Introduction to Python Programming

Toby Hodges, Holger Dinkel, Karin Sasaki, Marc Gouw
Malvika Sharan, Peter D Ashton

Contents

Chapter 1

Getting Started

Running Python Python itself normally exists just to run programs that you have written, but it can run as a program in its own right, allowing you to explore the language by typing commands which it then executes for you. Quite a lot can be done at this command prompt, but to do anything serious you will need to start creating your programs in a text editor and executing them with Python. However, we will start by looking at what you can do at the Python prompt.

When you install Anaconda, it creates a program group containing a few different items. Included in this an interactive python console, 'jupyter qtconsole', which you should launch and use as your Python Shell (don't worry if you don't know what a shell is at this stage!). A second item, 'spyder', is a powerful text editor with the capacity to run your Python scripts in a dedicated shell window. You should launch this and use it when you start writing multi-line code that you will want to go back and edit/append as you go through the course.

Below is an example of the IPython QtConsole window. Yours might look slightly different from this, depending on the operating system that you are working in, but the functionality of the program itself remains (almost) exactly the same.

If you are running Linux or Mac OS X, start a terminal, type `ipython` and press return to get the same program within the terminal window. (To leave the shell, type `exit()` and press enter.)

For this first worksheet, you can choose either to run the commands in the Python Shell or, if you are viewing this notebook interactively, you can execute your commands directly within the notebook code cells. We have inserted cells especially for you to play around with the code in. If you use these cells, you can execute the code within them with `CTRL+ENTER` and the output will be printed beneath the cell. Don't worry if you can't use this notebook interactively - you should just type all of the commands in the shell instead.

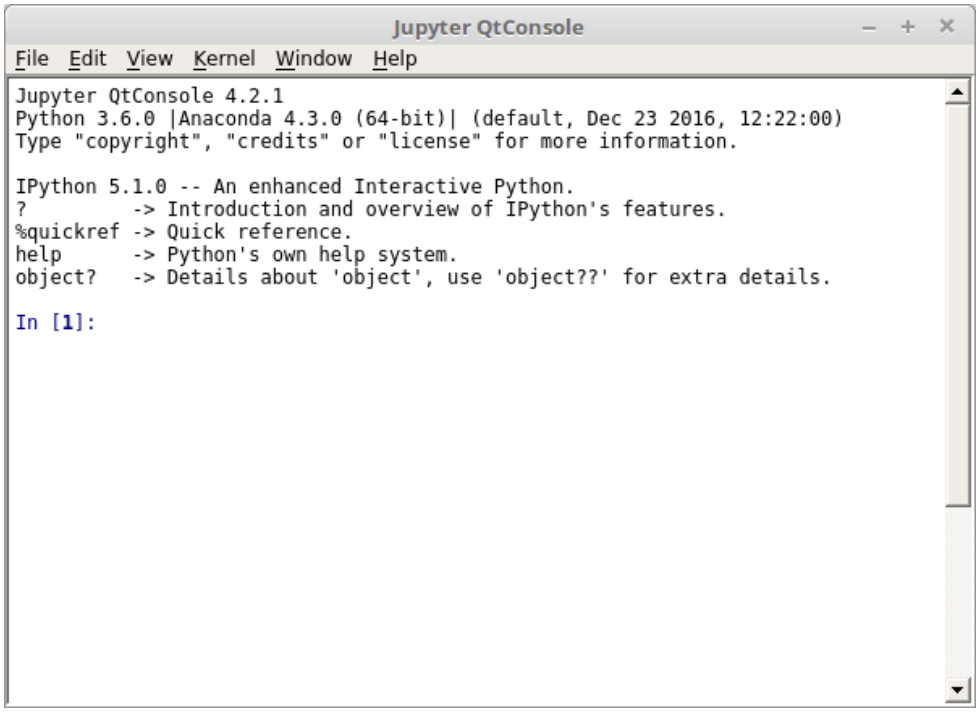
In the window image above, the `In[1]` is the Python prompt and it should have a flashing cursor. As is traditional under these circumstances, the first thing you should do is get Python to print out the text "Hello, World". To do this, type the command below at the prompt and press return (control+return in the notebook).

```
In [1]: print('Hello, World')
```

```
Hello, World
```

```
In [2]: print('hello, world')
```

```
hello, world
```



An example of the Jupyter-Qtshell window

Barring the odd typing mistake you have just run your first Python script. Here, 'print' is a Python function that tells the interpreter that you want it to output the things which follow in the brackets. The 'Hello, World' is a *string* and is what you want the `print` function to print out.

Playing with Python The Python Shell is a great way to experiment with Python. It's something that I return to every time I find myself thinking "I wonder what happens if you do this?" or "What's the best way to do that?". You can't break anything by trying something out and if you do make a mistake, at least you will get an error message that you might be able to decode.

If you are viewing this notebook interactively (i.e. not through nbviewer), the code cells behave in a very similar way to the Shell prompt. You can play around with Python in the notebook's code cells (we have left some blank ones for you to try out your own commands in) and execute the cell once you are done typing. The output of your commands will appear underneath. If you have made a mistake, or you want to try something else, you can edit the code in the cell and execute it again when you're done.

You are not limited in what you can do at the prompt. You can load modules, look at them to see what they do, play with them. The first versions of the bar charts in Worksheet 3 were all produced at the python prompt, which enabled me to tweak them before writing the program to produce the whole figure. This meant I could see how they looked after every command and get them looking just as I wanted. I could also check the documentation for the modules I wanted to use by typing `help()`. There is a lot of information in there and you will find yourself using it again and again.

In addition to these features of the Python Shell, the IPython Shell has some very helpful features that can make things easier while you learn the basics of Python. For example, IPython enables tab-completion for e.g. variable, function, and file names, gives hints about required &

optional arguments for functions, and has improved command history interaction.

For longer, multi-line programs you will probably find it easier to use a text editor, and we will cover that in Worksheet 2.

Evaluating Expressions The Python shell can also be used to evaluate expressions, allowing you either to perform calculations interactively, or more usually to check more complicated expressions interactively before putting them in your programs. The Python shell allows you to do all of the normal operations, in pretty much the way you would expect.

Have a go with some expressions, such as:

```
In [3]: 3 * 4      # Multiplication

Out[3]: 12

In [4]: # type your command(s) here or use the IPython shell...

In [5]: 7 + 10     # Addition

Out[5]: 17

In [6]: # type your command(s) here or use the IPython shell...

In [7]: 7 - 10     # Subtraction

Out[7]: -3

In [8]: # type your command(s) here or use the IPython shell...

In [9]: 10 / 3     # Division

Out[9]: 3.3333333333333335

In [10]: # type your command(s) here or use the IPython shell...
```

So far, so good, but don't limit yourself to the examples here. Try some of your own and make sure you understand the results. There are a few other operators, though, which you might not be as familiar with. Try these:

```
In [11]: 10 % 3      # Modulus (remainder)

Out[11]: 1

In [12]: # type your command(s) here or use the IPython shell...

In [13]: 2 ** 10     # Exponentiation (2 to the power of 10)

Out[13]: 1024

In [14]: # type your command(s) here or use the IPython shell...
```

Again, as expected. However, there are a few things that you need to be aware of when using arithmetic in any programming language. In Python v2.x, if your numbers are integers, Python with return an *integer* value. So try:

```
In [15]: 10 / 7

Out[15]: 1.4285714285714286

In [16]: # type your command(s) here or use the IPython shell...
```

Here you should get the answer 1.4285714285714286 if you're using Python v3.x, or 1 if you're using Python 2.x. This is because Python 2.x returns an integer value if division is done with two integers, which in this case is the result of the division ignoring any decimal values (*i.e.* "floor division"). This has been changed in Python 3.x., and 10/7 now gives the result you most likely expect. Of course, you might actually only want the integer result in the first place, and regardless of the version you can force Python to give you that as well using so-called "floor division":

```
In [17]: 10.0 // 7

Out[17]: 1.0

In [18]: # type your command(s) here or use the IPython shell...
```

Python can handle some very large numbers. For example, it can easily deal with raising 2 to the power of 32:

```
In [19]: 2 ** 32

Out[19]: 4294967296
```

Python can deal with numbers slightly larger than this too, so

```
In [20]: 2 ** 64

Out[20]: 18446744073709551616
```

and even

```
In [21]: 2 ** 1024

Out[21]: 179769313486231590772930519078902473361797697894230657273430081157732675805
```

work just fine. You can go even higher, so raising to the power of 100,000

```
In [22]: 2 ** 100000

Out[22]: 99900209301438450794403276433003359098042913905418169177152927386314583246
```

is quite OK. Though I must admit that I haven't actually checked that the 30103 digits of this result are correct.

Some of these operators don't just work on numbers, + and * can be used on strings as well. Strings are just sequences of characters enclosed in quotation marks like the 'Hello, World' above. Python doesn't mind if you use single or double quotes as long as you don't mix them. "Addition", +, concatenates two (or more) strings together to return a new longer string. "Multiplication", actually repetition, *, takes a number and a string and repeats the string that many times in a new string:

```
In [23]: 'Hello, ' + "world!"

Out[23]: 'Hello, world!'

In [24]: # type your command(s) here or use the IPython shell...

In [25]: 'Hello ' * 8

Out[25]: 'Hello Hello Hello Hello Hello Hello Hello Hello '

In [26]: # type your command(s) here or use the IPython shell...

In [27]: 9 * 'Hello...'

Out[27]: 'Hello...Hello...Hello...Hello...Hello...Hello...Hello...Hello...Hello...'

In [28]: # type your command(s) here or use the IPython shell...
```

Exercise 1.1 Try to use expressions that you would use in your normal work and see if they give the results you expect. Explore using brackets to group sub-expressions (things in brackets are always evaluated before everything else). Before you move on to the next section, which of the following expressions would correctly calculate the hypotenuse of a right-angled triangle, with sides length 12 and 5?

a) `(12*2 + 5*2)/2`

b) `(12**2 + 5**2)**0.5`

c) `(12^2 + 5^2)^0.5`

```
In [29]: # type your command(s) here or use the IPython shell...
```

Using Variables So far, we have just been playing with what Python calls values. When you are writing programs, it's useful to be able to give names to the values that we are dealing with so that once we do a calculation or string manipulation we can refer to the results later. We do this with an assignment statement, which looks like this:

```
In [30]: x = 3
```

You'll notice that, when this line is executed, Python doesn't return anything. This is also true if you capture the result of one of the expressions that we tried above:

```
In [31]: y = 10.0 / 7
```

To look at the values, just type the names of the variables that you have created:

```
In [32]: x

Out[32]: 3

In [33]: y

Out[33]: 1.4285714285714286
```

More normally, you would probably output the results using the `print` statement we started with:

```
In [34]: print(x, y)

3 1.4285714285714286
```

If you are using Python 2.x you would use `print x, y` (without brackets), another one of the major differences between Python 2.x and Python 3.x.

If the brackets are included in the output above, then you're working in version 2. If not, then you have a version 3 environment. (Whichever version you have, you will be able to work through the rest of this course.)

As well as being on the left of an assignment operation, variable names can be used in the expressions as well, so

```
In [35]: x = x + y
```

replaces the value currently referred to by `x` with the new value obtained from adding the values of `x` and `y`.

```
In [36]: x

Out[36]: 4.428571428571429
```

Variables that refer to numbers are fine and are incredibly useful, but they are also one of the less interesting types of Python data. This is because they only have a value. Some variables are much more interesting, and string variables are a good example. To assign a string to a variable you follow the same basic syntax as before:

```
In [37]: s = 'The quick brown fox jumps over the lazy dog'

In [38]: print(s)

The quick brown fox jumps over the lazy dog
```

Again, you can just type the variable name or use it in a `print` statement, but what makes a variable containing a string more interesting is that it is one of Python's object data types. This means that it doesn't just passively hold the value of the string, it also knows about things you can do to the string. For example:

```
In [39]: s.upper()

Out[39]: 'THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG'
```

The `.` here indicates that the method is part of the string `s`, and the brackets indicate that we want to execute it. Here the brackets are empty, but later we will be using them to pass information into the methods. There are many things that strings can do with themselves, and if you look at the Python cheat sheet, you will see what they all are. Try using them on the string, for example:

```
In [40]: s.capitalize()

Out[40]: 'The quick brown fox jumps over the lazy dog'

In [41]: s.title()

Out[41]: 'The Quick Brown Fox Jumps Over The Lazy Dog'
```

If you look at `s` itself after any of these, you'll see it hasn't changed, these object methods simply return a new version of the string with the appropriate transformation done to it which you can then store in another variable (or back in `s`) if you want to. This is because a string cannot be changed in place (in technical terms, it is *immutable*), only explicitly overwritten with a new value. So, to save the new version of the string, you can type the following:

```
In [42]: s = s.title()
```

You can use almost any combination of letters and numbers in the variable names, but you can't start a variable name with a number. You can't include spaces (a space is one of the ways that Python can tell that the name is finished) but you can include underscore characters. Variable names can also begin with underscore, but these tend to be used under special circumstance which you will discover once you start learning about object-oriented programming.

Exercise 1.2 Use the .count() method on the string "The quick brown fox jumps over the lazy dog" to count the occurrences of the word "the".

```
In [43]: s = "The quick brown fox jumps over the lazy dog"
         # type your command(s) here or use the IPython shell...
```

If this returns 1, how could you persuade it that it should be 2?

```
In [44]: # type your command(s) here or use the IPython shell...
```

Once you have that sorted out, try
`s.split()`

```
In [45]: # type your command(s) here or use the IPython shell...
```

and see if you can understand what it has done.

Lists The last exercise returned a result which you might think looks unusual. My interpreter gave me:

```
In [46]: s = "The quick brown fox jumps over the lazy dog"
         s.split()

Out[46]: ['The', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy', 'dog']
```

This result is in the form of a *list* object. A list is exactly what you might expect based on the name. It's a set of values (which can be numbers, strings, objects or even lists, but that is a bit advanced for now) that are kept in a specific order. You can create a new list using the same format as the result above:

```
In [47]: shopping = ['bread', 'potatoes', 'eggs', 'flour', 'rubber duck', 'pizza',
```

Like strings, lists are a type of object, and so they also have some methods associated with them, which you can use. However, unlike strings, these methods mostly change the list in place, rather than returning a new list. So, when you type

```
In [48]: shopping.sort()
```

Python doesn't return a value, but if you look at the list, you will see that the order of the items has changed.

```
In [49]: shopping

Out[49]: ['bread', 'eggs', 'flour', 'milk', 'pizza', 'potatoes', 'rubber duck']
```

This means that unlike strings, lists are *mutable*, and individual items and sets of items can be changed in place. If you decide you want to add items to the list, you can do it with the `.append()` method:

```
In [50]: shopping.append('mayonnaise')
         shopping

Out[50]: ['bread',
          'eggs',
          'flour',
          'milk',
          'pizza',
          'potatoes',
          'rubber duck',
          'mayonnaise']
```

If you feel it's important enough to go at the top of the list, you can use `.insert()` to insert the new item at a particular point and shuffle everything else up:

```
In [51]: shopping.insert(0, 'mayonnaise')
         shopping
```

```
Out[51]: ['mayonnaise',
          'bread',
          'eggs',
          'flour',
          'milk',
          'pizza',
          'potatoes',
          'rubber duck',
          'mayonnaise']
```

Removing items from the list is just as easy - you use `.pop()` to do that. If you don't give it an index, it will remove the last item in the list, otherwise `pop` removes the item with the index you specify and shuffles everything else up one position to close the gap. Give it a try, to remove one of the "mayonnaise" items in the shopping list.

```
In [52]: # type your command(s) here or use the IPython shell...
```

Sequences The two object types that we have talked about so far share a number of properties. Both strings and lists consist of ordered pieces of data. In the case of strings this is simple characters. In lists, the elements of the list can be of any object type.

