python™

فهرست

فصل اول: نصب و پیاده سازی ۳
فصل دوم: مبانی و دستور نحو ۷
فصل سوم: انواع داده (سطح یک) ۱۵
فصل چهارم: دستورات کنترلی (تصمیم) ۳٤
فصل پنجم: دستورات کنترلی (تکرار) ۳٦
فصل ششم: تابع ٤۱
فصل هفتم: انواع داده (سطح دو) ٦٥
فصل هشتم: ماژول ها و بسته ها ۷۸
فصل نهم: فایل ها، ورودی و خروجی ۸٤
فصل دهم: کلاس‌ها ۹٦
فصل یازدهم: مدیریت خطا ۱٤۱
فصل دوازدهم: همزمانی ۱٤٥
فصل سیزدهم: مباحث تکمیلی ۱٥۰
منبع ۱٥۳

فصل اول: نصب و پیاده سازی

برای نوشتن کد محیط های مختلفی داریم

۱- pycharm: IDE
۲- IDLE
۳- vs code:Code Editor
۴- Jupiter :Notebook
۵- Web
۶- ...

ابتدا از سایت python.org اخرین نسخه پایتون را دانلود میکنیم و دقت کنیم که هنگام نصب تیک add را بزنیم :

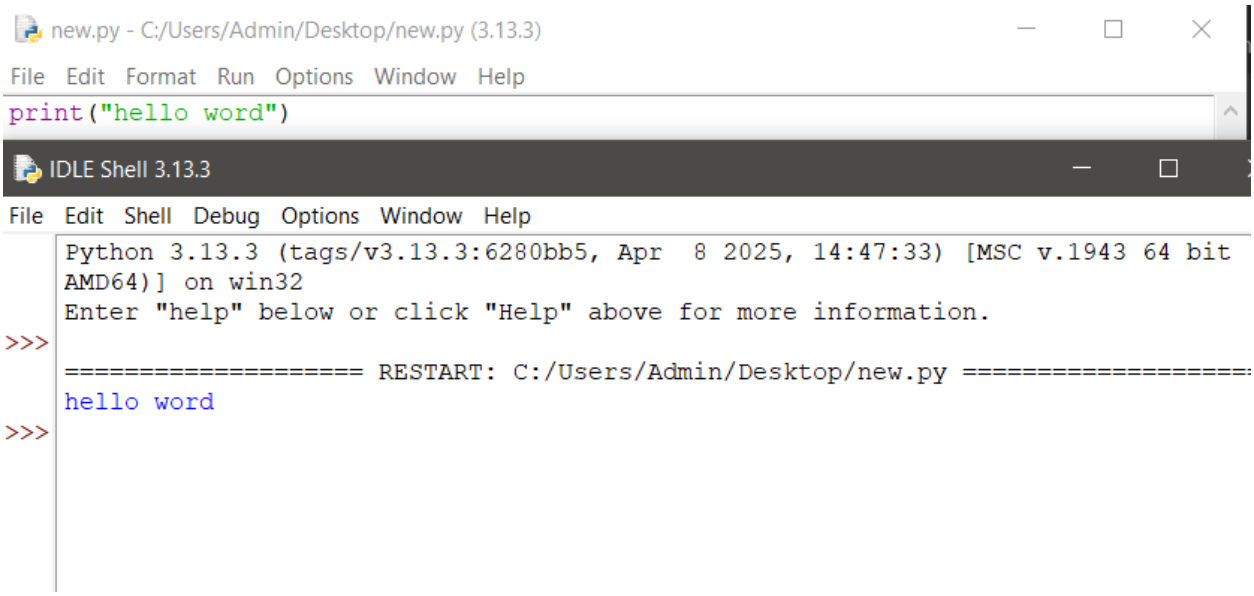
- اولین محیطی که میتوان با آن کد زنی کرد IDLE است که محیط نسبتا ساده است و برای کد زنی در آن ابتدا

طبق نسخه نصب شده پایتون به فولدر آن رفته و IDLE را باز میکنیم و روی new زده و یک فایل پایتونی برای ما ساخته میشود و با F5 میتوانیم آن را اجرا کنیم

همچنین یک محیط تعاملی هم دارد :

کد زنی در پایتون به دو صورت خط به خط اجرا شدن و به صورت تعاملی اجرا کردن میتوانیم تقسیم بندی کنیم

تعاملی یعنی هر خط کدی زدیم در همان لحظه جواب آن برگردانده شود



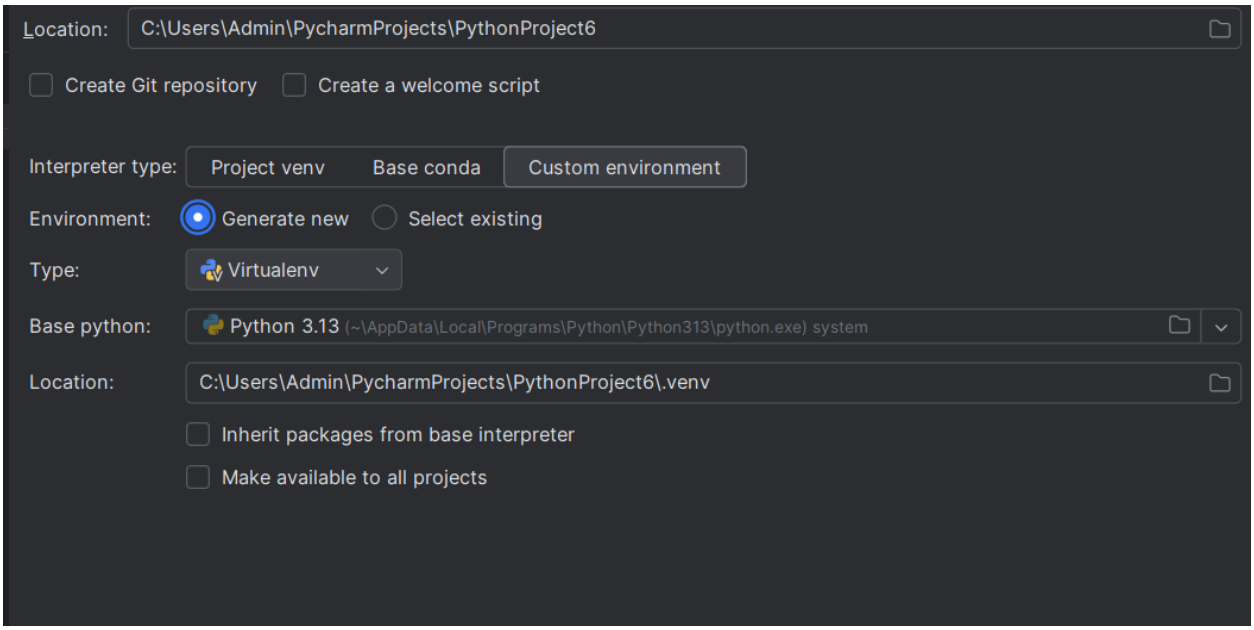
- برای اجرای کد در محیط cmd یا powerShell باید به صورت زیر کد را بزنیم

>>> python [path]

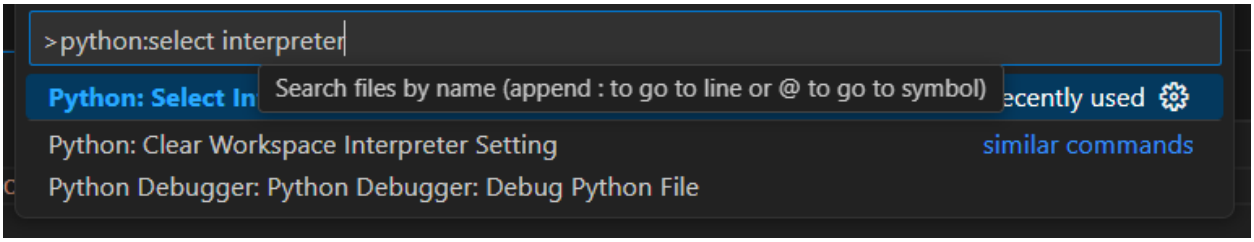
اگر python خالی را بزنیم یک محیط تعاملی ایجاد میشود

برای خروج از این حالت باید دستور exit() را بزنیم

- در Jupiter زمانی که پروژه ایجاد میشود میتوانیم virtualenv ایجاد کنیم که یک محیط ایزوله برای ایجاد فایل و کتابخانه ها و نسخه interpreter که بهتر است یک پوشه env درون فایل پروژه py. ایجاد کنیم و مسیر را در آنجا تنظیم کنیم . این به ما کمک میکند که اگر به سیستم دیگر رفتیم به دلیل نبودن پکیج ها و تفاوت در مفسرها مشکلی پیش نیاید



- برای انتخاب interpreter در vsCode باید در قسمت مشخص شده python:selesct interpreter را وارد کنیم



- محیط بعدی که با آن آشنا خواهیم شد Jupiter است

برای استفاده از آن ابتدا pip install ipython را میزنیم

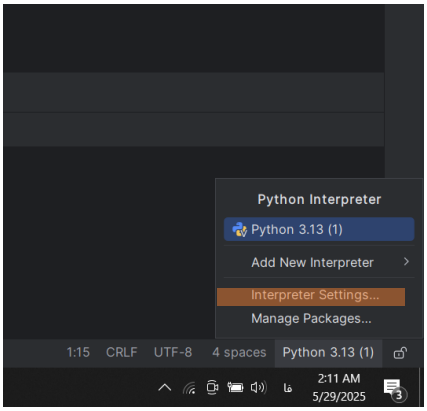
سپس anaconda را نصب میکنیم و به راحتی میتوانیم از آن استفاده کنیم

- برای استفاده python در اندروید میتوانیم از pydroid 3 استفاده کنیم که باید به همراه آن دو نرافزار دیگرهم

نصب شود : pydroid repository plugin و pydroid permission plugin

- حال میخواهیم با محیط pycharm بیشتر آشنا شویم

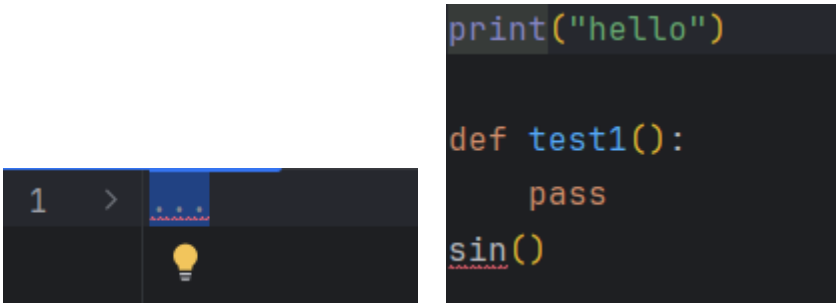
برای اضافه کردن پکیج ها میتوانیم از بخش مربوط به interpreter آن ها را نصب کنیم که نحوه دسترسی به آن در زیر آورده شده است :



