Assignment 0

Introduction

This is an introductory assignment. Please submit your solution before the deadline published on Moodle. Assignment 0 will mainly cover the following:

- A basic command line tutorial
- A basic Python 3 tutorial
- Some tasks with an autograder that checks for technical correctness

Getting Help: You are not alone! If you find yourself stuck on something, please let us know in the forum. We want this assignment to be rewarding and instructional, not frustrating and demoralizing. But, we don't know when or how to help unless you ask.

Installation

We suggest that you install anaconda.

Command Line Basics

Here are basic commands that help you to navigate in Linux using the command line.

Note: This section assumes that you are using Linux. However, everything is platform independent and should run on most other operating systems. In Windows like operating systems, the equivalent command to `ls` is `dir`. The commands of `mkdir` and `cd` are the same. In Windows, we suggest that you use the anaconda powershell prompt to have access to unix style commands.

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

```
scdirk@scdirk-office:~$
```

The prompt shows your current location in the directory structure (your path). Note your prompt may look slightly different. To make a directory, use the `mkdir` command. Use `cd` to change to that directory and use `dir` to see a listing of the contents of a directory:

```
scdirk@scdirk-Home:~$ ls
anaconda3  Documents  examples.desktop  nas       Public  Steam      Videos
Desktop    Downloads  Music             Pictures  snap    Templates
scdirk@scdirk-Home:~$ mkdir comp7404
scdirk@scdirk-Home:~$ cd comp7404/
scdirk@scdirk-Home:~/comp7404$
```

Download the assignment file a0.zip into the newly created directory.

Unzip it using the `unzip` command and inspect the content of the `a0` folder.

```
scdirk@scdirk-Home:~/comp7404$ unzip -q a0.zip
scdirk@scdirk-Home:~/comp7404$ ls -l
total 56
drwxr-xr-x 3 scdirk scdirk  4096 Sep 1 10:02 a0
-rw-r--r-- 1 scdirk scdirk 45954 Sep 1 11:44 a0.zip
scdirk@scdirk-Home:~/comp7404$ cd a0
scdirk@scdirk-Home:~/comp7404/a0$ ls -l
total 32
-rwxr-xr-x 1 scdirk scdirk  330 Sep 1 10:01 foreach.py
-rwxr-xr-x 1 scdirk scdirk   72 Sep 1 10:01 helloWorld.py
-rwxr-xr-x 1 scdirk scdirk  117 Sep 1 10:02 listcomp2.py
-rwxr-xr-x 1 scdirk scdirk  151 Sep 1 10:02 listcomp.py
-rwxr-xr-x 1 scdirk scdirk  321 Sep 1 10:02 quickSort.py
-rwxr-xr-x 1 scdirk scdirk 1366 Sep 1 10:02 shop.py
-rwxr-xr-x 1 scdirk scdirk  564 Sep 1 10:02 shopTest.py
drwxr-xr-x 3 scdirk scdirk 4096 Sep 1 10:35 task
scdirk@scdirk-Home:~/comp7404/a0$
```

Text Editor

Gedit is a simple text editor to edit source files.

```
scdirk@scdirk-Home:~/comp7404/a0$ gedit helloWorld.py
```

This part is just a recommendation for a text editor. You can use any other text editor.

Python Basics

The programming assignments in this course will be written in Python, an interpreted language.

Assignment 0 will walk you through the primary syntactic constructions in Python, using short examples.

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

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 command prompt.

```
scdirk@scdirk-Home:~/comp7404/a0$ python
Python 3.8.8 (default, Apr 13 2021, 15:08:03) [MSC v.1916 64 bit (AMD64)]
:: Anaconda, Inc. on win32
Type "help", "copyright", "credits" or "license" for more information.
```

Note that you must use python 3 in all our assignments. You cannot use python 2.

