Practical 4: Efficient Code

Packages, Decorators, and Functions

In this notebook we are going to look in more detail at how we can reduce, reuse, and recycle our code to make our lives easier and our code more efficient.

1. 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!

1.1 What Do We Do? Break It Down!

1.1.1 Step 1. Analyse the Problem

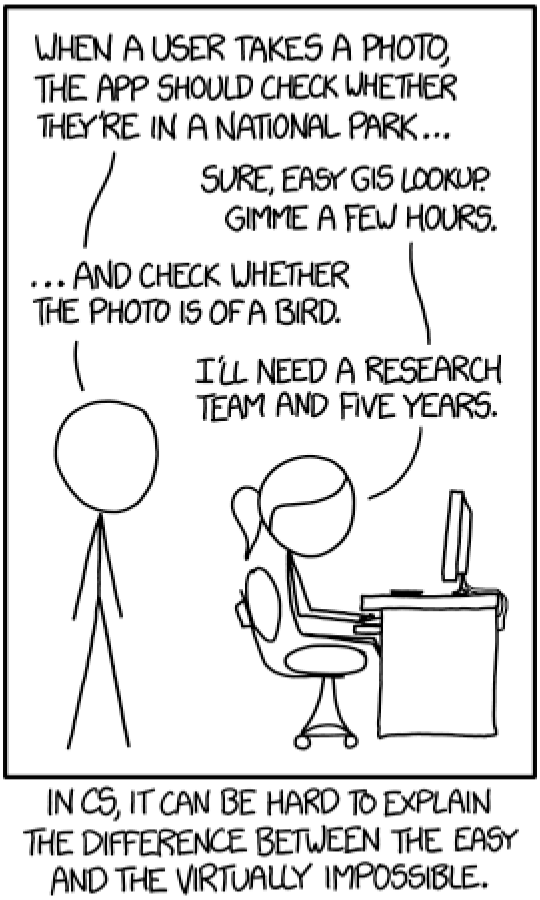
The first step to writing a program is thinking about your goal and the steps required to achieve that. We ***don't*** write programs like we write essays: all at once by writing a whole lot of code and then hoping for the best when we hit 'submit'.

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*.

1.1.2 Step 2. Functions & Packages

We have discussed how *functions* are a useful programming tool to enable us to re-use 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.



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 `1` s and `0` s 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.

1.1.3 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!

1.1.4 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.

2. 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.

In [1]:
```
from urllib.request import urlopen
help(urlopen)
```

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

urlopen(url, data=None, timeout=<object object at 0x72c57d3108b0>, *, cafile=Non
e, capath=None, cadefault=False, context=None)
    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.

In [33]:
```python
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://orca.casa.ucl.ac.uk/~jreades/data/Listings.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]}...")
```

```
'raw' variable is of type: 'bytes'.
Raw content is:
b'name,host_name,latitude,longitude,price\nEntire 3 Room Studio Apartment,Ker
r'...

'data' variable is of type: 'str'.
Decoded content is:
name,host_name,latitude,longitude,price
Entire 3 Room Studio Apartment,Kerr...
```

> **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.

In [34]:
```python
print(f"There are {len(data)} characters in the data variable.")
print(f"The first 125 characters are: '{data[:125]}'") # Notice that '\n' count
```

```
There are 746 characters in the data variable.
The first 125 characters are: 'name,host_name,latitude,longitude,price
Entire 3 Room Studio Apartment,Kerrie,51.45634,-0.3021,
2bed duplex w/roof terrace 11'
```

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):

2.0.0.1 Question

In [35]:
```python
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):

2.0.0.2 Question

In [36]:
```python
print(f"There are {len(rows)} rows of data.")
print("\n".join(rows[0:2])) # New syntax alert! notice we can *join* list elemen
```

```
There are 11 rows of data.
name,host_name,latitude,longitude,price
Entire 3 Room Studio Apartment,Kerrie,51.45634,-0.3021,
```

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

In [38]:
```
print(rows[-1])
```

```
Lansbury Heritage A Sunday Hotel Lansbury Classic,Abhishek,51.50975734746474,-0.0
150004514905809,$188.00
```

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

2.1 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` :

In [39]:
```
print('abcdefgh'.split('c'))
```

```
['ab', 'defgh']
```

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

In [40]:
```
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 str
ing.

      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*:

In [41]:
```
test = rows[-1].split(',')
print(test)
```

```
print(f"The price of {test[0]} is {test[-1]}")
```

```
['Lansbury Heritage A Sunday Hotel Lansbury Classic', 'Abhishek', '51.50975734746
474', '-0.0150004514905809', '$188.00']
The price of Lansbury Heritage A Sunday Hotel Lansbury Classic is $188.00
```

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!

