Software for Digital Innovation (CIS4044-N)
Tutorial 8: Object-Oriented Programming

Before You Start

Hint: Remember to think of classes as object templates that you can create as many of as you like (almost like dictionaries that can contain functions). Key terminology this week includes:

- *Constructor* – the function that creates a new instance of your class, like this: `myDiceRoller = DiceRoller(3)`
- *Instance* – a "copy" of a class created using a constructor (see above).
- *Inheritance* – when one class "inherits" the functionality of its parent class.

Consult the lecture material and/or your tutor for more guidance if needed.

Introduction

This session aims to familiarise you with the concepts and techniques you need to get started with object-oriented programming in Python. This is a big leap, and can be frustrating at times. Stick with it and reap the rewards!

Questions 5 onwards are especially challenging, do not be disheartened. Continue to seek feedback from your tutors on your Portfolio entries.

Question 1: Greeter

Let's get started with object-oriented programming by creating our first class and adding some functions to it.

1. Create a class called `Greeter` which has a constructor that takes one argument `name` (in addition to the usual `self`).
2. Now, create a function called `greet_to_screen()` inside this class that greets the user on the command line. Your class should look like this:

```
class Greeter:
    name = ""
    def __init__(self, name): # The constructor.
        self.name = name
    def greet_to_screen(self):
        print("Hello,", self.name)

greeter = Greeter("Fred")
greeter.greet_to_screen()
```

3. Test this file, what happens? See how `name` is stored at the *class level* as an *attribute* of the class when the *constructor* is called? How many different variables called `name` are there here?
4. Now, add another function to your class called `greet_to_file()` that takes an argument `filename` and writes the greeting to a file instead. For example, `greeter.greet_to_file("hi.txt")` would write `Hello, Fred` to the file `hi.txt`.

Question 2: A Class for Maths

Let's now take things up a level by returning some values from our class functions. Let's create a class that takes a number in its constructor, and allows us to add, subtract, multiply and divide using that number.

1. Create a new class called `MathEngine`, with a constructor that takes 1 argument `num` and stores it at the class level as an attribute (just like we did with name in question 1).
2. Now, add 4 functions to your class:
    a. The `add()` function should take a number as an argument, add `num` and return the result.
    b. The `sub()` function should take a number as an argument, subtract `num` and return the result.
    c. The `mult()` function should take a number as an argument, multiply it by `num` and return the result.
    d. The `div()` function should take a number as an argument, divide it by `num` and return the result.
3. Test your class by calling each of its functions and inspecting what they return.
4. Now, create a simple command-line based user interface for your program. Allow the user to specify a base number and construct an instance of your class. Then allow them to add, multiply, subtract and divide using that number until they type exit. <u>You are allowed to implement this however you like.</u>

Question 3: A Coin Flipper

Let's begin to build some really useful classes that approach problems we've faced in previous weeks, starting with a coin flipper.

1. Create a new class called `CoinFlipper` with a constructor that takes 1 argument `num_of_coins` that it stores as an attribute.
2. Now, add a function to your class called `flip()` that returns a list containing `num_of_coins` random Boolean values.
3. Copy across your answer to the exercise on coin flipping from week 3, and adjust it to use your new `CoinFlipper` class.
4. If you haven't done so already, move your `CoinFlipper` class into a separate file `coinflipper.py` stored in the same folder and import it into your solution with `from coinflipper import CoinFlipper`. You now have a reusable coin flipping algorithm encapsulated inside a class!

Question 4: A Dice Roller

Let's now implement a dice roller as a reusable program component in the spirit of the previous question.

1. Create a new class called `DiceRoller` with a constructor that takes 1 argument `num_of_dice` that it stores as an attribute.
2. Now, add a function to your class called `roll()` that rolls `num_of_dice` dice and returns the value of the roll. For example, if `num_of_dice` is 2, then 2 random numbers between 1 and 6 are generated, added together and returned as a value from 2-12.
3. Now, add another function to your class called `roll_many()` that takes an argument `n` and returns a list containing the results of calling `roll()` `n` times.
4. As for question 3, from your week 3 exercise on dice rolling, copy your solution across and integrate your new reusable `DiceRoller` class, imported from a separate file as before.

