Maktab Sharif

Python | Main course

Session 7

- Magic methods
- Multi-Inheritance & MRO
- Composition vs Inheritance
- Abstraction
- Polymorphism
- Advanced

by Mohammad Amin H.B. Tehrani - Reza Yazdani

www.maktabsharif.ir

Magic methods

Maktab Sharif

Magic methods

Maktab Sharif

Python - Magic or Dunder Methods

Magic methods in Python are the special methods that start and end with the double underscores. They are also called dunder methods. Magic methods are not meant to be invoked directly by you, but the invocation happens internally from the class on a certain action.

Starts and ends with double underscores (__)

__abs__	__dir__	**__eq__**	__format__	**__str__**
__and__	**__ge__**	**__gt__**	__bool__	__init_subclass__
__le__	**__lt__**	__new__	**__ne__**	__init__
__or__	__reduce__	**__repr__**	__setattr__	__sizeof__

Dir Function

Maktab Sharif

Objects properties and methods list: dir(...)

he **dir()** function returns all properties and methods of the specified object, without the values.

```
i = 10
print(dir(i))
```

```
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__',
'__delattr__', '__dir__', '__divmod__', '__doc__', '__eq__', '__float__',
'__floor__', '__floordiv__', '__format__', '__ge__', '__getattribute__',
'__getnewargs__', '__gt__', '__hash__', '__index__', '__init__',
'__init_subclass__', '__int__', '__invert__', '__le__', '__lshift__',
'__lt__', '__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__or__',
'__pos__', '__pow__', '__radd__', '__rand__', ...
```

___init___: Initialize

Maktab Sharif

The __init__ method is similar to constructors in C++ and Java. Constructors are used to initialize the object's state. The task of constructors is to initialize(assign values) to the data members of the class when an object of class is created.

Magic method: __init__()
Usage: instance = MyClass(...)

```
class User:

    def __init__(self, id, first_name, last_name):
        self.id = id
        self.first_name = first_name
        self.last_name = last_name
```

```
u1 = User(1, 'Akbar', 'Rezaii')
print(u1.id, u1.first_name, u1.last_name)
```

output:

```
1 Akbar Rezaii
```

___repr___ : Representation

Maktab Sharif

The repr() function returns a printable representation of the given object.
Internally, repr()function calls __repr__() of the given object.

Magic method: __repr__()
Usage: repr(my_obj)

```
class User:

  def __init__(self, id, first_name, last_name):
      self.id = id
      self.first_name = first_name
      self.last_name = last_name

  def __repr__(self):
      return f"<User #{self.id}>"
```

```
u1 = User(1, 'Akbar', 'Rezaii')
print(repr(u1))
print(u1)
```

output:

```
<User #1>
<User #1>
```

___str___ : String

Maktab Sharif

The __str__ method is useful for a string representation of the object, either when someone codes in str(your_object), or even when someone might do print(your_object).

Magic method: __str__()

Usage: str(my_obj)

```
class User:

    def __init__(self, id, first_name, last_name):
        self.id = id
        self.first_name = first_name
        self.last_name = last_name

    def __str__(self):
        return f"<User #{self.id}:" \
            f" {self.first_name} {self.last_name}>"
```

```
u1 = User(1, 'Akbar', 'Rezaii')
print(repr(u1))
print(str(u1))
print(u1)
```

output:

```
<__main__.User object at ...>
<User #1: Akbar Rezaii>
<User #1: Akbar Rezaii>
```

___eq___: Equal

Maktab Sharif

Uses on compare the object with another object.
Exactly on == operator

Magic method: __eq__()
Usage: my_object == other

```
class User:

  def __init__(self, id, first_name, last_name):
      self.id = id
      self.first_name = first_name
      self.last_name = last_name

  def __eq__(self, other):
      return self.id == other.id
```

```
u1 = User(1, 'Akbar', 'Rezaii')
u2 = User(2, "Reza", 'Akbari')
u3 = User(1, 'Ahmad', 'Bagheri')
print(u1 == u1, u1 == u2, u1 == u3)
```

output:

```
True False True
```

___del___: Delete (del)

Maktab Sharif

del keyword

The del keyword is used to delete objects. In Python everything is an object, so the del keyword can also be used to delete variables, lists, or parts of a list etc.

