1-Intro-to-Python

September 1, 2016

1 Introduction to Python

For those of you that know python, today's class aims to refresh your memory.

For those of you that don't know python – but do know programming – this lecture is to give you an idea of how python is similar to, and different from, your favorite programming language.

1.1 Environment

There are four ways to run python code:

- put your code in a file (say program.py) and run `python program.py`
- This is least desirable when you are first writing your code
- Later, once your code is debugged, this is the way to go

- type your code into the python interpreter
- This allows you to interact with the interpreter and fix mistakes as they happen
- However you have to type everything by hand

- type, cut/paste, or run your code in `ipython`
- This is a good method
- Allows you to cut/paste from a file you are working on

- run `ipython` in a browser, called `jupyter notebook`
- This is even better
- All the advantages of `ipython` plus interleaved documentation and graphical output
- That is what these slides are in

1.2 Python 2 vs Python 3

There are two versions of Python. I am using Python 3 and that's what I recommend. Here are the main differences:

In Python 2, integer division does *not* return a floating point value:
$3/5 = 1$
In Python 3, integer division does return a floating point value:

```
In [2]: 3/5
```

```
Out[2]: 0.6
```

In Python 2, the print statement does not use parentheses:
print 'a string'
In Python 3, print is a function and it uses parentheses:

```
In [3]: print('a string')
```

```
a string
```

1.3 Functions and Methods

Function calls use standard syntax:

```
func(argument1, argument2)
```

However most things you interact with in python are **objects** and they have **methods**. A method is a function that operates on an object:

```
object.method(argument1, argument2)
```

Note that the method might modify the object, or it might return a new, different object. You just have to know the method and keep track of what it does.

```
In [4]: number_list = [1, 3, 5, 7]
        number_list.append(8)

In [5]: number_list

Out[5]: [1, 3, 5, 7, 8]

In [6]: string = 'This is a string'
        string.split()

Out[6]: ['This', 'is', 'a', 'string']

In [7]: string

Out[7]: 'This is a string'
```

1.4 Printing

From the interactive python environment:

```
In [8]: print("Hello World")

Hello World
```

From a file:

```
In [9]: #!/usr/bin/env python

        print("Hello World!")

Hello World!
```

1.5 Data types

Basic data types:

1. Strings
2. Integers
3. Floats
4. Booleans

These are all objects in Python.

```
In [10]: a = "apple"
         type(a)
```

```
Out[10]: str

In [11]: b = 3
         type(b)

Out[11]: int

In [12]: c = 3.2
         type(c)

Out[12]: float

In [13]: d = True
         type(d)

Out[13]: bool
```

Python **doesn't require explicitly declared variable types** like C and other languages. Python is dynamically typed.

```
In [14]: myVar = 'I am a string'
         print(myVar)
         myVar = 2.3
         print(myVar)

I am a string
2.3
```

1.6 Strings

String manipulation will be very important for many of the tasks we will do. Here are some important string operations.

A string uses either single quotes or double quotes. Pick one option and be consistent.

```
In [15]: 'This is a string'

Out[15]: 'This is a string'

In [16]: "This is also a string"

Out[16]: 'This is also a string'
```

The '+' operator concatenates strings.

```
In [17]: a = "Hello"
         b = " World"
         a + b

Out[17]: 'Hello World'
```

Portions of strings are manipulated using indexing (which python calls 'slicing').

```
In [18]: a = "World"

         a[0]

Out[18]: 'W'

In [19]: a[-1]
```

```
Out[19]: 'd'

In [20]: "World"[0:4]

Out[20]: 'Worl'

In [21]: a[::-1]

Out[21]: 'dlroW'
```

Some important string functions:

```
In [22]: a = "Hello World"
         "-".join(a)

Out[22]: 'H-e-l-l-o- -W-o-r-l-d'

In [23]: a.startswith("Wo")

Out[23]: False

In [24]: a.endswith("rld")

Out[24]: True

In [25]: a.replace("o","0").replace("d","[)").replace("l","1")

Out[25]: 'He110 W0r1[)'

In [26]: a.split()

Out[26]: ['Hello', 'World']

In [27]: a.split('o')

Out[27]: ['Hell', ' W', 'rld']
```

Strings are an example of an **immutable** data type. Once you instantiate a string you cannot change any characters in it's set.

```
In [28]: string = "string"
         # string[-1] = "y"  # This will generate and error as we attempt to modify the string
```

To create a string with embedded objects use the `.format()` method:

```
In [29]: course_name = 'CS505'
         enrollment = 75
         percent_full = 100.0
         'The course {} has an enrollment of {} and is {} percent full.'.format(
             course_name,enrollment,percent_full)

Out[29]: 'The course CS505 has an enrollment of 75 and is 100.0 percent full.'
```

