Python for Healthcare Modelling

Michael Allen: pythonhealthcare.org

April 12, 2018

Contents

Chapter 1

Python basics

1.1 Introduction

Hello

This book is a collection of code snippets that help me use Python for health services research, modelling and analysis. When learning something new I always work on a small code example to understand how something works, and to keep as a handy reference.

I will be looking at data handling, some statistics, data plotting, discrete event simulation, and machine learning.

I will be including use of pure Python as well as commonly used libraries such as NumPy, Pandas, MatPlotLib, SciPy, SciKitLearn, TensorFlow and Simpy.

Everything described here is performed in Python 3, based on the Anaconda Scientific Python environment, available for free (for Windows, Mac or Linux). Follow installation instructions from the site `https://www.anaconda.com/download/`

Anaconda comes with its own environments for writing the code: Spyder or Jypyter Notebooks. Both are nice. Other Free and Open Source options are PyCharm for more functionality, Atom, or Visual Studio Code for a modern code text editor, or Vim for a more old-fashioned but very fast and lightweight code text editor.

Occasionally other free libraries may be installed. If so they will be described where appropriate.

I choose to do all my work in GNU/Linux, but everything should also work in Microsoft Windows or Mac OS.

If you are new to this I would recommend installing the Anaconda Scientific Python environment, and then look for 'Spyder' in your computer's application listing. That will open up an 'Integrated Development Environment' (IDE): a posh phrase that means a place where you can both write and run code.

Happy coding!

Michael

1.2 Lists

Lists, tuples, sets and dictionaries are four basic objects for storing data in Python. Later we'll look at libraries that allow for specialised handling and analysis of large or complex data sets.

Lists are a group of text or numbers (or more complex objects) held together in a given order. Lists are recognised by using square brackets , with members of the list separated by commas. Lists are mutable, that is their contents may be changed. Note that when Python numbers a list the first element is referenced by the index zero. Python is a 'zero-indexed' language.

Below is code demonstrating some common handling of lists.

1.2.1 Creating a list

Lists may mix text, numbers, or other Python objects. Here is code to generate and print a list of mixed text and numbers. Note that the text is in speech marks (single or double; both will work). If the speech marks were missing then Python would think the text represented a variable name (such as the name of another list).

```
my_list = ['Gandalf','Bilbo','Gimli',10,30,40]
print (my_list)

OUT:
['Gandalf', 'Bilbo', 'Gimli', 10, 30, 40]
```

1.2.2 Deleting a list

The *del* command will delete a list (or any other variable).

```
my_list = ['Gandalf','Bilbo','Gimli']
del my_list
```

1.2.3 Returning and printing an element from a list

Python list store their contents in sequence. A particular element may be referenced by its index number. Python indexes start at zero. Watch out for this this will almost certainly trip you up at some point.

The example below prints the second item in the list. The first item would be referenced with 0 (zero).

```
my_list = ['Gandalf','Bilbo','Gimli',10,30,40]
print (my_list[1])

OUT:
Bilbo
```

Negative references may be used to refer to position from the end of the list. An index of -1 would be the last item (because -0 is the same as 0 and would be the first item!)

1.2.4 Taking slices of a list

Taking more than one element from a list is called a slice. The slice defines the starting point, and the point just after the end of the slice. The example below takes elements indexed 2 to 4 (but not 5)

```
my_list = [0,1,2,3,4,5,6,7,8,9,10]
slice = my_list[2:5]
print(slice)
```

```
OUT:
[2, 3, 4]
```

Slices can take every n elements, the example below takes every second element from index 1 up to, but not including, index 9 (remembering that the first element is index 0):

```
y_list = [0,1,2,3,4,5,6,7,8,9,10]
slice = my_list[2:9:2]
print(slice)

OUT:
[2, 4, 6, 8]
```

Reverse slices may also be taken by defining the end point before the beginning point and using a negative step:

```
my_list = [0,1,2,3,4,5,6,7,8,9,10]
slice = my_list[9:2:-2]
print(slice)

OUT:
[9, 7, 5, 3]
```

1.2.5 Deleting or changing an item in a list

Unlike tuples (which are very similar to lists), elements in a list may be deleted or changed.

Here the first element in the list is deleted:

```
my_list = ['Gandalf','Bilbo','Gimli',10,30,40]
del my_list[0]
print (my_list)

OUT:
['Bilbo', 'Gimli', 10, 30, 40]
```

Here the second element of the list is replaced:

```
my_list = ['Gandalf','Bilbo','Gimli',10,30,40]
my_list[0] = 'Boromir'
print (my_list)

OUT:
['Boromir', 'Bilbo', 'Gimli', 10, 30, 40]
```

1.2.6 Appending an item to a list, and joining two lists

Individual elements may be added to a list with *append*.

```
my_list = ['Gandalf','Bilbo','Gimli',10,30,40]
my_list.append ('Elrond')
print (my_list)

OUT:
['Gandalf', 'Bilbo', 'Gimli', 10, 30, 40, 'Elrond']
```

Two lists may be joined with a simple +

```
my_list = ['Gandalf','Bilbo','Gimli']
my_second_list = ['Elrond','Boromir']
my_combined_list = my_list + my_second_list
print (my_combined_list)
```

```
OUT:
['Gandalf', 'Bilbo', 'Gimli', 'Elrond', 'Boromir']
```

Or the *extend* method may be used to add a list to an existing list.

```
my_list = ['Gandalf','Bilbo','Gimli']
my_second_list = ['Elrond','Boromir']
my_list.extend(my_second_list)
print (my_list)

OUT:
['Gandalf', 'Bilbo', 'Gimli', 'Elrond', 'Boromir']
```

1.2.7 Copying a list

If we try to copy a list by saying mycopy mylist we do not generate a new copy. Instead we generate an alias, another name that refers to the original list. Any changes to mycopy will change mylist.

There are two ways we can make a separate copy of a list, where changes to the new copy will not change the original list:

```
my_list = ['Gandalf','Bilbo','Gimli']
my_copy = my_list.copy()
# Or
my_copy = my_list[:]
```

1.2.8 Inserting an element into a given position in a list

Inserting an element into a given position in a list

An element may be inserted into a list with the *insert* method. Here we specify that the new element will be inserted at index 2 (the third element in the list, remembering that the first element is index 0). In a later post we will look at methods specifically for maintaining a sorted list, but for now let us simply inset an element at a given position in the list.

```
my_list = ['Gandalf','Bilbo','Gimli']
my_list.insert(2, 'Boromir')
print (my_list)

OUT:
['Gandalf', 'Bilbo', 'Boromir', 'Gimli']
```

1.2.9 Sorting a list

The *sort* method can work on a list of text or a list of numbers (but not a mixed list text and numbers). Sort order may be reversed if wanted.

A normal sort:

```
x = [10, 5, 24, 5, 8]
x.sort()
print (x)

OUT:
[5, 5, 8, 10, 24]
```

A reverse sort:

```
x = [10, 5, 24, 5, 8]
x.sort(reverse=True)
print (x)

OUT:
[24, 10, 8, 5, 5]
```

1.2.10 Checking whether the list contains a given element

The *in* command checks whether a particular element exist in a list. It returns a 'Boolean' True or False.

```
my_list = ['Gandalf','Bilbo','Gimli','Elrond','Boromir']
is_in_list = 'Elrond' in my_list
print (is_in_list)
is_in_list = 'Bob' in my_list
print (is_in_list)

OUT:
True
False
```

1.2.11 Reversing a list

The method to reverse a list looks a little odd. It is actually creating a 'slice' of a list that steps back from the end of list.

```
my_list = ['Gandalf','Bilbo','Gimli','Elrond','Boromir']
my_list = my_list[::-1]
print (my_list)

OUT:
['Boromir', 'Elrond', 'Gimli', 'Bilbo', 'Gandalf']
```

1.2.12 Counting the number of elements in a list

Use the len function to return the number of elements in a list.

```
my_list = ['Gandalf','Bilbo','Gimli','Elrond','Boromir']
print (len(my_list))

OUT:
5
```

1.2.13 Returning the index position of an element of a list

The list *index* will return the position of the first element of the list matching the argument given.

```
my_list = ['Gandalf','Bilbo','Gimli','Elrond','Boromir']
index_pos = my_list.index('Elrond')
print (index_pos)

OUT:
3
```

If the element is not in the list then the code will error. Later we will look at some different ways of coping with this type of error. For now let us introduce an if/then/else statement (and we'll cover those in more detail later as well).

```
my_list = ['Gandalf','Bilbo','Gimli','Elrond','Boromir']
test_element = 'Elrond'
is_in_list = test_element in my_list
if is_in_list: # this is the same as saying if is_in_list == True
    print (test_element,'is in list at position', my_list.index(test_element))
else:
    print (test_element, 'is not in list')

OUT:
Elrond is in list at position 3
```

1.2.14 Removing an element of a list by its value

An element may be removed from a list by using its value rather than index. Be aware that you might want to put in a method (like the if statement above) to check the value is present before removing it. But here is a simple remove (this will remove the first instance of that particular value:

```
my_list = ['Gandalf','Bilbo','Gimli','Elrond','Boromir']
my_list.remove('Bilbo')
print (my_list)

OUT:
['Gandalf', 'Gimli', 'Elrond', 'Boromir']
```

1.2.15 'Popping' an element from a list'

Popping an element from a list means to retrieve an element from a list, removing it from the list at the same time. If no element number is given then the last element of the list will be popped.

```
my_list = ['Gandalf','Bilbo','Gimli','Elrond','Boromir']
print ('Original list:', my_list)
x = my_list.pop()
print (x, 'was popped')
print ('Remaining list:',my_list)
x = my_list.pop(1)
print (x, 'was popped')
print ('Remaining list:', my_list)

OUT:
Original list: ['Gandalf', 'Bilbo', 'Gimli', 'Elrond', 'Boromir']
Boromir was popped
Remaining list: ['Gandalf', 'Bilbo', 'Gimli', 'Elrond']
Bilbo was popped
Remaining list: ['Gandalf', 'Gimli', 'Elrond']
```

1.2.16 Iterating (stepping) through a list

Lists are 'iterable', that is they can be stepped through. The example below prints one element at a time. We use the variable name 'x' here, but any name may be used.

```
my_list = [1,2,3,4,5,6,7,8,9]
for x in my_list:
    print(x, end=' ') # end command replaces a newline with a space
```

```
OUT:
1 2 3 4 5 6 7 8 9
```

1.2.17 Iterating through a list and changing element values

Iterating through a list and changing the value in the original list is a little more complicated. Below the range command creates a range of numbers from zero up to the index of the last element of the list. If there were 4 elements, for example, then the range function would produce '0, 1, 2, 3'. This allows us to iterate through the index numbers for the list (the position of each element). Using the index number we can change the value of the element in that position of the list.

```
my_list = [1,2,3,4,5,6,7,8,9]
for index in range(len(my_list)):
    my_list[index] = my_list[index]**2
print ('\nMy list after iterating and mutating:')
print (my_list)

OUT:

My list after iterating and mutating:
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

1.2.18 Counting the number of times an element exists in a list

The *count* method counts the number of times a value occurs in a list. If the value does not exist a zero is returned.

```
x =[1, 2, 3, 4, 1, 4, 1]
test_value = 1
print ('Count the number of times 1 occurs in',x)
print (x.count(test_value),'\n')

OUT:
Count the number of times 1 occurs in [1, 2, 3, 4, 1, 4, 1]
3
```

1.2.19 Turning a sentence into a list of words

The *split* method allows a long string, such as a sentence, to be separated into individual words. If no separator is defined any white space (such as a space or comma) will be used to divide words.

```
my_sentence ="Turn this sentence into a list"
words = my_sentence.split()
print (words)

OUT:
['Turn', 'this', 'sentence', 'into', 'a', 'list']
```

A separator may be specified, such as diving sentences at commas:

```
my_sentence = 'Hello there, my name is Bilbo'
divided_sentence = my_sentence.split(sep=',')
print(divided_sentence)

OUT:
['Hello there', ' my name is Bilbo']
```

1.3 Nested lists

So far we have looked at lists which contain a simple series of numbers, text, or a mixture of numbers and texts (Python lists can also hold any Python object, but in Healthcare modelling we are usually dealing with numbers or text in a list).

It is possible though to build nested lists. In the example below we generate a list manually, with each nested list on a separate line. This separation by line is just to make it easier to see; it is not needed in Python, but thought to layout of code is important if other people will be looking at your code.

```
my_list = [[1,2,3],
           [4,5,6],
           [7,8,9]]
print (my_list)

OUT:
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

We can then refer to items by using two reference indices. The first refers to the nested list block (so the first id [1, 2, 3] and the second refers to the position within that block. Remember that Python is zero indexed so the first element of the first list is 0[0].

```
print (my_list[1][1])

OUT:
5
```

Or we can use other variables to refer to the position:

```
x = 2
y = 0
print (my_list[x][y])

OUT:
7
```

More complex structures can be built up with further nesting of lists to give multi-dimensional lists.

WARNING: Handling large arrays of data this way is possible but slow. For modelling we are much better off using two libraries dedicated to fast handling of large data sets: NumPy and Pandas. We will be covering those libraries soon.

1.4 Tuples

1.4.1 Creating and adding to tuples

Tuples are like lists in many ways, apart from they cannot be changed - they are immutable. Tuples are defined used curved brackets (unlike square brackets for lists). Tuples may be returned, or be required, by various functions in Python, and tuples may be chosen where immutability would be an advantage (for example, to help prevent accidental changes).

```
my_tuple = ('Hobbit','Elf','Ork','Dwarf')
print (my_tuple[1])

OUT:
Elf
```

It is possible to add to a tuple. Note that if adding a single item an additional comma is used to indicate to Python that the variable being added is a tuple.

```
my_tuple = my_tuple + ('Man',)
my_tuple += ('Wizard','Balrog') # Note that the += is short hand to add something to itslef
print (my_tuple)

OUT:
('Hobbit', 'Elf', 'Ork', 'Dwarf', 'Man', 'Wizard', 'Balrog')
```

It is not possible to change or delete an item in a tuple. To change or delete a tuple a new tuple must be built (but if this is going to happen then a list would be a better choice).

```
my_new_tuple = my_tuple[0:2] + ('Goblin',) + my_tuple[4:]
print (my_new_tuple)

OUT:
('Hobbit', 'Elf', 'Goblin', 'Man', 'Wizard', 'Balrog')
```

1.4.2 Converting between tuples and lists, and sorting tuples

A tuple may be turned into a list. We can recognise that it is a list by the square brackets.

```
my_list = list(my_tuple)
print (my_list)

OUT:
['Hobbit', 'Elf', 'Ork', 'Dwarf', 'Man', 'Wizard', 'Balrog']
```

A list may also be converter into a tuple.

```
my_list.sort()
my_new_tuple = tuple(my_list)
print (my_new_tuple)

OUT:
my_list.sort()

my_new_tuple = tuple(my_list)

print (my_new_tuple)

('Balrog', 'Dwarf', 'Elf', 'Hobbit', 'Man', 'Ork', 'Wizard')
```

In the above example we sorted a list and converted it to a tuple. Tuples cannot be changed (apart from being added to), so it is not possible to directly sort a tuple. The sorted command will act on a tuple to

sort it, but returns a list.

```
print (sorted(my_tuple))
```

```
OUT:
['Balrog', 'Dwarf', 'Elf', 'Hobbit', 'Man', 'Ork', 'Wizard']
```

The tuple may be sorted by converting back to a list in a single step (but think of using lists, rather than tuples, is sorting will be common.

```
my_tuple = tuple(sorted(my_tuple))
print (my_tuple)
```

```
OUT:
('Balrog', 'Dwarf', 'Elf', 'Hobbit', 'Man', 'Ork', 'Wizard')
```

1.5 Sets

Sets are unordered collections of unique values. Common uses include membership testing, finding overlaps and differences between sets, and removing duplicates from a sequence.

Sets, like dictionaries, are defined using curly brackets.

1.5.1 Creating sets

Creating a set directly:

```
my_set = {'a','e','i','o','u'}
print (my_set)

OUT:
{'a', 'i', 'u', 'o', 'e'}
```

Creating a set from a list:

```
my_list = ['a','e','i','o','u']
my_set = set(my_list)
print (my_set)

OUT:
{'a', 'i', 'u', 'o', 'e'}
```

Creating an empty set. This cannot be done simply with

```
my_set = set()
print (my_set)
```

1.5.2 Adding to a set and removing duplicates

Adding one element to a set:

```
my_set.add('y')
print (my_set)

OUT:
{'a', 'y', 'i', 'u', 'o', 'e'}
```

Adding multiple elements to a set (note that the set has no duplication of entries already present; sets automatically remove duplicate items):

```
my_set.add('y')
print (my_set)

OUT:
{'a', 'y', 'i', 'u', 'o', 'e'}
```

1.5.3 Analysing the intersection between sets

Sets may be used to explore the intersection betwen sets. There are usually two orms of syntax which may be used, both of which are given in the examples below.

```
set1 = {'a','b','c','d','e','f'}
set2 = {'a','e','i','o','u'}
```

The difference between two sets:

```
print ('difference')
print (set1.difference(set2))
print (set1 - set2)

OUT:
difference
{'c', 'f', 'b', 'd'}
{'c', 'f', 'b', 'd'}
```

The intersection between two sets:

```
print ('Intersection')
print (set1.intersection(set2))
print (set1 & set2)

OUT:
Intersection
{'a', 'e'}
{'a', 'e'}
```

Is a subset? Is set 2 wholly included in set 1?

```
print ('Is subset?')
print (set1.issubset(set2))
print (set1 <= set2)

OUT:
Is subset?
False
False
```

Is a superset? Does set 1 wholly include set 2?

```
print ('Is superset?')
print (set1.issuperset(set2))
print (set1 >= set2)

OUT:
Is superset?
False
False
```

The union between two sets:

```
print ('Union')
print (set1.union(set2))
print (set1 | set2)

OUT:
Union
{'b', 'o', 'a', 'f', 'c', 'i', 'u', 'd', 'e'}
{'b', 'o', 'a', 'f', 'c', 'i', 'u', 'd', 'e'}
```

Symmetric difference: in either but not both (equivalent to xor)

```
print ('Symmetric difference')
print (set1.symmetric_difference(set2))
print (set1 ^ set2)

OUT:
Symmetric difference
{'c', 'i', 'b', 'u', 'f', 'o', 'd'}
{'c', 'i', 'b', 'u', 'f', 'o', 'd'}
```

1.6 Dictionaries

Dictionaries store objects such as text or numbers (or other Python objects). We saw that lists stored elements in sequence, and that each element can be referred to by an index number, which is the position of that element in a list. The contents of dictionaries are accessed by a unique *key* which may be a number or may be text.

Dictionaries take the form of key1:value1, key2:value2

The *value* may be a single number or word, or may be another Python object such as a list or a tuple.

Dictionaries allow fast random-access to data.

