Maktab
Sharif

Python | Main course

Session 7 & 8

- OOP in python
- Class vs. Instance (methods & attrs)
- Data hiding
- getters & setters
- Inheritance

by Mohammad Amin H.B. Tehrani

www.maktabsharif.ir

Object-Oriented Programing

In Python

Class

A user-defined prototype for an object that defines a set of attributes that characterize any object of the class.

➔ Create a class, which is like a **blueprint** for creating an object

Syntax

```
class ClassName:

    ...
```

```
class Square:
    x = 10
    y = 20
    ...
```

```
class Student:
    name = ‘Akbar’
    marks = []
    ...
```

Instantiate an Object in Python

Maktab Sharif

Instance: An individual object of a certain class. An object obj that belongs to a class Circle, for example, is an instance of the class Circle

Creating a new object from a class is called instantiating an object. You can instantiate a new object by typing the name of the class, followed by opening and closing parentheses:

Syntax

`ins = ClassName(...)`

```
class Square:
     x = 10
     y = 20
     ...

s = Square()
```

```
class Student:
     name = ‘Akbar’
     marks = []
     ...

S = Student()
```

Instance/Object Attributes (fields)

Maktab Sharif

> An **instance/object** attribute is a variable that belongs to one (and only one) object. Every instance of a class points to its own attributes variables.

```
class Human:
    first_name = ...
    last_name = ...
    age: int
    gender: str
    height: int
    ...
```

```
class Car:
    brand: str

    def __init__(self):
        self.model = ...
        self.color = ...
        self.fuel = ...
```

Instance/Object Methods

Methods are functions defined inside the body of a class. They are used to define the **behaviors** of an object.
A method is a function that “belongs to” an object.

```
class Human:
    name = ...

    def sleep(self, time):
        ...

    def eat(self, food):
        ...
```

```
class Car:
    speed = ...

    def start(self):
        ...

    def brake(self):
        ...
```

Initialize object (Constructor)

Method: __init__(self, ...)

__init__ is one of the reserved methods in Python. In object oriented programming, it is known as a constructor. The __init__ method can be called when an object is created from the class, and access is required to initialize the attributes of the class.

```
class Human:

    def __init__(self, first_name, last_name, **extra_information):
        self.name = first_name + last_name
        self.extra_info = extra_information


akbar = Human('Akbar', 'Rezaii', age=25, height=168)
```

Example

```
class Square:

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def area(self):
        return self.x * self.y


s = Square(2, 5)
print(s.area())
```

Example

```
class Square:

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def area(self):
        return self.x * self.y


s = Square(2, 5)
print(s.area())
```

Self keyword

Maktab Sharif

The **self** parameter is a reference to the **current instance** of the class, and is used to access variables that belongs to the class.
It does not have to be named self , you can call it whatever you like, but it has to be the first parameter of any function in the class

```
class MyClass:

    def __init__(self, a, ...):
        self.my_attr = a
        self.my_method()
        ...

    def my_method(my_self, ... ):  # Call as my_self
        my_self...
```

Class vs. Instance (methods & attrs)

Class vs. Instance attributes

- **An instance attribute** is a Python variable **belonging to one, and only one, object.** This variable is only accessible in the scope of this object and it is defined **inside** the constructor function, __init__(self,..) of the class.

- **A class attribute** is a Python variable that **belongs to a class rather than a particular object.** It is shared between all the objects of this class and it is defined **outside** the constructor function.

Example (1 of 3)

Class definition:

```
class MyClass:
    class_attr = "It's a class Attribute!"

    def __init__(self, x=None):
        self.my_attr = x or "It's my Attribute!"

    def some_method(self):
        self.class_attr = "MY class_attr modified!"

    def another_method(self):
        MyClass.class_attr = "class_attr modified!"
```

Example (2 of 3)

Instantiation:

```
ins1 = MyClass()
ins2 = MyClass("It's ins2 attribute!")

print(MyClass.class_attr, '', sep='\n')
print(ins1.class_attr, ins2.class_attr, '', sep='\n')

print(ins1.my_attr, ins2.my_attr, '', sep='\n')
print(id(MyClass.class_attr), id(ins1.class_attr), id(ins2.class_attr), end='\n\n')

ins1.some_method()
ins2.another_method()

print(ins1.class_attr, ins2.class_attr, '', sep='\n')
print(id(MyClass.class_attr), id(ins1.class_attr), id(ins2.class_attr), end='\n\n')

print(MyClass.my_attr)
```

Example (3 of 3)

output:

```
It's a class Attribute!

It's a class Attribute!
It's a class Attribute!

It's my Attribute!
It's ins2 attribute!

2366069311088 2366069311088 2366069311088

MY class_attr modified!
class_attr modified!

2366069443360 2366069444160 2366069443360

AttributeError...
```

Class vs. Instance Methods

The idea of class method is very similar to instance method, only difference being that instead of passing the instance hiddenly as a first parameter, we’re now passing the class itself as a first parameter.