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 does string concatenation on string values.

```
>>> 'pineapple'+"pen"
'pineapplepen'
```

Notice that we can use either single quotes ' ' or double quotes " " to surround the string. This allows for easy nesting of strings. There are many built-in methods which allow you to manipulate strings.

```
>>> 'Mangkhut'.upper()
'MANGKHUT'
>>> 'TYPHOON'.lower()
'typhoon'
>>> len()
>>> len('MANGKHUT')
8
```

We can 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 data type, use the `dir` and `help` commands.

```
>>> s = 'abs'
>>> dir(s)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mod__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'encode',
'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index',
'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'isidentifier', 'islower',
'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join',
'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind',
'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines',
'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
>>> help(s.upper)
Help on built-in function upper:

upper(...) method of builtins.str instance
    S.upper() -> str

    Return a copy of S converted to uppercase.
(END)
```

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.

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__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__',
'__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mul__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__',
'__str__', '__subclasshook__', 'append', 'clear', 'copy', 'count',
'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
Help on method_descriptor:

reverse(...)
    L.reverse() -- reverse *IN PLACE*
(END)
>>> lst = ['a','b','c']
>>> lst.reverse()
>>> lst
['c', 'b', 'a']
```

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
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' 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).

```
>>> setOfShapes = {'square', 'triangle', 'circle'}
>>> setOfShapes
{'circle', 'triangle', 'square'}
>>> setOfShapes.add('polygon')
>>> setOfShapes
{'circle', 'triangle', 'square', 'polygon'}
>>> 'circle' in setOfShapes
True
>>> 'rhombus' in setOfShapes
False
>>> setOfFavoriteShapes = {'circle','triangle','hexagon'}
>>> setOfShapes - setOfFavoriteShapes
{'polygon', 'square'}
>>> setOfShapes & setOfFavoriteShapes
{'circle', 'triangle'}
>>> setOfShapes | setOfFavoriteShapes
{'triangle', 'square', 'hexagon', 'circle', 'polygon'}
```

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. 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
{'turing': 56.0, 'nash': 'ninety-two', 'knuth': [42.0, 'forty-two']}
>>> studentIds.keys()
dict_keys(['turing', 'nash', 'knuth'])
>>> studentIds.values()
dict_values([56.0, 'ninety-two', [42.0, 'forty-two']])
>>> studentIds.items()
dict_items([('turing', 56.0), ('nash', 'ninety-two'), ('knuth', [42.0,
'forty-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 inspect the content.

```
>>> exit()
scdirk@scdirk-Home:~/comp7404/a0$ gedit foreach.py
```

```
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(fruit,'cost',str(price),'a pound')
    else:
        print(fruit,' are too expensive!')
```

At the command line, use the following command to run foreach.py.

```
scdirk@scdirk-Home:~/comp7404/a0$ python foreach.py
apples for sale
oranges for sale
pears for sale
bananas for sale
apples  are too expensive!
oranges cost 1.5 a pound
pears cost 1.75 a pound
```

Remember that the print statements listing the costs may be in a different order on your screen than in this document; because we are 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.

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

```
scdirk@scdirk-Home:~/comp7404/a0$ gedit listcomp.py
```

```
nums = [1,2,3,4,5,6]
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)
```

You can run this code as follows.

```
scdirk@scdirk-Home:~/comp7404/a0$ python listcomp.py
[1, 3, 5]
[2, 4, 6]
```

Exercise: List Comprehensions

Write a list comprehension which, from a list, generates a lowercase version of each string that has length greater than five. You can find the solution in `listcomp2.py`.

Beware of Indentation!

Unlike many other languages, Python uses 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.

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

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 " + fruit)
    else:
        cost = fruitPrices[fruit] * numPounds
        print("That'll be "+str(cost)+" please")

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

Rather than having a main function as in C++, 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.

```
scdirk@scdirk-Home:~/comp7404/a0$ python fruit.py
That'll be 4.8 please
Sorry we don't have coconuts
```

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 assignments, 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?

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 = 'HKU ParknShop'
fruitPrices = {'apples': 1.00, 'oranges': 1.50, 'pears': 1.75}
hkushop = shop.FruitShop(shopName, fruitPrices)
applePrice = hkushop.getCostPerPound('apples')
```

```
print('Apples cost $%.2f at %s.' % (applePrice, shopName))

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

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

```
scdirk@scdirk-Home:~/comp7404/a0$ python shopTest.py
Welcome to HKU ParknShop fruit shop
Apples cost $1.00 at HKU ParknShop.
Welcome to Wellcome Westwood fruit shop
Apples cost $4.50 at Wellcome Westwood.
Wow, 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 `hkushop = 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`

```
scdirk@scdirk-Home:~/comp7404/a0$ gedit 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
```

Now use the class as follows.

```
scdirk@scdirk-Home:~/comp7404/a0$ python
>>> 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
>>> exit()
```

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.

