Programming

with python™

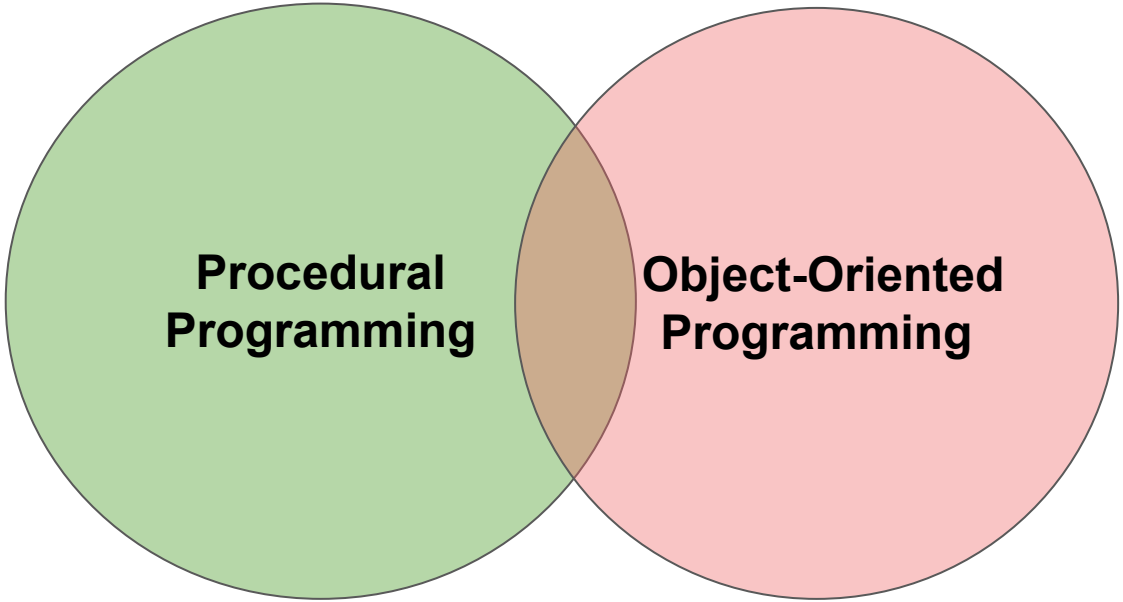
By
Rami Tailakh
Senior Software Engineer and Data Science Practitioner
MSc in Applied Computing and Information Technology

Day-11&12 Agenda

Object Oriented Programming in Python

Python Coding Styles

- Python is unlike Java and C++ which are languages built only for Object Oriented Programming
- Python can do both procedural and object-oriented programming



Procedural Coding Styles

- **Procedural** programming is a programming paradigm, in which, a set of commands are executed in order

- Tasks are treated as step-by-step iterations

- Common tasks are placed in functions that are called as required

- Python is one of the programming languages that excels in implementing such a coding style

Object Oriented Programming (OOP)

- **OOP** is a programming paradigm which provides a means of structuring programs

- It is useful to represent something much more complicated rather than just using the primitive data structures available in Python, such as numbers, strings, and lists

- Objects are a representation of real world objects; Someone/Something/Process is represented as an object

Class and Objects

- A **class** is a blueprint (or template) for creating objects; it contains all the names of details about a person, animal, thing or process
- A class in python is defined using the ‘class’ keyword
- A Python class may have:
    - attributes (identifying properties)
    - methods (identifying behavior)

Class and Objects/Continue

- An **object** is a unique instance of a data structure defined by its class. In other words, it is a “copy” of what is known as a class
- Think of a class as an idea while an object is its execution
- An object is created using the constructor of the class
- An object has two characteristics:
    - Properties (data)
    - behavior

Class and Objects/Continue

- Cars are an example of objects. Cars have data, such as year make, number of doors, capacity (etc.). Cars also can accelerate, stop, show speed meter, show fuel meter, and so many other behaviour.

- While the Car class is the model where we can define its attributes; year make, number of doors, and methods; accelerate, stop, show speed meter, show fuel meter.

Class Parts

- A **constructor** is a special method of a class. It is used for initializing an object. Initialization is to assign values to the data members of the class once an object of class is created. The **___init___()** method is called the constructor and is always called when an object is created
- An **attribute** represents the name of a property
- A **method** represents what behaviour can be performed to get/change the value of a property. It can be defined just like the Python functions using the **'def'** keyword

Class Parts/Continue

- In Python there are two types of variables:
  - **Class variables**
  - **Instance (or Object) variables**

- In Python there are three types of methods:
  - **Instance methods**
  - **Class methods**
  - **Static methods**

Naming Rules

- A class name should comply with a naming convention
- A class name should match with the following regex pattern:

```
^[A-Z][a-zA-Z0-9]*$
```

- E.g.:
    - Car >> Compliant
    - car >> Non-Compliant
- A class name must also follow general naming conventions in Python

