Introduction to Python

October 12, 2018

1 Introduction to Python

These exercises accompany the introduction to python session. They can be done using the interactive prompt or in a script file. It is often useful to play around with code from each topic until you fully understand it, in addition to the specific exercises given below.

1.1 Basic Types and Operations

Python supports a number of basic types of object - numbers (integer and floating point), strings, lists etc. The following Exercises will introduce you to basic manipulation of these.

1.1.1 Numerics and Calculations

Evaluate the following:

1. $1 + 2$
2. $4 \times (1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9})$
3. The area of a circle of radius 4.32 (hint: π is available in the `math` module as `math.pi`)
4. e^2
5. 12 mod 4

1.1.2 Boolean objects and Logical Operations

What would the following statements produce? Were you Right?

```
In [ ]: True and False

In [ ]: False or True

In [ ]: True and False or True

In [ ]: (True and False) or True

In [ ]: True or embl

In [ ]: True and embl
```

```
In [ ]: 1 or 0

In [ ]: 1 and 2

In [ ]: [] or ''

In [ ]: [] or 'embl'
```

As well as `and` and `or` there are various comparasion operators, which generally behave as you would expect them to based on maths. One thing to bare in mind is that the logical equality operator is `==` since a single `=` is already used to assign variables. In addition there are bitwise operators that work directly on the binary representation of object, but these are a bit advanced for the introduction session (see more here if you're curious). What do you expect the following statements will evaluate to:

```
In [ ]: 1 > 0

In [ ]: 2 == 2

In [ ]: 3 <= 2

In [ ]: type(1) is int
```

1.1.3 Strings

Start by storing a reasonably long string of your choice in a variable:

```
In [ ]: my_string = 'The Quick Brown Fox Jumps Over the Lazy Dog.'
```

Tasks: 1. Determine the length of the string 1. Extract characters 5 to 10 (inclusive) 1. Make the string all upper or lower case 1. Concatanate another string to it 1. Split the string up into words and store it in a list

1.1.4 Lists

Start by using your list of words. 1. Extract the 3rd to 5th words 1. Remove the last word and store it in a variable 1. Add another word to the end of the list 1. Add the stored previous last word to the start of the list

Now create a list of numbers 1. What is their sum? 1. What is the mean? 1. The mode? 1. The median?

1.1.5 Dictionaries

Copy the dictionary below:

```
In [ ]: ages = {'Alice':21, 'Bob':19, 'Charlie':25, 'Dave':23, 'Emma':29}
```

1. What is the average age of Alice, Charlie and Emma?
2. What is the sum of all ages?
3. How much older is Bob than Dave?

1.2 Control Flow

While simple manipulation of variables is useful it can't solve very many problems easily or efficiently on its own. We will now cover tools that allow you to produce useful programs.

1.2.1 While loops

While loops allow you to repeat a block of code until a condition ceases to be true, checking at the begining of each cycle. For example the following loop prints random numbers until it gets one greater than 0.75:

```
In [7]: import random
        random.seed()

        x = 0
        while x < 0.75:
            x = random.random()
            print(x)
```

```
0.1551412937816793
0.042449653133692755
0.6242560874745714
0.19823924558099004
0.9447063413358199
```

1. Write a loop to draw random values from a list until it gets a specific value
2. Write a while loop that creates a new list containing the squares of values in `[1,3,5,3,7,8,10]`

1.2.2 For loops

The second exercise was possible with a while loop, but would have been much more convenient using a for loop. These iterate through each object in a list, or other object containing multiple values.

For example the second exercise above could be written:

```
In [6]: x = [1,3,5,3,7,8,10]
        x_squared = []

        for i in x:
            x_squared.append(i)
        x_squared
```

```
Out[6]: [1, 3, 5, 3, 7, 8, 10]
```

A commonly used looping object is the `range()` function, which creates a series of numbers to loop over:

```
In [4]: for i in range(3, 11, 2):
            print(i)
```

```
3
5
7
9
```

Notice it doesn't include the finishing value. It is often used with a single argument, which creates a series up to but not including that value:

`range(10) -> 0, 1, 2, ..., 9`

Exercises: 1. Generate a list with the cubes of odd numbers from 5 to 25 1. Print the length of each word in the list generated in the strings section 1. For each key, value pair in your age dictionary, print the persons name and birth year (Hint: dict.items() gives a list of tuples containing key, value pairs that can be assigned like so `k, v in dict.items()`)