Syntax:

```
@classmethod
def method_name(cls, …):
    cls.class_attribute...
    cls.class_method(...)
    …
```

cls **parameter is a reference to the class itself.**
like self in instance methods, you can rename it, but it’s NOT recommended!

Example (1 of 3)

Class definition:

```
class MyClass:
    class_attr = "It's a class Attribute!"

    def __init__(self, x=None):
        self.my_attr = x or "It's my Attribute!"

    def instance_method(self):
        self.class_attr = "MY class_attr modified!"

    @classmethod
    def class_method(cls):
        cls.class_attr = "class_attr modified!"
```

Example (2 of 3)

Instantiation:

```
ins1 = MyClass()
ins2 = MyClass("It's ins2 attribute!")

print(MyClass.class_attr, '', sep='\n')
print(ins1.class_attr, ins2.class_attr, '', sep='\n')

print(ins1.my_attr, ins2.my_attr, '', sep='\n')
print(id(MyClass.class_attr), id(ins1.class_attr), id(ins2.class_attr), end='\n\n')

ins1.instance_method()
ins2.class_method()

print(ins1.class_attr, ins2.class_attr, '', sep='\n')
print(id(MyClass.class_attr), id(ins1.class_attr), id(ins2.class_attr), end='\n\n')

print(MyClass.my_attr)
```

Example (3 of 3)

output:

```
It's a class Attribute!

It's a class Attribute!
It's a class Attribute!

It's my Attribute!
It's ins2 attribute!

2366069311088 2366069311088 2366069311088

MY class_attr modified!
class_attr modified!

2366069443360 2366069444160 2366069443360

AttributeError...
```

Static Methods

A **static** method does **not receive an implicit first argument.**

- A static method is also a method which is bound to the class and not the object of the class.
- A static method can’t access or modify class state.
- It is present in a class because it makes sense for the method to be present in class.

Syntax:

```
@staticmethod
def method_name(...):
    ...
```

We generally use static methods to create utility functions.

Data hiding

Example

Let's start with an example:

```
class User:

    def __init__(self, username, password):
        self.username = username
        self.password = password


# Registering
akbar = User('akbar_rezaii', '!SD2&84!WASd')

# Observing akbar’s password by a bad staff!!
print("It's akbar’s password:", akbar.password)
```

Public members

Public members (generally methods declared in a class) are accessible from outside the class. The object of the same class is required to invoke a public method. This arrangement of private instance variables and public methods ensures the principle of data encapsulation.

All members in a Python class are public by default. Any member can be accessed from outside the class environment.

You can access the `Student` class's attributes and also modify their values:

```
class Student:
    schoolName = 'XYZ School'

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

```
>>> std = Student("Steve", 25)
>>> std.schoolName
'XYZ School'
>>> std.name
'Steve'
>>> std.age = 20
>>> std.age
20
```

Private members

Python doesn't have any mechanism that effectively restricts access to any instance variable or method. Python prescribes a convention of prefixing the name of the variable/method with a single or double underscore to emulate the behavior of protected and private access specifiers.

The double underscore __ prefixed to a variable makes it private. It gives a strong suggestion not to touch it from outside the class. Any attempt to do so will result in an AttributeError:

```
class Student:
    __schoolName = 'XYZ School' # private class attribute
    def __init__(self, name, age):
        self.__name=name  # private instance attribute
        self.__salary=age # private instance attribute
    def __display(self):  # private method
        print('This is private method.')
```

Example: Fixed

Make password attribute private:

```
class User:

    def __init__(self, username, password):
        self.username = username
        self.__password = password


# Registering
akbar = User('akbar_rezaii', '!SD2&84!WASd')

# Now it raises an AttributeError:
print("It's akbar's password:", akbar.__password)
```

Protected members

Protected members of a class are accessible from within the class and are also available to its sub-classes. No other environment is permitted access to it. This enables specific resources of the parent class to be inherited by the child class.

Python's convention to make an instance variable protected is to add a prefix _ (single underscore) to it. This effectively prevents it from being accessed unless it is from within a sub-class.

```
class User:

    def __init__(self, *args):
        self._father_name = 'akbar'
```

```
class Student(User):

    def some_method(self):
        print(f'{self._father_name=}')
```

```
u = User()
s = Student()

s.some_method()

print(u._father_name)   # ???
print(s._father_name)   # ???
```

Getter & Setter

Example

Let's start with an example again:

```
class Square:

    def __init__(self, x, y):
        self.x, self.y = x, y

    def area(self):
        return self.x * self.y


ins = Square(2, 10)
print(ins.area())

ins.x = '2'  # small mistake!
print(ins.area())  # !!!
```

Getters & Setters

We use getters & setters to add validation logic around getting and setting a value.
To avoid direct access of a class field i.e. private variables cannot be accessed directly or modified by external user.
Using normal function to achieve getters and setters behaviour.

