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]: 4.0 / 3.0, type(4.0), type(3.0), type(4.0 / 3.0)
```

```
Out[3]: (1.3333333333333333, float, float, float)
```

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

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

```
In [5]: int(4.0) / int(3.0), int(4.0 / 3.0)
```

```
Out[5]: (1.3333333333333333, 1)
```

```
In [6]: type(False), type([1, 2, 3]), type((1, 2, 3)), type('hello world')
```

```
Out[6]: (bool, list, tuple, str)
```

```
In [7]: int(True), int(False)
```

```
Out[7]: (1, 0)
```

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

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

```
Out[8]: 3
```

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

```
Out[9]: [1, 2, 3, 2, 3, 4]
```

Booleans

```
In [10]: a = True
         not a
```

```
Out[10]: False
```

```
In [11]: True or False, True and False
```

```
Out[11]: (True, False)
```

```
In [12]: 2 == 2, 2 == 4, 2 != 4, 2 is not 4
```

```
Out[12]: (True, False, True, True)
```

```
In [13]: "hello" is "world", "hello" is "hello"
```

```
Out[13]: (False, True)
```

Lists

```
In [14]: # Basic indexing
         l = [4, 2, 1, 5, 3]
         print(l[1])
```

```
2
```

```
In [15]: # Slicing
         print(l[1:3], l[:2], l[2:])
```

```
[2, 1] [4, 2] [1, 5, 3]
```

```
In [16]: # Negative indices
         print(l[-2])
         print(l[:-1])
```

```
5
[4, 2, 1, 5]
```

```
In [17]: # Repetition
         3 * [1, 2]
```

```
Out[17]: [1, 2, 1, 2, 1, 2]
```

```
In [18]: # Number of elements
         print(len(l))
```

```
5
```

```
In [19]: # Different datatypes
         ["Hello world", True, 4]
```

```
Out[19]: ['Hello world', True, 4]
```

Strings

```
In [20]: # Concatenation
         "hello" + " " + "world"
```

```
Out[20]: 'hello world'
```

```
In [21]: # Repetition
         3 * "Python"
```

```
Out[21]: 'PythonPythonPython'
```

```
In [22]: # String formatting
         "Today is {}, {}th of {}".format("Monday", 16, "April")
```

```
Out[22]: 'Today is Monday, 16th of April'
```

```
In [23]: print("{:.2f}".format(4/3))
         print("{:04d}".format(15))
```

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

Source: thepythonguru.com

```
1.33
0015
```

In [24]:
```
# Number of characters
len("Python")
```

Out[24]: 6

In [25]:
```
# Contains substring
"ell" in "hello"
```

Out[25]: True

In [26]:
```
# Indexing
s = "Hello world"
s[4], s[:5]
```

Out[26]: ('o', 'Hello')

Precedence of operators

In [27]:
```
1 * 2 + 3 * 4
```

Out[27]: 14

In [28]:
```
1 * (2 + 3) * 4
```

Out[28]: 20

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

Functions

```
In [29]: def f(x, y):
             z = (x ** 2 + y ** 2) ** 0.5
             return z

In [30]: f(3, 4)

Out[30]: 5.0
```

A function can be seen as a variable

```
In [31]: g = lambda x, y: (x ** 2 + y ** 2) ** 0.5

In [32]: g(3, 4)

Out[32]: 5.0

In [33]: # Reassign function to variable
         my_function = g
         my_function(3, 4)

Out[33]: 5.0
```

A function does not even need a name

```
In [34]: (lambda x, y: (x ** 2 + y ** 2) ** 0.5)(3, 4)

Out[34]: 5.0
```

Dictionaries

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

```
In [35]: x = {
             'color': 'green',
             'size': 'medium'
         }

In [36]: type(x)

Out[36]: dict
```

Analyze this data point

```
In [37]: x['color']

Out[37]: 'green'
```

Classifiying Fruits: Conditional Expressions

A decision tree for watermelon vs. apple vs. other

```
In [38]: def classify(x):
             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 [39]: x_new = {'color': 'green', 'size': 'big'}
         classify(x_new)

Out[39]: 'watermelon'

In [40]: classify({'color': 'green', 'size': 'medium'})

Out[40]: 'apple'

In [41]: classify({'color': 'red', 'size': 'small'})

Out[41]: 'other'

In [42]: # Ternary operator
         def compare(x, y):
             return "same" if x == y else "different"
         print(compare(1, 2))
         print(compare(1, 1))

different
same
```

Iterators

Making predictions for multiple observations

```
In [43]: for i in range(5):
             print(i)

0
1
2
3
4

In [44]: for i in [2, 1, 4]:
             print(i)

2
1
4

In [45]: 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'}
    ]
    type(data), type(data[0])
```

Out[45]: (list, dict)

In [46]:
```
results = list()
for x in data:
    results.append(classify(x))
print(results)
```

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

The same can be achieved with list comprehensions:

In [47]:
```
print([classify(x) for x in data])
```

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

This can also be combined with conditions:

In [48]:
```
print([classify(x) for x in data if x['color'] == 'green'])
```

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

Counting the number of objects "watermelon" in the data

In [49]:
```
result = [classify(x) for x in data]

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

```
2
```

Or in the "pythonic" way using list comprehension:

In [50]:
```
sum([classify(x) == 'watermelon' for x in data])
```

Out[50]: 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
`with` statement takes care of opening and closing the file.

```
In [52]: with open('scores.txt', 'r') as f:
             D = list()
             for line in f:
                 D.extend([float(x) for x in str.split(line[:-1], ',')])
         print(D)
```

```
[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, 5
```

Writing results back to a file:

```
In [54]: import os
         try:
             # Make sure not to overwrite an existing file
             outfile = 'scores_new.txt'
             if os.path.exists(outfile):
                 raise Exception("File '{}' already exists.".format(outfile))

             with open(outfile, 'w') as f:
                 f.write(str(D))
         except Exception as e:
             print("Exception occured: {}".format(e))
```

```
Exception occured: File 'scores_new.txt' already exists.
```

Classes

Let's separate our data into training and test data

```
In [55]: 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 [56]: 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 [57]: c = Classifier()
```

Train the classifier and inspect what the classifier has learned:

```
In [58]: c.train(Dtrain)
         print(c.avg)
```

```
41.9
```

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

In [59]:
```
Ytest = c.predict(Dtest)
list(zip(Dtest[:5], Ytest[:5]))
```

Out[59]:
```
[(63.0, 'above'),
 (71.0, 'above'),
 (69.0, 'above'),
 (24.0, 'below'),
 (54.0, 'above')]
```

In [ ]: