

نشی گرایبی

هادی فرهادی
مهر ۱۴۰۴



شی گرای - OOP



h.farhadi.py@gmail.com



شی گرایی - OOP



برنامه نویسی شی گرا (Object-Oriented Programming) یک پارادایم برنامه نویسی است

که بر اساس مفهوم "کلاس" و "شیء" کار می کند.



هر شیء می تواند ویژگی ها - داده ها (attribute) و رفتارهای (method) مرتبط با خود را در یک واحد به

نام کلاس encapsulate کند.

به زبان ساده تر، مدل کردن چیزها-اشیا دنیای واقعی به کد است که هم ویژگی ها را در بر داشته باشد

و هم رفتارها را.

```
class Animal:  
    Pass
```

```
cat = Animal()
```





این جانب برد پیت هستم، 61 ساله، سفید پوست، 180 سانت قد و 75 کیلو وزن

پیاده روی می کنم. غذا می خورم، عصبانی می شوم، خوش حال می شوم





```
full_name = "Brad Pitt"
age = 61
skin_color = "White"
height = 180
weight = 75
mood = "Relax"
```

```
def reaction(mood):
    if mood == "Happy":
        print("Hooooorrrraaaaa!")
    elif mood == "Angry":
        print("#%$@ ****")
    elif mood == "Sad":
        print("Crying ...")
    else:
        print("Hey")
```

```
def walking(step):
    print(f"Walking ... {step} steps")
```





class Person:

```
def __init__(self, full_name: str, age: int, skin_color: str, height: int, weight: int, mood: str): # Constructor
    self.full_name = full_name # Attribute, Field, Property
    self.age = age # Attribute, Field, Property
    self.skin_color = skin_color # Attribute, Field, Property
    self.height = height # Attribute, Field, Property
    self.weight = weight # Attribute, Field, Property
    self.mood = mood # Attribute, Field, Property
```

```
def reaction(self): # Method
    if self.mood == "Happy":
        print("Hooooorrrrraaaaa!")
    elif self.mood == "Angry":
        print("F**k ****")
    elif self.mood == "Sad":
        print("Crying ...")
    else:
        print("Hey")
```

```
def walking(self, step): # Method
    print(f"{self.full_name} Walking ... {step} steps")
```

با استفاده از این کد یک نوع جدید ساختیم





شی گرایی - OOP



نامگذاری کلاس به صورت **پاسکال کیس** (PascalCase) است: حروف اول هر کلمه به صورت

بزرگ تایپ می شود.



با تعریف یک کلاس به نوع جدید همانند dict, set, list و ... ایجاد می کنیم.

برای استفاده از آن باید یک **شی** یا **instance** یا **object** از آن بسازیم.

```
number_list = list() # []
```

```
feature_set = set()
```

```
first_object = ClassName()
```



دسترسی به **attributes** و **methods** از طریق کاراکتر '.' انجام می شود





شی گرای - OOP



```
number_list = list() # []
```

```
number_list.append(10)
```



```
feature_set = set()
```

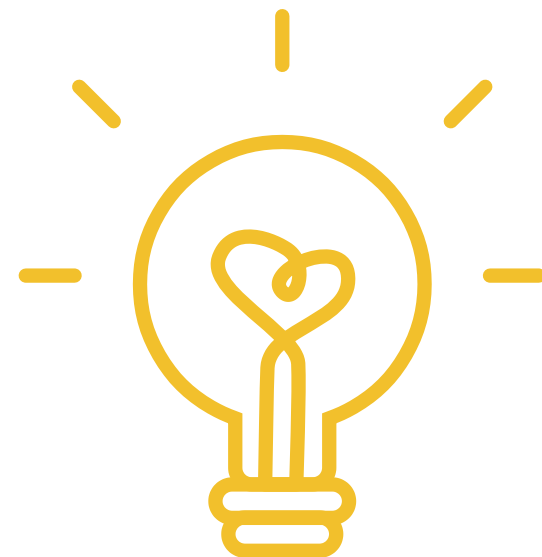
```
feature_set.add(12)
```



```
first_object = ClassName()
```

```
first_object.method_one()
```

```
print(first_object.attribute_one)
```