For both lists and strings we might want to refer to a particular item or range of items in a string or list and we can do this easily:

```
In [53]: words = s.split()
         print(words[3])

fox
```

You will see that "fox" is actually the fourth word and this is just one of the things that computers do that you have to get used to. The first element in a sequence has an index of 0, so

```
In [54]: print(words[0])

The
```

gives you the first item in the list. This is referred to as zero-based indexing or offset numbering. Its origins are in programming languages where variables actually refer to the memory location of the start of the list, and now has just become a tradition. Negative indices are assumed to be relative to the end of the array, so

```
In [55]: words[-1]

Out[55]: 'dog'
```

yields the final element in the list, in this case "dog". Of course, if we knew how long the sequence was, we could just use the number of the last element. For any sequence data type, `len()` will tell us how many elements it has:

```
In [56]: len(words)

Out[56]: 9

In [57]: len(s)

Out[57]: 43
```

So, since the sequences are indexed from zero, the last element, i.e. `words[-1]`, is the same as `words[len(words)-1]`.

Exercise 1.3 There are a few quirks to sequence indexing (apart from starting at 0), and I have tried to summarise these on the Python Cheat Sheet. Have a go with a few of the "Indices and Slices" and make sure you understand how they work. Then, instead of trying them on a list, try them on a string. Now, if you are really adventurous, try to print the third letter of the fourth word in the `words` list.

```
In [58]: # type your command(s) here or use the IPython shell...
```

Exercise 1.4 Below is a set of commands working with a list of cities. First, the list is extended by adding 'Athens' onto the end, and then 'Nairobi' - the third element in the list - is assigned to the variable `kenya`. Some of the code has been removed (replaced with `---`). Fill in the blanks in the code block to make it work.

```
In [59]: cities = ['Winnipeg', 'Guangzhou', 'Nairobi', 'Santiago', 'Glasgow']
         cities.---('Athens') # add 'Athens' onto the end of the list
         kenya = cities[---] # access the third entry in the cities list


          File "<ipython-input-59-1208d5feaf7b>", line 2
        cities.---('Athens') # add 'Athens' onto the end of the list
               ^
    SyntaxError: invalid syntax
```

Summary

- Python can be used to calculate the results of a wide range of expressions, and print out the results.
- Like in all programming languages, variables can be used to store the results of calculations or simple values that we might need to use later.
- Variables refer to values. The values can be numbers (integer and floating point), strings or lists. You will discover soon that there are other data types as well.
- Some data types are objects, which have methods associated with them. These methods perform common tasks on the values of the objects.
- Some data types are sequences, which let us access individual elements at will.
- Sequence data types allow us to step through the values in them, but to do that we need to take some first steps towards writing real programs.

Debugging Exercise The lines below have mistakes in them, which produce errors when they are run. Try to identify the problems in the code, and fix them so that they run without error.

```
In [60]: cake flavours = ['chocolate', 'coffee', 'carrot', 'vanilla']
         cake_flavours.append('lemon')
           print(cake_flavours)


          File "<ipython-input-60-650789e7bb95>", line 1
        cake flavours = ['chocolate', 'coffee', 'carrot', 'vanilla']
                      ^
    SyntaxError: invalid syntax
```

Chapter 2

Beginning Programming

First Steps in Programming So far, we've had fun playing with commands at the Python Shell prompt, but now we are going to need to start editing programs properly and saving them so that we can change them and re-use parts later. So, now start the Spyder program (or another text editor of your choice), and open a new file to start writing your code into. There is no prompt like in the Python Shell window, just a space for you to edit you first program. When you finish a line and press enter here, nothing will be executed. Instead, you will need to save and run your script each time you want to execute any changes that you've made. In Spyder, this is easy, as the interface includes a small Python Shell window dedicated to the output of the code that you write in the editor.

Using an editor instead of the shell allows you to quickly go back and change code that you've already written, which can make it easier to correct typos, add additional lines, and 'debug' your script to help figure out where an error or unwanted behaviour is occurring. Although you can use the command history at the shell prompt to access your previous lines of code, it is often easier to keep your scripting separate from the output. Later, we will see an example of where using an editor is really useful.

Start by entering the following code:

```
In [61]: shopping = ['bread', 'potatoes', 'eggs', 'flour', 'rubber duck', 'pizza',
         for item in shopping:
             print(item)

bread
potatoes
eggs
flour
rubber duck
pizza
milk
```

This is a very simple program, which creates a variable (`shopping`) that refers to a list and then prints out each of the items in turn. There are a couple of things to comment on here. Firstly, the `for` statement creates the variable `item` (the variable name can of course be anything that you want), then sets the value to each of the elements in the list. The line that is indented is then executed for each value assigned to the `item` variable, printing out this value.

To execute the program you first need to save it. You can save the file anywhere you like on your computer (it helps if you remember where), but it is a good idea (particularly when working in Windows) to give the file an extension of ".py". This will mean that the computer will recognise it as a Python program. Once you have saved the file, you can press F5 (or choose Run->Run module from the editor window's menu) and the output should appear in the Python shell window.

Whenever we want to execute a bit of Python code several times, a for loop is one of the ways that we can do it. Python recognises the lines we want to form part of the loop by the level of indentation and it is vital that you maintain consistent indentation throughout your programs. For example, you can choose to indent lines of code with spaces or with tabs but, whichever one you choose, you should only use one or the other for your whole program. Also, make sure that you keep the amount of indentation consistent across all the levels in your code. You will find that this approach makes your programs easier to read and understand, because you can see the structure of the program at a glance by the indentation.

Exercise 2.1 Change the program above by adding a second list (with a different variable name) to the program, which contains cheese, flour, eggs, spaghetti, sausages and bread. Change the loop so that instead of printing the element, it appends it to the old list. Then, at the end, print out the new list.

Making Decisions Don't look at this if you haven't done the exercise above. My solution:

```
In [62]: shopping = ['bread', 'potatoes', 'eggs', 'flour', 'rubber duck', 'pizza',
         extrashopping = ['cheese', 'flour', 'eggs', 'spaghetti', 'sausages', 'brea
         for item in extrashopping:
             shopping.append(item)
         print(shopping)

['bread', 'potatoes', 'eggs', 'flour', 'rubber duck', 'pizza', 'milk', 'cheese', 'f
```

This looks like it's worked exactly as I described, but maybe not quite as I intended. We seem to have too many eggs and too much bread. This might not be a problem (and it does show that the same value can be present in a list more than once), but I really just want one copy of each item. To achieve this, we need to include a check before we add an element to the list, to make sure that the value isn't in there already. Fortunately, Python lets us do this really easily. For an example of this, go back to the Python Shell for a minute and try:

```
In [63]: shopping = ['eggs', 'cheese', 'milk']

In [64]: 'eggs' in shopping

Out[64]: True

In [65]: 'frogs' in shopping

Out[65]: False

In [66]: 'frogs' not in shopping
```

```
Out[66]: True
```

We can use this in a new Python statement, which allows us to only execute statements if a particular condition is true. Back in the editor window, the program could be changed to:

```
In [67]: shopping = ['bread', 'potatoes', 'eggs', 'flour', 'rubber duck', 'pizza',
         extrashopping = ['cheese', 'flour', 'eggs', 'spaghetti', 'sausages', 'brea
         for item in extrashopping:
             if item not in shopping:
                 shopping.append(item)
         print(shopping)

['bread', 'potatoes', 'eggs', 'flour', 'rubber duck', 'pizza', 'milk', 'cheese', 's
```

Much better. A couple of points to notice with the indentation. The `if` statement is indented with respect to the `for` statement, so it will be executed every time the loop executes. The `.append` method call is indented with respect to the `if` statement, and so it will only be executed if the condition in the `if` statement (i.e., `item not in shopping`) is true.

Exercise 2.2

(i) Change the program above to print out a message when a duplicate item is found. To do this, you could add another `if` statement to see if the item is in the list. Alternatively, you can add an `else:` clause to the existing `if` statement. This will be executed when the condition in the `if` statement is false.

(ii) The example illustrated above is not the only solution to adding items to a list, whilst checking for duplicates. From the three choices below, choose the version that would achieve the same goal:

a)
```
shopping = ['bread', 'potatoes', 'eggs', 'flour', 'rubber duck', 'pizza', 'mil
extrashopping = ['cheese', 'flour', 'eggs', 'spaghetti', 'sausages', 'bread']
for item in extrashopping:
if item not in shopping:
    print item, "is already in the list."
else:
    shopping.append(item)
print(shopping)
```

b)
```
shopping = ['bread', 'potatoes', 'eggs', 'flour', 'rubber duck', 'pizza', 'mil
extrashopping = ['cheese', 'flour', 'eggs', 'spaghetti', 'sausages', 'bread']
for item in extrashopping:
if item in shopping:
    shopping.append(item)
else:
    print item, "is already in the list."
print(shopping)
```

c)
```
shopping = ['bread', 'potatoes', 'eggs', 'flour', 'rubber duck', 'pizza', 'mil
extrashopping = ['cheese', 'flour', 'eggs', 'spaghetti', 'sausages', 'bread']
for item in extrashopping:
if item in shopping:
    print item, "is already in the list."
else:
    shopping.append(item)
print(shopping)
```

Counting Loops Looping over elements of a list is great, but there are other circumstances where you just want to do something a set number of times. Most programming languages have a `for` statement which does exactly this, but Python doesn't. Fortunately, Python has a function which generates a list of numbers for us to use in a `for` loop. Go to the Python shell and type:

```
In [68]: range(10)

Out[68]: range(0, 10)
```

The `range()` function gives a `Range` object, which can be used to generate a list of integers. In Python 2 `range()` directly creates a list of integers instead of a `Range` generator, which is technically slightly different. In either case, a loop like this:

```
In [69]: for i in range(10):
             print(i)

0
1
2
3
4
5
6
7
8
9
```

prints out the numbers 0 to 9 one to a line. You can use the `range` function to produce most lists of numbers that you might need.

Note At first glance, it might seem inconvenient that you don't get a list as output from the `range()` function in Python 3. The reason behind this is that the `Range` object is a much faster and more efficient way of generating values that will be looped through one-at-a-time, and this is the aim of the vast majority of calls to the `range()` function. Of course, if you do actually want to create a full list of integer values in a range, in Python 3 you can pass the use of `range()` into the explicit initialisation of a list as below:

```
In [70]: range_list = list(range(10)) # or
         range_list = [range(10)]
```

Exercise 2.3 Explore what you can do with the `range` function. It can take just one number as we did above, or two as starting and ending values, or even three - the start, the end and a step value. Try all three versions of the `range` command, and then work out how to produce the list: `[4, 11, 18, 25]`.

Direct and Indirect Loops So, `range` can get us a list that we can use to count to any number that we want, but why does it stop short of the upper limit we give it? Why does `range(N)` mean 0..N-1 instead of 0..N or 1..N? Well, try out the following two pieces of code:

```
In [71]: for item in shopping:
             print(item)
```

```
bread
potatoes
eggs
flour
rubber duck
pizza
milk
cheese
spaghetti
sausages
```

and

```
In [72]: for i in range(len(shopping)):
             print(shopping[i])
```

```
bread
potatoes
eggs
flour
rubber duck
pizza
milk
cheese
spaghetti
sausages
```

They should be exactly the same: `range` behaves as it does so that you can use it to generate lists of indexes for sequence data types. In the blocks of code above, the first is an example of a direct loop, where you pull out the items one by one directly from the list. The second is an indirect list, where you step through the indices and use them to access the required elements from the list.

Which one is better? Generally, the direct method is slightly clearer and a bit more *Pythonesque*. However, there are circumstances where an indirect loop is the only option. If you have two lists of the same size, you might need to print out the corresponding elements of the two lists (although there might be better ways to do this, as well). In this case, you can use `range` with the size of one of the lists, and then use the index to get the corresponding elements from both.

Exercise 2.4 Start with your shopping list (or a new, shorter one to save some typing) and create a new list with the amounts you need to buy of each item. So for example:

```
In [73]: shopping = ['bicycle pump', 'sofa', 'yellow paint']
         amounts = ['1', '7', '9']
```

Then write a loop to step through and print the item and the amount on the same line.

String Formatting When you print out pairs of values like in the exercise above, the output is a bit boring. It's just a name and a number on a line. It could be a bit prettier, or at least more nicely formatted. You can put a few extra strings in there to make it clearer like this,

```
In [74]: print("I need to buy", amounts[i], shopping[i])
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)

<ipython-input-74-5fb07857e59d> in <module>()
----> 1 print("I need to buy", amounts[i], shopping[i])


IndexError: list index out of range
```

In Python 2.x this would be: `print 'I need to buy', amounts[i], shopping[i]`

```
In [75]: for i in range(len(shopping)):
             print('I need to buy', amounts[i], shopping[i])

I need to buy 1 bicycle pump
I need to buy 7 sofa
I need to buy 9 yellow paint
```

which is maybe a bit better. Taking this approach is ok, but it is difficult to control the formatting, particularly when you are mixing numbers and strings. Most programming languages have some function or facility for creating formatted strings and Python is no exception.

In Python's case, formatting of strings can be taken care of in several different ways.

1. by using the `%` operator that is common amongst a lot of languages
2. by using the `.format` method, or
3. (from Python v3.6 onwards) by using the `f''` syntax with variable names in placeholder.

Let's compare these options. We have three variables, `name`, `date` and `job`, which we want to substitute into some text. It's possible to control exactly the formatting of values such as dates when constructing strings like this, but for simplicity here we perform simple string substitutions at each step.

```
#variables for substitution
name = 'Betty'
date = '15th June 2016'
job = 'engineer'

# option 1 - using the % operator
text = 'Hi, my name is %s and I am an %s. I have been an %s since %s.' % (name, job
print('1. using %')
print(text)

# option 2 - using the .format method of the string object
text = 'Hi, my name is {0} and I am an {1}. I have been an {1} since {2}.'.format(n
print('2. using .format()')
print(text)

# option 3 - using f'' (v3.6 only)
text = f'Hi, my name is {name} and I am an {job}. I have been an {job} since {date}
print('3. using f\'\' (v3.6 only)')
print(text)

1. using %
Hi, my name is Betty and I am an engineer. I have been an engineer since 15th June
2. using the .format method of the string object
Hi, my name is Betty and I am an engineer. I have been an engineer since 15th June
3. using f'' (v3.6 only)
Hi, my name is Betty and I am an engineer. I have been an engineer since 15th June
```

From now on, we will use the newer `.format()` approach, but you might prefer to use `%` (or `f''` if you are using v3.6) - I recommend that you read this for a good introduction and comparison of the `.format` and `%` approaches.

