers as indexes to access elements, in dictonaries we use 'keys' (which can multiple different types; strings, integers, etc.). Consequently, an important concept for dicts is that of key-value pairs.

2.1 Creating an Atlas

The code below creates an Atlas using a dictionary. The dictionary key is a city name, and the value is the latitude, longitude, and main airport code.

```
cities = {
   'San Francisco': [37.77, -122.43, 'SFO'],
   'London': [51.51, -0.08, 'LDN'],
   'Paris': [48.86,2.29, 'PAR'],
   'Beijing': [39.92,116.40 ,'BEI'],
}
```

2.1.1 Adding to a Dict

Add a record to the dictionary for Chennai (data here)

2.1.1.1 Answer

```
cities['Chennai'] = [13.08, 80.28, 'MAA']
```

2.1.2 Accessing a Dict

In *one* line of code, print out the airport code for Chennai (MAA):

2.1.2.1 Answer

```
print(f"The airport code for Chennai is {cities['Chennai'][2]}")
```

```
The airport code for Chennai is MAA
```

2.2 Dealing With Errors

Check you understand the difference between the following two blocks of code by running them.

```
try:
   print(cities['Berlin'])
except KeyError as e:
   print("Error found")
   print(e)
```

```
Error found
'Berlin'
```

```
try:
   print(cities.get('Berlin','Not Found'))
except KeyError as e:
   print("Error found")
   print(e)
```

```
Not Found
```

Notice that trying to access a non-existent element of a dict triggers a KeyError, while asking the dict to get the *same element* does not, it simply returns None. Can you think why, depending on the situtation, *either* of these might be the 'correct' answer?

2.3 Thinking Data

This section makes use of both the Dictionaries lecture and the DOLs to Data lecture.

> **Tip**
>
> In this section you'll need to look up (i.e. Google) and make use of a few new functions that apply to dictionaries: <dictionary>.items(), <dictionary>.keys(). *Remember*: if in doubt, add print(...) statements to see what is going on!

2.3.1 Iterating over a Dict

Adapting the code below, print out the city name and airport code for every city in our Atlas.

2.3.1.1 Answer

```
for c in cities.keys():
  print(f"{c} -> {cities[c][2]}")
```

```
San Francisco -> SFO
London -> LDN
Paris -> PAR
Beijing -> BEI
Chennai -> MAA
```

2.3.2 More Complex Dicts

How would your code need to change to produce the *same output* from this data structure:

```
cities = {
  'San Francisco': {
    'lat': 37.77,
    'lon': -122.43,
    'airport': 'SFO'},
  'London': {
    'lat': 51.51,
    'lon': -0.08,
    'airport': 'LDN'},
  'Paris': {
    'lat': 48.86,
```

```
        'lon': 2.29,
        'airport': 'PAR'},
    'Beijing': {
        'lat': 39.92,
        'lon': 116.40,
        'airport': 'BEI'},
    'Chennai': {
        'lat': 13.08,
        'lon': 80.28,
        'airport': 'MAA'}
}
```

2.3.2.1 Answer

```
for c in cities.keys():
    print(f"{c} -> {cities[c]['airport']}")
```

```
San Francisco -> SFO
London -> LDN
Paris -> PAR
Beijing -> BEI
Chennai -> MAA
```

2.3.3 More Dictionary Action!

And how would it need to change to print out the name and latitude of every city?

2.3.3.1 Answer

```
for c in cities.keys():
    print(f"{c} is at latitude {cities[c]['lat']}")
```

```
San Francisco is at latitude 37.77
London is at latitude 51.51
Paris is at latitude 48.86
Beijing is at latitude 39.92
Chennai is at latitude 13.08
```

2.3.4 And Another Way to Use a Dict

Now produce the *same output* using this new data structure:

```
cities_alt = [
    {'name':    'San Francisco',
     'position': [37.77, -122.43],
     'airport':  'SFO'},
    {'name':    'London',
     'position': [51.51, -0.08],
     'airport':  'LDN'},
```

```
  {'name':    'Paris',
   'position': [48.86, 2.29],
   'airport':  'PAR'},
  {'name':    'Beijing',
   'position': [39.92, 116.40],
   'airport':  'BEI'},
  {'name':    'Chennai',
   'position': [13.08, 80.28],
    'airport': 'MAA'}
]
```

2.3.4.1 Question

```
for c in cities_alt:
  print(??)
```

The output should be something like this:

```
San Francisco is at latitude 37.77
London is at latitude 51.51
Paris is at latitude 48.86
Beijing is at latitude 39.92
Chennai is at latitude 13.08
```

2.3.4.2 Answer

```
for c in cities_alt:
  print(f"{c['name']} is at latitude {c['position'][0]}")
```

```
San Francisco is at latitude 37.77
London is at latitude 51.51
Paris is at latitude 48.86
Beijing is at latitude 39.92
Chennai is at latitude 13.08
```

2.3.5 Think Data!

What are some of the main differences that you can think of between cities and cities_alt *as* data? There is no right answer.

