

## **CS-361L ARTIFICIAL INTELLIGENCE**

### **LAB 0: PYTHON REFRESHER**

**Type of Lab:** Close Ended

**Weightage:** 0 (Pre-requisite for lab 01)

<b>eLearning Objective</b>	<b>Domain / Level</b>	<b>CLO</b>	<b>Evaluation</b>
Revision of Python / a Refresher Lab	Cognitive / Understanding	Not Mapped	No Evaluation

## Lab 0: Python Refresher

The programming assignments in this course will be written in Python, an interpreter, 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 section helpful if you run into problems. It contains a list of the frequent problems previous students have encountered when following this tutorial.

### 1. Invoking the Interpreter

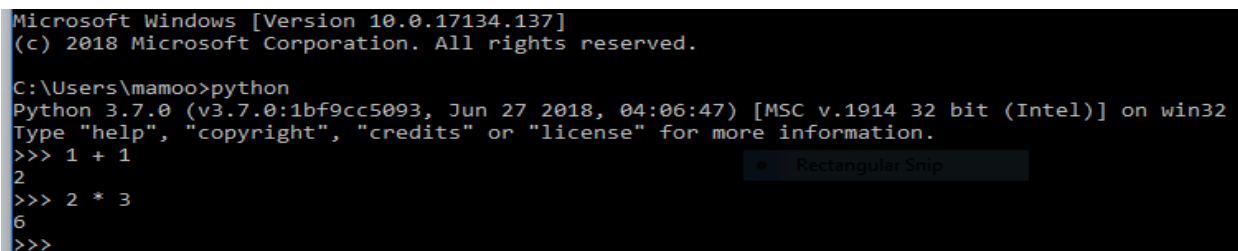
Python can be run in one of two modes. It can either be used *interactively*, via an interpreter, or it can be called from the command line to execute a *script*. We will first use the Python interpreter interactively. You invoke the interpreter by entering python at the command prompt.

Note: you may have to type python2.4, python2.5, python2.6 or python2.7, rather than python, depending on your machine.

```
[Lab ~]$ python
Python 2.6.5 (r265:79063, Jan 14 2011, 14:20:15)
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

### 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.



```
Microsoft Windows [Version 10.0.17134.137]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\mamoo>python
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:06:47) [MSC v.1914 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 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
```

**Things to do:**

- Good\_credit = True  
 Good\_loan = False  
 If Good\_credit and Good\_loan:  
     Print("eligible")  
 Else:  
     Print("ineligible")

**Define Output:****3. Strings**

Like Java, Python has a built-in string type. The + operator is overloaded to do string concatenation on string values.

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

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

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

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

We can also store expressions into variables.

```

>>> s = 'hello world'
>>> print (s)
hello world
>>> s.upper()
'HELLO WORLD'
>>> len(s.upper())
11
>>> num = 8.0
>>> num +=2.5
>>> print (num)
10.5

```

In Python, you do not have declare variables before you assign to them.

### Things to do:

If no of character in “name” is 3 print “name must have at least 6 characters” and if no of character in “name” is greater than 50 print “name must have character less than 50” else Print (“good name”)

## 4. 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:

```

>>> dir(s)
['_add_', '_class_', '_contains_', '_delattr_', '_dir_', '_doc_', '_eq_', '_format_', '_ge_', '_getat',
'_getitem_', '_getnewargs_', '_gt_', '_hash_', '_init_', '_init_subclass_', '_iter_', '_le_',
'_len_', '_lt_', '_mod_', '_mul_', '_ne_', '_new_', '_reduce_', '_reduce_ex_', '_repr_', '_rmul_',
'_setattr_', '_sizeof_', '_str_', '_subclasshook_', 'capitalize', 'casefold', 'center', 'count', 'e',
ncode', 'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal',
'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lo',
wer', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'spli',
t', '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

## 5. Lists

A list is an ordered collection of objects. The order of the elements is an innate characteristic of the list.

A list may contain any number of elements (constrained by the computer's memory, of course), of any type. The same object may occur any number of times.

