**Project 0: Unix/Python/Autograder Tutorial**

Version 3.0. Last Updated: 19 Dec 2024

Due: **see Canvas**

**Introduction**

The projects for this class assume you use Python 3.13 or compatible version for this class.

Project 0 will cover the following items. Skim what you already know and focus on the things you need to review or learn.

* A mini-UNIX tutorial (optional)
* Instructions on how to set up the right Python version (optional)
* A mini-Python tutorial, (optional)
* Project grading: Every project’s release includes its autograder for you to run yourself.

**Files to Edit and Submit:**

You will fill in portions of addition.py, buyLotsOfFruit.py, and shopSmart.py during the assignment. Please *do not* change any files other than the ones indicated in each given project. If you do, you will be given an E and be asked to fix them or any other changes that break autograding.

**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:** Copying someone else’s code and submitting it as your own is asking for a grade you did not earn and claiming mastery of skills of which you have not demonstrated mastery. We may or may not use a plagiarism tool on your code in this class.

**Getting Help:** You are not alone, though we do expect you to know and practice basic problem-solving skills. If you find yourself stuck on something, contact the instructor or a classmate for help. Class time, Office Hours, Discord and Teams are there for your support; please use them. If you need to, set up an appointment for help. These projects should be rewarding and instructional, not frustrating and demoralizing—but I don’t know when or how to help unless you ask.

**Discord and Teams:** Please be careful not to post spoilers nor executable code.

**Unix Basics**

Here are basic commands to navigate UNIX and edit files.

**File/Directory Manipulation**

When you open a terminal window on Linux, you’re placed at a command prompt:

[cs4470-ta@nova ~]$

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

[cs4470-ta@nova ~]$ mkdir foo

[cs4470-ta@nova ~]$ cd foo

[cs4470-ta@nova ~/foo]$

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

[cs4470-ta@nova ~/foo]$ ls

[cs4470-ta@nova ~/foo]$ touch hello\_world

[cs4470-ta@nova ~/foo]$ ls

hello\_world

[cs4470-ta@nova ~/foo]$ cd ..

[cs4470-ta@nova ~]$

Download <python_basics.zip> 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:

[cs4470-ta@nova ~]$ ls \*.zip

python\_basics.zip

[cs4470-ta@nova ~]$ unzip python\_basics.zip

[cs4470-ta@nova ~]$ cd python\_basics

[cs4470-ta@nova ~/python\_basics]$ 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
* xterm opens a new terminal window
* firefox opens a web browser
* Press “Ctrl-c” to kill a running process
* Append & to a command to run it in the background
* fg brings a program running in the background to the foreground

**Text Editing**

Use whatever text editor you like. If you use an IDE, it’s OK to write code in the editor, but you should run code from a terminal window OUTSIDE the IDE.

**Python Installation**

If you already have Python 3.13 installed and running, you can skip this step. Otherwise, download and install Python for your computer from python.org: <https://www.python.org/downloads/>.  
  
If you are installing Python on Windows, check the box to add Python to your system path when you run the installer.

**Using the Lab Machines**

If you use a Lab machine anywhere on Campus, it may or may not have Python 3.13 installed. The CS Tutor Lab on the 7th floor of CS should have a current version of Python installed if you need it.

**Python Basics**

**Required Files**

Please check the files.

**Table of Contents**

* [Invoking the Interpreter](https://inst.eecs.berkeley.edu/~cs188/sp20/project0/#Invoking_the_Interpreter)
* [Operators](https://inst.eecs.berkeley.edu/~cs188/sp20/project0/#Operators)
* [Strings](https://inst.eecs.berkeley.edu/~cs188/sp20/project0/#Strings)
* [Dir and Help](https://inst.eecs.berkeley.edu/~cs188/sp20/project0/#Dir_and_Help)
* [Built-in Data Structures](https://inst.eecs.berkeley.edu/~cs188/sp20/project0/#Built-in_Data_Structures)
  + [Lists](https://inst.eecs.berkeley.edu/~cs188/sp20/project0/#Lists)
  + [Tuples](https://inst.eecs.berkeley.edu/~cs188/sp20/project0/#Tuples)
  + [Sets](https://inst.eecs.berkeley.edu/~cs188/sp20/project0/#Sets)
  + [Dictionaries](https://inst.eecs.berkeley.edu/~cs188/sp20/project0/#Dictionaries)
* [Writing Scripts](https://inst.eecs.berkeley.edu/~cs188/sp20/project0/#Writing_Scripts)
* [Indentation](https://inst.eecs.berkeley.edu/~cs188/sp20/project0/#Indentation)
* [Tabs vs Spaces](https://inst.eecs.berkeley.edu/~cs188/sp20/project0/#TabsSpaces)
* [Writing Functions](https://inst.eecs.berkeley.edu/~cs188/sp20/project0/#Writing_Functions)
* [Object Basics](https://inst.eecs.berkeley.edu/~cs188/sp20/project0/#Object_Basics)
  + [Defining Classes](https://inst.eecs.berkeley.edu/~cs188/sp20/project0/#Defining_Classes)
  + [Using Objects](https://inst.eecs.berkeley.edu/~cs188/sp20/project0/#Using_Objects)
  + [Static vs Instance Variables](https://inst.eecs.berkeley.edu/~cs188/sp20/project0/#Static_vs_Instance_Variables)
* [Tips and Tricks](https://inst.eecs.berkeley.edu/~cs188/sp20/project0/#Tips_and_Tricks)
* [Troubleshooting](https://inst.eecs.berkeley.edu/~cs188/sp20/project0/#Troubleshooting)
* [More References](https://inst.eecs.berkeley.edu/~cs188/sp20/project0/#More_References)