The __del__() method is a known as a destructor method in Python. It is called when all references to the object have been deleted.

Magic method: __del__()
Usage: del my_object

Example ___del___: Delete (del)

Maktab Sharif

```
class User:
    NUM_OF_USERS = 0

    def __init__(self, id, first_name, last_name):
        self.id = id
        self.first_name = first_name
        self.last_name = last_name

        self.__class__.NUM_OF_USERS += 1

    def __del__(self):
        self.__class__.NUM_OF_USERS -= 1
```

```
print(User.NUM_OF_USERS)
u1 = User(1, 'Akbar', 'Rezaii')
u2 = User(2, 'Reza', 'Akbari')
print(User.NUM_OF_USERS)
del u1
print(User.NUM_OF_USERS)
del u2
print(User.NUM_OF_USERS)
```

output:

```
0
2
1
0
```

Some magic methods

Maktab Sharif

Magic method	Usage	Description
`__ne__`	`instance != other`	Not Equal
`__ge__`	`instance >= other`	Greater Equal
`__gt__`	`instance > other`	Greater Than
`__le__`	`instance < other`	Less Equal
`__lt__`	`instance <= other`	Less Than

Multi-Inheritance & MRO

Maktab Sharif

Multi inheritance

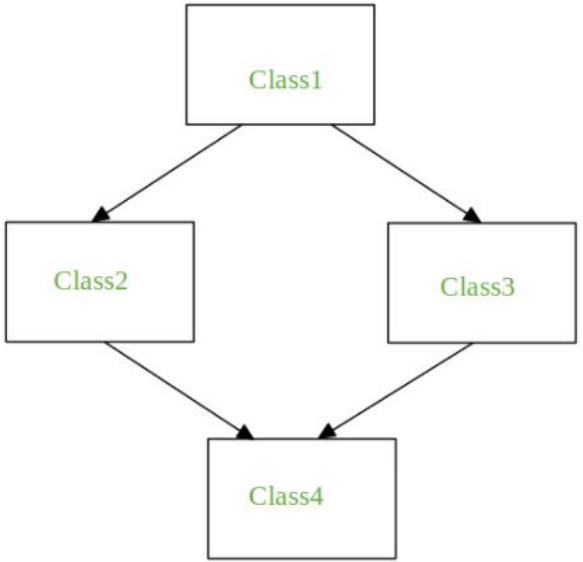
Maktab Sharif

Multi inheritance

When a class is derived from more than one base class it is called multiple Inheritance. The derived class inherits all the features of the base case.

Diamond problem

It refers to an ambiguity that arises when two classes Class2 and Class3 inherit from a superclass Class1 and class Class4 inherits from both Class2 and Class3



Example 1

Maktab Sharif

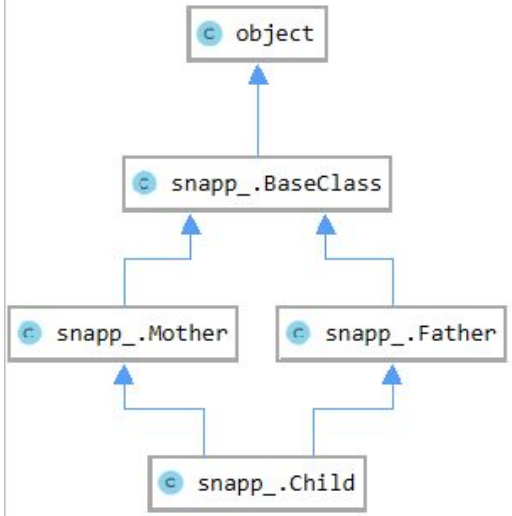
```
class BaseClass:
   def method(self):
       return "I'm in BaseClass"


class Father(BaseClass):
   def method(self):
        return "I'm in Father"


class Mother(BaseClass):
   def method(self):
       return "I'm in Mother"

class Child(Father, Mother):
   pass

c = Child()
print(c.method())
```