Lists store a sequence of mutable items:

We can use the + operator to do list concatenation:

```
>>> fruits = ['apple', 'orange', 'pear', 'banana']
>>> fruits[0]
'apple'
>>> otherFruits = ['kiwi', 'strawberry']
>>> fruits + otherFruits
['apple', 'orange', 'pear', 'banana', 'kiwi', 'strawberry']
>>> fruits[-2]
```

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]
'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__', '__getattr__', '__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(list.reverse)
Help on method_descriptor:

reverse(self, /)
    Reverse *IN PLACE*.

>>> lst = ['a','b','c']
>>> lst.reverse()
>>> lst
['c', 'b', 'a']
```

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

### Think:

Which of the following are true of Python lists?

These represent the same list:

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

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

- ✧ A given object may appear in a list more than once
- ✧ There is no conceptual limit to the size of a list
- ✧ A list may contain any type of object except another list
- ✧ All elements in a list must be of the same type

### 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.

- Suppose  $t = (1, 2, 4, 3)$ , which of the following is incorrect?
  - a) `print(t[3])`
  - b) `t[3] = 45`
  - c) `print(max(t))`
  - d) `print(len(t))`
- $t = (1,2,4,3,8,9)$   
`[t[i] for i in range (0, len(t), 2)]`
  - a) `[2, 3, 9].`
  - b) `[1, 2, 4, 3, 8, 9].`
  - c) `[1, 4, 8].`
  - d) `(1, 4, 8)`

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

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

## 7. Dictionaries

The last built-in data structure is the dictionary which stores a map from one type of object (the key) to another (the value). The key must be an immutable type (string, number, or tuple). The value can be any Python data type.

**Note:** In the example below, the printed order of the keys returned by Python could be different than shown below. The reason is that unlike lists which have a fixed ordering, a dictionary is simply a hash table for which there is no fixed ordering of the keys (like 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'
```

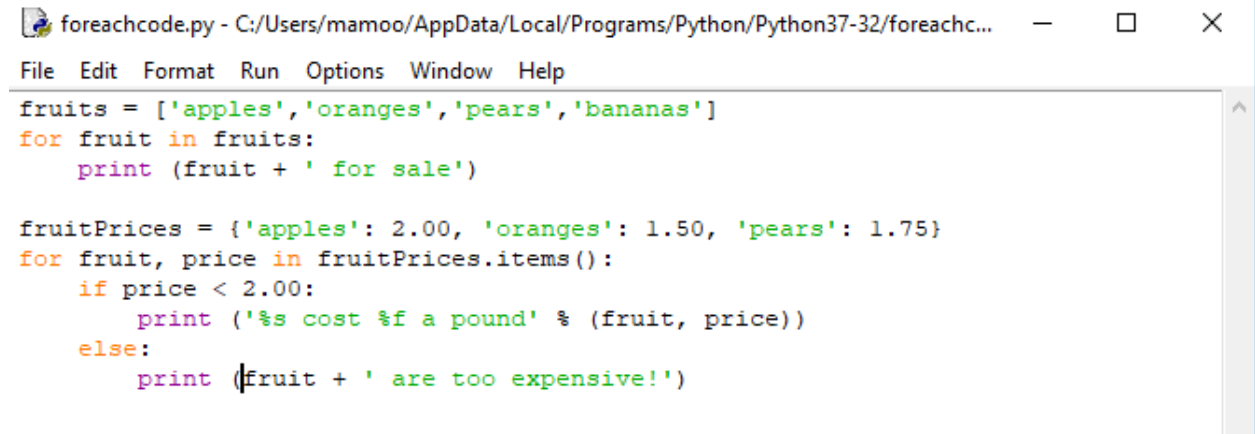
```
>>> studentIds
{'turing': 56.0, 'nash': 'ninety-two'}
>>> studentIds['knuth'] = [42.0, 'forty-two']
>>> studentIds
{'turing': 56.0, 'nash': 'ninety-two', 'knuth': [42.0, 'forty-two']}
>>> studentIds.keys()
dict_keys(['turing', 'nash', 'knuth'])
>>> studentIds.values()
dict_values([56.0, 'ninety-two',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

## 8. Writing Scripts

Now that you've got a handle on using Python interactively, let's write a simple Python script that demonstrates Python's for loop. Open the file called `foreach.py` and update it with the following:

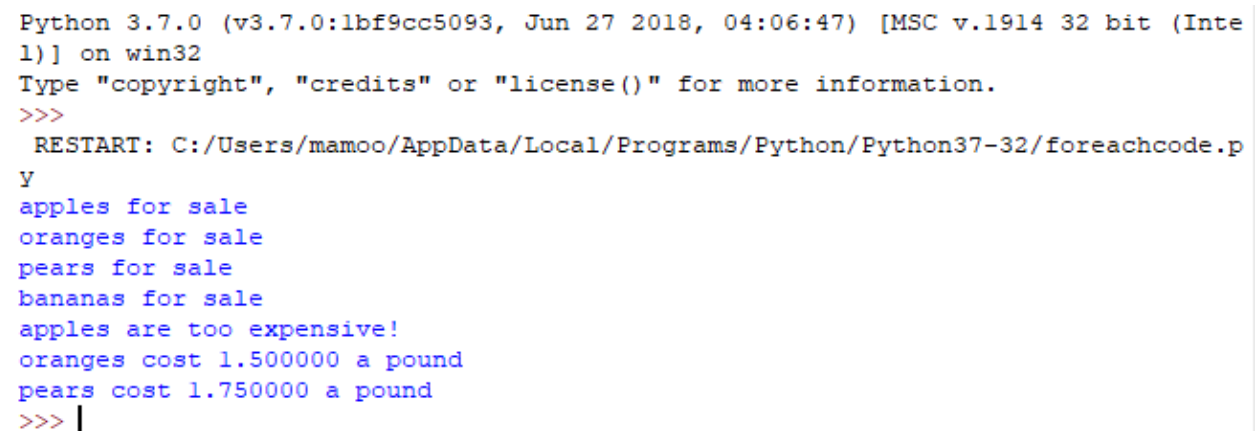


