Python for Healthcare Modelling

Michael Allen

March 22, 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