For loop

Use range to generate a sequence of integers, useful for generating traditional indexed `for` loop.

```
for index in range(3):
    print(lst[index])
```

More

This document has briefly touched on some major aspects of Python that will be relevant to the

course. For more, check out this Python book.

Autograding

All assignments in this course will be autograded after you submit your code through the moodle website. For all assignments you can submit as many times as you like until the deadline. Do not submit after the deadline or you will get 0 marks. Make sure that you submit the correct files. Submitted incorrect files will always result in 0 marks; no exceptions.

Some assignment's release includes its autograder for you to run yourself on your computer. This is the recommended, and fastest, way to test your code. We strongly suggest to use the autograder. If your code does not pass the autograder you will not receive any marks.

To get you familiarized with the autograder, we will ask you to code and test solutions for a few questions.

All of the files associated with assignment 0 are in the `task` directory. You can enter the `task` directory as follows.

```
scdirk@scdirk-Home:~/comp7404/a0$ cd task
scdirk@scdirk-Home:~/comp7404/a0/task$ ls
addition.py         LICENSE           testClasses.py
autograder.py       projectParams.py  testParser.py
average.py          shop.py           textDisplay.py
buyLotsOfFruit.py   shopSmart.py      tutorialTestClasses.py
grading.py          test_cases        util.py
```

Related Files

This directory contains a number of files you'll edit or run.

File	Description
`addition.py`	source file for question 1
`average.py`	source file for question 2

`buyLotsOfFruit.py`	source file for question 3
`shopSmart.py`	source file for questions 4, 5, 6
`shop.py`	source file for questions 4, 5, 6
`autograder.py`	autograding script (see below)

The following files / folders are for autograding purposes. You can safely ignore the following and other unmentioned files / folders.

- grading.py
- projectParams.py
- testClasses.py
- testParser.py
- tutorialTestClasses.py
- test_cases