1.6.1 Creating and adding to dictionaries

Dictionaries may be created with content, and are referred to by their *key*:

```
fellowship = {'Man': 2, 'Hobbit':4, 'Wizard':1, 'Elf':1, 'Dwarf':1}
print (fellowship['Man'])

OUT:
2
```

Dictionaries may also be created empty. New contened my be added to an an empty or an existing dictionary.

```
trilogy = {}
trilogy['Part 1'] = 'The fellowshipp of the rring'
trilogy['Part 2'] = 'The Two Towwwwers'
trilogy['Part 3'] = 'The return of the kingg'
print (trilogy['Part 2'])

OUT:
The Two Towwwwers
```

Dictionaries may contain a mixture of data types.

```
lord_of_the_rings = {'books':3,
                     'author':'JRR Tolkien',
                     'main_characters':['Frodo','Sam','Gandalf']}
print (lord_of_the_rings['main_characters'])

OUT:
['Frodo', 'Sam', 'Gandalf']
```

1.6.2 Changing a dictionary entry

Dictionary elements are over-written if a key already exists.

```
trilogy['Part 2'] = 'The Two Towers'
print (trilogy['Part 2'])

OUT:
The Two Towers
```

Dictionary items may also be updated using another dictionary:

```
entry_update = {'Part 1':'The Fellowship of the Ring',
                'Part 3':'The Return of the King'}
trilogy.update(entry_update)
print (trilogy)
```

```
OUT:
{'Part 1': 'The Fellowship of the Ring', 'Part 2': 'The Two Towers',
'Part 3': 'The Return of the King'}
```

1.6.3 Deleting a dictionary entry

Dictionary entries may be deleted, by reference to their key, with the *del* command. If the key does not exist an error will be caused, so in the example below we check that the entry exists before trying to delete it.

```
entry_to_delete = 'Part 2'
if entry_to_delete in trilogy:
    del trilogy[entry_to_delete]
    print (trilogy)
else:
    print('That key does not exist')

OUT:
{'Part 1': 'The Fellowship of the Ring', 'Part 3': 'The Return of the King'}
```

1.6.4 Iterating through a dictionary

There are two main methods to iterate through a dictionary. The first retrieves just the key, and the value may then be retrieved from that key. The second method retrieves the key and value together.

```
lord_of_the_rings = {'books':3,
                     'author':'JRR Tolkien',
                     'main_characters':['Frodo','Sam','Gandalf']}

# Iterating method 1
for key in lord_of_the_rings:
    print (key, '-', lord_of_the_rings[key])

# Iteraring method 2
print()
for key, value in lord_of_the_rings.items():
    print (key,'-', value)

OUT:
books - 3
author - JRR Tolkien
main_characters - ['Frodo', 'Sam', 'Gandalf']

books - 3
author - JRR Tolkien
main_characters - ['Frodo', 'Sam', 'Gandalf']
```

1.6.5 Converting dictionaries to lists of keys and values

Dictionary keys and values may be accessed separately. We can also return a list of tuples of keys and values with the items method.

```
lord_of_the_rings = {'books':3,
                     'author':'JRR Tolkien',
                     'main_characters':['Frodo','Sam','Gandalf']}
# print the keys:
```

```
print ('\nKeys:') # \n creates empty line before text
print (list(lord_of_the_rings.keys()))

# print the values
print ('\nValues:')
print (list(lord_of_the_rings.values()))

# print keys and values
print ('\nKeys and Values:')
print (list(lord_of_the_rings.items()))

OUT:
Keys:
['books', 'author', 'main_characters']

Values:
[3, 'JRR Tolkien', ['Frodo', 'Sam', 'Gandalf']]

Keys and Values:
[('books', 3), ('author', 'JRR Tolkien'), ('main_characters', ['Frodo', 'Sam',
'Gandalf'])]
```

1.6.6 Using the get method to return a null value if a key does not exist

Using the usual method of referring to a dictionary element, an error will be returned if the key used does not exist. An 'if key in dictionary' statement could be used to check the key exists. Or the get method will return a null value if the key does not exist. The get method allows for a default value to be returned if a key is not found.

```
print (lord_of_the_rings.get('author'))
print (lord_of_the_rings.get('elves'))
print (lord_of_the_rings.get('books','Key not found - sorry'))
print (lord_of_the_rings.get('dwarfs','Key not found - sorry'))

JRR Tolkien
None
3
Key not found - sorry
```

1.6.7 Intersection between dictionaries

Like sets the intersection between dictionaries may be evaluated. See the entry on sets for more intersection methods.

```
a = {'x':1,'y':2,'z':3,'zz':2}
b = {'x':10,'w':11,'y':2,'yy':2}
# Show keys in common
print('Keys in common: ',a.keys() & b.keys())
# Show keys in a not in b
print('Keys in a but not b:',a.keys()-b.keys())
# Show keys+values in common (items() return key+value, so both need to be the same)
print('Keys and values in common:',a.items() & b.items())

OUT:
Keys in common:  {'x', 'y'}
Keys in a but not b: {'z', 'zz'}
Keys and values in common: {('y', 2)}
```

1.6.8 Other dictionary methods

dictionary.clear() empties a dictionary

dictionary.pop(key) will return an item and remove it from the dictionary

1.6.9 Ordered dictionaries

Like sets, dictionaries do not usually keep the order of entries. If it is useful to keep the order of entry in a dictionary then the OrderDict object is useful. This needs an import statement before it is used (all import statements are usually kept together at the start of code).

```
from collections import OrderedDict
d=OrderedDict()

d['Gandalf']=1
d['Bilbo']=2
d['Frodo']=3
d['Sam']=4

for key in d:
    print(key,d[key])

OUT:
Gandalf 1
Bilbo 2
Frodo 3
Sam 4
```

1.7 Accessing math functions through the math module

Here we are going to use the math module as an introduction to using modules. The math module contains a range of useful mathematical functions that are not built into Python directly. So let's go ahead and start by importing the module. Module imports are usually performed at the start of a programme.

```
import math
```

When this type of import is used Python loads up a link to the module so that module functions and variables may be used. Here for example we access the value of pi through the module.

```
print (math.pi)

OUT:
3.141592653589793
```

Another way of accessing module contents is to directly load up a function or a variable into Python. When we do this we no longer need to use the module name after the import. This method is not generally recommended as it can lead to conflicts of names, and is not so clear where that function or variable comes from. But here is how it is done.

```
from math import pi
print (pi)

OUT:
3.141592653589793
```

Multiple methods and variables may be loaded at the same time in this way.

```
from math import pi, tau, log10
print (tau)
print (log10(100))

OUT:
6.283185307179586
2.0
```

But usually it is better practice to keep using the library name in the code.

```
print (math.log10(100))

OUT:
2.0
```

To access help on any Python function use the help command in a Python interpreter.

```
help (math.log10)

Help on built-in function log10 in module math:

log10(...)
    log10(x)

    Return the base 10 logarithm of x.
```

To find out all the methods in a module, and how to use those methods we can simply type help (module_name) into the Python interpreter. The module must first have been imported, as we did for math above.

```
help (math)

OUT:
Help on module math:
```

```
NAME
    math

MODULE REFERENCE
    https://docs.python.org/3.6/library/math

    The following documentation is automatically generated from the Python
    source files.  It may be incomplete, incorrect or include features that
    are considered implementation detail and may vary between Python
    implementations.  When in doubt, consult the module reference at the
    location listed above.

DESCRIPTION
    This module is always available.  It provides access to the
    mathematical functions defined by the C standard.

FUNCTIONS
    acos(...)
        acos(x)

        Return the arc cosine (measured in radians) of x.

    acosh(...)
        acosh(x)

        Return the inverse hyperbolic cosine of x.

    asin(...)
        asin(x)

        Return the arc sine (measured in radians) of x.

    asinh(...)
        asinh(x)

        Return the inverse hyperbolic sine of x.

    atan(...)
        atan(x)

        Return the arc tangent (measured in radians) of x.

    atan2(...)
        atan2(y, x)

        Return the arc tangent (measured in radians) of y/x.
        Unlike atan(y/x), the signs of both x and y are considered.

    atanh(...)
        atanh(x)

        Return the inverse hyperbolic tangent of x.

    ceil(...)
        ceil(x)

        Return the ceiling of x as an Integral.
        This is the smallest integer >= x.
```

```
    copysign(...)
        copysign(x, y)

        Return a float with the magnitude (absolute value) of x but the sign
        of y. On platforms that support signed zeros, copysign(1.0, -0.0)
        returns -1.0.

    cos(...)
        cos(x)

        Return the cosine of x (measured in radians).

    cosh(...)
        cosh(x)

        Return the hyperbolic cosine of x.

    degrees(...)
        degrees(x)

        Convert angle x from radians to degrees.

    erf(...)
        erf(x)

        Error function at x.

    erfc(...)
        erfc(x)

        Complementary error function at x.

    exp(...)
        exp(x)

        Return e raised to the power of x.

    expm1(...)
        expm1(x)

        Return exp(x)-1.
        This function avoids the loss of precision involved in the direct evaluation of
exp(x)-1 for small x.

    fabs(...)
        fabs(x)

        Return the absolute value of the float x.

    factorial(...)
        factorial(x) -> Integral

        Find x!. Raise a ValueError if x is negative or non-integral.

    floor(...)
        floor(x)
```

```
    Return the floor of x as an Integral.
    This is the largest integer <= x.

fmod(...)
    fmod(x, y)

    Return fmod(x, y), according to platform C.  x % y may differ.

frexp(...)
    frexp(x)

    Return the mantissa and exponent of x, as pair (m, e).
    m is a float and e is an int, such that x = m * 2.**e.
    If x is 0, m and e are both 0.  Else 0.5 <= abs(m) < 1.0.

fsum(...)
    fsum(iterable)

    Return an accurate floating point sum of values in the iterable.
    Assumes IEEE-754 floating point arithmetic.

gamma(...)
    gamma(x)

    Gamma function at x.

gcd(...)
    gcd(x, y) -> int
    greatest common divisor of x and y

hypot(...)
    hypot(x, y)

    Return the Euclidean distance, sqrt(x*x + y*y).

isclose(...)
    isclose(a, b, *, rel_tol=1e-09, abs_tol=0.0) -> bool

    Determine whether two floating point numbers are close in value.

       rel_tol
           maximum difference for being considered "close", relative to the
           magnitude of the input values
        abs_tol
           maximum difference for being considered "close", regardless of the
           magnitude of the input values

    Return True if a is close in value to b, and False otherwise.

    For the values to be considered close, the difference between them
    must be smaller than at least one of the tolerances.

    -inf, inf and NaN behave similarly to the IEEE 754 Standard.  That
    is, NaN is not close to anything, even itself.  inf and -inf are
    only close to themselves.

isfinite(...)
    isfinite(x) -> bool
```

```
    Return True if x is neither an infinity nor a NaN, and False otherwise.

isinf(...)
    isinf(x) -> bool

    Return True if x is a positive or negative infinity, and False otherwise.

isnan(...)
    isnan(x) -> bool

    Return True if x is a NaN (not a number), and False otherwise.

ldexp(...)
    ldexp(x, i)

    Return x * (2**i).

lgamma(...)
    lgamma(x)

    Natural logarithm of absolute value of Gamma function at x.

log(...)
    log(x[, base])

    Return the logarithm of x to the given base.
    If the base not specified, returns the natural logarithm (base e) of x.

log10(...)
    log10(x)

    Return the base 10 logarithm of x.

log1p(...)
    log1p(x)

    Return the natural logarithm of 1+x (base e).
    The result is computed in a way which is accurate for x near zero.

log2(...)
    log2(x)

    Return the base 2 logarithm of x.

modf(...)
    modf(x)

    Return the fractional and integer parts of x.  Both results carry the sign
    of x and are floats.

pow(...)
    pow(x, y)

    Return x**y (x to the power of y).

radians(...)
    radians(x)
```

```
        Convert angle x from degrees to radians.

    sin(...)
        sin(x)

        Return the sine of x (measured in radians).

    sinh(...)
        sinh(x)

        Return the hyperbolic sine of x.

    sqrt(...)
        sqrt(x)

        Return the square root of x.

    tan(...)
        tan(x)

        Return the tangent of x (measured in radians).

    tanh(...)
        tanh(x)

        Return the hyperbolic tangent of x.

    trunc(...)
        trunc(x:Real) -> Integral

        Truncates x to the nearest Integral toward 0. Uses the __trunc__ magic method.

DATA
    e = 2.718281828459045
    inf = inf
    nan = nan
    pi = 3.141592653589793
    tau = 6.283185307179586

FILE
    /home/michael/anaconda3/lib/python3.6/lib-dynload/math.cpython-36m-x86_64-linux-gnu.so
```

So now, for example, we know that to take a square root of a number we can use the math module, and use the sqrt() function, or use the pow() function which can do any power or root.

```
print (math.sqrt(4))
print (math.pow(4,0.5))

OUT:
2.0
2.0
```

In Python you might read about packages as well as modules. The two names are sometimes used interchangeably, but strictly a package is a collection of modules.

1.8 Variable types

Python is a dynamic language where variable type (e.g. whether a variable is a string or an integer) is not fixed. Sometime though you might like to force a particular type, or convert between types (most common where numbers may be contained within a string.

The most common types of variables in python are integers, float (nearly all types of numbers that are not integers) and strings.

Within python code the function *type* will show what variable type a string is.

```
x = 11 # This is an integer
print (type(x))

OUT:
<class 'int'>
```

Outside of the code, within the interpreter requesting help on a variable name will give the help for its variable type.

```
help (x)

OUT:
Help on int object:

class int(object)
 |  int(x=0) -> integer
 |  int(x, base=10) -> integer
 |
 |  Convert a number or string to an integer, or return 0 if no arguments
 |  are given.  If x is a number, return x.__int__().  For floating point
 |  numbers, this truncates towards zero.
 |
 |  If x is not a number or if base is given, then x must be a string,
 |  bytes, or bytearray instance representing an integer literal in the
 |  given base.  The literal can be preceded by '+' or '-' and be surrounded
 |  by whitespace.  The base defaults to 10.  Valid bases are 0 and 2-36.
 |  Base 0 means to interpret the base from the string as an integer literal.
 |  >>> int('0b100', base=0)
 |  4
 |
 |  Methods defined here:

...
```

Though x is an integer, if we divide it by 2, Python will dynamically change its type to float. This may be common sense to some, but may be irritating to people who are used to coding languages where variables have 'static' types (this particular behaviour is also different to Python2 where the normal behaviours is for integers to remain as integers).

```
x = x/2
print (x)
print (type(x))

OUT:
5.5
<class 'float'>
```

Python is trying to be helpful, but this dynamic behaviour of variables can sometimes need a little debugging. In our example here, to keep our answer as an integer (if that is what we needed to do) we need to specify that:

```
x = 11
x = int(x/2)
print (x)
print (type(x))

OUT:
5
<class 'int'>
```

We can convert between types, such as in the follow examples. Numbers may exist as text, particularly in some file imports.

```
x = '10'
print ('Look what happens when we try to multiply a string')
print (x *3)
x = float (x)
print ('Now let us multiply the converted variable')
print (x*3)

OUT:
Look what happens when we try to multiply a string
101010
Now let us multiply the converted variable
30.0
```

1.9 Random numbers and integers

Here we look at the standard Python random number generator. It uses a *Mersenne Twister*, one of the mostly commonly-used random number generators. The generator can generate random integers, random sequences, and random numbers according to a number of different distributions.

1.9.1 Importing the random module and setting a random seed

```
import random
```

Random numbers can be repeatable in the sequence they are generated if a 'seed' is given. If no seed is given the random number sequence generated each time will be different.

```
random.seed() # This will reset the random seed so that the random number sequence will be different
random.seed(10) # giving a number will lead to a repeatable random number sequence each time
```

1.9.2 Random numbers

Random integers between (and including) a minimum and maximum:

```
i
print (random.randint(0,5))
```

```
OUT:
4
```

Return a random number between 0 and 1:

```
print (random.random())
```

```
OUT:
0.032585065282054626
```

Return a number (floating, not integer) between a & b:

```
print (random.uniform(0,5))
```

```
OUT:
2.412808372754279
```

Select from normal distribution (with mu, sigma):

```
print (random.normalvariate(10,3))
```

```
OUT:
5.3538059684648855
```

Other distributions (see help(random) for more info after importing random module):

Lognormal, Exponential, Beta, Gaussian, Gamma, Weibul

1.9.3 Generating random sequences

The random library may also be used to shuffle a list:

```
deck = ['ace','two','three','four']
random.shuffle(deck)
print (deck)
```

```
OUT:
['three', 'ace', 'two', 'four']
```

Sampling without replacement:

```
sample = random.sample([10, 20, 30, 40, 50], k=4) # k is the number of samples to select
print (sample)

OUT:
[20, 30, 10, 50]
```

Sampling with replacement:

```
pick_from = ['red', 'black', 'green']
sample = random.choices(pick_from, k=10)
print(sample)

OUT:
['black', 'red', 'black', 'green', 'red', 'red', 'black', 'red', 'black', 'green']
```

Sampling with replacement and weighted sampling:

```
pick_from = ['red', 'black', 'green']
pick_weights = [18, 18, 2]
sample = random.choices(pick_from, pick_weights, k=10)
print(sample)

OUT:
['black', 'red', 'red', 'red', 'black', 'red', 'red', 'black', 'red', 'black']
```

1.10 if, elif, else, while, and logical operators

Like many programming languages, Python enables code to be run based on testing whether a condition is true. Python uses indented blocks to run code according to these tests.

1.10.1 if, else, elif

emphif statements run a block of code if a particular condition is true. emphelif or 'else if' statements can subsequently run to test more conditions, but these run only if none of the precious emphif or emphelif statements were true. emphelse statements may be used to run code if none of the previous emphif or emphelif were true.

emphif statements may be run alone, with emphelse statements or with emphelif statements. emphelse and emphelif statements require a previous emphif statement.

In the following example we use the input command to ask a user for a password and test against known value.

Here we use just one elif statement, but multiple statements could be used.

Note that the tests of equality uses double = signs

```
password = input ("Password? ") # get user to enter password
if password == "secret":
    print ("Well done")
    print ("You")
elif password=="Secret":
    print ("Close!")
else:
    print("Noodle brain")

OUT:
Password? Secret
Close!
```

1.10.2 while statements

emphwhile statements may be used to repeat some actions until a condition is no longer true. For example:

```
x = 0
while x <5:
    x +=1 # This is shorthand for x = x + 1
    print (x)

OUT:
1
2
3
4
5
```

1.10.3 Logical operators

Logical operators

The following are commonly used logical operators:

== Test of identity

!= Not equal to

>greater than

<less than

>= equal to or greater than

<= less than or equal to

in test whether element is in list/tuple/dictionary

not in test whether element is not in list/tuple/dictionary

and test multiple conditions are true

or test one of alternative conditions are true

any test whether all elements are true

all test whether all elements are true

When Python test conditions they are evaluated as either True or False. These values may also be used directly in tests:

```
x = 10
y = 20
z = 30

# using and/or
print (x>15)
print (x>15 and y>15 and z>15)
print (x>15 or y>15 or z>15)
print ()

# Using any and all
print (any([x>20, y>20, z>20]))
test_array = [x>20, y>20, z>20]
print (test_array)
print (any(test_array))
print (all(test_array))

OUT:
False
False
True

True
[False, False, True]
True
False
```

1.11 Loops and iterating

for loops can be used to step through lists, tuples, and other 'iterable' objects.