شی گرایى - OOP



```
rahim_sterling = Person("Rahim Sterling", 36, "Black", 177, 69, "Happy") # instantiate, Instance, Object
brad_pitt = Person(full_name="Brad Pitt", age=61, skin_color="White", height=180, weight=75, mood="Relax")

print(f'{brad_pitt.full_name}: {brad_pitt.age} years old, {brad_pitt.height} cm, {brad_pitt.weight} kg, {brad_pitt.mood}')

brad_pitt.walking(30)

jabar_sing = Person() # ?
```





هنگام ساخت یک شی از یک کلاس **همیشه** متدی به نام سازنده **constructor** فراخوانی می شود
همیشه **مقدار دهی های اولیه** را از این طریق انجام دهید

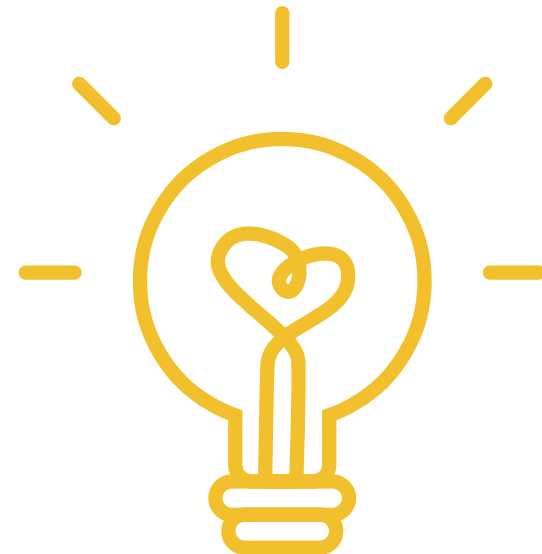


class ClassName:



```
def __init__(self, field_one: str, field_two: str):  
    self.field_one = field_one  
    self.field_two = field_two
```

...





تمرین: Person Class list vs Tuple list



h.farhadi.py@gmail.com



ارکان شی گرای





ارکان شی گرای



شی گرای 4 رکن اساسی دارد که اگر این چهار رکن را رعایت کنید:

قابلیت استفاده مجدد، قابلیت نگهداری، امنیت، انعطاف پذیری



(a) وراثت (Inheritance)



(b) کپسوله سازی (Encapsulation)

(c) چندریختی (Polymorphism)

(d) انتزاع (Abstraction)





وراثت (Inheritance)



وراثت یکی از پایه‌ای‌ترین مفاهیم شی‌گرایی است که به یک کلاس اجازه می‌دهد ویژگی‌ها و متدهای کلاس دیگر - والد را به ارث ببرد.

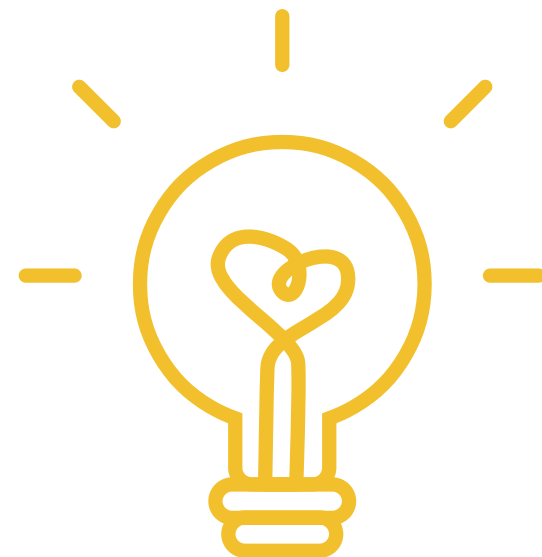


```
# Parent Class
class Animal:
    def __init__(self, name): # this one calls anyway, because we didn't define __init__ in Cat and Dog
        self.name = name
    def speak(self):
        pass

# Child Class
class Dog(Animal):
    def speak(self):
        return "Woof!"

# Another Child Class
class Cat(Animal):
    def speak(self):
        return "Meow!"

dog = Dog("Buddy")
cat = Cat("Whiskers")
print(dog.speak()) # Woof!
print(cat.speak()) # Meow!
```





کپسوله سازی (Encapsulation)



پنهان سازی داده ها و ارائه متدها کنترل شده برای دسترسی به آنها.



