

# AI Project #2: Python Tutorial

Due Date:

**February 12, 2018 23:59 hours**

The projects for this class assume you use Python 2.7.

- A mini-UNIX tutorial,
- A mini-Python tutorial

**Files to Edit and Submit:** You will create `listcomp3.py` and fill in portions of `totalCost.py`, for this assignment. These 2 files should be submitted via Canvas.

The ideal environment in my opinion to do many of these projects is a Virtualbox/VMware Linux environment. I personally recommend Debian-based distros: Linux Mint, Arch, Ubuntu, openSUSE, or Debian itself. I personally use Linux Mint 18.3 on my laptop and Ubuntu 16.04 as a web server.

This particular project works fine on a Windows-based machine, but future projects require multiple additional libraries which are difficult to set up in Windows but simple to set up in Linux.

Perform the following steps:

Install Python 2.7. There are many valid ways to do this. Simplest way is probably:

```
sudo apt-get install python2.7
```

## Windows/other environments: download and install python 2.7 directly

Some of the later projects will require additional libraries. If you wish to install now, run:

```
sudo apt-get install python-minimal python-numpy python-scipy python-dev  
python-pip python-nose g++ libopenblas-dev git
```

Download `python_tutor.zip` from Canvas and unzip it.

```
$ unzip python_basics.zip  
$ cd python_basics  
$ ls  
foreach.py  
helloWorld.py  
listcomp.py  
listcomp2.py  
quickSort.py  
shop.py  
shopTest.py
```

Useful Linux commands:

- `cp` copies a file or files
- `rm` removes (deletes) a file
- `mv` moves a file (i.e., cut/paste instead of copy/paste)
- `man` displays documentation for a command
- `pwd` prints your current path
- Press "Ctrl-c" to kill a running process
- Press "Ctrl-d" to exit from Python 2.7

## The Emacs text editor

To run Emacs, type `emacs` at a command prompt:

```
$ emacs helloWorld.py
```

Here we gave the argument `helloWorld.py` which will either open that file for editing if it exists, or create it otherwise. Emacs notices that this is a Python source file (because of the `.py` ending) and enters Python-mode, which is supposed to help you write code. When editing this file you may notice some of that text becomes automatically colored: this is syntax highlighting to help you distinguish items such as keywords, variables, strings, and comments. Pressing Enter, Tab, or Backspace may cause the cursor to jump to weird locations: this is because Python is very picky about indentation, and Emacs is predicting the proper tabbing that you should use.

Some basic Emacs editing commands (`C-` means "while holding the Ctrl-key"):

- `C-x C-s` Save the current file
- `C-x C-f` Open a file, or create a new file if it doesn't exist
- `C-<space>` Start a mark for cutting and pasting
- `C-w` Cut the selected text
- `C-y` Paste the contents of the clipboard
- `C-_` Undo
- `C-g` Abort a half-entered command

You can also copy and paste using just the mouse. Using the left button, select a region of text to copy. Click the middle button to paste.

Certain Linux distros include a GUI version of Emacs with additional mouse controls. Terminal versions of emacs often include menus which you can activate with the F10 key.

There are two ways you can use Emacs to develop Python code. The most straightforward way is to use it just as a text editor: create and edit Python files in Emacs; then run Python to test the code somewhere else, like in a terminal window. Alternatively, you can run Python inside Emacs: see the options under "Python" in the menubar, or type `C-c !` to start a Python interpreter in a split screen. (Use `C-x o` to switch between the split screens, or just click if `C-x` doesn't work).

Other CLI or GUI text editors are equally valid. You may also wish to set up an IDE with python support.

---

## Python Basics

### Table of Contents

- [Invoking the Interpreter](#)
- [Operators](#)
- [Strings](#)
- [Dir and Help](#)
- [Built-in Data Structures](#)
  - [Lists](#)
  - [Tuples](#)
  - [Sets](#)
  - [Dictionaries](#)
- [Writing Scripts](#)
- [Indentation](#)
- [Tabs vs Spaces](#)
- [Writing Functions](#)
- [Object Basics](#)
  - [Defining Classes](#)
  - [Using Objects](#)
  - [Static vs Instance Variables](#)
- [Tips and Tricks](#)
- [Troubleshooting](#)
- [More References](#)

The programming assignments in this course will be written in [Python](#), an interpreted, object-oriented language that shares some features with both Java and Scheme. This tutorial will walk through the primary syntactic constructions in Python, using short examples.

