Project 0: Unix/Python Tutorial

Introduction

This tutorial will cover the basics of working in the Unix environment for the Berkeley instructional machines and a small Python tutorial. It assumes you have an EECS Instructional account for CS 188 and that you know how to access it.

You can download all of the files associated with this tutorial (including this description) as a zip archive.

Table of Contents

Submission

To get you familiarized with the automatic grading system, we will ask you to submit answers for problems 1 (`buyLotsOfFruit` function) and 2 (`shopSmart` function). This is a good thing: learning the basics of python now will save you many headaches later in the course.

This tutorial should be submitted with the name `p0` using these submission instructions.

Please read the submission instructions - they contain important information on how to submit this and all further assignments.

Unix Basics

Here are basic commands to navigate UNIX and edit files.

File/Directory Manipulation

When you open a terminal window, you're placed at a command prompt.

```
solar%
```

The prompt shows your username, the host you are logged onto, and your current location in the directory structure (your path). The tilde character is shorthand for your home directory. To make a directory, use the `mkdir` command. Use `cd` to change to that directory:

```
[cs188-ta@midway ~]$ mkdir tutorial
[cs188-ta@midway ~]$ cd tutorial
[cs188-ta@midway ~/tutorial]$
```

The Python files used in this tutorial reside in the `~cs188/projects/tutorial` directory. To copy them to your directory, use the `cp` command. The `*` is a useful way to specify multiple files in a given directory; `*.py` refers to all filenames that end have the `.py` ending. Note that `.` is shorthand for the current directory. Use `ls` to see a listing of the contents of a directory.

```
[cs188-ta@midway ~]$ cp ~cs188/projects/tutorial/*.py .
[cs188-ta@midway ~]$ ls
buyLotsOfFruit.py
foreach.py
listcomp.py
listcomp2.py
quickSort.py
shop.py
shopSmart.py
shopTest.py
```

Some other useful Unix commands:

- `rm` removes (deletes) a file
- `mv` moves a file (ie. cut/paste instead of copy/paste)
- `man` displays documentation for a command
- `pwd` prints your current path
- `xterm` opens a new terminal window
- `mozilla` opens a web browser
- Press "Ctrl-c" to kill a running process
- Append `&` to a command to run it in the background
- `fg` brings a program running in the background to the foreground

The Emacs text editor

Emacs is a customizable text editor which has some nice features specifically tailored for programmers. However, you can use any other text editor that you may prefer (such as `vi`, `pico`, or `joe` on Unix; or Notepad on Windows; or TextWrangler on Macs; and many more). To run Emacs, type `emacs` at a command prompt:

```
[cs188-ta@midway ~]$ emacs helloWorld.py &
[1] 3262
```

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 some text becomes automatically colored: this is syntactic 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 it if doesn't exist
- `C-k` Cut a line, add it to the clipboard
- `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.

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).

If you want to spend some extra set-up time becoming a power user, you can try an IDE like Eclipse (Download the Eclipse Classic package at the bottom). Check out PyDev for Python support in Eclipse. See our eclipse setup instructions for help.

Python Basics

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.

You may find the Troubleshooting section helpful if you run into problems. It contains a list of the frequent problems previous CS188 students have encountered when following this tutorial.

Invoking the Interpeter

Like Scheme, Python can be run in one of two modes. It can either be used *interactively*, via an interpeter, 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 `python2.4` or `python2.5`, rather than `python`, depending on your machine.

```
[cs188-ta@midway ~]$ python
Python 2.5 (r25:51908, Sep 28 2008, 12:45:36)
[GCC 3.4.6] on sunos5
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Operators

The Python interpeter 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 wil 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.

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

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

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

Notice that we can use either single quotes `' '` or double quotes `" "` to surround string. This allows for easy nesting of strings.

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: 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__', '__getattribute__', '__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]` which 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: 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.

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','polyon'])
>>> 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 (see the FAQ about dictionary 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: 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`:

```
[cs188-tf@solar ~/tutorial]$ 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.

If you like functional programming (like Scheme) you might also like `map` and `filter`:

```
>>> map(lambda x: x * x, [1,2,3])
[1, 4, 9]
>>> filter(lambda x: x > 3, [1,2,3,4,5,4,3,2,1])
```

```
[4, 5, 4]
```

You can learn more about `lambda` if you're interested. 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:

```
[cs188-ta@midway ~]$ python listcomp.py
[1,3,5]
[2,4,6]
```

Those of you familiar with Scheme, will recognize that the list comprehension is similar to the `map` function. In Scheme, the first list comprehension would be written as:

```
(define nums '(1,2,3,4,5,6))
(map
   (lambda (x) (+ x 1))  nums)
```

Exercise: Write a list comprehension which, from a list, generates a lowercased version of each string that has length greater than five. Solution: `listcomp2.py`

Beware of Indendation!

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.

Writing Functions

As in Scheme or 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:

```
[cs188-ta@midway ~]$ python fruit.py
That'll be 4.800000 please
Sorry we don't have coconuts
```

Problem 1 (for submission): Add a `buyLotsOfFruit(orderList)` function to `buyLotsOfFruit.py` which takes a list of `(fruit,pound)` tuples and returns the cost of your list. If there is some `fruit` in the list which doesn't appear in `fruitPrices` it should print an error message and return `None` (which is like `nil` in Scheme). Please do not change the `fruitPrices` variable.

*Test Case:*We will check your code by testing that the script correctly outputs

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

Advanced Exercise: Write a `quickSort` function in Python using list comprehensions. Use the first element as the pivot. Solution: `quickSort.py`

Object Basics

Although this isn't a class in object-oriented programming, you'll have to use some objects in the 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? Download 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!")
```

You can download this code in shopTest.py and run it like this:

```
[cs188-ta@midway ~]$ 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 happended? 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 with illustrate 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:
[cs188-ta@midway ~]$ 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.

Problem 2 (for submission): Fill in the function `shopSmart(orders,shops)` in `shopSmart.py`, which takes an `orderList` (like the kind passed in to `FruitShop.getPriceOfOrder`) and a list of `FruitShop` and returns the `FruitShop` where your order costs the least amount in total. Don't change the file name or variable names, please. Note that we will provide the `shop.py` implementation as a "support" file, so you don't need to submit yours.

*Test Case:*We will check that, with the following variable definitions:

```
orders1 = [('apples',1.0), ('oranges',3.0)]
orders2 = [('apples',3.0)]
dir1 = {'apples': 2.0, 'oranges':1.0}
shop1 =  shop.FruitShop('shop1',dir1)
dir2 = {'apples': 1.0, 'oranges': 5.0}
```

```
shop2 = shop.FruitShop('shop2',dir2)
shops = [shop1, shop2]
```

The following are true:

```
shopSmart.shopSmart(orders1, shops).getName() == 'shop1'
```

and

```
shopSmart.shopSmart(orders2, shops).getName() == 'shop2'
```

More Python Tips and Tricks

This tutorial has briefly touched on some major aspects of Python that will be relevant to the course. Here's 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 similar to `import`.

More References!

- The place to go for more Python information: www.python.org
- A good reference book: Learning Python (From the UCB campus, you can read the whole book online)