Introduction to Computer Programming

Exercises – Week 5. Classes

Part 1. Classes in Python

Exercise 1 - Defining classes (Essential)

The goal of this exercise is to write a program that creates a shopping list and then prints out all of the items and the total price.

1. Write a class called `Item` that has a constructor

   ```
   __init__(self)
   ```

   that prints "This is an item". From your main program, create an object of class Item called Apple. Run the program.
2. Change the constructor to include three additional arguments:

   ```
   __init__(self, Description, Number, UnitPrice)
   ```

   Use these arguments to create three new class attributes.
3. Change the print statement to print "Created a new item: `X`", where `X` is the item description.
4. From your main program, create an object of class Item called Apple with parameters "Apple", 1, and 0.5. Call `print(Apple.Description,Apple.Number,Apple.UnitPrice)` to print out the information.
5. Include a function in your class called `PrintItemInfo(self)` that prints all the information about the item. Call `Apple.PrintItemInfo()` from the main program.
6. Override the built-in `__str__()` function so that printing the instance of `item` prints the name of that item. Then, create a list called `ShoppingList`, add the Apple, and two other items to the list. Loop through the list and print out each item using the overridden `__str__()` method.
7. Write a loop in the main program to go over all items in `ShoppingList`, print out the item information and sum the total price (the price for one item is `Number*UnitPrice`). Print out the total price at the end.

Part 2. Inheritance

Exercise 2 - Deriving a class with inheritance (Essential)

1. Add a new class called `SpecialItem` which *inherits* from the `Item` class. The class signature should look like the following:

   ```
   def __init__(self, Description, Number, UnitPrice, SpecialInfo):
   ```

   and should call the `__init__()` function of the Item class passing in the `Description`, `Number`, and `UnitPrice` arguments, but storing the new variable `SpecialInfo` as an attribute of the new class.

2. As we did in the `Item` class, override the built in `__str__()` function to print the item description, but this time have it also print out the special information via `self.SpecialInfo`.
3. Override the `PrintItemInfo()` method of the `Item` class so that the special information is also printed.
4. Add a special item to your shopping list that requires instructions via the `SpecialInfo` argument, such as Paracetamol, which has the following special information: take two tablets every 6 hours. Check that `print(Paracetamol)` and `Paracetamol.PrintItemInfo()` work as expected. Verify that `Apple.PrintItemInfo()` works the same as before.

Part 3. Advanced questions

Exercise 3 - A class for vectors

The purpose of this exercise is to create a class for vectors and carrying out operations on vectors. We'll consider vectors of the form $\vec{v} = \langle x, y, z \rangle$.

1. For this exercise, we'll need some additional mathematical functions that are not available in Python by default. To enable these, add the line

   ```
   from math import *
   ```

   You should now be able to compute square roots using the `sqrt` function. Trigonometric functions (sin, cos, tan, etc) will be available now too.
2. Create a class called `Vector` with attributes `x`, `y`, `z`.
3. Overwrite the `__str__()` function so that the `print()` function can be used to print the vector in the form `<x, y, z>`. Check that this works by creating a vector $\vec{v} = \langle 1, 3, 2 \rangle$ and then calling `print(v)`.
4. Add a function call `norm` that computes the length of a vector, defined as $|\vec{v}| = \sqrt{x^2 + y^2 + z^2}$. Compute the length of the vector $\vec{v}$.
5. Overload the + operator by defining the `__add()__` function in the vector class. Remember that addition is a binary operation, so the `__add__()` function requires two arguments: `self` and `other`. Define a second vector $\vec{w} = \langle 5, 0, 1 \rangle$ and check that $\vec{v} + \vec{w} = \langle 6, 3, 3 \rangle$.
6. Now we'll overload the * operator so that it computes the dot product of two vectors. Recall that if $\vec{v} = \langle x, y, z \rangle$ and $\vec{w} = \langle a, b, c \rangle$ then $\vec{v}.\vec{w} = ax + by + cz$. The * operator can be overloaded by defining the `__mul__()` function in the vector class.
7. Use these methods and operations to compute the angle between the vectors $\vec{v}$ and $\vec{w}$. Recall that the angle θ between two vectors is defined by

   $$\theta = \arccos\left(\frac{\vec{v}.\vec{w}}{|\vec{v}||\vec{w}|}\right).$$

   The function arccos is defined in Python as `acos`.
8. (Very advanced) Let's suppose that we want to multiply a vector $\vec{v} = \langle x, y, z \rangle$ by a float f such that $f\vec{v} = \vec{v}f = \langle fx, fy, fz \rangle$. This can also be done by overloading the * operator, but there are some subtleties. One issue is that we have already defined * using the `__mul__()` function in Question 6 and this assumes the * operation is being applied to two vectors. To

overcome this issue, redefine the `__mul__()` function and use an `if` statement to determine which operation to carry out based on the type of `other`. Check that this works by computing `v * w` and `v * 1.0`.

Now try to compute `1.0 * v`. You'll notice an error occurs. This is because the order of arguments matters when calling Python functions. Writing `v * 1.0` is the same as calling `v.__mul__(1.0)`. Since we define the function `__mul__` in the vector class, we can provide instructions for how to evaluate this function when the argument is a float. However, writing `1.0 * v` calls the `__mul__` function defined in the float class, and Python doesn't know how to evaluate this function when it is passed a vector. Thankfully, there is an easy fix which avoids editing the float class. This involves defining the reflected multiplication function `__rmul__` in the vector class, which has two arguments `self` and `other`. When the command `1.0 * v` is executed, Python first tries to call `1.0.__mul__(v)` and when this fails, it will then try to run `v.__rmul__(1.0)`. This means that the reflected multiplication function can be defined in the same way as the normal multiplication function. Implement the `__rmul__` function in your vector class and verify that it works by calculating `1.0 * v`.