```

foreachcode.py - C:/Users/mamoo/AppData/Local/Programs/Python/Python37-32/foreachc...
File Edit Format Run Options Window Help
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!')
```

save the file in `foreach.py`, and then run:

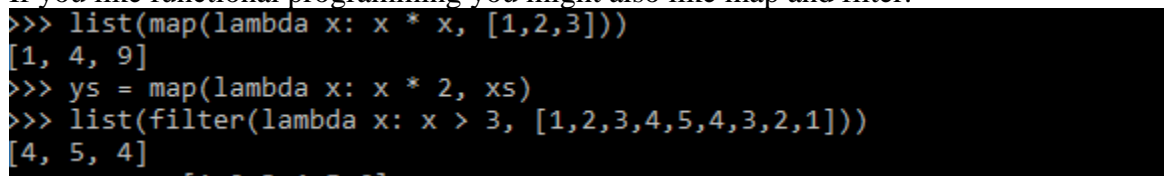


```

Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:06:47) [MSC v.1914 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
RESTART: C:/Users/mamoo/AppData/Local/Programs/Python/Python37-32/foreachcode.py
apples for sale
oranges for sale
pears for sale
bananas for sale
apples are too expensive!
oranges cost 1.500000 a pound
pears cost 1.750000 a pound
>>> |
```

Remember that the print statements listing the costs may be in a different order on your screen than in this tutorial; that's due to the fact that we're looping over dictionary keys, which are unordered.

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



```

>>> list(map(lambda x: x * x, [1,2,3]))
[1, 4, 9]
>>> ys = map(lambda x: x * 2, xs)
>>> 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)
[1, 3, 5]
>>> oddNumsPlusOne = [x+1 for x in nums if x % 2 ==1]
>>> print(oddNumsPlusOne)
[2, 4, 6]
>>>
```

### Beware of Indentation!

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.

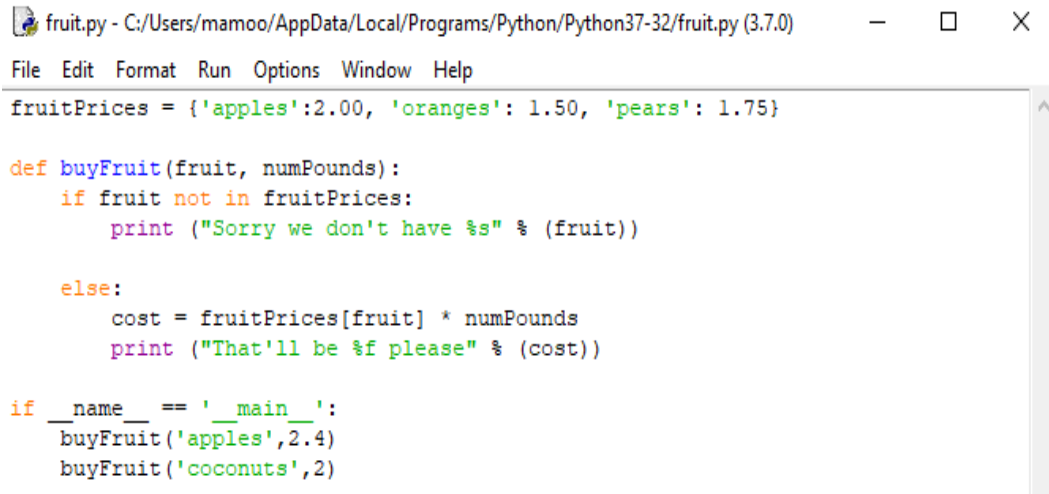
## 9. Tabs vs Spaces

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.

## 10. Writing Functions

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