When formatting, you start with a string containing placeholders: patterns of characters that indicate the position where you want to insert the values of your variables, and their format. Then, the variables to be inserted are supplied using the `format()` method of this string. The placeholders take the form of curly brackets `{}` containing a code that tells Python what to do with the variables being inserted. For example:

```
In [76]: s = 'I need to buy {} {}'.format(7, 'snakes')
         s

Out[76]: 'I need to buy 7 snakes'
```

Don't we all? In the example above, we didn't place anything inside the curly brackets, so the values of the variables provided as arguments to the `format()` method were inserted in the order and format that they were given. However, you can specify the order of insertion by including a number between the curly brackets, like so:

```
In [77]: s = 'I need to buy {0} {1} because I have {0} {2}'.format(7, 'mice', 'snak
         s
```

```
Out[77]: 'I need to buy 7 mice because I have 7 snakes to feed'
```

The placeholders can also contain information for formatting the inserted value. For example, to control level of precision on a floating point number, you can use `{:.Nf}` where `N` is the number of decimal places that you want to display.

```
In [78]: mousePrice = 9.5
         numberOfMice = 7
         s = 'Each mouse costs EUR {:.2f} and I need {} mice, so the total cost wil
         .format(mousePrice, numberOfMice, mousePrice*numberOfMice)
         s

Out[78]: 'Each mouse costs EUR 9.50 and I need 7 mice, so the total cost will be EU
```

There are a lot of other formatting options that can be controlled by these patterns in placeholders e.g. you can automatically print large numbers split with commas, or you can print text in clearly-defined columns buffered with whitespace. For a full list and explanation, you should check out the Python documentation at https://docs.python.org/3/library/string.html#format-string-syntax.

Exercise 2.5 In the example below, we have changed the program so it prints out a formatted message for each of the items in the shopping list along with the amount that needs to be bought of that item. Parts of the program are missing. You need to fill them in:

```
In [79]: shopping = ['bread', 'potatoes', 'eggs', 'flour', 'rubber duck', 'pizza',
         amounts = ['1', '10', '12', '1', '2', '5', '1']
         --- i in range(len(---)):
             s = 'I need to buy --- ---'.format(amounts[---], ---[i])
             print(---)


          File "<ipython-input-79-4dd3d8d22215>", line 3
        --- i in range(len(---)):
                                ^
    SyntaxError: invalid syntax
```

Looking Up Data Keeping data in parallel arrays like this is fine if you are really really careful and you don't need to change the arrays that much. Otherwise, it is prone to errors. One way of getting around this (and our last new data type) is to use a *dictionary*. Dictionaries are sort of like lists, but instead of holding just a single value, they hold a key-value pair. So, when you want to look up a value in the dictionary, you specify the key and the dictionary returns the value, rather than just using an index. An example might help:

```
In [80]: studentNumbers = { 'Bioscience Technology': 16,
                            'Computational Biology': 12,
                            'Post-Genomic Biology': 20,
```

```
                                'Ecology and Environmental Management': 3,
                                'Maths in the Living Environment': 0
                               }
studentNumbers['Bioscience Technology']
```

```
Out[80]: 16
```

The data is enclosed in curly brackets and is a comma separated list of key-value pairs. The key and value are separated by `:`. The key can be any immutable type (so, mainly strings, numbers or tuples). Notice I have split the assignment statement to create the dictionary over several lines, to make it easier to read. Normally, Python expects a command to be on a single line, but sometimes it recognises that a command isn't finished and lets you continue on the next line. This mainly happens when you haven't closed a set of brackets, which in the above example was deliberate, but in my case is usually because I have forgotten. Python will continue to prompt for input until you close the bracket properly before trying to execute the command.

Dictionaries themselves are a mutable datatype, so the values associated with a key can be changed:

```
In [81]: studentNumbers['Bioscience Technology'] += 1 # x += 1 does the same as x =
         studentNumbers['Bioscience Technology']
```

```
Out[81]: 17
```

If you try to assign a value to a key that doesn't exist, Python creates the entry for you automatically:

```
In [82]: studentNumbers['Gardening'] = 10
         studentNumbers['Gardening']
```

```
Out[82]: 10
```

Getting rid of entries in the dictionary is easy as well, using the `del` statement:

```
In [83]: del studentNumbers['Maths in the Living Environment']
         studentNumbers
```

```
Out[83]: {'Bioscience Technology': 17,
          'Computational Biology': 12,
          'Ecology and Environmental Management': 3,
          'Gardening': 10,
          'Post-Genomic Biology': 20}
```

If we know the keys in the dictionary we can look up the values. If we want to loop over the values in the dictionary, we could create a list of the keys and loop over that, but that's no better than keeping the keys and values in separate lists. Instead, Python can create a list of the keys for you when you need it:

```
In [84]: studentNumbers.keys()
```

```
Out[84]: dict_keys(['Bioscience Technology', 'Computational Biology', 'Post-Genomic
```

We can now put this into a `for` loop, with or without sorting it first. If we are not bothered about the order, then we can use `for` and `in` to loop directly over the keys in the dictionary:

```
In [85]: for key in studentNumbers:
             print(key, studentNumbers[key])

Bioscience Technology 17
Computational Biology 12
Post-Genomic Biology 20
Ecology and Environmental Management 3
Gardening 10
```

That should work as expected. Python doesn't make any promises about the order the keys will be supplied in: they will be given the way Python thinks is best. It almost certainly won't be either the order the keys were added to the dictionary or alphabetical order*.

***Note** The way that dictionaries are implemented in Python fundamentally changed in v3.6, resulting in them taking up ~1/2 the space and working ~2x as fast as they used to. A side effect of this is that dictionary objects in Python 3.6 remember the order that entries were created in and you should be able to access their entries in this order. Regardless, in the examples and exercises in this course, we assume that this order cannot be relied upon - we don't expect everyone to be using v3.6 or above, and anyway this is not yet considered a 'stable' feature of the language i.e. future versions of Python are not guaranteed to preserve the order of dictionaries. When writing your own code, if you want to access dictionary entries in a particular order, you should make sure to do so by providing keys in a specific order, as we will show below.

As well as getting the keys, you could also get the values as a list using `.values()`. Slightly more efficient is to get the key-value pairs in one step using `.items()`:

```
In [86]: studentNumbers.values()

Out[86]: dict_values([17, 12, 20, 3, 10])

In [87]: studentNumbers.items()

Out[87]: dict_items([('Bioscience Technology', 17), ('Computational Biology', 12),
```

Have a careful look at this output. The square brackets show that this is a list of things. But each item in that list is in fact two pieces of data in round brackets. We came across this briefly above, and it is a tuple. There are two ways we can use this in a `for` loop. Firstly, we can use a variable which will contain the tuple and unpack it in body of the loop:

```
In [88]: for data in studentNumbers.items():
             print(data[0], data[1])

Bioscience Technology 17
Computational Biology 12
Post-Genomic Biology 20
Ecology and Environmental Management 3
Gardening 10
```

or (this is usually my preference) you can unpack the data directly and more explicitly in the `for` statement:

```
In [89]: for course, students in studentNumbers.items():
             print(course, students)
```

```
Bioscience Technology 17
Computational Biology 12
Post-Genomic Biology 20
Ecology and Environmental Management 3
Gardening 10
```

This is a little terse, so let's use the `.format()` method that was introduced earlier.

```
In [90]: for course, students in studentNumbers.items():
             print('Course {} has {} students'.format(course, students))
```

```
Course Bioscience Technology has 17 students
Course Computational Biology has 12 students
Course Post-Genomic Biology has 20 students
Course Ecology and Environmental Management has 3 students
Course Gardening has 10 students
```

The output of `.items()` is our first example of a compound data structure (in this case a list of tuples). The ability to easily construct arbitrarily complex data structures like this is one of the most powerful features of Python and one we will explore more in the next worksheet.

Exercise 2.6 Go back to your shopping list code from exercise 2.5 and change the program so that the amounts and shopping items are stored in a dictionary, then print out the items and their respective amounts by looping over the dictionary. Do it twice, once looping over the the dictionary to get the keys (or use the keys to get the values) and once by getting the key-value pairs directly from the dictionary.

Parcelling Up Code Often we come across situations where we would want to do the same type of calculation several times in a single program. Many of the Python modules provide functions for doing just this (and some of you will probably have used the `math.sqrt()` function earlier). However, you can define your own functions if you want. This can be done anywhere in your program, but is conventionally done at the beginning. In any case, the important thing is that you define a function before you try to use it in your program.

As a trivial example, here is a function definition which squares a number:

```
In [91]: def square(x):
             return x*x
```

When Python comes across this in your program, it does nothing visible. Only afterwards when you call the function does it produce any effect. The `x` between the brackets in the `def` line is called an argument, and acts as a placeholder for whatever (in this case) you want to square. Once the function is defined, you can call it using anything in place of the `x`. For example to square the number 3, you would use:

```
In [92]: square(3)

Out[92]: 9
```

If you wanted to store the result in a variable, you could use

```
In [93]: y = square(3)
```

and you could even pass a variable into the function:

```
In [94]: z = square(y)
         z

Out[94]: 81
```

Functions are incredibly versatile and a single function may take many arguments. They can contain more than one line of code, and can do anything that you can do in other parts of a Python program. You will see a much more complex example in Worksheet 3. Parcelling up code like this means that you don't have to type it out every time the task is repeated in your program, and if you need to change it, it will only have to be changed once.

Exercise 2.7 In worksheet one, exercise 1.1, you should have worked out an expression for calculating the hypotenuse of a right angled triangle given the other two sides. Now, complete function below, which should calculate the hypotenuse, and test it by calling

`hypot(3,4)`

and it should return the value 5.

```
In [95]: --- hypot(---, sideb)---
             h = (sidea**--- --- ---**2)**0.5
             return ---


           File "<ipython-input-95-fbd24c01f21c>", line 1
         --- hypot(---, sideb)---
                       ^
     SyntaxError: invalid syntax
```

Summary

- `for` loops can be used to repeat a block of code for each item in a list.
- `range()` can be used to create a list of numbers, and to repeat the loop for each of those numbers, to execute the loop a given number of times.
- `if: elif: else:` statements can be used to choose one of a number of optional blocks of code depending on the conditions in the `if` and `elif` clauses.
- String interpolation allows you to insert values into a string, enabling sophisticated formatting.
- Tuples are a new data type which are like immutable lists.

- Dictionaries are another object data type which stores key-value pairs.
- The `.keys()`, `.values()` and `.items()` methods are used to get lists of the contents of a dictionary.
- Functions can contain pieces of code to be used repeatedly, which only need to be debugged and changed once.

Debugging Exercise The code below contains some typos and errors. Read the description and then follow the code, making sure you understand what each line does/is supposed to do, and correct any mistakes you encounter. Finally run the code and check that the output is the same as that indicated below.

Description:

The function `max_of_two` takes a list of tuples, with each member tuple containing two numbers each. The function allocates each number to either a 'Low' or 'High' category, in a pre-defined dictionary, and then returns this dictionary containing the new input. There should be no repeated values in each category. Finally, the code should print formatted strings containing the high and low values collected.

Output expected:

```
The high values are [0.026, 1.09, 0.953, 3.09, 0.943, 0.426, 1.29]
The low values are [0.023, 0.666, 0.346, 0.022, 0.336, 0.076, 0.356,
0.146]
```

```
In [96]: # define variables to run function on
         list0 = [ (0.023, 0.026), (1.09, 0.666) (0.346, 0.953), (0.022, 0.026), (3

         # define function
         def max_of_two(nmrs_list_list)

             # define dict with assessment values
             assessment_values = { 'Low': {}, 'High'; []}

             # run through elements (lists) in input list and categorise elements
             for item in nmrs_list_list:
                 a = item(0)
                 b = item[1]
                 if a < b:
                 # add a to low and b to high (no repeats)
                 if a not in assessment_values['Low']:
                         assessment_values['Low'].append(a)
                 if b not in assessment_values['High']:
                         assessment_values['High'].append(b)
                 else
                 # add a to high and b to low (no repeats)
                     if a not in assessment_values[High]:
                         assessment_values[High].append(a)
                     if b not in assessment_values['Low']:
                         assessment_values['Low'].append(b)
             return assessment_values
```

```
    # run function on all variables
    assessment_vals = max_of_two(list0)

    # get output of code
    print('The high values are ()'.format(assessment_vals['High']))
    print('The low values are {}'.format(assessment_vals['Low']))


     File "<ipython-input-96-104dd5dc0243>", line 5
   def max_of_two(nmrs_list_list)
                                 ^
SyntaxError: invalid syntax
```

Chapter 3

Nested Data Structures

You now know about the most commonly-used Python data types. There are more, each with characteristics suited to a particular task or situation and, if you like, you can make your own as well. You can see the full list of built-in data structures here: https://docs.python.org/3.5/library/stdtypes.html. You should also have an idea of the ways that you can repeat actions on lists and dictionaries with `for` loops, as well as take decisions based on data using `if` statements. This knowledge and these skills can be applied to some quite sophisticated programming.

One other key to writing effective programs is the ability to encapsulate your data and allow them to be accessed and analysed in a way that is efficient and appropriate. If you don't take the time to consider the best way to capture, store, and access your data, you can end up writing programs that are much more complicated and error-prone than they need to be. This can cost you a lot more time in the future. As with any project, a little time spent planning at the beginning can save a lot of time later on.

Warmup exercise Imagine, you woke up this morning and you had several ideas for things you want to do today, each can be described by a word or a short sentence. You want to write down those ideas (store them in Python), which data type is in your opinion best suited to host all of that data? - an integer - a string - a list - a group - a dictionary - other

[write down answer]

After compiling your collection of all the things you want to get done today, you realized that they can and should be categorized into different projects like 'work', 'home' etc. Given your response to the question above, how would you store your data into multiple levels?

[write down answer]

Combining Structures Often, datasets are best encapsulated by combining the basic data structures together to form larger, well-organised collections that fit the characteristics of the dataset. This combination of multiple data structures is often referred to as 'nesting'. Getting used to dealing with these structures as they grow larger and more complicated can take a bit of time, but it becomes easier with practise and is very important for good programming.

To make this easier to understand, let's begin with some examples. Let's say that you teach seminars to three different groups of students, and you are writing a program that will keep track of the names of these students. For each seminar group, you could store the names in a list.

```
In [97]: GroupA = ['Sarah', 'Richard', 'Matthew', 'Fiona', 'Sally', 'Samuel']
```

```
GroupB = ['Richard', 'Sally', 'Sandy', 'Peter', 'Rebecca', 'Steve', 'Lesle
GroupC = ['Simon', 'Laura', 'Gareth', 'Alan', 'Helen']
```

Now, you might want to go through all of the names for all of the groups. You could write three loops:

```
In [98]: for student in GroupA:
             print(student)

Sarah
Richard
Matthew
Fiona
Sally
Samuel


In [99]: for student in GroupB:
             print(student)

Richard
Sally
Sandy
Peter
Rebecca
Steve
Lesley
Stuart


In [100]: for student in GroupC:
              print(student)

Simon
Laura
Gareth
Alan
Helen
```

Teaching can be quite time-consuming at the best of times, and this is getting quite repetitive! Of course, it gets even worse if you teach even more groups, or if you are working with a larger dataset of some other kind. Instead, you could combine the lists together into a single list,

```
In [101]: AllGroups = GroupA + GroupB + GroupC
```

but you might not want to lose track of which students belong to which class. So instead, it is a good idea to store the individual lists as entries in another data structure that can be processed iteratively just like the lists themselves. We have a few options here, but let's begin with a list of lists - where each entry in the list is itself another list:

```
In [102]: AllGroups = [ GroupA , GroupB , GroupC ]
          # This is equivalent to the below, but much easier to read!
          AllGroups = [ ['Sarah', 'Richard', 'Matthew', 'Fiona', 'Sally', 'Samuel']
                        ['Richard', 'Sally', 'Sandy', 'Peter', 'Rebecca', 'Steve',
                        ['Simon', 'Laura', 'Gareth', 'Alan', 'Helen'] ]
```

Now, to work through all the names of all the students, we need to iterate over each entry in the top-level list `AllGroups` and, because we know that each of these entries is itself a list, iterate over the entries of this second-level list as well. As you will remember from the previous worksheet, iterating over a list can be acheived with a `for` loop. In this case, to iterate over everything in our two-level nested lists we will just use two nested `for` loops.

```
In [103]: for group in AllGroups:
              for student in group:
                  print(student)
```

```
Sarah
Richard
Matthew
Fiona
Sally
Samuel
Richard
Sally
Sandy
Peter
Rebecca
Steve
Lesley
Stuart
Simon
Laura
Gareth
Alan
Helen
```

Great! So now we can get a full list of our students' names, without writing an individual `for` loop for every class we teach. But we have lost some information in the process, because we can't tell by looking at the list above where the names for one class end and another begin.