Iterating through a list:

```
for item in [10,25,50,75,100]:
    print (item, item**2)

OUT:

10 100
25 625
50 2500
75 5625
100 10000
```

A *for* loop may be used to generate and loop through a sequence of numbers (note that a 'range' does not include the maximum value specified):

```
for i in range(100,150,10):
    print(i)

OUT:

100
110
120
130
140
```

A for loop may be used to loop through an index of positions in a list:

```
my_list = ['Frodo','Bilbo','Gandalf','Gimli','Sauron']

for i in range(len(my_list)):
    print ('Index:',i,', Value',my_list[i])

OUT:

Index: 0 , Value Frodo
Index: 1 , Value Bilbo
Index: 2 , Value Gandalf
Index: 3 , Value Gimli
Index: 4 , Value Sauron
```

1.11.1 Breaking out of loops or continuing the loop without action

Though it may not be considered best coding practice, it is possible to prematurely escape a loop with the *break* command:

```
for i in range(10): # This loop would normally go from 0 to 9
    if i == 5:
        break
    else:
        print(i)

print ('Loop complete')

OUT:
```

```
0
1
2
3
4
Loop complete
```

Or, rather than breaking out of a loop, it is possible to effectively skip an iteration of a loop with the *continue* command. This may be places anywhere in the loop and returns the focus to the start of the loop.

```
for i in range (10):
    if i%2 == 0: # This is the integer remainder after dividing i by 2
        continue
    else:
        print (i)

print ('Loop complete')

OUT:

1
3
5
7
9
Loop complete
```

1.11.2 Using pass to replace active code in a loop

The *pass* command is most useful as a place holder to allow a loop to be built and have contents added later.

```
for i in range (10):
    # Some code will be added here later
    pass

print ('Loop complete')

OUT:

Loop complete
```

1.12 List comprehensions: one line loops

List comprehensions are a very Pythonic way of condensing loops and action into a single line.

Here is the long form of a loop:

```
my_list=[1,4,5,7,9,10,3,2,16]
my_new_list=[]

for x in my_list:
    if x>5:
        my_new_list.append(x**2)

print (my_new_list)

OUT:

[49, 81, 100, 256]
```

This may be done by a single line list comprehension:

```
y = [x**2 for x in my_list if x>5]

print (y)

OUT:

[49, 81, 100, 256]
```

1.13 Else after while

The *else* clause is only executed when your while condition becomes false. If you *break* out of the loop it won't be executed.

The following loop will run the else code:

```
x = 0
while x < 5:
    x += 1
    print (x)
else:
    print ('Else statement has run')
print ('End')

OUT:

x = 0

while x < 5:

    x += 1

    print (x)

else:

    print ('Else statement has run')

print ('End')

OUT:

1
2
3
4
5
Else statement has run
End
```

The following loop with a *break* will not run the *else* code:

```
x = 0
while x < 5:
    x += 1
    print (x)
    if x >3:
        break
else:
    print ('Else statement has run')
print ('End')

OUT:

1
2
3
4
End
```

1.14 try ... except (where code might fail)

In perfect code all eventualities will be anticipated and code written for all eventualities. Sometimes though we might want to allow for more general failure of code and provide an alternative route.

In the following code we try to perform some code on a list. For the code to work the list must be long enough and the number must be positive. We use try .... except to perform the calculation where possible and return a null value if not.

In Python an error is called an 'exception'.

```
import math

x = [1, 2, -4]

# Try to take the square root of elements of a list.
# As a user try entering a list index too high (3 or higher)
# or try to enter index 2 (which has a negative value, -4)

test = int(input('Index position?: '))
try:
    print (math.sqrt(x[test]))
except:
    print ('Sorry, no can do')

OUT:

Index position?: 2
Sorry, no can do
```

Try .... except may be used in final code (but it is better to allow for all specific circumstances) or may be used during debugging to evaluate variables at the time of code failure

1.15 Decimal places in output

Python allows a lot of control over formatting of output. But here we will just look at controlling decimal places of output.

There are some different ways. Here is perhaps the most common (because it is most similar to other languages).

The number use is represented in the print function as %x.yf where x is the total number of spaces to use (if defined this will add padding to short numbers to align numbers on different lines, for example), and y is the number of decimal places. f informs python that we are formatting a float (a decimal number). The %x.yf is used as a placeholder in the output (multiple palceholders are possible) and the values are given at the end of the print statement as below:

```
import math
pi = math.pi
pi_square = pi**2

print('Pi is %.3f, and Pi squared is %.3f' %(pi,pi_square))

OUT:

Pi is 3.142, and Pi squared is 9.870
```

It is also possible to round numbers before printing (or sending to a file). If taking this approach be aware that this may limit the precision of further work using these numbers:

```
import math
pi = math.pi
pi = round(pi,3)
print (pi)

OUT:
3.142
```

1.16 Writing to and reading from files

For data analytics we will usually be using two libraries called NumPy and Pandas which have their own simple methods for importing data, but here is the standard Python method. This standard method may sometimes have an advantage that data may be written and read one line at a time without having all data held in memory. This may be useful for very large data files.

1.16.1 File access modes

Files may be opened with different access control models

Text file access modes:

"r" Read (if file does not exist there will be an error)

"w" Write If the file exists contents are overwritten. If the file does not exist, it is created.

"a" Append - will add to file, or will create a file if no file exists

"r+" Read from and write to a file

"w+" Write to and read from a text file. If the file exists contents are overwritten. If the file does not exist, it is created.

"a+" Append and read from a text file. If the file exists, new data is appended to it. If the files doesn't exist, it's created.

1.16.2 Creating a file with write

This method where we open a file to save to with the 'w' argument will create a new file or overwrite an old one.

In this standard way of using a file we begin by opening a file and end by closing the file:

```
text_file = open('write_it.txt', 'w')
text_file.write('Line 1\n')
text_file.write('This is line 2\n')
text_file.write('That makes this line 3\n')
text_file.close()
\begin{verbatim}

An alternative to opne and closing the file is using the with statement:

\begin{verbatim}
with open('write_it.txt', 'w') as text_file:
    text_file.write('Line 1\n')
    text_file.write('This is line 2\n')
    text_file.write('That makes this line 3\n')
```

1.16.3 Adding to a file

Using the 'a' mode we will add to an existing file. We will use the with statement below, but the open/close method will work as well.

```
with open('write_it.txt', 'a') as text_file:
    text_file.write('This is a fourth line\n')
```

1.16.4 Writing multiple lines

Multiple lines may be written using a loop with the *write* method, or may we can use *writelines*:

```
'write_it.txt'
text_file = open('write_it.txt', 'w')
lines = ['Line 1\n',
         'This is line 2\n',
         'That makes this line 3\n']
text_file.writelines(lines)
text_file.close()
```

1.16.5 Reading a file

We can read a file in one line at a time (useful for processing very large files which cannot be held in memory), or we can read in all lines.

Here we read one line at a time:

```
text_file = open('write_it.txt', 'r')
text_file = open('write_it.txt', 'r')
print(text_file.readline())
print(text_file.readline())
print(text_file.readline())
text_file.close()

OUT:

Line 1

This is line 2

That makes this line 3
```

Or we can read the entire file in as a list:

```
text_file = open('write_it.txt', 'r')
lines = text_file.readlines()
text_file.close()
print (lines)

OUT:

['Line 1\n', 'This is line 2\n', 'That makes this line 3\n']
\end{verabtim}

A text file may be stepped through as part of a loop:

\begin{verbatim}
text_file = open('write_it.txt', 'r')
for line in text_file:
    print(line)
text_file.close()
OUT:

Line 1

This is line 2

That makes this line 3
```

1.17 Functions

Functions are (or should be) short sections of code that may be called from within the main programme, or from another function. They usually take some inputs (arguments) and return one or more results.

Functions may be used for two purposes:

1) To avoid repeating code when the same code needs to be run in two or more locations in the main programme.

2) To structure the code, breaking the code down into smaller pieces. In functional programming the whole programme would be broken down into functions

If a function is more than 10 lines long you may want to think whether it could be broken down further, with some of the code pulled out as a separate function.

Ideally a function should do just one thing; it should be as simple as possible, and easy for another person to follow.

A docstring may be used at the start of a function to provide help to the user - in this case if the user types help(function_name) the docstring will be returned. These docstrings may also give examples of the function.

Any variables defined within functions are destroyed when the function closes. Functions are not usually used to change global variables (those variables defined in the main programme code). If a global variable is passed to a function any changes to that variable made by the function are lost when the function ends (and so global variables are protected from being changed by functions). There is a way to force the function to change the global variable, which we will cover in a later lesson, but this should usually be avoided if possible.

1.17.1 Defining a function

A function is defined with the def statement. It is followed by the function's inputs (arguments or 'args') which may also be given a default value. The user may enter the inputs either by name, or in the order in which the function expects them.

It is good practice to use verbs to define functions, to describe what they do.

Let's do a very simple example, raising a number to the power of another number, and we will set a default where, if no other number is given, we will raise to the power of zero (which always returns a value of 1). Note that we define a default value with the = sign.

```
def raise_to_power (number, power = 0):
    result = number ** power
    return result
```

Once the function has been defined we can call it:

```
print (raise_to_power(2,6))
```

```
OUT:
```

```
64
```

We could describe the inputs by their names if we wanted to:

```
print (raise_to_power(number = 2, power = 6))
```

```
OUT:
```

```
64
```

If we define the inputs by name, the order becomes unimportant (without the names the function expects inputs in the order defined in the function):

```
print (raise_to_power(power = 6, number = 2))
```

```
OUT:

64
```

And remember that we set a default of power to zero, so if we only pass the first argument ('number'), the power defaults to zero:

```
print (raise_to_power(2))
```

```
OUT:

1
```

1.17.2 Adding help to a function with a docstring

A help docstring may be added to a function using triple quotes (' or ") after the function has been defined. This may be called later by the user using help.

```
def raise_to_power (number, power = 0):
    """
    This functions takes two numbers, and raises the first number to the power of the second.
    If no second number, or power, is given then power defaults to zero
    """

    result = number ** power
    return result
```

Calling help:

```
help (raise_to_power)
```

```
OUT:

Help on function raise_to_power in module __main__:

raise_to_power(number, power=0)
    This functions takes two numbers, and raises the first number to the power of the second.
    If no second number, or power, is given then power defaults to zero
```

1.17.3 Returning two or more results

Sometimes we may wish to return more than one value. There are two ways of doing this:

1) Return the values separately (the code that calls the argument must also refer to both results:

```
def calculate_sum_and_product(a, b):
    """Return sum and product of two numbers"""

    my_sum = a + b
    my_product = a * b
    return my_sum,my_product

# When calling the function we need to put both results generated by the function

calculated_sum, calculated_product = calculate_sum_and_product(5,6)
print (calculated_sum)
print (calculated_product)
```

```
OUT:

11
30
```

2) A more common way would be to return a single result that is a container (commonly a list or a tuple) of the two results. Here we return a tuple of the two results (remember that tuples are like lists, but they cannot be changed). We can tell that the returned value is a tuple by the use of curved brackets (a list could also be used, which we would spot by square brackets.

```
def calculate_sum_and_product(a, b):
    """Return sum and product of two numbers"""
    my_sum = a + b
    my_product = a * b
    result = (my_sum, my_product) # this is a tuple
    return results

results = calculate_sum_and_product(5,6)

# Individual results are obtained by their index:

print (results[0])
print (results[1])

OUT:

11
30
```

1.18 Lambda functions (one line functions), and map/filter/reduce

Lambda functions are a Pythonic way of writing simple functions in a single line.

They may be applied to a list using the *map* statement.

The function *filter* offers a concise way to filter out all the elements of a list based on the results of a function.

The function *reduce* continually applies a function to a sequence until a single value is returned.

1.18.1 Defining one-line lambda functions

Instead of defining the function with def, we define the function with *lambda*.

Here is a 'normal' function:

```
# A named function that returns the power of a number:

def f(x, y):
    result = x ** y
    return result

print (f(4,3))

OUT:

64
```

Re-writing as a lambda function:

```
g = lambda x,y: x**y

print (g(4,3))

OUT:

64
```

1.18.2 Applying a lambda function to a list with map

In the example below we define a lambda function to convert Celsius to Fahrenheit, and then apply it to a list.

We will first define lambda separately, and then map, before looking at how they can be combined a single line.

The *map* command generates a 'map object'. To return a list we need to convert that to a list as below

```
celsius = [0, 10, 20, 30]
g = lambda x: (9/5)*x + 32
fahrenheit = list(map(g, celsius))

print (fahrenheit)

OUT:

[32.0, 50.0, 68.0, 86.0]
```

This may be compressed further into one line. But there is always a balance to be had between being as concise as possible, and being as clear for others as possible. It may, however, generally be assumed that other experienced Python coders will be familiar with a combined one-line lambda/map function.

```
celsius = [0, 10, 20, 30]
fahrenheit = list(map(lambda x: (9/5)*x + 32, celsius))

print (fahrenheit)

[32.0, 50.0, 68.0, 86.0]
```

1.18.3 Filtering a list with a lambda function

The function *filter* offers a concise way to filter out all the elements of a list, for which the function function returns True. The function filter(f,l) needs a function f as its first argument. f returns a Boolean value, i.e. either True or False.

This function will be applied to every element of the list l. Only if f returns True will the element of the list be included in the result list.

Here is an example where filter is used to return even numbers from a list. As with the map example above we will combine the filter and lambda function in a single line.

```
my_list = [0,1,2,3,4,5,6,7,8,9,10]
result = list(filter(lambda x: x % 2 == 0, my_list))

print (result)

OUT:

[0, 2, 4, 6, 8, 10]
```

1.18.4 Reducing a list with a repeated lambda function call

The reduce function will continually apply a lambda function until only a single value is returned. Though it used to be in core python, it is now found in the functools module.

A simple example is to use reduce to multiply all values in a list. The reduce function replaces the first two elements of a list with the product of those two elements. It will continue until one value is left.

```
import functools
my_list = [10,30,34,56,89]
result = functools.reduce(lambda x,y: x*y, my_list)

print (result)

OUT:

50836800
```

1.18.5 Alternatives to map, filter and reduce

Most results from map/filter and reduce may also be obtained from using list comprehensions, as previously outlined. Some see list comprehensions as more Pythonic than map, filter and reduce.

1.19 Accessing date and time, and timing code

1.19.1 Accessing date and time

It may sometimes be useful to access the current date and/or time. As an example, when writing code it may be useful to access the time at particular stages to monitor how long different parts of code are taking.

To access date and time we will use the *datetime* module (which is held in a package that is also, a little confusingly called *datetime*!):

```
from datetime import datetime
```

To access the current date and time we can use the now method:

```
print (datetime.now())
```

```
OUT:

2018-03-28 08:54:13.623198
```

You might not have expected to get the time to the nearest microsecond! But that may be useful at times when timing short bits of code (which may have to run many times).

But we can access the date and time in different ways:

```
current_datetime = datetime.now()

print ('Date:', current_datetime.date()) # note the () after date
print ('Year:', current_datetime.year)
print ('Month:', current_datetime.month)
print ('Day:', current_datetime.day)
print ('Time:', current_datetime.time()) # note the () after time
print ('Hour:', current_datetime.hour)
print ('Minute:', current_datetime.minute)
print ('Second:', current_datetime.second)
```

```
OUT:

Date: 2018-03-28
Year: 2018
Month: 3
Day: 28
Time: 08:54:13.636654
Hour: 8
Minute: 54
Second: 13
```

Having microseconds in the time may look a little clumsy. So let's format the date and using the *strftime* method.

```
now = datetime.now()
print (now.strftime('%Y-%m-%d'))
print (now.strftime('%H:%M:%S'))
```

```
OUT:

2018-03-28
08:54:13
```

1.19.2 Timing code

We can use datetime to record the time before and after some code to calculate the time taken, but it is a little simpler to use the time module which keeps all time in seconds (from January 1st 1970) and gives even more accuracy to the time:

```
import time

time_start = time.time()

for i in range(100000):
    x = i ** 2

time_end = time.time()
time_taken = time_end - time_start

print ('Time taken:', time_taken)

OUT:

Time taken: 0.03149104118347168
```

Chapter 2

NumPy and Pandas

2.1 Numpy and Pandas

For much of healthcare data analytics and modelling we will use NumPy and Pandas as the key containers of our data. These libraries allow efficient manipulation and analysis of very large data sets. They are very considerably faster than using a 'pure Python' approach.

NumPy and Pandas are distributed with all main scientific Python distributions.

2.1.1 NumPy

NumPy is a library for supporting work with large multi-dimensional arrays, with many mathematical functions. NumPy is compatible with many other libraries such as Pandas (see below), MatPlotLib (for plotting), and many Python maths, stats and optimisation libraries.

When we import NumPy as as library it is standard practice to use the *as* statement which allows it to be referenced with a shorter name. We will import as *np*:

```
import numpy as np
```

2.1.2 Pandas

Pandas is a library that allows manipulation of large arrays of data. Data may be indexed and manipulated based on index. Data is readily pivoted, reshaped, grouped, merged.

We will use pandas alongside numpy. Generally, NumPy is faster for mathemetical functions, but Pandas is more powerful for data manipulation.

As with numpy, we import with a shortened name:

```
import pandas as pd
```

2.2 Numpy basics: building an array from lists, basic statistics, converting to booleans, referencing the array, and taking slices

Most commonly we will be loading files into NumPy arrays, but here we build an array from lists and perform some basics stats on the array.

The examples below construct and use a 2 dimensional array (which may be though of as 'rows and columns'). Later we will look at higher dimensional arrays.

