# Project 0: Python/Autograder Tutorial

**Introduction**

The projects for this class assume you use Python 3.6. The autograder will error with a message about cgi.escape if you are using Python 3.8+

Project 0 will cover the following:

* A mini-UNIX tutorial (particularly important if you work on instructional machines),
* Instructions on how to set up the right Python version,
* A mini-Python tutorial,
* 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 in [tutorial.zip](https://inst.eecs.berkeley.edu/~cs188/su20/assets/files/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.

要编辑和提交的文件：您将在作业期间填写 tutorial.zip 中的 additional.py、buyLotsOfFruit.py 和 shopSmart.py 部分。您应该将这些文件与您的代码和注释一起提交。请不要更改此分发中的其他文件或提交除这些文件之外的任何我们的原始文件。

**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.

评估：您的代码将自动评分以确保技术正确性。请不要更改代码中提供的任何函数或类的名称，否则会对自动分级器造成严重破坏。然而，你的实施的正确性——而不是自动评分器的判断——将是你分数的最终评判者。如有必要，我们将单独审查和评分作业，以确保您的工作获得应有的荣誉。

**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.

学术不诚实：我们将根据课堂上的其他提交检查您的代码，以确保逻辑冗余。如果你复制了别人的代码并提交了一些小的改动，我们会知道的。这些作弊检测器很难被愚弄，所以请不要尝试。我们相信大家只提交自己的作品；请不要让我们失望。如果你这样做，我们将追求我们所能得到的最强烈的后果。

**Getting Help:** You are not alone! If you find yourself stuck on something, contact the course staff for help. Office hours, section, and the discussion forum 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.

获得帮助：您并不孤单！如果您发现自己遇到什么问题，请联系课程工作人员寻求帮助。办公时间、版块和论坛随时为您提供支持；请使用它们。如果您不能安排我们的办公时间，请告诉我们，我们会安排更多。我们希望这些项目是有益的和有指导意义的，而不是令人沮丧和士气低落。但是，除非您提出要求，否则我们不知道何时或如何提供帮助。

**Discussion:** Please be careful not to post spoilers.

讨论：请注意不要给他人直接提供完整答案。

**Python Installation**

Many of you will not have Python 3.6 already installed on your computers. [Conda](https://docs.conda.io/en/latest/) is an easy way to manage many different environments, each with its own Python versions and dependencies. This allows us to avoid conflicts between our preferred Python version and that of other classes. We’ll walk through how to set up and use a conda environment.

Prerequisite: [Anaconda](https://docs.anaconda.com/anaconda/install/). Many of you will have it installed; if you don’t, install it through the link.

**Python Basics**

**Required Files**

You can download all of the files associated with the Python mini-tutorial as a zip archive: [python\_basics.zip](https://inst.eecs.berkeley.edu/~cs188/su20/assets/files/python_basics.zip). If you did the unix tutorial in the previous tab, you’ve already downloaded and unzipped this file.

**Table of Contents**

* [Invoking the Interpreter](https://inst.eecs.berkeley.edu/~cs188/su20/project0/#Invoking_the_Interpreter)
* [Operators](https://inst.eecs.berkeley.edu/~cs188/su20/project0/#Operators)
* [Strings](https://inst.eecs.berkeley.edu/~cs188/su20/project0/#Strings)
* [Dir and Help](https://inst.eecs.berkeley.edu/~cs188/su20/project0/#Dir_and_Help)
* [Built-in Data Structures](https://inst.eecs.berkeley.edu/~cs188/su20/project0/#Built-in_Data_Structures)
  + [Lists](https://inst.eecs.berkeley.edu/~cs188/su20/project0/#Lists)
  + [Tuples](https://inst.eecs.berkeley.edu/~cs188/su20/project0/#Tuples)
  + [Sets](https://inst.eecs.berkeley.edu/~cs188/su20/project0/#Sets)
  + [Dictionaries](https://inst.eecs.berkeley.edu/~cs188/su20/project0/#Dictionaries)
* [Writing Scripts](https://inst.eecs.berkeley.edu/~cs188/su20/project0/#Writing_Scripts)
* [Indentation](https://inst.eecs.berkeley.edu/~cs188/su20/project0/#Indentation)
* [Tabs vs Spaces](https://inst.eecs.berkeley.edu/~cs188/su20/project0/#TabsSpaces)
* [Writing Functions](https://inst.eecs.berkeley.edu/~cs188/su20/project0/#Writing_Functions)
* [Object Basics](https://inst.eecs.berkeley.edu/~cs188/su20/project0/#Object_Basics)
  + [Defining Classes](https://inst.eecs.berkeley.edu/~cs188/su20/project0/#Defining_Classes)
  + [Using Objects](https://inst.eecs.berkeley.edu/~cs188/su20/project0/#Using_Objects)
  + [Static vs Instance Variables](https://inst.eecs.berkeley.edu/~cs188/su20/project0/#Static_vs_Instance_Variables)
* [Tips and Tricks](https://inst.eecs.berkeley.edu/~cs188/su20/project0/#Tips_and_Tricks)
* [Troubleshooting](https://inst.eecs.berkeley.edu/~cs188/su20/project0/#Troubleshooting)
* [More References](https://inst.eecs.berkeley.edu/~cs188/su20/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/su20/project0/#Troubleshooting) section helpful if you run into problems. It contains a list of the frequent problems previous AIINTRO 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 command prompt; or if you are using Windows that doesn’t work for you in Git Bash, using python -i.