Example 1

Maktab Sharif

```
class BaseClass:
   def method(self):
       return "I'm in BaseClass"

class Father(BaseClass):
   def method(self):
        return "I'm in Father"

class Mother(BaseClass):
   def method(self):
       return "I'm in Mother"

class Child(Father, Mother):
   pass

c = Child()
print(c.method())
```

output:

```
I'm in Father
```

Example 2

Maktab Sharif

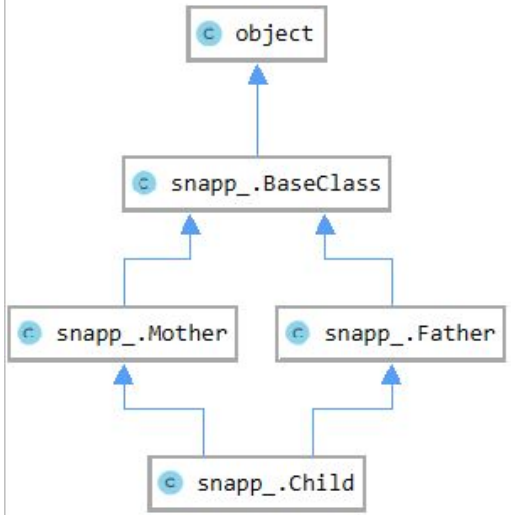
```
class BaseClass:
    def method(self):
        return "I'm in BaseClass"

class Father(BaseClass):
    def method(self):
        return super().method() + " -> Father"

class Mother(BaseClass):
    def method(self):
        return super().method() + " -> Mother"

class Child(Father, Mother):
    pass

c = Child()
print(c.method())
```



Example 2

Maktab Sharif

```
class BaseClass:
    def method(self):
        return "I'm in BaseClass"


class Father(BaseClass):
    def method(self):
        return super().method() + " -> Father"


class Mother(BaseClass):
    def method(self):
        return super().method() + " -> Mother"

class Child(Father, Mother):
    pass

c = Child()
print(c.method())
```

output:

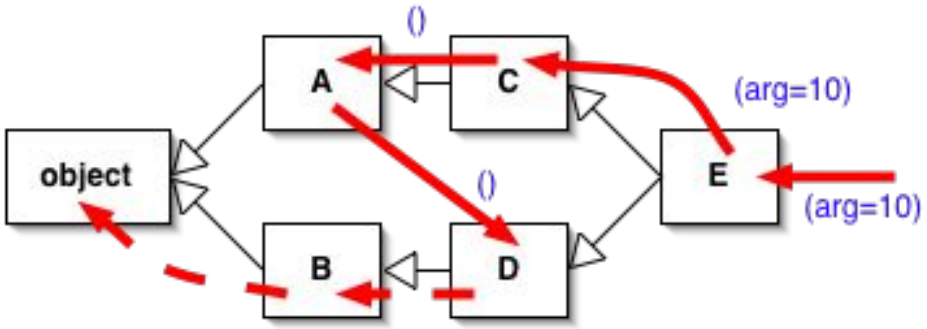
```
I'm in BaseClass -> Mother -> Father
```

MRO

Maktab Sharif

Method Resolution Order

Method Resolution Order(MRO) it denotes the way a programming language resolves a method or attribute. Python supports classes inheriting from other classes. The class being inherited is called the Parent or Superclass, while the class that inherits is called the Child or Subclass. In python, method resolution order defines the order in which the base classes are searched when executing a method.



Example

Maktab Sharif

```
class BaseClass:
  def method(self):
    return "I'm in BaseClass"


class Father(BaseClass):
  def method(self):
    return super().method() + " -> Father"


class Mother(BaseClass):
  def method(self):
    return super().method() + " -> Mother"

class Child(Father, Mother):
  pass

print(Child.mro())
```

output:

```
[<class '__main__.Child'>,
 <class '__main__.Father'>,
 <class '__main__.Mother'>,
 <class '__main__.BaseClass'>,
 <class 'object'>]
```