- Ctrl + F: برای جست وجو
- Ctrl + R: برای جست و جو و جایگزین کردن متن
- Ctrl + Shift + arrow keys: برای انتخاب کردن قسمتی از متن یا جابه‌جایی آن به بالا و پایین
- Alt + Click mouse : هرکجا بزنیم یک کرسر اضافه میکند و میتوانیم همزمان تایپ کنیم
- Ctrl + Alt + L: برای مرتب سازی کدها
- Ctrl + W : برای انتخاب به صورت سلسله مراتبی
- Ctrl + D: برای کپی کردن یک متن در زیر خودش دقیقا
- Ctrl + /: برای کامنت کردن چندین خط
- Ctrl + Space : برای دیدن پیش بینی ها

- اگر بخواهیم یک متن را جمع یا بسته کنیم از آن استفاده میکنیم: . + Ctrl

این عمل folding نام دارد



برای دسترسی به آن باید به بخش code,folding میرویم

- رفتن به بخش اکشن ها و در آنجا هر بخشی را بخواهیم میتوانیم سرچ بزنیم : Ctrl + Shift + a
- رفتن به درون تابع: Ctrl + B
- رفتن به خط بعدی از هر کجایی که هستیم: Shift + Enter

فصل دوم: مبانی و دستور نحو

Syntax : یک سری قواعد نوشتاری هستند که باید آن‌ها را رعایت کرد تا برنامه به درستی اجرا شود

دو مدل line داریم :

۱- Physical line : خط هایی که در ادیتور میبینیم
۲- Logical line : خط هایی که مفسر آن‌ها را تفسیر میکند

سناریو اول تعداد خط های فیزیکی کمتر از منطقی باشد .برای این کار از ; استفاده میکنیم . به مثال زیر دقت شود :

```
print("hello");print("world")
```

سناریو دوم تعداد خط های فیزیکی بیشتر از منطقی باشد . برای این کار از\ استفاده میکنیم . به مثال زیر دقت شود :

```
x = 25 + 6 + 5 + 7 \
      + 41\
      + 6
```

برای لیست ها و تاپل ها و () و... لازم نیست از \ استفاده کنیم

```
x = [25 ,35 ,
     45 , 45
     ,98]
```

برای مستند سازی میتوانیم از comment ها و docstring ها استفاده کنیم

برای گذاشتن کامنت از # استفاده میکنیم

برای docstring از """""" استفاده میکنیم

```
# this is a comment
"""
this is a docstring
"""
```

Comment ها برای توسعه دهنده ها هستند ولی docstringها برای user ها

Docstring ها در تابع و کلاس‌ها نادیده نمیشود و میتوان آن ها را خواند :

```
def F():  1 usage
    """
    this is a docstring
    :return:
    """
    pass

print(F().__doc__)
```

Docstring خارج از کلاس و تابع = comment

Indentation: دندانه گذاری برای مشخص کردنه اینکه این تکه کد مطعلق به کدام تابع یا کلاس است

معمولا از چهار space استفاده میشود

```
def F1():
    def F2():
        def F3():
            def F4():
                def F5():
                    pass
```

یک آشنایی مختصر با print(),input() داشته باشیم که در ادامه به مشکل نخوریم

- Print تابعی برای نمایش روی console است

دو ورودی که در آن میخواهیم معرفی کنیم :

۱- Sep : برای مشخص کردن نوع separator
۲- End : برای مشخص کردن اینکه زمانی که print به اتمام رسید چکونه به پایان برسد

```
print(1, 2, 3, sep="*")
```

```
1*2*3
```

```
print(1, 2, 3, sep="*",end="||||")
point(1, 2, 3, sep="*")
```

```
1*2*3||||1*2*3
```

- Input برای گرفتن ورودی است که باید کاربر وارد کند . همچنین همزمان یک خروجی دارد که برای راهنمایی است که به صورت زیر است

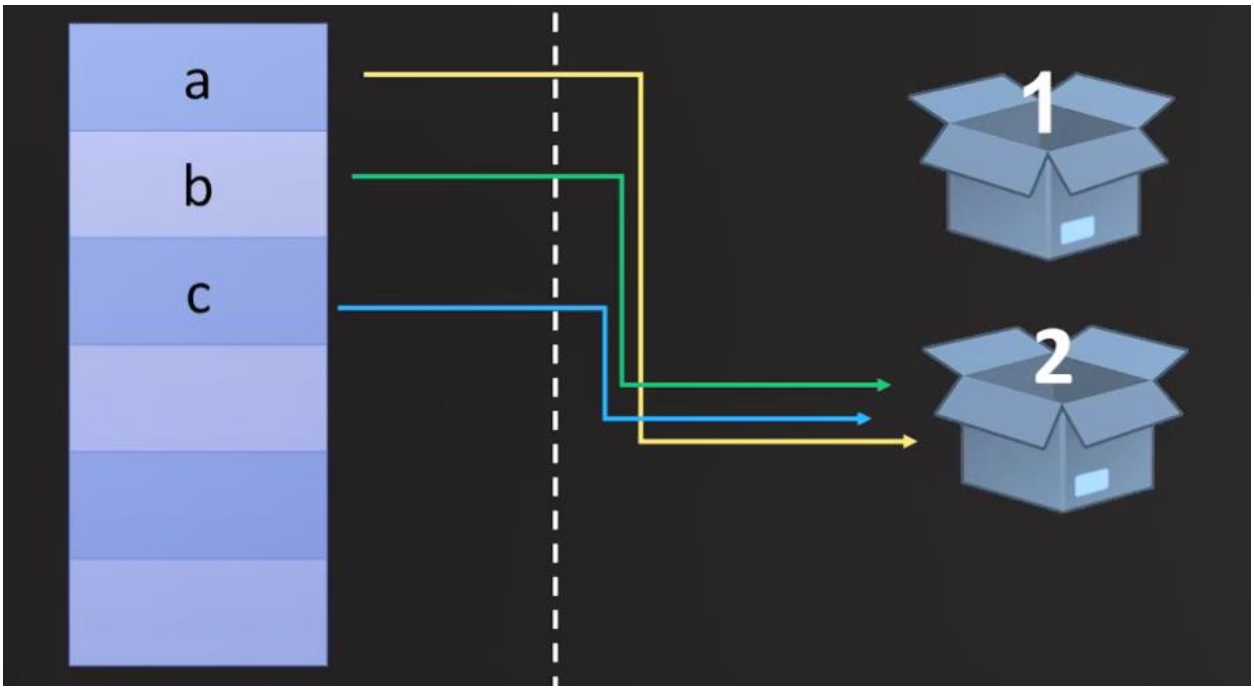
```
input("enter anything:")
```

```
enter anything
```

Variable : به یک مکان در حافظه اشاره دارد که می‌تواند داده‌ها را ذخیره کند. متغیرها نام‌هایی دارند که به آن‌ها نسبت داده می‌شود و می‌توانند مقادیر مختلفی را در طول اجرای برنامه به خود بگیرند. به عبارت دیگر، متغیرها ابزاری برای ذخیره‌سازی و مدیریت داده‌ها هستند که به برنامه‌نویس این امکان را می‌دهند تا با استفاده از نام متغیر، به داده‌ها دسترسی پیدا کند و آن‌ها را تغییر دهد.در ادامه با انواع آن آشنا خواهیم شد

در پایتون متغیرها به صورت اشاره‌گر هستند برای درک این موضوع به مثال زیر دقت شود

```
a = 1
a = 2
b = a
c = b
```



در صورت تغییر دادن a مقدار b,c هم تغییر میکند

برای مقدار دادن به متغیرها در یک خط میتوانیم به صورت a,b,c=1,2,3 بنویسیم

همچنین سوال معروف swap کردن را به صورت a,b=b,a میتوانیم پاسخ دهیم

Identifier:شناسه ها همان نام‌های متغیرهای ما هستند

شروع آن‌ها باید یا یک حروف انگلیسی یا _ باشد

بعد از آن میتوانیم نیز از اعداد هم استفاده کنیم

پایتون یک زبان case sensitive است

شناسه خصوصی یک ماژول _ در ابتد است مثل _a1

شناسه خصوصی یک کلاس به صورت __ است مثال __a2

یک سری متدهای خاص هم به صورت __name__ است مانند __doc__

قوانین pep8 برای نام گذاری :

۱- کلاس ها باید به صورت حروف بزرگ باشد :MyClass
۲- توابع باید به صورت حروف کوچک باشد و بین کلامات از _ استفاده شود : my_func
۳- متغیرها هم مانند توابع نام گذاری میشود

Key words : کلمات کلیدی کلماتی هستند که برای خود پایتون رزرو شده اند و نمیتوانیم برای نام متغیرها استفاده کنیم

برای دیدن لیست کلمات کلیدی میتوانیم از دستورات زیر استفاده کنیم :

```
import keyword

help("keywords")
print(keyword.kwlist)
```

Arithmetic operator : عملگرهایی که روی اعداد حسابی تعریف میشود که در پایین معرفی کردیم

+ : همان جمع بین دو عدد است

-: تفریق بین دو عدد است

* :ضرب بین دو عدد

/:تقسیم بین دو عدد

%:باقی مانده بین دو عدد

//:تقسیم بدون اعشار یا همان مقسوم علیه دو عدد

**: به توان رساندن یک عدد

Comparison operation : عملگرهای مقایسه ای برای مقایسه کردن دو عبارت حسابی است

== : مساوی

=! : نامساوی

< > : بزرگتر و کوچیک تر

=> =< : بزرگ تر مساوی و کوچیک تر مساوی

Logical operators : عملگرهای منطقی هستند که سنجش درستی یا نادرستی یک عبارت به کار میرود که خروجی آن false, true است

And : زمانی درست است که همه درست باشند

OR : زمانی درست است که حداقل یکی درست باشد

Not : زمانی درست که عبارت غلط باشد

Membership operator : برای چک کردن اینکه که آیا عضوی درون آن لیست یا تاپل یا رشته و... هست یا نه که خروجی آن false , true است

