IP2

February 21, 2015

1 Introduction to Python

This lecture is based loosely on the online tutorial : http://www.afterhoursprogramming.com/tutorial/Python/Introduction/

We will be using Python a fair amount in this class. Python is a high-level scripting language that offers an interactive programming environment. We assume programming experience, so this lecture will focus on the unique properties of Python.

Programming languages generally have the following common ingredients: variables, operators, iterators, conditional statements, functions (built-in and user defined) and higher-order data structures. We will look at these in Python and highlight qualities unique to this language.

1.1 Variables

Variables in Python are defined and typed for you when you set a value to them.

```
In [3]: my_variable = 2
        print(my_variable)
        type(my_variable)
```

```
2
```

```
Out[3]: int
```

This makes variable definition easy for the programmer. As usual, though, great power comes with great responsibility. For example:

```
In [4]: my_varible = my_variable+1
        print (my_variable)
```

```
2
```

"If you leave out word, spell-check will not put the word in you" – Taylor Mali, The the impotence of proofreading

If you accidentally mistype a variable name, Python will not catch it for you. This can lead to bugs that can be hard to track - so beware.

1.1.1 Types and Typecasting

The usual typecasting is available in Python, so it is easy to convert strings to ints or floats, floats to ints, etc. The syntax is slightly different than C:

```
In [7]: a = "1"
        b = 5
        print(a+b)
```

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

    <ipython-input-7-bf1e73fdfc72> in <module>()
      1 a="1"
      2 b=5
----> 3 print(a+b)


    TypeError: cannot concatenate 'str' and 'int' objects
```

```
In [8]: a = "1"
        b = 5
        print(int(a)+b)
```

```
6
```

Note that the typing is dynamic. I.e. a variable that was initally say an integer can become another type (float, string, etc.) via reassignment.

```
In [15]: a = "1"
         type(a)
         print(type(a))

         a = 1.0
         print(type(a))
```

```
<type 'str'>
<type 'float'>
```

Python has some other special data types such as lists, tuples and dictionaries that we will address later.

1.2 Operators

Python offers the usual operators such as +,-,/,*,=,>,<,==,!=,&,|, (sum, difference, divide, product, assignment, greater than, less than, equal - comparison,not equal, and, or, respectively).
Additionally, there are %,// and ** (modulo, floor division and 'to the power'). Note a few specifics:

```
In [23]: print(3/4)
         print(3.0 / 4.0)
         print(3%4)
         print(3//4)
         print(3**4)
```

```
0
0.75
3
0
81
```

Note the behavior of / when applied to integers! This is similar to the behavior of other strongly typed languages such as C/C++. The result of the integer division is the same as the floor division //. If you want the floating point result, the arguments to / must be floats as well (or appropriately typecast).

```
In [21]: a = 3
         b = 4
         print(a/b)
         print(float(a)/float(b))
```

```
0
0.75
```

1.3 Iterators

Python has the usual iterators, while, for, and some other constructions that will be addressed later. Here are examples of each:

```
In [29]: for i in range(1,10):
              print(i)
```

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

The most important thing to note above is that the range function gives us values up to, but not including, the upper limit.

```
In [34]: i = 1
         while i < 10:
             print(i)
             i+=1
```

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

This is unremarkable, so we proceeed without further comment.

1.4 Conditional Statements

```
In [37]: a = 20
         if a >= 22:
            print("if")
         elif a >= 21:
             print("elif")
         else:
             print("else")
```

```
else
```

Again, nothing remarkable here, just need to learn the syntax. Here, we should also mention spacing. Python is picky about indentation - you must start a newline after each conditional statemen (it is the same for the iterators above) and indent the same number of spaces for every statement within the scope of that condition.