Exercise 3.1 What other combination of data structures might we use to encapsulate our class lists? Before you look at the next section, try to figure out what the best nested structure might be for this data.

Choosing An Appropriate Structure Consider what information we have for each class: it has a name (A, B, C) and a list (the student names). This sounds a lot like the kind of information that is best stored in a dictionary - we have a key (the group name) and an associated value (the list of students). So, we can combine the data into a second type of nested structure: a dictionary of lists.

```
In [104]: AllGroups = { 'Group A': GroupA,
                        'Group B': GroupB,
                        'Group C': GroupC }
```

Now that we have produced this dictionary, we can iterate over it, and each of the lists contained within, using a verysimilar nested `for` loop structure to the one we had before for our list of lists:

```
In [105]: for group in AllGroups.keys():
              print(group)
              for student in AllGroups[group]:
                  print(student)
```

```
Group A
Sarah
Richard
Matthew
Fiona
Sally
Samuel
Group B
Richard
Sally
Sandy
Peter
Rebecca
Steve
Lesley
Stuart
Group C
Simon
Laura
Gareth
Alan
Helen
```

Things have improved a little. Now we are printing the name of the group each time we start a new one, but these group names are not in alphabetical order (unless you got lucky - remember that the order that keys are accessed from a dictionary using `.keys()` can't be relied upon or predicted, or you're using Python 3.6) and they're quite hard to spot amongst the names of the students. To make sure that you understand what just happened, let's take a look at that code block above step-by-step.

```
for group in AllGroups.keys():
```

Here, we create a `for` loop to iterate over every key in the dictionary `AllGroups`. Hopefully, you recognise this from the previous worksheet. At the start of each iteration in this `for` loop the value of the variable `group` will be assigned as the next key from the dictionary.

```
print(group)
```

We print the group name before starting to loop through the list of student names.

```
for student in AllGroups[group]:
```

Now, we are defining the second `for` loop, assigning the value of the variable `student` with the name of the next student in the list for the current group. Here, remember that `AllGroups[group]` returns the value associated with the key `group` in the dictionary `AllGroups`. `group` is whichever group name we are currently dealing with in this iteration of the first for loop. That is, if the first key returned by `AllGroups.keys()` is `'Group B'`, then:
- `group` has been assigned the value `'Group B'`,
- so `AllGroups[group]` is currently equivalent to `AllGroups['Group B']`,
- which returns the value associated with the key `'Group B'`, which is the list `GroupB`.

If you followed that, then you will understand that the last line

```
print(student)
```

will print out the name of the next student in the list referred to by `AllGroups[group]`.

The order of the four lines of code above is very important. Consider what would happen if you put the `print(group)` line within the second `for` loop. You can even give it a try and see if your hypothesis was right.

Exercise 3.2 As mentioned before, there are a couple of problems with the output that we are getting from the code block above. Try to find a way to make the names of the groups stand out a bit more from the names of the students.
After you have achieved that, see if you can find a way to explicitly control the order in which the groups are displayed, so that they appear alphabetically - Group A, Group B, Group C.

More Nesting There is one more type of nested data structure that we need to consider. To help with this, we need to expand the dataset slightly. Being the diligent and conscientious tutor that you are, you spent hours preparing, running and marking an assessment for each seminar group. You have collected the results, which are given below:

Group A

Student	Mark
Sarah	78
Richard	65
Matthew	53
Fiona	71
Sally	43
Samuel	80

Group B

Student	Mark
Richard	57

Student	Mark
Sally	89
Sandy	75
Peter	77
Rebecca	62
Steve	71
Lesley	75
Stuart	80

Group C

Student	Mark
Simon	47
Laura	91
Gareth	74
Alan	61
Helen	74

Now, we have pairs of data for each group - the student's name and their mark. We want to store this data in a way that keeps all of the results together for all of the groups, but allows the data for each group and individual student to be accessed independently. If you first consider the groups individually - pairs of names and marks - this type of dataset is clearly best represented as a dictionary. And, for the reasons discussed before, we know that storing each group and it's data in a dictionary that can be accessed by the name of the group is a good idea too. So, to store all of this information we want to build a dictionary of dictionaries. First, let's create our individual dictionaries for each group.

Note If you're not working interactively with the IPython Notebook version of this workbook, you could be about to do a lot of typing to enter all this data yourself. To save you the trouble, the data is available as a file from GitHub. However, we won't cover how to read data from a file until the next worksheet. So for now, you can either skip ahead to find out how to do it (but make sure that you come back later!), read through and try to follow along without typing everything in yourself (you might find it difficult to understand exactly what's going on this way), or type/copy the whole lot. Sorry!

```
In [106]: GroupA_Results = { 'Sarah'   : 78 ,
                             'Richard': 65 ,
                             'Matthew': 53 ,
                             'Fiona'  : 71 ,
                             'Sally'  : 43 ,
                             'Samuel' : 80 }
          GroupB_Results = { 'Richard': 57 ,
                             'Sally'  : 89 ,
                             'Sandy'  : 65 ,
                             'Peter'  : 77 ,
                             'Rebecca': 62 ,
                             'Steve'  : 71 ,
```

```
                          'Lesley' : 75 ,
                          'Stuart' : 80 }
GroupC_Results = { 'Simon'  : 47 ,
                   'Laura'  : 91 ,
                   'Gareth' : 74 ,
                   'Alan'   : 61 ,
                   'Helen'  : 74 }
```

Now, we can create a top-level dictionary with three entries — one for each group. The keys are the names of the groups, and the associated values the dictionary of results for the students in the groups.

```
In [107]: AllGroupResults = { 'Group A' : GroupA_Results ,
                              'Group B' : GroupB_Results ,
                              'Group C' : GroupC_Results }
```

All of the assessment results are stored in a single dictionary. You can access the results for a particular group quite easily, using the approach introduced earlier:

```
In [108]: AllGroupResults['Group C']

Out[108]: {'Alan': 61, 'Gareth': 74, 'Helen': 74, 'Laura': 91, 'Simon': 47}
```

But what if you want to know how a particular student scored? How can we access the value for a particular key in the dictionary that is itself the value associated with a key at the top level? Well, above we accessed the value associated with the key `'Group C'` using the syntax `dictionary[key]`. In this case, we know that that returns another dictionary. So, if we now want to get the score for a particular student in that group, we just query that dictionary in the same way.

```
In [109]: AllGroupResults['Group C']['Laura']

Out[109]: 91
```

If this looks strange to you, or you're struggling to make sense of it, remember that Python will interpret the line from left to right:
- first, it comes across the variable `AllGroupResults`, which it identifies as a dictionary
- then, it sees that you want to extract the value associated with the key `'Group C'`
- it fetches that value, `AllGroupResults['Group C']`, and identifies it as a dictionary, too
- then it moves on to the last part of the line, `['Laura']`, and recognises that you want to extract the value associated with the key `'Laura'` in the dictionary given by `AllGroupResults['Group C']`
- finally, it fetches that value and returns it.

Exercise 3.3 Using what you've learned above about iterating over nested data structures, write a program that will loop through every student in every group and print out their mark. Make sure that you can identify in the output of your program, the group that each student name has come from.
Once you have acheived this, try changing the behaviour of your program to calculate and output the (mean) average mark for each group.

Scaling Up Being able to store and access data in a suitable structure is all well and good, but the benefits of it become more obvious when you have a repetitive task that you need to complete. This is especially true when you consider this kind of problem applied to a dataset much larger than the toy examples that we are working with here. What if you ran a whole faculty, with tens of different seminar groups, containing hundreds of students, who each take multiple exams/assesments?!

In this kind of situation, manual entry of the data isn't practical. Instead, it is much more helpful to be able to read the information that you need from a file. Working with input from other sources, and output of results, will be covered in the next worksheet.

Exercise 3.4 The exercises in the next worksheet are designed to be more challenging for newcomers to programming/Python. In preparation for this, here is one last exercise designed to consolidate the things that you have learned so far.
Now that you have finished marking your students' assessments, you need to let them know how they got on. Write a program that will print out the body of an email to each student according to this template:

Dear *[name]*,
I have finished marking the assessment for your seminar group, *[group name]*. You scored *[their mark]*.
[an additional comment according to their score (see below)]
Kind regards,
[your name]

The additional comment on the third line of the email should be chosen according to the mark that they scored on the assessment.

Score range	Comment
<60:	'You must try harder next time. Are you taking this course seriously?'
60-79:	'Well done, that's a good score.'
>79:	'Congratulations! That's an excellent score!'

Of course, the potential for nested data structures doesn't stop at two-level lists of lists, or lists of dictionaries, or dictionaries of lists, or dictionaries of dictionaries! Depending on the situation, you might want to combine some of the other different structure types (which we don't cover so much here), build up three- or four-level structures, and so on. Be warned, though: with every additional layer, your program becomes more and more complicated. This makes it harder for you to keep track of what you're dealing with while you're writing it, harder to read when you or someone else comes back to the program at a later date, and more difficult to identify and correct mistakes in the code itself.

Summary

- The elements of data types like lists or dictionaries can themselves be things like lists or tuples or dictionaries, allowing arbitrarily complex data structures to be built up.
- `for` loops can be nested in order to access every entry in these complex structures.
- Individual entries can be accessed by stacking up indices/keys, which Python interprets one at a time, from left to right.

- The values of variables can be inserted into strings, with their format controlled, using the `.format()` string method.

Debugging Exercise In this exercise, you are presented with some code that doesn't work as it should. Your task is to debug it.

```
In [110]: # You start with a group of students and scores and you
          # would like to print the name and the score of each.
          # Can you spot and fix the errors?
          # Hint: Each line contains at least one error/mistake

          group = '{Sarah:20, Richard:30, Matthew:40, Fiona:50, Sally:60, Samuel:70

          for student in group_of_students:
              print("The score of {} is {}".format(student, group(student))


            File "<ipython-input-110-89d6b4eb6b85>", line 9
          print("The score of {} is {}".format(student, group(student))
                                                                        ^
      SyntaxError: unexpected EOF while parsing
```

Chapter 4

Plotting Data with Matplotlib and Bokeh

Opening Files What we have been doing so far has required you to type the data into the programs by hand, which is a bit cruel. For this worksheet, we will be using a larger dataset (still tiny by many standards) and you can download a file containing the data from the GitHub repository where these teaching materials are maintained. Just save the linked file into the same directory as you are keeping the Python scripts. (*Note: if you already obtained these materials from the repository, you probably downloaded the data file, into the same folder, at the same time.*)

Of course, this requires that we know how to get data out of the file and into our Python program and that is what we are going to do in this worksheet. Specifically we are talking about reading data out of text files. Binary files face their own challenges, and I am not going to get into that in this course since handling them is very dependent on the implementation of the binary file. In any case, for a number of significant classes of binary files, such as images, BAM files or NetCDF formatted data, there are already Python modules to enable you to access the data in a simple way. But in any case, we will look at text files for now and firstly we need to know how to open them.

If you have downloaded the file, you should make sure that it is saved into the same folder where you are going to save the python programs that you will use to analyse it. We will start simple, just by opening the file at the Python shell prompt.

```
In [111]: f = open('speciesDistribution.txt', 'r')
```

The file is now open, and `f` is a variable referring to a *file* data type. As you might have guessed, the first argument for `open` is a string containing the filename, but the `'r'` probably needs to be explained. This argument is called the file mode, and `'r'` means that you only want to read data. If you specify `'w'`, it means that you want to write data into the file, which we will talk about later. One very important point is that when you open a file that already exists for writing, the contents of the file are cleared, and can't be recovered. If you instead want to append data to an existing file you should specify `'a'` as the mode. If you specify `'r+'` then you can read and write to the file. These are the same regardless of the operating system that you are working on, but Windows has a few specific ones of it's own, which you shouldn't use if you can avoid them.

As you might expect by now, file objects have their own methods and you can use some of these to read data from the file. The easiest way of doing this is to use `.readlines()`:

```
In [112]: lines = f.readlines()
          lines
```

```
Out[112]: ['Site: Hetchell Wood N\n',
           'A\t12983\n',
           'B\t8493\n',
           'C\t948\n',
           'D\t9384\n',
           'E\t4942\n',
           'G\t9834\n',
           'I\t1293\n',
           'J\t9348\n',
           'Site: Hetchell Wood S\n',
           'A\t9380\n',
           'B\t13928\n',
           'D\t949\n',
           'E\t19023\n',
           'F\t9384\n',
           'G\t948\n',
           'H\t9284\n',
           'J\t1093\n',
           'K\t3029\n',
           'Site: Hagg Wood\n',
           'A\t2039\n',
           'B\t9394\n',
           'C\t19380\n',
           'D\t9102\n',
           'E\t932\n',
           'G\t893\n',
           'H\t5839\n',
           'J\t9302\n',
           'L\t984\n',
           'Site: Scoreby Wood\n',
           'A\t920\n',
           'B\t3928\n',
           'D\t9301\n',
           'E\t19384\n',
           'F\t12949\n',
           'H\t3892\n',
           'I\t9192\n',
           'K\t912\n',
           'Site: Grimston Wood\n',
           'A\t123\n',
           'B\t1340\n',
           'C\t11984\n',
           'E\t9389\n',
           'F\t4320\n',
           'G\t1283\n',
           'J\t8193\n',
           'K\t193\n',
           'Site: Sutton Wood\n',
```

```
'A\t883\n',
'B\t293\n',
'C\t893\n',
'D\t18990\n',
'F\t3910\n',
'G\t930\n',
'H\t1738\n',
'I\t819\n',
'M\t9934\n',
'Site: Wheldrake Wood\n',
'B\t91\n',
'C\t22649\n',
'D\t2949\n',
'E\t901\n',
'G\t9204\n',
'H\t2040\n',
'I\t8173\n',
'L\t6781\n',
'M\t9184\n']
```

The variable `lines` now refers to a list of strings containing each of the lines in the file. Try looking at one or two of them. If you didn't look at the contents of the file before you opened it with your program, have a look at it now. If you compare `lines[1]` in Python with the second line in the file, you will see some differences. Most obvious is the presence of a `\n` at the end of each line in the Python list. These are *newline characters* and we need to remember to remove these when we process the data from the file. Although it looks like two characters, it is what is called an escape character: just a single character but one with special meaning to the program and which we cannot normally see in a string. On most of the other lines there is another escape `\t`, which is a *tab character*. Again, we need to remember this for use later. Tabs are often used to separate data items on the lines of text files because, amongst other reasons, they are much less likely to occur within the data than spaces.

Getting Data from Files Using `.readlines()` to create a list containing all of the lines is nice and simple, but has a major drawback. It's fine when your file is small enough to read all of the lines into memory, but if you are reading a 32Gb SAM file, you are likely to run into problems. Here, you want to read one line at a time, and process it. Python files do have a `.readline()` method that will read only one line, but it's best to just use a `for` loop. Python has an idea of 'iterable' data types which you can put into `for` loops. We have seen two of these so far: the list and the dictionary. For a list you get each element in turn, and for a dictionary you get each key in turn. Strings are also iterable and return each character in turn. The point of mentioning this now is that files are also iterable, and Python tries to pass you exactly what we want: one line at a time. So we can start to write a program now to start processing this data file.