> **Tip**
>
> I just want you to think about these *as data*! If you were trying to use cities and cities_alt as data what differences would you find when accessing one or more 'records'?

- Point 1 here.
- Point 2 here.
- Point 3 here.

3 Tackling Programming Problems

> **Connections**
>
> You will find links here to the Code Camp sessions on Functions and Packages, as well as to this week's lectures on Functions and Packages.

> **Difficulty: Medium.**

Let's now think about how to approach problems in programming (with code) and how that might differ from other ways of thinking.

The problem we will use here as an example is: *download a data file that we know is hosted on a web site and output some information about those data*. This sounds hard. It *is* hard when you're just starting out in programming. But it is *not* hard for a computer… *iff* we can figure out what to tell it to do *and* make use of work that other people have done for us!

3.0.1 What Do We Do? Break It Down!

3.0.1.1 Step 1. Analyse the Problem

The first step to writing a program is thinking about your goal and the steps required to achieve that. We ***don't*** write programs like we write essays: all at once by writing a whole lot of code and then hoping for the best when we hit 'submit'.

When you're tackling a programming problem you break it down into separate, simpler steps, and then tick them off one by one. Doing this gets easier as you become more familiar with programming, but it remains crucial and, in many cases, good programmers in large companies spend more time on *design* than they do on actual *coding*.

3.0.1.2 Step 2. Functions & Packages

We have discussed how *functions* are a useful programming tool to enable us to reuse chunks of code. Basically, a function is a way to do something to something in a portable, easy-to-use little bundle of code.

Some steps in a program are done so many times by so many people that, eventually, someone writes a *package* that bundles up those operations into something easy to use that saves *you* having to figure out the gory details. Reading a file (even one on a computer halfway round the world) is one of those things. Making sense of the data *in* that file for you is probably not.

Figure 1: xkcd: Easy vs. Hard

To a computer, reading data from a remote location (e.g. a web site halfway around the world) is not really any different from reading one that's sitting on your your local hard drive (e.g. on your desktop). To simplify things a great deal: the computer really just needs to know the location of the file and an appropriate *protocol* for accessing that file (*e.g.* http, https, ftp, local...) and then a clever programming language like Python will typically have packages that can kind of take of the rest.

In all cases – local and remote – you use the package to handle the hard bit of knowing how to actually 'read' data (because all files are just 1s and 0s of data) at the *device* level and then Python gives you back a 'file handle' that helps you to achieve things like 'read a line' or 'close an open file'. You can think of a filehandle as something that gives you a 'grip' on a file-like object no matter where or what it is, and the package is the way that this magic is achieved.

3.0.1.3 Step 3. Look for Ways to Recycle

Always look for ways to avoid reinventing the wheel. This is where Python's packages (or R's for that matter) come into their own. If it's something that programmers often need to do, then chances are that someone has written a package to do it!

The point of packages is that they can help us to achieve quite a lot very quickly since we can just make use of someone else's code. In the same way that we won't mark you down for Googling the answer to a coding question, we *also* won't mark you down for using someone else's package to help you get going with your programming. ***That's the whole point!***

Often, if you're not sure where to start, Google (or StackOverflow) is the place to go:

how to read text file on web server python

Boom!

3.0.1.4 Step 4. Make a Plan

OK, so we need to break this *hard* problem down into something simpler. We can do this by thinking about it as three separate steps:

1. We want to read a remote file (i.e. a text file somewhere the planet),
2. We want to turn it into a local data structure (i.e a list or a dictionary),
3. We want to perform some calculations on the data (e.g. calculate the mean, find the easternmost city, etc.).

We can tackle each of those in turn, getting the first bit working, then adding the second bit, etc. It's just like using lego to build something: you take the same pieces and assemble them in different ways to produce different things.

3.1 Reading a Remote File

So, we are going to download a file from GitHub, but we **aren't going to to try to turn it into data** or otherwise make 'sense' of it yet, we just want to **read** it. We are then going to build from this first step towards the rest of the steps!

