

ما می‌توانیم و از حلقه ها برای پر کردن لیست ها و انجام عملیات روی اونها استفاده می‌کردیم برای مثال:

```
list_comperhension.py > ...
1  fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
2  newlist = []
3
4  for x in fruits:
5      if "a" in x:
6          newlist.append(x)
7
8  print(newlist)
9
```

```
cat /dev/null < /dev/null > /dev/null
['apple', 'banana', 'mango']
```

طبق معمول اینجا اومدیم این مار رو کردیم ولی حال همین کد رو می‌خواهیم معادل لیست کامپرشن شو بنویسیم:

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = [x for x in fruits if "a" in x]
print(newlist)
```

```
cat /dev/null < /dev/null > /dev/null
['apple', 'banana', 'mango']
```

حالا این کد اصلن چه معنی میده اصن؟ به سینتکس اون توجه کنید:

## The Syntax

```
newlist = [expression for item in iterable if condition == True]
```

اون ایکس که اولش گذاشتیم یعنی اون چیز رو **append** کن به لیست مون . بعد حلقه مونو زدیم و گفتیم به اعضای اون ایکس هایی که توی لیست میوه ها هست . و بعدش هم یک شرط گذاشتیم که اگر اون شرط روی ایکس برقرار بود میاد اون رو **append** میکنه به لیست مون

کد بالایی و پایینی یک کاری را انجام میدهند ولی با کد نویسی کمتر

مثال بعدی رو ببینید تا متوجه بشید بیشتر:

```
test2.py > ...
1  fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
2
3  newlist = [x for x in fruits if x != "apple"]
4
```

توی این کد گفتم که یه ایکس داریم که به اعضای اون در میوه ها است ولی فقط به شرطی اضافه شون کن که **apple** نباشن اگر پرینت کنیم لیست رو میبینیم که تمام میوه ها بجز سیب در لیست جدید وجود دارد.

مثال دیگه:

```
test.py > ...
1  fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
2
3
4
5  newlist = [x.capitalize() for x in fruits if x.islower()]
6  |
7
8  print(newlist)
9
```

اومدم گفتم که اون ایکس هارو بزرگ شده شو میخام . به اعضای اون ایکسایی که توی میوه ها هست اگر تابع **islower()** که روش اعمال میشه **True** برگردونه . بخش شرطی اگر درست باشه اون ایکس رو قبول میکنه و میاد اولش که نوشتم **x.capitalize()** شده اش رو میریزه توی لیست.

```
test.py > ...
1  fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
2
3  newlist = [x if x != "banana" else "orange" for x in fruits]
4
5  print(newlist)
6
```

توی این یکی مثال اومدم گفتم که : اون ایکس هایی رو برداری بیار که اول اینکه **bannana** نباشه ولی اگر بود بجاش **orange** بزار . اون هایی رو که در میوه ها هستند . میتونیم شرط دیگه ای رو هم بنزیم مثلاً بالا بگیریم اونایی که **islower()** اشون درسته و یا حتی اون اول ایکس رو روش بلایی بیاریم مثلاً بنزیم **x.upper()** تا بیاد حروف بزرگ شده شو اضافه کنه

انواع متود ها : استاتیک . کلاس و اینستنس متود ها

---

## اینستنس متود : instance method

تمام متود هایی که در کلاس ها مینوشتیم و به آن **self** رو پاس میدادیم به آن اینستنس متود ( متود هایی که روی اینستنس ها ( آبجکت ها ) اجرا میشوند ) میگویند :

```
test.py > ...
1 class Person:
2     def __init__(self , name , age):
3         self.name = name
4         self.age = age
5
6     #instance method -> will execute the self
7     def show(self):
8         print(f'{self.name} is {self.age} years old ')
9
10
11
12 p1 = Person('erfan' , 20)
13 p2 = Person('mamad' , 84)
14
15 p1.show()
16 p2.show()
```

```
erfan is 20 years old
mamad is 84 years old
```

اینجا تابع **show** یک اینستنس متود هست چون داره با سلف کار میکنه و روی آبجکت هامون اجرا میشه **p1.show()** **p2.show()** اینجا ابجکت هامونو صدا زدیم

یا مثلا حتی داندرا ها مثل داندرا اس تی ار . چون وقتی از ابجکت رشته میگیریم فعال میشه بهش میگی اینستنس متد

---

موقعی هست که ما میخاییم روی ورودی های داندن اینیت تغییر ایجاد کنیم و اون تغییر پیدا کرده رو ذخیره منیم مثلا ما از طرف سال تولد میگیریم ولی میخاییم سن رو ذخیره کنیم پس کسی که پایتون سرش نشه میاد و این کار رو میکنه:

```
test.py > Person > __init__
1 class Person:
2     def __init__(self , name , age):
3         self.name = name
4         self.age = 1403 - age
5
6     #instance method -> will execute the self
7     def show(self):
8         print(f'{self.name} is {self.age} years old ')
9
10
11
12 p1 = Person('erfan' , 1383)
13 p2 = Person('mamad' , 1350)
14
15 p1.show()
16 p2.show()
```

این کد به درستی کار میکنه ولی مانمیتونیم که هر سال جدید بیاییم برای کارفرما برنامه شو اپدیت کنیم ! پس باید از مازول تاریخ استفاده کنیم ولی وقتی این کارو میکنیم به یه مشکلی میخوریم:

```
test.py > Person > __init__
1 from datetime import datetime
2
3 class Person:
4     def __init__(self , name , age):
5         self.name = name
6         self.age = datetime.year[] - age
7
8     #instance method -> will execute the self
9     def show(self):
10         print(f'{self.name} is {self.age} years old ')
11
12
13
14 p1 = Person('erfan' , 1383)
15 p2 = Person('mamad' , 1350)
16
17 p1.show()
18 p2.show()
```

```
Traceback (most recent call last):
  File "/home/cuthbert/Desktop/New folder (2)/test/test.py", line 14, in <module>
    p1 = Person('erfan' , 1383)
  File "/home/cuthbert/Desktop/New folder (2)/test/test.py", line 6, in __init__
    self.age = datetime.year[] - age
TypeError: function missing required argument 'year' (pos 1)
```

حالا این اروری که میگیریم ربطی به این نداره که این کار رو نمیشه کرد این ارور از مازول کتابخانه تاریخ هست ولی در کل این رو میخاییم بگم که:

\*\*\*\* زمانی که میخاییم در ویژگی های ورودی ابجکت تغییری ایجاد کنیم و اون رو ذخیره کنیم از کلاس متود ها استفاده میکنیم \*\*\*\*

حالا چطور ؟ بیایید ص بعد....

این تعریف درست تر هست که بگم که وقتی از این استفاده میکنیم که نیاز داشته باشیم به خود کلاس دسترسی داشته باشیم مثلاً اینجا ما به خود کلاس دسترسی پیدا میکنیم تا بیاییم روی مقادیر ورودی تغییری ایجاد کنیم.

مراحل ساخت یک کلاس متود:

اول: میاییم و بالای تابع مورد نظر یک ادساین میزاریم و مینویسیم **classmethod** تا این تابع رو به عنوان کلاس متود در نظر بگیره

```
@classmethod
def from_birth
```

دوم: دیگه چون اینستنس متود نیست و ما با ابجکت ها کار نداریم نماییم **self** بدیم بهش . چون میخاییم روی خود کلاس تغییر ایجاد کنیم مینویسیم **cls** که این یک مقدار قرار دادی مثل سلف هست ولی بجای اینکه ابجکت رو بگیره مثل سلف میاد خود کلاس رو میگیره

```
@classmethod
def from_birth(cls ,
```

سوم: میاییم ارگومان هایی که قراره بگیریم رو مینویسم کنار اون سی ال اس . نکته: در آخر موقع بازگردانی این تابع باید مقادیری که در داندر اینیت داده ایم برگردانده شود تا به داندر اینیت برسد ولی حالا میتونید همینجا همه رو بگیرید و یا خودتون بسازید منظورم چیه:

```
p1 = Person.from_birth('erfan' , 2004)
```

موقع ابجکت سازی دیگه خود کلاس رو خالی صدا نمیزنیم تا داندر اینیت اش اجرا بشه اول میاییم اون کلاس متود رو صدا میزنیم تا اول اون اجرا بشه بعد بره سراغ داندر اینیت تا اول روی مقادیر ورودی تغییر ایجاد کرده باشیم بعد بره بشینه توی داندر اینیت

اینم که میگم یا خودتون بسازیدش مورد آخره بعد از این بعدیه

چهارم : باید اون تابع ریترن بکنه **cls** رو یعنی میاد خود اون کلاس رو برمیگردونه داخل اون متغیری که ساختیم یعنی **p1** و جلوی پرانتزش انتظار میره که مقادیر داندر اینیت رو بهش پاس بدیم چون داره خودشو برمیگردونه پس کلاس رو اجرا میکنه:

test.py > Person > from\_birth

```
1 import datetime
2
3 class Person:
4     def __init__(self, name, age, email):
5         self.name = name
6         self.age = age
7         self.email = email
8
9     def show(self):
10        print(f'{self.name} is {self.age} years old with {self.email} address')
11
12    @classmethod
13    def from_birth(cls, name, date):
14        email = f'{name}-{[date]}@gmail.com'
15        return cls(name, datetime.datetime.now().year - date, email)
16        #      ^name           ^age           ^email -> __init__() properties
17
18 p1 = Person.from_birth('erfan', 2004)
19 p2 = Person('mamad', 25, 'mamad@gmail.com')
20
21 p1.show()
22 p2.show()
```

