6. Object Oriented Programming (O.O.P.)

6.1 classes

6.1 classes

برنامه نویسی شیء گرا ابزاریست که برای ما قابلیت ساختاربندی متغیرها (اشیاء) را فراهم می کند

به این معنا که با استفاده از آن میتوانیم به متغیر ها خصوصیات، صفات، و ویژگی ها و وظایف دلخواه نسبت دهیم و متغیر ها را ساختار بندی کنیم

برای مثال میتوانیم کارمند های یک شرکت که هر یک دارای خصوصیات خود مانند نام، سمت، حقوق هستند،

یا انواع اتوموبیل که هر یک دارای سرعت، قیمت، سوخت مصرفی، توان و... هستند،

یا ذراتی که ویژگی های منحصر به فرد خود مانند سرعت، جرم، انرژی و ... دارد را به سادگی مدل سازی کنیم

شاید به نظر یک راه مناسب برای ساخت چنین متغیر هایی استفاده از لیست ها باشد به این شکل که به ترتیب هر ایندکس نمایانگر یکی از خصوصیات مورد نظر ما باشد

برای مثال در مورد مثال کارمندان یک شرکت، هر لیست نمایانگر یک کارمند باشد که عضو صفرم آن معرف نام، عضو دوم معرف سن و عضو سوم بیانگر حقوق کارمند و در نهایت سال ورود آن کارمند به شرکت باشد

```
emp1 = ["matin",22,10000000, 1401]
emp2 = ["kasra",15000000, 1398]
emp3 = ["sheyda",25, 2200000, 1398]
```

و یا همچنین استفاده از یک دیکشنری

```
emp1 = {"name":"matin", "age“:22, "salary":10000000,"entrance year":1401}
emp2 = {"name":"kasra", "salary":15000000,"entrance year":1398}
emp3 = {"name":"sheyda", "age“:25, "salary":22000000,"entrance year":1398}
```

اما این روش یک سری مشکلات به بار خواهد آورد

اما این روش یک سری مشکلات به بار خواهد آورد

1. مدیریت کد ها و پروژه های بزرگ را سخت کند
2. طولانی بودن نوشتاری تعریف اینچنین متغیر ها
3. دشواری به یاد داشتن این موضوع که هر کدام از ایندکس های لیست معرف چه پارامتری هستند
4. در صورت ناقص بودن داده ها امکان دارد به مشکل برخورد کنیم(برای مثال در مثال اسلاید قبل، کارمند دوم بر خلاف دیگران دارای متغیر age نیست و اگر ایندکس 1 لیست مربوطه را فراخوانی کنیم به جای سن کارمند، حقوق آن را مشاهده خواهیم کرد)

```
emp1 = ["matin",22,10000000, 1401]
emp2 = ["kasra", ?   15000000, 1398]
emp3 = ["sheyda",25, 22000000, 1398]
```

Emp2[1] → 15000000

کلاس ها (class) راه حل مناسب را در اختیار ما قرار می‌دهند

1. با استفاده از کلاس ها از ساختارمند بودن و منظم بودن داده های یک متغیر مطمین می‌شویم
2. دسترسی راحت تری به داده ها خواهیم داشت
3. امکان تعریف آسان خصوصیت های متعدد (attributes) و همچنین تعریف رفتار و توابع منحصر به فرد برای متغیر(یا شیء) ها (objects) وجود دارد (یعنی متغیر های ما، خود میتوانند دارای توابع مختص خود باشند) که به این توابع (methods) می‌گویند
4. امکان ناقص بودن داده ها با وجود کلاس ها وجود ندارد و به این ترتیب مشکل چهارم در اسلاید قبل از بین می‌رود و اگر داده ای ناقص وارد شود، برنامه خطا داده و ما را آگاه میکند
   بدین ترتیب متعیر ما دقیقا مطابق میل ما و به دور از هرگونه اشکال ساخته خواهد شد

در ادامه به نحوه تعریف کلاس ها و نمونه ها و تفاوت این دو مقوله می‌پردازیم

یک کلاس طرحی اولیه است از اینکه چگونه یک شیء (object) باید تعریف شود.

در واقع در هنگام تعریف، یک کلاس هیچ داده ای ندارد و صرفا به تعریف ساختار دلخواه خود می‌پردازیم و در ادامه برنامه هنگام ساختن نمونه ها (instance) است که از داده ها استفاده می‌کنیم

در حالی که کلاس طرح اولیه است، یک نمونه (instance) یک شی (متغیر) است که از روی قالب یک کلاس ساخته شده و حاوی داده های واقعی است که طبق ساختاری که کلاس مشخص کرده وارد شده

به عبارت دیگر، یک کلاس مانند یک فرم یا پرسشنامه است. اما یک نمونه (instance) مانند فرمی است که با اطلاعات پر شده است. درست مانند بسیاری از افراد که می توانند یک فرم را با اطلاعات منحصر به فرد خود پر کنند، نمونه های زیادی نیز می توانند از یک کلاس ایجاد شوند.