Note This will be the largest program you have written so far, and what I do when I am embarking on writing a large program is to start with just the basic structure and make sure that works then add to the program step by step and keep running it to make sure it is doing what I expect before it gets too complicated.

To begin, **in an editor window**, type

```
In [113]: datafile = open('speciesDistribution.txt', 'r')
          for line in datafile:
              print(line)
```

```
Site: Hetchell Wood N

A        12983

B        8493

C        948

D        9384

E        4942

G        9834

I        1293

J        9348

Site: Hetchell Wood S

A        9380

B        13928

D        949

E        19023

F        9384

G        948

H        9284

J        1093

K        3029

Site: Hagg Wood

A        2039

B        9394
```

C 19380

D 9102

E 932

G 893

H 5839

J 9302

L 984

Site: Scoreby Wood

A 920

B 3928

D 9301

E 19384

F 12949

H 3892

I 9192

K 912

Site: Grimston Wood

A 123

B 1340

C 11984

E 9389

F 4320

G 1283

J 8193

```
K        193

Site: Sutton Wood

A        883

B        293

C        893

D        18990

F        3910

G        930

H        1738

I        819

M        9934

Site: Wheldrake Wood

B        91

C        22649

D        2949

E        901

G        9204

H        2040

I        8173

L        6781

M        9184
```

OK, so far so good, the program is basically printing the whole file out to the Python Shell window. However, I forgot about the newline characters at the end of the lines. You have probably noticed that the `print` statement automatically adds a newline to the end of everything it prints, so now we are getting two after each line, which is why the output is double-spaced. So the

first thing to do is to fix that, by removing the newline characters from the lines as we read them in. Strings have a `.strip()` method which removes any newlines, spaces or tabs (we call these characters 'whitespace') at the start and end of each line. So add the line

```
In [114]: line = line.strip()
```

to the loop before the print statement (at the correct level of indentation) and try the program again. Now the output should look single spaced.

```
In [115]: datafile = open('speciesDistribution.txt', 'r')
          for line in datafile:
              line = line.strip()
              print(line)
```

```
Site: Hetchell Wood N
A         12983
B         8493
C         948
D         9384
E         4942
G         9834
I         1293
J         9348
Site: Hetchell Wood S
A         9380
B         13928
D         949
E         19023
F         9384
G         948
H         9284
J         1093
K         3029
Site: Hagg Wood
A         2039
B         9394
C         19380
D         9102
E         932
G         893
H         5839
J         9302
L         984
Site: Scoreby Wood
A         920
B         3928
D         9301
E         19384
F         12949
```

```
H        3892
I        9192
K        912
Site: Grimston Wood
A        123
B        1340
C        11984
E        9389
F        4320
G        1283
J        8193
K        193
Site: Sutton Wood
A        883
B        293
C        893
D        18990
F        3910
G        930
H        1738
I        819
M        9934
Site: Wheldrake Wood
B        91
C        22649
D        2949
E        901
G        9204
H        2040
I        8173
L        6781
M        9184
```

Processing the File If we look again at the file, we can see that it consists of two types of data. Some lines contain the names of sampling sites and some contain a letter and a number. The letters are taxon designators and the numbers represent abundance of that taxon at that particular site (in this case, as measured by high-throughput DNA sequencing of 18S rRNA). We need to process the two line types differently and store the information in a suitable data structure.

Take a moment to think about how you think we might go about doing that, and what the best data structure type to use might be for storing the taxon codes and counts for each site. Don't worry if you find this a little confusing and/or daunting: we are going to work through it one step at a time, starting by identifying each site described in the data.

The lines with the site names in them all start with the substring `Site:`, so they are easy to recognise. We can use the string's `.startswith()` method in an if statement to identify these lines so that we can process them separately. Try using this method at the Python command line so you understand how it works before putting it into the program.

Exercise 4.1 Change the program to only print out the lines that start with `Site:`

```
In [116]:
```

Once that works, remove the `Site:` substring (and the space that follows it) from the string and just print the actual site name. Make sure that you store the name in a variable at this point as well - we will need it later.

```
In [116]:
```

Starting to Build the Data Structure Now that we have isolated the site names we can think some more about what kind of data structures we will use to store the data we read from the file. Remember what you learned in the previous worksheet, about how important it is to choose an appropriate data structure. In this case, we have some named sites and then some data corresponding to those sites. That to me sounds like a dictionary. The data we have for each site consists of several lines, which each contain a taxon code (the letter) and a count for that taxon. Again, this sounds like a dictionary.

So we need a dictionary keyed by each site name, for which the associated value is another dictionary, keyed by the taxon IDs with values that are the counts for that site. So we need to create a dictionary of dictionaries. As with the whole program, it's probably best to start simple.

We need to create the top-level dictionary before we can populate it with the data from the file. We do this by defining an empty dictionary. You can do this by putting the line

```
In [116]: sites = {}
```

just before the start of the loop that reads the file. This is often referred to as "initialising" a data structure, and is a strategy that you will use a lot when working with data read into Python from other sources. Now every time you find the name of a new site in the file, you need to create the entry in this dictionary for that site name. Again, the value associated with this site name needs to be a new, empty dictionary. The example below shows how you can extract the site name from a line and create a new dictionary for it.

```
In [117]: datafile = open('speciesDistribution.txt', 'r')
          for line in datafile:
              line = line.strip()
              if line.startswith('Site: '):  # you should have come up with someth
                  siteName = line[6:]        # this in your solution to exercise .
                  sites[siteName] = {}
```

Exercise 4.2 Change your program to create the empty dictionaries as above, then right at the end, outside the loop, get it to print out the keys for the sites dictionary. These should be all of the site names.

Splitting Lines and Converting Data Now that we are creating a dictionary for each site, we need to parse the taxon/count lines from the file and put them into the appropriate dictionary for their site. As is common, on these lines we have two items of data. (We know too that once we see one of these lines we must also have the site name, which we have kept in a variable since it was extracted from the previously seen `Site:` line.)

We can split the line as we did before to get the separate fields. In this case there will be two fields and they are returned as a list, but Python allows us to unpack them directly into individual variables in the assignment statement if we want to. So, after inserting an `else:` statement to go with the `if` statement that contains the `.startswith()` test to find the site names, you could type (again with the appropriate level of indentation):

```
taxonID, count = line.split()
```

Just a few of words of caution. Firstly, this will split the string on all whitespace characters. This is fine in our case, but if any of your data were to contain spaces (for example if the single letter taxon names were classic binomial species names like *Homo sapiens* instead), they would be split too. You can limit to only split on tabs with:

```
taxonID, count = line.split('\t')
```

That solved the first problem. The second issue here is the data types. Type the following at the Python shell prompt:

```
In [118]: line = 'A\t29304'
          taxonID, count = line.split('\t')
          count

Out[118]: '29304'

In [119]: count = count + 99


        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-119-e0cb6e716f7d> in <module>()
    ----> 1 count = count + 99


        TypeError: must be str, not int


In [120]: count = 29304
          count

Out[120]: 29304

In [121]: count = count + 99
```

The first time that Python prints the value of count, it has quotation marks around it, and you get an error when you try to add 99 to it. The second time it doesn't have quotation marks and you don't receive an error when adding 99. This is because the first time, the value of count is not a number but a string representing the number. Perl programmers don't have to worry about this kind of thing, because Perl will automatically convert things for you when it thinks it needs to. With Python we have to be a bit more careful and convert the data ourselves. This is done with

```
In [122]: count = int(count)
          count
```

```
Out[122]: 29403
```

to convert to an integer and, if needed, you could convert it back again with:

```
In [123]: count = str(count)
          count

Out[123]: '29403'
```

Now when you add the lines to your program, you have variables containing the site name, the taxonID and the count (which you can now make sure is converted to a proper integer). You can put these into the dictionary of dictionaries like this:

```
In [124]: sites[siteName][taxonID] = count
```

In this statement, `sites[siteName]` refers to the dictionary we created for that site, so we append another subscript onto it to get a reference to the data item for this taxon in that site dictionary. Hopefully, that makes some sense. Take a look back at the discussion of nested dictionaries in Worksheet 3 if you need to. Now, finally, all of the data from the file is where we want it.

Exercise 4.3 Make the changes discussed above and make sure your program runs without errors. We will also need another change, to keep track of the names/IDs of taxa as we encounter them. At the top of the program, create a new empty list of taxon IDs e.g.,:

```
In [125]: sites = {}
          taxa = []
          datafile = open('speciesDistribution.txt', 'r')
          for line in datafile:
              line = line.strip()
              if line.startswith('Site: '):  # you should have come up with someth
                  siteName = line[6:]        # this in your solution to exercise .1
                  sites[siteName] = {}
              else:
                  taxonID, count = line.split('\t')
                  count = int(count)
                  if taxonID not in taxa:
                      taxa.append(taxonID)
                  sites[siteName][taxonID] = count

In [126]: sites

Out[126]: {'Grimston Wood': {'A': 123,
             'B': 1340,
             'C': 11984,
             'E': 9389,
             'F': 4320,
             'G': 1283,
             'J': 8193,
             'K': 193},
```

```
'Hagg Wood': {'A': 2039,
 'B': 9394,
 'C': 19380,
 'D': 9102,
 'E': 932,
 'G': 893,
 'H': 5839,
 'J': 9302,
 'L': 984},
'Hetchell Wood N': {'A': 12983,
 'B': 8493,
 'C': 948,
 'D': 9384,
 'E': 4942,
 'G': 9834,
 'I': 1293,
 'J': 9348},
'Hetchell Wood S': {'A': 9380,
 'B': 13928,
 'D': 949,
 'E': 19023,
 'F': 9384,
 'G': 948,
 'H': 9284,
 'J': 1093,
 'K': 3029},
'Scoreby Wood': {'A': 920,
 'B': 3928,
 'D': 9301,
 'E': 19384,
 'F': 12949,
 'H': 3892,
 'I': 9192,
 'K': 912},
'Sutton Wood': {'A': 883,
 'B': 293,
 'C': 893,
 'D': 18990,
 'F': 3910,
 'G': 930,
 'H': 1738,
 'I': 819,
 'M': 9934},
'Wheldrake Wood': {'B': 91,
 'C': 22649,
 'D': 2949,
 'E': 901,
 'G': 9204,
```

```
'H': 2040,
'I': 8173,
'L': 6781,
'M': 9184}}
```

Now, when you add a count to the dictionary of dictionaries, check if the taxon ID is in this new list and add it if not (just like you did when merging the shopping lists in Worksheet 2). We will then have a non-redundant list of taxon names to play with in a minute.

Filling in the Blanks Unfortunately, there is a problem with this data. Some of the taxa were not detected at every one of the sampled sites, so the data for these sites do not include counts associated with those taxa. This means that if we were, say, to plot the data in bar charts, some would have fewer bars than others or the bars would be in different positions, rather than just having a gap (or zero-height bar) where the taxon wasn't found. What you need to do to avoid this is create new entries with counts of zero for the missing taxa at each site.

Exercise 4.4 Put the zero values in the data structure. To do this you will need to loop through the sites, and for each site, loop through the IDs in the full, non-redundant taxon list and if a taxon ID is not in the keys of the dictionary for the site, add it with an associated count of zero. Then you will need to check your program is working correctly. A good way to do that is described in the next section.

```
In [127]:
```

Formatting Data Structures When you are building up data structures like this, they can get very complex and it's difficult to keep track and be sure that you are putting everything in the right place. Fortunately, there is a Python module (part of the standard library), which lets you print out the data in a comprehensible way. Of course, you could just print the entire data structure in one statement and this works, but it can be hard to read - there is no formatting at all - and it often doesn't really help. The `pprint` module formats the data in a hierarchical way, making it easier to understand. At the top of your program, you need to import the `pprint` module with:

```
In [127]: import pprint
```

You then create a formatter that will do the work for you with:

```
In [128]: pp = pprint.PrettyPrinter(indent=4)
```

Now, when you want to check a data structure, you can just do the following and get a nice readable printout of your data structure:

```
In [129]: # this is the dictionary from Worksheet 2
          studentNumbers = { 'Bioscience Technology': 16,
                             'Computational Biology': 12,
                             'Post-Genomic Biology': 20,
                             'Ecology and Environmental Management': 3,
                             'Maths in the Living Environment': 0
                           }
          variable = studentNumbers
          pp.pprint(variable)
```

```
{   'Bioscience Technology': 16,
    'Computational Biology': 12,
    'Ecology and Environmental Management': 3,
    'Maths in the Living Environment': 0,
    'Post-Genomic Biology': 20}
```

Compare this output to the way that the same dictionary is displayed by the default `print` function:

```
In [130]: print(studentNumbers)

{'Bioscience Technology': 16, 'Computational Biology': 12, 'Post-Genomic Biology':
```

I hope you'll agree that the `pprint` version is much easier to interpret by eye.

Exercise 4.5 Use the `pprint` module to dump out the contents of you data structure and check that the data corresponds with what you thought it should look like.

Plotting Data There are a number of options available for plotting data in Python. For many years the standard approach was to use a library called `pyplot` from the module `matplotlib`, which closely resembles the plotting interface of the mathematical programming language *MatLab*.

To start plotting the data, you will need to use a couple of new modules installed which don't come with the standard installation of Python. The first is `numpy`, which defines a new array data type which you can use in ways that will be familiar if you use *R* or *MATLAB* and is really just there because the second, `matplotlib`, relies heavily on it. `matplotlib` gives a whole range of graph plotting functions again similar to the facilities in *MATLAB*. We won't use `numpy` directly, but I encourage you to play with `matplotlib`, and in particular the `pyplot` parts of it.

If you're using the Anaconda distribution of Python, then you already have all of the modules that are needed for the course. If not, then you might need to install the modules before you can follow the rest of the material. A quick guide to installing these is given below.

A couple of notes before we begin:

- **If you are using the Anaconda Python distribution you don't need to follow the next few steps!**
- To install modules, you will need to have administrator priviledges for the computer that you're working on.
- If at any point you are unsure about how to follow these instructions, you should ask for help.

First of all, you should make sure that you have `pip` installed. To do this, you need to open a terminal/command prompt (*not the Python shell*) on your system (Applications -> Terminal on Mac) and type

```
pip help
```

If you have `pip` installed, you should see some helpful output listing all the available options for running the package manager. If not, you will get an (equally helpful) error message. To install `pip` go here and follow the instructions.

To install the modules that we need with `pip`, you simply have to run the commands

```
pip install numpy
pip install matplotlib
pip install bokeh
```

at the command line and respond to any prompts from the package manager. That's it. (If you are working on a different operating system and/or distribution, ask for help and we will find a way for you to install the packages that you need.) Now you can return to the Python shell and type:

```
In [131]: import numpy as np
          from matplotlib import pyplot as plt

          % matplotlib inline
          # The above line is a 'magic' line for the Jupyter notebook which allows
```

These statements make the functions, variables and classes of the two modules available to your program, but keep them at arms length, in their own 'namespaces'. This is to make sure that none of the names they use clash with anything in your program. It does mean that you have to type the prefix `np.` or `plt.` when you need to call them, but that's not much of a price to pay for the safety that namespaces give you. If you are feeling reckless, you could, for example, have typed

```
from numpy import *
```

and then you wouldn't have to deal with the prefix. This might seem like a good idea, but in fact it is quite risky. Our advice is not to do that. *Ever.* One day it will trip you up and it will take weeks to find out exactly what you have done wrong. Not that I'm talking from experience, or anything...