In : ایا درون آن مجموعه هست

Not in : ایا درون آن مجموعه نیست

Identifier operator : این عملگر به این صورت عمل میکند که اگر دو متغیر به یک خانه از حتفظه اشاره داشتند در آن صورت true برمیگرداند

Is : اگر به یک خانه از حافظه اشاره داشته باشد true

Is not : برعکس is است

مثال‌های زیر دقت شود :

```
a = 5
b = 5
print(a is b)
```

```
D:\python_s
True
```

برای int ها is و = یک معنی میدهد

```
a = [1,2,3]
b = [1,2,3]
print(a is b)
```

```
D:\python_
False
```

دو لیست شبیه به هم دو خانه مجزا از حافظه را اشغال مبکنند

```
a = [1,2,3]
b = a
print(a is b)
```

```
D:\python_
True
```

اگر به این صورت نوشته شود یگ خانه از حافظه اشغال میشود

Bitwise operator : عملگرهای بیتی هستند که به صورت بیت به بیت اعمال میشود

برای نوشتن یک عدد به صورت باینری به صورت x = 0b1011100 مینویسیم

برای تبدیل به باینری از دستور bin() استفاده میکنیم :print(bin(x))

Operator	Name	Description
&	AND	Sets each bit to 1 if both bits are 1
|	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

Assignment operator : عملگرهای انتسابی هستند

به طور خلاصه به صورت زیر هستند

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Walrus : این عملگر از نسخه پایتون 3.8 به بعد آمد این عملگر به شما امکان می‌دهد تا یک مقدار را در حین ارزیابی یک عبارت به یک متغیر نسبت دهید. این ویژگی می‌تواند کد را کوتاه‌تر و خواناتر کند.

```
x = 5
print(x:=10)
```

```
D:\python_s
10
```

در آینده بیشتر با آن آشنا خواهیم شد

اولویت عملگرها به صورت زیر است :

Level	Operator	Description
18	()	Grouping
17	f()	Function call
16	[index:index]	Slicing
15	[]	Array Subscription
14	**	Exponential
13	~	Bitwise NOT
12	+ -	Unary plus / minus
11	* / %	Multiplication Division Modulo
10	+ -	Addition / Subtraction
9	<< >>	Bitwise Left Shift Bitwise Right Shift
8	&	Bitwise AND
7	^	Bitwise XOR
6	|	Bitwise OR
5	in , not in, is , is not <, <=, >, >= == !=	Membership Relational Equality Inequality
4	not	Boolean NOT
3	and	Boolean AND
2	or	Boolean OR
1	lambda	Lambda Expression

بهتر است به جای حفظ جدول بالا از پرانتز گذاری استفاده کنیم که دچار مشکل نشویم

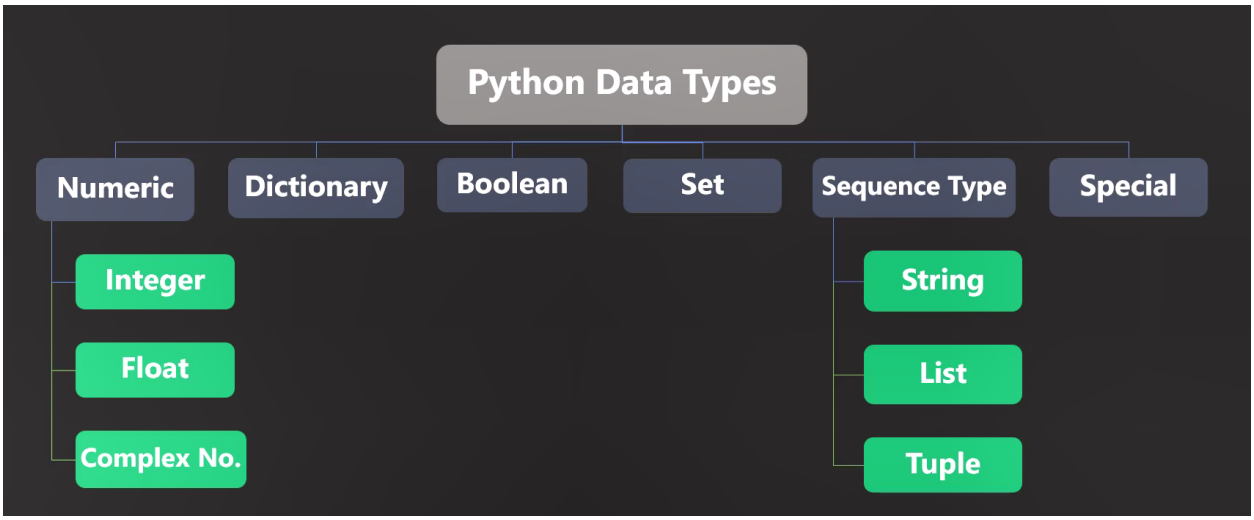
Pep : یک سری اصول و قوانینی که برای زیبایی و خوانا شدن کد است

برای دیدن این اصول باید import this را بزنیم و اجرا کنیم

با زدن ctrl + B میتوانیم وارد کدهای کتابخانه شویم و جالب این جاست که هیچکدام از این اصول رعایت نشده است

```
Vs gur vzcyrzragngvba vf uneq gb rkcynva, vg'f n onq vqrn.
Vs gur vzcyrzragngvba vf rnfl gb rkcynva, vg znl or n tbbq vqrn.
Anzrfcnprf ner bar ubaxvat terng vqrn -- yrg'f qb zber bs gubfr!"""

d = {}
for c in (65, 97):
    for i in range(26):
        d[chr(i+c)] = chr((i+13) % 26 + c)

print("".join([d.get(c, c) for c in s]))
```

فصل سوم: انواع داده (سطح یک)



انواع data type های ما به صورت بالا است و برای اینکه ببینیم که یک متغیر چه data type دارد print (type(x)) را بزنیم

دقت شود که پایتون به صورت خودکار data type ها را تغییر میدهد :

```
x = 5
print(type(x))
x += x + 0.2
print(type(x))
```

```
D:\python_sabzlern\1\.
<class 'int'>
<class 'float'>
```

- اعداد numeric

برای اینکه برحسب مبناهای مختلف اعداد را بنویسیم باید درابتدای آن

۱- برای یابنری 0b و برای نمایش آن از bin()
۲- برای اوکتاو 0o و برای نمایش آن از oct()
۳- برای هگز 0x و برای نمایش آن از hex()

```
x1 = 0b1001
x2 = 0o1254
x3 = 0x45a5
print(bin(x1), oct(x2), hex(x3))
```

```
D:\python_sabzlern\1\.venv
0b1001 0o1254 0x45a5
```

- برای تبدیل اعداد به int از تابع int() استفاده میکنیم
- برای تبدیل اعداد به float از تابع float() استفاده میکنیم

- برای تبدیل اعداد مختلط از تابع complex استفاده میکنیم
- چالش : اگر دو عدد 0.2 + 0.1 کنیم و جواب چاپ کنیم میبینیم که عدد نمایش داده شده دقیقا 0.3 نیست

```
x = 0.1 + 0.2
print(x)
```

```
D:\python_sabzlern\1\.venv
0.30000000000000004
```

برای حل این مشکل از کتابخانه decimal کمک میگیریم که کد آن در زیر آمده است :

```
from decimal import Decimal
number = float(Decimal("0.1")+Decimal("0.2"))
print(number)
```

```
D:\python_sabzlern\1\
0.3
```

- برای نمایش اعداد به صورت کسری میتوانیم از کتابخانه fraction بهره ببریم :

```
from fractions import Fraction
print(Fraction(1.5))
```

```
D:\python_sabzle
3/2
```

- برای جدا کردن اعداد به صورت سه تا سه تا میتوانیم از _ استفاده کنیم
- برای نمایش اعداد با نماد علمی میتوانیم از e استفاده کنیم که تایپ آن float است

```
x = 2e15
print(type(x))
```

```
D:\python_sabzlern\1\.ve
<class 'float'>
```

اعداد int محدودیت ندارند ولی floatها محدودیت دارند و اگر زیاد بزرگ باشد و یا به صورت float('inf') نوشته شود بی نهایت در نظر گرفته میشود

```
x = 2e400
y = float("inf")
print(x,y)
```

```
D:\python_sabzler
inf inf
```

- اعداد مختلط هم در پایتون داریم که متدهایی برای نمایش قسمت حقیقی و موهومی و مزدوج دارد

```
x = 2 + 3j
x1 = x.imag
x2 = x.real
x3 = x.conjugate()
print(x1, x2, x3)
```

```
D:\python_sabzlern\1\.venv\Scripts\
3.0 2.0 (2-3j)

Process finished with exit code 0
```

- برای گرد کردن اعداد از تابع round(x,y) استفاده میکنیم
- برای به توان رساندن اعداد از تابع pow() استفاده میکنیم

- String :

در پایتون برای نوشتن رشته میتوانیم از "" یا '' استفاده کنیم

- تابع isinstance برای این است که چک کنیم ایا نوع داده مدنظر درست است یا خیر که خروجی آن false ,true است

```
x = "Hossein Ghorbani"
print(isinstance(x, str))
```

```
D:\python_sabzlern\1\.ve
True
```

- کاربرد \ : back slash برای لغو کردن یک دستور است مثل \n

به طور مثال میخواهیم متن 'I'm Hossein' را بنویسیم اما مفسر نمیداند رشته در کجا به پایان رسیده برای این کار :

```
x = 'i\'m Hossein'
print(x)
```

```
x = 'c:\\user\\new'
print(x)
```

اگر میخواهیم کار بالا را به روش ساده تر کنیم قبل از رشته یک r میگذاریم

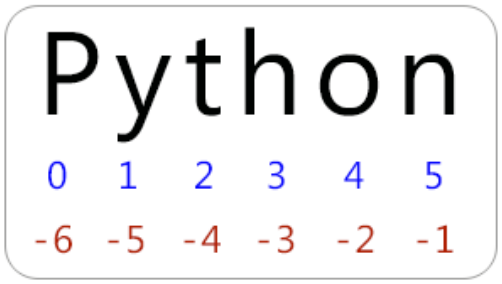
```
x = r'c:\user\new'
print(x)
```

- عملیات هایی مانند + و * بر روی رشته ها امکان پذیر است

```
print(4*"Hossein"+"Ghorbani")
```

```
HosseinHosseinHosseinHosseinGhorbani
```

- Index گذاری در پایتون از سمت راست به چپ از شماره صفر شروع میشود و از چپ به راست از شماره ۱-