1.2.3 If statements

It is also common to need to do different things with different inputs or make other branching decisions in your programs. Conditional statements do this for you, using the same condition expressions we previously used in while loops. For example we can print even numbers from a list:

```
In [3]: x = [1,2,3,5,6,5,45,7,3,48,5,4,8,6,23,232,74]
        for i in x:
            if i % 2 == 0:
                print(i)

2
6
48
4
8
6
232
74
```

You are also able to include `elif` blocks, to sequentially test different conditions and a final `else` block, which is run if none of the conditions are true:

```
In [9]: for i in [1,'str', [1,2,3], 'str', 12.3, 'str', {'dict':1}]:
            if type(i) is str:
                print('String')
            elif type(i) is list:
                print('List')
            elif type(i) in (int, float):
                print('Number')
            else:
                print('Something else')
```

```
Number
String
List
String
Number
String
Something else
```

Exercises: 1. print words containing the letter 'e' from your sentence list 1. Print all integers < 100 but replace multiples of 3 with 'fizz', multiples of 5 with 'buzz' and multiples of both with 'fizzbuzz'.

1.3 Functions

Often in a program you will want to do the same opperation many times and writing the same code over again wastes time. Functions allow you to easily reuse code while also allowing you to make it more general and apply it to many similar situations. For example suppose we wanted to calculate the kinetic energy of many objects:

```
In [14]: def kinetic_energy(velocity, mass=1):
             ke = 1/2 * mass * velocity**2
             return ke

         # print the ke for a series of objects in J/kg
         velocities = [2,4,3,6,5,7]
         for v in velocities:
             print('An object traveling at {}m/s has a kinetic energy of {}J/kg'.format(v, kine

         print('A 5kg object traveling at 3m/s has a kinetic energy of {}J'.format(kinetic_ener
```

```
An object traveling at 2m/s has a kinetic energy of 2.0J/kg
An object traveling at 4m/s has a kinetic energy of 8.0J/kg
An object traveling at 3m/s has a kinetic energy of 4.5J/kg
An object traveling at 6m/s has a kinetic energy of 18.0J/kg
An object traveling at 5m/s has a kinetic energy of 12.5J/kg
An object traveling at 7m/s has a kinetic energy of 24.5J/kg
A 5kg object traveling at 3m/s has a kinetic energy of 22.5J
```

The critical parts of the function are the arguments, defined in brackets after the function name (`name(arg1, arg2)`) and the return statement, which when run will exit the function and pass the value to the code that called the function. The function also demonstrates the ability to set defaults for arguments (`name(arg=default)`).

Exercises: Write functions that 1. Calculate $e = mc^2$ 1. Returns Hardy-Weinberg equilibria as a tuple 1. Sorts the words in a sentence, with an option for ascending or descending sort 1. **Advanced**: Implements a bubble sort

Advanced: How does the following function work? (Don't call it with a large input!)

```
In [15]: def fac(x):
             x = int(x)
             if x > 0:
                 return x * fac(x-1)
             else:
                 return 1
```

1.3.1 Common Errors when passing mutable objects

There are two errors python begginers make very commonly in relation to using mutable objects (ones that can be edited in place, meaning `list`, `dict`, etc.). One relates to functions the second does not. Firstly, when you set a functions argument default as a mutable object it is maintained whenever you use the function, meaning your function will likely not work as intended the second time you use it:

```
In [3]: def add_to_list(itm, lst=[]):
            lst.append(itm)
            return lst

        # Specifying a list works as intended
        print(add_to_list(4, [1,2,3]))

        # The first use of the defualt empty list works as expected
        print(add_to_list(1))

        # Subsequent calls do not!
        print(add_to_list(2))
```

```
[1, 2, 3, 4]
[1]
[1, 2]
```

The proper way to get an empty list (or similar) defualt argument is to use `None` as the default. `None` is a special python value that indicates a variable is empty.

```
In [4]: def add_to_list(itm, lst=None):
            if lst is None:
                lst = []

            lst.append(itm)
            return lst

        # The first use of the defualt empty list works as expected
        print(add_to_list(1))

        # Now subsequent calls work too!
        print(add_to_list(2))
```

```
[1]
[2]
```