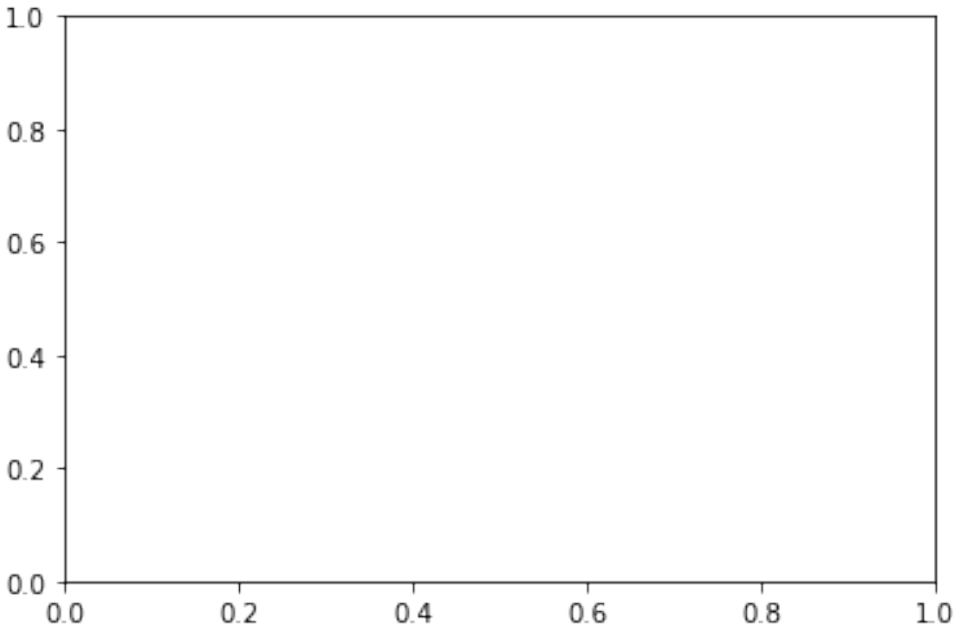
Anyway, we now have the modules loaded and ready to go, so we can get on with trying to plot a bar chart of our data. We will generate some fairly pretty plots. `pyplot` is a bit unusual as a Python module, because it doesn't define objects and methods for you, it's just a set of function calls. Normally, you might expect to create a figure object, then use it's methods, to e.g., add data or change the layout. However, `pyplot` just remembers what you are doing and performs on the last figure or subplot that you used. This type of interface is called "stateful". If you really don't like working this way, there is an object interface as well, but the documentation isn't quite as good for it.

In our case, though, we want to start by creating a new figure (this is actually optional, but good practice - Python says "Explicit is better that implicit" - Try: `import this`).

```
In [132]: plt.figure(1)

Out[132]: <matplotlib.figure.Figure at 0x109d8dd68>

<matplotlib.figure.Figure at 0x109d8dd68>
```

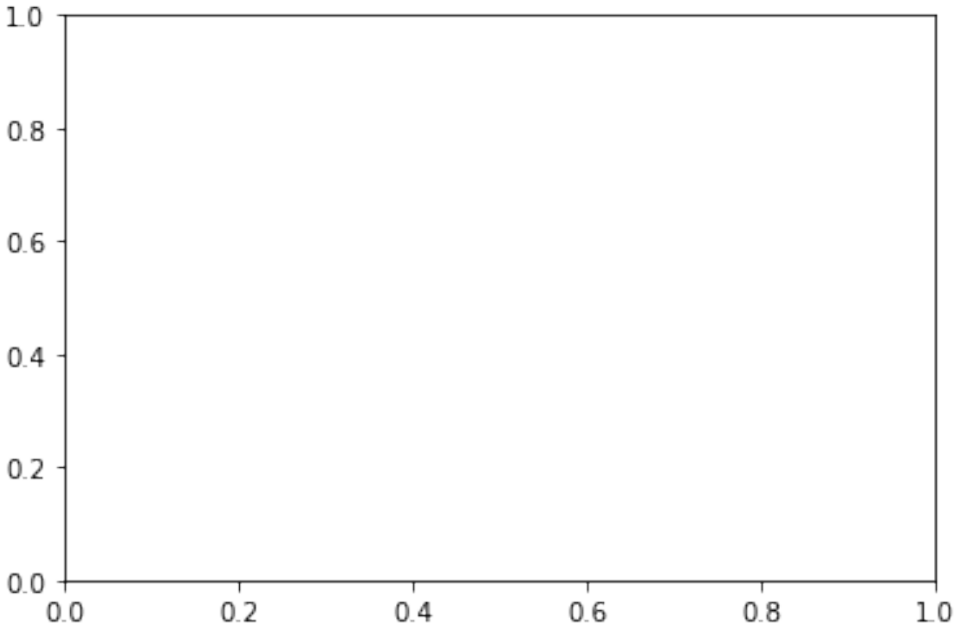
As soon as you executed this line, a new window might have appeared (it depends on your operating system) and this is where the figure will be drawn. Don't worry if the window didn't appear at this stage - it should show up when you're done building the figure instead. Inline plotting is switched on in the IPython Notebook, so as we add things to the plot, you will see them appear below. The number argument to `plt.figure` is just a reference and lets you switch back to this figure later if you need to. The next step is to create a subplot. This is mostly used for figures with multiple panels, where each panel is a subplot. In this case, we'll just do one for now, so we can just type one of the following statements:

```
In [133]: plt.subplot(1,1,1)

Out[133]: <matplotlib.axes._subplots.AxesSubplot at 0x1106bad68>
```



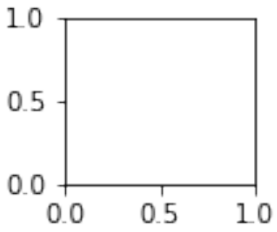
or

```
In [134]: plt.subplot(111)

Out[134]: <matplotlib.axes._subplots.AxesSubplot at 0x1106cf780>
```



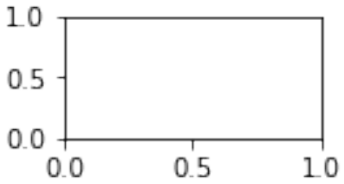
These are again optional if you only have one panel in your figure, but again it's generally better to be explicit. The arguments for the second form are (rows, columns, subfig), specifying the number of rows of panels, the number of columns and which one you want to draw now (which is in the range 1 - rows*colums). So if you have twelve panels and you wanted to select the seventh, you would use one of

```
In [135]: plt.subplot(3,4,7)

Out[135]: <matplotlib.axes._subplots.AxesSubplot at 0x113ad3d68>
```



or

```
In [136]: plt.subplot(4,3,7)

Out[136]: <matplotlib.axes._subplots.AxesSubplot at 0x113ae3e10>
```



depending on how many rows and columns you used. Subfigures/subplots are numbered as if read like text: left-right and top-bottom. If rows*columns is less than 10, you can use the top form without the commas e.g. `(321)` instead of `(3,2,1)`, but you might consider this to be less explicit.

Simple matplotlib plots Now, we can make up some data and plot it as a bar chart. The first thing is to create a list containing the heights of some bars, for which we can use:

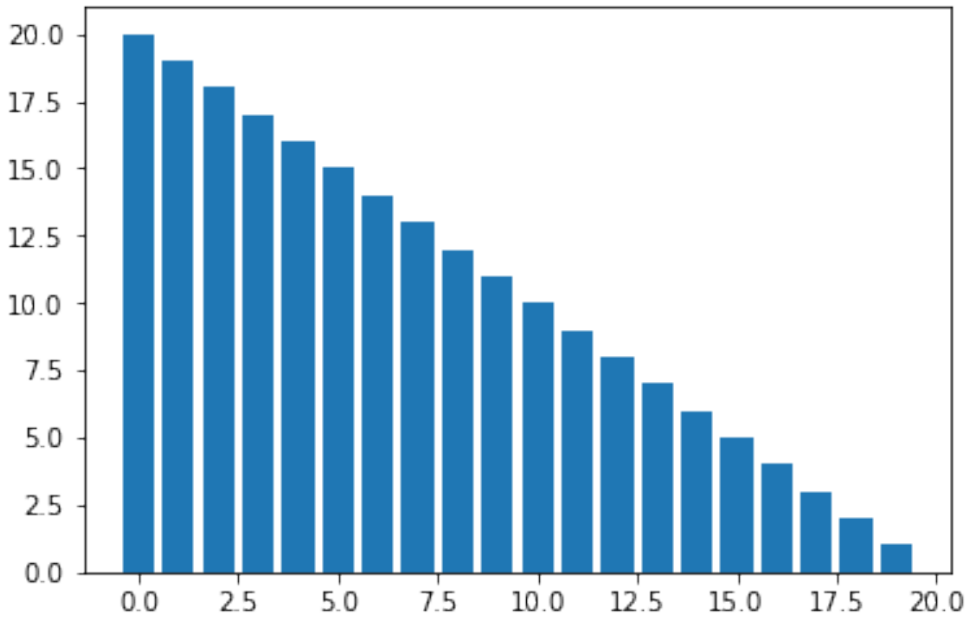
```
In [137]: barHeights = range(20,0,-1)
```

We also need to specify where each of the bars will be drawn. This is the x-coordinate of the bottom left corner of the bar. For this we will use:

```
In [138]: positions = range(20)
```

Now we can draw the bar itself with

```
In [139]: plt.bar(positions, barHeights)

Out[139]: <Container object of 20 artists>
```



```
In [140]: plt.show() # you might need to run this line for your plot to appear
```

and the chart should appear in the window. You will notice that the figure has rescaled to fit the data and the axes are just labelled with numbers. There are lots of ways to tweak the figures, such as the `plt.xlabel()` and `plt.ylabel()` methods, which put titles on the axes, or `plt.title()`, which puts a title on the plot.

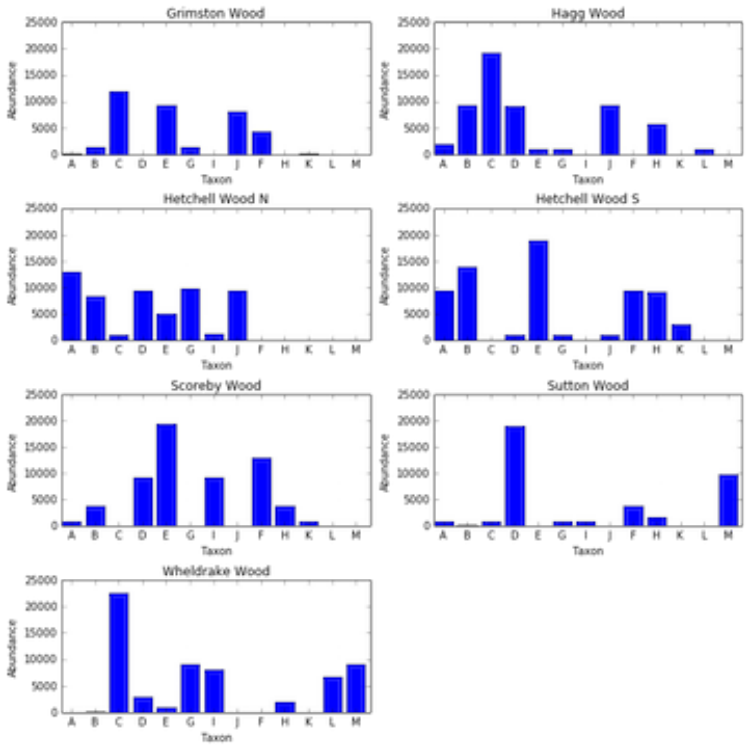
Of course, what we really want to do is plot the data from our file of sites and taxa. To make it as easy as possible to compare the distributions at the different sites, we want to plot all of the data in a single figure. To help us do this, we are going to define a function to plot the figure, then we can pass the data for each figure into the function, one site at a time. We haven't done this before, so I will give you the code for the function:

```
In [141]: def doBarChart(heights, labels, title, rows, columns, subplot):
              plt.subplot(rows, columns, subplot)
              plt.bar(range(len(labels)), heights)
              plt.title(title)
              plt.xlabel('Taxon')
              plt.ylabel('Abundance')
              plt.axis([0, len(labels), 0, 25000])
              tickPos = []
              for pos in range(len(labels)):
                  tickPos.append(pos+0.4)
              plt.xticks(tickPos,labels)
```

This function takes a list of bar heights, a list of labels, a string to use as the title, then the number of rows and columns and the subplot number. First, the function passes these last three to `plt.subplot()` to initialise a new subplot on the figure. It then draws the bar chart, adds the title, labels the axes, and sets the minimum and maximum values for the axes. The last fiddly bit is to create a new list for the labels. By default, when you draw a bar chart, the widths of the bars are 0.8 units. So, we create a new list of positions which are offset by 0.4 units, so that the labels will centred relative to the bars. You can now add this function definition to the top of your program file along with the import statements and you will have everything you need to create the multi-panel figure.

Exercise 4.6 Plot all of the data that you read from the file earlier into a single figure of bar plots for each site. This is challenging, but take it bit by bit and you should be able to do it.

- You will need to start by calling `plt.figure(1)`, then start a loop over the sites.
- In that loop, you need to
- gather the data for the bar heights (the counts),
- then call the `doBarChart` function, passing in the row heights, labels, the site name as a title and the number of rows, columns and subplot number.
- Once you have it working like that, try changing the program so that the sites and taxon IDs are in alphabetical order.
- Then use the plot configuration window to tidy up the figure and save it.



Example output from Exercise 4.6

You should be able to get it to look as in the figure below.

Once you have got this to work, or if you get really stuck, you can check out a solution to the exercise above in this notebook.

Exercise 4.7 *Note that this exercise is optional.* The data that we read in and plotted in the first parts of this chapter was not in the ideal format. This is deliberate, to provide you with practice in choosing appropriate data structures and reading data from a file. Unfortunately, it's also not uncommon to be provided with data in a weird format that can be inconsistent and annoying to read (*parse*) programmatically.

With counts collected across a set of site names, for a set of taxa, it would be more sensible to store these data in a tabular format. Now that we have gone to the trouble of reading the data in and adding zero counts where appropriate, let's write a new file with the data arranged in a table.

To write a new file, we use the same `open` function that we read the data in with, but with `'w'` as the mode instead of `'r'`. The tabular file should have the structure:

```
taxonID  site1   site2   site3   ...   siteN
taxon1   count   count   count   ...   count
taxon2   count   count   count   ...   count
...      ...     ...     ...     ...   ...
taxonM   count   count   count   ...   count
```

where each field on a row is separated by a `\t` tab character.

Rearrange the lines of code below to write the collected data to a file. (Be careful to observe and preserve the indentation levels.)

```
In [142]: outfh.close()
          outfh.write('taxonID\t')
          for tax in sorted(taxa):
          sitenames = list(sites.keys())
          outfh.write('\t'.join(sitenames))
          outfh.write('\n')
          outfh = open('speciesDistribution_tabular.txt', 'w')
              outfh.write('\t'.join([str(sites[site][tax]) for site in sitenames]))
              outfh.write('{}\t'.format(tax))
              outfh.write('\n')
          sitenames.sort()


            File "<ipython-input-142-706fe20f1be2>", line 4
          sitenames = list(sites.keys())
                  ^
      IndentationError: expected an indented block
```

Using pandas for tabular data Now that we have the data in a tabular file (you can download the tabular version here if you chose not to complete the exercise above), we can take advantage of the `pandas` library. `pandas` is one of the most commonly-used libraries for data analysis in Python, and has been designed to handle, analyse, and visualise tabular data quickly and easily. If you have data in tabular format, you can avoid the process of building up complex data structures that we were dealing with earlier, load in your data table, and jump straight to organising, filtering, summarising, etc.

