

Project 0: Unix & Python Tutorial

Deadline: 2021-05-17 23:59

May 10, 2021

Abstract

The goal of this project is to help you get familiar with Unix commands and Python grammar. The grade of this project will not count into your total grade. But failing to submit it before the deadline will cause a deduction.

Introduction

The projects for this class assume you use Python 3.6. Project 0 will cover the following:

- A mini-UNIX tutorial
- Instructions on how to set up the right Python version,
- A mini-Python tutorial.

Files to Edit and Submit: You will fill in portions of `addition.py`, `buyLotsOfFruit.py`, and `shopSmart.py` in `P0.zip` during the assignment. You should submit these files with your code and comments to online judge. Please *do not* change the other files in this distribution or submit any of our original files other than these files.

Evaluation: Your code will be autograded for technical correctness. Please *do not* change the names of any provided functions or classes within the code, or you will wreak havoc on the autograder. However, the correctness of your implementation – not the autograder’s judgements – will be the final judge of your score. If necessary, we will review and grade assignments individually to ensure that you receive due credit for your work.

Academic Dishonesty: We will be checking your code against other submissions in the class for logical redundancy. If you copy someone else's code and submit it with minor changes, we will know. These cheat detectors are quite hard to fool, so please don't try. We trust you all to submit your own work only; please don't let us down. If you do, we will pursue the strongest consequences available to us.

Getting Help: You are not alone! If you find yourself stuck on something, contact the course staff for help. Office hours and the discussion forum on Piazza are there for your support; please use them. If you can't make our office hours, let us know and we will schedule more. We want these projects to be rewarding and instructional, not frustrating and demoralizing. But, we don't know when or how to help unless you ask.

Part 1: 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:

```
ve492-ta@myPC:~$
```

The prompt generally 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. Note your prompt may look slightly different. To make a directory, use the `mkdir` command. Use `cd` to change to that directory:

```
ve492-ta@myPC:~$ mkdir foo
ve492-ta@myPC:~$ cd foo
ve492-ta@myPC:~/foo$
```

Use `ls` to see a listing of the contents of a directory, and `touch` to create an empty file:

```
ve492-ta@myPC:~$ ls
ve492-ta@myPC:~$ touch alloha
ve492-ta@myPC:~/foo$ ls
alloha
ve492-ta@myPC:~/foo$ cd ..
ve492-ta@myPC:~$
```

Download `python_basics.zip` from [Canvas](#) into your home directory (note: the zip file's name may be slightly different when you download it). Use `unzip` to extract the contents of the zip file:

```
ve492-ta@myPC:~$ ls *.zip
python_basics.zip
ve492-ta@myPC:~$ unzip python_basics.zip
ve492-ta@myPC:~/foo$ cd python_basics
ve492-ta@myPC:~/foo$ ls
foreach.py
helloWorld.py
listcomp.py
listcomp2.py
quickSort.py
shop.py
shopTest.py
```

Some other useful Unix 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
- Append the character `&` to a command to run it in the background
- `fg` brings a program running in the background to the foreground

Part 2: Python Installation

Many of you may not have Python 3.6 already installed on your computers. Conda is an easy way to manage many different environments, each with its own Python version and dependencies. This allows us to avoid conflicts between our preferred Python version and that of other classes. We'll walk through how to set up and use a conda environment.

Prerequisite: [Anaconda](#). If you haven't installed it, install it through the previous link.

Creating a Conda Environment The command for creating a conda environment with Python 3.6 is:

```
ve492-ta@myPC:~$ conda create --name <env-name> python=3.6
```

For us, we decide to name our environment **ve492**, so we run the following command, and press **y** to confirm installing any missing packages.

```
ve492-ta@myPC:~/python_basics$ conda create --name ve492 python=3.6
```

Entering the Environment To enter the conda environment that we just created, do the following. Note that the Python version within the environment is 3.6, just what we want. The tag (**<env-name>**) shows you the name of the conda environment that is active. In our case, we have (ve492), as what we'd expect.

```
ve492-ta@myPC:~/python_basics$ source activate ve492
(ve492) ve492-ta@myPC:~/python_basics$ python -V
Python 3.6.6 :: Anaconda, Inc.
```

Leaving the Environment Leaving the environment is just as easy.

```
(ve492) ve492-ta@myPC:~/python_basics$ source deactivate
ve492-ta@myPC:~/python_basics$ python -V
Python 3.7.4
```

Our python version has now returned to whatever the system default is!