Mixin

Maktab Sharif

Mixin

In object-oriented programming languages, a mixin is a class that contains methods for use by other classes without having to be the parent class of those other classes. How those other classes gain access to the mixin's methods depends on the language.

```
class Human:
    pass

class InformationMixin:
    fist_name = 'Akbar'
    last_name = 'Rezanii'
    phone = '09367778889'
    email = 'akbar@gmail.com'

class User(Human, InformationMixin):
    pass
```

```
u = User()
print(u.fist_name, u.last_name, u.phone)
```

output:

```
Akbar Rezanii 09367778889
```

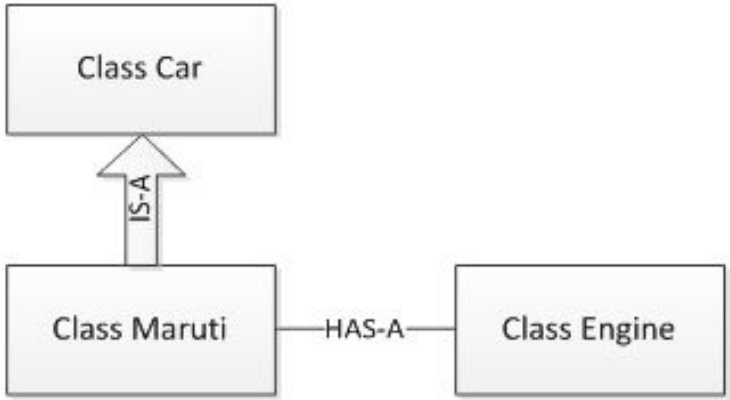
Composition vs. Inheritance

Composition

Maktab Sharif

Composition

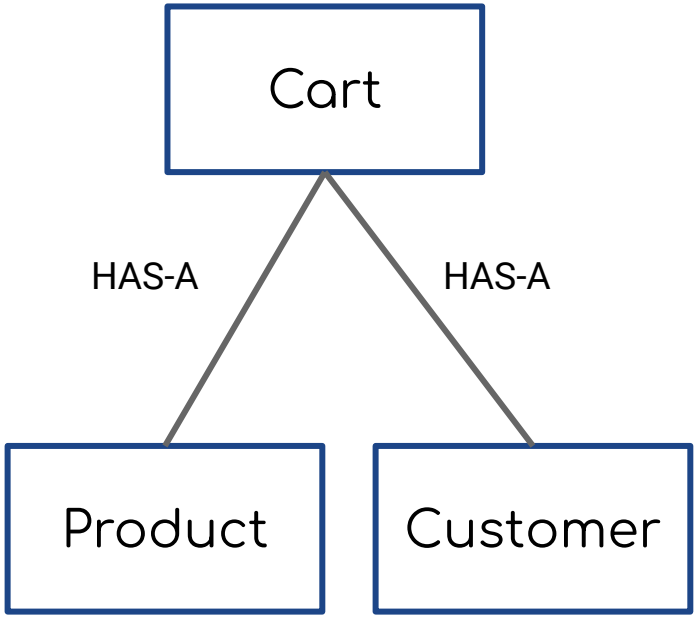
Composition is a concept that models a has a relationship. It enables creating complex types by combining objects of other types. This means that a class Composite can contain an object of another class Component. This relationship means that a Composite has a Component.



Example

```
class Product:
    id: int
    name: str
    company: str
    price: float


class Cart:
    customer: Customer    # Customer component
    orders: List[Product]    # Products component
```



Example

Maktab Sharif

```
class Product:
    id: int
    name: str
    company: str
    price: float
```

Now What's better in this case?

Composition

```
class Order:
    product: Product
    num: int
```

Inheritance

```
class Order(Product):
    num: int
```

Inheritance vs. Composition

Maktab Sharif

It's big confusing among most of the people that both the concepts are pointing to Code Reusability then what is the difference b/w Inheritance and Composition and when to use Inheritance and when to use Composition?

Inheritance is used where a class wants to derive the nature of parent class and then modify or extend the functionality of it. Inheritance will extend the functionality with extra features allows overriding of methods,
but in the case of **Composition**, we can only use that class we can not modify or extend the functionality of it. It will not provide extra features. Thus, when one needs to use the class as it without any modification, the composition is recommended and when one needs to change the behavior of the method in another class, then inheritance is recommended.