2.2.1 Building an array from lists

We use the *np.array* function to build an array from existing lists. Here each list represents a row of a data table.

```
import numpy as np

row0 = [23, 89,100]
row1 = [10, 51, 99]
row2 = [40, 78, 102]
row3 = [35, 81, 110]
row4 = [50, 75, 95]
row5 = [65, 51, 101]

data = np.array([row0, row1, row2, row3, row4, row5])
```

We now have a data array:

```
print (data)

OUT:

[[ 23  89 100]
 [ 10  51  99]
 [ 40  78 102]
 [ 35  81 110]
 [ 50  75  95]
 [ 65  51 101]]
```

2.2.2 Performing basic statistics on the array

We can see, for example, the mean of all the data

```
print (data.mean())

OUT:

69.72222222222223
```

Or we can use the *axis* argument to show the mean by column or mean by row:

```
print (np.mean(data,axis=0)) # average all rows in each column
print (np.mean(data,axis=1)) # average all columns in each row

OUT:

[ 37.16666667  70.83333333 101.16666667]
[70.66666667 53.33333333 73.33333333 75.33333333 73.33333333 72.33333333]
```

Some other commonly used statistics (all of which are by column, or dimension 0, below):

```
print ('Mean:\n', np.mean(data, axis=0))
print ('Median:\n', np.median(data, axis=0))
print ('Sum:\n', np.sum(data, axis=0))
print ('Maximum:\n', np.max(data,axis=0))
print ('Minimum\n:', np.min(data,axis= 0))
print ('Range:\n', np.ptp(data, axis=0))
print ('10th percentile:\n', np.percentile(data, 10, axis=0))
print ('90th percentile:\n', np.percentile(data, 90, axis=0))
print ('Population standard deviation\n' ,np.std(data, axis=0))
print ('Sample standard deviation:\n', np.std(data, axis=0, ddof=1))
print ('Population variance:\n', np.var(data, axis=0))
print ('Sample variance:\n', np.var(data, axis=0, ddof=1))

OUT:

Mean:
 [ 37.16666667  70.83333333 101.16666667]
Median:
 [ 37.5  76.5 100.5]
Sum:
 [223 425 607]
Maximum:
 [ 65  89 110]
Minimum
: [10 51 95]
Range:
 [55 38 15]
10th percentile:
 [16.5 51.  97. ]
90th percentile:
 [ 57.5  85.  106. ]
Population standard deviation
 [17.75215167 14.6562463   4.52462399]
Sample standard deviation:
 [19.44650783 16.05511341  4.95647724]
Population variance:
 [315.13888889 214.80555556  20.47222222]
Sample variance:
 [378.16666667 257.76666667  24.56666667]
```

The returned array may be referenced by index number (beginning at zero):

```
results = np.mean(data, axis=0)

print (results[0])
OUT:

37.166666666666664
```

Basic python may also be incorporated into the statistics. For example the 10th percentiles, from 0 to 100 may be calculated in a loop (remember that the standard Python range function goes up to, but does not include, the maximum value given, so we need to put a higher maximum than 100 in order to include the 100th percentile in the loop):

```
for percent in range(0,101,10):
    print(percent,'percentile:',np.percentile(data, percent, axis=0))

OUT:
```

```
0 percentile: [10. 51. 95.]
10 percentile: [16.5 51.  97. ]
20 percentile: [23. 51. 99.]
30 percentile: [29.  63.  99.5]
40 percentile: [ 35.  75. 100.]
50 percentile: [ 37.5  76.5 100.5]
60 percentile: [ 40.  78. 101.]
70 percentile: [ 45.   79.5 101.5]
80 percentile: [ 50.  81. 102.]
90 percentile: [ 57.5  85.  106. ]
100 percentile: [ 65.  89. 110.]
```

2.2.3 Converting values to a boolean True/False

We can use test values against some standard. For example, here we test whether a value is equal to, or greater than, the mean of the column.

```
column_mean = np.mean(data, axis=0)
greater_than_mean = data >= column_mean

print (greater_than_mean)

OUT:

[[False  True False]
 [False False False]
 [ True  True  True]
 [False  True  True]
 [ True  True False]
 [ True False False]]
```

True/False values in Python may also be used in calculations where False has an equivalent value to zero, and True has an equivalent value to 1.

```
print (np.mean(greater_than_mean, axis=0))

OUT:

[0.5        0.66666667 0.33333333]
```

2.2.4 Referencing arrays and taking slices

NumPy arrays are referenced similar to lists, except we have two (or more dimensions). For a two dimensional array the reference is [dimension_0, dimension_1], which is equivalent to [row, column].

REMEMBER: Like lists, indices start at index zero, and when taking a slice the slice goes up to, but does not include, the maximum value given.

```
print ('Row zero:\n', data[0,:])
print ('Column one:\n', data[:,1])
# Note in the example below a slice goes up to ,but dot include, the maximum index
print ('Rows 0-3, Column zero:\n', data[0:4,0])
print ('Row 1, Column 2\n', data[1,2])

OUT:

Row zero:
 [ 23  89 100]
Column one:
```

```
 [89 51 78 81 75 51]
Rows 0-3, Column zero:
 [23 10 40 35]
Row 1, Column 2
 99
```

2.3 Pandas basics: building a dataframe from lists, and retrieving data from the dataframe using row and column index references

Here we will repeat basic actions previously described for NumPy. There is significant overlap between NumPy and Pandas (not least because Pandas is built on top of NumPy). Generally speaking Pandas will be used more for data manipulation, and NumPy will be used more for raw calculations (but that is probably somewhat of an over-simplification!).

Pandas allows us to access data using index names or by row/column number. Using index names is perhaps more common in Pandas. You may find having the two different methods available a little confusing at first, but these dual methods are one thing that help make Pandas powerful for data manipulation.

As with NumPy, we will often be importing data from files, but here we will create a dataframe from existing lists.

2.3.1 Creating an empty data frame and building it up from lists

We start with importing pandas (using pd as the short name we will use) and then create a dataframe.

```
import pandas as pd
df = pd.DataFrame()
```

Let's create some data in lists and add them to the dataframe:

```
names = ['Gandolf','Gimli','Frodo','Legolas','Bilbo']
types = ['Wizard','Dwarf','Hobbit','Elf','Hobbit']
magic = [10, 1, 4, 6, 4]
aggression = [7, 10, 2, 5, 1]
stealth = [8, 2, 5, 10, 5]


df['names'] = names
df['type'] = types
df['magic_power'] = magic
df['aggression'] = aggression
df['stealth'] = stealth
```

We can print the dataframe. Notice that a column to the left has appeared with numbers. This is the index, which has been added automatically.

```
print(df)

OUT:
     names    type  magic_power  aggression  stealth
0  Gandolf  Wizard           10           7        8
1    Gimli   Dwarf            1          10        2
2    Frodo  Hobbit            4           2        5
3  Legolas     Elf            6           5       10
4    Bilbo  Hobbit            4           1        5
```

2.3.2 Setting an index column

We can leave the index as it is, or we can make one of the columns the index. Note that to change something in an existing dataframe we use 'inplace=True'

```
df.set_index('names', inplace=True)
print (df)
```

```
OUT:
           type  magic_power  aggression  stealth
names
Gandolf  Wizard           10           7        8
Gimli     Dwarf            1          10        2
Frodo    Hobbit            4           2        5
Legolas     Elf            6           5       10
Bilbo    Hobbit            4           1        5
```

2.3.3 Accessing data with *loc* and *iloc*

Dataframes have two basic methods of accessing data by row (or index) and by column (or header):

loc selects data by index name and column (header) name.

iloc selects data by row or column number

2.3.4 Selecting rows by index

The *loc* method selects rows by index name, like in Python dictionaries:

```
{print (df.loc['Gandolf']

OUT:
print (df.loc['Gandolf'])

type           Wizard
magic_power        10
aggression          7
stealth             8
Name: Gandolf, dtype: object
```

We can pass multiple index references to the *loc* method using a list:

```
to_find = ['Bilbo','Gimli','Frodo']
print (df.loc[to_find])

OUT:
         type  magic_power  aggression  stealth
names
Bilbo  Hobbit            4           1        5
Gimli   Dwarf            1          10        2
Frodo  Hobbit            4           2        5
```

Row slices may also be taken. For example let us take a row slice from Gimli to Legolas. Unusually for Python this slice includes both the lower and upper index references.

```
print (df.loc['Gimli':'Legolas'])

OUT:
           type  magic_power  aggression  stealth
names
Gimli     Dwarf            1          10        2
Frodo    Hobbit            4           2        5
Legolas     Elf            6           5       10
```

As with other Python slices a colon may be used to represent the start or end. :Gimli would take a slice from the beginning to Gimli. Bilbo: would take a row slice from Bilbo to the end.

2.3.5 Selecting records by row number

Rather than using an index, we can use row numbers, using the *iloc* method. As with most references in Python the range given starts from the lower index number and goes up to, but does not include, the upper index number.

```
           type  magic_power  aggression  stealth
names
Gandolf  Wizard           10           7        8
Gimli     Dwarf            1          10        2
```

Discontinuous rows may be accessed with *iloc* by building a list:

```
print (df.iloc[[0,1,4]])

OUT:

           type  magic_power  aggression  stealth
names
Gandolf  Wizard           10           7        8
Gimli     Dwarf            1          10        2
Bilbo    Hobbit            4           1        5
```

Or, building up a more complex list of row numbers:

```
rows_to_find = list(range(0,2))
rows_to_find += (list(range(3,5)))


print ('List of rows to find:',rows_to_find)
print()
print (df.iloc[rows_to_find])

OUT:
List of rows to find: [0, 1, 3, 4]

           type  magic_power  aggression  stealth
names
Gandolf  Wizard           10           7        8
Gimli     Dwarf            1          10        2
Legolas     Elf            6           5       10
Bilbo    Hobbit            4           1        5
```

2.3.6 Selecting columns by name

Columns are selected using square brackets after the dataframe:

```
print (df['type'])

OUT:
names
Gandolf    Wizard
Gimli       Dwarf
Frodo      Hobbit
Legolas       Elf
Bilbo      Hobbit
Name: type, dtype: object

print (df[['type','stealth']])
```

```
OUT:
           type  stealth
names
Gandolf  Wizard        8
Gimli     Dwarf        2
Frodo    Hobbit        5
Legolas     Elf       10
Bilbo    Hobbit        5
```

To take a slice of columns we need to use the *loc* method, using : to select all rows.

```
print (df.loc[:,'magic_power':'stealth'])
```

```
         magic_power  aggression  stealth
names
Gandolf           10           7        8
Gimli              1          10        2
Frodo              4           2        5
Legolas            6           5       10
Bilbo              4           1        5
```

2.3.7 Selecting columns by number

Columns may also be referenced by number using the *column* method (which allows slicing):

```
print (df[df.columns[1:4]])
```

```
         magic_power  aggression  stealth
names
Gandolf           10           7        8
Gimli              1          10        2
Frodo              4           2        5
Legolas            6           5       10
Bilbo              4           1        5
```

Or *iloc* may be used to select columns by number (the colon shows that we are selecting all rows):

```
print (df.iloc[:,1:3])
```

```
OUT:
         magic_power  aggression
names
Gandolf           10           7
Gimli              1          10
Frodo              4           2
Legolas            6           5
Bilbo              4           1
```

2.3.8 Selecting rows and columns simultaneously

We can combine row and column references with the *loc* method:

```
rows_to_find = ['Bilbo','Gimli','Frodo']
print (df.loc[rows_to_find,'magic_power':'stealth'])
```

```
OUT:
rows_to_find = ['Bilbo','Gimli','Frodo']

print (df.loc[rows_to_find,'magic_power':'stealth'])
```

```
       magic_power  aggression  stealth
names
Bilbo            4           1        5
Gimli            1          10        2
Frodo            4           2        5
```

Or with *iloc* (referencing row numbers):

```
print (df.iloc[0:2,2:4])
```

```
OUT:
print (df.iloc[0:2,2:4])

         aggression  stealth
names
Gandolf           7        8
Gimli            10        2
```

2.4 Basic statistics in Pandas

Like NumPy, Pandas may be used to give us some basic statistics on data.

Let's start by building a very sample dataframe.

```
import pandas as pd
df = pd.DataFrame()

names = ['Gandolf','Gimli','Frodo','Legolas','Bilbo']
types = ['Wizard','Dwarf','Hobbit','Elf','Hobbit']
magic = [10, 1, 4, 6, 4]
aggression = [7, 10, 2, 5, 1]
stealth = [8, 2, 5, 10, None]


df['names'] = names
df['type'] = types
df['magic_power'] = magic
df['aggression'] = aggression
df['stealth'] = stealth
```

2.4.1 Overview statistics

We can get an overview with the describe() method.

```
print (df.describe())
```

```
OUT:
       magic_power  aggression  stealth
count     5.000000    5.000000     4.00
mean      5.000000    5.200000     6.25
std       3.316625    3.420526     3.50
min       1.000000    2.000000     2.00
25%       4.000000    2.000000     4.25
50%       4.000000    5.000000     6.50
75%       6.000000    7.000000     8.50
max      10.000000   10.000000    10.00
```

We can modify the percentiles reported:

```
print (df.describe(percentiles=[0.05,0.1,0.9,0.95]))
```

```
OUT:

       magic_power  aggression  stealth
count     5.000000    5.000000     4.00
mean      5.000000    5.200000     6.25
std       3.316625    3.420526     3.50
min       1.000000    2.000000     2.00
5%        1.600000    2.000000     2.45
10%       2.200000    2.000000     2.90
50%       4.000000    5.000000     6.50
90%       8.400000    8.800000     9.40
95%       9.200000    9.400000     9.70
max      10.000000   10.000000    10.00
```

Specific statistics may be returned:

```
print (df.mean())
```

```
OUT:
magic_power    5.00
aggression     5.20
stealth        6.25
dtype: float64
```

2.4.2 List of key statistical methods

mean() = mean

median() = median

min() = minimum

max() =maximum

quantile(x)

var() = variance

std() = standard deviation

mad() = mean absolute variation

skew() = skewness of distribution

kurt() = kurtosis

cov() = covariance

corr() = Pearson Correlation coefficent

autocorr() = autocorelation

diff() = first discrete difference

cumsum() = cummulative sum

comprod() = cumulative product

cummin() = cumulative minimum

2.4.3 Returning the index of minimum and maximum

idxmin and *idxmax* will return the index row of the min/max. If two values are equal the first will be returned.

```
print ('Minimum:', df['aggression'].min())
print ('Index row:',df['aggression'].idxmin())
print ('\nFull row:\n', df.iloc[df['aggression'].idxmin()])
```

```
OUT:
Minimum: 2
Index row: 2

Full row:
 names           Frodo
type           Hobbit
magic_power         4
aggression          2
stealth             5
Name: 2, dtype: object
```

2.4.4 Removing rows with incomplete data

We can extract only those rows with a complete data set using the *dropna()* method.

```
print (df.dropna())
```

```
OUT:
```

```
     names    type  magic_power  aggression  stealth
0  Gandolf  Wizard           10           7      8.0
1    Gimli   Dwarf            1          10      2.0
2    Frodo  Hobbit            4           2      5.0
3  Legolas     Elf            6           5     10.0
```

We can use this directly in the describe method.

```
print (df.dropna().describe())
```

```
OUT:
```

```
magic_power  aggression  stealth
count     4.000000    4.000000     4.00
mean      5.250000    6.000000     6.25
std       3.774917    3.366502     3.50
min       1.000000    2.000000     2.00
25%       3.250000    4.250000     4.25
50%       5.000000    6.000000     6.50
75%       7.000000    7.750000     8.50
max      10.000000   10.000000    10.00
```

To create a new dataframe with compplete rows only, we would simply assign to a new variable name:

```
df_na_dropped = df.dropna()
```

2.5 Converting between NumPy and Pandas

Conversion between NumPy and Pandas is simple.

Let's start with importing NumPy and Pandas, and then make a Pandas dataframe.

```
import numpy as np
import pandas as pd

df = pd.DataFrame()

names = ['Gandolf','Gimli','Frodo','Legolas','Bilbo']
types = ['Wizard','Dwarf','Hobbit','Elf','Hobbit']
magic = [10, 1, 4, 6, 4]
aggression = [7, 10, 2, 5, 1]
stealth = [8, 2, 5, 10, 5]


df['names'] = names
df['type'] = types
df['magic_power'] = magic
df['aggression'] = aggression
df['stealth'] = stealth

print (df)

OUT:
     names    type  magic_power  aggression  stealth
0  Gandolf  Wizard           10           7        8
1    Gimli   Dwarf            1          10        2
2    Frodo  Hobbit            4           2        5
3  Legolas     Elf            6           5       10
4    Bilbo  Hobbit            4           1        5
```

2.5.1 Converting from Pandas to NumPy

We will use the values method to convert from Pandas to NumPy. Notice that we loose our column headers when converting to a NumPy array, and the index filed (name) simply becomes the first column.

```
my_array = df.values

print (my_array)

OUT:

[['Gandolf' 'Wizard' 10 7 8]
 ['Gimli' 'Dwarf' 1 10 2]
 ['Frodo' 'Hobbit' 4 2 5]
 ['Legolas' 'Elf' 6 5 10]
 ['Bilbo' 'Hobbit' 4 1 5]]
```

2.5.2 Converting from NumPy to Pandas

We will use the dataframe method to convert from a NumPy array to a Pandas dataframe. A new index has been created, and columns have been given numerical headers.

```
my_new_df = pd.DataFrame(my_array)
```

```
print (my_new_df)

OUT:

         0       1   2   3   4
0  Gandolf  Wizard  10   7   8
1    Gimli   Dwarf   1  10   2
2    Frodo  Hobbit   4   2   5
3  Legolas     Elf   6   5  10
4    Bilbo  Hobbit   4   1   5
```

If we have column names, we can supply those to the dataframe during the conversion process. We pass a list to the dataframe method:

```
names = ['name','type','magic_power','aggression','strength']

my_new_df = pd.DataFrame(my_array, columns=names)

print(my_new_df)

OUT:

      name    type magic_power aggression strength
0  Gandolf  Wizard          10         7        8
1    Gimli   Dwarf           1        10        2
2    Frodo  Hobbit           4         2        5
3  Legolas     Elf           6         5       10
4    Bilbo  Hobbit           4         1        5
```

And, as we have seen previously, we can set the index to a particular column:

```
my_new_df.set_index('name', inplace=True)

print (my_new_df)

OUT:

           type magic_power aggression strength
name
Gandolf  Wizard          10         7        8
Gimli     Dwarf           1        10        2
Frodo    Hobbit           4         2        5
Legolas     Elf           6         5       10
Bilbo    Hobbit           4         1        5
```

2.6 Array maths in NumPy

NumPy allows easy standard mathematics to be performed on arrays, a well as moire complex linear algebra such as array multiplication.

Lets begin by building a couple of arrays. We'll use the np.arange method to create an array of numbers in range 1 to 12, and then reshape the array into a 3 x 4 array.

```
import numpy as np

# note that the arange method is 'half open'
# that is is includes the lower number, and goes up yo, but not including,
# the higher number

array_1 = np.arange(1,13)
array_1 = array_1.reshape (3,4)

print (array_1)

OUT:

[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

2.6.1 Maths on a single array

We can multiple an array by a fixed number (or we can add, subtract, divide, raise to power, etc):

```
print (array_1 *4)

OUT:

[[ 4  8 12 16]
 [20 24 28 32]
 [36 40 44 48]]


print (array_1 ** 0.5) # square root of array

OUT:

[[1.         1.41421356 1.73205081 2.        ]
 [2.23606798 2.44948974 2.64575131 2.82842712]
 [3.         3.16227766 3.31662479 3.46410162]]
```

We can define a vector and multiple all rows by that vector:

```
vector_1 = [1, 10, 100, 1000]

print (array_1 * vector_1)

OUT:

[[    1    20   300  4000]
 [    5    60   700  8000]
 [    9   100  1100 12000]]
```

To multiply by a column vector we will transpose the original array, multiply by our column vector, and transpose back:

```
vector_2 = [1, 10, 100]

result = (array_1.T * vector_2).T

print (result)

OUT:

[[   1    2    3    4]
 [  50   60   70   80]
 [ 900 1000 1100 1200]]
```

2.6.2 Maths on two (or more) arrays

Arrays of the same shape may be multiplied, divided, added, or subtracted.

Let's create a copy of the first array:

```
array_2 = array_1.copy()

# If we said array_2 = array_1 then array_2 would refer to array_1.
# Any changes to array_1 would also apply to array_2
```

Multiplying two arrays:

```
print (array_1 * array_2)

OUT:

[[  1   4   9  16]
 [ 25  36  49  64]
 [ 81 100 121 144]]
```

2.6.3 Matrix multiplication ('dot product')

See `https://www.mathsisfun.com/algebra/matrix-multiplying.html` for an explanation of matrix multiplication, if you are not familiar with it.

We can perform matrix multiplication in numpy with the np.dot method.

