Classes and Objects

Object Oriented Programming

Represent self-contained ‘things’ using *classes*.

A class consists of:

- Data (stored in variables)
- Operations on that data (written as functions)

Represent individual *instances* of classes as *objects*.

Example: Cats

Represent cats using a class called `Cat`.

- Data: attributes of a cat
  - name, age, colour, etc
- Operations: actions a cat can perform
  - walk, eat, meow, etc

Individual cats would be objects - instances of the `Cat` class.

Creating a Class

Definition of a class:

```
class class_name:
    ... class definition ...
```

Data and operations are placed within the class definition.

- Data: *member variables* or *members*
- Operations: *methods*

Members and Methods

Things to note:

- If we specify member variables, the values given are the default values for all instances.
- Methods must have a first parameter, conventionally called `self`. This is automatically filled in with a reference to the object the method was called on.

Creating Objects

Instantiating an object:

```
object_name = class_name()
```

Accessing members:

```
object_name.variable_name
```

Accessing methods:

```
object_name.function_name(param1, param2)
```

Example: Circles

```
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
```

Example: Circles

```
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: Rectangles

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 class_name:
    ... class definition ...

object_name = class_name()

object_name.variable_name

object_name.function_name(param1, param2)
```

Exercise: Rectangles

```
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

Manually setting the initial values of members is annoying and poor practice!

The special method `__init__` is automatically called when an object is created.

We can use this to perform initial setup of an object.

Initialisation

```
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 with Initialiser

Adapt your rectangle class to use an initialiser to give the width and height.

```
def __init__(self, param1, etc):
```

Exercise: Rectangle with Initialiser

```
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())
```

References

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, but can cause confusion.

References

```
x = Rectangle(2.0, 4.5)
y = x
x.width = 5.0
print(y.width)
```

Cloning an Object

A cloning method for `Rectangle`:

```
def clone(self):
    return Rectangle(self.width, self.height)
```

Creating a cloned copy:

```
x = Rectangle(2.0, 4.5)
y = x.clone()
x.width = 5.0
print(y.width)
```

References

```
class MyClass:
    mydata = Rectangle(1.0, 1.0)

    def change_width(self, width):
        self.mydata.width = width
```

References

```
class MyClass:
    mydata = Rectangle(1.0, 1.0)

    def change_width(self, width):
        self.mydata.width = width

x = MyClass()
y = MyClass()
x.change_width(8.0)
print(y.mydata.width)
```

Inheritance

Inheritance allows us to *derive* one object from another.

```
class child_class(parent_class):
    ... class definition ...
```

Example: Animals

Animals have many shared features, but individual animals have many differences.

We could define an `Animal` class containing the common features, then derive specialised classes for particular animals from it.

We could even have multiple layers of inheritance - Dogs, Cats, Hamsters, etc are all Mammals; Snakes, Lizards, etc are all Reptiles; and Mammals, Reptiles, etc are all Animals.

Example: Animals

```
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))
```

Redefined methods replace parent methods.

We use *`parent_class.function_name`* to access the parent's version.

Exercise: Zoo

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!

Summary

- 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.