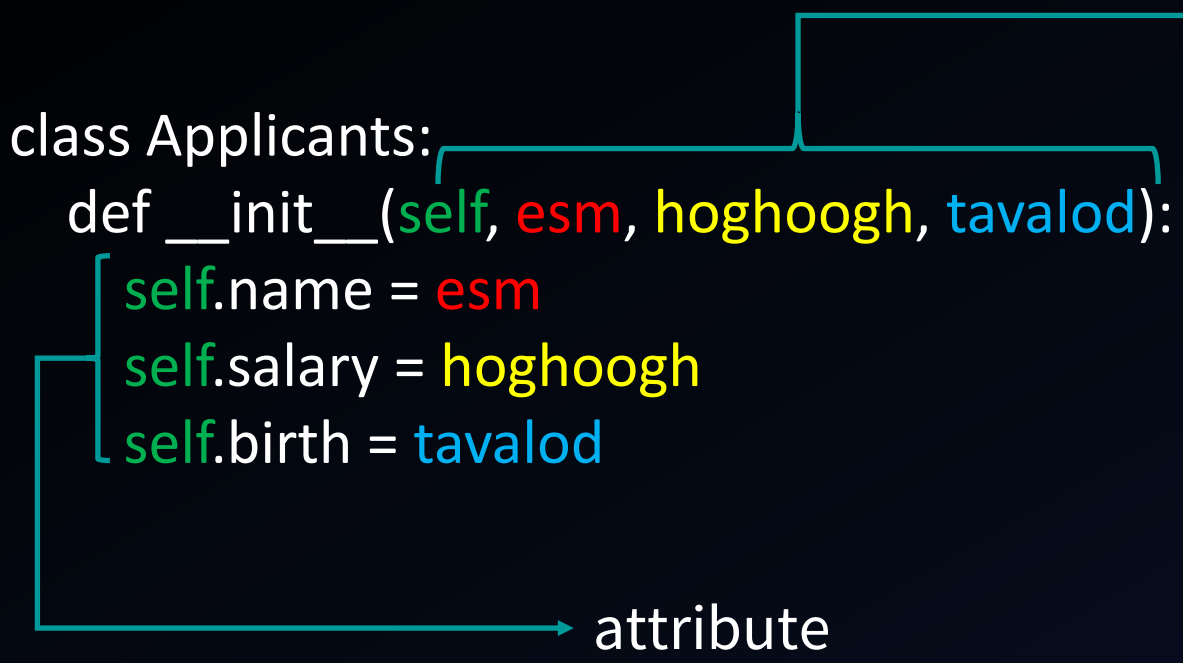
برای مثال فرض می‌کنیم که یک فرم استخدامی داریم که شامل نام، حقوق درخواستی و سال تولد است

class

attributes

و فرض می‌کنیم که تعداد ده نفر این فرم را طبق مشخصات و حقوق درخواستی خود پر کرده اند

instance



برای تعریف ویژگی های دلخواه کلاس خود، همواره از تابع __init__() استفاده میکنیم
بعد از پارامتر دهی، به تعریف ویژگی های ابتدایی کلاس میپردازیم

```
class Applicants:
    entrance_year = 1402
    name_of_organization = NASA

    def __init__(self, esm, hoghoogh, tavalod):
        self.name = esm
        self.salary = hoghoogh
        self.birth = tavalod
```

Class attributes

Instance attribute

برای تعریف ویژگی های دلخواه کلاس خود، همواره از تابع __init__() استفاده میکنیم
بعد از پارامتر دهی، به تعریف ویژگی های ابتدایی کلاس میپردازیم

```
class Applicants:
    entrance_year = 1402
    name_of_organization = NASA

   def __init__(self, esm, hoghoogh, tavalod):
      self.name = esm
      self.salary = hoghoogh
      self.birth = tavalod
```

برای ساختن نمونه ها کافیست تا به این شکل آرگومان های مورد نیاز شیء مورد نظر را وارد کنیم

```
matin = Applicants("Matin",22000000 , 1379)
kasra = Applicants("Kasra",10000000 , 1381)
sheyda = Applicants("Sheyda",15000000 , 1380)
```