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

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

You may find the [Troubleshooting](https://inst.eecs.berkeley.edu/~cs188/sp20/project0/#Troubleshooting) section helpful if you run into problems. It contains a list of the frequent problems previous students have encountered when following this tutorial.

**Invoking the Interpreter**

Python can be run in one of two modes. It can either be used *interactively*, via an 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 using the command python at the Unix or Windows command prompt. An example running Python on Windows is shown below.

C:\Users\nova>python

Python 3.10.1 (tags/v3.10.1:2cd268a, Dec 6 2021, 19:10:37) [MSC v.1929 64 bit (AMD64)] on win32

Type "help", "copyright", "credits" or "license" for more information.

>>>

**Operators**

The Python interpreter can be used to evaluate expressions, for example simple arithmetic expressions. Expressions that are entered at the prompt (>>>) are 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 refer to 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.

**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\_\_', '\_\_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(...) 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 Java’s collections package.

**Lists**

*Lists* store a sequence of mutable items:

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

>>> fruits[0]

'apple'

We can use the + operator to do list concatenation:

>>> otherFruits = ['kiwi', 'strawberry']

>>> fruits + otherFruits

>>> ['apple', 'orange', 'pear', 'banana', 'kiwi', 'strawberry']

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

>>> fruits[-2]

'pear'

>>> fruits.pop()

'banana'

>>> fruits

['apple', 'orange', 'pear']

>>> fruits.append('grapefruit')

>>> fruits

['apple', 'orange', 'pear', 'grapefruit']

>>> fruits[-1] = 'pineapple'

>>> fruits

['apple', 'orange', 'pear', 'pineapple']

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

>>> fruits[0:2]

['apple', 'orange']

>>> fruits[:3]

['apple', 'orange', 'pear']

>>> fruits[2:]

['pear', 'pineapple']

>>> len(fruits)

4

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

>>> lstOfLsts = [['a', 'b', 'c'], [1, 2, 3], ['one', 'two', 'three']]

>>> lstOfLsts[1][2]

3

>>> lstOfLsts[0].pop()

'c'

>>> lstOfLsts

[['a', 'b'], [1, 2, 3], ['one', 'two', 'three']]

**Exercise: Lists**

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

>>> dir(list)

['\_\_add\_\_', '\_\_class\_\_', '\_\_contains\_\_', '\_\_delattr\_\_', '\_\_delitem\_\_',

'\_\_delslice\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_',

'\_\_getitem\_\_', '\_\_getslice\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_iadd\_\_', '\_\_imul\_\_',

'\_\_init\_\_', '\_\_iter\_\_', '\_\_le\_\_', '\_\_len\_\_', '\_\_lt\_\_', '\_\_mul\_\_', '\_\_ne\_\_',

'\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_reversed\_\_',

'\_\_rmul\_\_', '\_\_setattr\_\_', '\_\_setitem\_\_', '\_\_setslice\_\_', '\_\_str\_\_',

'append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse',

'sort']

>>> help(list.reverse)

Help on built-in function reverse:

reverse(...)

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

>>> lst = ['a', 'b', 'c']

>>> lst.reverse()

>>> ['c', 'b', 'a']`

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

**Tuples**

A tuple is like a list except that it is immutable. Once it is created, you cannot change its content—but you can make a new one. Tuples are surrounded with parentheses instead of 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'])

Objects in a set are unordered—do not 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 key-value pairs. The key must be an immutable type (string, number, or tuple). The value can be any Python data type. The key-value pairs are not an ordered set, by definition, so your code should not rely on an implicit key ordering. If you need one, sort the keys, for example.

>>> 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:

[cs4470-ta@nova ~/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 because we’re looping over dictionary keys, which are unordered. To learn more about control structures in Python, check out the official [Python tutorial section on this topic](https://docs.python.org/3.9/tutorial/).

If you like functional programming you might also like map and filter. Why write a loop if you do not need one?

>>> 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:

[cs4470-ta@nova ~]$ 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.

**Pay Attention to Indentation!**

In Python indentation and whitespace are meaningful in the source code for interpretation. 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. If some code you are running is not working as you expect, check the indentation. It’s best to use four spaces for indentation – that’s what the course code uses. Avoid using tabs.

**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](http://stackoverflow.com/questions/119562/tabs-versus-spaces-in-python-programming) discussion on StackOverflow.