The second common error occurs when reusing mutable objects carelessly. For example, imagine you were representing a matrix as a list of lists, and wanted a 0 matrix:

```
In [8]: # Function to pretty print a matrix list
        def prnt_mat(lst):
            for i in lst:
                print(i)

        a = [[0,0,0]] * 3
        print('Original Matrix:')
        prnt_mat(a)

        a[1][1] = 1
        print('Modified Matrix:')
        prnt_mat(a)
```

```
Original Matrix:
[0, 0, 0]
[0, 0, 0]
[0, 0, 0]
Modified Matrix:
[0, 1, 0]
[0, 1, 0]
[0, 1, 0]
```

When you change one of the sublists they all changed! This is because when we constructed the list we told python to use the `[0,0,0]` list three times and so a ended up containing 3 references to the same object. Usually this is not what you want, so you must be careful and make sure python is adding a fresh list each time:

```
In [10]: a = []
         for i in range(3):
             a.append([0] *3)

         print('Original Matrix:')
         prnt_mat(a)

         a[1][1] = 1
         print('Modified Matrix:')
         prnt_mat(a)
```

```
Original Matrix:
[0, 0, 0]
[0, 0, 0]
```

```
[0, 0, 0]
Modified Matrix:
[0, 0, 0]
[0, 1, 0]
[0, 0, 0]
```

We can now modify sublists separately. Note that this method still uses the `[x] * n` syntax, but to create the sublists with multiple integers, which are imutable and can not be modified in place.

1.4 Input/Output

In many cases you want your programs to be able to interact with files, for example to read in data or write your results. This is fairly straightforward in base Python and the fileinput standard module is available if you need more power. In base python the `open` command is used to open a file object in various modes (e.g. `open(file, 'rw)` to open in read and write mode). You can then iterate through the lines in the file using a for loop or read all lines at once using the `readlines` method. Generally the former method is more efficient, unless you need information from multiple lines at once. For example to read in a short Fasta file:

```
In [26]: fasta_file = open('example.fa', 'r+')
         seqs = {}

         for line in fasta_file:
             # Check if seqs is empty and you've encountered a header: initialise a new seq wi
             if not seqs and line[0] == '>':
                 name = line.strip('>\t\n ')
                 seqs[name] = []

             # Check if line is subsequent header: finish writing the old seq and initialise t
             elif line[0] == '>':
                 seqs[name] = ''.join(seqs[name])

                 name = line.strip('>\t\n ')
                 seqs[name] = []

             # Else append the sequence to the lise
             else:
                 seqs[name].append(line.strip())

         # write the final sequence
         seqs[name] = ''.join(seqs[name])

         print(seqs)
```

```
{'Sequence1': 'ACTGACTAGCTACGAGCGCTTTCGATCAGGCATCGGCGTATCAGCTACG', 'Sequence2': 'CCACTGACTTAGCA
```

It is usually safer to encase this sort of code in a `with` block, which makes sure the file is opened and closed correctly after you finish with it. The `with` block runs the entrance and exit code associated with the object you pass to it as it starts and finishes, in our case opening and closing the file. For example:

```
In [29]: with open('example.fa', 'r+') as fasta_file:
             for line in fasta_file:
                 print('The line is {} characters long'.format(len(line.strip())))

The line is 11 characters long
The line is 49 characters long
The line is 0 characters long
The line is 11 characters long
The line is 48 characters long
The line is 40 characters long
The line is 0 characters long
The line is 11 characters long
The line is 48 characters long
```

You can also print to file objects or use the `write` method, which writes a string directly to the file without any formatting (e.g. the `\n` appended by print). This requires the file to be in `w` (write) or `a` (append) mode, the former clearing the file and the latter letting you write to the end. Examine the file generated to see the effects of the statements.

```
In [31]: with open('writing_test.txt', 'w') as f:
             f.write('line1')
             f.write('line2\n')
             print('line3', file=f)
             print('line4', file=f)
```

Exercise: Read the iris.tsv file into a data structure of your choice, calculate the mean, median, mode, variance and range of each column and write this summary to a new file.

In general most files you want to read will have a package that is able to easily and reliably read and write them, for example `biopython` provides functions to deal with sequence files and `pandas`, which we will cover next session, allows you to read and write data tables. Usually it's best to use these methods where you can, since they are efficient and have been rigourously tested, but it's still good to know how to manually manipulate files when necessary.