Week 1 - Review of Objects and Classes

Updated Sept 12, 6:30pm

Some Terminology (review, hopefully!)

From your previous studies, you should recall that an **object** in a program is a structured collection of data, bundled together with some functions that can operate on this data. A **class** is a type of object, which allows us to create many objects with the same properties without redundant code. Just as how identifying a variable as an "int" or "string" gives us a lot of information about its value and what operations we can perform on it, classes are *custom* types that are designed specifically for some specific goal or software in mind.

Each piece of data associated with a class is an **attribute**. Classes can have an arbitrary number of attributes, and they can all be different types: integers, floats, strings, lists, dictionaries, and even other classes. The operations associated with classes are called the **methods** of the class. We'll see lots of examples of these later on this page.

Classes give us a way of organizing our functions and data into logical blocks based on their purpose and relationships with each other. This is helpful both when we're designing programs, and when we decide later to add or remove related features. In fact, in Python, everything is an object, from a list to a string to even other functions!

A common programming task is to take an English description of a problem, and try to design classes to model the problem. The main idea is that the class(es) should correspond to the most important noun(s) in the problem, the attributes should be the information (other nouns or adjectives) associated with these noun(s), and the methods correspond to the verbs. Here are two examples.

People

We'd like to create a simple model of a person. A person normally can be identified by her name, but might commonly be asked about her age (in years). We want to be able to keep track of a person's mood throughout the day: happy, sad, tired, etc. Every person also has a favourite food: when she eats that food, her mood becomes 'ecstatic'. And though people are capable of almost anything, we'll only model a few other actions that people can take: changing their name, and greeting another person with the phrase 'Hi ____, it's nice to meet you! I'm ____.'

A class design

Taking the above description, we might have a `Person` class, with `name`, `age`, `mood`, and `favourite_food` attributes. *What might the types of these attributes be?*

As for methods, our `Person` class would support operations like `eat`, `change_name`, and `greet`; notice that these are the main verbs that appear in the above description!

Rational Numbers

It's slightly annoying for math people to use Python, because fractions are always converted to decimals and rounded, rather than kept in exact form. Let's fix that! A rational number consists of a numerator and denominator; the denominator cannot be 0. Rational numbers are written like 7/8. Typical operations include determining whether the rational is positive, adding two rationals, multiplying two rationals, comparing two rationals, and converting a rational to a string.

Some code

Let's see some basic Python code for creating a class. We'll use the person example.

```
class Person:

    def __init__(self, name, age, food):
        """ (Person, string, int, string) -> NoneType
        Create new Person object with given name, age, and favourite food.
        """
        self.name = name
        self.age = age
        self.favourite_food = food
        self.mood = "Happy"
```

This creates a new `Person` class with just a single method: `__init__` is a special Python name for the **constructor** method of a class, and is how we create new objects of this class.

The `self` parameter is another important keyword: it refers to the current object for which the method is being called. All this initializer does is take two parameters and set the new `Person` object's attributes to them.

In the shell:

```
>>> yoda = Person('Yoda', 500, 'chicken') # Notice *how* we call the constructor
>>> yoda.name
'Yoda'
>>> yoda.age
500
>>> yoda.mood
'Happy'
>>> yoda.name = 'Sidious'
>>> yoda.name
'Sidious'
```

Public vs. Private

Notice that there is no concept of "private" attributes; we can both access and change all attributes outside of the class at will. Even though it seems like a good thing because it's easier to write code, we can also do things like creating new attributes at runtime:

```
>>> yoda.powers = 'the force'
>>> yoda.powers
'the force'
```

Not only does this lead to careless errors for beginners, it is most often the case that accessing attributes is *unnecessary* and *undesirable* for using the class. This is the principle of **information hiding**: when designing a class, we want to make a clear distinction between the *public interface* (the features of the class that other code outside the class may use), and the *private implementation* (i.e., how we've written the methods and attributes of the class). One of the biggest advantages of designing our programs in this way is that after our initial implementation, we can feel free to modify it (e.g., add new features or make it more efficient) without disturbing the public interface, and rest assured that this doesn't affect other code that might be using this class. Moreover, the public interface should be as "small" as possible: we want to restrict precisely what others can do with our classes, to others from abusing them.

In this course, the class attributes will almost always belong the private implementation, and because Python won't enforce this, it will up to you to remain vigilant about how you use these attributes. This will be especially true once we start talking about different abstract data types, where the *only* defining traits will be the *methods* available for that data type.

A Class Method

Here's a simple example of creating a `Person` method for eating some food.

```
    def eat(self, food):
        """ (Person, string) -> NoneType

        Make this person eat the food. Change the mood of this person if food
        is the same as this person's favourite food.
        """

        if self.favourite_food == food:
            self.mood = 'ecstatic'
```

Notice again the use of the `self` keyword to stand in for the calling object.

```
>>> yoda = Person('Yoda', 500, 'chicken')
>>> yoda.eat('chicken')
>>> yoda.mood
'ecstatic'
```

Computer Science
UNIVERSITY OF TORONTO

David Liu (liudavid at cs dot toronto dot edu)
Come find me in BA4260
Site design by Deyu Wang (dyw999 at gmail dot com)