همچنین می‌توانیم برای ساختن نمونه ها از کاربر ورودی گرفته و هرتعداد نمونه که نیاز بود
بدون تعریف دستی آنها و با استفاده از حلقه ها بسازیم

```
class Applicants:
    entrance_year = 1402
    name_of_organization = "NASA"

    def __init__(self, esm, hoghoogh, tavalod):
        self.name = esm
        self.salary = int(hoghoogh)
        self.years = int(tavalod)

persons = []
for i in range(2):
    persons.append(Applicants(input(),input(),input()))
```

در این صورت نمونه ها به صورت خودکار تشکیل شده و با استفاده از لیست مورد نظر و
حلقه زدن در آن می‌توان به محتویات و اطلاعات نمونه ها دسترسی داشت

در این صورت نمونه ها به صورت خودکار تشکیل شده و با استفاده از لیست مورد نظر و حلقه زدن در آن می‌توان به محتویات و اطلاعات نمونه ها دسترسی داشت

```
persons = []
for i in range(2):
    persons.append(Applicants(input(),input(),input()))


for person in persons:
    print(person.name,person.salary)
```

یا حتی در بین نتایج جست و جو و اعمال فیلتر کرد:

یا حتی در بین نتایج جست و جو و اعمال فیلتر کرد:

```
persons = []
for i in range(2):
    persons.append(Applicants(input(),input(),input()))


for person in persons:
    if person.salary >= 5000000:
        print(person.name,person.salary)
```

و یا حتی مقادیر attribute ها را تغییر داد

و یا حتی مقادیر attribute ها را تغییر داد

```
persons = []
for i in range(2):
    persons.append(Applicants(input(),input(),input()))


for person in persons:
    if person.salary <= 5000000:
        person.salary = 1000000
        print( f"{person.name} had a salary under 5 millions, now it's {person.salary}")
```

6.2 Instance and class Methods

6.2.1 Instance Methods

متدهای نمونه (Instance Methods)، توابعی هستند که در داخل یک کلاس تعریف می‌شوند و فقط نمونه‌های آن کلاس توانایی فراخوانی آن تابع را دارد.

نکته: مانند __init__() پارامتر اول این متد ها self می‌باشد

برای نمونه در مثال کارمندان شرکت میتوانیم متد های مختلفی برای کارمندان بنویسیم
برای مثال متد افزایش حقوق کارمند، اگر حقوق وی کمتر از حد خاصی بود

یا متدی برای چاپ کردن یک سری اطلاعات مختصر و مفید از آن کارمند

اینچنین متد هایی که صرفا روی یک نمونه کار انجام میدهند متد های نمونه (instance methods) نامیده می‌شوند

در ادامه متد افزایش حقوق و چاپ اطلاعات را برسی میکنیم

برای افزایش مقدار حقوق نمونه مورد نظر نیاز به دسترسی به آن نمونه و اطلاعات داخل آن خواهیم داشت

پس در هنگام تعریف تابعِ افزایش حقوق، اجازه این دسترسی را به تابع با دادن پارامتر self تضمین می‌کنیم

با این کار تابع به نمونه ای که توسط آن فراخوانی شده و همچنین داده های خود آن نمونه دسترسی خواهد داشت
شاید گفتن این جمله به انگلیسی درک بهتری از چرایی استفاده از کلمه self به شما دهد

By doing this, the function will have access to the instance that was called by it, as well as the data of that instance itself


```python
class Applicants:
    entrance_year = 1402
    name_of_organization = "NASA"

    def __init__(self, esm, hoghoogh, modat_khedmat):
        self.name = esm
        self.salary = int(hoghoogh)
        self.years = int(modat_khedmat)

    def PersonRaise(self):
        if self.salary <= 5000000:
            self.salary = 10000000
            print(f" {self.name}'s salary was under 5M now it's 10 M")
        else:
            print(f"{self.name}'s salary was not higher than 5M so it remains unchanged")
```

البته الزامی به استفاده از کلمه self وجود ندارد
و شما قادر به استفاده هر کلمه دیگری که با آن احساس راحتی می‌کنید هستید
ولی استفاده از کلمه self در بین تمام برنامه نویسان یک رسم است مانند یک قانون نا نوشته

چاپ اطلاعات:

فرض کنیم نمونه ای ساختیم و در میان برنامه آن نمونه را پرینت میکنیم

```
a = Applicants("ali",20000000,1379)

print(a)
```

در این صورت با چنین خروجی مواجه می‌شوید

```
<__main__.Applicants object at 0x0000022F7B9D2F10>
```

