P0 due 10pm ET

Due Sep 10th 19:00

1 Introduction

The programming assignments for this course assume you use Python 3.6. You may need to call `python36` or `python3.6` if `python --version` shows a Python version otherr than 3.6.x. Python 3.6 is already installed on the Andrew Linux machines, and you should be able to call it with `python3.6`. You are also free to use your own machine; we'll try to help you we can, but we cannot officially support all personal environments. Whether you work on an instructional machine or your own, the project submission for grading will always be through Gradescope.

Project 0 will provide the following:

- A mini-Python tutorial (Sections 2-3).
- A mini-debugging tutorial using pdb (Section 5).
- An autograding tutorial (Section 4): all programming assignments in this course will be autograded after you submit your code through Gradescope. For all assignments you can submit as many times as you like until the deadline. Every assignment's release includes its autograder for you to run yourself. This is the recommended, and fastest, way to test your code, but keep in mind you need to submit in Gradescope to get your grade registered.

Feel free to read only the sections relevant to you. The programming assignment itself is Sections 6-9; all other exercises are just for practice.

Files to Edit and Submit: You will fill in portions of `addition.py`, `buyLotsOfFruit.py`, `shopSmart.py` and `shopAroundTown.py` in tutorial.zip during the assignment. You should submit these files with your code and comments. 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, recitation, and Diderot 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.

2 Setting up your environment

You may use the CMU Andrew Linux machines or your personal computer. Our documentation and official support is for the CMU machines, but we will try to help you as much as we can with your personal computing environment.

Download tutorial.zip and use `unzip` to extract the contents of the zip file.

If you experience any troubles getting up and running with this tutorial, please come see us in office hours.

3 Python Basics

This is a barebones tutorial designed to familiarize you with the very basics of Python necessary to get through the course projects. If you would like more depth on any of these topics or more coverage of useful tools that are not as core to the language, check out the longer official Python tutorial.

3.1 Required Files

You can download all of the files associated with the Python mini-tutorial as a zip archive: tutorial.zip.

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

We encourage you to type all Python 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. It contains a list of the frequent problems previous students have encountered when following this tutorial.

3.2 Invoking the Interpreter

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 `python3.6` at the Unix command prompt.

Depending on your setup, you may have to type `python36` or just `python` if it already points to Python 3.6.

```
andrewid@unix:~/private$ python3.6 --version
Python 3.6.8
```

To exit the interpreter, call `exit()`.

3.3 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
```

3.4 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.s

```
>>> '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 using 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.

3.5 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__', '__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). Press 'q' to back out of a help screen.

3.6 Built-in Data Structures

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

3.7 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']
```

N

Note

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

3.8 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.

3.9 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
{'circle', 'square', 'triangle'}
>>> setOfShapes.add('polygon')
>>> setOfShapes
{'polygon', 'circle', 'square', 'triangle'}
>>> 'circle' in setOfShapes
True
>>> 'rhombus' in setOfShapes
False
>>> favoriteShapes = ['circle','triangle','hexagon']
>>> setOfFavoriteShapes = set(favoriteShapes)
>>> setOfShapes - setOfFavoriteShapes
{'polygon', 'square'}
>>> setOfShapes &amp; setOfFavoriteShapes
{'circle', 'triangle'}
>>> setOfShapes | setOfFavoriteShapes
{'polygon', 'square', 'circle', 'hexagon', 'triangle'}
```

N

Note

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

3.10 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.