Because we're accessing data from a 'URL' we need to use the urlopen function from the urllib.request package. If you're wondering how we know to use this function and package, you might google something like: *read remote csv file python 3* which in turn might get you to a StackOverflow question and answer like this.

```
from urllib.request import urlopen
help(urlopen)
```

```
Help on function urlopen in module urllib.request:

urlopen(url, data=None, timeout=<object object at 0x10091cb70>, *, cafile=None, capath=None, cadefault=False, c
    Open the URL url, which can be either a string or a Request object.

    *data* must be an object specifying additional data to be sent to
    the server, or None if no such data is needed.  See Request for
    details.

    urllib.request module uses HTTP/1.1 and includes a "Connection:close"
    header in its HTTP requests.

    The optional *timeout* parameter specifies a timeout in seconds for
```

blocking operations like the connection attempt (if not specified, the
global default timeout setting will be used). This only works for HTTP,
HTTPS and FTP connections.

If *context* is specified, it must be a ssl.SSLContext instance describing
the various SSL options. See HTTPSConnection for more details.

The optional *cafile* and *capath* parameters specify a set of trusted CA
certificates for HTTPS requests. cafile should point to a single file
containing a bundle of CA certificates, whereas capath should point to a
directory of hashed certificate files. More information can be found in
ssl.SSLContext.load_verify_locations().

The *cadefault* parameter is ignored.

This function always returns an object which can work as a
context manager and has the properties url, headers, and status.
See urllib.response.addinfourl for more detail on these properties.

For HTTP and HTTPS URLs, this function returns a http.client.HTTPResponse
object slightly modified. In addition to the three new methods above, the
msg attribute contains the same information as the reason attribute ---
the reason phrase returned by the server --- instead of the response
headers as it is specified in the documentation for HTTPResponse.

For FTP, file, and data URLs and requests explicitly handled by legacy
URLopener and FancyURLopener classes, this function returns a
urllib.response.addinfourl object.

Note that None may be returned if no handler handles the request (though
the default installed global OpenerDirector uses UnknownHandler to ensure
this never happens).

In addition, if proxy settings are detected (for example, when a *_proxy
environment variable like http_proxy is set), ProxyHandler is default
installed and makes sure the requests are handled through the proxy.

As you can see, there is *lot* of information here about how things work. A *lot* of it won't make much sense at the moment. That's ok. *Some* of this doesn't make much sense to me, but that's because this is the *full* documentation from Python so it's trying to cover *all* the bases. You don't need to read every line of this, what you are looking is information about things like the 'signature' (what parameters the function accepts) and its output. Of course, you can also *just Google it*!

> **Tip**
>
> Remember that you can use `dir(...)` and `help(...)` to investigate what a package offers. You can also get help in Jupyter by typing `?` before the function that you want to call.

> **URLError?**
>
> If you are working behind a firewall (esp. if working on this practical in, say, China) then there is a *chance* you will get a URLError (<urlopen error [Errno 110044] getaddrinfo failed>). This is a 'proxy error' and in this case you may need to configure your environment as follows:
>
> ```
> import os
> os.environ['HTTP_PROXY'] = 'http://127.0.0.1:10809'
> os.environ['HTTPS_PROXY'] = 'http://127.0.0.1:10809'
> ```

Before you start working on the code, why not open the data file directly in your browser? It's pretty small, and it will give you a sense of what is going on.

```
from urllib.request import URLError
from urllib.request import urlopen

# Given the info you were given above, what do you
# think the value of 'url' should be? What
# type of variable is it? int or string?
url = 'https://raw.githubusercontent.com/jreades/fsds/master/data/src/Wikipedia-Cities-simple.csv'

# Read the URL stream into variable called 'response'
# using the function that we imported above
try:
   response = urlopen(url)
except URLError as e:
   print("Unable to connect to URL!")
   print(e)

# You might want to explore what `__class__` and `__name__`
# are doing, but basically the give us a way of finding out what
# is 'behind' more complex variables

# Now read from the stream, decoding so that we get actual text
raw = response.read()

print(f"'raw' variable is of type: '{raw.__class__.__name__}'.")
print(f"Raw content is:\n{raw[:75]}...\n")

data = raw.decode('utf-8')

print(f"'data' variable is of type: '{data.__class__.__name__}'.")
print(f"Decoded content is:\n{data[:75]}...")
```

> **Note**
>
> Notice that the raw data has the format b'...' with all of the data seemingly on one line, while the *decoded* version in data is 'correctly' structured with lines! The 'raw' data is in *bytecode* format which is not, strictly, a string. It only becomes a string when we 'decode it' to utf-8 (which is the 'encoding' of text that supports

> most human languages). While the computer doesn't particularly care, we do!