```
In [42]: a = 23
         if a >= 22:
            print("if")
             print("greater than or equal 22")
         elif a >= 21:
             print("elif")
         else:
             print("else")


        IndentationError: unexpected indent
```

```
In [43]: a = 23
         if a >= 22:
            print("if")
            print("greater than or equal 22")
         elif a >= 21:
             print("elif")
         else:
             print("else")
```

```
if
greater than or equal 22
```

Four spaces are customary, but you can use whatever you like. Consistency is necessary.

Exceptions Python has another type of conditional expression that is very useful. Suppose your program is processing user input or data from a file. You don't always know for sure what you are getting in that case, and this can lead to problems. The 'try/except' conditional can solve them!

```
In [47]: a = "1"

         try:
           b = a + 2
         except:
           print(a, " is not a number")
```

```
('1', ' is not a number')
```

Here, we have tried to add a number and a string. That generates an exception - but we have trapped the exception and informed the user of the problem. This is much preferable to the programming crashing with some cryptic error like:

```
In [49]: a = "1"
         b = a + 2
```

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

    <ipython-input-49-1b1ef016a6b2> in <module>()
      1 a = "1"
----> 2 b = a + 2


    TypeError: cannot concatenate 'str' and 'int' objects
```

1.5 Functions

```
In [55]: def Division(a, b):
             print(a/b)
         Division(3,4)
         Division(3.0,4.0)
         Division(3,4.0)
         Division(3.0,4)
```

```
0
0.75
0.75
0.75
```

Notice that the function does not specify the types of the arguments, like you would see in statically typed languages. This is both useful and dangerous. For example:

```
In [56]: def Division(a, b):
             print(a/b)
         Division(2,"2")
```

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

    <ipython-input-56-663c15047357> in <module>()
      1 def Division(a, b):
      2     print(a/b)
----> 3 Division(2,"2")


    <ipython-input-56-663c15047357> in Division(a, b)
      1 def Division(a, b):
----> 2     print(a/b)
      3 Division(2,"2")


    TypeError: unsupported operand type(s) for /: 'int' and 'str'
```

In a statically typed language, the programmer would have specified the type of a and b (float, int, etc.) and the compiler would have complained about the function being passed a variable of the wrong type. This does not happen here, but we can use the try/except construction.

```
In [58]: def Division(a, b):
             try:
                 print(a/b)
             except:
                 if b == 0:
                    print("cannot divide by zero")
                 else:
                    print(float(a)/float(b))
         Division(2,"2")
         Division(2,0)
```

```
1.0
cannot divide by zero
```

2 Strings and String Handling

One of the most important features of Python is its powerful and easy handling of strings. Defining strings is simple enough in most languages. But in Python, it is easy to search and replace, convert cases, concatenate, or access elements. We'll discuss a few of these here. For a complete list, see: http://www.tutorialspoint.com/python/python_strings.htm

```
In [60]: a = "A string of characters, with newline \n CAPITALS, etc."
         print(a)
         b=5.0
         newstring = a + "\n We can format strings for printing %.2f"
         print(newstring %b)
```

```
A string of characters, with newline
 CAPITALS, etc.
A string of characters, with newline
 CAPITALS, etc.
 We can format strings for printing 5.00
```

Now let's try some other string operations:

```
In [2]: a = "ABC DEFG"
        print(a[1:3])
        print(a[0:5])
```

```
BC
ABC D
```

There are several things to learn from the above. First, Python has associated an index to the string. Second the indexing starts at 0, and lastly, the upper limit again means 'up to but not including' (a[0:5] prints elements 0,1,2,3,4).

```
In [72]: a = "ABC defg"
         print(a.lower())
         print(a.upper())
         print(a.find('d'))
         print(a.replace('de','a'))
         print(a)
         b = a.replace('def','aaa')
         print(b)
         b = b.replace('a','c')
         print(b)
         b.count('c')
```

```
abc defg
ABC DEFG
4
ABC afg
ABC defg
ABC aaag
ABC cccg

Out[72]: 3
```

This is fun! What else can you do with strings in Python? Pretty much anything you can think of!

3 Lists, Tuples, Dictionaries

3.1 Lists

Lists are exactly as the name implies. They are lists of objects. The objects can be any data type (including lists), and it is allowed to mix data types. In this way they are much more flexible than arrays. It is possible to append, delete, insert and count elements and to sort, reverse, etc. the list.