برای دسترسی به index مورد نظر میتوانیم از [] استفاده کنیم

```
x = "python"
print(x[2],x[-3])
```

- در ادامه میخواهیم درباره slicing صحبت کنیم

برای این کار از [:] استفاده میکنیم

عنصر دوم خودش حساب نمیشود

```
x = "python"
print(x[-3:-1])
```

```
D:\pytho
ho
```

```
x = "python"
print(x[1:4])
```

```
D:\pytho
yth
```

اگر به صورت [::] بنویسیم عنصر سوم برای step های آن است

```
x = "python"
print(x[0:5:2])
```

```
D:\pyth
pto
```

همچنین میتوانیم اعداد منفی قرار دهیم که برعکس حرکت کنه (مثال زیر تکنیک inverse کردن است)

```
x = "python"
print(x[::-1])
```

```
D:\python_
nohtyp
```

اگر اعداد عناصر بزرگ تر از طول رشته باشد در آن صورت خطایی نمیگیرد و تا آخر چاپ میکند

- در ادامه میخواهیم با متدها و توابع مربوط به رشته آشنا شویم

۱- len() : تابعی است که طول رشته را برمیگرداند

۲- upper() : متدی است که حروف را بزرگ میکند ولی باید دقت شود که متغیر را باید ابدیت کرد

```
x = "python"
x = x.upper()
print(x)
```

```
D:\python_sa
PYTHON
```

۳- lower() : عکس upper است

۴- count() :متدی است که تعداد تکرار یک یا چندین کارکتر را به ما نمایش میدهد

```
x = "python python"
print(x.count("python"))
```

```
D:\py
2
```

۵- endswith() :این متد میگوید که آیا رشته با با کارکتر مورد نظر به پایان رسیده است یا خیر

۶- startswith() :این مانند متد بالا است با این تفاوت که شروع را چک میکند

۷- find() :این متد با گرفتن ورودی یک رشته یا تک کارکتر اولین رشته ای که با آن همخوانی داشته باشد برمیگرداند

```
x = "python python"
print(x.find("thon"))
```

```
D:\py
2
```

۸- rfind() :مانند find() است با این فرق که ار سمت راست شروع به جستوجو میکند

۹-isallnum() :چک میکند که آیا همه رشته عدد و حروف است(فارسی انگلیسی فرق ندارد)

۱۰-isnumeric: چک میکند که همه اعضا رشته عدد باشد

۱۱-join() : متدی است که عناصر تاپل یا لیست را بهم میچسباند و تشکیل یک رشته میدهد

```
x = ["p","y","t","h","o","n"]
s = "*".join(x)
print(s)
```

```
D:\python_sabzlern\1\
p*y*t*h*o*n
```

۱۲- split() : متدی است که عکس این کار را انجام میدهد و یک رشته را تبدیل به لیست میکند

```
x = "python,matlab,c++,c#"
l = x.split(",")
print(l)
```

```
D:\python_sabzlern\1\.venv\Scripts\pytho
['python', 'matlab', 'c++', 'c#']
```

۱۳-replice() : عناصر یک رشته را با عناصر دیگر جایگزین میکند

```
x = "python,matlab,c++,c#"
l = x.replace( _old: "matlab", _new: "java")
print(l)
```

```
D:\python_sabzlern\1\.venv\S
python,java,c++,c#
```

۱۴-strip,rstrip.lstrip : این متد فاصله های اضافی را از اول و اخر پاک میکند.دیفالت آن space است

```
x = "+++python,matlab,c++,c#++"
l = x.strip("+")
print(l)
```

```
D:\python_sabzlern\1\.venv\
python,matlab,c++,c#
```

۱۵- capitalize() : حرف اول str را بزرگ میکند

- Escape characters

خیلی از کارکترها در کیبرد نیست علاوه بر این یک سری کارکترهای کنترلی داریم که نمیتوان این ها را تایپ کرد

پس از استانداردهای ascii , Unicode استفاده میکنیم

برای مثال :

```
print("this is a test:\u01aa")
```

```
this is a test:ƪ
```

یک سری ها که کاربرد زیادی دارند برای راحتی از \ استفاده شده است :

۱- \n :برای رفتن به خط بعد
۲- \t : یک tab میدهد
۳- \b : معادله back space است
۴- \r : شروع خط را معلوم میکند

Ord(): تابعی است که با دادن کراکتری مانند a کد اسکی آن را میفهمیم

Chat() : برعکس آن است و با زدن عدد اسکی کاراکتر آن به ما داده میشود

- Traditional string formatting

یک روش قدیمی برای فرمت دادن به صورت زیر است

```
x = 25
print("this is number : %i" % (x))
```

فرمت کلی آن به صورت روبه رو است :

[(key)] [flag] [w] [.p] type

تایپ های مختلف که به جای i میتوان گذاشت عبارت است از

C : به معنی کارکتر که میتوانیم یک تک کارکتر یا اسکی کد قرار دهیم

int : I	octave : o
string : S	hexa : X
decimal : D	e : نماد علمی
F : float که تا شش رقم بعد اعشار نمایش میدهد	r : مانند str است که دور آن '' میگذارد

حال به معرفی بقیه بخش ها میپردازیم :

p. : تعداد ارقام بعد اعشار را مشخص میکند

```
x = 1.251546453
print("%.3f" % (x))
```

```
D:\python_s
1.252
```

W : طول میدان میگویند که اگر به طور مثال تعداد کارکترهای ما ۳ و طول میدان ۵ باشد به جای آن دو تا space قرار میدهد

Flag : سه حالت دارد

۱- اگر + باشد اعداد مثبت را پیش از آن یک علامت قرار میدهد

```
x = 1.251546453
print("%+5.3f" % (x))
```

```
D:\python_sabz
+1.252
```

۲- اگر صفر باشد به حای space برای طول میدان صفر قرار میدهد

۳- اگر – باشد طول میدان را از راست spaceقرار میدهد

Key برای دیکشنری ها است و به صورت زیر کار میکند :

```
x = {"a":1,"b":2,"c":3}
print("%(b)i"%x)
```

```
D:\python_
2
```

اگر بخواهیم از کاربر ورودی بگیریم به صورت زیر مینویسم :

```
number = 2.13546531
p = 2
w = 4
print("%*.*f" % (w, p, number))
```

```
D:\python
2.14

Process f
```

- Format

با یک مثال ساده شورع میکنیم

```
x = 254
print("x is {}".format(x))
```

```
D:\python_sabzl
x is 254

Process finishe
```

فرمت کلی آن به صورت زیر است

"{" [field_name] ["!" conversion] [":" format_spec] "}"

Filed name به حالت های زیر میتوان باشد

```
txt1 = "My name is {fname}, I'm {age}".format(fname = "John", age = 36)
txt2 = "My name is {0}, I'm {1}".format("John",36)
txt3 = "My name is {}, I'm {}".format("John",36)
```

conversation میتواند سه حالت باشد :

۱- S : به معنی string
۲- R : string که بین آن '' است
۳- A : بر اساس Unicode اگر جز ascii ها نباشد عدد آن را نمایش میدهد

```
print("x is {!a}".format("ə"))
```

```
x is '\u01dd'
```

حال میخواهیم از بعد دو نقطه صحبت کنیم

[[fill]align][sign][#][0][width][grouping_option][.precision][type]

۱. fill: این بخش مشخص می‌کند که چه کاراکتری برای پر کردن فضاهای خالی استفاده شود. به طور پیش‌فرض، این کاراکتر فضای خالی است، اما می‌تواند هر کاراکتری باشد، مانند صفر (۰) یا هر کاراکتر دیگری.

۲. align: این بخش نحوه تراز کردن مقدار را مشخص می‌کند. گزینه‌های معمول شامل:

- < : تراز چپ
- > : تراز راست
- ^ : تراز وسط
- = : تراز با پر کردن فضای خالی در سمت چپ عدد (برای اعداد)

۳. sign: این بخش تعیین می‌کند که آیا علامت منفی (-) برای اعداد منفی نمایش داده شود یا خیر. گزینه‌ها شامل:

- + : همیشه علامت را نشان بده
- - : فقط برای اعداد منفی علامت را نشان بده
- (فضای خالی) : برای اعداد مثبت فضایی را قبل از عدد قرار بده

۴. #: این علامت برای نمایش مقادیر هگزادسیمال یا باینری استفاده می‌شود. اگر این علامت وجود داشته باشد، پیشوند مناسب (مانند ۰x برای هگزادسیمال) به عدد اضافه خواهد شد.

۵. ۰: اگر در اینجا یک صفر وجود داشته باشد، به این معنی است که از صفر برای پر کردن فضای خالی استفاده شود (به عنوان مثال، برای اعداد صحیح).

۶. width: این عدد مشخص می‌کند که حداقل عرض نهایی رشته خروجی چقدر باید باشد. اگر طول رشته کمتر از این مقدار باشد، با استفاده از کاراکتر پرکننده (که در بخش fill مشخص شده) فضاهای خالی پر خواهد شد.

۷. grouping_option: این گزینه برای گروه‌بندی اعداد بزرگ استفاده می‌شود. به عنوان مثال، در اعداد بزرگ ممکن است بخواهید از کاما (,) برای جدا کردن هزارگان استفاده کنید.

۸. .precision: این قسمت برای تعیین دقت اعشاری در اعداد اعشاری (float) استفاده می‌شود. به عنوان مثال، ۲. نشان‌دهنده دو رقم اعشار است.

۹. type: این بخش نوع داده‌ای که می‌خواهید فرمت‌دهی کنید را مشخص می‌کند. برخی از انواع رایج عبارتند از:

- d : عدد صحیح
- f : عدد اعشاری
- s : رشته
- x : عدد هگزادسیمال

به طور کلی، این ساختار به شما امکان می‌دهد تا خروجی‌های خود را با دقت و زیبایی بیشتری فرمت‌دهی کنید و کنترل بیشتری بر نحوه نمایش داده‌ها داشته باشید.