Question 5: A Shopping Cart

This question is **much more advanced**, but give it a go! We're going to be creating several classes for this one, and getting them all to work together with each other.

1. First, create 2 classes. One called `Apple` and one called `Pear`. Each of these has a constructor that takes two arguments—`quantity` and `price`—and stores them as attributes.
2. Each of these classes additionally has a `get_name()` function that returns either the string `"Apple"` or `"Pear"` depending on the class.
3. Give each of these classes a function called `get_total()` which returns the `price` of the instance multiplied by the `quantity` of the instance.
4. Test each of these classes thoroughly through the REPL or using `print()` statements in your code file. Does `get_total()` give you what you expect?
5. Now, create another class called `ShoppingCart` that has a constructor that simply initialises an empty list (`[]`) called `items` as a class attribute. This class will have four functions:
    a. The `add_to_cart()` function takes one argument `item` which should be either an `Apple` or a `Pear` and adds it to `items`.
    b. The `remove_from_cart()` function takes one argument `name` which should result in any item that returns a matching string from its `get_name()` function being removed from `items`.
    c. The `empty_cart()` function should set `items` to the empty list.
    d. The `cart_total()` function should return the result of adding up the return values of `get_total()` for all items in the cart.
6. Create a simple command-line user interface that allows a user to add however many apples and pears they like to their cart then prints the cart total. You are allowed to implement this however you like.

Question 6: A Shopping Cart (Inheritance)

You'll notice that in question 5, you have a lot of repetition between the Apple and Pear classes you created. We can work around this, and boost the maintainability and readability of our code in the process! **This question is conceptually very difficult. Ask if you're unsure!**

1. Observe that both your `Apple` and `Pear` classes share two attributes (`price` and `quantity`) and one function `get_total()` which are identical between them. Move these to another class called `CartItem`.
2. Now, ensure that your `Apple` and `Pear` classes *subclass* this `CartItem` class to *inherit* its attributes and functions.
3. Now, make sure your `Apple` and `Pear` subclasses call the *superconstructor* with the appropriate price hard-coded. Your `Apple` and `Pear` class constructors should take only one argument `quantity`.
   Hint: Call the *superconstructor* in for a 70p apple/pear like so:
   `super().__init__(quantity, 0.70)`
4. Your application should work just as before, but with no code repetition between your classes! If it doesn't, debug it until it does or ask your tutor for assistance.

Extension Exercise: Discounts

The fact that this is the extension exercise should make it clear that this question is **extremely challenging**. Attempt it, but don't worry if it takes a while to wrap your head around it.

1. Adapt your solution to question 6 by adding another empty list as an attribute to your `ShoppingCart` class called `vouchers`. Vouchers can be either flat or percentage-based:
   a. Flat vouchers give you a discount of a flat amount of money (say, £5).
   b. Percentage based vouchers give you a percentage off (say, 50% off).
2. Create a `Voucher` class with a Boolean attribute `flat` that indicates whether or not it's a flat voucher (if `True`) or a percentage voucher (if `False`). Initialise this attribute via the constructor.
3. The `Voucher` class should also have an attribute (again, initialised via the constructor) called `amount` to indicate its value.
4. Add an `add_voucher()` function to your `ShoppingCart` class to enable adding vouchers to the vouchers list in your cart.
2. Finally, ensure your `cart_total()` function correctly applies these vouchers in the order they were added before returning the total. The number should never be negative. If it ends up negative, return `0` instead.
5. Thoroughly test your work using the REPL and `print()` statements.

Document History

Revision 0 (14-Nov-20): This is the initial version of the 2020/21 exercise.