در لیست‌ها اگر لیست تو در تو داشتیم، آدرس refrence نگهداری می‌شود.

```
L1 = [1,2,3]
L2 = [12,15, L1]
Print (L2)
L1.append(4)
Print (L2)
```

Output:

```
[12,15,[1,2,3]]
[12,15,[1,2,3,4]]
```

در اینجا می بینیم که آدرس L_1 در خط دوم نگه داشته شده است و وقتی L_1 را تغییر می‌دهیم L_2 هم تغییر می‌کند چون لیست mutable است.

وقتی خروجی را می‌بینیم متوجه می‌شویم هر اتفاقی برای L_2 می افتد در L_3 هم همانطور است، ولی اگر نخواهیم اینطور باشد و تغییر در آنها بدهیم نیاز داریم که یک کپی اساسی‌تر بگیریم.

```
L3 = L2
L1.append(5)
Print(L2)
Print(L3)
L2.append(10)
Print(L2)
Print(id(L2))
Print(id(L3))
L1.append(6)
Print(L2)
```

Output:

```
[12,15,[1,2,3,4,5]]
[12,15,[1,2,3,4,5]]
 [12,15,[1,2,3,4,5],10]
67438905694……..    آدرس
32811100467……..    آدرس
 [12,15,[1,2,3,4,5,6],10]
```

در این مثال به آدرس‌های تودرتو اشاره شد که آدرس را نگه می‌دارد و اینجا shallocopy گویند که می تواند کلاس هم باشد . حال اگر بخواهیم هر اتفاقی در مثالL_1 افتاد در L_2 که شامل L_1 هست نیفتد باید deepcopy انجام بدهیم، باید copy را import کنیم و به صورت زیر:

```
import copy
L1 = [1,2,3]
L2 = [12,15, L1]
Print (L2)
L1.append(4)
Print (L2)
```

$L_3 = L_2$
L_1.append(5)
Print(L_2)
Print(L_3)
L_4 = list(L_2)
L_5 = copy.deepcopy
L_2.append(10)
Print(L_2)
Print(id(L_2))
Print(id(L_3))
L_1.append(6)
Print(L_2) output: [12,15,[1,2,3,4,5,6],10]
Print(L_5) output: [12,15,[1,2,3,4,5]]

وقتی deepcopy انجام می دهیم ۱۰ و ۶ اضافه نشد و تمام اعضا کپی به طور کامل کپی شده است.

در deepcopy از رفرنسش کپی می‌گیرد و خودش را ذخیره می‌کند نه آدرسش را. در import copy برای کپی معمولی از copy.copy استفاده می‌کند.

اگر نیاز داشتیم می‌توان متدش را پیدا کرد و برای کلاس ها نیز owerrite کرد،این از متدهایی است که برای object ها می توان استفاده کرد.

در یک کلاس سه مدل متد وجود دارد:

```
class MyClass:

    def method(self):
    print('instance method')

    @classmethod
    def   class_method(cls):
    print('class method called')

    @staticmethod
    def    static_method( ):
    print('static method called')
```

متد عادی که self به یک آبجکت از کلاس اشاره دارد و cls به کلاس برمی‌گردد. از staticmethod جاهایی استفاده کرده که به instance ربطی ندارد .

به طور پیش فرض متدهایی که داریم instance method هستند، هر کدام از instance ها آدرسی برای خودشان دارند و attribute های خودشان را دارند. کاری که متد معمولی می‌کند می‌تواند attribute آن آبجکت خاص را تغییر بدهد.

```
Class MyClass:
      def  _ _init_ _(self,name):
           self.name = name
            print (self.name)
```

```
    def instance_method(self):
        print('Instance method')
        self.name = self.name.lower

a = MyClass('class A')
b = MyClass('class B')
a.instance_method( )
b.instance_method( )
```

output:
Instance method
class a
Instance method
class b

instance method با self که دستش است (آدرس آن آبجکت است)، می‌تواند محتوای object را تغییر دهد، attribute آبجکت را تغییر بدهد ، اضافه کند و ... .