3. The Advantages of a Package

> **Caution**
>
> There are two problems to the `data.splitlines()` and `row.split(',')` approach! One of them *can* be seen in the examples above, the other cannot.

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.

In [42]:
```
from urllib.request import urlopen
import csv

url = 'https://orca.casa.ucl.ac.uk/~jreades/data/Listings.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])) +
print(urlData[-1]) # Check it worked!

print(urlData)
```

```
urlData has 11 rows and 5 columns.
['Lansbury Heritage A Sunday Hotel Lansbury Classic', 'Abhishek', '51.50975734746
474', '-0.0150004514905809', '$188.00']
[['name', 'host_name', 'latitude', 'longitude', 'price'], ['Entire 3 Room Studio
Apartment', 'Kerrie', '51.45634', '-0.3021', ''], ['2bed duplex w/roof terrace 11
min to Islington', 'Michael', '51.55718', '-0.08944', '$120.00'], ['Cosy UK Fla
t', 'Garbhán', '51.53950816898676', '-0.0754128078813238', ''], ['Lovely flat in
Brixton/Herne Hill', 'Nick', '51.46288', '-0.09996', ''], ['Modern   &   Bright
Apt', 'Monica', '51.59279', '0.04634', '$50.00'], ['Double Room in Haggerston',
'Matt', '51.53607', '-0.0711', ''], ['Loft space in Notting Hill', 'Chiara', '51.
51756', '-0.21569', ''], ['Excel Docks 2 Bedroom/2 Bathrooms', 'Kehinde Raheem',
'51.507523071215786', '0.0219408247281377', '$225.00'], ['Contemporary Apartment
in Bloomsbury/Chancery Lane', 'Sam & Kerri', '51.5217665', '-0.1141964', '$514.0
0'], ['Lansbury Heritage A Sunday Hotel Lansbury Classic', 'Abhishek', '51.509757
34746474', '-0.0150004514905809', '$188.00']]
```

If it worked, then you should have this output:

```
default
urlData has 11 rows and 5 columns.
['Lansbury Heritage A Sunday Hotel Lansbury Classic',
'Abhishek', '51.50975734746474', '-0.0150004514905809',
'$188.00']
```

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:

In [43]:
```
for u in urlData[1:4]:
    print(f"The price of '{u[0]}' is '{u[-1]}'")
```

```
The price of 'Entire 3 Room Studio Apartment' is ''
The price of '2bed duplex w/roof terrace 11 min to Islington' is '$120.00'
The price of 'Cosy UK Flat' is ''
```

3.0.0.1 Question

Why did I use `urlData[1:]` instead of `urlData`?

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 what happens!

In [44]:
```
from urllib.request import urlopen
import csv

url = "https://orca.casa.ucl.ac.uk/~jreades/data/Listings-lg.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(urlData[0])

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

for u in urlData[:20]:  # For each row in the list
    print(f"The listing for '{u[0]}' has a price of {u[-1]}")
```

```
['name', 'property_type', 'room_type', 'price', 'bedrooms', 'beds', 'host_name',
'latitude', 'longitude']
urlData has 1001 rows and 9 columns.
The listing for 'name' has a price of longitude
The listing for 'Gorgeous 2 bed flat w easy access to Earlsfield St' has a price
of -0.18903
The listing for 'Welcome to London!' has a price of -0.0934109485544038
The listing for '2 bedroom 8th floor serviced apartment.' has a price of -0.02224
The listing for 'Prime city studio apartment' has a price of -0.0783275711639452
The listing for 'Cozy Room near Canary Wharf' has a price of -0.0299
The listing for 'Paddington Apartments by DC London Rooms B3' has a price of -0.1
8276
The listing for 'Stylish 3BR house Dalston/Hackney' has a price of -0.06201
The listing for '3 People - St John's Wood - Zone 2' has a price of -0.18575
The listing for 'Private room 3 mins walk from Abbey Wood station' has a price of
0.11857
The listing for 'Quiet | 7-mins walk to Paddington | Women Only' has a price of -
0.18016
The listing for 'Spacious Modern New Listing' has a price of -0.2123317
The listing for 'Central London Stylish flat Baker Street' has a price of -0.1622
569
The listing for 'free parking,Top location, Shops' has a price of -0.278033717849
8748
The listing for 'Cosy 1 Bedroom Modern Flat in New Cross, London' has a price of
-0.04877
The listing for 'Creative room in heart of London' has a price of -0.04357
The listing for 'Spacious,  clean bedroom in warm & friendly home' has a price of
0.21817
The listing for 'New Build Haven with en-suite by Colindale Station' has a price
of -0.2467760137180713
The listing for 'Tranquil Lodgings: Spacious DBR, Parking & WI-FI' has a price of
0.19661
The listing for 'Bright and spacious maisonette with terrace' has a price of -0.1
168853032453689
```

