Classes and Objects

Overview

- When writing large programs, we want to design them in a well-organised way.
    - Easier to write.
    - Easier to reason about.
    - Easier to make work as intended.
- A popular approach in many languages is *Object Oriented Programming*.
- OOP is a core feature of languages such as Java.
- Python supports OOP in general, although not as fully or rigorously as other languages.
- We will first explain what OOP is, and then what you can do with it in Python.

Object Oriented Programming

- Key idea:
    - Identify types of 'thing' that your program deals with, and code them as self-contained *classes* - pieces of code that contain some data (variables), and operations on that data (functions).
    - Create instances - *objects* - of a particular class, with their own specific values for the class data, and modify them by calling the class operations.
- Example: Cat Simulator 2016
    - Say we are writing a program to (*very simplistically*) simulate cats.
    - We could define a `Cat` class representing the key features and abilities of a generic cat.
    - The data would be attributes - things that *describe* a cat - like name, age, colour, etc.
    - The operations would be actions - things that a cat can *do* - like walk, eat, meow, etc.
    - Once the `Cat` class was defined, we could create instances of that class - i.e. `Cat` objects - representing individual cats.
    - All cats would have a name, an age, a colour, etc, but the values might be different for each cat.
        - Whiskers, 2, Brown
        - Snowball, 5, White
        - Tom, 5, Grey
    - Similarly, all cats could walk, eat, meow, etc, and all cats would do so in the same way, but performing these actions only affects the cat that is doing so.
        - If Snowball walks, that doesn't necessarily mean that Tom also walks.
    - By making cats into a class, we make it much easier to create new cats or change how cats in general behave - all the code that makes a cat is wrapped up in one place.

OOP in Python

- We define a class using the syntax:
    - ```
      class class_name:
          ... class definition ...
      ```
      ```

- Within the class definition, we place variables (data) and functions (operations) that belong to the class.
    - Variables in classes are often known as *member variables* or *members*.
    - Functions in classes are often known as *methods*.
    - If we specify variables, the value we assign to those variables will be the default starting value for all objects of that class.
        - Unlike in other, stricter object oriented languages, we don't have to define all the variables we want to use in the class definition - so long as they exist before they are used, everything is alright. We will cover this more in a moment.
    - All functions that belong to classes must have a special first parameter, which by convention is called `self`. When the function is called on an object, this parameter is automatically filled in with the object by Python. This is so that the function can then access the object itself.
- After a class has been defined, we can create an object (aka instance) of that class, and assign it to a variable, using the syntax:
    - *`object_name = class_name()`*
    - This looks a lot like a function call, and it is! More or than later.
- We can access variables and functions that belong to an object using the syntax:
    - *`object_name.variable_name`*
    - *`object_name.function_name(param1, param2)`*
    - We've seen something similar when using modules!
- Example: Circle Class
    - This class represents a circle, defined only by its radius. The default radius is 1.0.
    - The operations that can be performed on a circle are to calculate its circumference and area.
    - These are all things we have seen in previous lessons, only now wrapped up into a class!
    - ```
      pi = 3.14159    # (not part of the class)

      class Circle:
          radius = 1.0

          def circumference(self):
              return 2 * pi * self.radius

          def area(self):
              return pi * self.radius * self.radius
      ```
    - Once this is defined, we can create and use circle objects - we can change their radii and measure their circumferences and areas.
    - Note how the function calls do not take any parameters, but the `self` parameter is automatically filled in by Python.
    - ```
      x = Circle()
      y = Circle()
      y.radius = 2.5
      ```

```
print("x:")
print(x.circumference())
print(x.area())
print("y:")
print(y.circumference())
print(y.area())
```

- Exercise: Rectangle Class
  - Define a class to represent a rectangle. A rectangle has a width and a height, and should be able to report its perimeter and its area. Create some rectangle objects to demonstrate your class works.
  - ```
    class Rectangle:
        width = 1.0
        height = 1.0

        def perimeter(self):
            return 2 * (self.width + self.height)

        def area(self):
            return self.width * self.height

    x = Rectangle()
    x.width = 2.0
    x.height = 4.5
    print(x.perimeter())
    print(x.area())
    ```

Initialisation

- We will often want to perform some initial setup when creating an instance of a class.
  - For instance, giving specific values to the variables.
  - Doing this manually when we create the class is annoying, and poor design!
- We can define a special initialisation function called `__init__` in the class definition.
  - This function is automatically called when an instance of the class is created.
  - Being a class function, its first parameter must be `self`.
  - Any remaining parameters must be specified when we create an object.
- Example:
  - ```
    pi = 3.14159    # (not part of the class)

    class Circle:
        def __init__(self, radius):
            self.radius = radius

        def circumference(self):
            return 2 * pi * self.radius

        def area(self):
            return pi * self.radius * self.radius
    ```

    - ```
      x = Circle(1.0)
      y = Circle(2.5)
      ```
- Exercise: Rectangle Class with Initialiser
    - Adapt your rectangle class to use an initialiser to give the width and height.
    - ```
      class Rectangle:
          def __init__(self, width, height):
              self.width = width
              self.height = height

          def perimeter(self):
              return 2 * (self.width + self.height)

          def area(self):
              return self.width * self.height

      x = Rectangle(2.0, 4.5)
      print(x.perimeter())
      print(x.area())
      ```