اگر بخواهیم مانند قبل به طور مثال تعداد نقاط بعد اعشار را معلوم کنیم به صورت زیر عمل میکنیم :

```
x = 2.25

print("x is {0:0{align}15.3f}".format( *args: x,align = "^"))
```

```
D:\python_sabzlern\1\.
x is 000002.25000000
```

- F string

از دو فرمت قبلی f string نسبتا جدیدتر است . یک مثال ساده در زیر آمده است :

```
x = 2.25

print(f"this is x {x}")
```

```
D:\python_sabzlern
this is x 2.25
```

اگر بخواهیم به صورت {x} = 2.25 بنویسیم آنگاه باید به صورت زیر عمل کنیم :

```
x = 2.25

print(f"this is {{x}} {x}")
```

```
D:\python_sabzlern\1\.venv\
this is {x} 2.25
```

میتوان همان قواعد بالا را بعد از دو نقطه پیاده سازی کرد :

```
x = 2.25

print(f"this is {{x}} {x:.5f}")
```

```
D:\python_sabzlern\1\.ven
this is {x} 2.25000
```

برای نمایش تاریخ و فرمت دهی آن باید از کتابخانه datetime استفاده کنیم

```
from datetime import datetime
now = datetime.now()
print(f"{now:%Y/%m/%d}")
```

```
D:\python_sabzlern
2025/06/01
```

- List :

لیست‌ها در پایتون یکی از ساختارهای داده‌ای مهم و پرکاربرد هستند که برای ذخیره مجموعه‌ای از مقادیر استفاده می‌شوند. لیست‌ها می‌توانند شامل انواع مختلف داده‌ها، از جمله اعداد، رشته‌ها و حتی دیگر لیست‌ها باشند.

ویژگی‌های اصلی لیست‌ها در پایتون عبارتند از:

۱. قابل تغییر (Mutable): شما می‌توانید عناصر یک لیست را تغییر دهید، اضافه کنید یا حذف کنید.
۲. ترتیب‌دار (Ordered): عناصر لیست به ترتیبی که اضافه شده‌اند، ذخیره می‌شوند و قابل دسترسی هستند.
۳. ایندکس‌پذیری: هر عنصر در لیست با یک ایندکس (شمارنده) مشخص می‌شود که از ۰ شروع می‌شود.

- به سه روش میتوانیم لیست ها را ایجاد کنیم که استفاده از دستور list,split,[] است که در زیر آمده شده :

```
list1 = ["p","y","t","h","o","n"]
list2 = list("python")
list3 = "p y t h o n".split()
print(list1)
print(list2)
print(list3)
```

```
D:\python_sabzlern\1\.venv\Scripts\pyth
['p', 'y', 't', 'h', 'o', 'n']
['p', 'y', 't', 'h', 'o', 'n']
['p', 'y', 't', 'h', 'o', 'n']
```

- مبحث indexing , slicing همانند رشته ها در لیست ها وجود دارد

```
list1 = ["p","y","t","h","o","n"]
print(list1[5])
print(list1[2:7:2])
```

```
D:\python_sabzler
n
['t', 'o']
```

- اپراتورهایی مانند in,==,in,+,... روی لیست ها عمل میکند

```
list1 = ["p","y","t","h","o","n"]
list1 = list1 + ["c++"]
print(list1)
print("p" in list1)
```

```
D:\python_sabzlern\1\.venv\Scripts\pyth
['p', 'y', 't', 'h', 'o', 'n', 'c++']
True

Process finished with exit code 0
```

- وقتی دو لیست را برابر قرار میدهیم ادرس‌های آن یکسان است

```
list1 = ["p","y","t","h","o","n"]
list2 = list1
print(id(list1))
print(id(list2))
```

```
D:\python_sabzlern\1\.venv\Sc
1778587708096
1778587708096
```

حال تغییری در هر یک موجب تغییر در دیگری میشود

```
list1 = ["p","y","t","h","o","n"]
list2 = list1
list2[0] = 'a'
print(list1)
print(list2)
```

برای جلوگیری از این اتفاق میتوانیم از متد copy استفاده کنیم

```python
list1 = ["p","y","t","h","o","n"]
list2 = list1.copy()
list2[0] = 'a'
print(list1)
print(list2)
```

```
D:\python_sabzlern\1\.venv\Scripts\p
['p', 'y', 't', 'h', 'o', 'n']
['a', 'y', 't', 'h', 'o', 'n']
```

- اگر لیست‌ها و رشته های تو درتو داشته باشیم برای اینکه به عناصر آن دست پیدا کنیم باید براکت های پشت سرهم قرار دهیم

```python
list1 = [1, 2, 3, [1, 2, 'python']]
print(list1[3][2][0])
```

```
D:\python_sa
p
```

- حال اگر لیست‌های تودرتو داشته باشیم و تغییری در یکی از لیست‌های داخلی رخ دهد و از آن کپی گرفته باشیم تغییر در هم لیست اصلی و هم کپی شده رخ میدهد

دو نوع کپی داریم :

۱- Shallow copy
۲- Deep copy

این اتفاق ناخوشایند به دلیل shallow copy است

برای حل این مشکل از deep copy کمک میگیریم

```python
import copy
list1 = [1, 2, 3, [1, 2, 'python']]
list2 = copy.deepcopy(list1)
```

- همانطور گفته شد میتوانیم لیست ها را تغییر دهیم که این به صورت یک عنصر یا بازه ای از اعضا لیست خواهد بود

```python
list1 = [1,2,3,4,5]
list1[0] = "s"
print(list1)
```

```
D:\python_sabzlern\1
['s', 2, 3, 4, 5]
```

```python
list1 = [1, 2, 3, 4, 5]
list1[1:3] = [12, 13, 14]
print(list1)
```

```
D:\python_sabzlern\1\.v
[1, 12, 13, 14, 4, 5]

Process finished with
```

- Append این متد برای اضافه کردن یک عنصر به لیست
- del برای پاک کردن یک عنصر و یا کل متغیر
- Len برای به دست آوردن طول لیست

```
list1 = [1, 2, 3, 4, 5]
del list1[0]
list1.append(6)
print(list1)
print(len(list1))
```

```
D:\python_sabzlern\1\.ver
[2, 3, 4, 5, 6]
5
```

- انتصاب چندگانه

میتوانیم چند متغیر را به صورت همزمان مقدار دهی کنیم

```
a, b, c = [1, 2, 3]
print(a, b, c)
```

```
D:\python
1 2 3
```

اگر اعضای لیست بیشتر از متغیرها بود میتوانیم به صورت زیر بنویسیم

```
a, b, *c = [1, 2, 3, 4, 5]
print(a, b, c)
```

```
D:\python_sabzlern\1\.
1 2 [3, 4, 5]
```

- Tuple

تاپل‌ها (Tuples) در پایتون یکی از ساختارهای داده‌ای هستند که مشابه لیست‌ها عمل می‌کنند، اما با یک تفاوت کلیدی: تاپل‌ها غیرقابل تغییر (immutable) هستند. این به این معنی است که پس از ایجاد یک تاپل، نمی‌توانید عناصر آن را تغییر دهید، اضافه کنید یا حذف کنید.

ویژگی‌های اصلی تاپل‌ها عبارتند از:

۱. غیرقابل تغییر (Immutable): پس از تعریف، نمی‌توان عناصر را تغییر داد.
۲. ترتیب‌دار (Ordered): مانند لیست‌ها، عناصر در تاپل‌ها به ترتیبی که اضافه شده‌اند، ذخیره می‌شوند.
۳. ایندکس‌پذیری: می‌توانید به عناصر با استفاده از ایندکس دسترسی پیدا کنید.

- برای ایجاد تاپل ها میتوانیم از پرانتزاستفاده کنیم یا بدون پرانتز

```python
t1 = (1, 2, 3)
t2 = 4, 5, 6
print(type(t1), type(t2))
```

```
D:\python_sabzlern\1\.venv\Scrip
<class 'tuple'> <class 'tuple'>
```

- مباحث slicing,indexing برای تاپل ها نیز برقرار است
- اگر لیستی درون تاپل باشد آن لیست قابل تغییر است و برای اینکه از آن یک کپی میگیریم باید از deepcopy استفاده کنیم
- برای ایجاد یک تاپل تک عضوی باید به صورت (1,) بنویسیم
- مبحث انتصاب چندگانه در تاپل ها مانند لیست ها هست فقط دقت شود به صورت لیست انتصاب داده میشود نه تاپل

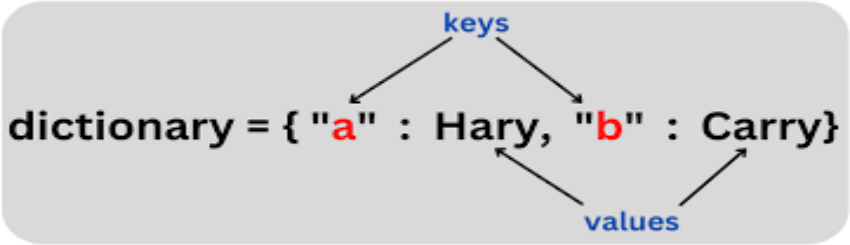
```python
a, b, *c = 1, 2, 3, 4, 5
print(type(a),type(b),type(c))
```

```
D:\python_sabzlern\1\.venv\Scripts\python.exe D:
<class 'int'> <class 'int'> <class 'list'>
```

- Dictionary

دیکشنری‌ها (Dictionaries) در پایتون یکی از ساختارهای داده‌ای بسیار مهم و پرکاربرد هستند که به صورت مجموعه‌ای از جفت‌های کلید-مقدار (key-value pairs) عمل می‌کنند. ویژگی‌های اصلی دیکشنری‌ها عبارتند از:

۱. غیرترتیبی (Unordered): دیکشنری‌ها ترتیب خاصی برای ذخیره‌سازی عناصر ندارند. از نسخه ۳.۷ به بعد، دیکشنری‌ها به صورت حفظ ترتیب اضافه شده‌اند، اما به طور کلی می‌توان گفت که دیکشنری‌ها مانند لیست‌ها یا تاپل‌ها ترتیب ثابت ندارند.
۲. قابل تغییر (Mutable): می‌توانید عناصر جدیدی را به دیکشنری اضافه کنید، عناصر موجود را تغییر دهید یا حذف کنید.
۳. کلیدهای منحصر به فرد: هر کلید در دیکشنری باید منحصر به فرد باشد و می‌تواند از انواع داده‌ای مختلفی مانند رشته، عدد یا تاپل استفاده کند.