> **What mistake have I made here?**
>
> I have assumed that, just because the files have similar names, they must also have similar layouts!
>
> The URL's data labels are:
>
> ```
> default
> name, property_type, room_type, price, bedrooms, beds,
> host_name, latitude, longitude
> ```

3.1 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'

> 🔗 **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...

4.1 Understanding What's an 'Appropriate' Data Structure

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 or floats), 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.

4.2 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 the CSV file:

```
default
name, property_type, room_type, price, bedrooms, beds,
host_name, latitude, longitude
Gorgeous 2 bed flat w easy access to Earlsfield St, Entire
serviced apartment, Entire home/apt, , 2.0, , Katie, 51.44243,
-0.18903
Welcome to London!, Private room in bed and breakfast, Private
room, $54.00, 1.0, 1.0, Sabahat, 51.59339662817043,
-0.0934109485544038
2 bedroom 8th floor serviced apartment., Entire serviced
apartment, Entire home/apt, $187.00, 2.0, 2.0, Cubo Apartments
Canary Wharf, 51.49926, -0.02224
Prime city studio apartment, Entire rental unit, Entire
home/apt, $82.00, 1.0, 1.0, Alan & Wanlin, 51.52548766243926,
-0.0783275711639452
Cozy Room near Canary Wharf, Private room in home, Private room,
, , , Sujata, 51.51468, -0.0299
```

Here's how it would look as a dictionary of lists organised by *column*, and *not* by row, though note that (for now) I've changed the `NaN` (Not a Number) values to something that will be easier to work with:

In [45]:
```python
myData = {
    'id'         : [0, 1, 2, 3, 4],
    'Name'       : ['Gorgeous 3 bed flat w easy access to Earlsfiel...', 'Welcom
    'Longitude'  : [-0.189030, -0.093411, -0.022240, -0.078328, -0.029900],
    'Latitude'   : [51.442430, 51.593397, 51.499260, 51.525488, 51.514680],
    'Bedrooms'   : [3, 4, 2, 0, 1],
}
```

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

```
['Gorgeous 3 bed flat w easy access to Earlsfiel...', 'Welcome to London!', '2 bedroom 8th floor serviced apartment.', 'Prime city studio apartment', 'Cozy Room near Canary Wharf']
[3, 4, 2, 0, 1]
```

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...

4.3 Behold the Power of the DoL

Now let's look at what you can do with this data structure. I'd encourage you to try to answer the questions without tips or help, but because this is quite a big shift in complexity, for each of the questions there's also substantial guidance in the 'Unpacking' section.

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

We can find the latitude of 'Prime city studio apartment' by putting together what we know about searching for a value in a list with what we know about dictionaries:

In [46]:
```
loc = 'Prime city studio apartment'
lat = myData['Latitude'][ myData['Name'].index(loc) ]
print(f"{loc}'s latitude is {lat}")
```

```
Prime city studio apartment's latitude is 51.525488
```

So to print the location of '2 bedroom 8th floor serviced apartment.':

4.3.0.1 Question

In [59]:
```
loc = "2 bedroom 8th floor serviced apartment."
print(f"The listing for {loc} can be found at " +
      f"{abs(myData['Longitude'][myData['Name'].index(loc)])}ºW, {myData['Latitu
```

```
The listing for 2 bedroom 8th floor serviced apartment. can be found at 0.02224ºW, 51.49926ºN
```

If you need help, then you can find it below.

To find the easternmost listing we need to adapt what we've done above and think about how we'd get the **max**imum value out:

4.3.0.2 Question

In [71]:
```
listing = myData['Name'][ myData['Longitude'].index(max(myData['Longitude']) ) ]
print(f"The easternmost listing is: {listing}")
```

```
The easternmost listing is: 2 bedroom 8th floor serviced apartment.
```

If you need help, then you can find it below with some additional tips on just finding a longitude that really breaks it all down into small steps.

To find the `mean` number of bedrooms we can use a package called numpy. `numpy` (Numerical Python) 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 simple arithmatic `mean`.