Example

Maktab Sharif

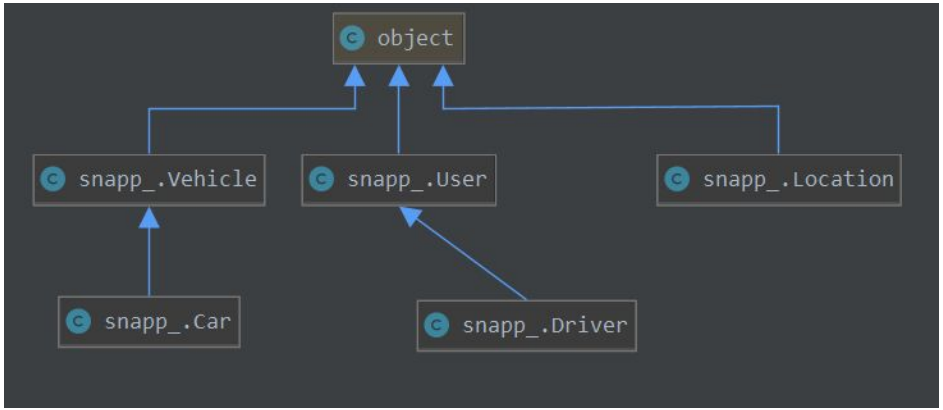
```
class Location:
  lat: float
  lng: float

class Vehicle:
  tag: str

class User:
  first_name: str
  last_name: str
  phone: str
  location: Location

class Car(Vehicle):
  model: str
  brand: str
  color: ...

class Driver(User):
  car: Car
```



Abstraction

Maktab Sharif

Abstract class

Abstract class

An abstract class can be considered as a blueprint for other classes. It allows you to create a set of methods that must be created within any child classes built from the abstract class. A class which contains one or more abstract methods is called an abstract class. An abstract method is a method that has a declaration but does not have an implementation. While we are designing large functional units we use an abstract class. When we want to provide a common interface for different implementations of a component, we use an abstract class.

Example

Maktab Sharif

```
from abc import ABC, abstractmethod

class Polygon(ABC):

    @abstractmethod
    def no_of_sides(self):
        pass
```

```
Triangle().no_of_sides()
Quadrilateral().no_of_sides()
Pentagon().no_of_sides()
Hexagon().no_of_sides()

p = Polygon()   # ???
p.no_of_ides() # ???
```

```
class Quadrilateral(Polygon):
    def no_of_sides(self):
        print("I have 4 sides")
```