Basic Usage

The command `python autograder.py` grades your solution. If we run it before editing any files we get the following output.

```
scdirk@scdirk-Home:~/comp7404/a0/task$ python autograder.py
Starting on 9-16 at 14:43:27

Question q1
===========
*** FAIL: test_cases/q1/addition1.test
***   add(a,b) must return the sum of a and b
***   student result: "0"
***   correct result: "2"
*** FAIL: test_cases/q1/addition2.test
***   add(a,b) must return the sum of a and b
***   student result: "0"
***   correct result: "5"
*** FAIL: test_cases/q1/addition3.test
***   add(a,b) must return the sum of a and b
***   student result: "0"
```

```
***   correct result: "7.9"
*** Tests failed.

### Question q1: 0/1 ###


Question q2
===========
*** FAIL: test_cases/q2/average1.test
***   average(priceList) must compute the average cost of UNIQUE prices in
priceList
***   student result: "None"
***   correct result: "1.0"
*** FAIL: test_cases/q2/average2.test
***   average(priceList) must compute the average cost of UNIQUE prices in
priceList. Make sure to not loose precision.
***   student result: "None"
***   correct result: "3.25"
*** FAIL: test_cases/q2/average3.test
***   average(priceList) must compute the average cost of UNIQUE prices in
priceList
***   student result: "None"
***   correct result: "4.5"
*** Tests failed.

### Question q2: 0/1 ###


Question q3
===========
*** FAIL: test_cases/q3/food_price1.test
***   buyLotsOfFruit must compute the correct cost of the order
***   student result: "0.0"
***   correct result: "12.25"
*** FAIL: test_cases/q3/food_price2.test
***   buyLotsOfFruit must compute the correct cost of the order
***   student result: "0.0"
***   correct result: "14.75"
*** FAIL: test_cases/q3/food_price3.test
***   buyLotsOfFruit must compute the correct cost of the order
***   student result: "0.0"
***   correct result: "6.4375"
```

```
*** Tests failed.

### Question q3: 0/1 ###


Question q4
===========
Welcome to shop1 fruit shop
Welcome to shop2 fruit shop
*** FAIL: test_cases/q4/select_shop1.test
***   shopSmart(order, shops) must select the cheapest shop
***   student result: "None"
***   correct result: "<FruitShop: shop1>"
Welcome to shop1 fruit shop
Welcome to shop2 fruit shop
*** FAIL: test_cases/q4/select_shop2.test
***   shopSmart(order, shops) must select the cheapest shop
***   student result: "None"
***   correct result: "<FruitShop: shop2>"
Welcome to shop1 fruit shop
Welcome to shop2 fruit shop
Welcome to shop3 fruit shop
*** FAIL: test_cases/q4/select_shop3.test
***   shopSmart(order, shops) must select the cheapest shop
***   student result: "None"
***   correct result: "<FruitShop: shop3>"
*** Tests failed.

### Question q4: 0/1 ###


Question q5
===========
Welcome to shop1 fruit shop
Welcome to shop2 fruit shop
*** FAIL: test_cases/q5/arbitrage_shop1.test
***   shopArbitrage(order, shops) must return the maximum arbitrage profit
***   student result: "None"
***   correct result: "13.0"
Welcome to shop1 fruit shop
Welcome to shop2 fruit shop
*** FAIL: test_cases/q5/arbitrage_shop2.test
```

```
***   shopArbitrage(order, shops) must return the maximum arbitrage profit
***   student result: "None"
***   correct result: "3.0"
Welcome to shop1 fruit shop
Welcome to shop2 fruit shop
Welcome to shop3 fruit shop
*** FAIL: test_cases/q5/arbitrage_shop3.test
***   shopArbitrage(order, shops) must return the maximum arbitrage profit
***   student result: "None"
***   correct result: "22.0"
*** Tests failed.

### Question q5: 0/1 ###


Question q6
===========
Welcome to shop1 fruit shop
Welcome to shop2 fruit shop
*** FAIL: test_cases/q6/minimum_cost1.test
***   shopMinimum(order, shops) must output the minimum cost
***   student result: "None"
***   correct result: "4.0"
Welcome to shop1 fruit shop
Welcome to shop2 fruit shop
*** FAIL: test_cases/q6/minimum_cost2.test
***   shopMinimum(order, shops) must output the minimum cost
***   student result: "None"
***   correct result: "3.0"
Welcome to shop1 fruit shop
Welcome to shop2 fruit shop
Welcome to shop3 fruit shop
*** FAIL: test_cases/q6/minimum_cost3.test
***   shopMinimum(order, shops) must output the minimum cost
***   student result: "None"
***   correct result: "13.0"
*** Tests failed.

### Question q6: 0/1 ###


Finished at 14:43:27
```

```
Provisional grades
==================
Question q1: 0/1
Question q2: 0/1
Question q3: 0/1
Question q4: 0/1
Question q5: 0/1
Question q6: 0/1
------------------
Total: 0/6
```

For each of the six questions, this shows the results of that question's tests, the marks, and a final summary at the end. Because you haven't yet solved the questions, all the tests fail. As you solve each question you may find some tests pass while others fail. When all tests pass for a question, you get full marks.

Looking at the results for question 1, you can see that it has failed three tests with the error message `add(a,b) must return the sum of a and b`. The answer your code gives is always 0, but the correct answer is different. We'll fix that now.

Question 1: Addition

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

```
scdirk@scdirk-Home:~/comp7404/a0/task$ gedit addition.py
```

```
"""
Run python autograder.py
"""

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 to read.

```
"""
```

```
Run python autograder.py
"""

def add(a, b):
    "Return the sum of a and b"
    print("Passed a=%s and b=%s, returning a+b=%s" % (a,b,a+b))
    return a+b
```

Now rerun the autograder (omitting the results for other questions).