Remember that you can treat strings *as lists*, so when we print below we cut off the output using the list[:<Some Number>] syntax.

```
print(f"There are {len(data)} characters in the data variable.")
print(f"The first 125 characters are: '{data[:125]}'") # Notice that '\n' count here!
```

```
There are 352 characters in the data variable.
The first 125 characters are: 'City,Population,Latitude,Longitude
Perth,45770,56.39583,-3.43333
Armagh,14777,54.3499,-6.6546
Dundee,147268,56.462,-2.9707'
```

So this is definitely text, but it doesn't (yet) look entirely like the data *we* see because it's still just one long string, and not *data* which has individual records on each line. To split the text into individual lines, we can use the handily named .splitlines() method (more on methods below):

3.1.0.1 Answer

```
rows = data.splitlines()
print(f"'rows' variable is of type: {rows.__class__.__name__}'.")
```

```
'rows' variable is of type: list'.
```

Note now, how the *data* variable has type list. So to view the data as we see them in the original online file, we can now use a for loop to print out each element of the list (each element being a row of the original online file):

3.1.0.2 Answer

```
print(f"There are {len(rows)} rows of data.")
print("\n".join(rows[0:2])) # New syntax alert! notice we can *join* list elements
```

```
There are 11 rows of data.
City,Population,Latitude,Longitude
Perth,45770,56.39583,-3.43333
```

That's a little hard to read, though something has clearly changed. Let's try printing the last row:

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

```
Bangor,18808,53.228,-4.128
```

Congratulations! You've now read a text file sitting on a server in, I think, Canada and Python *didn't care*. You've also converted a plain-text file to a row-formatted list.

3.2 Text into Data

We now need to work on turning the response into useful data. We got partway there by splitting on line-breaks (splitlines()), but now we need to get columns for each line. You'll notice that we are dealing with a *CSV* (Comma-Separated Value) file and that the format *looks* quite simple… So, in theory, to turn this into data we 'just' need to *split* each row into separate fields using the commas.

There's a handy function associated with strings called split:

```
print('abcdefgh'.split('d'))
```

```
['abc', 'efgh']
```

You can also investigate further how the split function works using:

```
help('abcdefgh'.split)
```

```
Help on built-in function split:

split(sep=None, maxsplit=-1) method of builtins.str instance
    Return a list of the substrings in the string, using sep as the separator string.

      sep
        The separator used to split the string.

        When set to None (the default value), will split on any whitespace
        character (including \n \r \t \f and spaces) and will discard
        empty strings from the result.
      maxsplit
        Maximum number of splits.
        -1 (the default value) means no limit.

    Splitting starts at the front of the string and works to the end.

    Note, str.split() is mainly useful for data that has been intentionally
    delimited.  With natural text that includes punctuation, consider using
    the regular expression module.
```

So this seems like a good solution to turn our text into *data*:

```
test = rows[-1].split(',')
print(test)
print(f"The population of {test[0]} is {int(test[1]):,}")
```

```
['Bangor', '18808', '53.228', '-4.128']
The population of Bangor is 18,808
```

I'd say that we're now getting quite close to something that looks like 'real data': I know how to convert a raw response from a web server into a string, to split that string into rows, and can even access individual elements from a row!

> **Caution**
>
> There are two problems to the data.splitlines() and row.split(',') approach! One of them is visible (though not obvious) in the examples above, the other is not.

1. Remember that 10 and '10' are *not* the same thing. To comma-format the population of Sheffield you'll see that I had to do int(...) in order to turn '685368' into a number. So our approach so far doesn't know anything about the *type* of data we're working with.
2. We are also implicitly *assuming* that commas can only appear at field boundaries (i.e. that they can only appear to separate one column of data from the next). In other words, just using split(',') doesn't work if *any* of the fields can themselves contain a comma!
3. There's actually a *third* potential issue, but it's so rare that we would need to take a completely different approach to deal with it: we are also assuming that newlines (\n) can only appear at record boundaries (i.e. that the can only appear to separate one row of data from the next). In those cases, using splitlines() also doesn't work, but this situation is (thankfully) very rare indeed.

This is where using code that someone *else* who is much more interested (and knowledgeable) has written and contributed is helpful: we don't need to think through how to deal with this sort of thing ourselves, we can just find a library that does what we need and make use of *its* functionality. I've given you the skeleton of the answer below, but you'll need to do a little Googling to find out how to "read csv python".