```
erfan is 20 years old with erfan-2004@gmail.com address
mamad is 25 years old with mamad@gmail.com address
```

اون قضیه متغیر هارو بگم : ببینید توی تابع **from\_birth** که من اومدم کلاس متودش کردم اومدم ۲ تا مقدار فقط گرفتم ولی در صورتی که داندر اینیت مقدار ایمیل رو هم میخاد . خب اینجا مشکلی نیست چون من توی **p1** مستقیم کلاس رو صدا نزدم که اومدم کلاس متد شو صدا زدم پس فقط ازم نام و تاریخ رو میخاد اومدم ایمیل رو خودم درست کردم با تلفیق اسم و تاریخ تولد و بعد موقع **return** کردن که گفتم باید اون س ال اس ریترن بشه بهش مقادیر داندر اینیت رو پاس دادم زیرشم که هر کدوم به کدوم از ویژگی های داندر اینیت میره رو هم نوشتم به صورت کامنت ولی مثلا الان توی **p2** اینکارو نکردم و مستقیم ابجکت ساختم و به داندر اینیت وصل شدم و ایمیل رو پاس دادم بهش و در خروجی هم مشکلی نمیبینم.

دوباره ص بعد از اول میگم که چه اتفاقی افتاد

پس شد زمانی از کلاس متد استفاده میکنیم که بخاییم در رفتار کلاس تغییری ایجاد کنیم



```

test.py > Person > from_birth
1 import datetime
2
3 class Person:
4     def __init__(self, name, age, email):
5         self.name = name
6         self.age = age
7         self.email = email
8
9     def show(self):
10        print(f'{self.name} is {self.age} years old with {self.email} address')
11
12    @classmethod
13    def from_birth(cls, name, date):
14        email = f'{name}-{date}@gmail.com'
15        return cls(name, datetime.datetime.now().year - date, email)
16        # ^name          ^age          ^email -> __init__() properties
17
18 p1 = Person.from_birth('erfan', 2004)
19 p2 = Person('mamad', 25, 'mamad@gmail.com')
20
21 p1.show()
22 p2.show()

```

```

erfan is 20 years old with erfan-2004@gmail.com address
mamad is 25 years old with mamad@gmail.com address

```

توضیح دوباره عکس:

اول اومدیم با نوشتن **@classmethod** بالای متود **from\_birth** اون رو تبدیل به یک کلاس متد کردیم تا رفتار کلاس رو عوض کنیم. توی ارگومان های دیگه **self** ندادیم چون با ابجکت ها کاری نداریم. ما فقط موقع ابجکت سازی میخاییم بجای صدا زدن مسقتیم خود کلاس ایم متد رو صدا بزنینم تا ویژگی شو عوض کنیم. توی ارگومان ها دوست دارید همه ی ارگومان های داندنر اینیت رو بگیرید یا دوست ندارید و میخاید بعضی هاشون رو از کاربر نگیرید و خودتون درست کنید مشکلی نداره. اینجا **name** و **date** رو میگیرم و بعد یک متغیر میسازم به اسم **email** و یک ایمیل با ترکیبی از اسم و تاریخ تولدش میسازم. بعد باید **cls** رو ریترن کنم ( تا کلاس صدا زده بشه و چون داریم ریترن میکنیم ریخته بشه توی **p1** تا ابجکت ساخته بشه ) بعد موقع ساختن ابجکت همینجوری نمایم کلاس رو صدا بزنام میایم بجای صدای مستقیم کلاس ( که موجب میشه داندنر اینیتش صدا زده بشه ) اون کلاس متد رو صدا میزنم تا ویژگی های دستی رو کلاس اضافه کنم.

اون تیکش که باید **cls** ریترن بشه خیلی مهمه دیاگرامش اینشکلیه:

```

test.py
1  import datetime
2
3  class Person:
4      def __init__(self , name , age , email):
5          self.name = name
6          self.age = age
7          self.email = email
8
9      def show(self):
10         print(f'{self.name} is {self.age} years old with {self.email} address')
11
12     @classmethod
13     def from_birth(cls , name , date):
14         email = f'{name}-{date}@gmail.com'
15         return cls(name , datetime.datetime.now().year- date , email)
16         # ^name          ^age          ^email -> __init__() properties
17
18 p1 = Person.from_birth('erfan' , 2004)
19 p2 = Person('mamad' , 25 , 'mamad@gmail.com')
20
21 p1.show()
22 p2.show()

```

موقع ریترن شدن کلاس صدا زده میشه و داندر اینیتش صدا زده میشه و متغیر هایی که توی پرانتز دادم (خط پونزده) ارسال میشن به داندر اینیت و بعد از تشکیل ابجکت ریترن داده میشه به داخل p1 بعد حالا هرکاری میخایی بکن با اون ابجکت

---



استاتیک متود ها:

این چیز خواصی نداره فقط باید بدونید که:

این کد رو ببینید:

```
test.py > ...
1  class A :
2      age = 13
3
4
5  Codiumate: Options | Test this
6  def show():
7      if A.age >12:
8          print('ok')
9      else :
10         print("no")
11
12  show()
```

PROBLEMS OUTPUT DEBUG CONSOLE

cuthbert@cuthbert-pc:~/Des  
ok

این تابع خارج از کلاس هست و داره به خوبی کار میکنه ولی در برنامه نویسی پیشرفته وقتی یک تابع به مقادیری از کلاس در ارتباطه به صورت غیر مستقیم ما میاییم اون تابع رو به صورت استاتیک توی خود کلاس مینویسیم تا کد مون قشنگ تر بشه یعنی چی ؟

```

test.py > ...
Codiumate: Options | Test this class
1  class A :
2      age = 13
3
4      Codiumate: Options | Test this m
5      @staticmethod
6      def show():
7          if A.age >12:
8              print('ok')
9          else :
10             print("no")
11
12  A.show()

```

ببینید فقط کافی‌ه بالاش **@staticmethod** رو بزنی و بعد باهاش کار کنید

و برای صدا زدنش هم از داخل کلاس صداش می‌زنیم . وقتی توابع یه جووری نخشون وصله به کلاس بهتره که داخل کلاس نوشته بشن . این کد رو اینجوری هم میشه نوشت:

```

test.py > A > show > age
Codiumate: Options | Test this class
1  class A :
2      age = 13
3
4      Codiumate: Options | Test this meth
5      @staticmethod
6      def show(age):
7          if age >12:
8              print('ok')
9          else :
10             print("no")
11
12  A.show()


```

چون دیگه داخل کلاس هست نمی‌خاد بنویسیم **A.age** همون **age** رو پاس بدیم به عنوان آرگومان براش کافی‌ه.

مفهوم پلیمورفیسم polymorphism چیست.

چیز خاصی نیست از قبل هم میدونستید فقط نه به این اسم :

```
test.py > Boat > move
Codiumate: Options | Test this class
1 class Vehicle:
  Codiumate: Options | Test this method
2   def __init__(self, brand, model):
3       self.brand = brand
4       self.model = model
5
6   def move(self):
7       print("Move!")
8
9 class Car(Vehicle):
10    pass
11
12 Codiumate: Options | Test this class
13 class Boat(Vehicle):
14     def move(self):
15         print("Sail!")
16
17 Codiumate: Options | Test this class
18 class Plane(Vehicle):
19     def move(self):
20         print("Fly!")
21
22 car1 = Car("Ford", "Mustang")
23 boat1 = Boat("Ibiza", "Touring 20")
24 plane1 = Plane("Boeing", "747")
25
26 car1.move()
27 boat1.move()
28 plane1.move()
```



یه کلاس **vehicle** داریم که ۳ تا کلاس دیگه ازش ارث بری میکنن . در کلاس قایق و هواپیما تابع **move** به قول معروفی **overwrite** شده و در خروجی رفتاری غیر از رفتار والد خود رو نشون میده . ولی کلاس ماشین توش تابع **move** اوررایت نشده و داره رفتار والد رو نشون میده . به این میگن رفتار پولیمورسیم . میتونیم حتی کاری کنیم که علاوه بر رفتار خودش رفتار والدش رو هم نشون بده . چجور ؟ شما بگین

بیا ص بعد ببینم درست حدث زدی یا نه

```
test.py > ...
Codiumate: Options | Test this class
1 class Vehicle:
  Codiumate: Options | Test this method
2   def __init__(self, brand, model):
3       self.brand = brand
4       self.model = model
5
6   def move(self):
7       print("Move!")
8
9 class Car(Vehicle):
10    pass
11
12 Codiumate: Options | Test this class
13 class Boat(Vehicle):
14     Codiumate: Options | Test this method
15     def move(self):
16         print("Sail!")
17         super().move()
18
19 Codiumate: Options | Test this class
20 class Plane(Vehicle):
21     Codiumate: Options | Test this method
22     def move(self):
23         print("Fly!")
24         super().move()
25
26 car1 = Car("Ford", "Mustang")
27 boat1 = Boat("Ibiza", "Touring 20")
28 plane1 = Plane("Boeing", "747")
29
30 car1.move()
31 boat1.move()
32 plane1.move()
```



Move!  
Sail!  
Move!  
Fly!  
Move!

اینجا باز اوررایت کردیم و داخل اوررایت شدع گفتیم **super().move()** یعنی برو از توی بابات تابع **move** شو بیار و هم مال خودشو اجرا میکنه هم مال باباشو. البته فقط مال هواپیما و قایق همچین کاری کردم.