در مقابل کلاس classmethod را داریم که آدرس سلف را ندارد و به نمونه‌هایی که از روی کلاس ساختیم دسترسی ندارد و به جای آن به خود آن کلاس دسترسی دارد و چیزهای عمومی برای آن کلاس است.

attribute ها می تواند دو مدل باشند: یکی این هایی که با آنها کار می‌کنیم که با سلف مقداردهی می‌کنیم که این instance attribute است، یعنی برای هر instance object ی مقدار خاص خودش را دارد، وقتی ۱۰ نمونه از روی کلاس می‌سازیم، attribute هر کدام از آنها مقدار خاص خودش را می گیرد چون با سلف آمده. ولی می‌توان attribute ی تعریف کرد مثل count مساوی صفر گذاشت، این دیگر attribute آن آبجکت نیست، attribute کلاس است. اگر حتی نمونه‌ای از آن کلاس هم نساخته باشم، باز آن attribute را دارم، باز هم می‌توان با آن کار کرد.

```
Class MyClass:
    count = 0

a = MyClass('count')
print(a.count)              output:  0
print(MyClass.count)        output:  0
```

ولی آیا می‌توان برای name هم انجام داد؟

```
print(MyClass.name)         output:   type object 'MyClass' has no attribute 'name'.
```

پیام می‌دهد که چنین attribueی ندارم، چون name برای instance هاست، باید یک instance بسازم و یک اسم به آن بدهم، بعد می‌توان استفاده کرد ولی count چنین ویژگی‌ای نداشت چون attribute خود کلاس بوده نه برای هر instance.

MyClass قالب و الگو است نه instance.

مثلا شناسنامه کلاس است ولی اسم آدم ویژگی آن آبجکت، برای یک شناسنامه خالی نمی‌توان گفت اسمش چیست که هنوز پر نشده است اسمی ندارد ولی تعداد صفحه دارد، چون ویژگی خود شناسنامه است. صفحه ، class attribute است و اسم instance attribute شناسنامه است.

```
Class MyClass:
    count = 0

a = MyClass('count')
print(a.count)              output:  0
print(MyClass.count)        output:  0
a.count = 5
print(a.count)              output:  5
*print(MyClass.count)       output:  0
```

در اینجا * عوض نشده است، انگار به شناسنامه‌ی یک نفر ۴ صفحه چسبانده شده باشد، که در این صورت کل شناسنامه‌ها تعداد صفحاتشان عوض نمی‌شود آن‌ها ثابتند ولی اگر یک ورژن داده شود که از این به بعد شناسنامه‌ها مثلا ۲ صفحه داشته باشند، برای همه عوض می‌شود.

```
MyClass.count = 2
print(MyClass.count)        output:  2
```

اگر attribute کلاس را برای یکی از instance ها عوض کنم فقط برای همان عوض شده ولی اگر با کلاس تغییر بدهیم برای همه تغییر می‌کند.

@classmethod اصلاً دسترسی به سلف ندارد، نمی‌تواند سلف‌ها را عوض کند، یعنی وقتی آدرس نمونه ها را نمی بیند چطور می‌تواند تغییر دهد ولی در عوض آدرس کلاس را دارد پس می‌تواند روی کلاس عملیات انجام دهد می‌تواند مقدار کانت را عوض کند.

```
Class MyClass:
    count = 0

  @classmethod
  def  class_method(cls):
      print('class method called')
      cls.count +=1

  a = MyClass('class A')
  a.class_method( )
  print(a.count)
  b.class_mrthod( )
  print(b.count)

  output:
  class method called
  1
  class method called
  1
```

چون تغییر را روی کلاس اعمال کردیم به وسیله a برای b هم count = 1 شده است. اگر سه بار a.class-method() را بنویسیم، count برابر ۳ می‌شود و برای

```
  b.class_method( )
Print(b.count)          output: 3
```

مثل اینکه سایت سازمان ثبت احوال را هک کرده و کل صفحات برای همه شناسنامه ها را تغییر داده باشند.

```
b.class_method( )
print(MyClass.count)
```

```
output:
class method called
4
```

به ورودی نیاز ندارد و به کلاس دسترسی دارد.

```
MyClass.Class_method( )
print(MyClass.count)
```

```
output:
class method called
5
```

مدل سوم از متدها staticmethod ها هستند که نه آدرس کلاس را دارد، نه attribute های مربوط به آن instance را می‌تواند تغییر دهد. برای زمانی کاربرد دارد که عملیاتی را می‌خواهیم انجام دهیم و می‌خواهیم در حوزه‌ی همین کلاس باشد ولی نمی خواهیم سلف کلاس را عوض کند.

این روش نام گذاری های متدها و کارکردشان برای منظم تر شدن و بهینه شدن و باگ کمتر خوردن است که کار را راحت می‌کند. constructor برای هر کلاسی که با _ _init_ _ _تعریف می‌شود.