سطوح دسترسی:

- عمومی - **Public**: تمامی متدها و ویژگی های public در همه جا قابل دسترسی هستند.



- محافظت شده - **Protected**: تمامی متدها و ویژگی های protected تنها در خود کلاس

و کلاس های فرزند قابل دسترسی هستند. از یک - قبل از نام attribute و method استفاده می شود.

- خصوصی - **Private**: تمامی متدها و ویژگی های private تنها در خود کلاس قابل

دسترسی هستند. از دو - () قبل از نام attribute و method استفاده می شود.





کپسوله سازی (Encapsulation)



class Animal:

```
def __init__(self, name: str):  
    self.name = name # public  
    self._id = hash(self.name) # protected  
    self.__privacy = f"{self._id} {self.name}" # private
```

```
def show_reaction(self):  
    pass
```

```
def show_info(self):  
    print(f"{self.name}-{self._id}-{self.__privacy}: Woof!")
```

class Dog(Animal):

```
def __init__(self, name):  
    super().__init__(name)
```

```
def show_reaction(self):  
    print(f"{self.name}-{self._id}-{self.__privacy}: Woof!") # error
```

```
def __show_result(self):  
    print("Wow")
```

```
<def _call_me(self):  
    print("Dring")
```





کپسوله سازی (Encapsulation)

```
dog = Dog("Puppy")  
dog.show_reaction() # error
```

```
print(dog.name)  
print(dog._id)  
dog._call_me()  
dog.show_info() # show result  
dog.__show_result() # error
```





دکوراتور @property در پایتون



@property یک دکوراتور built-in در پایتون است که به شما اجازه می دهد متدهای کلاس



را مانند **attribute**های عادی فراخوانی کنید.

این ویژگی برای ایجاد **setters**، **getters** و **deleters** به روشی پایتونی استفاده می شود.





دکوراتور @property در پایتون

```
class Temperature:
    def __init__(self, celsius):
        self.__celsius = celsius

    @property
    def celsius(self):
        return self.__celsius

    @celsius.setter
    def celsius(self, value):
        if value < -273.15:
            raise ValueError("Temperature cannot be below absolute zero!")
        self.__celsius = value

    @property
    def fahrenheit(self):
        return (self.__celsius * 9/5) + 32

    @fahrenheit.setter
    def fahrenheit(self, value):
        self.celsius = (value - 32) * 5/9
```

```
temp = Temperature(25)
print(f"Celsius: {temp.celsius}") # 25
print(f"Fahrenheit: {temp.fahrenheit}") # 77.0
```

```
temp.fahrenheit = 100
print(f"New Celsius: {temp.celsius:.2f}") # 37.78
```





انتزاع (Abstraction)



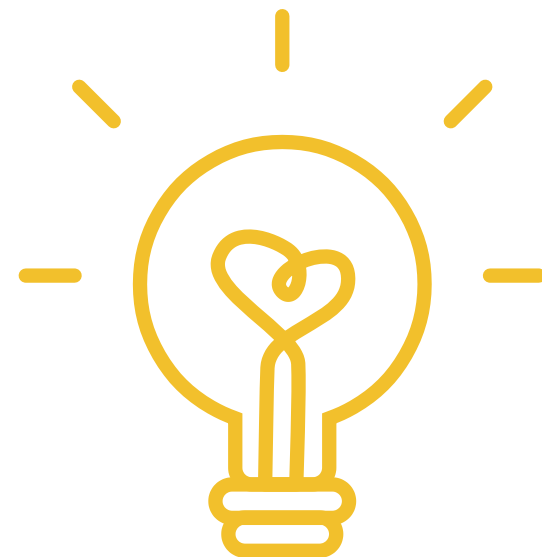
جزئیات پیچیده implementation را پنهان می کند ، فقط ویژگی ها و رفتارهای ضروری را نمایش می دهد.