```
fruit.py - C:/Users/mamoo/AppData/Local/Programs/Python/Python37-32/fruit.py (3.7.0)
File Edit Format Run Options Window Help
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))

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

Save this script as fruit.py and run it:

```
= RESTART: C:/Users/mamoo/AppData/Local/Programs/Python/Python37-32/fruit.py =
>>>
= RESTART: C:/Users/mamoo/AppData/Local/Programs/Python/Python37-32/fruit.py =
That'll be 4.800000 please
Sorry we don't have coconuts
>>> |
```

### Exercises:

1. Write a list comprehension which, from a list, generates a lowercased version of each string that has length greater than five.

.

```
strings = ['Some string', 'Art', 'Music', 'Artificial Intelligence']

for x in strings:
    if len(x) > 5:
        print(x.upper())
```

2. Let we have a dictionary D which contain roll number of students as key and its values shall be a list. The list has tuples as its item. Each tuple represents the name of the subject and GPA of that subject. You can visualize the data structure as of following:

Reference: The Lab contents are extracted from the BerkeleyX/CS188

Key	Value
2016-CS-700	[(DSA,3),(Algo,2.5),(AI,3)]
2016-CS-701	[(LA,3),(Algo,3.5),(PF,2.8)]
...	...
...	..
2016-CS-710	[(OOP,3),(DB,3.5),(PF,2.8)]

1. Write a program which shall print GPA of all students. Assume all subject have 3 credit hours.

```
listOflist = {'2016-CS-700' : [('DSA',3),('Algo',2.5),('AI',3)], '2016-CS-701' : [('LA',3),('Algo',3.5),('PF',2.8)], '2016-CS-710' : [('OOP',3),('DB',3.5),('PF',2.8)]}
for x in listOflist:
    y = listOflist[x]
    for n in y:
        print(n[1])
```

```
[('DSA', 3), ('Algo', 2.5), ('AI', 3)]
3
2.5
3
[('LA', 3), ('Algo', 3.5), ('PF', 2.8)]
3
3.5
2.8
[('OOP', 3), ('DB', 3.5), ('PF', 2.8)]
3
3.5
2.8
>>> |
```

2. Write a program that print the highest number obtained in DSA

```
highestnum = 0
listOflist = {'2016-CS-700' : [('DSA',2),('Algo',2.5),('AI',3)], '2016-CS-701' : [('DSA',3),('Algo',3.5),('PF',2.8)]}
for x in listOflist:
    yy = listOflist[x]
    print(yy)
    for y in yy:
        if y[0] == 'DSA' and highestnum < y[1]:
            highestnum = y[1]
print(highestnum)
```

```
>>>
RESTART: C:/Users/mamoo/AppData/Local/Programs/Python/Python37-32/exercise 1.py

SOME STRING
ARTIFICIAL INTELLIGENCE
[('DSA', 2), ('Algo', 2.5), ('AI', 3)]
[('DSA', 3), ('Algo', 3.5), ('PF', 2.8)]
[('OOP', 3), ('DB', 3.5), ('PF', 2.8)]
3
>>> |
```