Note: For now just focus on problem #2.

```
from urllib.request import urlopen
import csv

url = 'https://raw.githubusercontent.com/jreades/fsds/master/data/src/Wikipedia-Cities-simple.csv'
response = urlopen(url)
raw = response.read()

# Now take the raw data, decode it, and then
# pass it over to the CSV reader function
csvfile  = csv.reader(raw.decode('utf-8').splitlines())

urlData = [] # Somewhere to store the data
for row in csvfile:
   urlData.append( row )

print("urlData has " + str(len(urlData)) + " rows and " + str(len(urlData[0])) + " columns.")
print(urlData[-1]) # Check it worked!
```

If it worked, then you should have this output:

```
urlData has 11 rows and 4 columns.
['Bangor', '18808', '53.228', '-4.128']
```

To you that might look a lot *worse* that the data that you originally had, but to a computer that list-of-lists is something it can work with; check it out:

```
for u in urlData[1:6]: # For each row in the first 5 items in list
    print(f"The city of '{u[0]}' has a population of {int(u[1]):,}") # Print out the name and pop
```

```
The city of 'Perth' has a population of 45,770
The city of 'Armagh' has a population of 14,777
The city of 'Dundee' has a population of 147,268
The city of 'Colchester' has a population of 194,706
The city of 'Salisbury' has a population of 40,302
```

> **Note**
>
> Why did I use `urlData[1:]` instead of `urlData`?
> If you print `urlData[0]` you'll see that this is the 'header' row that tells us what each column contains! So if we try to convert the column name to an integer (`int(u[1])`) we will get an error!
> The advantage of using the `csv` library over plain old `string.split` is that the `csv` library knows how to deal with fields that contain commas (*e.g.* `"Cardfiff, Caerdydd"` or `"An Amazing 4 Bedroom Home, Central London, Sleeps 12"`) and so is much more flexible and consistent that our naive `split` approach. The vast majority of *common* tasks (reading certain types of files, getting remote files, etc.) have libraries that do exactly what you want without you needing to write much code yourself to take advantage of it. You should always have a look around online to see if a library exists before thinking that you need to write everything/anything from scratch. The tricky part is knowing what words to use for your search and how to read the answers that you find…

Let's try this with a 'bigger' data set… In an ideal world, the 'power' of code is that once we've solved the problem *once*, we've solved it more generally as well. So let's try with the 'scaled-up' data set and see waht happens!

```
from urllib.request import urlopen
import csv

url = "https://raw.githubusercontent.com/jreades/fsds/master/data/src/Wikipedia-Cities.csv"
response = urlopen(url)
raw = response.read()

csvfile = csv.reader(raw.decode('utf-8').splitlines())

urlData = [] # Somewhere to store the data

for row in csvfile:
    urlData.append( row )

print(f"urlData has {len(urlData)} rows and {len(urlData[0])} columns.")

for u in urlData[70:]:  # For each row in the list
    print(f"The city of '{u[0]}' has a population of {u[1]}") # Print out the name and pop
```

⚠ What mistake have I made here?

I have assumed that, just because the files have similar names, they must also have similar layouts!

```
print(f"The URL's data labels are: {', '.join(urlData[0])}")
```

```
The URL's data labels are: City, Population, Latitude, Longitude
```

3.4 Insight!

So, although the code was basically the same for both of these files (good), we would need to change quite a bit in order to print out the *same* information from different versions of the *same data*. So our code is rather **brittle**.

One of the issues is that our *instincts* about how to manage data doesn't align with how the computer can most *efficiently* manage it. We make the mistake of thinking that the computer needs to do things that same way that we do when reading text and so assume that we need to:

1. Represent the rows as a list.
2. Represent the columns as a list for each row.

This thinking suggests that the 'right' data structure would clearly be a list-of-lists (LoLs!), but if you understand what happened here then the next section will make a *lot* more sense!

4 Why 'Obvious' is Not Always 'Right'

ℹ Connections

This section builds on the material covered by the DOLs to Data lecture.

Difficulty: Hard.

But you need to be careful assuming that, just because something is hard for you to read, it's also hard for a computer to read! The way a computer 'thinks' and the way that we think doesn't always line up naturally. Experienced programmers can think their way *around* a problem by working *with* the computer, rather than against it.

Some issues to consider:

- Is the first row of data *actually* data, or is it *about* data?
- Do we really care about column *order*, or do we just care about being able to pick the *correct* column?

Let's apply this approach to the parsing of our data...

If you stop to think about it, then our list-of-lists approach to the data isn't very easy to navigate. Notice that if the position or name of a column changes then we need to change our program *every* time we re-run it! It's not very easy to read *either* since we don't really know what `u[5]` is supposed to be. That way lies all kinds of potential errors!

Also consider that, in order to calculate out even a simple aggregate such as the sum of a field for all rows we need to step through a lot of irrelevant data as well: we have to write a `for` loop and then step through each row with an 'accumulator' (somewhere to store the total). That's slow.

That doesn't make much sense since this should all be *easier* and *faster* in Python than in Excel, but right now it's *harder*, and quite possibly *slower* as well! So how does the experienced programmer get around this? 'Simple' (i.e. neither simple, nor obvious, until you know the answer): she realises that the data is organised the wrong way! We humans tend to think in rows of data: this apartment has the following *attributes* (price, location, etc.), or that city has the following *attributes* (population, location). We read across the row because that's the easiest way for *us* to think about it. But, in short, a list-of-lists does *not* seem to be the right way to store this data!

Crucially, a computer doesn't have to work that way. For a computer, it's as easy to read *down* a column as it is to read *across* a row. **In fact, it's easier**, because each column has the same *type* of data: one column contains names (strings), another column contains prices (integers), and other columns contain other types of data (floats, etc.). Better still, the order of the columns often doesn't matter as long as we know what the columns are called: it's easier to ask for the 'description column' than it is to ask for the 6th column since, for all we know, the description column might be in a different place for different files but they are all (relatively) likely to use the 'description' label for the column itself.

4.1.1 A Dictionary of Lists to the Rescue

So, if we don't care about column order, only row order, then a dictionary of lists would be a nice way to handle things. And why should we care about column order? With our CSV files above we already saw what a pain it was to fix things when the layout of the columns changed from one data set to the next. If, instead, we can just reference the 'description' column then it doesn't matter where that column actually is. Why is that?

Well, here are the first four rows of data from a list-of-lists for city sizes:

```
myData = [
 ['id', 'Name', 'Rank', 'Longitude', 'Latitude', 'Population'],
 ['1', 'Greater London', '1', '-18162.92767', '6711153.709', '9787426'],
 ['2', 'Greater Manchester', '2', '-251761.802', '7073067.458', '2553379'],
 ['3', 'West Midlands', '3', '-210635.2396', '6878950.083', '2440986']
]
```

Now, here's how it would look as a dictionary of lists organised by *column*, and *not* by row:

```
myData = {
   'id'       : [0, 1, 2, 3, 4, 5],
   'Name'     : ['London', 'Manchester', 'Birmingham','Edinburgh','Inverness','Lerwick'],
   'Rank'     : [1, 2, 3, 4, 5, 6],
   'Longitude'  : [-0.128, -2.245, -1.903, -3.189, -4.223, -1.145],
   'Latitude'   : [51.507, 53.479, 52.480, 55.953, 57.478, 60.155],
   'Population' : [9787426, 2705000, 1141816, 901455, 70000, 6958],
}

print(myData['Name'])
print(myData['Population'])
```

```
['London', 'Manchester', 'Birmingham', 'Edinburgh', 'Inverness', 'Lerwick']
[9787426, 2705000, 1141816, 901455, 70000, 6958]
```

What does this do better? Well, for starters, we know that everything in the 'Name' column will be a string, and that everything in the 'Longitude' column is a float, while the 'Population' column contains integers. So that's made life easier already, but the real benefit is coming up…

4.1.2 Behold the Power of the DoL

Now let's look at what you can do with this… but first we need to import one *more* package that you're going to see a *lot* over the rest of term: numpy (Numerical Python), which is used *so* much that most people simply refer to it as np. This is a *huge* package in terms of features, but right now we're interested only in the basic arithmatic functions: mean, max, and min.

> We'll step through most of these in detail below.

Find the latitude of Manchester:

```
city = "Manchester"
lat = myData['Latitude'][ myData['Name'].index(city) ]
print(f"{city}'s latitude is {lat}")
```

```
Manchester's latitude is 53.479
```

Print the location of Lerwick:

4.1.2.1 Answer

```
city = "Lerwick"
print(f"The town of {city} can be found at " +
    f"{abs(myData['Longitude'][myData['Name'].index(city)])}°W, {myData['Latitude'][myData['Name'].index(city)]}°
```

```
The town of Lerwick can be found at 1.145°W, 60.155°N
```

Find the easternmost city:

4.1.2.2 Answer

```
city = myData['Name'][ myData['Longitude'].index( max(myData['Longitude']) ) ]
print(f"The easternmost city is: {city}")
```

```
The easternmost city is: London
```

Find the mean population of the cities using a handy package called numpy:

4.1.2.3 Answer

```
import numpy as np
mean = np.mean(myData['Population'])
print(f"The mean population is: {mean}")
```

```
The mean population is: 2435442.5
```

> **Warning**
>
> **Stop!** Look closely at what is going on. There's a *lot* of content to process in the code above, so do *not* rush blindly on if this is confusing. Try pulling it apart into pieces and then reassemble it. Start with the bits that you understand and then *add* complexity.