We encourage you to type all python shown in the tutorial onto your own machine. Make sure it responds the same way.

You may find the [Troubleshooting](#) section helpful if you run into problems. It contains a list of the frequent problems students have encountered when following this tutorial.

### Invoking the Interpreter

Python can be run in one of two modes. It can either be used *interactively*, via an interpreter, or it can be called from the command line to execute a *script*. We will first use the Python interpreter interactively.

You invoke the interpreter by entering `python` at the Unix command prompt.  
Note: you may have to type `python2.7`, rather than `python`, depending on your machine.

```
$ python
Python 2.7.12 (default, Dec 4 2017, 14:50:18)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

## Operators

The Python interpreter can be used to evaluate expressions, for example simple arithmetic expressions. If you enter such expressions at the prompt (`>>>`) they will be evaluated and the result will be returned on the next line.

```
>>> 1 + 1
2
>>> 2 * 3
6
```

Boolean operators also exist in Python to manipulate the primitive `True` and `False` values.

```
>>> 1==0
False
>>> not (1==0)
True
>>> (2==2) and (2==3)
False
>>> (2==2) or (2==3)
True
```

## Strings

Like Java, Python has a built in string type. The `+` operator is overloaded to do string concatenation on string values. Notice that we can use either single quotes `' '` or double quotes `" "` to surround string. This allows easier visualization when you are nesting strings.

```
>>> 'artificial' + "intelligence"
'artificialintelligence'
```

There are many built-in methods which allow you to manipulate strings.

```
>>> 'artificial'.upper()
'ARTIFICIAL'
>>> 'HELP'.lower()
'help'
>>> len('Help')
4
```

We can also store expressions into variables.

```

>>> s = 'hello world'
>>> print s
hello world
>>> s.upper()
'HELLO WORLD'
>>> len(s.upper())
11
>>> num = 8.0
>>> num += 2.5
>>> print num
10.5

```

In Python, you do not have declare variables before you assign to them.

### Exercise: Dir and Help

Learn about the methods Python provides for strings. To see what methods Python provides for a datatype, use the `dir` and `help` commands:

```

>>> s = 'abc'

>>> dir(s)
['_add_', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__',
'__ge__', '__getattr__', '__getitem__', '__getnewargs__',
'__getslice__', '__gt__', '__hash__', '__init__', '__le__', '__len__',
'__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__str__',
'capitalize', 'center', 'count', 'decode', 'encode', 'endswith',
'expandtabs', 'find', 'index', 'isalnum', 'isalpha', 'isdigit', 'islower',
'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip',
'replace', 'rfind', 'rindex', 'rjust', 'rsplit', 'rstrip', 'split',
'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate',
'upper', 'zfill']

>>> help(s.find)

Help on built-in function find:

find(...)
    S.find(sub [,start [,end]]) -> int

    Return the lowest index in S where substring sub is found,
    such that sub is contained within s[start,end]. Optional
    arguments start and end are interpreted as in slice notation.

    Return -1 on failure.

>> s.find('b')
1

```

Try out some of the string functions listed in `dir` (ignore those with underscores '\_' around the method name).

## Built-in Data Structures

Python comes equipped with some useful built-in data structures, broadly similar to Java's collections package.

### Lists

*Lists* store a sequence of mutable items:

```
>>> fruits = ['apple', 'orange', 'pear', 'banana']
>>> fruits[0]
'apple'
```

We can use the + operator to do list concatenation:

```
>>> otherFruits = ['kiwi', 'strawberry']
>>> fruits + otherFruits
>>> ['apple', 'orange', 'pear', 'banana', 'kiwi', 'strawberry']
```

Python also allows negative-indexing from the back of the list. For instance, `fruits[-1]` will access the last element 'banana':