```
array_2 = np.arange(1,13)
array_2 = array_1.reshape (4,3)

print ('Array 1:')
print (array_1)
print ('\nArray 2:')
print (array_2)
print ('\nDot product of two arrays:')
print (np.dot(array_1, array_2))

OUT:

Array 1:
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]

Array 2:
[[ 1  2  3]
 [ 4  5  6]
```

```
 [ 7  8  9]
 [10 11 12]]

Dot product of two arrays:
[[ 70  80  90]
 [158 184 210]
 [246 288 330]]
```

2.7 Reading and writing CSV files using NumPy and Pandas

Here we will load a CSV called iris.csv. This is stored in the same directory as the Python code.

As a general rule, using the Pandas import method is a little more 'forgiving', so if you have trouble reading directly into a NumPy array, try loading in a Pandas dataframe and then converting to a NumPy array.

2.7.1 Reading a csv file into a NumPy array

NumPy's *loadtxt* method reads delimited text. We specify the separator as a comma. The data we are loading also has a text header, so we use *skiprows=1* to skip the header row, which would cause problems for NumPy.

```
import numpy as np

my_array = np.loadtxt('iris_numbers.csv',delimiter=",", skiprows=1)

print (my_array[0:5,:]) # first 5 rows

OUT:

[[5.1 3.5 1.4 0.2 1. ]
 [4.9 3.  1.4 0.2 1. ]
 [4.7 3.2 1.3 0.2 1. ]
 [4.6 3.1 1.5 0.2 1. ]
 [5.  3.6 1.4 0.2 1. ]]
```

2.7.2 Saving a NumPy array as a csv file

We use the *savetxt* method to save to a csv.

```
np.savetxt("saved_numpy_data.csv", my_array, delimiter=",")
```

2.7.3 Reading a csv file into a Pandas dataframe

The *read_csv* will read a CSV into Pandas. This import assumes that there is a header row. If there is no header row, then the argument *header = None* should be used as part of the command. Notice that a new index column is created.

```
import pandas as pd

df = pd.read_csv('iris.csv')

print (df.head(5)) #  First 5 rows

OUT:

   sepal.length  sepal.width  petal.length  petal.width variety
0           5.1          3.5           1.4          0.2  Setosa
1           4.9          3.0           1.4          0.2  Setosa
2           4.7          3.2           1.3          0.2  Setosa
3           4.6          3.1           1.5          0.2  Setosa
4           5.0          3.6           1.4          0.2  Setosa
```

2.7.4 Saving a Pandas dataframe to a CSV file

The *to_csv* will save a dataframe to a CSV. By default column names are saved as a header, and the index column is saved. If you wish not to save either of those use *header=True* and/or *index=True* in the command. For example, in the command below we save the dataframe with headers, but not with the index column.

```
df.to_csv('my_pandas_dataframe.csv', index=False)
```

2.8 Applying user-defined functions to NumPy and Pandas

Both NumPy and Pandas allow user to functions to applied to all rows and columns (and other axes in NumPy, if multidimensional arrays are used)

2.8.1 Numpy

In NumPy we will use the *apply_along_axis* method to apply a user-defined function to each row and column.

Let's first set up a array and define a function. We will use a simple user-defined function for illustrative purposes - one that returns the position of the highest value in the slice passed to the function. In NumPy we use *argmax* for finding the position of the highest value.

```
import numpy as np
import pandas as pd

my_array = np.array([[10,2,13],
                     [21,22,23],
                     [31,32,33],
                     [10,57,20],
                     [20,20,20],
                     [101,91,10]])


def my_function(x):
    position = np.argmax(x)
    return position
```

Using *axis=0* we can apply that function to all columns:

```
print (np.apply_along_axis(my_function, axis=0, arr=my_array))
```

```
OUT:
[5 5 2]
```

Using *axis=1* we can apply that function to all rows:

```
print (np.apply_along_axis(my_function, axis=1, arr=my_array))
```

```
OUT:
[2 2 2 1 0 0]
```

2.8.2 Pandas

Pandas has a similar method, the apply method for applying a user function by either row or column. The Pandas method for determining the position of the highest value is *idxmax*.

We will convert our NumPy array to a Pandas dataframe, define our function, and then apply it to all columns. Notice that because we are working in Pandas the returned value is a Pandas series (equivalent to a DataFrame, but with one one axis) with an index value.

```
import pandas as pd

df = pd.DataFrame(my_array)

def my_function(x):
    z= x.idxmax()
    return z
```

```
print(df.apply(my_function, axis=0))
```

```
OUT:
0    5
1    5
2    2
dtype: int64
```

And applying it to all rows:

```
print(df.apply(my_function, axis=1))
```

```
OUT:
0    2
1    2
2    2
3    1
4    0
5    0
dtype: int64
```

2.9 Adding data to NumPy arrays and Pandas dataframes

2.9.1 Numpy

Adding more rows of data

To add more rows to an existing numpy array use the *vstack* method which can add multiple or single rows. New data may be in the form of a numpy array or a list. All combined data must have the same number of columns.

```
import numpy as np

# Starting with a NumPy array
array1 = np.array([[1,2,3,4,5],
         [6,7,8,9,10],
         [11,12,13,14,15]])

# An additional 2d list
array2 = [[16,17,18,19,20],
         [21,22,23,24,25]]

# An additional single row Numpy array
array3 = np.array([26,27,28,29,30])

# We will combine all data into existing array, array1
# But a new name could be given
array1 = np.vstack([array1, array2, array3])

print (array1)

OUT:

[[ 1  2  3  4  5]
 [ 6  7  8  9 10]
 [11 12 13 14 15]
 [16 17 18 19 20]
 [21 22 23 24 25]
 [26 27 28 29 30]]
```

Adding more columns of data

To add more columns to an existing numpy array use the *hstack* method which can add multiple or single rows. All combined data must have the same number of rows.

```
df1 = pd.DataFrame()
names = ['Gandolf','Gimli','Frodo','Legolas','Bilbo']
types = ['Wizard','Dwarf','Hobbit','Elf','Hobbit']

df1['names'] = names
df1['type'] = types

print (df1)

# Add another column
magic = [10, 1, 4, 6, 4]
df1['magic'] = magic
```

```
print ('\n Added column:\n',df1)

OUT:

     names    type
0  Gandolf  Wizard
1    Gimli   Dwarf
2    Frodo  Hobbit
3  Legolas     Elf
4    Bilbo  Hobbit

 Added column:
     names    type  magic
0  Gandolf  Wizard     10
1    Gimli   Dwarf      1
2    Frodo  Hobbit      4
3  Legolas     Elf      6
4    Bilbo  Hobbit      4
```

We can use *concat* also to add multiple columns (in the form of another dataframe), in which case the data will be combined based on the index column. We pass the argument *axis=1* to the concat statement to instruct the method to combine by column (it defaults to axis=0, or row concatenation).

```
df1 = pd.DataFrame()
names = ['Gandolf','Gimli','Frodo','Legolas','Bilbo']
types = ['Wizard','Dwarf','Hobbit','Elf','Hobbit']

df1['names'] = names
df1['type'] = types

print (df1)

df2 = pd.DataFrame()

magic = [10, 1, 4, 6, 4]
aggression = [7, 10, 2, 5, 1]
stealth = [8, 2, 5, 10, 5]

df2['magic_power'] = magic
df2['aggression'] = aggression
df2['stealth'] = stealth

df1 = pd.concat([df1,df2], axis=1)
print(df1)

OUT:

df1 = pd.concat([df1,df2], axis=1)

print(df1)

     names    type  magic_power  aggression  stealth
0  Gandolf  Wizard           10           7        8
1    Gimli   Dwarf            1          10        2
2    Frodo  Hobbit            4           2        5
3  Legolas     Elf            6           5       10
4    Bilbo  Hobbit            4           1        5
```

There is more information here: `https://pandas.pydata.org/pandas-docs/stable/merging.html`

2.10 Using Pandas to merge or lookup data

Sometimes we may want to cross-reference data between different data tables. This may be in order to perform a full merge of data, or just to produce a summary lookup table referencing across different tables.

```
import numpy as np
import pandas as pd

# Set up the first data frame

df1 = pd.DataFrame()

names = ['Gandolf','Gimli','Frodo','Legolas','Bilbo']
types = ['Wizard','Dwarf','Hobbit','Elf','Hobbit']
magic = [10, 1, 4, 6, 4]
aggression = [7, 10, 2, 5, 1]
stealth = [8, 2, 5, 10, 5]

df1['names'] = names
df1['type'] = types
df1['magic_power'] = magic
df1['aggression'] = aggression
df1['stealth'] = stealth

# Set up the second dataframe

names = ['Gandolf','Gimli','Frodo','Aragorn','Sauron']
popularity = ['High','Medium','High','Medium','Low']

df2 = pd.DataFrame()

df2['name'] = names
df2['popularity'] = popularity
```

We will look here at where the reference fields are not the index fields. We are going to want to merge using 'names' in df1 and 'name' in df2, and we are going to keep all records from df1, and add data from df2 where it is available.

```
merged_df = pd.merge(df1, df2,
                     left_on = 'names',
                     right_on = 'name',
                     how='left')

print (merged_df)

OUT:

     names    type  magic_power  aggression  stealth     name popularity
0  Gandolf  Wizard           10           7        8  Gandolf       High
1    Gimli   Dwarf            1          10        2    Gimli     Medium
2    Frodo  Hobbit            4           2        5    Frodo       High
3  Legolas     Elf            6           5       10      NaN        NaN
4    Bilbo  Hobbit            4           1        5      NaN        NaN
```

We can keep all data from both databases by using *how=outer* (an outer database join).

```
merged_df = pd.merge(df1, df2,
                     left_on = 'names',
                     right_on = 'name',
```

```
                    how='outer')

print (merged_df)


OUT:

     names    type  magic_power  aggression  stealth     name popularity
0  Gandolf  Wizard         10.0         7.0      8.0  Gandolf       High
1    Gimli   Dwarf          1.0        10.0      2.0    Gimli     Medium
2    Frodo  Hobbit          4.0         2.0      5.0    Frodo       High
3  Legolas     Elf          6.0         5.0     10.0      NaN        NaN
4    Bilbo  Hobbit          4.0         1.0      5.0      NaN        NaN
5      NaN     NaN          NaN         NaN      NaN  Aragorn     Medium
6      NaN     NaN          NaN         NaN      NaN   Sauron        Low
```

Or we could use *how=inner* to produce a dataframe for only the rows with reference columns in both dataframes, or *how=right* to keep all rows in df2 and add data from df1 where it exists.

In the above examples we have returned all fields both both dataframes. We can choose to select just the fields we wish to return (though we need the reference field from both dataframes):

```
merged_df = pd.merge(df1[['names','type']],
                     df2[['name','popularity']],
                     left_on = 'names',
                     right_on = 'name',
                     how='left')

print (merged_df)

OUT:

     names    type     name popularity
0  Gandolf  Wizard  Gandolf       High
1    Gimli   Dwarf    Gimli     Medium
2    Frodo  Hobbit    Frodo       High
3  Legolas     Elf      NaN        NaN
4    Bilbo  Hobbit      NaN        NaN
```

Using this method we need our reference fields, used to join data, as part of the dataframe and not as the index. To use this method where there is an index column that you wish to use as a joining field you will need to reset the index to a new numbered column. That is done by the command *df.reset_index()*. Alternatively you can use a method that joins using the index fields of each dataframe, as described in the link below.

There are many ways of joining and merging Pandas dataframes. I have set out just one method here. See `https://pandas.pydata.org/pandas-docs/stable/merging.html` for all methods.

2.11 Sorting and ranking with Pandas

2.11.1 Sorting

Pandas allows easy and flexible sorting.

As usual, let's first build a dataframe:

```
import pandas as pd
df = pd.DataFrame()

names = ['Gandolf','Gimli','Frodo','Legolas','Bilbo']
types = ['Wizard','Dwarf','Hobbit','Elf','Hobbit']
magic = [10, 1, 4, 6, 4]
aggression = [7, 10, 2, 5, 1]
stealth = [8, 2, 5, 10, 5]

df['names'] = names
df['type'] = types
df['magic_power'] = magic
df['aggression'] = aggression
df['stealth'] = stealth
```

And now let's sort first by magic power and then (in reverse order aggression.

```
new_df = df.sort_values(['magic_power','aggression'], ascending=[False,True])
print (new_df)

OUT:
```

```
     names    type  magic_power  aggression  stealth
0  Gandolf  Wizard           10           7        8
3  Legolas     Elf            6           5       10
4    Bilbo  Hobbit            4           1        5
2    Frodo  Hobbit            4           2        5
1    Gimli   Dwarf            1          10        2
```

Usually it is fine to use the default sorting method. Sometimes though you may wish to do a series of sequential sorts where you maintain the previous order within the sorted the dataframe. In that case use a mergesort by passing *kind = 'mergesort'* as one of the arguments.

We can use *sort_index* to sort by the index field. Let's sort our new dataframe by reverse index order:

```
print (new_df.sort_index(ascending=False))
i
```

```
     names    type  magic_power  aggression  stealth
4    Bilbo  Hobbit            4           1        5
3  Legolas     Elf            6           5       10
2    Frodo  Hobbit            4           2        5
1    Gimli   Dwarf            1          10        2
0  Gandolf  Wizard           10           7        8
```

2.11.2 Ranking

Pandas allows easy ranking of dataframes by a single column. Where two values are identical the result is the average of the number of ranks they would cover. Notice that a higher number is a higher rank.

```
i
print (df['magic_power'].rank())

OUT:
```

```
0    5.0
1    1.0
2    2.5
3    4.0
4    2.5
Name: magic_power, dtype: float64
```

Pandas does not offer a direct method for ranking using multiple columns. One way would be to sort the dataframe, reset the index with *df.reset_index()* and compare the index values to the original table.

2.12 Using masks to filter data, and perform search and replace, in NumPy and Pandas

In both NumPy and Pandas we can create masks to filter data. Masks are 'Boolean' arrays - that is arrays of true and false values and provide a powerful and flexible method to selecting data.

2.12.1 NumPy

creating a mask

Let's begin by creating an array of 4 rows of 10 columns of uniform random number between 0 and 100.

```
import numpy as np

array1 = np.random.randint(0,100,size=(4,10))

print (array1)

OUT:

[[68 56 72 91 64 98  3 54 49 67]
 [ 1  6 54 65 24 97 68  9 28 47]
 [30 88 52 11 22 12 35 65 66  3]
 [13 83 81 32 87 74 79 34 26  1]]
```

Now we'll create a mask to show those numbers greater than 70.

```
mask = array1 > 70

print(mask)

OUT:

[[False False  True  True False  True False False False False]
 [False False False False False  True False False False False]
 [False  True False False False False False False False False]
 [False  True  True False  True  True  True False False False]]
```

We can use that mask to extract the numbers:

```
print (array1[mask])

OUT:
[72 91 98 97 88 83 81 87 74 79]
```

Using *any* and *all*

any and all allow us to check for all true or all false.

We can apply that to the whole array:

```
print (mask.any())
print (mask.all())

OUT:

True
False
```

Or we can apply it column-wise (by passing *axis=1*) or row-wise (by passing *axis=1*).

```
print ('All tests in a column are true:')
print (mask.all(axis=0))
print ('\nAny test in a row is true:')
print (mask.any(axis=1))

OUT:

All tests in a column are true:
[False False False False False False False False False False]

Any test in a row is true:
[False  True  True  True]
```

We can use != to invert a mask if needed (all trues become false, and all falses become true). This can be useful, but can also become a little confusing!

```
inverted_mask = mask!=True
print (inverted_mask)

OUT:

[[ True  True  True  True  True  True  True  True  True  True]
 [ True  True False False  True False  True False False False]
 [False False False  True False  True  True  True False  True]
 [ True False  True  True  True  True  True  True  True  True]]
```

Adding or averaging trues

Boolean values (True/False) in Python also take the values 1 and 0. This can be useful for counting trues/false, for example:

```
print ('Number of trues in array:')
print (mask.sum())

OUT:
Number of trues in array:
12

print('Number of trues in array by row:')
print (mask.sum(axis=1))

OUT:
Number of trues in array by row:
[0 6 5 1]

print('Average of trues in array by column:')
print (mask.mean(axis=0))

OUT:
Average of trues in array by column:
[0.25 0.5  0.5  0.25 0.25 0.25 0.   0.25 0.5  0.25]
```

Selecting rows or columns based on one value in that row or column

Let's select all columns where the value of the first element is equal to, or greater than 50:

```
mask = array1[0,:] >= 50 # colon indicates all columns, zero indicates row 0
```

```
print ('\nHere is the mask')
print (mask)
print ('\nAnd here is the mask applied to all columns')
print (array1[:,mask]) # colon represents all rows of chosen columns

OUT:

Here is the mask
[ True  True False False  True False False False False  True]

And here is the mask applied to all columns
[[60 68 57 57]
 [47 52 64 74]
 [77 78 79 21]
 [11 78  5 14]]
```

Similarly if we wanted to select all rows where the 2nd element was equal to, or greater, than 50

```
mask = array1[:,1] >= 50 # colon indicates all rows, 1 indicates row 1)
print ('\nHere is the mask')
print (mask)
print ('\nAnd here is the mask applied to all rows')
print (array1[mask,:]) # colon represents all rows of chosen columns

OUT:


Here is the mask
[ True  True  True  True]

And here is the mask applied to all rows
[[60 68 34 25 57 33 49  5 33 57]
 [47 52 78 89 64 75  8 98 93 74]
 [77 78 74 41 79 50 43 21 81 21]
 [11 78 39 18  5 67 69  1 50 14]]
```

Using *and* and *or*, and combining filters from two arrays

We may create and combine multiple masks. For example we may have two masks that look for values less than 20 or greater than 80, and then combine those masks with or which is represented by — (stick).

```
print ('Mask for values <20:')
mask1 = array1 < 20
print (mask1)

print ('\nMask for values >80:')
mask2 = array1 > 80
print (mask2)

print ('\nCombined mask:')
mask = mask1  | mask2 # | (stick) is used for 'or' with two boolean arrays
print (mask)

print ('\nSelected values using combined mask')
print (array1[mask])

OUT:

Mask for values <20:
```

```
[[False False False False False False False  True False False]
 [False False False False False False  True False False False]
 [False False False False False False False False False False]
 [ True False False  True  True False False  True False  True]]

Mask for values >80:
[[False False False False False False False False False False]
 [False False False  True False False False  True  True False]
 [False False False False False False False False  True False]
 [False False False False False False False False False False]]

Combined mask:
[[False False False False False False False  True False False]
 [False False False  True False False  True  True  True False]
 [False False False False False False False False  True False]
 [ True False False  True  True False False  True False  True]]

Selected values using combined mask
[ 5 89  8 98 93 81 11 18  5  1 14]
```

We can combine these masks in a single line:

```
mask = (array1 < 20) | (array1 > 80)
print (mask)
```

```
OUT:

mask = (array1 < 20) | (array1 > 80)

print (mask)

[[False False False False False False False  True False False]
 [False False False  True False False  True  True  True False]
 [False False False False False False False False  True False]
 [ True False False  True  True False False  True False  True]]
```

We can combine masks derived from different arrays, so long as they are the same shape. For example let's produce an another array of random numbers and check for those element positions where corresponding positions of both arrays have values of greater than 50. When comparing boolean arrays we represent 'and' with &.