Basic Example of a Class in Python

```
class Customer:
    """Docstring: Describe here what the class for in addition
to giving description about attributes
    """

    # Constructor/Initializer
    def __init__(self, customer_id, full_name, id_card_number, email, address):
        self.customer_id = customer_id
        self.full_name = full_name
        self.id_card_number = id_card_number
        self.email = email
        self.address = address
```

Attributes

- In Python, there two types of attributes: **class** and **instance (or object) attributes**
- An instance attribute is a variable only belonging an object
- An instance variable is only accessible in the scope of an object
- An instance variable is defined inside the constructor function, __init__(self,..) of a class
- A class attribute is a Python variable that belongs to a class (more is presented in the slide “Static Members - Variables”)

Instance Methods

- It is owned by the instance (object) of a class
- For each object, instance variables are different.
- It does not require decorators when defined
- A class method takes a mandatory argument "**self**"
- The class method can be called by its object

Adding an Instance Method to a Class

```
class Customer:
    def __init__(self, customer_id, full_name, id_card_number, email, address,
mobile_numbers=[]):
        self.id = customer_id
        self.first_name = full_name
        self.id_card_number = id_card_number
        self.email = email
        self.address = address
        self.mobile_numbers = mobile_numbers

    def add_mobile_number(self, mobile_number):
        # what could be added here to improve this method?
        self.mobile_numbers.append(mobile_number)
```

Adding an Instance Method to a Class/Continue

```
class Customer:
    def __init__(self, customer_id, full_name, id_card_number, email, address,
mobile_numbers=[]):
        self.customer_id = customer_id
        self.full_name = full_name
        self.id_card_number = id_card_number
        self.email = email
        self.address = address
        self.mobile_numbers = mobile_numbers

    def add_mobile_number(self, mobile_number):
        if mobile_number not in self.mobile_numbers:
            self.mobile_numbers.append(mobile_number)

    def describe(self):
        # you can add all other attributes
        return 'Customer Name: ' + self.full_name + ', Email: ' + self.email
```

Creating an Object

- The class itself is nothing without an execution; **Object**
- An object can be created by calling the class name with the required arguments and assigning it to a variable
- When creating an object, the **___init___()** method is called implicitly
- Here is we pass values for the attributes of the Customer class as follows:

```
>>> customer = Customer('A-0001', 'Sophie', 'Black', '92923942394', 's.b@test.ps')
```

The self Keyword

- The **self** keyword represents the instance of the class
- It is to access the attributes and methods of the class in python
- Its use is similar to the use of this keyword in Java, that is, it gives a reference to the current object
- When **object.method**(arg1, arg2, ...) is called, it is implicitly converted as follows:

```
Class.method(object, arg1, arg2, ...)
```

Accessing an Object

- An object attribute can be directly accessed as follows:

```
>>> customer.full_name
```

- An object attribute can be directly modified as follows:

```
>>> customer.email = 's.black@test.ps'
>>> customer.email
```

- BUT .. Is the above (directly accessing attributes) a good practice? We should do it through methods as follows:

  - Adding new values using the add_mobile_number method:

```
>>> customer.add_mobile_number('0599111222')
>>> customer.mobile_numbers
```

  - Call a method that shows data of an object:

```
>>> customer.describe()
```

Deleting Object Properties

- Properties on objects can be deleted by using the **del** keyword
- Note that this does not empty its value but rather it deletes its reference
- This can be done as follows:

```
>>> del customer.email
```

Static Members

- In Python, a static data member (variable) or method can be declared.

- Static means, that the member is on a class level rather only on the instance level

- Static members are not required to be instantiated

Static Members - Variables

- It is also known as **Class Attributes**
- In Python, we can declare attributes that belong to a class itself (defined on the class level)
- They are considered static, that is, they are declared inside the class definition, but not inside a method definition.
- It is **not** setting a "default value" for an object attribute. Rather, it can be expressed as a value shared among objects rather than being only owned by a specific object.
- If a static variable is changed in one instance of the class, the change will affect its value in all other instances. That is, it is shared by all objects
- It could be more useful in inheritance and inherited classes

Static Members - Variables

- Class attributes can be called directly without the need to create an object as follows:

```
>>> <ClassName>.<variable_name>
```

- It is declared and called, for instance, as follows:

```
>>> class Test:
        i = 10
        .
        .

>>>> Test.i
```

Static Members - Methods

- With methods, static definition differs from variables

- In Python, there are two ways of defining methods within a class:
  - Static method using the decorator **@staticmethod**
  - Class method using the decorator **@classmethod**

Static Members - Methods *@classmethod*

- With **@classmethod** decorator, a method is bound to a class rather than its object
- A class method takes a mandatory argument "**cls**", so, it works with the class as it always has the class itself
- A class method can access and modify class state
- The class method can be called both by the class and its object

Static Members - Methods *@staticmethod*

- With **@staticmethod** decorator, the method is only shared with the class namespace

- No arguments are mandatory in the method definition

- Static method can be used to just access and deal with a class static variables

- A static method, knows nothing about the class.