N

Note

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

```
>>> studentIds = {'knuth': 42.0, 'turing': 56.0, 'nash': 92.0 }
>>> studentIds['turing']
56.0
>>> studentIds['nash'] = 'ninety-two'
>>> studentIds
{'knuth': 42.0, 'turing': 56.0, 'nash': 'ninety-two'}
>>> del studentIds['knuth']
>>> studentIds
{'turing': 56.0, 'nash': 'ninety-two'}
>>> studentIds['knuth'] = [42.0,'forty-two']
>>> studentIds
{'knuth': [42.0, 'forty-two'], 'turing': 56.0, 'nash': 'ninety-two'}
>>> studentIds.keys()
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.

3.11 Writing Scripts

Now that you've got a handle on using Python interactively, let's look at a simple Python script that demonstrates Python's `for` loop. Open the file called `foreach.py` and you will see 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('{} cost {} a pound'.format(fruit, price))
    else:
        print(fruit + ' are too expensive!')
```

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

```
andrewid@unix:~/private/tutorial$ python3.6 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 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]
```

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]
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:

```
andrewid@unix:~/private$ python3.6 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.

3.12 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.

3.13 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.

3.14 Writing Functions

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

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

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

```
andrewid@unix:~/private$ python3.6 fruit.py
That'll be 4.800000 please
Sorry we don't have coconuts
```

Exercise (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`.

3.15 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.

3.16 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 {} fruit shop'.format(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 {}".format(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.

3.17 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 = 'Entropy'
fruitPrices = {'apples': 1.00, 'oranges': 1.50, 'pears': 1.75}
entropyShop = shop.FruitShop(shopName, fruitPrices)
applePrice = entropyShop.getCostPerPound('apples')
print(applePrice)
print('Apples cost ${:.2f} at {}.'.format(applePrice, shopName))

otherName = 'Enthalpy'
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 {}.'.format(otherPrice, otherName))
print("My, that's expensive!")
```

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

```
andrewid@unix:~/private$ python3.6 shopTest.py
Welcome to Entropy fruit shop
1.0
Apples cost $1.00 at Entropy.
Welcome to Enthalpy fruit shop
4.5
Apples cost $4.50 at Enthalpy.
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 `entropyShop = 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).

3.18 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:

```
andrewid@unix:~/private$ python3.6 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.

3.19 More Python Tips and Tricks

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

- Use `range` to generate a sequence of integers, useful for generating traditional indexed `for` loops:

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

- After importing a file, if you edit a source file, the changes will not be immediately propagated in the interpreter. For this, use the `reload` command:

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

3.20 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, \textit{color} in pairList:
print('{} fruit costs {} and is the color {}'.format(fruit, price, color))
```

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

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

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

  Solution:

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

3.21 More References

- The place to go for more Python information: www.python.org
- You may also refer to the 15-112 website for more Python resources.

4 Autograding

All projects in this course will be autograded after you submit your code through Gradescope. For all projects you can submit as many times as you like until the deadline. Every project's release includes its autograder for you to run yourself. This is the recommended, and fastest, way to test your code, but keep in mind you need to submit in Gradescope to get your grade registered.

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

You can download all of the files associated the autograder tutorial as a zip archive, tutorial.zip. Unzip this file and examine its contents:

```
andrewid@unix:~/private$ unzip tutorial.zip
andrewid@unix:~/private$ cd tutorial
andrewid@unix:~/private/tutorial$ ls
VERSION
addition.py
autograder.py
buyLotsOfFruit.py
grading.py
projectParams.py
shop.py
shopAroundTown.py
shopSmart.py
testClasses.py
testParser.py
test_cases
textDisplay.py
town.py
tutorialTestClasses.py
```

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

- `addition.py`: source file for question 1
- `buyLotsOfFruit.py`: source file for question 2

- `shop.py`: source file for question 3
- `shopSmart.py`: source file for question 3
- `town.py`: source file for question 4
- `shopAroundTown.py`: source file for question 4
- `autograder.py`: autograding script (see below)

The remaining files you won't have to touch, but can look through if you're curious. Some that may be of interest are:

- `test_cases`: directory contains the test cases for each question
- `foreach.py`: example of Python's `for` loops
- `listcomp.py`: example of Python's list comprehension
- `grading.py`: autograder code
- `testClasses.py`: autograder code

The command `python3.6 autograder.py` grades your solution to all four problems. If we run it before editing any files we get a page or two of output:

```
andrewid@unix:~/private/tutorial$ python3.6 autograder.py
Starting on 6-20 at 13:41:08

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/food_price1.test
***   buyLotsOfFruit must compute the correct cost of the order
***   student result: "0.0"
***   correct result: "12.25"
*** FAIL: test_cases/q2/food_price2.test
***   buyLotsOfFruit must compute the correct cost of the order
***   student result: "0.0"
***   correct result: "14.75"
*** FAIL: test_cases/q2/food_price3.test
***   buyLotsOfFruit must compute the correct cost of the order
***   student result: "0.0"
***   correct result: "6.4375"
*** Tests failed.