```
Starting on 9-16 at 15:03:21

Question q1
===========
Passed a=1 and b=1, returning a+b=2
*** PASS: test_cases/q1/addition1.test
***   add(a,b) returns the sum of a and b
Passed a=2 and b=3, returning a+b=5
*** PASS: test_cases/q1/addition2.test
***   add(a,b) returns the sum of a and b
Passed a=10 and b=-2.1, returning a+b=7.9
*** PASS: test_cases/q1/addition3.test
***   add(a,b) returns the sum of a and b

### Question q1: 1/1 ###


Question q2
===========
...
### Question q6: 0/1 ###


Finished at 15:03:21

Provisional grades
==================
Question q1: 1/1
Question q2: 0/1
Question q3: 0/1
Question q4: 0/1
```

```
Question q5: 0/1
Question q6: 0/1
-----------------
Total: 1/6
```

You now pass all tests, getting full marks for question 1. Notice the new lines `Passed a=...` which appear before `*** PASS: ....` These are produced by the `print` statement in `add`. You can use `print` statements like that to output information useful for debugging. You can also run the autograder with the option `--mute` to temporarily hide such lines, as follows.

```
Starting on 9-16 at 15:05:42

Question q1
===========
*** PASS: test_cases/q1/addition1.test
***   add(a,b) returns the sum of a and b
*** PASS: test_cases/q1/addition2.test
***   add(a,b) returns the sum of a and b
*** PASS: test_cases/q1/addition3.test
***   add(a,b) returns the sum of a and b

### Question q1: 1/1 ###
...
```

Question 2: average Function

Implement the `average(priceList)` function in `average.py` which takes a list of prices and returns the average value of unique prices in the list.

Run `python autograder.py` until question 2 passes all tests and you get full marks. Each test will confirm that `average(priceList)` returns the correct answer given various possible inputs. For example, `test_cases/q2/average2.test` tests whether `average.average([1,1,3,3,4,5]) == 3.25`.

```
scdirk@scdirk-Home:~/comp7404/a0/task$ more test_cases/q2/average2.test
class: "EvalTest"
success: "average(priceList) correctly computes the average cost in
priceList"
failure: "average(priceList) must compute the average cost of UNIQUE prices
in priceList. Make sure to not lose precision."
```

```
# A python expression to be evaluated. This expression must return the same
result for the student and instructor's code.
test: "average.average([1,1,3,3,4,5])"
scdirk@scdirk-Home:~/comp7404/a0/task$ more test_cases/q2/average2.solution
# This is the solution file for test_cases/q2/average2.test.
# The result of evaluating the test must equal the below when cast to
a string.
result: "3.25"
```

Once you think that your implementation is correct you can then go ahead and run the autograder just for question 2 as follows.

```
scdirk@scdirk-Home:~/comp7404/a0/task$ python autograder.py -q q2
Starting on 9-16 at 15:54:05

Question q2
===========
*** PASS: test_cases/q2/average1.test
***   average(priceList) correctly computes the average cost in priceList
*** PASS: test_cases/q2/average2.test
***   average(priceList) correctly computes the average cost in priceList
*** PASS: test_cases/q2/average3.test
***   average(priceList) correctly computes the average cost in priceList

### Question q2: 1/1 ###


Finished at 15:54:05

Provisional grades
==================
Question q2: 1/1
------------------
Total: 1/1
```

Question 3: 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.

Run `python autograder.py` until question 3 passes all tests and you get full marks. Each test will confirm that `buyLotsOfFruit(orderList)` returns the correct answer given various possible inputs. For example, `test_cases/q3/food_price1.test` tests whether cost of `[('apples', 2.0), ('pears', 3.0), ('limes', 4.0)] == 12.25`.

When done, go ahead and run the autograder just for question 3 as follows.

