3 Days Training on Python3

Day 2 : Module 6

Muhammad Saufy Rohmad

Module 6 (90 minutes)

Objectives

1. Introduction to Object Orientation
2. Python Classes
3. Class Site and Static Behaviour

1. Introduction to Object Orientation

- A class is one of the basic building blocks of Python
- It is also a core concept in a style of programming known as Object Oriented Programming (or OOP)
- OOP provides an approach to structuring programs/applications so that the data held, and the operations performed on that data, are bundled together into classes and accessed via objects.
- As an example, in an OOP style program, employees might be represented by a class Employee where each employee has an id, a name, a department and a desk_number etc. They might also have operations associated with them such astake_a_holiday() or get_paid().

1. Introduction to Object Orientation(2)

- In many cases classes are used to represent real world entities (such as employees) but they do not need to, they can also represent more abstract concepts such as a transaction between one person and another (for example an agreement to buy a meal).
- Classes act as templates which are used to construct instances or examples of a class of things
- It would be extremely time-consuming (as well as inefficient) for programmers to define each object individually. Instead, they define classes and create instances or objects of those classes

1. Introduction to Object Orientation(3)

- In Python classes are used:
  - as a template to create instances (or objects) of that class,
  - define instance methods or common behaviour for a class of objects,
  - define attributes or fields to hold data within the objects,
  - be sent messages.

1. Introduction to Object Orientation(4)

- Objects (or instances), on the other hand, can:
  - be created from a class
  - hold their own values for instance variables,
  - be sent messages
  - execute instance methods
  - may have many copies in the system (all with their own data).

1.1 Class Terminology

- The following terms are used in Python (and other languages that support object orientation):
  - Class
  - Instance or Objects
  - Attributes/field/instance variable
  - Method
  - Message

2. Python Classes

- In Python everything is an object and as such is an example of a type or class of things.
- For example, integers are an example of the int class, real numbers are examples of the float class etc.

```
print(type(4))
print(type(5.6))
print(type(True))
print(type('Ewan'))
print(type([1, 2, 3, 4]))
```

- This produce

```
<class ‘int>
<class ‘float’>
<class ‘bool’>
<class ‘str’>
<class ‘list’>
```

2.1 Class Definition

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

- Although this is not a hard and fast rule, it is common to define a class in a file named after that class.
- There is also a special method defined called __init__. This is an initialiser (also known as a constructor) for the class. It indicates what data must be supplied when an instance of the Person class is created and how that data is stored internally.
- self is the object itself.

2.1 Class Definition(2)

```
p1 = Person('John', 36)
p2 = Person('Phoebe', 21)
```

- Object attributes access using

```
print(p1.name, 'is', p1.age)
print(p2.name, 'is', p2.age)
```

- We can also update it directly

```
p1.name = 'Bob'
p1.age = 54
```

2.1 Class Definition(3)

- Default string representation : def __str__(self)

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def __str__(self):
        return self.name + ' is ' + str(self.age)
```

2.1 Class Definition(4)

- If we now try to print out p1 and p2:

```
print(p1)
print(p2)
```

- It produce

```
Saufy is 38
Nurul is 35
```

2.2 Class Comment

```
class Person:
    """ An example class to hold a persons name and age"""
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def __str__(self):
        return self.name + ' is ' + str(self.age)
```

- The docstring is accessible through the __doc__ attribute of the class. The intention is to make information available to users of the class, even at runtime.

2.3 Adding Methods

```
class Person:
""" An example class to hold a persons name and age"""
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def __str__(self):
        return self.name + ' is ' + str(self.age)
    def birthday(self):
        print ( ' Happy birthday you were', self.age)
        self.age += 1
        print('You are now', self.age)
```

2.4 Delete Keyword

```
p1 = Person('John', 36)
print(p1)
del p1
```

- After the del statement the object held by p1 will no longer be available and any attempt to reference it will generate an error.

2.5 Automatic Memory Management

- The creation and deletion of objects (and their associated memory) is managed by the Python Memory Manager.
- Indeed, the provision of a memory manager (also known as automatic memory management) is one of Python’s advantages when compared to languages such as C and C++.
- The point about high level languages, however, is that they are more productive,introduce fewer errors, are more expressive and are efficient enough (given modern computers and compiler technology)
- If the system automatically handles the allocation and deallocation of memory, then the programmer can concentrate on the application logic.
- Python therefore provides automatic memory management. Essentially, it allocates a portion of memory as and when required. When memory is short, it looks for areas which are no longer referenced. These areas of memory are then freed up (deallocated) so that they can be reallocated. This process is often referred to as Garbage Collection.

2.6 Instrinsic Attributes

- Every class (and every object) in Python has a set of intrinsic attributes set up by the Python runtime system. Some of these intrinsic attributes are given below for classes and objects.
- Classes have the following intrinsic attributes:
  - __name__ the name of the class
  - __module__ the module (or library) from which it was loaded
  - __bases__ a collection of its base classes (see inheritance later in this book)
  - __dict__ a dictionary (a set of key-value pairs) containing all the attributes
  - (including methods)
  - __doc__ the documentation string.
- For objects:
  - __class__ the name of the class of the object
  - __dict__ a dictionary containing all the object’s attributes.

3. Class Side and Static Behaviour

- Python classes can hold data and behaviour that is not part of an instance or object. instead they are part of the class.
- Class Side Data

```
class Person:
""" An example class to hold a persons name and age"""
    instance_count = 0
    def __init__(self, name, age):
        Person.instance_count += 1
        self.name = name
        self.age = age
```

3. Class Side and Static Behaviour(2)

- Run with:

```
p1 = Person('Jason', 36)
p2 = Person('Carol', 21)
p3 = Person('James', 19)
p4 = Person('Tom', 31)
print(Person.instance_count)
```

3.1 Class Side Methods

```
class Person:
    """ An example class to hold a persons name and age"""
    instance_count = 0
    @classmethod
    def increment_instance_count(cls):
        cls.instance_count += 1
    def __init__(self, name, age):
        Person.increment_instance_count()
        self.name = name
        self.age = age
```

3.2 Static Method

```
class Person:
    @staticmethod
    def static_function():
        print('Static method')
```

- Static methods are invoked via the name of the class they are defined

```
    Person.static_function()
```