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


Question q3
===========

Welcome to shop1 fruit shop
Welcome to shop2 fruit shop
*** FAIL: test_cases/q3/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/q3/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/q3/select_shop3.test
***   shopSmart(order, shops) must select the cheapest shop
***   student result: "None"
***   correct result: "<fruitshop: shop3="">"
*** Tests failed.

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


Question q4
===========
Welcome to shop1 fruit shop
Welcome to shop2 fruit shop
Welcome to shop3 fruit shop
*** FAIL: test_cases/q4/find_route1.test
***   shopAroundTown(orders, fruitTown, price) must select the best route
***   student result: "None"
***   correct result: "['shop1', 'shop2', 'shop3']"
Welcome to shop1 fruit shop
Welcome to shop2 fruit shop
Welcome to shop3 fruit shop
*** FAIL: test_cases/q4/find_route2.test
***   shopAroundTown(orders, fruitTown, price) must select the best route
***   student result: "None"
***   correct result: "['shop1', 'shop3']"
Welcome to shop1 fruit shop
Welcome to shop2 fruit shop
Welcome to shop3 fruit shop
*** FAIL: test_cases/q4/find_route3.test
***   shopAroundTown(orders, fruitTown, price) must select the best route
***   student result: "None"
***   correct result: "['shop2']"
*** Tests failed.

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


Finished at 13:41:08

Provisional grades
==================
Question q1: 0/1
Question q2: 0/1
Question q3: 0/1
Question q4: 0/1
------------------
Total: 0/4

Your grades are NOT yet registered.  To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.
```

For each of the four questions, this shows the results of that question's tests, the questions grade, 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 other 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 in the next tab.

5 Debugging

When working on projects **you will most likely spend more time debugging** than actually writing code, so it's important to know what tools are available to you. `pdb` is a useful module built in to Python for debugging that allows you to set breakpoints and step through your code line by line. While stepping through, you can examine the values of variables, run arbitrary test code, and even change values of variables.

To use `pdb`, you must import it at the top of your file then insert the line `pdb.set_trace()` somewhere in your code. When you run the code, execution will stop as soon as it hits that line (your breakpoint) and you will enter the debugger. As an example, we'll use the following file, which we'll call `breakpoints.py`:

```
import pdb

def foo():
    a = 5
    b = [7,8,9]
    print(a*b)

def bar():
    a = 3
    b = "cool "
    pdb.set_trace() # This sets a breakpoint here
    foo()
    print(a*b)
```

Now run your code interactively using `python3.6 -i breakpoints.py`:

```
>>> bar()
> breakpoints.py(12)bar()
-> foo()
(Pdb)
```

You are now in the debugger! There are four main commands available to you:

1. `list` : Prints the current line and a few surrounding lines in the code being stepped through.
2. `next` : Steps to the next line over any function calls. The subsequent line will be the next line in the same file.
3. `step` : Steps into a function call. The subsequent line will be the first line of the code run while executing that function.
4. `continue` : Exit the debugger, continuing execution until the program halts or another breakpoint is reached.

In addition, you can examine the values of variables in your current environment and even modify them. Try experimenting!

For more detailed info on `pdb` including examples, more commands, and abbreviations for commands, consult the official documentation.

Eclipse combined with PyDev has a nice UI for debugging. PyDev installation can be messy but is possible if you persevere.

6 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 to read:

```
def add(a, b):
    "Return the sum of a and b"
    print("Passed a={} and b={}, returning a+b={}".format(a,b,a+b))
    return a+b
```

Now rerun the autograder (omitting the results for questions 2 and 3):

```
andrewid@unix:~/private/tutorial$ python3.6 autograder.py -q q1
Starting on 1-21 at 23:52:05

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 ###

Finished at 23:41:01

Provisional grades
==================
Question q1: 1/1
Question q2: 0/1
Question q3: 0/1
Question q4: 0/1
------------------
Total: 1/4
```

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:

```
andrewid@unix:~/private/tutorial$ python3.6 autograder.py -q q1 --mute
Starting on 1-22 at 14:15:33

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 ###
```

7 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.

Run `python3.6 autograder.py` until question 2 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/q2/food_price1.test` tests whether the cost of `[('apples', 2.0), ('pears', 3.0), ('limes', 4.0)]` is 12.25.

8 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.

Run `python3.6 autograder.py` until question 3 passes all tests and you get full marks. Each test will confirm that `shopSmart(orders, shops)` returns the correct answer given various possible inputs.

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/q3/select_shop1.test` tests whether `shopSmart.shopSmart(orders1, shops) == shop1`, and `test_cases/q3/select_shop2.test` tests whether `shopSmart.shopSmart(orders2, shops) == shop2`.

9 Question 4: shopAroundTown

The `shopAroundTown(orderList, fruitTown, gasCost)` function takes as input a list of (fruit, numPounds) pairs as in question 3, a town object, and a number representing the cost of gas per mile traveled, and determines the best route to take to fill the fruit order. A town object contains a list of shops and the distances between each pair of shops and from each shop to 'home' in miles. See the documentation in the file `town.py` for a more detailed description of the representation of a town object. A valid route must start and end at home, so function returns a list of shops such that the sum of the total gas cost accrued from starting at home, going to each shop in the list in order, and returning to home, plus the total cost of buying the necessary fruit at the stores on the list, is minimized.

This question has a few more working parts than the previous three. Fortunately, we've provided an implementation of the solution! Unfortunately, it doesn't work.

Our approach is simple: We start with our list of all shops in town. We get all subsets of this list of shops, and filter the list of subsets to only contain subsets of shops that are carrying all the fruit we are trying to purchase. We then make a list of all permutations of all of these subsets (i.e. all possible routes through all possible valid choices of stores) and return the one which allows us to fill our order for the lowest cost. It's not the most clever algorithm (it checks \(O(n! * 2^n)\) routes where \(n\) is the number of shops), but it gets the job done.

Using your newly acquired debugging skills, find the bugs in `shopAroundTown.py`. You shouldn't have to make any major changes (specifically, there are four small changes, none of which require you to add or remove any lines). Bugs may be in the functions `shopAroundTown`, `getAllSubsets`, and `getAllPermutations`.

Try to find the bugs on your own; it’s good practice for the later projects.

If you get stuck though, expand this for a hint:

H

Hint

Put breakpoints at lines 44, 58, 71, and 75.

Don’t change the file name or function names, please. Note that we will provide the `town.py` implementation as a "support" file, so you don’t need to submit yours.

Run `python3.6 autograder.py` until question 4 passes all tests and you get full marks. Each test will confirm that `shopAroundTown(orderList, fruitTown, gasCost)` returns the correct answer given various possible inputs.

To run the autograder on an individual test case, use the `-t` flag:

```
python3.6 autograder.py -t test_cases/q4/find_route1
```

To look at what the test case is doing, you can open the `test_cases/q*/*.test` files with a text editor. `test_cases/q*/*.solution` provides the expected result for its corresponding .test file.

10 Submission

To submit your solution, upload the following files to the appropriate assignment on Gradescope:

- `addition.py`
- `buyLotsOfFruit.py`
- `shopSmart.py`
- `shopAroundTown.py`

Please do **not** upload the files in a zip file or a directory as the autograder will not work if you do so.