آماده شو بریم - > انتزاع است برای مجموعه ای از کارها. 1. دست و صورت را بشور 2. لباسهایت را اتو کن 3. لباسهایت را بپوش 4. مسواک بزن و ...

از طریق `interface` یا `abstract class` قابل پیاده سازی است که پایتون به صورت پیش فرض آن ها را ندارد.

اما یک ماژول built-in به نام `abc` دارد که به ما در این زمینه کمک می کند





انتزاع (Abstraction)

```
from abc import ABC, abstractmethod
```

```
class Shape(ABC):
```

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

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

```
class Rectangle(Shape):
```

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

```
    def calculate_area(self): # must be implemented
        return self.x * self.y
```

```
    def show(self): # must be implemented
        print(f"Rectangle({self.x}, {self.y})")
```

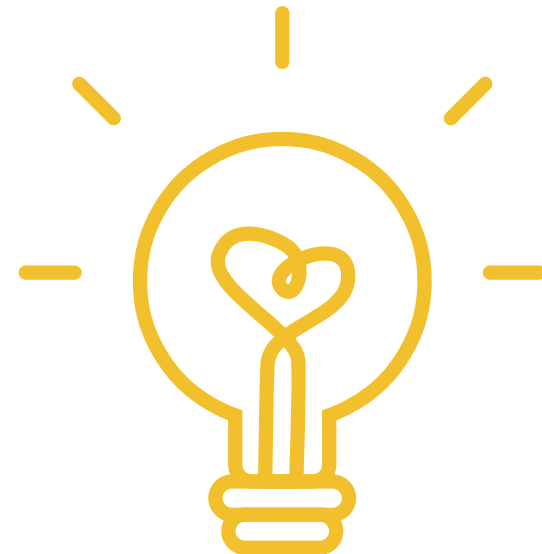
```
# shape = Shape() # Can't instantiate abstract class Shape
```

```
rectangle = Rectangle(10, 12)
```

```
print(f"Rectangle({rectangle.x}, {rectangle.y}) Area = {rectangle.calculate_area()}")
```

```
rectangle.show()
```

```
print(rectangle)
```





چندریختی (Polymorphism)



چندریختی یکی از مفاهیم اصلی در برنامه نویسی شیءگرا است که به معنی "چندشکلی" می باشد.

این مفهوم به ما اجازه می دهد که از یک رابط یکسان برای انواع مختلف اشیاء استفاده کنیم.



فرض کنید می خواهیم انواع مختلف وسیله های نقلیه را در یک کارخانه بسازیم. همه باید ویژگی گاز دادن

و ترمز را داشته باشند. پس در کلاس پایه وسایل نقلیه ما دو متد یا رفتار به نام های گاز دادن و ترمز کردن

داریم. اما وسایل مختلفی از این کلاس پایه ارث بری می کنند مثلاً ماشین، قطار، هواپیما، موتور. هرکدام

از این وسایل قاعده تاً پیاده سازی های مختلفی را برای این دو متد دارند.

پس هر وسیله نقلیه برای گاز دادن و ترمز کردن، **شکل متفاوتی** را پیاده سازی کرده است





چندریختی (Polymorphism)

```
from abc import ABC, abstractmethod
```

```
class Animal(ABC):
```

```
    @abstractmethod
```

```
    def make_sound(self):
```

```
        pass
```

```
    def move(self):
```

```
        return "Moving around"
```

```
class Dog(Animal):
```

```
    def make_sound(self):
```

```
        return "Woof! Woof!"
```

```
    def move(self):
```

```
        return "Running on four legs"
```

```
class Cat(Animal):
```

```
    def make_sound(self):
```

```
        return "Meow! Meow!"
```

```
    def move(self):
```

```
        return "Walking silently"
```

```
class Bird(Animal):
```

```
    def make_sound(self):
```

```
        return "Chirp! Chirp!"
```

```
    def move(self):
```

```
        return "Flying in the sky"
```





چندریختی (Polymorphism)

```
class Fish(Animal):  
    def make_sound(self):  
        return "Blub blub"  
  
    def move(self):  
        return "Swimming in water"
```

```
def animal_concert(animals): # different animal has the same method make sound and move  
    for animal in animals:  
        print(f"Sound: {animal.make_sound()}")  
        print(f"Movement: {animal.move()}")  
    print("-" * 30)
```

```
animals = [Dog(), Cat(), Bird(), Fish()]  
animal_concert(animals)
```