Accessing Methods

Let's practice the following example by experiencing how to call each method:

```
class TestClass:
    def instancemethod(self):
        return 'instance method was called', self

    @classmethod
    def classmethod(cls):
        return 'class method was called', cls

    @staticmethod
    def staticmethod():
        return 'static method was called'
```

Accessing Methods/Continue

- Calling the instance method

```
>>> obj = TestClass()

>>> obj.instancemethod()
# OR
>>> TestClass.instancemethod(obj)
```

- What if we call it directly by the class namespace?

```
>>> TestClass.instancemethod(obj)
```

Accessing Methods/Continue

- Calling the **class method**

```
>>> TestClass.classmethod()
# OR
>>> obj.classmethod()
```

Accessing Methods/Continue

- Calling the **static method**

```
>>> TestClass.staticmethod()
# OR
>>> obj.staticmethod()
```

- It can be seen that static methods have no access to the attributes of an instance of a class (like an instance method does). They also no have access to the attributes of the class itself (like a class method does), so, what is the use of static methods?

Encapsulation

- Encapsulation is one of the fundamental concepts in OOP
- It is about the idea of wrapping data and the methods within one unit, thus, it puts restrictions on accessing variables and methods.
- It helps prevent accidental (not intentional) modification of data; It could restrict how to modify data through methods.
- There are 3 types of such modifiers:
  - Public
  - Protected
  - Private

Encapsulation/Continue

- The **protected** members of a class, in JAVA, are those members which cannot be accessed outside the class but can be accessed from within the class and its subclasses.

- In Python, there is no existence of the the private modifier like Java has. However, a private member can defined by following the convention; by prefixing the name of the member by a double underscore “__”

Encapsulation/Continue

- The **private** members of a class, in JAVA, are those members which can neither be accessed outside the class nor by any of its subclasses.

- This can be accomplished in Python by following the convention; by prefixing the name of the member by a double underscore “__”

Encapsulation/Continue

```
>>> class Test:
        def __init__(self):
            self.a = 1
            self._b = 2
            self.__c = 3

>>> obj = Test()
>>> print(obj.a)
>>> print(obj._b)
>>> print(obj.__c) # todo: what to add to access the private member?
```

Encapsulation/Continue

```
>>> class Test:
        def __init__(self):
                self.a = 1
                self._b = 2
                self.__c = 3
        def show_c(self):
          return self.__c


>>> obj = Test()
>>> print(obj.a)
>>> print(obj._b)
>>> print(obj.show_c()) # Todo: How to change c?
```

Encapsulation/Continue

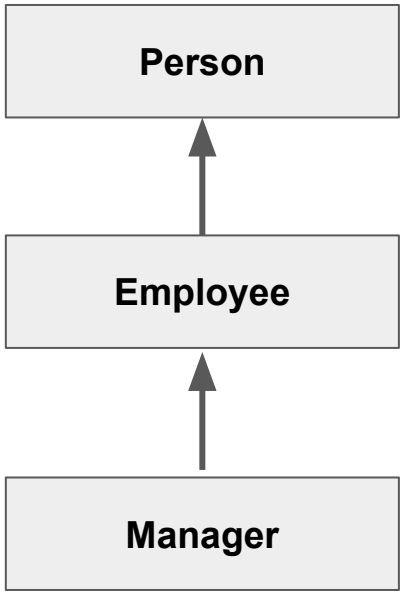
```
>>> class Test:
        def __init__(self):
                self.a = 1
                self._b = 2
                self.__c = 3
        def show_c(self):
            return self.__c
        def change_c(self, new_c):
             self.__c = new_c

>>> obj = Test()
>>> print(obj.a)
>>> print(obj._b)
>>> obj.change_c(30)
>>> print(obj.show_c())
```

Inheritance

- Inheritance is to allow defining a class that inherits all the properties and methods from another class, so we have parent and child classes
- **Parent** class is the class being inherited from, also called base class.
- **Child** class is the class that inherits from another class, also called derived class.

Inheritance/Continue

- Think of it as the relation between two entities in life, e.g.: Every manager is an employee. However, not every employee is a manger.
- Likewise, every employee is a person, but not Every person is an employee.
- In the mentioned representation, the figure shows how the relation is. It can be seen that we use an arrow towards the base class.



Inheritance/Continue

- In code, we make a class inherit from another by applying the name of the base class in parentheses to the definition of the derived class as follows:

```
>>> class Person:
                pass

>>> class Employee(Person):
                Pass

>>> class Manager(Employee):
                pass
```

Class Name of an Instance

- In Python, in order to get the class name of an object, we execute the following:

```
>>> type(<ClassName>).__name__
```

E.g.:

```
>>> type(customer).__name__
```

Challenges

1. Create an empty constructor for the Customer class, so we can just do the following instead of passing values when creating an object:
   >>> customer = Customer()
2. Create a Calculator class with basic mathematical operations: + - × ÷.
3. Create a class that accepts different shapes: circle, square, rectangle, triangle, and enables to calculate the area of each.
4. For the customer class, restrict accessing the instance attributes in addition to add some validation on instance methods as you see required.