```
In [76]: a_list = [1,2,3,"this is a string",5.3]
         b_list = ["A","B","F","G","d","x","c",Alist,3]
         print(b_list)
         print(b_list[7:9])

['A', 'B', 'F', 'G', 'd', 'x', 'c', [1, 2, 3, 'this is a string', 5.3], 3]
[[1, 2, 3, 'this is a string', 5.3], 3]

In [92]: a = [1,2,3,4,5,6,7]
         a.insert(0,0)
         print(a)
         a.append(8)
         print(a)
         a.reverse()
         print(a)
         a.sort()
         print(a)
         a.pop()
         print(a)
         a.remove(3)
         print(a)
         a.remove(a[4])
         print(a)

[0, 1, 2, 3, 4, 5, 6, 7]
[0, 1, 2, 3, 4, 5, 6, 7, 8]
[8, 7, 6, 5, 4, 3, 2, 1, 0]
[0, 1, 2, 3, 4, 5, 6, 7, 8]
[0, 1, 2, 3, 4, 5, 6, 7]
[0, 1, 2, 4, 5, 6, 7]
[0, 1, 2, 4, 6, 7]
```

Just like with strings, elements are indexed beginning with 0.

Lists can be constructed using 'for' and some conditional statements. These are called, 'list comprehensions'. For example:

```
In [22]: even_numbers = [x for x in range(100) if x % 2 == 0]
         print(even_numbers)
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52,
```

List comprehensions can work on strings as well:

```
In [26]: first_sentence = "It was a dark and stormy night."
         characters = [x for x in first_sentence]
         print(characters)
```

```
['I', 't', ' ', 'w', 'a', 's', ' ', 'a', ' ', 'd', 'a', 'r', 'k', ' ', 'a', 'n', 'd', ' ', 's', 't', 'o
```

For more on comprehensions see: https://docs.python.org/2/tutorial/datastructures.html?highlight=comprehensions
Another similar feature is called 'map'. Map applies a function to a list. The syntax is map(aFunction, aSequence). Consider the following examples:

```
In [30]: def sqr(x): return x ** 2
         a = [2,3,4]
         b = [10,5,3]
         c = map(sqr,a)
         print(c)
         d = map(pow,a,b)
         print(d)
```

```
[4, 9, 16]
[1024, 243, 64]
```

Note that map is usually more efficient than the equivalent list comprehension or looping contruct.

3.1.1 Tuples

Tuples are like lists with one very important difference. Tuples are not changeable.

```
In [94]: a = (1,2,3,4)
         print(a)
         a[1] = 2
```

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

<ipython-input-94-3f7ee3924a64> in <module>()
  1 a=(1,2,3,4)
  2 print(a)
----> 3 a[1]=2


TypeError: 'tuple' object does not support item assignment
```

```
(1, 2, 3, 4)
```

```
In [96]: a = (1,"string in a tuple",5.3)
         b = (a,1,2,3)
         print(a)
         print(b)
```

```
(1, 'string in a tuple', 5.3)
((1, 'string in a tuple', 5.3), 1, 2, 3)
```

As you can see, all of the other flexibility remains - so use tuples when you have a list that you do not want to modify.