<u>References</u>

- We have alluded to references before, but now we know objects, we can demonstrate them properly.
- Variables storing an object in Python actually store a *reference* to that object, rather than the object itself.
    - Think of it as the object itself being stored somewhere in the computer's memory.
    - The variable stores a signpost, pointing to that piece of memory where the object can be found.
- References are powerful, and have many advantages - but if you're not careful, they can cause confusion as well.
- Consider the following:
    - ```
      x = Rectangle(2.0, 4.5)
      y = x
      x.width = 5.0
      print(y.width)        # Outputs 5.0
      ```
    - There is only one `Rectangle` object in memory!
    - The line `y = x` copies the *reference* - i.e. the signpost - rather than the object itself, so both variables refer to the same object!
    - Hence, a change to one changes the other.
    - To create a copy of the rectangle, we would need to create a function that made a new `Rectangle` object with the same width and height.
        - ```
          def clone(self):
              return Rectangle(self.width, self.height)
          ```
        - ```
          x = Rectangle(2.0, 4.5)
          y = x.clone()
          ```
          ```

```
x.width = 5.0
print(y.width)        # Outputs 2.0
```

- Now consider this:
  - ```
    class MyClass:
        mydata = Rectangle(1.0, 1.0)

        def change_width(self, width):
            self.mydata.width = width
    ```
  - When class variables are defined within the class, every instance of the class will have the same value in the variable.
    - For simple data types like numbers and strings, this is fine.
    - For objects, that means that each object variable will contain the same reference, though!
  - ```
    x = MyClass()
    y = MyClass()
    x.change_width(8.0)
    print(y.mydata.width)      # Outputs 8.0
    ```
  - It is therefore safest to define each of your classes' variables inside the initialiser function `__init__`, rather than in the body of the class. That way, each variable containing an object will contain a unique object.

Inheritance

- A very useful feature of OOP is the concept of *inheritance*.
- We can define one class to be *derived* from another class, *inheriting* its features.
- This is useful when we want to specialise a particular class, or have some functionality shared between lots of different classes.
- Example: Animal Simulator 2016
  - Rather than just simulating cats, maybe we want to simulate all animals.
  - Animals have a lot of shared features - they all eat, move, sleep, etc.
  - But they all have their differences - not all animals have legs, for instance.
  - Some animals also do similar things, but in different ways - many animals can make noises, but they all make different noises.
  - So we could define an Animal class, containing features applicable to all animals.
  - Then we could derive Cat, Dog, Hamster, Snake, Lizard, etc classes from it, and specialise each class with the features specific to each creature.
  - We could introduce multiple levels of inheritance: we could have a Mammal class and a Reptile class derived from Animal, then derive the Cat, Dog, Hamster, etc classes from Mammal and the Snake, Lizard, etc classes from Reptile.
- We can derive a new class from an existing class using the syntax:
  - ```
    class child_class(parent_class):
        ... class definition ...
    ```
- The derived class will have all of the variables and functions of its parent class, plus any new variables and functions that are defined.
- If we redefine any function, it replaces the one from the parent class.
    ```

    - If we still need access to the parent class function, we can do so with:
        - *parent_class_name*.*function_name*(self)
    - This is particularly useful for accessing the `__init__` function of a parent class.
- Example:

```
class Animal:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def speak(self):
        print("%s makes a noise!" % (self.name))

class Hamster(Animal):
    def __init__(self, name, age):
        Animal.__init__(self, name, age)

class Cat(Animal):
    def __init__(self, name, age):
        Animal.__init__(self, name, age)

    def speak(self):
        print("%s goes meow!" % (self.name))

class Dog(Animal):
    def __init__(self, name, age, breed):
        Animal.__init__(self, name, age)
        self.breed = breed

    def speak(self):
        print("%s goes woof!" % (self.name))
```

- Exercise:
    - Extend our zoo!
    - You could:
        - Add new types of animal.
        - Make existing animals more detailed - add colours, number of limbs, etc.
        - Give animals new abilities.
        - Or anything else you can think of!
    - Write some code that demonstrates your animals!

<u>Summary</u>

- Object oriented programming lets us structure our code in a way that is easy to reason about.
- We can devise ways to represent ‘real’ objects within our code.

    - We can also extend this to ‘theoretical’ objects, such as complicated data structures and algorithms.
- Classes can inherit from others, allowing code reuse and specialisation.

Next lesson: Random Numbers, Files, and Onwards from Here