First, we need to import `pandas`, which we will do using the standard shortened version of the namespace:

```
In [143]: import pandas as pd
```

This allows us to access all of the functions and object classes included in the library, via the `pd` namespace. One function included in `pandas`, which we will use now, is `read_table`. We will use this to read our tabular data file into Python, storing it in memory as a `pandas.DataFrame` object:

```
In [144]: data = pd.read_table('speciesDistribution_tabular.txt')
          print(data)

   taxonID  Grimston Wood  Hagg Wood  Hetchell Wood N  Hetchell Wood S  \
0        A            123       2039            12983             9380
1        B           1340       9394             8493            13928
2        C          11984      19380              948                0
3        D              0       9102             9384              949
4        E           9389        932             4942            19023
5        F           4320          0                0             9384
6        G           1283        893             9834              948
```

```
7         H              0       5839                0              9284
8         I              0          0             1293                 0
9         J           8193       9302             9348              1093
10        K            193          0                0              3029
11        L              0        984                0                 0
12        M              0          0                0                 0

    Scoreby Wood  Sutton Wood  Wheldrake Wood
0            920          883               0
1           3928          293              91
2              0          893           22649
3           9301        18990            2949
4          19384            0             901
5          12949         3910               0
6              0          930            9204
7           3892         1738            2040
8           9192          819            8173
9              0            0               0
10           912            0               0
11             0            0            6781
12             0         9934            9184
```

As the output above shows, we now have the full table stored with the variable name `data`. However, to really be able to easily access the data, we need to specify the column headers and row names. This will allow us to access individual rows, columns, or data points, as we were doing before via `sites[sitename]`, `sites[sitename][taxonID]` and so on. We could set the column and row names with the current `data` object, but it's actually easier to simply set them when we first read the table in, so let's edit and re-run our call to `read_table`:

```
In [145]: data = pd.read_table('speciesDistribution_tabular.txt', header=0, index_c
          print(data)

          Grimston Wood  Hagg Wood  Hetchell Wood N  Hetchell Wood S  \
taxonID
A                   123       2039            12983             9380
B                  1340       9394             8493            13928
C                 11984      19380              948                0
D                     0       9102             9384              949
E                  9389        932             4942            19023
F                  4320          0                0             9384
G                  1283        893             9834              948
H                     0       5839                0             9284
I                     0          0             1293                0
J                  8193       9302             9348             1093
K                   193          0                0             3029
L                     0        984                0                0
M                     0          0                0                0
```

```
          Scoreby Wood  Sutton Wood  Wheldrake Wood
taxonID
A                  920          883               0
B                 3928          293              91
C                    0          893           22649
D                 9301        18990            2949
E                19384            0             901
F                12949         3910               0
G                    0          930            9204
H                 3892         1738            2040
I                 9192          819            8173
J                    0            0               0
K                  912            0               0
L                    0            0            6781
M                    0         9934            9184
```

We don't have time to go into detail about all the many wonderful things that you can do with `pandas` and dataframes. For a more comprehensive introduction to the subject, I recommend the Software Carpentry lesson "Plotting and Programming in Python", available at swcarpentry.github.io/python-novice-gapminder/.

Here, it suffices to say that we can access individual columns of the dataframe similarly to accessing values in a dictionary, by providing the column title in square brackets:

```
In [146]: data['Grimston Wood']
```

```
Out[146]: taxonID
          A      123
          B     1340
          C    11984
          D        0
          E     9389
          F     4320
          G     1283
          H        0
          I        0
          J     8193
          K      193
          L        0
          M        0
          Name: Grimston Wood, dtype: int64
```

Note, that we get the row names alongside the count values for our chosen site. If you wanted to, you could access a specific count by also providing the taxon ID:

```
In [147]: print(data['Hagg Wood']['G'])
          print(data['Hagg Wood']['F'])
          print(data['Hagg Wood']['B'])
```

```
893
0
9394
```

To recreate the plots that we were producing before with matplotlib, we will need to access all of the columns and rows, so I will also show you how to achieve that. The dataframe object has `index` and `columns` attributes that carry the row and column names respectively:

```
In [148]: data.index

Out[148]: Index(['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M'],

In [149]: data.columns

Out[149]: Index(['Grimston Wood', 'Hagg Wood', 'Hetchell Wood N', 'Hetchell Wood S'
                 'Scoreby Wood', 'Sutton Wood', 'Wheldrake Wood'],
                dtype='object')
```

We can loop through these `Index` type objects just like we were doing with the lists and dictionary keys earlier. Now that we've covered another way of handling data, it's time to introduce you to an alternative plotting library.

Alternatives to matplotlib The `matplotlib` library is very powerful and extremely flexible, and it is still widely used, but I find the interface a little hard to work with and it is often confusing for beginners. Over recent years, the range of options for plotting data in Python has expanded, with several new modules being introduced that make it easy to create many standard types of plot. (For a great overview of the many options for plotting in Python, I recommend Jake VanderPlas' talk, "Python Visualisation Landscape" from PyCon 2017 - available on YouTube at https://www.youtube.com/watch?v=FytuB8nFHPQ.)

One example is the `bokeh` plotting library, which can be used to create attractive, interactive plots that render in HTML.

Simple bokeh Plots To provide a comparison of the two plotting libraries, we'll use one of `bokeh`'s methods with our `pandas` dataframe to recreate the plots above.

First, we need to import the relevant functions from the `bokeh.plotting` module:

```
In [150]: from bokeh.plotting import figure, show, output_notebook, output_file
```

Note that I have imported by `output_file` and `output_notebook` in the line above. We only need to use one of these, but which one will depend on the interface that you're working in. I will use `output_notebook` here because it allows me to plot the results directly in the Jupyter Notebook interface that this course was written in. If you're using a text editor, IDE such as Spyder, the Python shell, etc, you should use `output_file`, which will create an HTML file containing your plot(s).

If not using Jupyter, you can call `output_file` as demonstrated below:

```
output_file('myFirstPlot.html') # call the file whatever you like, but you should u
```

After you have prepared your data (as we did above with `pandas`), plotting with `bokeh.plotting` revolves around three steps: creating a Figure object; calling a plotting method on that object to add data points to the figure; calling the `show` function to render the figure. The second step can be performed multiple times, to add multiple sets/types of data points to the same figure.

To demonstrate this, let's create a single plot of one of the sites described in our dataframe.

```
In [151]: output_notebook()

          p = figure(plot_width=600, plot_height=400, x_range=list(data.index))
          p.vbar(x=[n+0.5 for n in range(13)],
                 width=0.8,
                 bottom=0,
                 top=data['Wheldrake Wood'],
                 color="firebrick")

          show(p)
```

Great! We've produced our first `bokeh` figure! You might have noticed that the scaling of axes has been taken care of for you, and there is a toolbar along the top of the plot. Included in these tools is panning and scroll zooming, alowing you to zoom in and out of the plot and navigate around to better interrogate plotted data. This can be really helpful, but can also get a little annoying when we want our view of the data to remain static. That's ok: it's really easy to switch off:

```
In [152]: p = figure(plot_width=600, plot_height=400, x_range=list(data.index), too
          p.vbar(x=[n+0.5 for n in range(13)],
                 width=0.8,
                 bottom=0,
                 top=data['Wheldrake Wood'],
                 color="firebrick")

          show(p)
```

In the codeblock above, we first created a figure, `p`, with set dimensions and specified the range on the x-axis to be the rownames (index) from our dataframe.

Next, we called the `vbar` method of `p`, to draw vertical bars on the figure. This method is given multiple arguments, which are covered individually below:

- the mid-points of the bars are specified via the `x` argument: `x=[n+05 for n in range(13)]`. The value provided to `x` is a *list comprehension* - a list of numbers created on the fly using the `range` function discussed earlier. To see why this list comprehension is necessary, try replacing it with only a call to `list(range(13))`.
- `width` specifies the desired width of the bars. This is similar to the matplotlib version we have already seen.
- `bottom` specifies the lower boundary of each bar. We set this to zero.
- `top` specifies the upper boundary of each bar. This is where we provide the count values for our site, by accessing the approriate column of the dataframe.

- `color` allows us to set the color of the bars. We could provide multiple values in a list if we wanted to.

Once the bars have been added to the figure, we call `show` to render the plot. Hopefully, when you ran this you saw a similar output to the plot above.

Exercise 4.8 Try adjusting the code above to add an appropriate title to the plot. Can you change the colour of the bars?

Now that you have an example of plotting with `bokeh`, let's look at how to arrange multiple plots into a specific layout. `bokeh` provides several functions and objects for creating and handling plot layouts, which are collected in the `bokeh.io` module. Here, we will only use one of them:

```
In [153]: from bokeh.io import gridplot
```

`gridplot` can be used to arrange multiple plots in a grid. The function takes lists of figures as arguments, with each list rendered as a row of plots in the grid. To demonstrate this let's create a few simple plots, using some of the other plotting methods of the Figure object.

```
In [154]: fig1 = figure(plot_width=300, plot_height=300)
          fig1.circle([1, 2, 3, 4, 5], [3, 1, 2, 1, 2], size=20, color="navy", alph

          fig2 = figure(plot_width=300, plot_height=300)
          fig2.line([1, 2, 3, 4, 5], [3, 1, 2, 1, 2], line_width=5, color="coral",

          fig3 = figure(plot_width=300, plot_height=300)
          fig3.hbar(y=[3, 2, 1], height=0.5, left=0, right=[1.2, 3.7, 2.7], color='

          layout = gridplot([fig1, fig2], # two rows, with fig1 and fig2 on the fi
                            [fig3])       # and fig3 on the second

          show(layout) # render the layout
```

Feel free to play around with other layouts available in `bokeh.io`, with other plotting options in `bokeh.plotting`, and with the arguments to the plotting methods etc. You can find the documentation for `bokeh` at bokeh.pydata.org/, which should help as a reference and guide for what's possible. It's time for one final challenge exercise, to pull together what you've learned about `pandas`, `bokeh`, and Python in general :)

Exercise 4.9 Similarly to what you did before with `matplotlib`, plot all of the data that you read from the file earlier into a single file of bar plots, one for each site. This is challenging, but take it bit by bit and you should be able to do it. You might need to refer to the `help()` documentation and the online user guide linked above for the plotting functions to achieve everything listed above.

If you are really adventurous, plot all of the data on a single set of axes, with the data interleaved and the bars for different sites in different colours.

After you have finished on this exercise, or if you get really stuck and need to look at a solution, take a look at this notebook, which runs through my way of producing the site plots with a different bar color for each taxon.

Summary

- Files are opened with the `open()` command, and this returns a file object.
- Methods of the file object, such as `.readline()` or `.readlines()` can be used to get data from the file.
- Files can also be used as iterable data type in `for` statements (and other contexts).
- Python doesn't convert data types automatically, so you need to use functions like `str()` and `int()` to convert between strings and numbers.
- Python modules provide additional functionality for the language, and can perform many common data analysis tasks.

Chapter 5

Walkthrough: Exercise 4.6 - Plotting with `matplotlib`

This notebook runs briefly through a solution to the exercise at the end of Worksheet 4 of the *Introduction to Python Programming* course workbook.

Start by importing the additional functionality that you need, to print the data structure and draw the plots.

```
In [155]: import pprint
          import matplotlib.pyplot as plt

          # the next line is for plotting in the IPython notebook only
          %matplotlib inline
```

Now define the function that will draw each barchart of observations. (This function definition contains a list of colors for the bars, as tuples of RGB values, that will plot each species in a different color.)

```
In [156]: def doBarChart(heights, labels, title, rows, columns, subplot):
              plt.subplot(rows, columns, subplot)
              plt.bar(range(len(labels)), heights)#, color=[(0.9,0.05,0.05),(0.35,1
              plt.title(title)
              plt.xlabel('Taxon')
              plt.ylabel('Abundance')
              plt.axis([0, len(labels), 0, 25000])
              tickPos = []
              for pos in range(len(labels)):
                  tickPos.append(pos+0.4)
              plt.xticks(tickPos,labels)
```

Initialise an empty dictionary, to store the data for each site, and an empty list, to store the species names as you find them. Then, open a file object and read the data line-by-line, populating the data structures as you go.

```
In [157]: sites = {}
          taxa = []
```

```
# make sure that you change the file path below to specify the location o
datafile = open('speciesDistribution.txt', 'r')

for line in datafile:
    line = line.strip()
    if line.startswith('Site:'):
        tag, siteName = line.split(" ", 1)
        sites[siteName] = {}
    else:
        taxonID, count = line.split()
        count = int(count)
        sites[siteName][taxonID] = count
        if taxonID not in taxa:
            taxa.append(taxonID)
```

Now that you have read all of the data from the file, you need to loop over each site again, to add zero values for each species not observed. (Exercise 4.4)

```
In [158]: for site in sites:
              for taxon in taxa:
                  if taxon not in sites[site]:
                      sites[site][taxon] = 0
```

Now, you should have a dictionary, `sites`, keyed by site names, with values that are themselves dictionaries keyed by species ID. Each of these dictionaries within `sites` should have the same number of entries because you added the zero counts. To check out the overall data structure, you can use `pprint.PrettyPrinter`. (Exercise 4.5)

```
In [159]: pp = pprint.PrettyPrinter(indent=4)
          pp.pprint(sites)
```

```
{   'Grimston Wood': {   'A': 123,
                         'B': 1340,
                         'C': 11984,
                         'D': 0,
                         'E': 9389,
                         'F': 4320,
                         'G': 1283,
                         'H': 0,
                         'I': 0,
                         'J': 8193,
                         'K': 193,
                         'L': 0,
                         'M': 0},
    'Hagg Wood': {   'A': 2039,
                     'B': 9394,
                     'C': 19380,
                     'D': 9102,
                     'E': 932,
```

```
                  'F': 0,
                  'G': 893,
                  'H': 5839,
                  'I': 0,
                  'J': 9302,
                  'K': 0,
                  'L': 984,
                  'M': 0},
 'Hetchell Wood N': {   'A': 12983,
                        'B': 8493,
                        'C': 948,
                        'D': 9384,
                        'E': 4942,
                        'F': 0,
                        'G': 9834,
                        'H': 0,
                        'I': 1293,
                        'J': 9348,
                        'K': 0,
                        'L': 0,
                        'M': 0},
 'Hetchell Wood S': {   'A': 9380,
                        'B': 13928,
                        'C': 0,
                        'D': 949,
                        'E': 19023,
                        'F': 9384,
                        'G': 948,
                        'H': 9284,
                        'I': 0,
                        'J': 1093,
                        'K': 3029,
                        'L': 0,
                        'M': 0},
 'Scoreby Wood': {   'A': 920,
                     'B': 3928,
                     'C': 0,
                     'D': 9301,
                     'E': 19384,
                     'F': 12949,
                     'G': 0,
                     'H': 3892,
                     'I': 9192,
                     'J': 0,
                     'K': 912,
                     'L': 0,
                     'M': 0},
 'Sutton Wood': {   'A': 883,
```

```
                        'B': 293,
                        'C': 893,
                        'D': 18990,
                        'E': 0,
                        'F': 3910,
                        'G': 930,
                        'H': 1738,
                        'I': 819,
                        'J': 0,
                        'K': 0,
                        'L': 0,
                        'M': 9934},
    'Wheldrake Wood': {   'A': 0,
                          'B': 91,
                          'C': 22649,
                          'D': 2949,
                          'E': 901,
                          'F': 0,
                          'G': 9204,
                          'H': 2040,
                          'I': 8173,
                          'J': 0,
                          'K': 0,
                          'L': 6781,
                          'M': 9184}}
```

