Lecture Notes 1: Python Basics

Introduction

- Python is an interpreted language (no need to compile the code, like C or Java)
- Python is a scripting language (compact code, fast prototyping)
- Python is the most frequently teached programming language in CS departments of universities

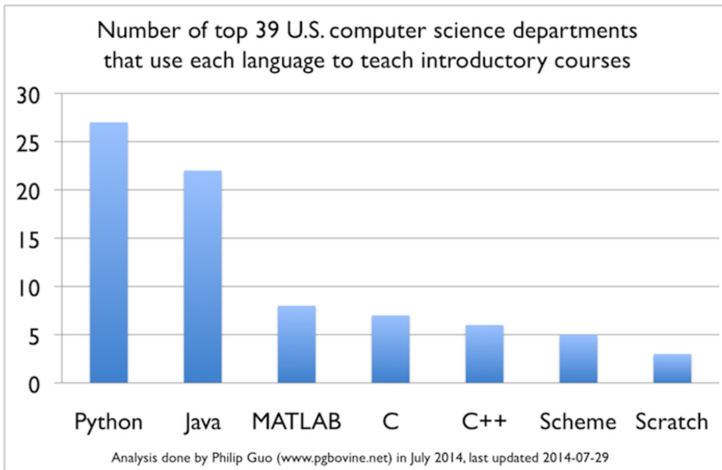


Figure 1: Source: Philip Guo, pgbovine.net

- Many machine learning libraries are developed in Python (e.g. Scikit-Learn, PyBrain, Theano, . . .)
- Fast machine learning libraries written in other languages also have Python bindings (e.g. LibSVM, SHOGUN)

Hello world

```
In [1]: print('hello world')
```

```
hello world
```

```
In [2]: print 'hello world'
```

```
hello world
```

```
In [3]: 'hello world'
```

```
Out[3]: 'hello world'
```

Typing and casting

```
In [4]: x = 1
        x,type(x)
```

```
Out[4]: (1, int)
```

```
In [5]: x = 1.0
        x,type(x)
```

Out[5]: (1.0, float)

```
In [6]: x = float(1)
        x,type(x)
```

Out[6]: (1.0, float)

```
In [7]: x = int(1.0)
        x,type(x)
```

Out[7]: (1, int)

Operators

```
In [8]: 1+2
```

Out[8]: 3

```
In [9]: 1.0+2.0
```

Out[9]: 3.0

Operators can be applied to more complex types of objects, and the way they apply depend on these types:

```
In [10]: list([1,2,3])+list([2,3,4])
```

Out[10]: [1, 2, 3, 2, 3, 4]

Precedence of operators

```
In [11]: 1+2*3
```

Out[11]: 7

```
In [12]: (1+2)*3
```

Out[12]: 9

```
In [13]: 1.0/2.0/2.0
```

Out[13]: 0.25

```
In [14]: 1.0/(2.0/2.0)
```

Out[14]: 1.0

Exhaustive list:
In case you are not sure, add parentheses.

Operators and casting

```
In [15]: x = 3 / 2
         x,type(x)
```

Out[15]: (1, int)

```
In [16]: x = 3.0 / 2.0
         x,type(x)
```

Out[16]: (1.5, float)

```
In [17]: x = 3 / 2.0
         x,type(x)
```

Out[17]: (1.5, float)

Operator	Description
()	Parentheses (grouping)
f(args...)	Function call
x[index:index]	Slicing
x[index]	Subscription
x.attribute	Attribute reference
**	Exponentiation
~x	Bitwise not
+x, -x	Positive, negative
*, /, %	Multiplication, division, remainder
+, -	Addition, subtraction
<<, >>	Bitwise shifts
&	Bitwise AND
^	Bitwise XOR
		Bitwise OR
in, not in, is, is not, <, <=, >, >=, <>, !=, ==	Comparisons, membership, identity
not x	Boolean NOT
and	Boolean AND
or	Boolean OR
lambda	Lambda expression

Figure 2: Source: thepythonguru.com

Functions

```
In [18]: def f(x):
             y = x**2
             return y

         f(2)
```

```
Out[18]: 4
```

A function can be seen as a variable

```
In [19]: f = lambda x: x**2
         f(2)
```

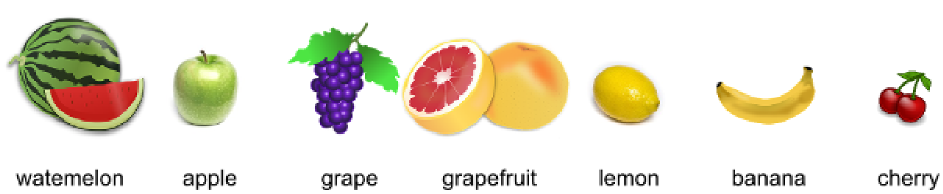
```
Out[19]: 4
```

A function does not even need a name

```
In [20]: (lambda x: x**2)(2)
```

```
Out[20]: 4
```