3. Write a program that print the total number of students those have less than 2.5 GPA in AI?

```
totalNumStudent = 0
listOflist = {'2016-CS-700' : [('DSA',2),('Algo',2.5),('AI',3)], '2016-CS-701' : [('DSA',3),('AI',2.0),('PF',2.8)], '2016-CS-710' : [('AI',1.5),('DB',3.5),('PF',2.8)]}
for x in listOflist:
    yy = listOflist[x]
    print(yy)
    for y in yy:
        if y[0] == 'AI' and y[1] < 2.5:
            totalNumStudent += 1
print(totalNumStudent)
```

```
ARTIFICIAL INTELLIGENCE
[('DSA', 2), ('Algo', 2.5), ('AI', 3)]
[('DSA', 3), ('AI', 2.0), ('PF', 2.8)]
[('AI', 1.5), ('DB', 3.5), ('PF', 2.8)]
2
>>> |
```

### Object and Classes

**Class** — A blueprint created by a programmer for an object. This defines a set of attributes that will characterize any object that is instantiated from this class.

We define classes by using the class keyword, similar to how we define functions by using the define keyword.

Let's define a class called Shark that has two functions associated with it, one for swimming and one for being awesome:

**class Shark:**

```
    def swim(self):
```

```

    print("The shark is swimming.")

def be_awesome(self):
    print("The shark is being awesome.")

```

Because these functions are indented under the class `Shark`, they are called methods. **Methods** are a special kind of function that are defined within a class.

The argument to these functions is the word `self`, which is a reference to objects that are made based on this class. To reference instances (or objects) of the class, `self` will always be the first parameter, but it need not be the only one.

Defining this class did not create any `Shark` objects, only the pattern for a `Shark` object that we can define later. That is, if you run the program above at this stage nothing will be returned.

Creating the `Shark` class above provided us with a blueprint for an object.

- **Object** — An instance of a class. This is the realized version of the class, where the class is manifested in the program.

An object is an instance of a class. We can take the `Shark` class defined above and use it to create an object or instance of it.

We'll make a `Shark` object called `sammy`:

```
sammy = Shark()
```

Here, we initialized the object `sammy` as an instance of the class by setting it equal to `Shark()`.

Now, let's use the two methods with the `Shark` object `sammy`:

```

sammy = Shark()
sammy.swim()
sammy.be_awesome()

```

The `Shark` object `sammy` is using the two methods `swim()` and `be_awesome()`. We called these using the dot operator (`.`), which is used to reference an attribute of the object. In this case, the attribute is a method and it's called with parentheses, like how you would also call with a function.

Because the keyword `self` was a parameter of the methods as defined in the `Shark` class, the `sammy` object gets passed to the methods. The `self` parameter ensures that the methods have a way of referring to object attributes.

When we call the methods, however, nothing is passed inside the parentheses, the object sammy is being automatically passed with the dot operator.

Let's add the object within the context of a program:

```
class Shark:

    def swim(self):

        print("The shark is swimming.")

    def be_awesome(self):

        print("The shark is being awesome.")

def main():

    sammy = Shark()

    sammy.swim()

    sammy.be_awesome()

if __name__ == "__main__":

    main()
```

Let's run the program to see what it does:

### Output

The shark is swimming.

The shark is being awesome.

Here another example to make student more familiar with classes:

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.

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:

```
[cs188-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 happened? The `import shop` statement told Python to load all of the functions and classes in `shop.py`. The line `berkeleyShop = shop.FruitShop(shopName, fruitPrices)` constructs an *instance* of the `FruitShop` class defined in `shop.py`, by calling the `__init__` function in that class. Note that we only passed two arguments in, while `__init__` seems to take three arguments: (`self`, `name`, `fruitPrices`). The reason for this is that all methods in a class have `self` as the first argument. The `self`-variable's value is automatically set to the object itself; when calling a method, you only supply the remaining arguments. The `self`-variable contains all the data (`name` and `fruitPrices`) for the current specific instance (similar to this in Java). The print statements use the substitution operator.

## 1. 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:

Reference: The Lab contents are extracted from the BerkeleyX/CS188

```
[cs188-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.