**Writing Functions**

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

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

def buyFruit(fruit, numPounds):

if fruit not in fruitPrices:

print("Sorry we don't have %s" % (fruit))

else:

cost = fruitPrices[fruit] \* numPounds

print("That'll be %f please" % (cost))

# Main Function

if \_\_name\_\_ == '\_\_main\_\_':

buyFruit('apples', 2.4)

buyFruit('coconuts', 2)

Rather than having a required main function, the \_\_name\_\_ == '\_\_main\_\_' check is used to delimit expressions which are executed when the file is called as a standalone program script from the command line. That code is NOT executed when the is imported as a module. The code after the main check is thus the same sort of code you would put in a main function in other languages. It can be convenient to enclose program startup code in a main() function (as we have done in previous course) even though Python does not require one.

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

(cs4470) [cs4470-ta@nova ~]$ 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**

You’ll have to use some objects in the programming projects, and so it’s worth reviewing the basics of classes and 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?

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

**Using Objects**

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

import shop

shopName = 'the Berkeley Bowl'

fruitPrices = {'apples': 1.00, 'oranges': 1.50, 'pears': 1.75}

berkeleyShop = shop.FruitShop(shopName, fruitPrices)

applePrice = berkeleyShop.getCostPerPound('apples')

print(applePrice)

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

otherName = 'the Stanford Mall'

otherFruitPrices = {'kiwis': 6.00, 'apples': 4.50, 'peaches': 8.75}

otherFruitShop = shop.FruitShop(otherName, otherFruitPrices)

otherPrice = otherFruitShop.getCostPerPound('apples')

print(otherPrice)

print('Apples cost $%.2f at %s.' % (otherPrice, otherName))

print("My, that's expensive!")

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

[cs4470-ta@nova ~]$ python shopTest.py

Welcome to the Berkeley Bowl fruit shop

1.0

Apples cost $1.00 at the Berkeley Bowl.

Welcome to the Stanford Mall fruit shop

4.5

Apples cost $4.50 at the Stanford Mall.

My, that's expensive!

So what just happended? The import shop statement told Python to load all of the functions and classes in shop.py. The line berkeleyShop = shop.FruitShop(shopName, fruitPrices) constructs an *instance* of the FruitShop class defined in *shop.py*, by calling the \_\_init\_\_ function in that class. Note that we only passed two arguments in, while \_\_init\_\_ seems to take three arguments: (self, name, fruitPrices). The reason for this is that all methods in a class have self as the first argument. The self variable’s value is automatically set to the object itself; when calling a method, you only supply the remaining arguments. The self variable contains all the data (name and fruitPrices) for the current specific instance (similar to this in Java). The print statements use the substitution operator (described in the [Python docs](https://docs.python.org/2/library/stdtypes.html#string-formatting) 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:

[cs4470-ta@nova ~]$ python person\_class.py

Now use the class as follows:

>>> import person\_class

>>> p1 = person\_class.Person(12)

>>> p1.get\_population()

1

>>> p2 = person\_class.Person(63)

>>> p1.get\_population()

2

>>> p2.get\_population()

2

>>> p1.get\_age()

12

>>> p2.get\_age()

63

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

**More Python Tips and Tricks**

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

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

for index in range(3):

print(lst[index])

* If you don’t need the indices of items in a for loop, prefer using range-based for loops instead to iterate over the values in the loop directly:

for x in lyst:

print(x)

* 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. 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 must 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: [ i ]. NOT parenthesis: ( i ).

* **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 or tuple. 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(f'{fruit} fruit costs {price} and is the color {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.

**More References**

* The place to go for more Python information: [www.python.org](http://www.python.org/)
* A good reference book: [Learning Python](http://oreilly.com/catalog/9780596513986/)

**Autograding**

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

You can download all of the files associated the autograder tutorial as a zip archive: <tutorial.zip> (note this is **different** from the zip file used in the UNIX and Python mini-tutorials, python\_basics.zip). Unzip this file and examine its contents:

[cs4470-ta@nova ~]$ unzip tutorial.zip

[cs4470-ta@nova ~]$ cd tutorial

[cs4470-ta@nova ~/tutorial]$ ls

addition.py

autograder.py

buyLotsOfFruit.py

grading.py

projectParams.py

shop.py

shopSmart.py

testClasses.py

testParser.py

test\_cases

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
* autograder.py: autograding script (see below)

and others you can ignore:

* test\_cases: directory contains the test cases for each question
* grading.py: autograder code
* testClasses.py: autograder code
* tutorialTestClasses.py: test classes for this particular project
* projectParams.py: project parameters

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

[cs4470-ta@nova ~/tutorial]$ python autograder.py

Starting on 1-21 at 23:39:51

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 ###

Finished at 23:39:51

Provisional grades

==================

Question q1: 0/1

Question q2: 0/1

Question q3: 0/1

------------------

Total: 0/3

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 three 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 section.

**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(f"Passed a = {a} and b = {b}, returning a + b = {a+b}")

return a + b

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

[cs4470-ta@nova ~/tutorial]$ python 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

------------------

Total: 1/3

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.

**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 python 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:

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

**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. We provide the shop.py implementation as a “support” file, so you don’t need to submit yours.

Run python 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. 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/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

**Submission**

You don’t need to submit this. It is only to recall your knowledge.