```
scdirk@scdirk-Home:~/comp7404/a0/task$ python autograder.py -q q3
Starting on 9-16 at 17:13:05

Question q3
===========
[('apples', 2.0), ('pears', 3.0), ('limes', 4.0)]
*** PASS: test_cases/q3/food_price1.test
***   buyLotsOfFruit correctly computes the cost of the order
[('apples', 4.0), ('pears', 3.0), ('limes', 2.0)]
*** PASS: test_cases/q3/food_price2.test
***   buyLotsOfFruit correctly computes the cost of the order
[('apples', 1.25), ('pears', 1.5), ('limes', 1.75)]
*** PASS: test_cases/q3/food_price3.test
***   buyLotsOfFruit correctly computes the cost of the order

### Question q3: 1/1 ###


Finished at 17:13:05

Provisional grades
==================
Question q3: 1/1
------------------
Total: 1/1
```

Question 4: 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.

Run `python autograder.py` until question 4 passes all tests and you get full marks. Each test will confirm that `shopSmart(orders,shops)` returns the correct answer given various possible inputs.

For example, 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]
```

`test_cases/q4/select_shop1.test` tests whether `shopSmart.shopSmart(orders1, shops) == shop1` and `test_cases/q4/select_shop2.test` tests whether `shopSmart.shopSmart(orders2, shops) == shop2`.

Question 5: shopArbitrage Function

Shops may sell the same fruit at different prices. Let's assume you can buy a fruit at one shop and that you can sell it to another shop at the same price that it is offered there. For example, if `shop1` sells apples at $2 and shop2 at $3 you will be able to make $1 profit. Let's write a function to take advantage of this arbitrage opportunity. Implement the function `shopArbitrage(orders,shop)` in `shopSmart.py`, which takes an `orderList` (like the kind passed into `FruitShop.getPriceOfOrder`) and a list of `FruitShop`. Return the maximum profit where `orderList` reward you most in total. Don't change the file name or variable names.

Run `python autograder.py` until question 5 passes all tests and you get full marks. Each test will confirm that `shopArbitrage(orders,shops)` returns the correct answer given various possible inputs. For example, with the following variable definitions.

```
dir1 = {'apples': 2.0, 'oranges':1.0}
shop1 =  shop.FruitShop('shop1',dir1)
dir2 = {'apples': 1.0, 'oranges': 5.0}
shop2 = shop.FruitShop('shop2',dir2)
```

```
dir3 = {'apples': 1.5, 'oranges': 2.0}
shop3 = shop.FruitShop('shop3',dir3)
shops = [shop1, shop2, shop3]
order = [('apples',10.0), ('oranges',3.0)]
```

`test_cases/q5/arbitrage_shop3.test` tests whether `shopSmart.shopArbitrage(order, shops) == 22.0`.

Question 6: shopMinimum Function

Fill in the function `shopMinimum(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 minimum cost that you spend to buy the fruits. This question differs from Question 4 in that you can buy fruits in different shops rather than in only one shop. Don't change the file name or variable names. Note that we will provide the `shop.py` implementation as a "support" file, so you don't need to submit yours.

Run `python autograder.py` until question 6 passes all tests and you get full marks. Each test will confirm that `shopMinimum(orders,shops)` returns the correct answer given various possible inputs. For example, with the following variable definitions:

```
dir1 = {'apples': 2.0, 'oranges':1.0}
shop1 =  shop.FruitShop('shop1',dir1)
dir2 = {'apples': 1.0, 'oranges': 5.0}
shop2 = shop.FruitShop('shop2',dir2)
dir3 = {'apples': 1.5, 'oranges': 2.0}
shop3 = shop.FruitShop('shop3',dir3)
shops = [shop1, shop2, shop3]
order = [('apples',10.0), ('oranges',3.0)]
```

`test_cases/q6/minimum_cost2.test` tests whether `shopSmart.shopMinimum(order, shops) == 13.0`.

Submission

Zip your solution files: `addition.py`, `average.py`, `buyLotsOfFruit.py`, `shopSmart.py`, `shop.py` into `*.zip` file and submit to Moodle before the deadline. Do not submit any other files. You should not have edited any other files. Use the zip file format and not any other format for your submission. Ensure that you have submitted the correct files by downloading your submission. No submissions after the deadline will be accepted.

Acknowledgments

This work is based on previous work by John DeNero and Dan Klein et al. of berkeley.edu