One other handy feature of tuples is known as 'tuple unpacking'. Essentially, this means we can assign the values of a tuple to a list of variable names, like so:

```
In [19]: my_pets = ("Chestnut", "Tibbs", "Dash", "Bast")
         (aussie,b_collie,indoor_cat,outdoor_cat) = my_pets
         print(aussie)
         cats=(indoor_cat,outdoor_cat)
         print(cats)

Chestnut
('Dash', 'Bast')
```

3.2 Dictionaries

Dictionaries are unordered, keyed lists. Lists are ordered, and the index may be viewed as a key.

```
In [99]: a = ["A","B","C","D"] #list example
         print(a[1])

B

In [3]: a = {'anItem': "A", 'anotherItem': "B",'athirdItem':"C",'afourthItem':"D"} # dictionary example
        print(a[1])


        ---------------------------------------------------------------------------
    KeyError                                  Traceback (most recent call last)

        <ipython-input-3-d314b6c47c6e> in <module>()
          1 a = {'anItem': "A", 'anotherItem': "B",'athirdItem':"C",'afourthItem':"D"} # dictionary exa
    ----> 2 print(a[1])


        KeyError: 1


In [104]: a = {'anItem': "A", 'anotherItem': "B",'athirdItem':"C",'afourthItem':"D"} # dictionary examp
          print(a['anItem'])
          print(a)

A
{'athirdItem': 'C', 'afourthItem': 'D', 'anItem': 'A', 'anotherItem': 'B'}
```

The dictionary does not order the items, and you cannot access them assuming an order (as an index does). You access elements using the keys.

3.2.1 Sets

Sets are unordered collections of *unique* elements. Intersections, unions and set differences are supported operations. They can be used to remove duplicates from a collection or to test for membership. For example:

```
In [11]: from sets import Set
         fruits = Set(["apples","oranges","grapes","bananas"])
         citrus = Set(["lemons","oranges","limes","grapefruits","clementines"])
         citrus_in_fruits = fruits & citrus   #intersection
         print(citrus_in_fruits)
         diff_fruits = fruits - citrus        # set difference
         print(diff_fruits)
         diff_fruits_reverse = citrus - fruits  # set difference
         print(diff_fruits_reverse)
         citrus_or_fruits = citrus | fruits     # set union
         print(citrus_or_fruits)
```

```
Set(['oranges'])
Set(['apples', 'grapes', 'bananas'])
Set(['grapefruits', 'clementines', 'lemons', 'limes'])
Set(['clementines', 'grapes', 'limes', 'oranges', 'grapefruits', 'apples', 'lemons', 'bananas'])
```

```
In [15]: a_list = ["a", "a","a", "b",1,2,3,"d",1]
         print(a_list)
         a_set = Set(a_list)  # Convert list to set
         print(a_set)         # Creates a set with unique elements
         new_list = list(a_set) # Convert set to list
         print(new_list)        # Obtain a list with unique elements
```

```
['a', 'a', 'a', 'b', 1, 2, 3, 'd', 1]
Set(['a', 1, 2, 'b', 'd', 3])
['a', 1, 2, 'b', 'd', 3]
```

More examples and details regarding sets can be found at: https://docs.python.org/2/library/sets.html

3.3 Classes

A class (or object) bundles data (known as attributes) and functions (known as methods) together. We access the attributes and mehtods of a class using the '.' notation. Since everything in Python is an object, we have already been using this attribute acccess - e.g. when we call `'hello'.upper()`, we are using the `upper` method of the instance `'hello'` of the `string` class.

The creation of custom classes will not be covered in this course.

3.4 Modules

As the code base gets larger, it is convenient to organize them as *moudles* or packages. At the simplest level, modules can just be regular python files. We import functions in modules using one of the following `import` variants:

```
import numpy
import numpy as np # using an alias
import numpy.linalg as la # modules can have submodules
from numpy import sin, cos, tan # bring trig functions into global namespace
from numpy import * # frowned upon because it pollutes the namespace
```

3.5 The standard library

Python comes with "batteries included", with a diverse collection of functionality available in standard library moduules and functions.

References

- Standard library docs
- Python Module of the Week gives examples of usage.

