Chapter 7

Classes and Object Orientation

Object Orientation is a programming paradigm that gained incredible importance in the last decades. Many programming languages (Java being the most popular) put Object Orientation in centre stage. Python also includes primitives for OOP (Object Oriented Programming).

OOP structured programs so that properties and behaviors are bundled into individual objects.

For instance, an object could represent a person with a name property, age, address, etc., with behaviors like walking, talking, breathing, and running. Or an email with properties like recipient list, subject, body, etc., and behaviors like adding attachments and sending.

7.1 Classes

Classes are used to create new user-defined data structures that contain arbitrary information about something. In the case of an animal, we could create an Animal() class to track properties about the Animal like the name and age.

It's important to note that a class just provides structure—it's a blueprint for how something should be defined, but it doesn't actually provide any real content itself. The Animal() class may specify that the name and age are necessary for defining an animal, but it will not actually state what a specific animal's name or age is.

It may help to think of a class as an idea for how something should be defined.

```
class Dog:

    # Class Attribute
    species = 'mammal'

    # Initializer / Instance Attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # instance method
    def description(self):
        return "{} is {} years old".format(self.name, self.age)

    # instance method
    def speak(self, sound):
        return "{} says {}".format(self.name, sound)

# Instantiate the Dog object
mikey = Dog("Mikey", 6)

# call our instance methods
print(mikey.description())
print(mikey.speak("Gruff Gruff"))
```

Figure 7.1: Class example

7.2 Methods and instance variables

In classes, properties are defined in instance variables (also called atributes). Behaviour is defined in methods. In the example below we can see the definition of class Dog with attributes like species and methods like description and speak. We then define an object of the class Dog called monkey. We can then call methods on the objects with the . operator (this is the same operator in almost all OOP languages)

7.3 Inheritance

An important concept in OOP is Inheritance. Classes can be extended to subclasses. These inherit all the attributes and methods of the parent class.

```
  # Parent class
class Dog:

    # Class attribute
    species = 'mammal'

    # Initializer / Instance attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # instance method
    def description(self):
        return "{} is {} years old".format(self.name, self.age)

    # instance method
    def speak(self, sound):
        return "{} says {}".format(self.name, sound)


# Child class (inherits from Dog() class)
class RussellTerrier(Dog):
    def run(self, speed):
        return "{} runs {}".format(self.name, speed)


# Child class (inherits from Dog() class)
class Bulldog(Dog):
    def run(self, speed):
        return "{} runs {}".format(self.name, speed)


# Child classes inherit attributes and
# behaviors from the parent class
jim = Bulldog("Jim", 12)
print(jim.description())

# Child classes have specific attributes
# and behaviors as well
print(jim.run("slowly"))

# Is jim an instance of Dog()?
print(isinstance(jim, Dog))

# Is julie an instance of Dog()?
julie = Dog("Julie", 100)
print(isinstance(julie, Dog))

# Is johnny walker an instance of Bulldog()
johnnywalker = RussellTerrier("Johnny Walker", 4)
print(isinstance(johnnywalker, Bulldog))

# Is julie and instance of jim?
print(isinstance(julie, jim))
```

7.4 Exceptions

Programs can have two kinds of errors. Syntax errors when programmers did not obey to the right syntax (e.g. print instead of print). The second type are **Exceptions**. Exceptions happen whenever syntactically correct Python code results in an error. For example when opening a non-existing file, the program will raise an exception.

7.4.1 Catching Exceptions

Good programs try to catch as many exceptions as possible. This is done with the **try** and **except**.

```
try:
    with open('file.log') as file:
        read_data = file.read()
except FileNotFoundError as fnf_error:
    print(fnf_error)
```

To know which exceptions you can catch you should check the Python documents (https://docs.python.org/3/library/exceptions.html)

You can also catch more than one exception at the same time:

```
 import math

number_list = [10,-5,1.2,'apple']

for number in number_list:
    try:
        number_factorial = math.factorial(number)
    except TypeError:
        print("Factorial is not supported for given input type.")
    except ValueError:
        print("Factorial only accepts positive integer values.", number," is not a pos
    else:
        print("The factorial of",number,"is", number_factorial)
    finally:
        print("Release any resources in use.")
```

7.4.2 Throwing exceptions

Although not a usual thing to do, your program can also raise exceptions:

```
x = 10
if x > 5:
    raise Exception('x should not exceed 5. The value of x was: {}'.format(x))
```

7.4.3 Exercises

1. Define a class **network** with a connectivity matrix of the nodes as an attribute and a method that calculates the shortest path between two given nodes using the Dijkstra algorithm. It should return the list of nodes of the path. One suggestion on how to store the connectivity matrix is to use nested lists.

2. Define a subclass of network called **tree**. A tree is a network without any possibility for loops. It should have a new attribute root and a method depth that calculates the depth of the tree. This method can use the shortest path method.