Maktab Sharif

Python | Main course

Session 6

- Review
- Data hiding
- getters & setters
- Inheritance

by Mohammad Amin H.B. Tehrani - Reza Yazdani

www.maktabsharif.ir

Review

Basic of Python Contents

Maktab Sharif

1) Variables
2) Types
3) Operators
4) String
5) Data structures (List, Dict, Tuple, Set)
6) Conditional Statements (if .. else ..)
7) Loops (While, For)
8) Function
9) Built-in functions: map, sorted, filter, …
10) List comprehension (inline for), Trenary expression (inline if), lambda

OOP Contents

Maktab Sharif

1) Class
2) Object
3) Instance
4) Attributes
5) Methods
6) Fundamental (Hierarchy, Encapsulation, Abstraction, Inheritance, Polymorphism)
7) Initialize object
8) Self keyword
9) Static Methods

Data hiding

Maktab Sharif

Example

Maktab Sharif

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

Maktab Sharif

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

Maktab Sharif

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

Maktab Sharif

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

Maktab Sharif

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

Maktab Sharif

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

Maktab Sharif

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

Maktab Sharif

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

Maktab Sharif

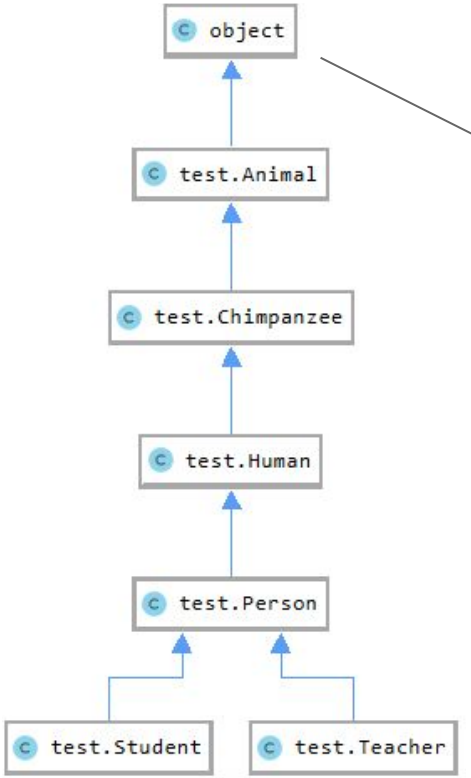
Inheritance allows us to define a class that inherits all the methods and properties from another class.

- **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

Maktab Sharif

```
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

Maktab Sharif

```
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

Maktab Sharif

```
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

Maktab Sharif

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

Maktab Sharif

```
class MyParentClass:

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

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

```
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:

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

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

Pre-reading

Search about:

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

Maktab Sharif