3.5.1 Installing additional modules

Most of the time, we can use the `pip` package manager to install and uninstall modules for us. In general, all that is needed is to issue the command

```
pip install <packagename>
```

at the command line or

```
! pip install <packagename>
```

from within an IPython notebook.
Pakcage that can be installed using `pip` are listed in the Python Package Index (PyPI).
Pip documentation is at `https://pip.pypa.io/en/latest/`.

3.6 Keeping the Anaconda distributoin up-to-date

Just issue

```
conda update conda
conda update anaconda
```

at the command line.
Note that `conda` can do much, much, more.

3.7 Exercises

1. Solve the FizzBuzz probelm
"Write a program that prints the numbers from 1 to 100. But for multiples of three print "Fizz" instead of the number and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz".

```
In []: # YOUR CODE HERE
```

2. Given x=3 and y=4, swap the values of x and y so that x=4 and y=3.

```
In []: x = 3
       y = 4
       # YOUR CODE HERE
```

3. Write a function that calculates and returns the euclidean distance between two points u and v, where u and v are both 2-tuples (x, y). For example, if $u = (3, 0)$ and $v = (0, 4)$, the function should return 5.

```
In []: # YOUR CODE HERE
```

4. Using a dictionary, write a program to calculate the number times each character occurs in the given string s. Ignore differneces in capitalization - i.e 'a' and 'A' should be treated as a single key. For example, we should get a count of 7 for 'a'.

```
In [6]: s = """
        Write a program that prints the numbers from 1 to 100.
        But for multiples of three print 'Fizz' instead of the number and f
        or the multiples of five print 'Buzz'. For numbers which are
        multiples of both three and five print 'FizzBuzz'
        """

        # YOUR CODE HERE
```

5. Write a program that finds the percentage of sliding windows of length 5 for the sentence s that contain at least one 'a'. Ignore case, spaces and punctuation. For example, the first sliding window is 'write' which contains 0 'a's, and the second is 'ritea' which contains 1 'a'.

```
In []: s = """
       Write a program that prints the numbers from 1 to 100.
       But for multiples of three print 'Fizz' instead of the number and f
       or the multiples of five print 'Buzz'. For numbers which are
       multiples of both three and five print 'FizzBuzz'
       """

       # YOUR CODE HERE
```

6. Find the unique numbers in the following list.

```
In [15]: x = [36, 45, 58, 3, 74, 96, 64, 45, 31, 10, 24, 19, 33, 86, 99, 18, 63, 70, 85,
          85, 63, 47, 56, 42, 70, 84, 88, 55, 20, 54, 8, 56, 51, 79, 81, 57, 37, 91,
          1, 84, 84, 36, 66, 9, 89, 50, 42, 91, 50, 95, 90, 98, 39, 16, 82, 31, 92, 41,
          45, 30, 66, 70, 34, 85, 94, 5, 3, 36, 72, 91, 84, 34, 87, 75, 53, 51, 20, 89, 51, 20]

         # YOUR CODE HERE
```

7. Write two functions - one that returns the square of a number, and one that returns the cube. Now write a third function that returns the number raised to the 6^{th} power using the two previous functions.

```
In [16]: # YOUR CODE HERE

         def square(x):
             pass

         def cube(x):
             pass

         def f(x):
             pass
```

8. Create a list of the cubes of x for x in [0, 10] using

- a for loop
- a list comprehension
- the map function

```
In []: # YOUR CODE HERE
```

9. A Pythoagorean triple is an integer solution to the Pythagorean theorem $a^2 + b^2 = c^2$. The first Pythagonrean triple is (3,4,5). Find all unique Pythagorean triples for the positive integers a, b and c less than 100.

```
In []: # YOUR CODE HERE
```

10. Fix the bug in this function that is intended to take a list of numbers and return a list of normalized numbers.

```
def f(xs):
    """Retrun normalized list summing to 1."""
    s = 0
    for x in xs:
        s += x
    return [x/s for x in xs]
```

```
In [2]: # YOUR CODE HERE
```

```
In []:
```