```
array2 = np.random.randint(0,100,size=(4,10))

print ('Mask for values of array1 > 50:')
mask1 = array1 > 50
print (mask1)

print ('\nMask for values of array2 > 50:')
mask2 = array2 > 50
print (mask2)

print ('\nCombined mask:')
mask = mask1  & mask2
print (mask)
```

```
OUT:

Mask for values of array1 > 50:
[[ True  True False False  True False False False False  True]
 [False  True  True  True  True  True False  True  True  True]
```

```
 [ True  True  True False  True False False False  True False]
 [False  True False False False  True  True False False False]]

Mask for values of array2 > 50:
[[ True False  True False False  True  True False False  True]
 [ True False  True  True False  True  True  True False False]
 [False  True  True False  True False  True  True  True False]
 [ True False  True False False  True  True  True  True  True]]

Combined mask:
[[ True False False False False False False False False  True]
 [False False  True  True False  True False  True False False]
 [False  True  True False  True False False False  True False]
 [False False False False False  True  True False False False]]
```

We could shorten this to:

```
mask = (array1 > 50) & (array2 > 50)
print (mask)

OUT:

[[ True False False False False False False False False  True]
 [False False  True  True False  True False  True False False]
 [False  True  True False  True False False False  True False]
 [False False False False False  True  True False False False]]
```

Setting values based on mask

We can use masks to reassign values only for elements that meet the given criteria. For example we can set the values of all cells with a value less than 50 to zero, and set all other values to 1.

```
print ('Array at sttart:')
print (array1)
mask = array1 < 50
array1[mask] = 0
mask = mask != True # invert mask
array1[mask] = 1
print('\nNew array')
print (array1)

OUT:

Array at sttart:
[[60 68 34 25 57 33 49  5 33 57]
 [47 52 78 89 64 75  8 98 93 74]
 [77 78 74 41 79 50 43 21 81 21]
 [11 78 39 18  5 67 69  1 50 14]]

New array
[[1 1 0 0 1 0 0 0 0 1]
 [0 1 1 1 1 1 0 1 1 1]
 [1 1 1 0 1 1 0 0 1 0]
 [0 1 0 0 0 1 1 0 1 0]]
```

We can shorten this, by making the mask implicit in the assignment command.

```
array2[array2<50] = 0
array2[array2>=50] = 1
```

```
print('New array2:')
print(array2)

OUT:

New array2:
[[1 0 1 0 0 1 1 0 0 1]
 [1 0 1 1 0 1 1 1 0 0]
 [0 1 1 0 1 0 1 1 1 0]
 [1 0 1 0 0 1 1 1 1 1]]
```

Miscellaneous examples

Select columns where the average value across the column is greater than the average across the whole array, and return both the columns and the column number.

```
array = np.random.randint(0,100,size=(4,10))
number_of_columns = array.shape[1]
column_list = np.arange(0, number_of_columns) # create a list of column ids
array_average = array.mean()
column_average = array.mean(axis=0)
column_average_greater_than_array_average = column_average > array_average
selected_columns = column_list[column_average_greater_than_array_average]
selected_data = array[:,column_average_greater_than_array_average]

print ('Selected columns:')
print (selected_columns)
print ('\nSeelcted data:')
print (selected_data)

OUT:

Selected columns:
[1 2 4 9]

Seelcted data:
[[56 48 97 78]
 [26 87  6 45]
 [56 65 71 59]
 [41 34 98 70]]
```

2.12.2 Pandas

Filtering with masks in Pandas is very similar to numpy. It is perhaps more usual in Pandas to be creating masks testing specific columns, with resulting selection of rows. For example let's use a mask to select characters meeting conditions on magical power and aggression:

```
i
import pandas as pd

df = pd.DataFrame()

names = ['Gandolf','Gimli','Frodo','Legolas','Bilbo']
types = ['Wizard','Dwarf','Hobbit','Elf','Hobbit']
magic = [10, 1, 4, 6, 4]
aggression = [7, 10, 2, 5, 1]
stealth = [8, 2, 5, 10, 5]
```

```
df['names'] = names
df['type'] = types
df['magic_power'] = magic
df['aggression'] = aggression
df['stealth'] = stealth

mask = (df['magic_power'] > 3) & (df['aggression'] < 5)
print ('Mask:')
print (mask) # notice mask is a 'series'; a one dimensial DataFrame
# when passing a Boolean series to a dataframe we select the appropriate rows
filtered_data = df[mask]
print (filtered_data)

OUT:

Mask:
0    False
1    False
2     True
3    False
4     True
dtype: bool
   names    type  magic_power  aggression  stealth
2  Frodo  Hobbit            4           2        5
4  Bilbo  Hobbit            4           1        5
```

Though creating masks based on particular columns will be most common in Pandas. We can also filter on the entire dataframe. Look what happens when we filter on values >3:

```
mask = df > 3
print('Mask:')
print (mask)
print ('\nMasked data:')
df2 = df[mask]
print (df2)

OUT:

Mask:
   names  type  magic_power  aggression  stealth
0   True  True         True        True     True
1   True  True        False        True    False
2   True  True         True       False     True
3   True  True         True        True     True
4   True  True         True       False     True

Masked data:
     names    type  magic_power  aggression  stealth
0  Gandolf  Wizard         10.0         7.0      8.0
1    Gimli   Dwarf          NaN        10.0      NaN
2    Frodo  Hobbit          4.0         NaN      5.0
3  Legolas     Elf          6.0         5.0     10.0
4    Bilbo  Hobbit          4.0         NaN      5.0
```

The structure of the dataframe is maintained, and all text is maintained. Those values not >3 have been removed (NaN represents 'Not a Number').

Conditional replacing of values in Pandas

Replacing values in Pandas, based on the current value, is not as simple as in NumPy. For example, to replace all values in a given column, given a conditional test, we have to (1) take one column at a time, (2) extract the column values into an array, (3) make our replacement, and (4) replace the column values with our adjusted array.

For example to replace all values less than 4 with zero (in our numeric columns):

```
columns = ['magic_power','aggression','stealth']
# note: to get a list of all columns you can use list(df)

for column in columns: # loop through our column list
    values = df[column].values # extract the column values into an array
    mask = values < 4 # create Boolean mask
    values [mask] = 0 # apply Boolean mask
    df[column] = values # replace the dataframe column with the array

print (df)

OUT:

     names    type  magic_power  aggression  stealth
0  Gandolf  Wizard           10           7        8
1    Gimli   Dwarf            0          10        0
2    Frodo  Hobbit            4           0        5
3  Legolas     Elf            6           5       10
4    Bilbo  Hobbit            4           0        5
```

2.13 Summarising data by groups in Pandas using pivot tables and groupby

Pandas offers two methods of summarising data - groupby and pivot_table*. The data produced can be the same but the format of the output may differ.

*pivot_table summarises data. There is a similar command, pivot, which we will use in the next section which is for reshaping data.

As usual let's start by creating a dataframe.

```
import pandas as pd
df = pd.DataFrame()

names = ['Gandolf',
         'Gimli',
         'Frodo',
         'Legolas',
         'Bilbo',
         'Sam',
         'Pippin',
         'Boromir',
         'Aragorn',
         'Galadriel',
         'Meriadoc']
types = ['Wizard',
         'Dwarf',
         'Hobbit',
         'Elf',
         'Hobbit',
         'Hobbit',
         'Hobbit',
         'Man',
         'Man',
         'Elf',
         'Hobbit']
magic = [10, 1, 4, 6, 4, 2, 0, 0, 2, 9, 0]
aggression = [7, 10, 2, 5, 1, 6, 3, 8, 7, 2, 4]
stealth = [8, 2, 5, 10, 5, 4 ,5, 3, 9, 10, 6]


df['names'] = names
df['type'] = types
df['magic_power'] = magic
df['aggression'] = aggression
df['stealth'] = stealth
```

2.13.1 Pivot tables

To return the median values by type:

```
import numpy as np # we will use a numpy method to summarise data

pivot = df.pivot_table(index=['type'],
                       values=['magic_power','aggression', 'stealth'],
                       aggfunc=[np.mean],
                       margins=True) # margins summarises all

# note: addgunc can be any valid function that can act on data provided
```

```
print (pivot)

OUT:

             mean
       aggression magic_power    stealth
type
Dwarf        10.0    1.000000   2.000000
Elf           3.5    7.500000  10.000000
Hobbit        3.2    2.000000   5.000000
Man           7.5    1.000000   6.000000
Wizard        7.0   10.000000   8.000000
All           5.0    3.454545   6.090909
```

Or we may group by more than one index. In this case we'll return the average and summed values by type and magical power:

```
pivot = df.pivot_table(index=['type','magic_power'],
                       values=['aggression', 'stealth'],
                       aggfunc=[np.mean,np.sum],
                       margins=True) # margins summarises all

print (pivot)

OUT:

                        mean                   sum
                  aggression    stealth aggression stealth
type   magic_power
Dwarf  1                10.0   2.000000         10       2
Elf    6                 5.0  10.000000          5      10
       9                 2.0  10.000000          2      10
Hobbit 0                 3.5   5.500000          7      11
       2                 6.0   4.000000          6       4
       4                 1.5   5.000000          3      10
Man    0                 8.0   3.000000          8       3
       2                 7.0   9.000000          7       9
Wizard 10                7.0   8.000000          7       8
All                      5.0   6.090909         55      67
```

2.13.2 Groupby

Grouby is a very powerful method in Pandas which we shall return to in the next section. Here we will use groupby simply to summarise data.

```
print(df.groupby('type').median())

OUT:

        magic_power  aggression  stealth
type
Dwarf           1.0        10.0      2.0
Elf             7.5         3.5     10.0
Hobbit          2.0         3.0      5.0
Man             1.0         7.5      6.0
Wizard         10.0         7.0      8.0
```

Instead of built in methods we can also apply user-defined functions. To illustrate we'll define a simple

function to return the lower quartile.

```
def my_func(x):
    return (x.quantile(0.25))

print(df.groupby('type').apply(my_func))

# Note we need not apply a lambda function
# We may apply any user-defined function
```

```
OUT:

0.25    magic_power  aggression  stealth
type
Dwarf          1.00       10.00      2.0
Elf            6.75        2.75     10.0
Hobbit         0.00        2.00      5.0
Man            0.50        7.25      4.5
Wizard        10.00        7.00      8.0
```

As with pivot-table we can have more than one index column.

```
print(df.groupby(['type','magic_power']).median())
```

```
OUT:

                    aggression  stealth
type   magic_power
Dwarf  1                  10.0      2.0
Elf    6                   5.0     10.0
       9                   2.0     10.0
Hobbit 0                   3.5      5.5
       2                   6.0      4.0
       4                   1.5      5.0
Man    0                   8.0      3.0
       2                   7.0      9.0
Wizard 10                  7.0      8.0
```

Or we can return just selected data columns.

```
print(df.groupby('type').median()[['magic_power','stealth']])
```

```
OUT:

        magic_power  stealth
type
Dwarf           1.0      2.0
Elf             7.5     10.0
Hobbit          2.0      5.0
Man             1.0      6.0
Wizard         10.0      8.0
```

To return multiple types of results we use the *agg* argument.

```
print(df.groupby('type').agg([min, max]))
```

```
OUT:

            names          magic_power     aggression     stealth
              min      max         min max        min max     min max
type
Dwarf       Gimli    Gimli           1   1         10  10       2   2
```

```
Elf     Galadriel  Legolas          6   9          2   5      10  10
Hobbit      Bilbo      Sam          0   4          1   6       4   6
Man       Aragorn  Boromir          0   2          7   8       3   9
Wizard    Gandolf  Gandolf         10  10          7   7       8   8
```

Pandas built-in groupby functions

Remember that apply can be used to apply any user-defined function

.all # Boolean True if all true

.any # Boolean True if any true

.count count of non null values

.size size of group including null values

.max

.min

.mean

.median

.sem

.std

.var

.sum

.prod

.quantile

.agg(functions) # for multiple outputs

.apply(func)

.last # last value

.nth # nth row of group

2.14 Reshaping Pandas data with stack, unstack, pivot and melt

Sometimes data is best shaped where the data is in the form of a wide table where the description is in a column header, and sometimes it is best shaped as as having the data descriptor as a variable within a tall table.

To begin with you may find it a little confusing what happens to the index field as we switch between different formats. But hang in there and you'll get the hang of it!

Lets look at some examples, beginning as usual with creating a dataframe.

```
import pandas as pd
df = pd.DataFrame()

names = ['Gandolf',
         'Gimli',
         'Frodo',
         'Legolas',
         'Bilbo',
         'Sam',
         'Pippin',
         'Boromir',
         'Aragorn',
         'Galadriel',
         'Meriadoc']
types = ['Wizard',
         'Dwarf',
         'Hobbit',
         'Elf',
         'Hobbit',
         'Hobbit',
         'Hobbit',
         'Man',
         'Man',
         'Elf',
         'Hobbit']
magic = [10, 1, 4, 6, 4, 2, 0, 0, 2, 9, 0]
aggression = [7, 10, 2, 5, 1, 6, 3, 8, 7, 2, 4]
stealth = [8, 2, 5, 10, 5, 4 ,5, 3, 9, 10, 6]


df['names'] = names
df['type'] = types
df['magic_power'] = magic
df['aggression'] = aggression
df['stealth'] = stealth
```

When we look at this table, the data descriptors are columns, and the data table is 'wide'.

```
print (df)
```

```
OUT:

        names    type  magic_power  aggression  stealth
0     Gandolf  Wizard           10           7        8
1       Gimli   Dwarf            1          10        2
2       Frodo  Hobbit            4           2        5
3     Legolas     Elf            6           5       10
4       Bilbo  Hobbit            4           1        5
```

```
5         Sam  Hobbit           2           6        4
6      Pippin  Hobbit           0           3        5
7     Boromir     Man           0           8        3
8     Aragorn     Man           2           7        9
9   Galadriel     Elf           9           2       10
10   Meriadoc  Hobbit           0           4        6
```

2.14.1 Stack and unstack

We can convert between the two formats of data with *stack* and *unstack*. To convert from a wide table to a tall and skinny, use *stack*. Notice this creates a more complex index which has two levels the first level is person id, and the second level is the data header. This is called a multi-index.

```
df_stacked = df.stack()
print(df_stacked.head(20)) # pront forst 20 rows

OUT:

0  names          Gandolf
   type            Wizard
   magic_power         10
   aggression           7
   stealth              8
1  names            Gimli
   type             Dwarf
   magic_power          1
   aggression          10
   stealth              2
2  names            Frodo
   type            Hobbit
   magic_power          4
   aggression           2
   stealth              5
3  names          Legolas
   type               Elf
   magic_power          6
   aggression           5
   stealth             10
dtype: object
```

We can convert back to wide table with *unstack*. This recreates a single index for each line of data.

```
df_unstacked = df_stacked.unstack()
print (df_unstacked)

OUT:

        names    type magic_power aggression stealth
0     Gandolf  Wizard          10          7       8
1       Gimli   Dwarf           1         10       2
2       Frodo  Hobbit           4          2       5
3     Legolas     Elf           6          5      10
4       Bilbo  Hobbit           4          1       5
5         Sam  Hobbit           2          6       4
6      Pippin  Hobbit           0          3       5
7     Boromir     Man           0          8       3
8     Aragorn     Man           2          7       9
9   Galadriel     Elf           9          2      10
10   Meriadoc  Hobbit           0          4       6
```

Returning to our stacked data, we can convert our multi-index to two separate fields by resetting the index. By default this method names the separated index field 'level_0' and 'level_1' (multi-level indexes may have further levels as well), and the data field '0'. Let's rename them as well (comment out that row with a # to see what it would look like without renaming them). You can see the effect below:

```
reindexed_stacked_df = df_stacked.reset_index()
reindexed_stacked_df.rename(
    columns={'level_0': 'ID', 'level_1': 'variable', 0:'value'},inplace=True)

print (reindexed_stacked_df.head(20)) # print first 20 rows

OUT:

    ID     variable    value
0    0        names  Gandolf
1    0         type   Wizard
2    0  magic_power       10
3    0   aggression        7
4    0      stealth        8
5    1        names    Gimli
6    1         type    Dwarf
7    1  magic_power        1
8    1   aggression       10
9    1      stealth        2
10   2        names    Frodo
11   2         type   Hobbit
12   2  magic_power        4
13   2   aggression        2
14   2      stealth        5
15   3        names  Legolas
16   3         type      Elf
17   3  magic_power        6
18   3   aggression        5
19   3      stealth       10
```

We can return to a multi-index, if we want to, by setting the index to the two fields to be combined. Whether a multi-index is preferred or not will depend on what you wish to do wit the dataframe, so it useful to know how to convert back and forth between multi-index and single-index.

```
reindexed_stacked_df.set_index(['ID', 'variable'], inplace=True)
print (reindexed_stacked_df.head(20))

OUT:


                 value
ID variable
0  names        Gandolf
   type          Wizard
   magic_power       10
   aggression         7
   stealth            8
1  names          Gimli
   type           Dwarf
   magic_power        1
   aggression        10
   stealth            2
2  names          Frodo
   type          Hobbit
   magic_power        4
```

```
   aggression         2
   stealth            5
3  names        Legolas
   type             Elf
   magic_power        6
   aggression         5
   stealth           10
```

2.14.2 Melt and pivot

melt and *pivot* are like *stack* and *unstack*, but offer some other options.

melt de-pivots data (into a tall skinny table)

pivot will re-pivot data into a wide table.

Let's return to our original dataframe created (which we called 'df') and create a tall skinny table of selected fields using *melt*. We will separate out one or more of the fields, such as 'names' as an ID field, as below:

```
unpivoted = df.melt(id_vars=['names'], value_vars=['type','magic_power'])
print (unpivoted)
```

```
OUT:
```

```
        names     variable   value
0     Gandolf         type  Wizard
1       Gimli         type   Dwarf
2       Frodo         type  Hobbit
3     Legolas         type     Elf
4       Bilbo         type  Hobbit
5         Sam         type  Hobbit
6      Pippin         type  Hobbit
7     Boromir         type     Man
8     Aragorn         type     Man
9   Galadriel         type     Elf
10   Meriadoc         type  Hobbit
11    Gandolf  magic_power      10
12      Gimli  magic_power       1
13      Frodo  magic_power       4
14    Legolas  magic_power       6
15      Bilbo  magic_power       4
16        Sam  magic_power       2
17     Pippin  magic_power       0
18    Boromir  magic_power       0
19    Aragorn  magic_power       2
20  Galadriel  magic_power       9
21   Meriadoc  magic_power       0
```

And we can use the *pivot* method to re-pivot the data, defining which field identifies the data to be grouped together, which column contains the new column headers, and which field contains the data.

```
pivoted = unpivoted.pivot(index='names', columns='variable', values='value')
print (pivoted_2)
```

```
OUT:
```

```
variable  magic_power    type
names
Aragorn             2     Man
```

```
Bilbo              4  Hobbit
Boromir            0     Man
Frodo              4  Hobbit
Galadriel          9     Elf
Gandolf           10  Wizard
Gimli              1   Dwarf
Legolas            6     Elf
Meriadoc           0  Hobbit
Pippin             0  Hobbit
Sam                2  Hobbit
```

2.15 Subgrouping data in Pandas with groupby

A very powerful feature in Pandas is to use groupby to create groups of data. Each group may then be further acted on as if it were an independent dataframe. This allows for very sophisticated operations broken down by group.

Here we will create a very simple example to illustrate this.

Let's create our usual dataframe:

```
import pandas as pd
df = pd.DataFrame()

names = ['Gandolf',
         'Gimli',
         'Frodo',
         'Legolas',
         'Bilbo',
         'Sam',
         'Pippin',
         'Boromir',
         'Aragorn',
         'Galadriel',
         'Meriadoc']
types = ['Wizard',
         'Dwarf',
         'Hobbit',
         'Elf',
         'Hobbit',
         'Hobbit',
         'Hobbit',
         'Man',
         'Man',
         'Elf',
         'Hobbit']
magic = [10, 1, 4, 6, 4, 2, 0, 0, 2, 9, 0]
aggression = [7, 10, 2, 5, 1, 6, 3, 8, 7, 2, 4]
stealth = [8, 2, 5, 10, 5, 4 ,5, 3, 9, 10, 6]

df['names'] = names
df['type'] = types
df['magic_power'] = magic
df['aggression'] = aggression
df['stealth'] = stealth
```

And now we will create an object 'groups' that allows us to work on those groups individually. We will group just by one parameter (type), but the groups can combine more parameters (each group produced will have identical values for the parameters used for the grouping).

We create an 'itterable' object (an object that can be stepped through using groupby. We can step through members of that new groupby object. Each member has an index and a datatframe of data.

Lets just print out the index and the data for each group.