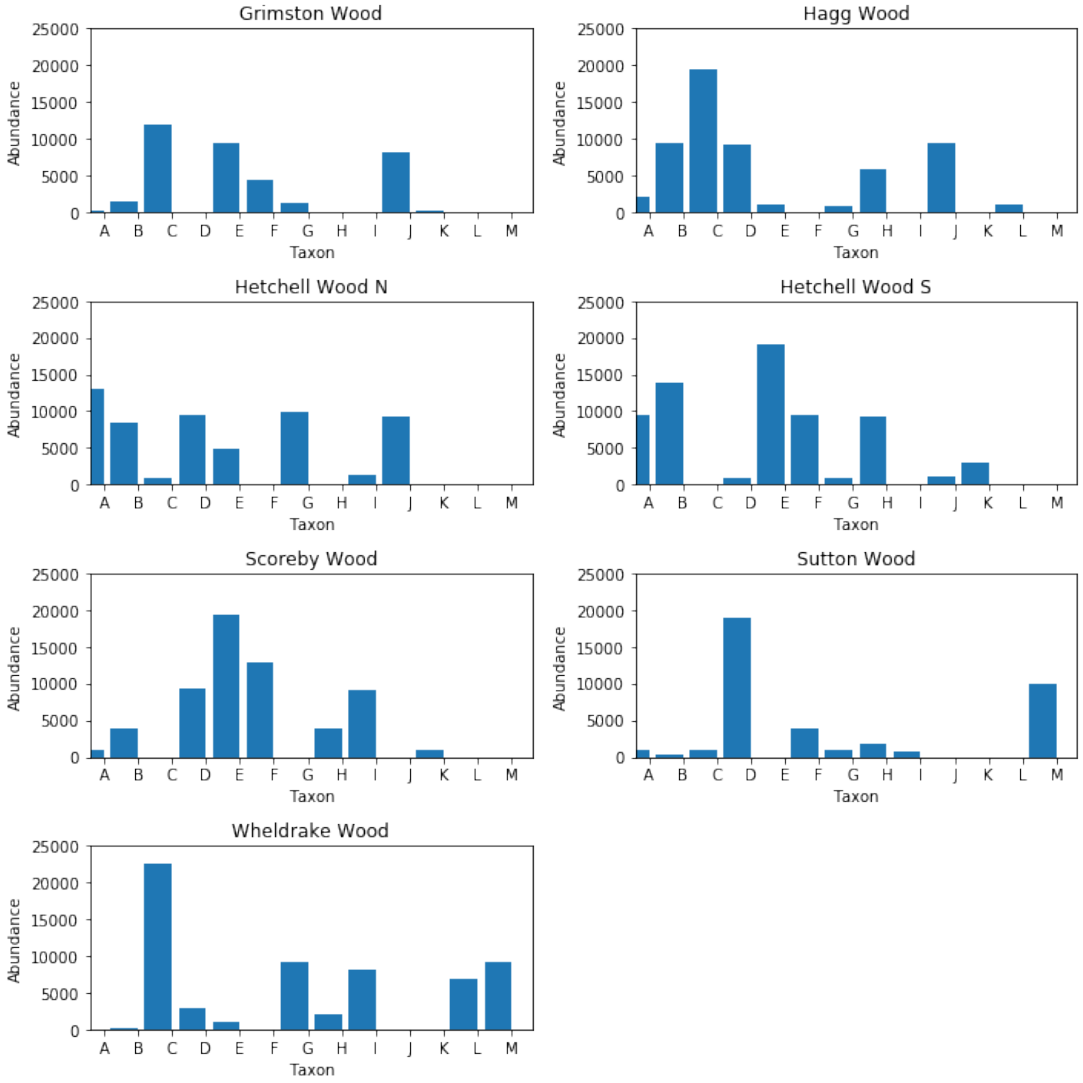
Looks good to me! Now, you want to be able to control the order in which the species and counts are extracted from this dictionary for plotting. To do this, you use the dictionary keys and the list of species names that were compiled when the data was read from the file. To make the plot more intuitive, you can make sure that both of these are sorted alphabetically.

```
In [160]: taxa.sort()
          siteNames = sites.keys()
          siteNames = sorted(siteNames)
```

Now, you are ready to plot the data using the function defined above. (Exercise 4.6)

```
In [161]: plt.figure(1, figsize=(10,10)) # you can choose your own figure size or l
                                         # this argument out to use the default set

          subnumber = 0

          for site in siteNames:
              subnumber += 1
              barValues = []
              for taxon in taxa:
                  barValues.append(sites[site][taxon])
              doBarChart(barValues, taxa, site, 4, 2, subnumber)
```

```
plt.tight_layout()
plt.show()
```



Chapter 6

Walkthrough: Exercise 4.9 - Plotting with Bokeh

First things first, we need to import the functions that we need from the third-party libraries, Pandas and Bokeh. We import all of Pandas, with the abbreviated namespace `pd`, and `figure`, `show` and `output_notebook` from `bokeh.plotting`. We use `output_notebook` here, to allow the plot to be displayed in this Jupyter Notebook - you might want to use `output_file` instead, which will result in the figure being rendered in an HTML file instead (see below). `figure` is used to construct the figure, and `show` is used to render the layout in the end. Speaking of layout, we import `gridplot` from `bokeh.io`, to create rows and columsn of the figures after we've created them.

```
In [162]: import pandas as pd
          from bokeh.plotting import figure, show, output_notebook
          from bokeh.io import gridplot
```

Now, we need to read the data from the file into Python. Unlike in the matplotlib example, we will use a Pandas `DataFrame` here. (Note that this approach would work with matplotlib too - we're just choosing to introduce it at the same time as we start doing things with Bokeh.) We use the `read_table` function, with the `index_col` and `header` arguments, to read the tabular data into a dataframe.

```
In [163]: inputFile = 'speciesDistribution_tabular.txt'
          data = pd.read_table(inputFile, header=0, index_col=0)
          data
```

Out[163]:

taxonID	Grimston Wood	Hagg Wood	Hetchell Wood N	Hetchell Wood S	\
A	123	2039	12983	9380	
B	1340	9394	8493	13928	
C	11984	19380	948	0	
D	0	9102	9384	949	
E	9389	932	4942	19023	
F	4320	0	0	9384	
G	1283	893	9834	948	
H	0	5839	0	9284	

```
I                    0          0             1293                0
J                 8193       9302             9348             1093
K                  193          0                0             3029
L                    0        984                0                0
M                    0          0                0                0

          Scoreby Wood  Sutton Wood  Wheldrake Wood
taxonID
A                  920          883               0
B                 3928          293              91
C                    0          893           22649
D                 9301        18990            2949
E                19384            0             901
F                12949         3910               0
G                    0          930            9204
H                 3892         1738            2040
I                 9192          819            8173
J                    0            0               0
K                  912            0               0
L                    0            0            6781
M                    0         9934            9184
```

Great! Now we have all of our data loaded and arranged in the way that we need, we can move onto plotting.

Now we use `output_notebook()` to make sure that our plots can be displayed in this Notebook.

```
In [164]: output_notebook()
```

Note: to save the plot to a file instead, use:

```
output_file('my_amazing_plot.html') # give files descriptive names so you can easil
```

Now we can start creating the indivual bar plots, one for each site. We want to display these plots all together, so we should start by storing them in a list. For each site we pull out the bar heights - this is the dataframe column corresponding to the site name; and the taxon names as a list created from the index column of the dataframe.

Next, we create a figure object for the site, using the site name as the title, and setting the range for the x-axis as the taxon names. `tools=[]` disables the interactive tool bar for these plots. Finally, we call the `figure` object's `vbar` method to add the bars to the figure axes. The mid-points of the bars on the axis need to be labelled with the taxon names, so we construct a list comprehension* adding 0.5 to the index numbers along the axis. The heights of the bars are given as the column that we pulled out earlier. Once the figure is constructed, we add it to the list of plots.

```
In [165]: plots = []
          for site in data.columns:
              heights = data[site]
              taxa = list(data.index)
```

```
            fig = figure(plot_width=300, plot_height=200, title=site, x_range=tax
            fig.vbar(x=[n+0.5 for n in range(13)],
               width=0.8,
               bottom=0,
               top=heights,
               color='firebrick')
            plots.append(fig)
```

The final step is to lay out and display the plots. Here, we will use the `gridplot` function from `bokeh.io`, but there are plenty of other options. Check out the documentation here.

```
In [166]: layout = gridplot([[plots[0],plots[1]],\
                             [plots[2],plots[3]],\
                             [plots[4],plots[5]],\
                             [plots[6]]])
          show(layout)
```

As a bonus, let's colour the bars individually according to the taxon that they refer to. There are 13 taxa, which is a bit too large to avoid using some similar colours, so we use the Category20 color palette instead. Several palettes are available via the `bokeh.palettes` module. Choose which one you want, then specify how many colours you need when you use the palette in your plot(s).

```
In [167]: from bokeh.palettes import Category20
```

```
In [168]: plots = []
          for site in data.columns:
              heights = data[site]
              taxa = list(data.index)
              fig = figure(plot_width=300, plot_height=200, title=site, x_range=tax
              fig.vbar(x=[n+0.5 for n in range(13)],
                 width=0.8,
                 bottom=0,
                 top=heights,
                 color=Category20[13])
              plots.append(fig)
```

```
In [169]: layout = gridplot([[plots[0],plots[1]],\
                             [plots[2],plots[3]],\
                             [plots[4],plots[5]],\
                             [plots[6]]])
          show(layout)
```

***Note on list comprehensions** A list comprehension is a way of building up a list from a loop on the fly. For example, the following two code blocks are equivalent:

```
In [170]: originals = range(10)
          squares = []
          for num in originals:
              squares.append(num**2)
          print(squares)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
In [171]: squares = [ x**2 for x in range(10) ]
          print(squares)

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

The list comprehension was necessary above, to align the x-axis labels correctly with the centre of each taxon bar. Another approach to get the same result is possible using the `arange` function from the `numpy` library, which provides an *array* of numbers in the range requested. Consider the difference in output between the two blocks below:

```
In [172]: some_numbers = list(range(2, 20)) # a standard list object
          print(some_numbers)
          some_numbers + 0.5

[2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]



        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-172-bcfdd1c9120f> in <module>()
          1 some_numbers = list(range(2, 20)) # a standard list object
          2 print(some_numbers)
    ----> 3 some_numbers + 0.5


        TypeError: can only concatenate list (not "float") to list
```

```
In [173]: from numpy import arange
          some_numbers = arange(2, 20)
          some_numbers + 0.5

Out[173]: array([  2.5,   3.5,   4.5,   5.5,   6.5,   7.5,   8.5,   9.5,  10.5,
                  11.5,  12.5,  13.5,  14.5,  15.5,  16.5,  17.5,  18.5,  19.5])
```

So, if we wanted to, we could use `arange` to create our plots as follows:

```
In [174]: plots = []
          for site in data.columns:
              heights = data[site]
              taxa = list(data.index)
              fig = figure(plot_width=300, plot_height=200, title=site, x_range=tax
```

```
            fig.vbar(x=arange(13)+0.5,
               width=0.8,
               bottom=0,
               top=heights,
               color=Category20[13])
            plots.append(fig)

In [175]: layout = gridplot([[plots[0],plots[1]],\
                             [plots[2],plots[3]],\
                             [plots[4],plots[5]],\
                             [plots[6]]])
          show(layout)
```

Chapter 7

About Bio-IT

The Bio-IT Project

The Bio-IT project is a community initiative, which aims to build, support, and promote computational biology activity at EMBL Heidelberg. The objective is for community members to help each other conduct their research, and to share their skills, experience, and tools with others.

The Bio-IT Project aims to achieve its objectives through activity in four areas:

- training and consultation
- regular networking of interested parties, encouraging community building and knowledge transfer between computational biologists
- establishing and maintaining community resources
- providing support for bioinformatics tools and services

You can find out more about the project and its activities, as well as listings for upcoming courses and events, by visiting the Bio-IT Portal at https://bio-it.embl.de.

Contact: bio-it@embl.de

EMBL Centres

The EMBL Centres are 'horizontal', cross-departmental structures that promote innovative research projects across disciplines. The EMBL Centres listed below have a strong computational component.

Bioimage Analysis The Centre for Bioimage Analysis (CBA) supports scientists in extracting quantitative information from images acquired with light- or electron-microscopy.

Visit the CBA hompage at https://bio-it.embl.de/centres/cba/

Contact: Christian Tischer (christian.tischer@embl.de)

Biological Modelling The CBM provides consultation appointments, performs collaborative modelling tasks, trains EMBL members in relevant programming languages and software tools and will foster collaboration and interactions between EMBL researchers using seminars, journal clubs as well as interdisciplinary retreats.

Visit the CBM hompage at https://bio-it.embl.de/centres/cbm/

Contact: Eva Geissen (geissen@embl.de)

Biomolecular Network Analysis The mission of the CBNA is to disseminate expertise, know-how and guidance in the field of biological network integration and analysis throughout EMBL to computational biologists and experimentalists alike.

Visit the CBNA hompage at https://bio-it.embl.de/centres/cbna/

Contact: Matt Rogon (rogon@embl.de)

Statistical Data Analysis The CSDA helps EMBL scientists to use adequate statistical methods for their specific technological or biological applications (testing, regression, clustering, classification, error rate estimation, sampling, visualization, ...).

Visit the CSDA hompage at https://bio-it.embl.de/centres/csda/

Contact: Bernd Klaus (bernd.klaus@embl.de)

Chapter 8

Acknowledgements

Handouts provided by EMBL Heidelberg Photolab (Many thanks to Udo Ringeisen and colleagues)

These materials are based on the original course written by Peter D Ashton (University of York), with additional material added/updated by members of the EMBL Bio-IT community, including:

- Toby Hodges
- Holger Dinkel
- Karin Sasaki
- Marc Gouw
- Malvika Sharan
- Renato Alves

Chapter 9

Links

Resources

- https://docs.python.org/3/tutorial - Python Tutorial
- http://www.learnpython.org - Another, interactive, Python tutorial
- http://awesome-python.com - A curated list of awesome Python frameworks, libraries, software and resources
- https://github.com/rasbt/python_reference - Useful functions, tutorials, and other Python-related things
- http://stackoverflow.com/questions/tagged/python - Search a history of thousands of questions and answers from frustrated programmers
- http://pythontutor.com - Visually inspect what happens when running Python step by step.
- http://pbpython.com/effective-matplotlib.html - A blogpost describing an approach to using `matplotlib` to plot data. Highly recommended for newcomers, to remove some of the potential frustrations of learning to use this very powerful plotting library.

Books

- http://www.onlineprogrammingbooks.com/python
- http://www.diveintopython3.net
- http://www.greenteapress.com/thinkpython/thinkpython.pdf
- http://learnpythonthehardway.org/book
- http://python.swaroopch.com/

Videos

- https://github.com/s16h/py-must-watch
- https://www.fullstackpython.com/best-python-videos.html

Bioinformatics

- http://rosalind.info Rosalind is a platform for learning bioinformatics and programming through problem solving