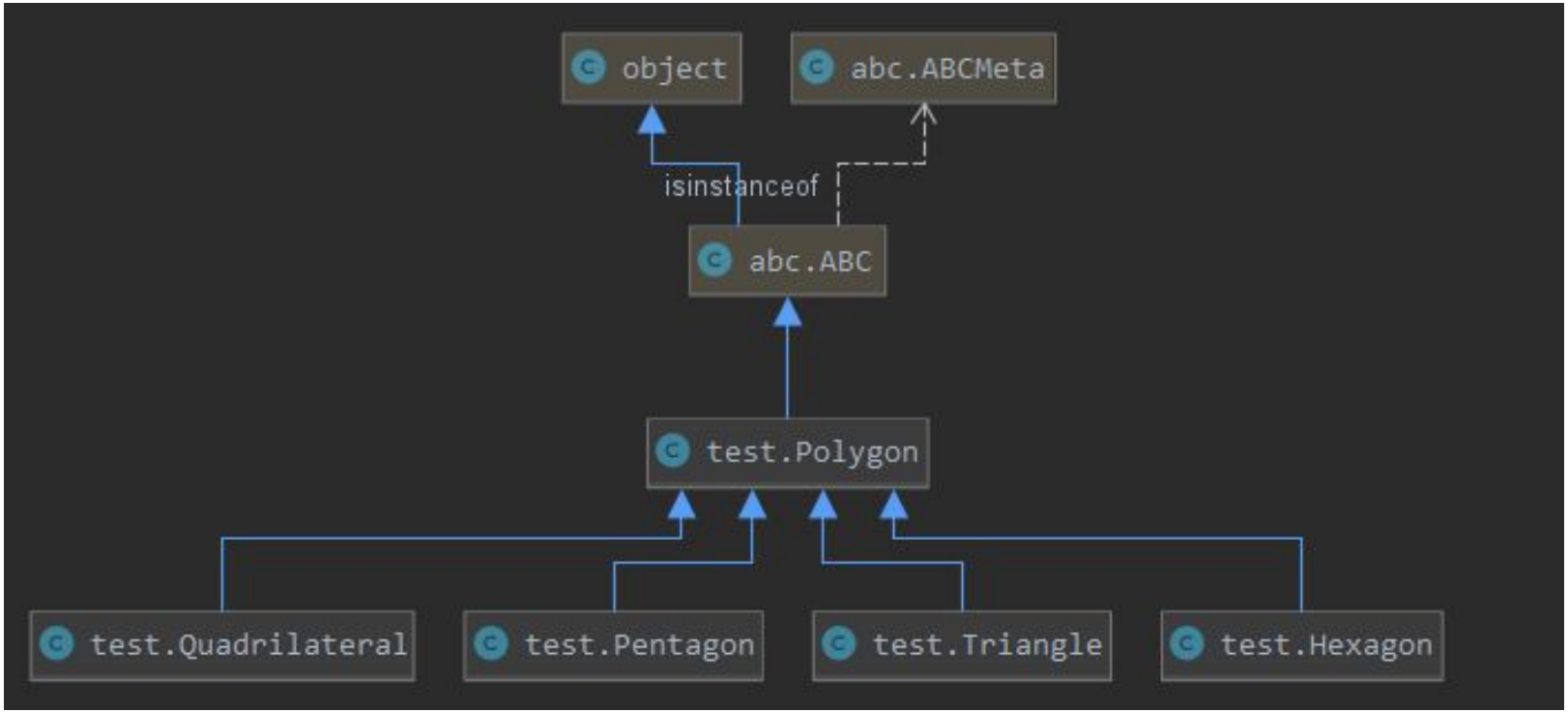
```
class Hexagon(Polygon):
    def no_of_sides(self):
        print("I have 6 sides")
```

```
class Pentagon(Polygon):
    def no_of_sides(self):
        print("I have 5 sides")
```

```
class Triangle(Polygon):
    def no_of_sides(self):
        print("I have 3 sides")
```

Example

Maktab Sharif



Polymorphism

Polymorphism

Polymorphism

The word polymorphism means having many forms. In programming, polymorphism means same function name (but different signatures) being uses for different types.

Polymorphism in python defines methods in the child class that have the same name as the methods in the parent class. In inheritance, the child class inherits the methods from the parent class. Also, it is possible to modify a method in a child class that it has inherited from the parent class.

- Functions Polymorphism
- Class Polymorphism

Functions Polymorphism

Maktab Sharif

There are some functions in Python which are compatible to run with multiple data types.

```
class Car:
   tag: str


class Driver:

   def __init__(self, car):
      if isinstance(car, Car):
         self.car_tag = car.tag
      elif isinstance(car, str):
         self.car_tag = car
      ...
```

Python does not support function overloading !

```
def surface(x, y, z=1):
   return x * y * z
```

```
def pow(x):
   return x ** 2
```

Class Polymorphism

Maktab Sharif

In Python, Polymorphism lets us define methods in the child class that have the same name as the methods in the parent class. In inheritance, the child class inherits the methods from the parent class. However, it is possible to modify a method in a child class that it has inherited from the parent class.

```
from abc import ABC, abstractmethod

class Polygon(ABC):

  @abstractmethod
  def no_of_sides(self):
    pass
```

```
Triangle().no_of_sides()
Quadrilateral().no_of_sides()
Pentagon().no_of_sides()
Hexagon().no_of_sides()
```

```
class Quadrilateral(Polygon):
  def no_of_sides(self):
    print("I have 4 sides")
```

```
class Pentagon(Polygon):
  def no_of_sides(self):
    print("I have 5 sides")
```

```
class Triangle(Polygon):
  def no_of_sides(self):
    print("I have 3 sides")
```

Advanced

- Dynamic class
- Properties
- Metaclass

Maktab Sharif

Pre-reading
For next session (8)

Search about:

1. Dynamic class
2. Properties
3. Metaclass
4. File in python
5. Logging in python

Maktab Sharif