Conditional expressions



Example 1: Classifying apples vs. rest

```
In [21]: def classify(x):

             if x['size'] == 'medium' and \
               (x['color']=='green' or x['color']=='red' or \
                 (x['color']=='yellow' and x['taste']=='sweet')):
                 return 'apple'
             else:
                 return 'not an apple'
```

Example 2: Full decision tree (inspired from Duda et al. Pattern Classification)

```
In [22]: def classifybetter(x):

             if x['color']=='green':
                 if x['size']=='big':
                     return 'watermelon'
                 elif x['size']=='medium':
                     return 'apple'
                 elif x['size']=='small':
                     return 'grape'
                 else:
                     return 'unknown'

             elif x['color']=='yellow':
                 if x['size']=='big':
                     return 'grapefruit'
                 elif x['size']=='medium':
                     if x['shape'] == 'round':
                         if x['taste'] == 'sour':
                             return 'lemon'
                         elif ['taste'] == 'sweet':
                             return 'apple'
                         else:
                             return 'unknown'
                     elif x['shape'] == 'long':
                         return 'banana'
                     else:
                         return 'unknown'
                 else:
                     return 'unknown'

             elif x['color'] == 'red':
                 if x['size'] == 'medium':
                     return 'apple'
                 if x['size'] == 'small':
                     if x['taste'] == 'sweet':
                         return 'cherry'
                     if x['taste'] == 'sour':
                         return 'grape'

             else:
                 return 'unknown'
```

Collect a data point and classify it

```
In [29]: # Collecting a data point
         x1 = {}
         x1['size'] = raw_input('size (small/medium/big): ')
         x1['color'] = raw_input('color (red/yellow/green): ')
         x1['taste'] = raw_input('taste (sweet/sour): ')
         x1

size (small/medium/big): small
color (red/yellow/green): red
taste (sweet/sour): sweet

Out[29]: {'color': 'red', 'size': 'small', 'taste': 'sweet'}
```

```
In [30]: # Classify the data point
         classify(x1)

Out[30]: 'not an apple'
```

```
In [31]: # Classify the data point
         classifybetter(x1)

Out[31]: 'cherry'
```

Iterators

Making predictions for multiple observations

```
In [32]: data = [
           {'color':'green','size':'big'},
           {'color':'yellow','shape':'round','size':'big'},
           {'color':'red','size':'medium'},
           {'color':'red','size':'small','taste':'sour'},
           {'color':'green','size':'small'}
         ]
```

```
In [33]: cl = []
         for x in data:
             cl += [classifybetter(x)]
         cl

Out[33]: ['watermelon', 'grapefruit', 'apple', 'grape', 'grape']
```

The same can be achieved with list comprehensions:

```
In [34]: [classifybetter(x) for x in data]

Out[34]: ['watermelon', 'grapefruit', 'apple', 'grape', 'grape']
```

The same can also be achieved with the map function:

```
In [35]: map(classifybetter,data)

Out[35]: ['watermelon', 'grapefruit', 'apple', 'grape', 'grape']
```

Counting the number of objects "grape" in the data

```
In [36]: sum([(1 if classifybetter(x) == 'grape' else 0) for x in data])
```

```
Out[36]: 2
```

Or similarly

```
In [37]: len(filter(lambda x: classifybetter(x) == 'grape',data))
```

```
Out[37]: 2
```

Or similarly

```
In [38]: reduce(lambda x,y: x+(1 if classifybetter(y) == 'grape' else 0),data,0)
```

```
Out[38]: 2
```

Reading Data from a File

Content of file scores.txt that lists the performance players at a certain game:

```
80,55,16,26,37,62,49,13,28,56
43,45,47,63,43,65,10,52,30,18
63,71,69,24,54,29,79,83,38,56
46,42,39,14,47,40,72,43,57,47
61,49,65,31,79,62,9,90,65,44
10,28,16,6,61,72,78,55,54,48
```

The following program reads the file and stores the scores into a list

```
In [39]: scores = []
         f = open('scores.txt')
         for line in f:
             for entry in line.split(","):
                 scores += [int(entry)]
```

The same program can also be written in more compact form as

```
In [40]: scores = sum([map(int,l[:-1].split(',')) \
                       for l in open('scores.txt','r')],[])
```

Classes

Classes are useful for modeling anything that has an internal state, for example, machine learning classifiers.

```
In [36]: class ScoresTracker:

             def __init__(self):

                 self.best  = 0
                 self.mean  = 50.0
                 self.n     = 0

             def add(self,score):

                 # Make a comment about the new score
                 if   score > self.best: print '%d is a new record'%score
                 elif score > self.mean: print '%d is above average'%score
```

```
            else:                    print '%d is below average'%score

            # Update internal state
            self.best = max(self.best,score)
            self.mean = (self.mean*self.n + score)/(self.n+1)
            self.n    = self.n+1

        type(ScoresTracker)
```

Out[36]: classobj

```
In [37]: scores = ScoresTracker()
         type(scores)
```

Out[37]: instance

```
In [38]: scores.add(92)
         scores.add(60)
         scores.add(20)
         scores.add(60)
         scores.add(92)
```

```
92 is a new record
60 is below average
20 is below average
60 is above average
92 is above average
```