Python for Healthcare Modelling

Michael Allen

March 13, 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:] # The 4: is a slice from index 4 to end
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')
```