1.7 Code Structure

Python uses indents and whitespace to group statements together. To write a short loop in C, you might use:

```
c  for (i = 0, i < 5, i++){        printf("Hi! \n");     }
```

Python does not use curly braces like C, so the same program as above is written in Python as follows:

```
In [30]: for i in range(5):
             print("Hi")
```

```
Hi
Hi
Hi
Hi
Hi
```

If you have nested for-loops, there is a further indent for the inner loop.

```
In [31]: for i in range(3):
             for j in range(3):
                 print(i, j)

             print("This statement is within the i-loop, but not the j-loop")
```

```
0 0
0 1
0 2
This statement is within the i-loop, but not the j-loop
1 0
1 1
1 2
This statement is within the i-loop, but not the j-loop
2 0
2 1
2 2
This statement is within the i-loop, but not the j-loop
```

1.8 File I/O

`open()` and `close()` are used to access files. However if you use the `with` statement the file close is automatically done for you.

You should use `with`.

```
In [32]: with open("example.txt", "w") as f:
             f.write("Hello World! \n")
             f.write("How are you? \n")
             f.write("I'm fine. OK.\n")
```

Reading from a file:

```
In [33]: with open("example.txt", "r") as f:
             data = f.readlines()
             for line in data:
                 words = line.split()
                 print(words)
```

```
['Hello', 'World!']
['How', 'are', 'you?']
["I'm", 'fine.', 'OK.']
```

Here is an example of counting the number of lines and words in a file:

```
In [34]: lines = 0
         words = 0
         the_file = "example.txt"

         with open(the_file, 'r') as f:
             for line in f:
                 lines += 1
                 words += len(line.split())
         print("There are {} lines and {} words in the {} file.".format(
                 lines, words, the_file))
```

```
There are 3 lines and 8 words in the example.txt file.
```

1.9 Lists, Tuples, Sets and Dictionaries

Number and strings alone are not enough! We need data types that can hold multiple values.

1.9.1 Lists:

A list is a collection of data items, which can be of differing types.

Here is an empty list:

```
In [35]: groceries = []
```

A list is **mutable**, meaning that it can be altered.

Adding to the list:

```
In [36]: groceries.append("oranges")
         groceries.append("meat")
         groceries.append("asparagus")
         groceries
```

```
Out[36]: ['oranges', 'meat', 'asparagus']
```

Accessing list items by index:

```
In [37]: groceries[0]
```

```
Out[37]: 'oranges'
```

```
In [38]: groceries[2]
```

```
Out[38]: 'asparagus'
```

```
In [39]: len(groceries)
```

```
Out[39]: 3
```

Sort the items in the list:

```
In [40]: groceries.sort()
         groceries
```

```
Out[40]: ['asparagus', 'meat', 'oranges']
```

Remove an item from a list:

```
In [41]: groceries.remove('asparagus')
         groceries

Out[41]: ['meat', 'oranges']
```

Because lists are mutable, you can arbitrarily modify them.

```
In [42]: groceries[0] = 'peanut butter'
         groceries

Out[42]: ['peanut butter', 'oranges']
```

1.9.2 List Comprehensions

A **list comprehension** makes a new list from an old list. It is incredibly useful (learn how to use it!)

```
In [43]: groceries = ['asparagus', 'meat', 'oranges']
         veggie = [x for x in groceries if x is not "meat"]
         veggie

Out[43]: ['asparagus', 'oranges']
```

This is the same as:

```
In [44]: newlist = []
         for x in groceries:
             if x is not 'meat':
                 newlist.append(x)
         newlist

Out[44]: ['asparagus', 'oranges']
```

Recall the mathematical notation:

$$L_1 = \left\{x^2 : x \in \{0 \ldots 9\}\right\}$$

$$L_2 = \left\{1, 2, 4, 8, \ldots, 2^{12}\right\}$$

$$M = \{x \mid x \in L_1 \text{ and } x \text{ is even}\}$$

```
In [45]: L1 = [x**2 for x in range(10)]
         L2 = [2**i for i in range(13)]
         print('L1 is {}'.format(L1))
         print('L2 is {}'.format(L2))

L1 is [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
L2 is [1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096]

In [46]: M = [x for x in L1 if x % 2 == 0]
         print('M is {}'.format(M))

M is [0, 4, 16, 36, 64]
```

A sort-of "Sieve of Eratosthenes" in list comprehensions.

Basic idea: generate all composite numbers, remove them from the set of all numbers, and what is left are the prime nnumbers.

```
In [47]: composites = [i*j for i in range(2,8) for j in range(2,8)]
```

```
In [48]: primes = [x for x in range(1,50) if x not in composites]
         print(primes)
```

```
[1, 2, 3, 5, 7, 11, 13, 17, 19, 22, 23, 26, 27, 29, 31, 32, 33, 34, 37, 38, 39, 40, 41, 43, 44, 45, 46,
```

Notice how much more concise and clear the list comprehension is. It's more efficient too.