یکی از کاربردهای کلاس متدها این است که می‌توان factory method با آن بسازیم یعنی برای شرایط متفاوت که قرار است با آن کلاس تعامل کنیم و از آن کلاس instance بسازیم، متد تعریف می‌کنیم ولی مطمئن هستیم که این با سبک کلاس کار می کند.

```
@classmethod
def get_data(cls):
   name= input('enter name')
   return cls(name)

new project = MyClass.get_data( )
```

مثال دیگر فرض کنید کلاس ما تولید یک سری ماشین باشد، یک سری ویژگی می‌گیرد مثلا نوع سوخت، رنگ و غیره ...، یک سری مدل هست که به طور پیش فرض می توان از آن ها ساخت به جای اینکه ما مدل به آن داد، حالا با کلاس متد می توان کانستراکتور آن را ساخت و بعد یک instance از آن ساخت:

```
@classmethod
def    bmw(cls):
      return cls('bmw')
```

اینجا مستقیم الگو ساخته شده است.

```
factory1 = MyClass.bmw( )
```

مثلا در تمرین می‌توانستیم کتاب ها را به عنوان پیش فرض در اینجا تعریف کنیم که به این صورت class method به آدرس‌ها دسترسی داشت.

حالا استاتیک متد مثلاً قرار است یک اسمی را دو بار چاپ کند. در اینجا نیازی نیست که به آبجکت ها دسترسی داشته و بعد در جای دیگر از آن استفاده کرد.

```
@staticmethod
def   double_name(name):
  print(name*2)

def   printing(self):
  self.double_name(self.name )
```

Property:

```
class Book:
  def _ _init_ _(self,name,pages):
        self.name = name
        self.pages  = pages
        self._ _read=0
 #setter method
   def set_read(self,read_page):
    if read_page <= self.pages:
       self.__read = read_page
    else:
       raise ValueError('No more than book pages')
# getter method
  def get_read(self):
    return self.__read

 shallows = Book('shallows',350)
 shallows.set_read(300)
 shallows.set_read(500)
 print(shallows.get_read())
```

output:

```
300
error
```

در read_ _ متغیر را private کرده که از بیرون نتوان تغییر داد(تعداد صفحات خوانده شده).

اگر خواستم مقدار دهم set کرده و برای مقدار گرفتن get کرد در اینجا ما یک attribute داریم که با استفاده از دو متد، کنترل روی read_ _ داریم، این خیلی راحت نیست و برای حل مشکل می توان از property استفاده کرد که راحت تر با read_ _ کار کرد:

```
class Book:
   def __init__(self,name,pages):
      self.name = name
      self.pages = pages
      self.__read =0
```

```
# getter
   @property
1    def read*(self):
2      return   self.__read

  # setter
   @read*.setter
3   def read*(self,read_pages):
4      if read_pages <=self.pages:
5         self.__read = read_pages

  shallows = Book('shallows',350)
6 print(shallows.read)              output: 0
  print(shallows.name)             output: shallows
  shallows.read = 450
  print(shallows.read)             output: 0
  shallows.read = 248
  print(shallows.read)             output: 248
```

به این صورت که با خط ۱و ۲ get کرده و با خط ۳و ۴ ، set شده است و با خط ۶ خروجی گرفته و می‌توان در یک متغیر دیگر ریخته، با قسمت setter فراخوانی می‌شود.در خط ۶ read پرانتز نمی‌خواهد. در اینجا اگر مقداری که دادیم معتبر باشد assigne می‌کند ولی اگر از ۳۵۰ بزرگتر باشد update نمیکند و صفر بر می‌گرداند. دقت شود که در خط ۱ و ۳ و قبل از ۳ که * گذاشته شده است باید همانند یکدیگر نوشته شده شود. پس property مقادیر را چک می‌کند و فقط مقادیر درست را assigne می‌کند و اگر بخواهیم بر می‌گرداند. property را طور دیگر هم می‌توان تعریف کرد:

```
   @property
   def progress(self):
      return f'{self.__read} out of {self.pages}'

shallows.read = 248
print(shallows.progress)        output: 248 out of 350
shallows.read=345
print(shallows.progress)        output: 345 out of 350
```

مثال دیگر از property:

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

   @property
   def name(self):
```

```
        print("Getter for the name")
        return f"{self.first_name} {self.last_name}"

    @name.setter
    def name(self, name):
        print("Setter for the name")
        self.first_name, self.last_name = name.split()

student = Student("John", "Smith")
print(student.first_name)
print(student.last_name)
print("Student Name:", student.name)
student.name = "Johnny Smith"
print("After setting:", student.name)
```

output:

```
John
Smith
Getter for the name
Student Name: John Smith
Setter for the name
Getter for the name
After setting: Johnny Smith
```

همه چیز در پایتون آبجکت هستند حتی func هم آبجکت هستند، یعنی خیلی از عملیات‌ها را که روی متغیرها می توانم انجام می دهند می توان روی توابع هم انجام داد. چه عملیاتی؟ مثلاً assignment، یه چیزی را تعریف کنیم ، داخلش بگذاریم و بخواهیم برگردانیم. هر کدام از اینها که ظاهرش کاری است که برایvariableها انجام می‌دهیم در صورتی که برای توابع هم می‌توان انجام داد . توابع را هم به آرگومان می توان به هم پاس داد.

```
def   say_hello(name):
    return f"Hello {name}"

def   be_awesome(name):
    return f"Yo {name}, together we are the awesomest!"


def   greet_bob(greeter_func):
1     return greeter_func("Bob")


print(greet_bob(say_hello))
```

output:

```
Hello Bob
```

توابع say_hello و be_awesome هر کدام یک رشته بر می‌گردانند. say_hello یک تابع بوده که به عنوان ورودی به greet_bob داده و در خط 1 call کرده و return می کند، با مقدار Bob برمی‌گرداند.

```
print(greet_bob(be_awesome*))
```

output: Yo Bob, together we are the awesomest!

اگر پرانتز تابع در * بگذاریم، call می‌کنیم ولی اگر پرانتز تابع را نگذاریم فقط رفرنس آن تابع را می‌دهد. مثال:

```
def test( ):
    Return True
print(test( ))          output: True
print(test)             output: <function test at ox098786(address)>
```

دراینجا تابع را صدا می‌کنیم.

در اینجا چون پرانتز را نگذاشتیم ، رفرنس را برمی‌گرداند.

می‌توان توابع را تو در تو تعریف کرد، نه تنها یکی که دو تا هم می توان تعریف کرد و همان جا صدا کرد.

```
def parent():
1 print("Printing from the parent() function")

  def first_child():
    print("Printing from the first_child() function")

  def second_child():
    print("Printing from the second_child() function")

2 second_child()
3 first_child()

parent()
```

output:

```
Printing from the parent() function
Printing from the second_child() function
Printing from the first_child() function
```

اگر تنها بنویسیم second_child()، نمی‌شناسد و فقط داخل parent می‌شناسد. دیده می‌شود که به ترتیب فراخوانی‌اش چاپ می‌کند نه به ترتیب نوشتن.

در حالت سوم،

```
def parent(num):
  def first_child():
    return "Hi, I am Emma"

  def second_child():
    return "Call me Liam"

  if num == 1:
```

```
        return first_child
    else:
        return second_child

 first = parent(1)
1 print(first)
 sec = parent(5)
2 print(sec)
3 print(first())
4 print(sec())
```

output:
```
1<function parent.<locals>.first_child at 0x0000000001E7B828>
2<function parent.<locals>.second_child at 0x0000000001E7B9D8>
3Hi, I am Emma
4Call me Liam
```

در return آدرس تابع را برمی‌گرداند مثلا آدرس first_child را بر می‌گرداند (بدون پرانتز).
در `parent` بعد از تعریف دوتا func شرط کرده که اگر `num=1`، آدرس first را برگردان و در غیر این صورت second را برگرداند که اجرا نمی‌کند، رفرنسش را برمی‌گرداند(1,2)، ولی با پرانتز آن طور که میخواهیم اجرا می‌شود(3,4).

Decorator:

دکوراتور فانکشنی است که یک تابع به عنوان وروری می گیرد و داخلش یک تابع wrapper دارد که wrapper روی تابع ورودی کاری که دادیم انجام می‌دهد و return می کند.

```
def my_decorator(func):
  *  def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
     return wrapper


 @my_decorator
 def say_whee():
    print("Whee!")

 say_whee()
```

output:
```
Something is happening before the function is called.
Whee!
Something is happening after the function is called.
```

در اینجا say_whee( ) که در زیر @my_decorator قرار دارد داخل دکوریتوری که تعریف کردیم ❊ برده شده و کارهای داخل wrapper انجام شده که در اینجا چاپ یک رشته قبل و بعد از func است، می‌باشد. func( ) در اینجا say_whee است و رشته‌ی !whee را پرینت می‌کند.

@ ها decorator هستند.Property برای کلاس معنی می‌دهد ولی decorator را جاهای دیگر هم می توان استفاده کرد.

کلاس استاد جلیلیان