(AIIntro) [AIIntro-ta@nova ~]$ python

Python 3.6.6 |Anaconda, Inc.| (default, Jun 28 2018, 11:07:29)

[GCC 4.2.1 Compatible Clang 4.0.1 (tags/RELEASE\_401/final)] on darwin

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

>>>

Copy

**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

Copy

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

Copy

**Strings**

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

>>> 'artificial' + "intelligence"

'artificialintelligence'

Copy

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

>>> 'artificial'.upper()

'ARTIFICIAL'

>>> 'HELP'.lower()

'help'

>>> len('Help')

4

Copy

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

Copy

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

Copy

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'

Copy

We can use the + operator to do list concatenation:

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

>>> fruits + otherFruits

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

Copy

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']

Copy

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

Copy

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']]

Copy

**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']

Copy

>>> help(list.reverse)

Help on built-in function reverse:

reverse(...)

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

Copy

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

>>> lst.reverse()

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

Copy

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

**Tuples**

A data structure similar to the list is the *tuple*, which is like a list except that it is immutable once it is created (i.e. you cannot change its content once created). Note that tuples are surrounded with parentheses while lists have square brackets.

>>> pair = (3, 5)

>>> pair[0]

3

>>> x, y = pair

>>> x

3

>>> y

5

>>> pair[1] = 6

TypeError: object does not support item assignment

Copy

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)

Copy

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'])

Copy

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

**Dictionaries**

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

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

['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

Copy

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!')

Copy

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

[AIIntro-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

Copy

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](https://docs.python.org/3.6/tutorial/).

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]

Copy

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)

Copy

This code is in a file called listcomp.py, which you can run:

[AIIntro-ta@nova ~]$ python listcomp.py

[1, 3, 5]

[2, 4, 6]

Copy

**Exercise: List Comprehensions**

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

**Beware of Indendation!**

Unlike many other languages, Python uses the indentation in the source code for interpretation. So for instance, for the following script:

if 0 == 1:

print('We are in a world of arithmetic pain')

print('Thank you for playing')

Copy

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')

Copy

there would be no output. The moral of the story: be careful how you indent! It’s best to use four spaces for indentation – that’s what the course code uses.

**Tabs vs Spaces**

Because Python uses indentation for code evaluation, it needs to keep track of the level of indentation across code blocks. This means that if your Python file switches from using tabs as indentation to spaces as indentation, the Python interpreter will not be able to resolve the ambiguity of the indentation level and throw an exception. Even though the code can be lined up visually in your text editor, Python “sees” a change in indentation and most likely will throw an exception (or rarely, produce unexpected behavior).

This most commonly happens when opening up a Python file that uses an indentation scheme that is opposite from what your text editor uses (aka, your text editor uses spaces and the file uses tabs). When you write new lines in a code block, there will be a mix of tabs and spaces, even though the whitespace is aligned. For a longer discussion on tabs vs spaces, see [this](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)

Copy

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:

(AIIntro) [AIIntro-ta@nova ~]$ python fruit.py

That'll be 4.800000 please

Sorry we don't have coconuts

Copy

**Advanced Exercise**

Write a quickSort function in Python using list comprehensions. Use the first element as the pivot. You can find the solution in quickSort.py.

**Object Basics**

Although this isn’t a class in object-oriented programming, you’ll have to use some objects in the programming projects, and so it’s worth covering the basics of objects in Python. An object encapsulates data and provides functions for interacting with that data.

**Defining Classes**

Here’s an example of defining a class named FruitShop:

class FruitShop:

def \_\_init\_\_(self, name, fruitPrices):

"""

name: Name of the fruit shop

fruitPrices: Dictionary with keys as fruit

strings and prices for values e.g.

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

"""

self.fruitPrices = fruitPrices

self.name = name

print('Welcome to %s fruit shop' % (name))

def getCostPerPound(self, fruit):

"""

fruit: Fruit string

Returns cost of 'fruit', assuming 'fruit'

is in our inventory or None otherwise

"""

if fruit not in self.fruitPrices:

return None

return self.fruitPrices[fruit]

def getPriceOfOrder(self, orderList):

"""

orderList: List of (fruit, numPounds) tuples

Returns cost of orderList, only including the values of

fruits that this fruit shop has.

"""

totalCost = 0.0

for fruit, numPounds in orderList:

costPerPound = self.getCostPerPound(fruit)

if costPerPound != None:

totalCost += numPounds \* costPerPound

return totalCost

def getName(self):

return self.name

Copy

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!")

Copy

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

[AIIntro-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!

Copy

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

Copy

We first compile the script:

[AIIntro-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

Copy

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])

Copy

* 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)

Copy

**Troubleshooting**

These are some problems (and their solutions) that new Python learners commonly encounter.

* **Problem:** ImportError: No module named py

**Solution:** For import statements with import <package-name>, do *not* include the file extension (i.e. the .py string). For example, you should use: import shop NOT: import shop.py

* **Problem:** NameError: name ‘MY VARIABLE’ is not defined Even after importing you may see this.

**Solution:** To access a member of a module, you have to type MODULE NAME.MEMBER NAME, where MODULE NAME is the name of the .py file, and MEMBER NAME is the name of the variable (or function) you are trying to access.

* **Problem:** TypeError: ‘dict’ object is not callable

**Solution:** Dictionary 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, color in pairList:

print('%s fruit costs %f and is the color %s' % (fruit, price, color))

Copy

* **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/) (From the UCB campus, you can read the whole book online)

**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](https://inst.eecs.berkeley.edu/~cs188/su20/assets/files/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:

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

[AIIntro-ta@nova ~]$ cd tutorial

[AIIntro-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

Copy

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:

[AIIntro-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.

Copy

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 tab.

**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

Copy

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

Copy

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

[AIIntro-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

Copy

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. 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)

shops = [shop1, shop2]

Copy

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**

In order to submit your project, run python submission\_autograder.py and submit the generated token file tutorial.token to the Project 0 assignment on Gradescope.