Lecture Notes 1: Python Basics

Hello world

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

```
hello world
```

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

```
Out[2]: 'hello world'
```

Operators, Types and Casting

```
In [3]: 4/3,type(4),type(3),type(4/3)
```

```
Out[3]: (1, int, int, int)
```

```
In [4]: 4.0/3.0,type(4.0),type(3.0),type(4.0/3.0)
```

```
Out[4]: (1.3333333333333333, float, float, float)
```

```
In [5]: 4/3.0,type(4),type(3.0),type(4/3.0)
```

```
Out[5]: (1.3333333333333333, int, float, float)
```

```
In [6]: int(4.0)/int(3.0)
```

```
Out[6]: 1
```

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

```
In [7]: 1+2
```

```
Out[7]: 3
```

```
In [8]: [1,2,3]+[2,3,4]
```

```
Out[8]: [1, 2, 3, 2, 3, 4]
```

Precedence of operators

```
In [9]: 1*2+3*4
```

```
Out[9]: 14
```

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

```
Out[10]: 20
```

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

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 1: Source: thepythonguru.com

Functions

```
In [11]: def f(x,y):
             z = (x**2+y**2)**.5
             return z
```

```
In [12]: f(3,4)
```

```
Out[12]: 5.0
```

A function can be seen as a variable

```
In [13]: g = lambda x,y: (x**2+y**2)**.5
```

```
In [14]: g(3,4)
```

```
Out[14]: 5.0
```

A function does not even need a name

```
In [15]: (lambda x,y: (x**2+y**2)**.5)(3,4)
```

```
Out[15]: 5.0
```

Dictionaries

Create a data point (e.g. a fruit)

```
In [16]: x = {
             'color':'green',
             'size':'medium',
         }
```

Analyze this data point

```
In [17]: x['color']
```

```
Out[17]: 'green'
```

Classifiying Fruits: Conditional Expressions

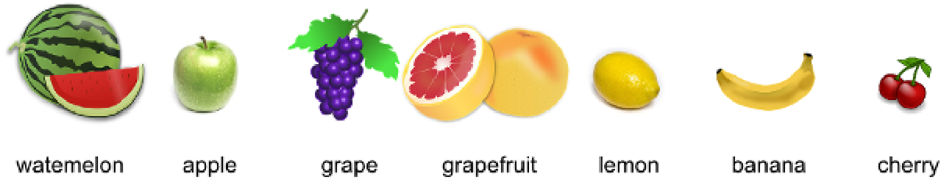

Figure 2:

A decision tree for watermelon vs. apple vs. other

In [18]:
```
def classify(x):
    decision = ''
    if x['color'] == 'green':
        if x['size'] == 'big':
            decision = 'watermelon'
        elif x['size'] == 'medium':
            decision = 'apple'
        else:
            decision = 'other'
    else:
        decision = 'other'
    return decision
```

In [19]:
```
classify({'color':'green','size':'big'})
```

Out[19]:
```
'watermelon'
```

In [20]:
```
classify({'color':'green','size':'medium'})
```

Out[20]:
```
'apple'
```

In [21]:
```
classify({'color':'red','size':'small'})
```

Out[21]:
```
'other'
```

Iterators

Making predictions for multiple observations

In [22]:
```
data = [
  {'color':'green','size':'big'},
  {'color':'yellow','shape':'round','size':'big'},
  {'color':'red','size':'medium'},
  {'color':'green','size':'big'},
  {'color':'red','size':'small','taste':'sour'},
  {'color':'green','size':'small'}
]
```

In [23]:
```
results = []
for x in data:
    results = results + [classify(x)]
print(results)
```

```
['watermelon', 'other', 'other', 'watermelon', 'other', 'other']
```

The same can be achieved with list comprehensions:

```
In [24]: print([classify(x) for x in data])
```

```
['watermelon', 'other', 'other', 'watermelon', 'other', 'other']
```

The same can also be achieved with the map function:

```
In [25]: print(map(classify,data))
```

```
['watermelon', 'other', 'other', 'watermelon', 'other', 'other']
```

Counting the number of objects "watermelon" in the data

```
In [26]: result = map(classify,data)

         count = 0
         for r in result:
             if r == 'watermelon':
                 count = count + 1
         count
```

```
Out[26]: 2
```

Or similarly

```
In [27]: sum([classify(x)=='watermelon' for x in data])
```

```
Out[27]: 2
```

Or similarly

```
In [28]: len(filter(lambda x: classify(x)=='watermelon', data))
```

```
Out[28]: 2
```

Or similarly

```
In [29]: reduce(lambda C,x: C+1 if classify(x) == 'watermelon' else C,data,0)
```

```
Out[29]: 2
```

Reading Data from a File

Content of file scores.txt that lists the performance of 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 [30]: f = open('scores.txt','r')
         L = []
         for line in f:
             L = L + map(float,str.split(line[:-1],','))
         print(L)
```

```
[80.0, 55.0, 16.0, 26.0, 37.0, 62.0, 49.0, 13.0, 28.0, 56.0, 43.0, 45.0, 47.0, 63.0, 43.0, 65.0, 10.0,
```

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

```
In [31]: D = sum([map(float,str.split(line[:-1],','))
                     for line in open('scores.txt','r')],[])
```

Classes

Let's separate our data into training and test data

```
In [32]: Dtrain = D[:20]
         Dtest  = D[20:]
```

Classes are useful for modeling anything that has an internal state, for example, machine learning models. The model below classifies whether a score is above/below the average.

```
In [33]: class Classifier:

             def train(self,X):
                 self.avg = sum(X)/len(X)

             def predict(self,X):
                 return ['above' if x > self.avg else 'below'  for x in X]
```

Build the classifier:

```
In [34]: c = Classifier()
```

Train the classifier and inspect what the classifier has learned:

```
In [35]: c.train(Dtrain)
         print(c.avg)
```

```
41.9
```

Apply the model to the test data verifies that it works correctly:

```
In [36]: Ytest = c.predict(Dtest)
         zip(Dtest[:5],Ytest[:5])
```

```
Out[36]: [(63.0, 'above'),
          (71.0, 'above'),
          (69.0, 'above'),
          (24.0, 'below'),
          (54.0, 'above')]
```