Composition





ترکیب (Composition)

```
class Engine:
    def start(self):
        return "Engine started"

class Wheels:
    def rotate(self):
        return "Wheels rotating"

class Car:
    def __init__(self):
        self.engine = Engine() # composite
        self.wheels = Wheels() # composite

    def drive(self):
        return f"{self.engine.start()} and {self.wheels.rotate()}"

car = Car()
print(car.drive()) # Engine started and Wheels rotating
```

خیلی وقت ها به اشتباه به جای composition از inheritance استفاده می شود





Method Resolution Order (MRO)





Method Resolution Order (MRO)



```
class A:  
    def method(self):  
        return "A"
```

```
class B(A):  
    def method(self):  
        return "B"
```

```
class C(A):  
    def method(self):  
        return "C"
```

```
class D(B, C):  
    Pass
```

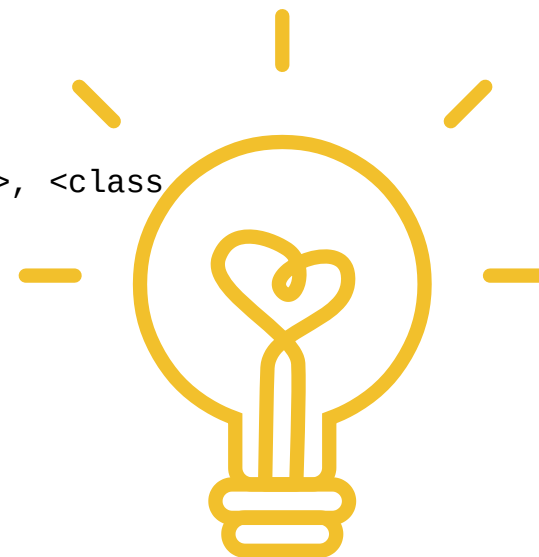
```
# Check MRO
```

```
print(D.__mro__)
```

```
# Output: (<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class  
'__main__.A'>, <class 'object'>)
```

```
obj = D()
```

```
print(obj.method()) # ?
```





انواع متدها






انواع متدها



کلاس ها سه نوع متد دارند



- **Class method**: روی کلاس کار می کنند اولین پارامتر آن ها cls است با دکوریتور `@classmethod` تعریف می شوند. 

- **Instance method**: روی Instance ایجاد شده از کلاس کار می کنند اولین پارامتر آن `self` است.

- **Static method**: به کلاس یا instance وابسته نیستند، با دکوراتور `@staticmethod` تعریف می شوند و پارامتر **خاصی** دریافت نمی کنند.





انواع متدها

```
class Animal:
    can_walk = True

    def __init__(self, name: str, age: int):
        self.name = name
        self.age = age

    def speak(self):
        print(f'{self.name} says: Woof {self.can_walk}')
        self.info()

    @classmethod
    def run(cls):
        print(f'animals can walk: {cls.can_walk}')

        # cat = cls("Cat", 8)
        # cat.speak()

    @staticmethod
    def info():
        print("All animal can generate sound")
```

ادامه در صفحه بعد





انواع متدها



```
dog = Animal("Dog", 12)
```

```
print("Instance access".center(100, "-"))
```

```
dog.speak()
```

```
dog.run()
```

```
dog.info()
```

```
print("Class access".center(100, "-"))
```

```
#Animal.speak() # error
```

```
Animal.run()
```

```
Animal.info()
```

```
print("Change a class variable access".center(100, "-"))
```

```
Animal.can_walk = False
```

```
dog.speak()
```

```
dog.run()
```

```
dog.info()
```

```
Animal.run()
```

```
Animal.info()
```

```
print("Change a class variable access with an instance".center(100, "-"))
```

```
dog.can_walk = True
```

```
dog.speak()
```

```
dog.run()
```

```
dog.info()
```

```
Animal.run()
```

```
Animal.info()
```





THANK YOU

h.farhadi.py@gmail.com