```
>>> fruits[-2]
'pear'
>>> fruits.pop()
'banana'
>>> fruits
['apple', 'orange', 'pear']
>>> fruits.append('grapefruit')
>>> fruits
['apple', 'orange', 'pear', 'grapefruit']
>>> fruits[-1] = 'pineapple'
>>> fruits
['apple', 'orange', 'pear', 'pineapple']
```

We can also index multiple adjacent elements using the slice operator. For instance, `fruits[1:3]`, returns a list containing the elements at position 1 and 2. In general `fruits[start:stop]` will get the elements in `start`, `start+1`, ..., `stop-1`. We can also do `fruits[start:]` which returns all elements starting from the `start` index. Also `fruits[:end]` will return all elements before the element at position `end`:

```
>>> fruits[0:2]
['apple', 'orange']
>>> fruits[:3]
['apple', 'orange', 'pear']
>>> fruits[2:]
['pear', 'pineapple']
>>> len(fruits)
4
```

The items stored in lists can be any Python data type. So for instance we can have lists of lists:

```
>>> lstOfLsts = [['a','b','c'],[1,2,3],['one','two','three']]
>>> lstOfLsts[1][2]
3
>>> lstOfLsts[0].pop()
'c'
>>> lstOfLsts
[['a', 'b'],[1, 2, 3],['one', 'two', 'three']]
```

## Exercise: Lists

Play with some of the list functions. You can find the methods you can call on an object via the `dir` and get information about them via the `help` command:

```
>>> dir(list)
['_add_', '__class__', '__contains__', '__delattr__', '__delitem__',
 '__delslice__', '__doc__', '__eq__', '__ge__', '__getattribute__',
 '__getitem__', '__getslice__', '__gt__', '__hash__', '__iadd__', '__imul__',
 '__init__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__',
 '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__',
 '__rmul__', '__setattr__', '__setitem__', '__setslice__', '__str__',
 'append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse',
 'sort']
```

```
>>> help(list.reverse)
Help on built-in function reverse:
```

```
reverse(...)
    L.reverse() -- reverse *IN PLACE*
```

```
>>> lst = ['a','b','c']
>>> lst.reverse()
>>> ['c','b','a']
```

Note: Ignore functions with underscores "\_" around the names; these are private helper methods. Press 'q' to back out of a help screen.

## Tuples

A data structure similar to the list is the *tuple*, which is like a list except that it is immutable once it is created (i.e. you cannot change its content once created). Note that tuples are surrounded with parentheses while lists have square brackets.

```
>>> pair = (3,5)
>>> pair[0]
3
>>> x,y = pair
>>> x
3
>>> y
5
>>> pair[1] = 6
TypeError: object does not support item assignment
```

The attempt to modify an immutable structure raised an exception. Exceptions indicate errors: index out of bounds errors, type errors, and so on will all report exceptions in this way.

## Sets

A *set* is another data structure that serves as an unordered list with no duplicate items. Below, we show how to create a set, add things to the set, test if an item is in the set, and perform common set operations (difference, intersection, union):

```
>>> shapes = ['circle', 'square', 'triangle', 'circle']
>>> setOfShapes = set(shapes)
>>> setOfShapes
set(['circle', 'square', 'triangle'])
>>> setOfShapes.add('polygon')
>>> setOfShapes
set(['circle', 'square', 'triangle', 'polygon'])
>>> 'circle' in setOfShapes
True
>>> 'rhombus' in setOfShapes
False
>>> favoriteShapes = ['circle', 'triangle', 'hexagon']
>>> setOfFavoriteShapes = set(favoriteShapes)
>>> setOfShapes - setOfFavoriteShapes
set(['square', 'polygon'])
>>> setOfShapes & setOfFavoriteShapes
set(['circle', 'triangle'])
>>> setOfShapes | setOfFavoriteShapes
set(['circle', 'square', 'triangle', 'polygon', 'hexagon'])
```

**Note that the objects in the set are unordered; you cannot assume that their traversal or print order will be the same across machines!**

## Dictionaries

The last built-in data structure is the *dictionary* which stores a map from one type of object (the key) to another (the value). The key must be an immutable type (string, number, or tuple). The value can be any Python data type.

Note: In the example below, the printed order of the keys returned by Python could be different than shown below. The reason is that unlike lists which have a fixed ordering, a dictionary is simply a hash table for which there is no fixed ordering of the keys (like HashMaps in Java). The order of the keys depends on how exactly the hashing algorithm maps keys to buckets, and will usually seem arbitrary. Your code should not rely on key ordering, and you should not be surprised if even a small modification to how your code uses a dictionary results in a new key ordering.