We'll go through each one in turn, but they nearly all work in the same way and the really key thing is that you'll notice that we no longer have any loops (which are slow) just index or np.<function> (which is *very* fast).

4.1.3 The Population of Manchester

The code can look pretty daunting, so let's break it down into two parts. What would you get if you ran just this code?

```
myData['Population'][1]
```

```
2705000
```

Remember that this is a dictionary-of-lists (DoL). So, Python first looks for a key named Population in the myData dictionary. It finds out that the value associated with this key is a *list* and in this example, it just pulls out the second value (index 1). Does **that part** make sense?

Now, to the second part:

```
myData['Name'].index('Manchester')
```

```
1
```

Here we look in the dictionary for the key Name and find that that's *also* a list. All we're doing here is asking Python to find the index of 'Manchester' for us in that list. And myData['Name'].index('Manchester') gives us back a 1, so *instead* of just writing myData['Population'][1] we can replace the 1 with myData['Name'].index('Manchester')! Crucially, notice the complete *absence* of a for loop?

Does that make sense? If it does then you should be having a kind of an moment because what we've done by taking a column view, rather than a row view, is to make Python's index() command do the work for us. Instead of having to look through each row for a field that matches 'Name' and then check to see if it's 'Manchester', we've pointed Python at the right column immediately and asked it to find the match (which it can do very quickly). Once we have a match then we *also* have the row number to go and do the lookup in the 'Population' column because the index *is* the row number!

4.1.4 The Easternmost City

Where this approach really comes into its own is on problems that involve maths. To figure out the easternmost city in this list we need to find the *maximum* Longitude and then use *that* value to look up the city name. So let's do the same process of pulling this apart into two steps. Let start with the easier bit:

```
myData['Name'][0]
```

```
'London'
```

That would give us the name of a city, but we don't just want the first city in the list, we want the one with the maximum longitude. To achieve *that* we need to somehow replace the 0 with the ***index of the maximum longitude***. Let's break this down further:

1. We first need to *find* the maximum longitude.
2. We then need to *find* the **index** of that maximum longitude.

So Step 1 would be:

```
max_lon = max(myData['Longitude'])
```

Because the max(...) helps us to find the maximum longitude in the Longitude list. Now that we have that we can proceed to Step 2:

```
myData['Longitude'].index(max_lon)
```

```
0
```

So now we ask Python to find the position of max_lon in the list. But rather than doing this in two steps we can combine into one if we write it down to make it easier to read:

```
myData['Longitude'].index(
   max(myData['Longitude'])
)
```

```
0
```

There's the same .index which tells us that Python is going to look for something in the list associated with the Longitude key. All we've done is change what's *inside* that index function to max(myData['Longitude']). This is telling Python to find the *maximum* value in the myData['Longitude'] list. So to explain this in three steps, what we're doing is:

- Finding the maximum value in the Longitude column (we know there must be one, but we don't know what it is!),
- Finding the index (position) of that maximum value in the Longitude column (now that we know what the value is!),
- Using that index to read a value out of the Name column.

I *am* a geek, but that's pretty cool, right? In one line of code we managed to quickly find out where the data we needed was even though it involved three discrete steps. Think about how much work you'd have to do if you were still thinking in *rows*, not *columns*!

4.1.5 The Location of Lerwick

Lerwick is a small town in the Shetlands, way up to the North of mainland U.K. and somewhere I've wanted to go ever since I got back from Orkney–but then I spent my honeymoon in the far North of Iceland, so perhaps I just don't like being around lots of people…

Anyway, this one *might* be a tiny bit easier conceptually than the other problems, except that I've deliberately used a slightly different way of showing the output that might be confusing:

Print the location of Lerwick:

```
city = "Lerwick"
print(f"The town of {city} can be found at " +
    f"{abs(myData['Longitude'][myData['Name'].index(city)])}°W, {myData['Latitude'][myData['Name'].index(city)]}°
```

```
The town of Lerwick can be found at 1.145°W, 60.155°N
```

The first thing to do is to pull apart the print statement: you can see that this is actually just two 'f-strings' joined by a +–having that at the end of the line tells Python that it should carry on to the next line. That's a handy way to make your code a little easier to read. If you're creating a list and it's getting a little long, then you can also continue a line using a , as well!