- برای ساخت دیکشنری میتوانیم از { } استفاده کنیم . دیکشنری اینگونه است که به صورت کلید مقدار است
- کلید ها باید نوع داده هایی باشد که غیرقابل تغییر باشد حتی تاپلی که درون آن لیست است نمیتواند کلید باشد
- مقدارها هرمقداری میتواند باشد
- اگر دو کلید با نام یکسان داشته باشیم مقدار دوم آن را درنظر میگیرد

```python
d = {"a": 1, "b": 2, "c": 3, "a": 4}
print(d["a"])
```

```
D:\python_sabzle
4
```

- اگر کلیدی را بنویسیم و مقدار آن نباشد در آن صورت آن به دیشکنری اضافه میشود

```
d = {"a": 1, "b": 2, "c": 3}
d["d"] = 4
print(d)
```

```
D:\python_sabzlern\1\.venv\Scripts\p
{'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

- چهار متد پرکاربرد :
  ۱- Keys() : کلید ها را به صورت تایپ dict_keys برمیگرداند
  ۲- Values() : مقدارها را به صورت dict_values میدهد
  ۳- Items() : جفت کلید و مقدار را به صورت dict_items میدهد
  ۴- Get() : اگر مقدار کلیدی را بخواهیم که آن کلید وجود نداشته باشد در آن صورت به جای خطا مقدار null میدهد

```
d = {"a": 1, "b": 2, "c": 3}

print(d.keys())
print(d.values())
print(d.items())
```

```
D:\python_sabzlern\1\.venv\Scripts\python.e
dict_keys(['a', 'b', 'c'])
dict_values([1, 2, 3])
dict_items([('a', 1), ('b', 2), ('c', 3)])
```

- برای ایجاد دیکشنری قبل هم گفتیم میتوانیم از {} استفاده کنیم . روش دیگر برای ایجاد آن استفاده از تاپل ها است

```
d = dict([("a",1),("b",2),("c",3)])
print(d)
```

```
D:\python_sabzlern\1\.venv\
{'a': 1, 'b': 2, 'c': 3}
```

اگر همه کلید ها به صورت رشته باشد راه ساده تر این است که به صورت زیر نوشته شود :

```
d = dict(a=1,b=2,c=3)
print(d)
```

```
D:\python_sabzlern\1\.venv\Sc
{'a': 1, 'b': 2, 'c': 3}
```

دقت شود در این روش کلیدها نمیتواند تکراری باشد

- مباحث مربوط به deepcopy هم صادق است
- اپراتورهای + * برای دیکشنری کار نمیکند
- Zip دو تاپل و یا دو لیست ارتباط برقرار میکند و میتوان از آن برای ساخت دیکشنری استفاده کرد

```
a = (1, 2, 3)
b = (4, 5, 6)
c = dict(zip(a, b))
print(c)
```

```
D:\python_sabzlern\1
{1: 4, 2: 5, 3: 6}
```

- Set

در پایتون، set یک نوع داده‌ای است که برای ذخیره مجموعه‌ای از عناصر منحصر به فرد استفاده می‌شود. مجموعه‌ها در پایتون غیرقابل تغییر (mutable) هستند، به این معنی که می‌توانیم عناصر جدیدی به آن‌ها اضافه کنیم یا عناصر موجود را حذف کنیم، اما خود مجموعه نمی‌تواند به طور مستقیم تغییر کند (یعنی نمی‌توانیم یک عنصر خاص را تغییر دهیم).

ویژگی‌های set در پایتون:

۱. غیر تکراری بودن عناصر: هر عنصر تنها یک بار در مجموعه وجود دارد. اگر سعی کنید یک عنصر تکراری اضافه کنید، فقط یک نمونه از آن حفظ خواهد شد.

۲. غیر مرتب بودن: عناصر در یک مجموعه ترتیب خاصی ندارند. بنابراین، نمی‌توان به عناصر با استفاده از ایندکس دسترسی پیدا کرد.

۳. نوع داده‌ای قابل تغییر: می‌توانید عناصر را به مجموعه اضافه یا از آن حذف کنید.

- عناصر مجموعه باید غیرقابل تغییر باشند
- برای اضافه کردن یک عنصر از add() استفاده میکنیم

```
s = {1,2,3,4,5}
s.add(6)
print(s)
```

```
D:\python_sabzlern\1\.venv\
{1, 2, 3, 4, 5, 6}
```

- {} خالی را دیکشنری در نظر میگیرد برای اینکه مجموعه در نظر بگیرد باید از عبارت set() استفاده کنیم
- تفاوت add(), update() را در زیر میتوانیم ببینیم

```
s1 = {1,2,3,4,5}
s1.add((6,7))
print(s1,"\n","----------------------")
s2 = {1,2,3,4,5}
s2.update((6,7))
print(s2,"\n","----------------------")
```

```
D:\python_sabzlern\1\.venv\Scrip
{1, 2, 3, 4, 5, (6, 7)}
 ----------------------
{1, 2, 3, 4, 5, 6, 7}
 ----------------------
```

- برای پاک کردن یک عنصر میتوانیم از remove , discard استفاده کنیم فرق این دو این است که اگر عنصری که خواستیم پاک کینم وجود نداشت remove خطا میدهد ولی discard نه
- برای تفاضل بین مجموعه ها میتوانیم از دو روش زیر استفاده کنیم

```
a = {1,2,3,4}
b = {3,4,5,6}
print(a-b)
print(a.difference(b))
```

```
D:\python_
{1, 2}
{1, 2}
```

- برای اجتماع دو روش زیر را داریم :

```
a = {1,2,3,4}
b = {3,4,5,6}
print(a|b)
print(a.union(b))
```

```
D:\python_sabzlern\1\.
{1, 2, 3, 4, 5, 6}
{1, 2, 3, 4, 5, 6}
```

- برای اشتراک نیز :

```
a = {1,2,3,4}
b = {3,4,5,6}
print(a&b)
print(a.intersection(b))
```

```
D:\python
{3, 4}
{3, 4}
```

- برای تفاضل قرینه (اجتماع دو مجموعه ، منهای اشتراک آن ها) :

```
a = {1,2,3,4}
b = {3,4,5,6}
print(a^b)
print(a.symmetric_difference(b))
```

```
D:\python_sabzlern\1\.venv\Scr
{1, 2, 5, 6}
{1, 2, 5, 6}
```

برای بررسی زیر مجموعه :

```
a = {1,2,3,4}
b = {3,4,}
print(a>b)
print(a.issuperset(b))
print(b.issubset(a))
```

```
D:\python_sa
True
True
True
```

- None:

در پایتون، None یک شیء خاص است که نشان‌دهنده عدم وجود مقدار یا عدم تعریف است. این شیء به عنوان یک نوع داده‌ی خاص (NoneType) شناخته می‌شود و تنها یک نمونه از آن وجود دارد

- Bool :

در پایتون، نوع داده‌ای به نام bool وجود دارد که برای نمایش مقادیر منطقی (Boolean) استفاده می‌شود. این نوع داده تنها دو مقدار ممکن دارد: True و False. این مقادیر به طور گسترده در برنامه‌نویسی برای کنترل جریان برنامه و انجام عملیات منطقی استفاده می‌شوند.

تعریف bool

در پایتون، bool یک نوع داده‌ی پایه است که نمایانگر دو حالت منطقی است:

- True: به معنای درست
- False: به معنای نادرست

ایجاد مقادیر Boolean

مقادیر Boolean به سادگی با استفاده از کلمات کلیدی True و False تعریف می‌شوند

- مقادیر false عبارت است از :
  ۱- شی صفر(0 , 0.0 ,0j)
  ۲- False
  ۳- None
  ۴- ()
  ۵- “”
  ۶- []
  ۷- {}
  ۸- Set()

- and or with data type

اگر یک سمت and or یک data type باشد در آن صورت false true برگشته نمیشود

```
x = 5
y = 3
print(x and y)
```

```
D:\pytho
3
```

قاعده آن برای and به صورت زیر است :

If x false return x

If x true return y

قاعده برای or :

If x false return y

If x true return x

برای ترکیبی ها که یک سمت data type است سمت دیگر عبارت منطقی قواعد بالا صادق است

فصل چهارم: دستورات کنترلی (تصمیم)

- multiple input in line

برای گرفتن چند ورودی در یک خط میتوانیم از تکه خط کد زر استفاده کنیم

```
x = (input("Enter a number: ").split())
```

نوع داده این لیست int است برای تبدیل به یک نوع داده دیگر میتوانیم از for کمک گرفت

```
x = [int(i) for i in(input("Enter a number: ").split())]
```

- decision :

در پایتون، ساختارهای کنترلی if, elif و else برای تصمیم‌گیری در برنامه‌ها استفاده می‌شوند. این ساختارها به شما این امکان را می‌دهند که بر اساس شرایط مختلف، بخش‌های متفاوتی از کد را اجرا کنید. در ادامه به توضیح هر یک از این ساختارها می‌پردازیم:

۱. دستور if

دستور if برای بررسی یک شرط خاص استفاده می‌شود. اگر این شرط درست (True) باشد، بلوک کد مربوط به آن اجرا می‌شود.

۲. دستور elif

دستور elif (که مخفف "else if" است) به شما این امکان را می‌دهد که چندین شرط را بررسی کنید. اگر شرط اول (if) نادرست باشد، پایتون به بررسی شرط‌های elif می‌پردازد. فقط اولین شرط صحیح اجرا خواهد شد.

۳. دستور else

دستور else برای زمانی استفاده می‌شود که هیچ‌یک از شرایط قبلی (if و elif) درست نباشند. اگر همه‌ی شرایط نادرست باشند، کد داخل بلوک else اجرا خواهد شد.

- اگر if های زیر هم بنویسیم در آن صورت تک تک آن ها چک میشود و ممکن است دوتای آن ها درست باشد ولی اگر elif قرار دهیم اگر یکی از آن ها درست بود بقیه چک نمیشود

- Min , max , sum

در این بخش میخواهیم تابع های پرکاربرد را بررسی کنیم

min : برای به دست آوردن کوچک ترین عدد که میتواند یک سری عدد پشت سرهم باشد یا لیست و تاپل . همچنین میتواند روی رشته‌ها مقایسه انجام شود

```
print(min((1, 2, 3), default=20))
```

```
D:\python_sabzlern\1\
1
```

یک متد defult دارد که اگر خالی بود آن را به عنوان min در نظر میگیرد

max : مانند min است با این فرق که بزرگ ترین را بررسی میکند

```
print(max((), default=20))
```

```
D:\python_sa
20
```

sum: مجموع یک لیست یا تاپل و یا مجموعه را میدهد .یک متد start دارد که مجموع را با آن جمع میکند.مانند یک بایاس