We first compile the script:

```
[cs188-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.

## 2. Autograding

Reference: The Lab contents are extracted from the BerkeleyX/CS188

Before the submission, you can check your code through autograder. Every task 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 the code to GA.

You can download all of the files associated the autograder tutorial as a zip archive: [tutorial.zip](#) . Unzip this file and examine its contents:

```
[cs188-ta@nova ~]$ unzip tutorial.zip
[cs188-ta@nova ~]$ cd tutorial
[cs188-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

**Reference:** The Lab contents are extracted from the BerkeleyX/CS188

- 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:

```
[Lab]$ 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

```
=====
```

REFERENCE: THE LAB CONTENTS ARE EXTRACTED FROM THE BERKELEY/CS100

```

*** 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

```

Reference: The Lab contents are extracted from the BerkeleyX/CS188

\*\*\* 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".

### 3. Working of AutoGrader

#### 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=%s and b=%s, returning a+b=%s" % (a,b,a+b)
    return a+b
```

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

```
[cs188-ta@nova ~/tutorial]$ python autograder.py -q q1
```

```
Starting on 1-21 at 23:52:05
```

```
=====
```

```
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. You can also run the autograder with the option `--mute` to temporarily hide such lines, as follows:

```
[cs188-ta@nova ~/tutorial]$ python 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 ###

```

#### 4. Question No 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

#### 5. 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 `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)

```

Reference: The Lab contents are extracted from the BerkeleyX/CS188

```
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
```

### Test Yourself:

1)What Will Be The Output Of The Following Code Snippet?

```
class Sales:
```

```
    def __init__(self, id):
```

```
        self.id = id
```

```
        id = 100
```

```
val = Sales(123)
```

```
print (val.id)
```

✧ SyntaxError, this program will not run

✧ 100

✧ 123

✧ None of the above

2)What Will Be The Output Of The Following Code Snippet?

```
class Person:
```

```
    def __init__(self, id):
```

```
        self.id = id
```

```
sam = Person(100)
```

```
sam._dict_['age'] = 49
```

```
print (sam.age + len(sam._dict_))
```

✧ 1

Reference: The Lab contents are extracted from the BerkeleyX/CS188

- ✧ 2
- ✧ 49
- ✧ 50
- ✧ 51

3) Which Of The Following Can Be Used To Invoke The `_init_` Method In B From A, Where A Is A Subclass Of B?

- ✧ A. `super()._init_()`
- ✧ B. `super()._init_(self)`
- ✧ C. `B._init_()`
- ✧ D. `B._init_(self)`

4) class A:

```
def _init_(self, i = 0):
    self.i = i
```

class B(A):

```
def _init_(self, j = 0):
    self.j = j
```

def main():

```
b = B()
print(b.i)
print(b.j)
```

main()

- ✧ A. Class B inherits A, but the data field “i” in A is not inherited.
- ✧ B. Class B inherits A, thus automatically inherits all data fields in A.
- ✧ C. When you create an object of B, you have to pass an argument such as B(5).
- ✧ D. The data field “j” cannot be accessed by object b.

5)What Will Be The Output Of The Following Code Snippet?

```
class A:
    def __init__(self):
        self.calcI(30)
        print("i from A is", self.i)
    def calcI(self, i):
        self.i = 2 * i;

class B(A):
    def __init__(self):
        super().__init__()

    def calcI(self, i):
        self.i = 3 * i;
```

b = B()

- ✧ A. The `__init__` method of only class B gets invoked.
- ✧ B. The `__init__` method of class A gets invoked and it displays “i from A is 0”.
- ✧ C. The `__init__` method of class A gets invoked and it displays “i from A is 60”.
- ✧ D. The `__init__` method of class A gets invoked and it displays “i from A is 90”.

6)What Will Be The Output Of The Following Code Snippet?

```
class A:
    def __init__(self):
        self.calcI(30)

    def calcI(self, i):
        self.i = 2 * i;

class B(A):
```

Reference: The Lab contents are extracted from the BerkeleyX/CS188

```
def __init__(self):  
    super().__init__()  
    print("i from B is", self.i)  
def calcI(self, i):  
    self.i = 3 * i;
```

```
b = B()
```

- ✧ A. The `__init__` method of only class B gets invoked.
- ✧ B. The `__init__` method of class A gets invoked and it displays “i from B is 0”.
- ✧ C. The `__init__` method of class A gets invoked and it displays “i from B is 60”.
- ✧ D. The `__init__` method of class A gets invoked and it displays “i from B is 90”.

Once your code passes all tests, **submit your code through the right most tab, to register your grades with us**. Verify that our autograder gives the same output as yours.