که بیانگر مکان ذخیره سازی آن نمونه در حافظه دستگاه شماست

اما چنین چیزی مطلوب کاربر نیست، وی نیاز دارد تا اطلاعات مختصر و مفیدی درباره آن نمونه در اختیار داشته باشد

در این حالت از تابع __str__() استفاده می‌کنیم

چاپ اطلاعات:

```
class Applicants:
    def __init__(self, esm, hoghoogh, modat_khedmat):
        self.name = esm
        self.salary = int(hoghoogh)
        self.years = int(modat_khedmat)

    def __str__(self):
        return(f" this is {self.name} and their salary is {self.salary}")
```

در این صورت در هنگام پرینت، نمونه مورد نظر با پیغام و اطلاعاتی که خود انتخاب می‌کنیم به کاربر نمایش داده می‌شود

در واقع در این حالت استفاده از تابع print همان حکم استفاده از متد __str__() را دارد و منظور از self همان متغیر a خواهد بود

```
a = Applicants("ali",20000000,1379)
print(a)
```

Output: → this is ali and their salary is 20000000

6.2.2 class methods

6.2.2 class methods

متد های کلاس نیز مانند متد های نمونه هستند اما تفاوت اصلی آنها سطح دسترسی آنهاست

در حالی که متد های نمونه صرفا توانایی کار با همان نمونه و داده هایش و داده هایی که در آن تابع تعریف شده را دارد و به داده های خارج از آن تابع دسترسی ندارد، متد های کلاس، در سطح کل کلاس به داده ها و متغیر ها دسترسی دارد.

برای ایجاد یک متد کلاس در ابتدا قبل از کلمه کلیدی def، از عبارت `@classmethod` استفاده میکنیم

سپس تابع را تعریف کرده و به جای `self` از `cls` استفاده میکنیم(اجباری بر استفاده از خود کلمه `cls` نیست)

مثال instance_class_methods.py

```
@classmethod
def SomeName(cls,par1, par2, ...):
    some commands
    some commands
    some commands
```

6.3 Inheritance

6.3 Inheritance

وراثت (inheritance) فرآیندی است که طی آن یک کلاس ویژگی ها و متد های کلاس دیگر را می گیرد

کلاس های تازه تشکیل شده را کلاس های فرزند (Child classes) و به کلاس هایی که کلاس های فرزند از آنها مشتق شده اند، کلاس های والد (Parent classes) می گویند.

در موضوع وراثت و در حالت عادی، کلاس فرزند، تمامی متد ها و ویژگی های کلاس والد، یا والدین خود را به ارث می‌برد و وراثت انتخابی ویژگی ها و متد ها ممکن نیست

برای ساخت کلاس فرزند کافیست در هنگام تعریف کلاس، نام کلاس یا کلاس های والد را داخل پرانتز روبروی نام کاس فرزند بنویسیم

```
class FirstParent:
    pass

class SecondParent:
    pass

class Child(FirstParent, SecondParent):
    pass
```

کلاس های فرزند می توانند ویژگی ها و متد های کلاس های والد را تغییر (override) یا همچنین گسترش (extend) دهند. به عبارت دیگر، کلاس‌های فرزند تمام ویژگی‌ها و متدهای والدین را به ارث می‌برند، اما می‌توانند ویژگی‌ها و متدهایی را نیز داشته باشند که مختص خودشان است.

برای تغییر (override) یک متد یا یک ویژگی در کلاس فرزند، کافیست به سادگی متد والد را بازنویسی کرده و متد جدید دلخواه را به جای آن برنویسیم

```
class First:
    def __init__(self,name):
        self.name = name

    def show(self):
        print('First')

class Second(First):
    def show(self):
        print("Second")
```

هر کلاس فرزند می‌تواند علاوه بر متد ها و ویژگی هایی که از والدین خود به ارث برده، ویژگی ها و متد های خودش را نیز داشته باشد (extend)

```
class First:
    def __init__(self,name):
        self.name = name

    def show(self):
        print('First')

class Second(First):
    def show(self):
        print("Second")

    def Some_more_methods(self):
        commands
```

هر کلاس فرزند می‌تواند علاوه بر متد ها و ویژگی هایی که از والدین خود به ارث برده، ویژگی ها و متد های خودش را نیز داشته باشد (extend)

```python
class First:
    def __init__(self,name):
        self.name = name

    def show(self):
        print('First')

class Second(First):
    def __init__(self,name,last_name,age):
        self.name = name
        self.last_name = last_name
        self.age = age

    def Show(self):
        print(f"{self.name} and {self.last_name} and {self.age}")
```