4.3.0.3 Question

In [62]:
```
import numpy as np
mean = np.mean(myData['Bedrooms'])
print(f"The mean number of bedrooms is: {mean}")
```

```
The mean number of bedrooms is: 2.0
```

Again, you can find help below.

> **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.

5. Unpacking a DoL

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).

5.1 The Location of a Listing

Let's have a look at how we might print a listing's lat/long:

In [63]:
```
loc = 'Cozy Room near Canary Wharf'
print(f"The listing of {loc} can be found at " +
    f"{abs(myData['Longitude'][myData['Name'].index(loc)])}ºW, " +
    f"{myData['Latitude'][myData['Name'].index(loc)]}ºN")
```

```
The listing of Cozy Room near Canary Wharf can be found at 0.0299ºW, 51.51468ºN
```

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 reading code 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!

5.1.1 1. The first f-string

The first string will help you to make sense of the second: recall that 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)`. Instead, 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 listing of {loc} can be found at "` becomes `f"The listing of Cozy Room near Canary Wharf can be found at "` because `{loc}` is replaced by the value of the variable `loc`. This makes for code that is easier for humans to read and so I'd consider that a good thing.

5.1.2 2. The second and third f-strings

These are harder 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:

In [64]:
```
myData['Name'].index(loc)
```

Out[64]: 4

You've *got* this. This is just asking Python to work out the index of 'Cozy Room near Canary Wharf' (because `loc = 'Cozy Room near Canary Wharf'`). So it's a number. And we can then think, 'OK so what does this return:

In [65]:
```
myData['Longitude'][4]
```

Out[65]: -0.0299

And the answer is `{python} o3`. That's the Longitude of the listing! 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!

In [66]:
```
help(abs)
```

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

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

5.2 The Longitude of a Listing

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

In [67]:
```
myData['Longitude'][4]
```

Out[67]: -0.0299

Remember that this is a dictionary-of-lists (DoL). So, Python first looks for a key named `Longitude` 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:

In [68]:
```
myData['Name'].index('Cozy Room near Canary Wharf')
```

Out[68]: 4

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 'Cozy Room near Canary Wharf' for us in that list. And `myData['Name'].index('Cozy Room near Canary Wharf')` gives us back a `4`, so *instead* of just writing `myData['Longitude'][4]` we can replace the `4` with `myData['Name'].index('Cozy Room near Canary Wharf')`! Crucially, notice the complete *absence* of a for loop, which means that this code is *fast*!

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 '2 bedroom 8th floor serviced apartment.', 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 'Longitude' column because the index *is* the row number!

5.3 The Easternmost Listing

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:

In [69]:
```
myData['Name'][0]
```

Out[69]: 'Gorgeous 3 bed flat w easy access to Earlsfiel...'

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:

In [72]:
```
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:

In [73]:
```
myData['Longitude'].index(
    max_lon
)
```

Out[73]: 2

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:

In [74]:
```
myData['Longitude'].index(
    max(myData['Longitude'])
)
```

Out[74]: 2

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*!

In [75]:
```
loc = myData['Name'][
    myData['Longitude'].index(
        max(myData['Longitude'])
    )
]
print(f"The easternmost listing is {loc}.")
```

```
The easternmost listing is 2 bedroom 8th floor serviced apartment..
```

5.3.1 The Average Number of Bedrooms

So 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.

In [40]:
```
import numpy as np
print(f"The mean number of bedrooms is {np.mean(myData['Bedrooms'])}")
```

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.

5.3.2 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!

6. Brain Teaser

> **Difficulty: 🤯.**

Why not have a stab at writing the code to print out the listing with the *3rd most bedrooms*? 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 *3rd* largest value in a list?
2. How do I find the *index* of the 3rd 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.

6.0.0.1 Question

myData = { 'id' : [0, 1, 2, 3, 4], 'Name' : ['Gorgeous 3 bed flat w easy access to Earlsfiel...', 'Welcome to London!', '2 bedroom 8th floor serviced apartment.','Prime city studio apartment','Cozy Room near Canary Wharf'], 'Longitude' : [-0.189030, -0.093411, -0.022240, -0.078328, -0.029900], 'Latitude' : [51.442430, 51.593397, 51.499260, 51.525488, 51.514680], 'Bedrooms' : [3, 4, 2, 0, 1], }

Print out the name of the 3rd most bedrooms

sort = myData["Bedrooms"].sort(reverse=True) loc = (myData["Name"][myData["Bedrooms"].index[2]])

print(myData["Bedrooms"]) print("The third most bedrooms are in: " + str(loc))

The answer is `2 bedroom 8th floor serviced apartment.`.

6.1 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.