```
groups = df.groupby('type') # creates a new object of groups of data
for index, group_df in groups: # each group has an index and a dataframe of data
    print ('group index:', index)
    print ('\nData')
    print (group_df)
```

```
OUT:
```

```
group index: Dwarf

Data
   names   type  magic_power  aggression  stealth
1  Gimli  Dwarf            1          10        2
group index: Elf

Data
       names type  magic_power  aggression  stealth
3    Legolas  Elf            6           5       10
9  Galadriel  Elf            9           2       10
group index: Hobbit

Data
       names    type  magic_power  aggression  stealth
2      Frodo  Hobbit            4           2        5
4      Bilbo  Hobbit            4           1        5
5        Sam  Hobbit            2           6        4
6     Pippin  Hobbit            0           3        5
10  Meriadoc  Hobbit            0           4        6
group index: Man

Data
     names type  magic_power  aggression  stealth
7  Boromir  Man            0           8        3
8  Aragorn  Man            2           7        9
group index: Wizard

Data
     names    type  magic_power  aggression  stealth
0  Gandolf  Wizard           10           7        8
```

See what happened? We created six smaller dataframes, each of which was one group of the original data. We can then perform any amount of code in each of those sections. As an example, below is some code for a rather crazy method that extracts the third member of the group, but if there are fewer than three members of the group it will take the highest member that it can (the last).

```
col_names = list(df) # get column names from existing dataframe
groups = df.groupby('type') # creates a new object of groups of data
output_df = pd.DataFrame(columns = col_names) # create empty dataframe to store new data

for index, group_df in groups: # each group has an index and a dataframe of data
    number_of_members = len(group_df)
    get_index = min (3, number_of_members)
    get_index -= 1 # subtract 1 to correct for zero-indexing
    retrieved_data = group_df.iloc[get_index].values
    output_df.loc[index] = retrieved_data

print (output_df)

OUT:
            names    type magic_power aggression stealth
Dwarf       Gimli   Dwarf           1         10       2
Elf     Galadriel     Elf           9          2      10
Hobbit        Sam  Hobbit           2          6       4
Man       Aragorn     Man           2          7       9
Wizard    Gandolf  Wizard          10          7       8
```

2.16 Iterating through columns and rows in NumPy and Pandas

Using *apply_along_axis* (NumPy) or *apply* (Pandas) is a more Pythonic way of iterating through data in NumPy and Pandas. But there may be occasions you wish to simply work your way through rows or columns in NumPy and Pandas. Here is how it is done.

2.16.1 NumPy

NumPy is set up to iterate through rows when a loop is declared.

```
import numpy as np

# Create an array of random numbers (3 rows, 5 columns)
array = np.random.randint(0,100,size=(3,5))

print ('Array:')
print (array)
print ('\nAverage of rows:')

# iterate through rows:
for row in array:
    print (row.mean())

OUT:

Array:
[[12 40 30 93 99]
 [62 85 89 26 17]
 [93 34 67 59 56]]

Average of rows:
54.8
55.8
61.8
```

To iterate through columns we transpose the array with .T so that rows become columns (and vice versa):

```
print('\nTranposed array:')
print (array.T)
print ('\nAverage of original columns:')

for row_t in array.T:
    print (row_t.mean())


Transposed array:
[[12 62 93]
 [40 85 34]
 [30 89 67]
 [93 26 59]
 [99 17 56]]

Average of original columns:
55.666666666666664
53.0
62.0
```

```
59.333333333333336
57.333333333333336
```

2.16.2 Pandas

Lets first create our data:

```
import pandas as pd
df = pd.DataFrame()

names = ['Gandolf',
         'Gimli',
         'Frodo',
         'Legolas',
         'Bilbo',
         'Sam',
         'Pippin',
         'Boromir',
         'Aragorn',
         'Galadriel',
         'Meriadoc']
types = ['Wizard',
         'Dwarf',
         'Hobbit',
         'Elf',
         'Hobbit',
         'Hobbit',
         'Hobbit',
         'Man',
         'Man',
         'Elf',
         'Hobbit']
magic = [10, 1, 4, 6, 4, 2, 0, 0, 2, 9, 0]
aggression = [7, 10, 2, 5, 1, 6, 3, 8, 7, 2, 4]
stealth = [8, 2, 5, 10, 5, 4 ,5, 3, 9, 10, 6]

df['names'] = names
df['type'] = types
df['magic_power'] = magic
df['aggression'] = aggression
df['stealth'] = stealth
```

To iterate throw rows in a Pandas dataframe we use .iterrows():

```
for index, row in df.iterrows():
    print(row[0], 'is a', row[1])

OUT:

Gandolf is a Wizard
Gimli is a Dwarf
Frodo is a Hobbit
Legolas is a Elf
Bilbo is a Hobbit
Sam is a Hobbit
Pippin is a Hobbit
Boromir is a Man
Aragorn is a Man
Galadriel is a Elf
```

```
Meriadoc is a Hobbit
```

To iterate through columns we need to do just a bit more manual work, creating a list of dataframe columns and then iterating through that list to pull out the dataframe columns:

```
columns = list(df)
for column in columns:
    print (df[column][2]) # print the third element of the column

OUT:

Frodo
Hobbit
4
2
5
```

2.17 Removing duplicate data in NumPy and Pandas

Both NumPy and Pandas offer easy ways of removing duplicate rows. Pandas offers a more powerful approach if you wish to remove rows that are partly duplicated.

2.17.1 NumPy

With numpy we use *np.unique()* to remove duplicate rows or columns (use the argument *axis=0* for unique rows or *axis=1* for unique columns).

```
import numpy as np

array = np.array([[1,2,3,4],
                  [1,2,3,4],
                  [5,6,7,8],
                  [1,2,3,4],
                  [3,3,3,3],
                  [5,6,7,8]])

unique = np.unique(array, axis=0)
print (unique)

OUT:

[[1 2 3 4]
 [3 3 3 3]
 [5 6 7 8]]
```

We can return the index values of the kept rows with the argument *return_index=True* (the argument *return_inverse=True* would return the discarded rows):

```
array = np.array([[1,2,3,4],
                  [1,2,3,4],
                  [5,6,7,8],
                  [1,2,3,4],
                  [3,3,3,3],
                  [5,6,7,8]])

unique, index = np.unique(array, axis=0, return_index=True)
print ('Unique rows:')
print (unique)
print ('\nIndex of kept rows:')
print (index)

OUT:

Unique rows:
[[1 2 3 4]
 [3 3 3 3]
 [5 6 7 8]]

Index of kept rows:
[0 4 2]
```

We can also count the number of times a row is repeated with the argument *return_counts=True*:

```
array = np.array([[1,2,3,4],
                  [1,2,3,4],
                  [5,6,7,8],
                  [1,2,3,4],
```

```
                  [3,3,3,3],
                  [5,6,7,8]])

unique, index, count = np.unique(array, axis=0,
                           return_index=True,
                           return_counts=True)

print ('Unique rows:')
print (unique)
print ('\nIndex of kept rows:')
print (index)
print ('\nCount of duplicate rows')
print (count)

OUT:

Unique rows:
[[1 2 3 4]
 [3 3 3 3]
 [5 6 7 8]]

Index of kept rows:
[0 4 2]

Count of duplicate rows
[3 1 2]
```

2.17.2 Pandas

With Pandas we use *drop_duplicates*.

```
import pandas as pd
df = pd.DataFrame()

names = ['Gandolf','Gimli','Frodo', 'Gimli', 'Gimli']
types = ['Wizard','Dwarf','Hobbit', 'Dwarf', 'Dwarf']
magic = [10, 1, 4, 1, 3]
aggression = [7, 10, 2, 10, 2]
stealth = [8, 2, 5, 2, 5]


df['names'] = names
df['type'] = types
df['magic_power'] = magic
df['aggression'] = aggression
df['stealth'] = stealth
```

Let's remove duplicated rows:

```
df_copy = df.copy() # we'll work on a copy of the dataframe
df_copy.drop_duplicates(inplace=True)
print (df_copy)

OUT:

df_copy = df.copy() # we'll work on a copy of the dataframe

df_copy.drop_duplicates(subset=['names','type'], inplace=True)
```

```
print (df_copy)
```

```
     names    type  magic_power  aggression  stealth
0  Gandolf  Wizard           10           7        8
1    Gimli   Dwarf            1          10        2
2    Frodo  Hobbit            4           2        5
```

We can choose to keep the last entered row with the argument *keep='last'*:

```
df_copy = df.copy() # we'll work on a copy of the dataframe
df_copy.drop_duplicates(subset=['names','type'], inplace=True, keep='last')
print (df_copy)
```

```
OUT:
```

```
     names    type  magic_power  aggression  stealth
0  Gandolf  Wizard           10           7        8
2    Frodo  Hobbit            4           2        5
4    Gimli   Dwarf            3           2        5
```

We can also remove all duplicate rows by using the argument *keep=False*:

```
df_copy = df.copy() # we'll work on a copy of the dataframe
df_copy.drop_duplicates(subset=['names','type'], inplace=True, keep=False)
print (df_copy)
```

```
OUT:
```

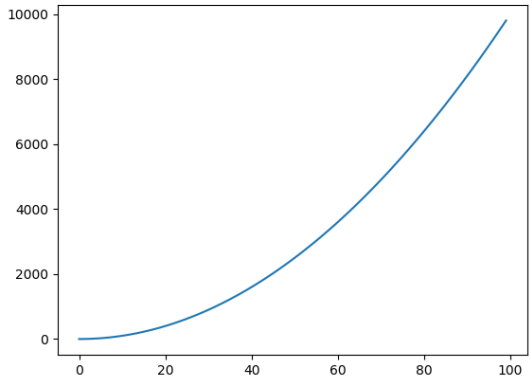
```
     names    type  magic_power  aggression  stealth
0  Gandolf  Wizard           10           7        8
2    Frodo  Hobbit            4           2        5
```

More complicated logic for choosing which record to keep would best be performed using a groupby method.

Chapter 3

Matplotlib for charting

Figure 3.1: A simple xy line chart



3.1 Simple xy line charts, and simple save to file

Matplotlib is a powerful library for plotting data. Data may be in the form of lists, or data from NumPy and Pandas may be used directly. Charts are very highly configurable, but here we'll focus on the key basics first.

A useful resource for matplotlib is a gallery of charts with associated code: `https://matplotlib.org/gallery.html`

3.1.1 Plotting a single line

The following code plots a simple line plot and saves the file in png format. Other formats may be used (e.g. jpg), but png offers an open source high quality format which is ideal for charts.

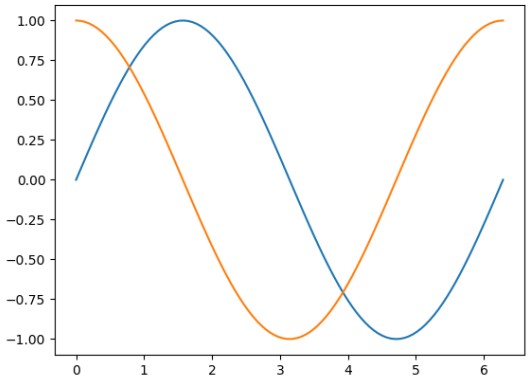
```
import matplotlib.pyplot as plt

# The following line is only needed to display chart in Jupyter notebooks
%matplotlib inline

X=range(100)
Y=[value ** 2 for value in X] # (A list comprehension)

plt.plot(X,Y)
plt.show()
plt.savefig('plot_01.png') # optional save as png file
```

3.1.2 Plotting two lines

To plot two lines we simply add another plot before generating the chart with plt.show()

```
import numpy as np

import matplotlib.pyplot as plt

# The following line is only needed to display chart in Jupyter notebooks
%matplotlib inline

# np.linsapce creates an array of equally spaced numbers
X=np.linspace(0,2*np.pi,100) # 100 points between 0 and 2*pi
```

Figure 3.2: A simple xy line chart with two lines

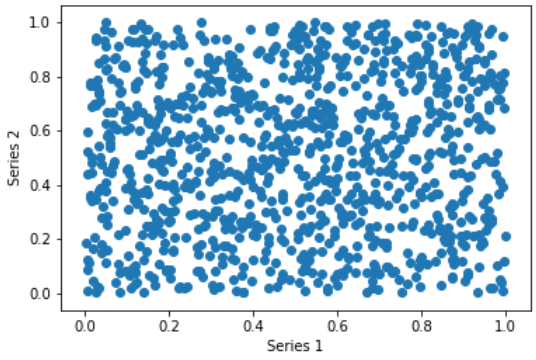



```
Ya=np.sin(X)
Yb=np.cos(X)

plt.plot(X,Ya)
plt.plot(X,Yb)
plt.show()
```

3.1.3 Saving figures

To save a figure use *plt.savefig('my_figname.png')* before *plt.show()* to save as png format (best for figures, but you can also use jpg or tif).

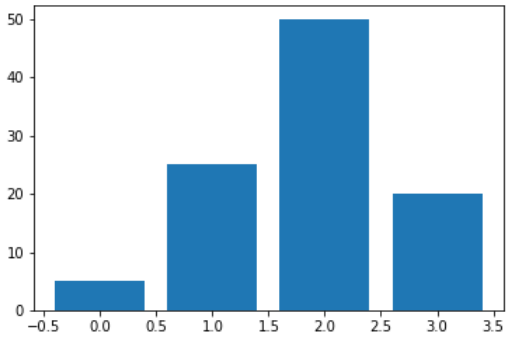
Figure 3.3: A scatter plot



3.2 Scatter plot, and labelling axes

```
import matplotlib.pyplot as plt
import numpy as np

# Create a 1024 by 2 array of random points
data=np.random.rand(1024,2)

x = data[:,0]
y = data[:,1]
plt.scatter(x, y)

plt.xlabel ('Series 1')
plt.ylabel ('Series 2')

plt.show()
```

Figure 3.4: A simple bar chart



3.3 Bar charts

3.3.1 A simple bar chart

```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.pyplot as plt
import numpy as np

data =[5.,25.,50.,20.]
x = range(len(data)) # creates a range of 0 to 4

plt.bar(x, data)

plt.show()
```

3.3.2 Multiple series

```
import matplotlib.pyplot as plt
import numpy as np

data =[[5.,25.,50.,20],
       [4.,23.,51.,17],
       [6., 22.,52.,19]]

X=np.arange(4)

plt.bar(X+0.00,data[0],color='b',width=0.25) # colour is blue
plt.bar(X+0.25,data[1],color='g',width=0.25) # colour is green
plt.bar(X+0.50,data[2],color='r',width=0.25) # colour is red

plt.show()
```

For larger data sets we could automate the creation of the bars:

```
import matplotlib.pyplot as plt
import numpy as np

data =[[5.,25.,50.,20],
[4.,23.,51.,17],
  [6., 22.,52.,19]]
```

Figure 3.5: A multiple series bar chart

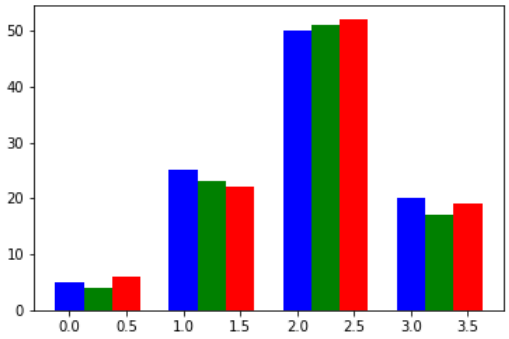
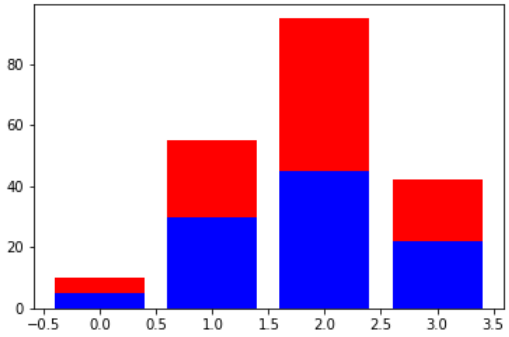


Figure 3.6: A stacked bar chart




```
color_list=['b','g','r']
gap=0.8/len(data)

for i, row in enumerate(data): # creates int i and list row
X=np.arange(len(row))
plt.bar(X+i*gap,row,width=gap,color=color_list[i])
plt.show()
```

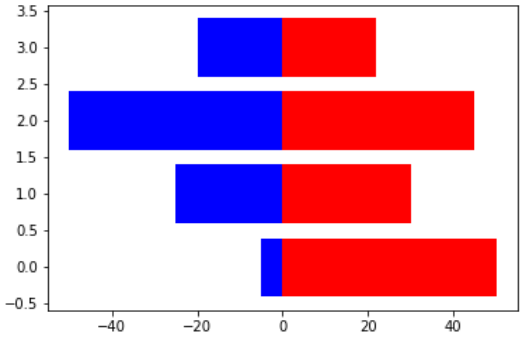
3.3.3 Stacked bar chart

```
a=[5.,30.,45.,22.]
b=[5.,25.,50.,20.]
x=range(4)

plt.bar(x,a,color='b')
plt.bar(x,b,color='r',bottom=a) # bottom specifies the starting position for a bar

plt.show()
```

3.3.4 Back to back bar chart

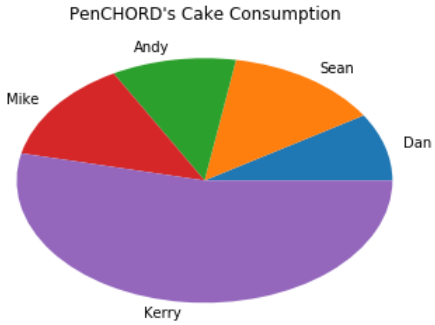
```
women_pop=np.array([50,30,45,22])
men_pop=np.array([5,25,50,20])
x=np.arange(4)
```

Figure 3.7: A back to back bar chart




```
plt.barh(x,women_pop,color='r')
plt.barh(x,-men_pop,color='b') # the minus sign creates an opposite bar

plt.show()
```

Figure 3.8: A pie chart



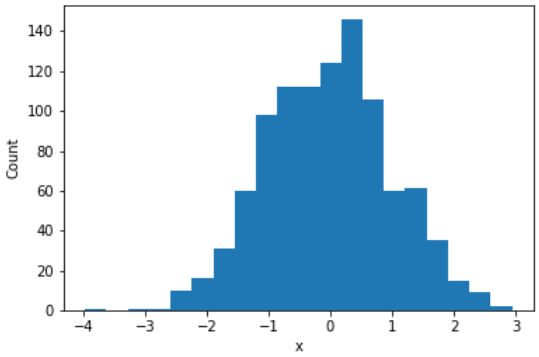
3.4 Pie charts, and adding a title

3.4.1 A simple bar chart

As with all matplotlib charts, pie charts are highly configurable. Here we just show the simplest of pie charts.