```
print(sum((1,2,3,),start=10))
```

```
16
```

فصل پنجم: دستورات کنترلی (تکرار)

- while

دستور while در پایتون یکی از ابزارهای کنترلی است که به شما این امکان را می‌دهد تا یک بلوک کد را به‌طور مکرر اجرا کنید، تا زمانی که یک شرط خاص برقرار باشد. این نوع حلقه به‌ویژه زمانی مفید است که تعداد تکرارها از قبل مشخص نیست و به یک وضعیت خاص یا ورودی کاربر بستگی دارد.

while شرط:

بلوک کد برای اجرا #

• شرط: یک عبارت بولی است که در هر بار اجرای حلقه بررسی می‌شود. اگر این شرط True باشد، بلوک کد داخل حلقه اجرا می‌شود.

• بلوک کد: شامل دستورات و عملیاتی است که باید در هر تکرار انجام شوند. این بلوک می‌تواند شامل چندین خط کد باشد و باید با استفاده از فاصله (indentation) مناسب مشخص شود.

- اگر در حلقه braek بنویسیم در آن صورت حلقه شکسته میشود و همان‌جا متوقف میشود :

```
n = 0
while True:
    if n==5:
        break
    print(n,end="\t")
    n+=1
```

```
Run    test5
D:\python_sabzlern\1\.ven
0    1    2    3    4
Process finished with exi
```

- اگر در حلقه continue بنویسیم ادامه آن یک loop را درنظر نمیگیرد و به سراغ loop بعدی میرود

```
n = 0
while n != 10:
    if n == 5:
        n += 1
        continue
    print(n, end="\t")
    n += 1
```

```
Run    test5
D:\python_sabzlern\1\.venv\Scripts\py
0    1    2    3    4    6    7    8    9
Process finished with exit code 0
```

- اگر حلقه طوری اجرا شود که به هیچ break برخورد نکند در آن صورت اگر بعد حلقه else نوشته باشیم آن else اجرا میشود

```
n = 0
while n != 10:
    if n == 20:
        break
    print(n, end="\t")
    n += 1
else:
    print()
    print("this else ")
```

```
Run    test5
D:\python_sabzlern\1\.venv\Scripts\pytho
0    1    2    3    4    5    6    7    8    9
this else

Process finished with exit code 0
```

- For :

حلقه for در پایتون یکی از ابزارهای کنترلی است که به شما این امکان را می‌دهد تا بر روی یک دنباله (sequence) مانند لیست، تاپل، رشته یا هر نوع دیگر از دنباله‌ها تکرار کنید. این حلقه به‌ویژه برای تکرار بر روی مجموعه‌ای از عناصر بسیار مفید است.

for متغیر in دنباله:

بلوک کد برای اجرا #

• متغیر: نامی است که برای هر عنصر از دنباله در هر تکرار استفاده می‌شود. در هر بار اجرای حلقه، این متغیر به عنصر بعدی دنباله اختصاص داده می‌شود.

• دنباله: مجموعه‌ای از عناصر است که می‌خواهید بر روی آن تکرار کنید. این می‌تواند شامل لیست‌ها، تاپل‌ها، رشته‌ها، دیکشنری‌ها و حتی اشیاء قابل تکرار دیگر باشد.

• بلوک کد: شامل دستورات و عملیاتی است که باید در هر تکرار انجام شوند و باید با استفاده از فاصله (indentation) مناسب مشخص شود.

- مباحث مربوط به else , break , continue برای for هم مانند while برقرار است
- به کد زیر دقت شود :

```
for a, b, c in [[1, 2, 3], [4, 5, 6], [7, 8, 9]]:
    print(a, b, c)
```

```
D:\python_s
1 2 3
4 5 6
7 8 9
```

همانطور که میتوان دید abc همزمان به 1,2,3 و بعد به 4,5,6 و... اشاره دارد

- Range :

- تابع range() در پایتون یکی از توابع بسیار مفید است که برای تولید دنباله‌ای از اعداد استفاده می‌شود. این تابع به‌ویژه در حلقه‌ها، به‌خصوص در حلقه‌های for، کاربرد زیادی دارد.

تابع range() می‌تواند به سه صورت استفاده شود:

۱. با یک آرگومان:

range(stop)

- stop: عددی است که دنباله تا آن عدد (غیرشامل) تولید می‌شود.
- مثال: range(۵) دنباله‌ای از اعداد ۰ تا ۴ را تولید می‌کند.

۲. با دو آرگومان:

range(start, stop)

- start: عددی است که دنباله از آن شروع می‌شود.
- stop: عددی است که دنباله تا آن عدد (غیرشامل) تولید می‌شود.
- مثال: range(۲, ۶) دنباله‌ای از اعداد ۲ تا ۵ را تولید می‌کند.

۳. با سه آرگومان:

range(start, stop, step)

- start: عددی است که دنباله از آن شروع می‌شود.
- stop: عددی است که دنباله تا آن عدد (غیرشامل) تولید می‌شود.
- step: مقدار افزایشی است که به هر عدد در دنباله اضافه می‌شود.

برای اینکه دنباله اعداد را ببینیم میتوانیم آن را تبدیل به لیست کنیم و پرینت کنیم

```
print(list(range(10)))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- For tips

۱- اشنایی با enumerate :

تابع enumerate() در پایتون یک تابع بسیار مفید است که به شما این امکان را می‌دهد تا در حین تکرار بر روی یک دنباله (مانند لیست یا رشته)، هم به عناصر آن دنباله دسترسی پیدا کنید و هم به اندیس (شماره) هر عنصر. این تابع معمولاً در حلقه‌های for استفاده می‌شود و باعث می‌شود کد شما ساده‌تر و خواناتر شود.

```
l = ["ali" ,"reza" ,"sara"]
print(list(enumerate(l)))
```

```
D:\python_sabzlern\1\.venv\Scripts\pyth
[(0, 'ali'), (1, 'reza'), (2, 'sara')]
```

میتوانیم برای حلقه از ازآن استفاده کنیم :

```
l = ["ali" ,"reza" ,"sara"]

for a,b in enumerate(l):
    print(f"name:{b},index:{a}")
```

```
D:\python_sabzlern\1\.ver
name:ali,index:0
name:reza,index:1
name:sara,index:2
```

میتوانیم در آرگامان دوم بایاس دهیم مثلا به جای شروع از صفر از ۵ شروع شود

- از reversed برای معکوس کردن و از sorted برای مرتب کردن میتوانیم استفاده کنیم

```
l = [5, 3, 1, 2, 4]
for i in reversed(sorted(l)):
    print(i,end="\t")
```

```
D:\python_sabzlern\1\
5	4	3	2	1
Process finished with
```

- Random

برای تولید اعداد تصادفی میتوانیم یکی از توابع زیر را استفاده کنیم :

۱- random : برای تولید عددی تصادفی بین [0 , 1)

```
from random import random
print(random())
```

```
D:\python_sabzlern\1
0.2825326514227503
```

۲- uniform : برای تولید تصادفی float بین دو بازه

```
from random import uniform
print(uniform( a: 1, b: 10))
```

```
D:\python_sabzlern\1\.venv
5.961315228886342
```

۳- randint : برای تولید تصادفی صحیح بین دو بازه

```python
from random import randint
print(randint( a: 1, b: 10))
```

```
D:\python
7
```

۴- randrange : اگر بخواهیم گام هم معلوم کنیم میتوانیم از randrange استفاده کنیم

```python
from random import randrange
print(randrange( start: 1, stop: 10, step: 2))
```

```
D:\python_
3
```

۵- choice : برای انتخاب رندوم از لیست یا تاپل یا ...

```python
from random import choice
l1 = ["a","b","c","d","e","f"]
for _ in range(100):
    print(choice(l1))
```

```
D:\python_s
d
e
f
a
```

۶- sample : برای انتخاب یک زیر مجموعه به صورت رندوم

```python
from random import sample
l1 = ["a","b","c","d","e","f"]
for _ in range(100):
    print(sample(l1, k: 4))
```

```
D:\python_sabzlern\1\.ve
['e', 'a', 'd', 'b']
['a', 'f', 'd', 'e']
['b', 'e', 'f', 'a']
['f', 'd', 'a', 'c']
```

۷- shuffle : برای بهم ریختن ترتیب لیست و تاپل و ... باید دقت شود که این عمل روی خود لیست اعمال میشود

```python
from random import shuffle
l1 = ["a","b","c","d","e","f"]
for _ in range(100):
    shuffle(l1)
    print(l1)
```

```
D:\python_sabzlern\1\.venv\Scripts\p
['e', 'd', 'b', 'f', 'c', 'a']
['d', 'e', 'c', 'b', 'a', 'f']
['b', 'a', 'e', 'c', 'f', 'd']
['c', 'f', 'a', 'e', 'd', 'b']
['e', 'b', 'f', 'd', 'a', 'c']
```

۸- seed : برای اینکه اعداد رندوم در هر اجرا عوض نشود از seed کمک میگیریم

فصل ششم: تابع

- Function synatax

تابع‌ها در پایتون بلوک‌های کدی هستند که می‌توانند چندین بار فراخوانی شوند و به شما این امکان را می‌دهند که کد خود را سازماندهی و مدیریت کنید. تعریف یک تابع در پایتون با استفاده از کلمه کلیدی def انجام می‌شود. در زیر یک توضیح کلی و یک مثال ساده برای تعریف و استفاده از تابع آورده شده است.