1.9.3 Sets:

A set is a collecton of items that cannot contain duplicates. Sets handle operations like sets in mathematics.

```
In [49]: numbers = range(10)
         numbers = set(numbers)

         evens = {0, 2, 4, 6, 8}

         odds = numbers - evens
         odds
```

```
Out[49]: {1, 3, 5, 7, 9}
```

Sets also support the use of union (|), and intersection (&)

1.9.4 Dictionaries:

A dictionary is a map of keys to values. **Keys must be unique**.

```
In [50]: simple_dict = {}

         simple_dict['cs505'] = 'data-mining tools'

         simple_dict['cs505']
```

```
Out[50]: 'data-mining tools'
```

Creating an already-initialized dictionary. Note the use of curly braces.

```
In [51]: classes = {
             'cs505': 'data-mining tools',
             'cs565': 'data-mining algorithms'
         }
```

Check if item is in dictionary

```
In [52]: 'cs530' in classes
```

```
Out[52]: False
```

Add new item

```
In [53]: classes['cs530'] = 'algorithms'
         classes['cs530']
```

```
Out[53]: 'algorithms'
```

Get just the keys

```
In [54]: classes.keys()

Out[54]: dict_keys(['cs565', 'cs530', 'cs505'])
```

Get just the values

```
In [55]: classes.values()

Out[55]: dict_values(['data-mining algorithms', 'algorithms', 'data-mining tools'])
```

Get the items in the dictionary

```
In [56]: classes.items()

Out[56]: dict_items([('cs565', 'data-mining algorithms'), ('cs530', 'algorithms'), ('cs505', 'data-minir
```

Get dictionary pairs another way

```
In [57]: for key, value in classes.items():
             print(key, value)

cs565 data-mining algorithms
cs530 algorithms
cs505 data-mining tools
```

Dictionaries can be combined to make complex (and very useful) data structures.
Here is a list within a dictionary within a dictionary.

```
In [58]: professors = {
             "prof1": {
                 "name": "Evimaria Terzi",
                 "interests": ["algorithms", "data mining", "machine learning"]
             },
             "prof2": {
                 "name": "Mark Crovella",
                 "interests": ["computer networks", "data mining", "biological networks"]
             },
             "prof3": {
                 "name": "George Kollios",
                 "interests": ["databases", "data mining"]
             }
         }

In [59]: for prof in professors:
             print('{} is interested in {}.'.format(
                     professors[prof]["name"],
                     professors[prof]["interests"][0]))

Mark Crovella is interested in computer networks.
George Kollios is interested in databases.
Evimaria Terzi is interested in algorithms.
```

1.9.5 Tuples:

Tuples are an **immutable** type. Like strings, once you create them, you cannot change them.

Because they are immutabile you can use them as keys in dictionaries.

However, they are similar to lists in that they are a collection of data and that data can be of differing types.

Here is a tuple version of our grocery list.

```
In [60]: groceries = ('orange', 'meat', 'asparagus', 2.5, True)
         groceries

Out[60]: ('orange', 'meat', 'asparagus', 2.5, True)

In [61]: groceries[2]

Out[61]: 'asparagus'
```

What will happen here?

```
In [62]: # groceries[2] = 'milk'
```

1.9.6 Iterators and Generators

We can loop over the elements of a list using **for**

```
In [63]: for i in [1,2,3,4]:
             print(i)

1
2
3
4
```

When we use **for** for dictionaries it loops over the keys of the dictionary

```
In [64]: for k in {'evimaria': 'terzi', 'george': 'kollios'}:
             print(k)

george
evimaria
```

When we use **for** for strings it loops over the letters of the string:

```
In [65]: for l in 'python is magic':
             print(l)

p
y
t
h
o
n

i
s

m
a
g
i
c
```

What do these cases all have in common? All of them are **iterable** objects.

```
In [66]: list({'evimaria': 'terzi', 'george': 'kollios'})

Out[66]: ['george', 'evimaria']

In [67]: list('python is magic')

Out[67]: ['p', 'y', 't', 'h', 'o', 'n', ' ', 'i', 's', ' ', 'm', 'a', 'g', 'i', 'c']

In [68]: '-'.join('evimaria')

Out[68]: 'e-v-i-m-a-r-i-a'

In [69]: '-'.join(['a','b','c'])

Out[69]: 'a-b-c'
```

1.10 Defining Functions

```
In [70]: def displayperson(name,age):
             print("My name is {} and I am {} years old.".format(name,age))
             return

         displayperson("Larry","40")
```

```
My name is Larry and I am 40 years old.
```

1.11 Functional Programming

Functional programming is particularly valuable and common when working with data. We'll see more sophisticated examples of this sort of programming later.