```
class MyClass:

    __attr: ...     # Private attribute

    def set_attr(self, _):  # Setter method
        self.__attr = _

    def get_attr(self):     # Getter method
        return self.__attr
```

Example: Fixed

Let's start with an example again:

```
class Square:

    def __init__(self, x, y):
        self.__x, self.__y = x, y

    def set_x(self, x):
        self.__x = float(x)

    def set_y(self, y):
        self.__y = float(y)

    def get_xy(self):
        return self.__x, self.__y
```

Inheritance

Inheritance

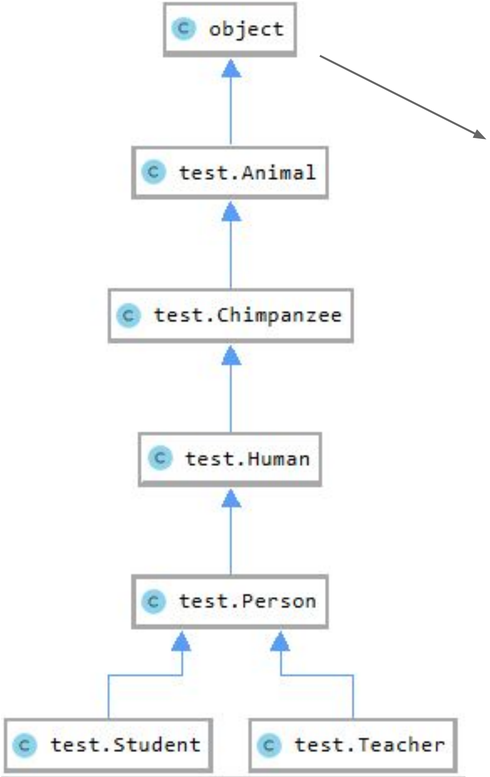
Inheritance allows us to define a class that inherits all the methods and properties from another class.

- **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**.

```
# Create a class (Parent class)
class Person:
    ...

# Create child class:
class Student(Person):
    ...
```

Example: Human evolution in python

```
class Animal:
    pass

class Chimpanzee(Animal):
    pass

class Human(Chimpanzee):
    pass

class Person(Human):
    pass

class Student(Person):
    pass

class Teacher(Person):
    pass
```



Every class in python Inherits from **object** class.

Data hiding in derived class

```
class MyParentClass:

    def __init__(self):
        self.public_attr = "It's PUBLIC"
        self._protected_attr = "It's PROTECTED"
        self.__private_attr = "It's PRIVATE"

    def print_attributes(self):
        print(self.public_attr)
        print(self._protected_attr)
        print(self.__private_attr)
        print()
```

```
class MyChildClass(MyParentClass):
    def some_method(self):
        self.public_attr = "Modifying PUBLIC attr"
        self._protected_attr = "Modifying PROTECTED attr"
        self.__private_attr = "Modifying PROTECTED attr"
```

Data hiding in derived class

```
parent_ins = MyParentClass()
child_ins = MyChildClass()

parent_ins.print_attributes()
child_ins.print_attributes()

child_ins.some_method()
child_ins.print_attributes()

parent_ins.some_method()
```

Data hiding in derived class

```
parent_ins = MyParentClass()
child_ins = MyChildClass()

parent_ins.print_attributes()
child_ins.print_attributes()

child_ins.some_method()
child_ins.print_attributes()

parent_ins.some_method()
```

```
It's PUBLIC
It's PROTECTED
It's PRIVATE

It's PUBLIC
It's PROTECTED
It's PRIVATE

Modifying PUBLIC attr
Modifying PROTECTED attr
It's PRIVATE


AttributeError: ...
```

Method overriding

Method overriding is a concept of object oriented programming that allows us to change the implementation of a method in the child class that is defined in the parent class.
It is the ability of a child class to change the implementation of any method which is already provided by one of its parent class(ancestors).

Super:

The super() function is used to give access to methods and properties of a parent or sibling class.

The super() function returns an object that represents the parent class.

Example: Method overriding

```python
class MyParentClass:

    def __init__(self, name):
        print("(ParentClass > __init__)")
        self._name = name

    def welcome(self):
        print("(ParentClass > some_method)")
        return f'Hello {self._name}!'
```

```python
class MyChildClass(MyParentClass):

    def __init__(self, name='Akbar'):
        super().__init__('Mr. ' + name)
        print("(ChildClass > __init__)")

    def welcome(self):
        print("(ChildClass > some_method)")
        return super().welcome()
```

What’s Output of code below:

```python
print('Parent Instantiation:')
parent_ins = MyParentClass('Reza')
print('\nParent Welcoming:')
print(parent_ins.welcome())
```

```python
print('Child Instantiation:')
child_ins = MyChildClass()
print('\nChild Welcoming:')
print(child_ins.welcome())
```

Predefined methods overriding

Some of most important predefined method:

__delattr__	__dir__	**__eq__**	__format__	**__str__**
__getattribute__	**__ge__**	**__gt__**	__hash__	__init_subclass__
__le__	**__lt__**	__new__	**__ne__**	__init__
__reduce_ex__	__reduce__	**__repr__**	__setattr__	__sizeof__

Pre-reading

Search about:

1. Property in python
2. Multi inheritance in python
3. Decorator in python
4. getattr, setattr and delattr functions