```
>>> studentIds = {'knuth': 42.0, 'turing': 56.0, 'nash': 92.0 }
>>> studentIds['turing']
56.0
>>> studentIds['nash'] = 'ninety-two'
>>> studentIds
```



```

{'knuth': 42.0, 'turing': 56.0, 'nash': 'ninety-two'}
>>> del studentIds['knuth']
>>> studentIds
{'turing': 56.0, 'nash': 'ninety-two'}
>>> studentIds['knuth'] = [42.0, 'forty-two']
>>> studentIds
{'knuth': [42.0, 'forty-two'], 'turing': 56.0, 'nash': 'ninety-two'}
>>> studentIds.keys()
['knuth', 'turing', 'nash']
>>> studentIds.values()
[[42.0, 'forty-two'], 56.0, 'ninety-two']
>>> studentIds.items()
[('knuth', [42.0, 'forty-two']), ('turing', 56.0), ('nash', 'ninety-two')]
>>> len(studentIds)
3

```

As with nested lists, you can also create dictionaries of dictionaries.

## Exercise: Dictionaries

Use `dir` and `help` to learn about the functions you can call on dictionaries.

## Writing Scripts

Now that you've got a handle on using Python interactively, let's write a simple Python script that demonstrates Python's `for` loop. Open the file called `foreach.py` and update it with the following code:

```

# This is what a comment looks like
fruits = ['apples', 'oranges', 'pears', 'bananas']
for fruit in fruits:
    print fruit + ' for sale'

fruitPrices = {'apples': 2.00, 'oranges': 1.50, 'pears': 1.75}
for fruit, price in fruitPrices.items():
    if price < 2.00:
        print '%s cost %f a pound' % (fruit, price)
    else:
        print fruit + ' are too expensive!'

```

At the command line, use the following command in the directory containing `foreach.py`:

```

$ python foreach.py
apples for sale
oranges for sale
pears for sale
bananas for sale
oranges cost 1.500000 a pound
pears cost 1.750000 a pound
apples are too expensive!

```

Remember that the print statements listing the costs may be in a different order on your screen than in this tutorial; that's due to the fact that we're looping over dictionary keys, which are unordered. To learn more about control structures (e.g., `if` and `else`) in Python, check out the official [Python tutorial section on this topic](http://docs.python.org/tut/). <http://docs.python.org/tut/>

The next snippet of code demonstrates Python's *list comprehension* construction:

```
nums = [1,2,3,4,5,6]
plusOneNums = [x+1 for x in nums]
oddNums = [x for x in nums if x % 2 == 1]
print oddNums
oddNumsPlusOne = [x+1 for x in nums if x % 2 ==1]
print oddNumsPlusOne
```

This code is in a file called `listcomp.py`, which you can run:

```
$ python listcomp.py
[1, 3, 5]
[2, 4, 6]
```

## Question 1: List Comprehension

Write a list comprehension `listcomp3.py` which from a list, swaps the case of every string and prints the element that ends with the letter `c`. You can find a similar example in `listcomp2.py`.

```
strings = ['Some string', 'Art', 'Music', 'Artificial Intelligence']
print ["Your code here"]
```

## Beware of Indentation!

Unlike many other languages, Python uses the indentation in the source code for interpretation. So for instance, for the following script:

```
if 0 == 1:
    print 'We are in a world of arithmetic pain'
print 'Thank you for playing'
```

will output

```
Thank you for playing
```

But if we had written the script as

```
if 0 == 1:
    print 'We are in a world of arithmetic pain'
    print 'Thank you for playing'
```

there would be no output. The moral of the story: be careful how you indent! It's best to use four spaces for indentation -- that's what the course code uses.

## Tabs vs Spaces

Because Python uses indentation for code evaluation, it needs to keep track of the level of indentation across code blocks. This means that if your Python file switches from using tabs as indentation to spaces as indentation, the Python interpreter will not be able to resolve the ambiguity of the indentation level and throw an exception. Even though the code can be lined up visually in your text editor, Python "sees" a change in indentation and most likely will throw an exception (or rarely, produce unexpected behavior).

This most commonly happens when opening up a Python file that uses an indentation scheme that is opposite from what your text editor uses (aka, your text editor uses spaces and the file uses tabs). When you write new lines in a code block, there will be a mix of tabs and spaces, even though the whitespace is aligned.

## Writing Functions

As in Java, in Python you can define your own functions:

```
fruitPrices = {'apples':2.00, 'oranges': 1.50, 'pears': 1.75}

def buyFruit(fruit, numPounds):
    if fruit not in fruitPrices:
        print "Sorry we don't have %s" % (fruit)
    else:
        cost = fruitPrices[fruit] * numPounds
        print "That'll be %f please" % (cost)

# Main Function
if __name__ == '__main__':
    buyFruit('apples',2.4)
    buyFruit('coconuts',2)
```

Rather than having a `main` function as in Java, the `__name__ == '__main__'` check is used to delimit expressions which are executed when the file is called as a script from the command line. The code after the main check is thus the same sort of code you would put in a `main` function in Java.

Save this script as *fruit.py* and run it:

```
$ python fruit.py
That'll be 4.800000 please
Sorry we don't have coconuts
```

## Example

`quicksort.py` defines and runs an example of a quicksort function in Python using list comprehensions. It uses the first element as the pivot.

## Question 2: totalCost function

Add a `totalCost(orderList)` function to `totalCost.py` which takes a list of (fruit,pound) tuples and returns the total cost of your list. Please do not change the `fruitPrices` variable.

The built-in test case should output:

```
Cost of [('apples', 2.0), ('pears', 3.0), ('limes', 4.0)] is 12.25
```

```
totalCost = """ Your code here """
```

## Object Basics

Although this isn't a class in object-oriented programming, you may run into some objects in programming projects, and so it's worth covering the basics of objects in Python. An object encapsulates data and provides functions for interacting with that data.

## Defining Classes

Here's an example of defining a class named `FruitShop`:

```
class FruitShop:

    def __init__(self, name, fruitPrices):
        """
            name: Name of the fruit shop

            fruitPrices: Dictionary with keys as fruit
            strings and prices for values e.g.
            {'apples':2.00, 'oranges': 1.50, 'pears': 1.75}
        """
        self.fruitPrices = fruitPrices
        self.name = name
        print 'Welcome to the %s fruit shop' % (name)

    def getCostPerPound(self, fruit):
        """
            fruit: Fruit string
            Returns cost of 'fruit', assuming 'fruit'
            is in our inventory or None otherwise
        """
        if fruit not in self.fruitPrices:
            print "Sorry we don't have %s" % (fruit)
            return None
        return self.fruitPrices[fruit]

    def getPriceOfOrder(self, orderList):
        """
            orderList: List of (fruit, numPounds) tuples

            Returns cost of orderList. If any of the fruit are
        """
```

```

        totalCost = 0.0
        for fruit, numPounds in orderList:
            costPerPound = self.getCostPerPound(fruit)
            if costPerPound != None:
                totalCost += numPounds * costPerPound
        return totalCost

    def getName(self):
        return self.name

```

The `FruitShop` class has some data, the name of the shop and the prices per pound of some fruit, and it provides functions, or methods, on this data. What advantage is there to wrapping this data in a class?

1. Encapsulating the data prevents it from being altered or used inappropriately,
2. The abstraction that objects provide make it easier to write general-purpose code.

## Using Objects

So how do we make an object and use it? Make sure you have the `FruitShop` implementation in `shop.py`. We then import the code from this file (making it accessible to other scripts) using `import shop`, since `shop.py` is the name of the file. Then, we can create `FruitShop` objects as follows:

```

import shop

shopName = 'the Berkeley Bowl'
fruitPrices = {'apples': 1.00, 'oranges': 1.50, 'pears': 1.75}
berkeleyShop = shop.FruitShop(shopName, fruitPrices)
applePrice = berkeleyShop.getCostPerPound('apples')
print applePrice
print('Apples cost $%.2f at %s.' % (applePrice, shopName))

otherName = 'the Stanford Mall'
otherFruitPrices = {'kiwis': 6.00, 'apples': 4.50, 'peaches': 8.75}
otherFruitShop = shop.FruitShop(otherName, otherFruitPrices)
otherPrice = otherFruitShop.getCostPerPound('apples')
print otherPrice
print('Apples cost $%.2f at %s.' % (otherPrice, otherName))
print("My, that's expensive!")

```

This code is in `shopTest.py`; you can run it like this:

```

$ python shopTest.py
Welcome to the Berkeley Bowl fruit shop
1.0
Apples cost $1.00 at the Berkeley Bowl.
Welcome to the Stanford Mall fruit shop
4.5
Apples cost $4.50 at the Stanford Mall.

```

My, that's expensive!

So what just happened? The `import shop` statement told Python to load all of the functions and classes in `shop.py`. The line `berkeleyShop = shop.FruitShop(shopName, fruitPrices)` constructs an *instance* of the `FruitShop` class defined in *shop.py*, by calling the `__init__` function in that class. Note that we only passed two arguments in, while `__init__` seems to take three arguments: `(self, name, fruitPrices)`. The reason for this is that all methods in a class have `self` as the first argument. The `self` variable's value is automatically set to the object itself; when calling a method, you only supply the remaining arguments. The `self` variable contains all the data (`name` and `fruitPrices`) for the current specific instance (similar to `this` in Java). The print statements use the substitution operator (described in the [Python docs](#) if you're curious).

## Static vs Instance Variables

The following example illustrates how to use static and instance variables in Python.

Create the `person_class.py` containing the following code:

```
class Person:
    population = 0
    def __init__(self, myAge):
        self.age = myAge
        Person.population += 1
    def get_population(self):
        return Person.population
    def get_age(self):
        return self.age
```

We first compile the script:

```
$ python person_class.py
```

Now use the class as follows:

```
>>> import person_class
>>> p1 = person_class.Person(12)
>>> p1.get_population()
1
>>> p2 = person_class.Person(63)
>>> p1.get_population()
2
>>> p2.get_population()
2
>>> p1.get_age()
12
>>> p2.get_age()
63
```

In the code above, `age` is an instance variable and `population` is a static variable. `population` is shared by all instances of the `Person` class whereas each instance has its own `age` variable.

## More Python Tips and Tricks

This tutorial has briefly touched on some major aspects of Python that will be relevant to the course. Here are some more useful tidbits:

- Use `range` to generate a sequence of integers, useful for generating traditional indexed for loops:
- ```
for index in range(3):
```
- ```
    print lst[index]
```
- After importing a file, if you edit a source file, the changes will not be immediately propagated in the interpreter. For this, use the `reload` command:

```
>>> reload(shop)
```

## Troubleshooting

These are some problems (and their solutions) that new Python learners commonly encounter.

- **Problem:**  
ImportError: No module named `py`

### Solution:

When using `import`, do not include the `".py"` from the filename.

For example, you should say: `import shop`

NOT: `import shop.py`

- **Problem:**  
NameError: name 'MY VARIABLE' is not defined  
Even after importing you may see this.

### Solution:

To access a member of a module, you have to type `MODULE NAME.MEMBER NAME`, where `MODULE NAME` is the name of the `.py` file, and `MEMBER NAME` is the name of the variable (or function) you are trying to access.

- **Problem:**  
TypeError: 'dict' object is not callable

### Solution:

Dictionary looks up are done using square brackets: `[ and ]`. NOT parenthesis: `( and )`.

- **Problem:**  
ValueError: too many values to unpack

**Solution:**

Make sure the number of variables you are assigning in a `for` loop matches the number of elements in each item of the list. Similarly for working with tuples.

For example, if `pair` is a tuple of two elements (e.g. `pair = ('apple', 2.0)`) then the following code would cause the "too many values to unpack error":

```
(a,b,c) = pair
```

Here is a problematic scenario involving a `for` loop:

```
pairList = [('apples', 2.00), ('oranges', 1.50), ('pears', 1.75)]
for fruit, price, color in pairList:
    print '%s fruit costs %f and is the color %s' % (fruit, price,
color)
```

- **Problem:**  
AttributeError: 'list' object has no attribute 'length' (or something similar)

**Solution:**

Finding length of lists is done using `len(NAME OF LIST)`.

- **Problem:**  
Changes to a file are not taking effect.

**Solution:**

1. Make sure you are saving all your files after any changes.
  2. If you are editing a file in a window different from the one you are using to execute python, make sure you `reload(YOUR_MODULE)` to guarantee your changes are being reflected. `reload` works similarly to `import`.
-