1.11.1 Lambda functions

Python supports the creation of anonymous functions (i.e. functions that are not bound to a name) at runtime, using a construct called `lambda`.

```
In [71]: def f(x):
             return x**2
         f(8)

Out[71]: 64

In [72]: g = lambda x: x**2
         g(8)

Out[72]: 64

In [73]: (lambda x: x**2)(8)

Out[73]: 64
```

The above pieces of code are all equivalent! Note that there is no `return` statement in the `lambda` function. Instead there is just a single expression, which defines what the function returns.

A `lambda` function can take multiple arguments. However it has to get all its work done in a single line of code!

```
In [74]: f = lambda x, y : x + y
         f(2,3)

Out[74]: 5
```

A `lambda` function does not need to be assigned to variable, but it can be used within the code wherever a function is expected.

Here is an example of 'currying': a function that returns a new function, with some of the original arguments bound.

```
In [75]: def multiply (n):
             return lambda x: x*n

         f = multiply(2)
         g = multiply(6)
         f

Out[75]: <function __main__.multiply.<locals>.<lambda>>

In [76]: f(10)

Out[76]: 20

In [77]: g(10)

Out[77]: 60

In [78]: multiply(3)(30)

Out[78]: 90
```

1.11.2 Map

Our first example of functional programming will be the **map** operator:

`r = map(func, s)`

`func` is a function and `s` is a sequence (e.g., a list).

`map()` returns an object that will apply function `func` to each of the elements of `s`.

```
In [79]: def dollar2euro(x):
             return 0.89*x
         def euro2dollar(x):
             return 1.12*x

         amounts= (100, 200, 300, 400)
         dollars = map(dollar2euro, amounts)
         list(dollars)

Out[79]: [89.0, 178.0, 267.0, 356.0]

In [80]: amounts= (100, 200, 300, 400)
         euros = map(euro2dollar, amounts)
         list(euros)

Out[80]: [112.00000000000001,
          224.00000000000003,
          336.00000000000006,
          448.00000000000006]
```

```
In [81]: list(map(lambda x: 0.89*x, amounts))
```

```
Out[81]: [89.0, 178.0, 267.0, 356.0]
```

map can also be applied to more than one list as long as they are of the same size and type

```
In [82]: a = [1,2,3,4,5]
         b = [10, 20 , 30, 40, 50]

         l1 = map(lambda x,y: x+y, a,b)
         list(l1)
```

```
Out[82]: [11, 22, 33, 44, 55]
```

1.11.3 Filter

The next functional operator is **filter**.

`filter(function, list)` returns a new list containing all the elements of `list` for which `function()` evaluates to `True`.

```
In [83]: nums = [i for i in range(100)]
         even = filter(lambda x: x%2==0 and x!=0, nums)
         print(even)
```

```
<filter object at 0x10c647b38>
```

1.11.4 Reduce

The last functional operator is `reduce()`.

The job of `reduce(function,list)` is to return a single value that combines all the elements of the list.

`reduce(function, list)` sequentially applies `function()` to its previously returned value, and the next element of `list`.

For example if `list = [a1,a2,a3,...,a10]`, then the first step of `reduce(function, list)` will compute `[function(a1,a2),a3,...,a10]`, and so on.

The function supplied to reduce is typically commutative.

```
In [84]: from functools import reduce
         reduce(lambda x,y: x+y, [x for x in range(10)])
```

```
Out[84]: 45
```

```
In [85]: reduce (lambda x,y: x if x>y else y, [1, 15, 26, -27])
```

```
Out[85]: 26
```

1.12 Libraries

Python is a high-level open-source language. But the *Python world* is inhabited by many packages or libraries that provide useful things like array operations, plotting functions, and much more. We can (and we will) import many different libraries of functions to expand the capabilities of Python in our programs.

```
In [86]: import random
         myList = [2, 109, False, 10, "data", 482, "mining"]
         random.choice(myList)
```

```
Out[86]: False
```

```
In [87]: from random import shuffle
         x = list(range(10))
         shuffle(x)
         x
```

```
Out[87]: [9, 4, 5, 0, 8, 1, 3, 6, 7, 2]
```

1.13 APIs

For example, there are libraries that make it easy to interact with RESTful APIs.

A RESTful API is a service available on the Internet that uses the HTTP protocol for access.

```
In [88]: import requests

         width = '200'
         height = '300'
         response = requests.get('http://loremflickr.com/' + width + '/' + height)

         print(response)

         with open('img.jpg', 'wb') as f:
             f.write(response.content)
```

```
<Response [200]>
```

```
In [89]: from IPython.display import Image
         Image(filename="img.jpg")
```

```
Out[89]:
```