Part3: Python Basics

Prerequisite: `python_basics.zip` from [Canvas](#).

The programming assignments in this course will be written in Python, an interpreted, object-oriented language that shares some features with both C++ and Matlab. This tutorial will walk through the primary syntactic constructions in Python, using short examples.

We encourage you to type all python code 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.

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 using the command `python` at the Unix command prompt; or if you are using Windows that doesn't work for you in Git Bash, using `python -i`.

```
(ve492) ve492-ta@myPC:~/python_basics$ python
Python 3.6.6 |Anaconda, Inc.| (default, Sept 3 2019, 23:11:49)
[GCC 4.2.1 Compatible Clang 4.0.1 (tags/RELEASE_401/final)] on darwin
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

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 to 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(...) method of builtins.str instance
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 C++ STL Containers.

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']
```

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:

```
>>> shapes = ['circle', 'square', 'triangle', 'circle']
>>> setOfShapes = set(shapes)
```

Another way of creating a set is shown below:

```
>>> setOfShapes = {'circle', 'square', 'triangle', 'circle'}
```

Next, we show how to add things to the set, test if an item is in the set, and perform common set operations (difference, intersection, union):

```
>>> 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 `unordered_map` in C++). 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`, which should contain 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`:

```
ve492-ta@myPC:~/tutorial$ python foreach.py
apples for sale
oranges for sale
pears for sale
bananas for sale
apples are too expensive!
oranges cost 1.500000 a pound
pears cost 1.750000 a pound
```

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 you might also like `map` and `filter`:

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

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:

```
ve492-ta@myPC:~/tutorial$ python listcomp.py
[1, 3, 5]
[2, 4, 6]
```

Exercise: List Comprehensions

Write a list comprehension which, from a list, generates a lowercased version of each string that has length greater than five. You can find the solution in `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.

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. For a longer discussion on tabs vs spaces, see [this](#) discussion on StackOverflow.

Writing Functions

As in C++, 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 C++.

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

```
ve492-ta@myPC:~/tutorial$ python fruit.py
That'll be 4.800000 please
Sorry we don't have coconuts
```

Advanced Exercise

Write a `quickSort` function in Python using list comprehensions. Use the first element as the pivot. You can find the solution in `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 %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:
            return None
        return self.fruitPrices[fruit]

    def getPriceOfOrder(self, orderList):
        """
```

```

        orderList: List of (fruit, numPounds) tuples

        Returns cost of orderList, only including the values of
        fruits that this fruit shop has.
        """
        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?

- Encapsulating the data prevents it from being altered or used inappropriately,
- 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:

```

ve492-ta@myPC:~/tutorial$ 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 C++). 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:

```
ve492-ta@myPC:~/tutorial$ 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(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: For import statements with `import <package-name>`, do not include the file extension (i.e. the `.py` string). For example, you should use: `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 look ups 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: Make sure you are saving all your files after any changes. 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`.

More References

- The place to go for more Python information: www.python.org
- A good reference book: Learning Python

Part 4: Question 1: Addition

Open `addition.py` and look at the definition of `add`:

```
def add(a, b):  
    "Return the sum of a and b"  
    "*** YOUR CODE HERE ***"  
    return 0
```

The tests called this with `a` and `b` set to different values, but the code always returned zero. Modify this definition.

Part 5: Question 2: buyLotsOfFruit function

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`. Please do not change the `fruitPrices` variable.

Part 6: Question 3: shopSmart function

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.

Submission

Submit it to online judge, see the announcement later.