4.1.5.1 1. The first f-string

The first string will help you to make sense of the second: f-strings allow you to 'interpolate' a variable into a string directly rather than having to have lots of str(x) + " some text " + str(y). You can write f"{x} some text {y}" and Python will automatically convert the variables x and y to strings and replace {x} with the *value of* x and {y} with the *value of* y.

So here f"The town of {city} can be found at " becomes f"The town of Lerwick can be found at " because {city} is replaced by the value of the variable city. This makes for code that is easier for humans to read and so I'd consider that a good thing.

4.1.5.2 2. The second f-string

This one is hard because there's just a *lot* of code there. But, again, if we start with what we recognise that it gets just a little bit more manageable... Also, it stands to reason that the only difference between the two outputs is that one asks for the 'Longitude' and the other for the 'Latitude'. So if you can make sense of one you have *automatically* made sense of the other and don't need to work it all out.

Let's start with a part that you might recognise:

```
myData['Name'].index(city)
```

```
5
```

You've *got* this. This is just asking Python to work out the index of Lerwick (because `city = 'Lerwick'`). So it's a number. 5 in this case. And we can then think, 'OK so what does this return:

```
myData['Longitude'][5]
```

```
-1.145
```

And the answer is -1.145. That's the Longitude of Lerwick! There's just *one* last thing: notice that we're talking about degrees West here. So the answer isn't a negative (because negative West degrees would be *East*!), it's the *absolute* value. And that is the final piece of the puzzle: `abs(...)` gives us the absolute value of a number!

```
help(abs)
```

```
Help on built-in function abs in module builtins:

abs(x, /)
    Return the absolute value of the argument.
```

4.1.6 The Average City Size

Here we're going to 'cheat' a little bit: rather than writing our own function, we're going to import a package and use someone *else's* function. The `numpy` package contains a *lot* of useful functions that we can call on (if you don't believe me, add "`dir(np)`" on a new line after the `import` statement), and one of them calculates the average of a list or array of data.

```
print(f"The mean population is {np.mean(myData['Population'])}")
```

```
The mean population is 2435442.5
```

This is where our new approach really comes into its own: because all of the population data is in one place (a.k.a. a *series* or column), we can just throw the whole list into the `np.mean` function rather than having to use all of those convoluted loops and counters. Simples, right?

No, not *simple* at all, but we've come up with a way to *make* it simple.

4.1.7 Recap!

So the *really* clever bit in all of this isn't switching from a list-of-lists to a dictionary-of-lists, it's recognising that the dictionary-of-lists is a *better* way to work *with* the data that we're trying to analyse and that that there are useful functions that we can exploit to do the heavy lifting for us. Simply by changing the way that we stored the data in a 'data structure' (i.e. complex arrangement of lists, dictionaries, and variables) we were able to do away with lots of for loops and counters and conditions, and reduce many difficult operations to something that could be done on one line!

4.2 Brain Teaser

Difficulty: .

Why not have a stab at writing the code to print out the *4th most populous* city? This can *still* be done on one line, though you might want to start by breaking the problem down:

1. How do I find the *4th* largest value in a list?
2. How do I find the *index* of the 4th largest value in a list?
3. How do I use that to look up the name associated with that index?

You've already done #2 and #3 above so you've *solved* that problem. If you can solve #1 then the rest should fall into place.

Tip

You don't want to use <list>.sort() because that will sort your data *in place* and break the link between the indexes across the 'columns'; you want to research the function sorted(<list>) where <list> is the variable that holds your data and sorted(...) just returns whatever you pass it in a sorted order *without* changing the original list. You'll see why this matters if you get the answer... otherwise, wait a few days for the answers to post.

4.2.0.1 Answer

```
# Print out the name of the 4th most populous city-region
city = myData['Name'][
   myData['Population'].index(sorted(myData['Population'], reverse=True)[3])
]

print("The fourth most populous city is: " + str(city))
```

```
The fourth most populous city is: Edinburgh
```

The answer is Edinburgh.

4.3 Bringing it all together...

Conceptually, this is one of the hardest practicals in the entire term because it joins up so many of the seemingly simple ideas that you covered in Code Camp into a very complex 'stew' – all our basic ingredients (lists, dictionaries, etc.) have simmered for a bit, been stirred up together, and become something entirely new and more complex.

So if this practical doesn't make sense to you on the *first* runthrough, I'd suggest going back through the second half of the practical *again* in a couple of days' time – that will give your brain a little time to wrap itself around the basics before you throw the hard stuff at it again. *Don't* panic if it doesn't all make sense on the *second* runthrough either – this is like a language, you need to practice! With luck, the second time you went through this code a little bit *more* made sense. If you need to do it a third time you'll find that even *more* makes sense… and so on.