```
import matplotlib.pyplot as plt

labels = 'Dan','Sean','Andy','Mike','Kerry'

cake_consumption = [10, 15, 12, 30, 100]

plt.pie(cake_consumption, labels=labels)
plt.title ("PenCHORD's Cake Consumption")

plt.show()
```

Figure 3.9: A histogram with defined number of bins



3.5 Plotting histograms with matplotlib and NumPy

Matplotlib has an easy method for plotting data. NumPy has an easy method for obtaining histogram data.

3.5.1 Plotting histograms with Matplotlib

Plotting a histogram with a defined number of bins:

```
import matplotlib.pyplot as plt
import numpy as np

%matplotlib inline

x=np.random.randn(1000) # samples from a normal distribution

plt.hist(x,bins=20)
plt.title ('Defined number of bins')
plt.xlabel ('x')
plt.ylabel ('Count')

plt.show()
```

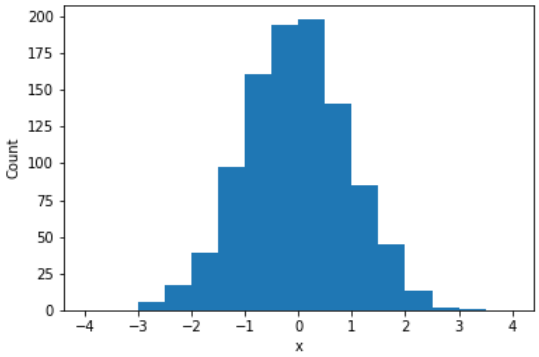
Plotting a histogram with a defined range of bins:

```
x=np.random.randn(1000) # samples from a normal distribution

# Use np.arange to create bins from -4 to +4 in steps of 0.5
# A custom list could also be used

plt.hist(x,bins=np.arange(-4,4.5,0.5))
plt.title ('Defined bin range and width')
plt.xlabel ('x')
plt.ylabel ('Count')

plt.show()
```

3.5.2 Obtaining histogram data with NumPy

If histogram data is needed in addition to, or instead of, a plot, NumPy may be used. Here a defined number of bins is used:

Figure 3.10: A histogram with defined bin range and width




```
import numpy as np
count, bins = np.histogram(x, bins=20)
print ('Bins:')
print (bins)
print ('\nCount:')
print (count)

OUT:

Bins:
[-2.88652288 -2.57566476 -2.26480664 -1.95394852 -1.6430904  -1.33223228
 -1.02137416 -0.71051604 -0.39965792 -0.0887998   0.22205832  0.53291644
  0.84377456  1.15463268  1.4654908   1.77634892  2.08720704  2.39806516
  2.70892328  3.0197814   3.33063952]

Count:
[  5   9  12  22  32  69  80 113 125 116 108  94  77  59  40  22  10   4
   1   2]
```

And here a defined bin range is used.

```
import numpy as np
count, bins = np.histogram(x, bins=np.arange(-5,6,1))
print ('Bins:')
print (bins)
print ('\nCount:')
print (count)

OUT:

Bins:
[-5 -4 -3 -2 -1  0  1  2  3  4  5]

Count:
[  0   0  20 134 332 340 154  18   2   0]
```

Figure 3.11: A simple boxplot

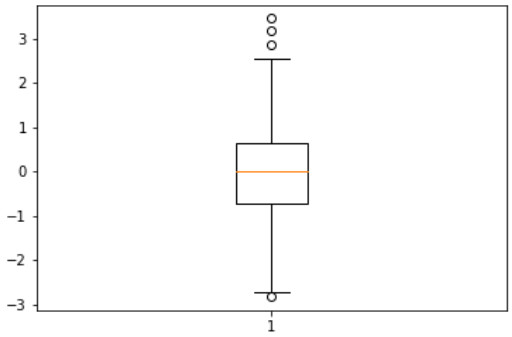
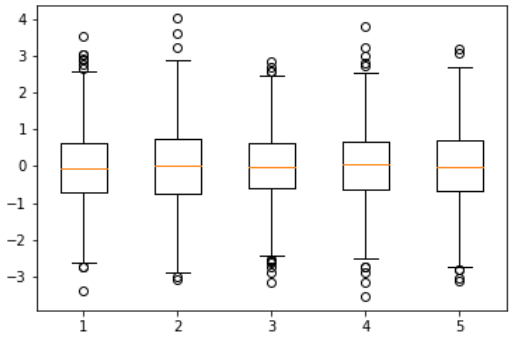


Figure 3.12: A grouped boxplot from a single NumPy array



3.6 Boxplots

Matplotlib allows easy creation of boxplots. These traditionally show median (middle line across box), uper and lower quartiles (box), range excluding outliers (whiskers) and outliers (points). The default setting for outliers is points more than 1.5xIQR above or below the quartiles.

*IQR = inter-quartile range.

```
import matplotlib.pyplot as plt
import numpy as np

x=np.random.randn(500) # samples from a normal distribution

plt.boxplot(x)

plt.show()
```

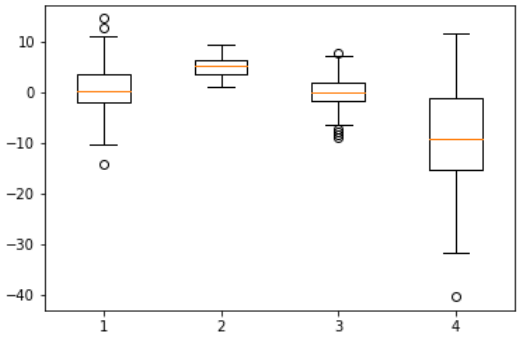
3.6.1 Plotting groups

Boxplot can take data from multiple columns in a NumPy array.

```
x=np.random.randn(1000,5) # samples from a normal distribution

plt.boxplot(x)

plt.show()
```

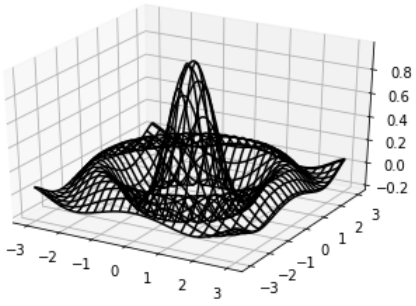
Or data may come from separate sources:

Figure 3.13: A grouped boxplot from multiple sources




```
x1=list(np.random.randn(100)*5)
x2=list((np.random.randn(50)*2)+5)
x3=list(np.random.randn(250)*3)
x4=list((np.random.randn(70)*10)-10)

x=[x1,x2,x3,x4]

plt.boxplot(x)

plt.show()
```

Figure 3.14: A 3D wireframe



3.7 3D wireframe and surface plots

We can create 3D wireframe or surface plots easily in MatPlotLib

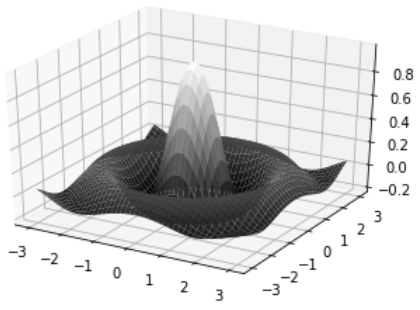
3.7.1 Wireframe

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm

# Create a 3D array
# meshgrid produces all combinations of given x and y
x=np.linspace(-3,3,256) # x goes from -3 to 3, with 256 steps
y=np.linspace(-3,3,256) # y goes from -3 to 3, with 256 steps
X,Y=np.meshgrid(x,y) # combine all x with all y

# A function of x and y for demo purposes
Z=np.sinc(np.sqrt(X**2 + Y**2))

fig=plt.figure()
ax=fig.gca(projection='3d')

# rstride: Array row stride (step size), defaults to 1
# cstride: Array column stride (step size), defaults to 1
# rcount: Use at most this many rows, defaults to 50
# ccount: Use at most this many columns, defaults to 50

ax.plot_wireframe(X,Y,Z,color='k',rcount=25,ccount=25)

plt.show()
```

3.7.2 Surface

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm

# Create a 3D array
```

Figure 3.15: A 3D surface plot




```
# meshgrid produces all combinations of given x and y
x=np.linspace(-3,3,256) # x goes from -3 to 3, with 256 steps
y=np.linspace(-3,3,256) # y goes from -3 to 3, with 256 steps
X,Y=np.meshgrid(x,y) # combine all x with all y

# A function of x and y for demo purposes
Z=np.sinc(np.sqrt(X**2 + Y**2))

fig=plt.figure()
ax=fig.gca(projection='3d')
ax.plot_surface(X,Y,Z,cmap=cm.gray)

plt.show()}
```

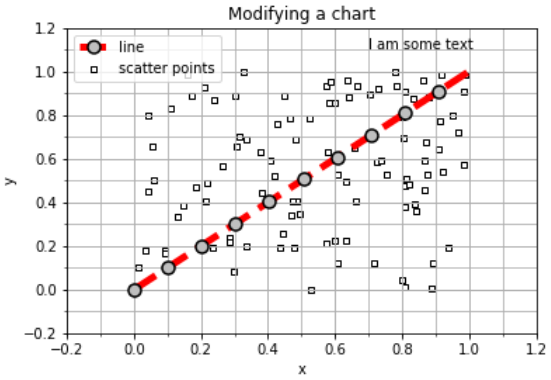
3.8 Common modifications to charts

Here we show some common modifications to charts. These include:

- Changing scatter plot point style
- Changing line plot line and marker style
- Adding a legend
- Adding some text
- Changing axis scales
- Changing axis ticks
- Adding a grid
- Adding axis tiles
- Adding chart title

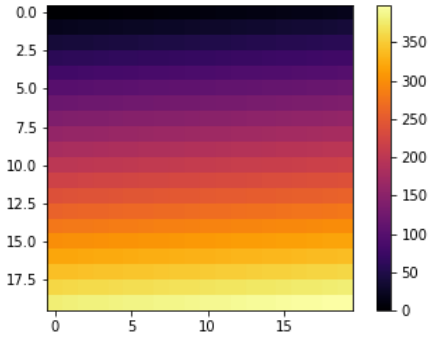
```
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.ticker as ticker

data=np.random.rand(100,2)

# To give us maximum control over axes we set up the figure in this way:

ax1 = plt.figure().add_subplot(111)

# Add scatter plot
# Adjust scatter plot points by shape (marker), size (s),
# color, and edgecolour. Add a label for legend.

ax1.scatter(data[:,0],data[:,1],
            marker='s',
            s = 20,
            color = 'w',
            edgecolors = 'k',
            label  = 'scatter points')

x = np.linspace(0,1,100)
y = np.linspace(0,1,100)

# Add line plot
# Adjust line for colour, style, and width, and marker
# shape, frequency, and coloring. Colours may be given by letter
# or by a number between '0.0' (black) and '1.0' (white)
# Add a label for legend.

ax1.plot(x,y,
        color = 'r',
        linestyle = '--',
        linewidth = 5,
        marker = 'o',
        markevery = 10,
        markersize=9,
        markeredgewidth=1.5,
        markerfacecolor='0.75',
        markeredgecolor='k',
        label  = 'line')
```

Figure 3.16: A modified chart




```
# Adjust axes limits

ax1.set_xlim(-0.2,1.2)
ax1.set_ylim(-0.2,1.2)

# Adjust axes tickmarks

ax1.xaxis.set_major_locator(ticker.MultipleLocator(0.2))
ax1.xaxis.set_minor_locator(ticker.MultipleLocator(0.1))
ax1.yaxis.set_major_locator(ticker.MultipleLocator(0.2))
ax1.yaxis.set_minor_locator(ticker.MultipleLocator(0.1))

# Add a grid

ax1.grid(True, which='both') # which may be major, minor or both

# Add axis titles

ax1.set_xlabel('x', size = 15)
ax1.set_ylabel('y', size = 15)
# Add a title

ax1.set_title ('Modifying a chart', size = 20)

# Add some text at a given position

plt.text(0.7,1.1,'I am some text')

# Add the legend

ax1.legend() # see help (plt.legend) for options

plt.show()
```

Figure 3.17: A simple heat map



3.9 A simple heatmap

Heatmaps may be generated with *imshow*.

We import a colour map from the library *cm*.

For a list of colour maps available see: `https://matplotlib.org/examples/color/colormaps_reference.html`

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm # This allows different color schemes

# Generate an array of increasing values
a=np.arange(0,400)
a = a.reshape(20,20)

# Plot the heatmap using 'inferno' from the cm colour schemes

plt.imshow(a,interpolation='nearest', cmap=cm.inferno)

# Add a scale bar

plt.colorbar()

plt.show()
```

3.10 Creating a grid of subplots

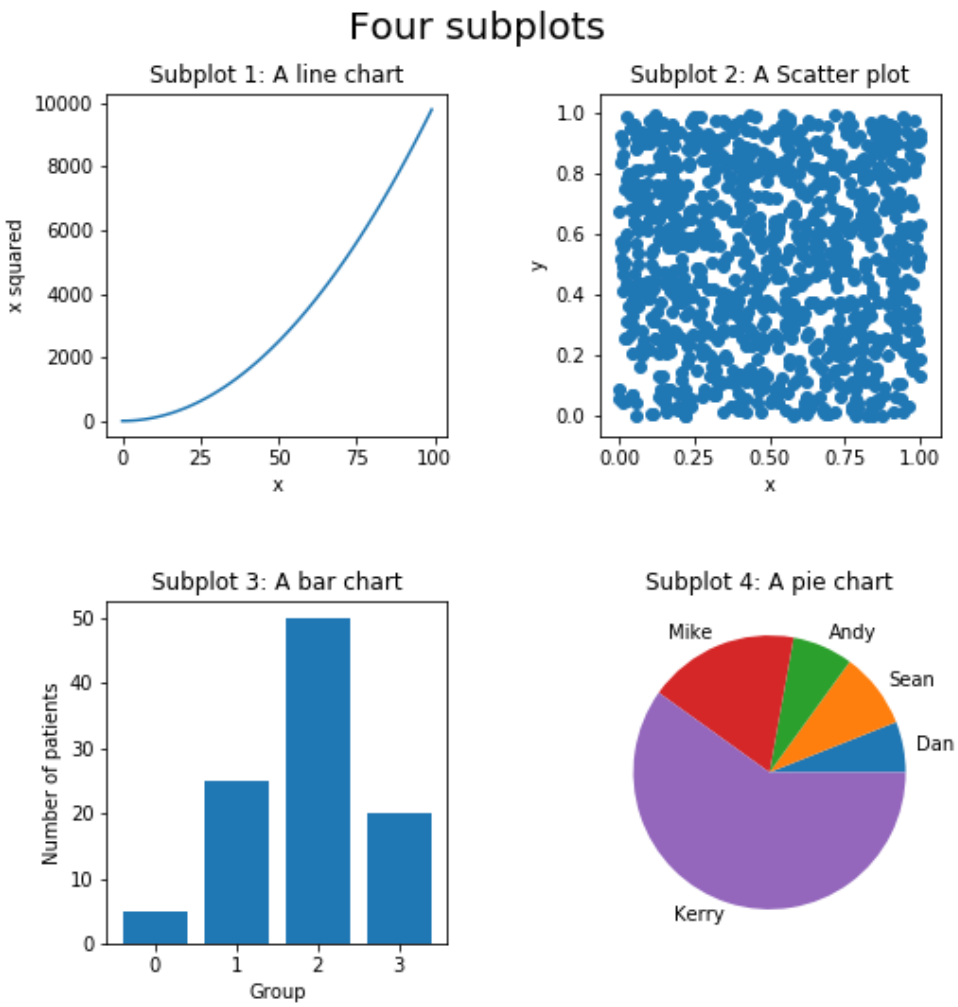
There are various ways of creating subplots in Matplotlib.

Here we will use add_subplot to bring four plots together.

It is also worth looking at *subplot2grid* if you want plots of different sizes bought together.

```
import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline

# Define the size of the overall figure
fig = plt.figure(figsize=(8,8)) # 8 inch * 8 inch

# Create subplot 1

ax1 = fig.add_subplot(221) # Grid of 2x2, this is suplot 1

x=range(100)
y=[value ** 2 for value in x]

ax1.plot(x,y)

ax1.set_xlabel('x')
ax1.set_ylabel('x squared')
ax1.set_title('Subplot 1: A line chart')

# Create subplot 2

ax2 = fig.add_subplot(222) # Grid of 2x2, this is suplot 2

data=np.random.rand(1024,2)

ax2.scatter(data[:,0],data[:,1])

ax2.set_xlabel('x')
ax2.set_ylabel('y')
ax2.set_title('Subplot 2: A Scatter plot')

# Create subplot 3

ax3 = fig.add_subplot(223) # Grid of 2x2, this is suplot 3

data=[5.,25.,50.,20.]

ax3.bar(range(len(data)),data)

ax3.set_xlabel('Group')
ax3.set_ylabel('Number of patients')
ax3.set_title('Subplot 3: A bar chart')

# Create subplot 4

ax4 = fig.add_subplot(224) # Grid of 2x2, this is suplot 4

labels = 'Dan','Sean','Andy','Mike','Kerry'
cake_consumption = [10, 15, 12, 30, 100]
```

Figure 3.18: Creating a grid of subplots

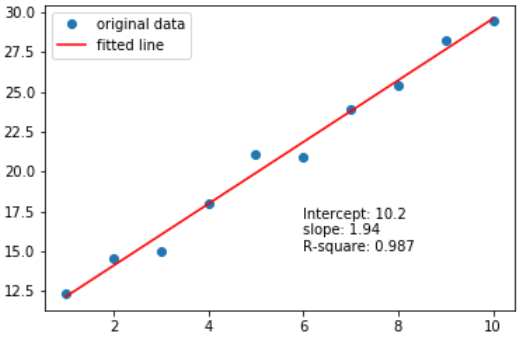



```
ax4.pie(cake_consumption, labels=labels)

ax4.set_title ("Subplot 4: A pie chart")

# Add an overall title

plt.suptitle('Four subplots', size = 20)

# Adjust the spacing between plots

plt.tight_layout(pad=4)

# Show plot

plt.show()
```

Chapter 4

Statistics

Figure 4.1: Linear regression with scipy.stats



4.1 Linear regression with scipy.stats

```
%matplotlib inline

import numpy as np
import matplotlib.pyplot as plt
from scipy import stats

# Set up x any arrays

x=np.array([1,2,3,4,5,6,7,8,9,10])
y=np.array([2.3,4.5,5.0,8,11.1,10.9,13.9,15.4,18.2,19.5])
y=y+10

# scipy linear regression

gradient, intercept, r_value, p_value, std_err = stats.linregress(x,y)

# Calculated fitted y

y_fit=intercept + (x*gradient)

# Plot data

plt.plot(x, y, 'o', label='original data')
plt.plot(x, y_fit, 'r', label='fitted line')

# Add text box and legend

text='Intercept: %.1f\nslope: %.2f\nR-square: %.3f' %(intercept,gradient,r_value**2)
plt.text(6,15,text)
plt.legend()

# Display plot

plt.show()
```