: Syntax

```
def func (parameter1,parameter2,...):
        Body function
        return result
```

- اگر return نوشته نشود پایتون به صورت خودکار None برمیگرداند
- برای نام گذاری تابع بهتر است هر کلمه با _ جدا شود و با حروف کوچک شروع شود
- pass : برای اینکه بدنه تابع را بعد بنویسی و خطا نگیرد از pass استفاده میکنیم

- Argument syntax

۱- Normal : بر اساس موقعیت ارگمان‌ها مقدار دهی میشود :

```
def func(a,b,c):  1 usage
    print(a)
    print(b)
    print(c)

func( a: 1,  b: 2,  c: 3)
```

```
D:\python_sab
1
2
3

Process finis
```

۲- Name value : بر اساس نام‌دهی مقدار دهی میشود

```
def func(a,b,c):  1 usage
    print(a)
    print(b)
    print(c)

func(b=2,a=1,c=3)
```

```
D:\python_
1
2
3
```

۳- Normal + name value : ترکیب مورد اول و دوم است . باید دقت شود که باید ابتدا normal ها مقدار دهی شود و سپس name value ها

```
def func(a,b,c):  1 usage
    print(a)
    print(b)
    print(c)

func( a: 1,b=2,c=3)
```

```
D:\pyth
1
2
3
```

۴- *iterable(list,tuple,list,set,...) : براساس موقعیت آن ها unpack میشود و مقدار دهی میشود

```
def func(a,b,c):  1 usage
    print(a)
    print(b)
    print(c)

func(*[1,2,3])
```

```
D:\pyt
1
2
3
```

۵- **dict : براساس کلیدها مقدار دهی میشود

```
def func(a,b,c):  1 usage
    print(a)
    print(b)
    print(c)

func(**{"a":1,"b":2,"c":3})
```

```
D:\python_sabzlern\1
1
2
3

Process finished wit
```

Parameter syntax

۱- Normal : به صورت عادی و بر اساس جایگاه

۲- Default value : مقدار defult میدهیم اگر در ارگمان‌ها وارد نکردیم آن را برمیدارد

```
def func(a=4, b=5, c=6):  1 usage
    print(a)
    print(b)
    print(c)


func( a: 1,  b: 2)
```

```
Run    test5
D:\python_sabzlern\1\.
1
2
6

Process finished with
```

۳- *name : برای iterable ها کاربرد دارد . اگر ارگمان ها بیشتر از ورودی های تابع بود استفاده میکنیم

```
def func(a=4, b=5, *c):  1 usage
    print(a)
    print(b)
    print(c)


func( a: 1,  b: 2,  *c: 3, 4, 5, 6)
```

```
Run    test5
D:\python_sabzl
1
2
(3, 4, 5, 6)
```

در نسخه های قبلی پایتون نمیتواند بعد از*c پارامتر دیگر نوشت ولی در نسخه ۳ به بعد میتوان با نوشتن نام ارگمان به صورت name value میتوان نوشت

```
def func(a=4, b=5, *c ,d):  1 usage
    print(a)
    print(b)
    print(c)
    print(d)


func( a: 1,  b: 2,  *c: 3, 4, 5, d=6)
```

```
D:\python_sabzlern\1\.
1
2
(3, 4, 5)
6

Process finished with
```

۴- **name : پارامتر به صورت دیکشنری دریافت میشود

```
def func(a=4, b=5, **c ):  1 usage
    print(a)
    print(b)
    print(c)


func( a: 1,  b: 2, d=3, e=4)
```

```
D:\python_sabzlern\1\.ve
1
2
{'d': 3, 'e': 4}

Process finished with ex
```

همچنین میتواند ترکیبی از هر دو باشد :

```
def func(*a, **c ):  1 usage
    print(a)
    print(c)


func( *a: 1, 2, d=3, e=4)
```

```
D:\python_sabzlern\1\
(1, 2)
{'d': 3, 'e': 4}

Process finished with
```

- اگر در پارامترها * قرار دادیم یعنی پارامترهای بعدی * حتما باید به صورت نام مقدار باشند
- اگر در پارمترها / قرار دادیم یعنی پارامترهای قبل / باید به صورت جایگاهی باشند

```
def func(a, b, /, c, d, *, e, f):  1 usage
    print(a, b, c, d, e, f)


func( a: 1,  b: 2,  c: 3,  d: 4, e=5, f=6)
```

```
D:\python_sabzlern
1 2 3 4 5 6

Process finished w
```

- Doc string

- اگر خارج از بدنه تابع و کلاس """ """ قرار دهیم در آن صورت کامنت حساب میشود
- اگر درون متغیر قرار دهیم یک string است
- اگر داخل بدنه تابع یا کلاس باشد یک doc string است
- برای نوشتن اصولی در خط اول یک توضیح برای عملکرد آن نوشته و در ادامه برای هر پرامتر و مقدار return شده یک توضیح نوشته
- حرف اول توضیح باید بزرگ باشد و در آخر آن باید . گذاشته شود

```
def func(a,b):
    """this function a+b

    :param a: (int)
    :param b: (int)
    :return: (int) a+b
    """
    return a+b
```

براتی دیدن doc string باید به صورت زیر عمل کنیم

```
print(func.__doc__)
```

```
this function a+b

:param a: (int)
:param b: (int)
:return: (int) a+b
```

- Function annotation

در پایتون لازم نیست نوع داده ورودی و خروجی را برای تابع مشخص کنیم ولی بهتر است annotation قرار بدهیم

برای این کار برای ورودی تابع از : و برای خروجی تابع از -> استفاده میکنیم

```
def func(a:int , b:int=1)->int :
    return a+b
```

برای دیدن annotation ها میتوانیم از متد زیر استفاده کنیم

```
def func(a:int , b:int=1)->int :
    return a+b
print(func.__annotations__)
```

```
D:\python_sabzlern\1\.venv\Scripts\python.exe D:\python_
{'a': <class 'int'>, 'b': <class 'int'>, 'return': <clas
```

- First class

در پایتون، "first class" به مفهوم "کلاس اول" اشاره دارد که به این معنی است که یک شیء می‌تواند به عنوان یک نوع داده در نظر گرفته شود و می‌تواند به راحتی به عنوان آرگومان به توابع منتقل شود، از توابع بازگشتی استفاده شود و در متغیرها ذخیره شود. این مفهوم معمولاً در مورد توابع و اشیاء در زبان‌های برنامه‌نویسی اشاره دارد.

در پایتون، توابع به عنوان اشیاء کلاس اول (first-class objects) محسوب می‌شوند. این بدان معناست که

1- Can be created at runtime.

```
x = True

if x:
    def func(x):  1 usage
        print(x**2)


func(5)
```

```
Run    test5
D:\python_sabzl
25

Process finishe
```

اگر شرط x غلط باشد در آن صورت با خطا مواجه خواهیم شد که این به معنی Can be created at runtime است

2- Can be assigned to a variable.

```
def func(x):  1 usage
    return x**2


f = func
print(f(2))
```

```
Run    test5
D:\python_sabzlern\
4

Process finished wi
```

در اینجا تابع func به متغیر f پاس داده شده است

3- Can be passed as an argument to a function.

```python
def pow2(x:int):  1 usage
    return x ** 2

def pow4(x:int,func):  1 usage
    return func(x)**2

print(pow4( x: 4,pow2))
```

```
Run    test5
D:\python_sabzlern\1\.venv\Scr
256

Process finished with exit cod
```

در این مثال تابع به عنوان ارگمان تابع گرفته است

4- Can be returned as a result from a function.

```python
def ascending(my_list: list):  1 usage
    print(sorted(my_list))

def descending(my_list: list):  1 usage
    print(sorted(my_list,reverse=True))

def select_sorted(s:str):  1 usage
    if s=="a":
        return ascending
    elif s=="d":
        return descending
    return None


x = select_sorted("a")
x([1,54,83,4,5])
```

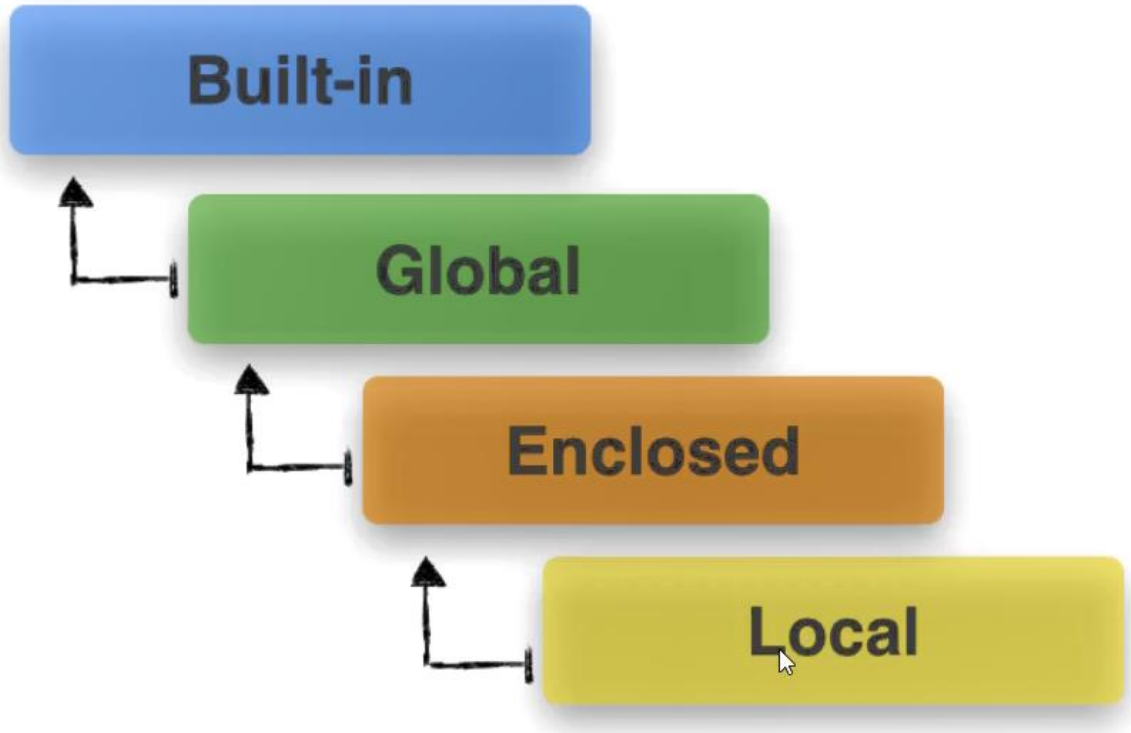
```
Run    test5
D:\python_sabzlern\1\.venv\Scripts
[1, 4, 5, 54, 83]

Process finished with exit code 0
```

در این مثال میبینیم که تابع به عنوان یک نتیجه برگشته شده

5- Can have properties and methods.

میتوانیم متدهای مختلفی داشت که قبلا با چندتای آن ها مانند --name-- , --doc-- آشنا شدیم

- Name space



۱- Built in :

این فضای نام برای همه ماژول‌ها معتبراست مانند len(),sum()

برای دیدن این نام ها میتوانیم از دستور print(dir(__builtins__)) استفاده کنیم

۲- Global :

این فضای نام در سراسر یک ماژول معتبراست به طور مثال x = 10

برای دیدن نام‌های global میتوانیم از print(globals()) استفاده کنیم

```
x = 10
print(globals())
```

```
n_sabzlern\\1\\test5.py', '__cached__': None, 'x': 10}
```

فضای نام ها به صورت دیکشنری ذخیره میشود

۳- Local :

درونی ترین فضای نام یک تابع است

برای دیدن نام‌ها print(locals()) مینویسیم .

دستور print(locals()) با دستور print(globals()) در بیرون تابع یکسان هستند

```
x = 10

def func():  1 usage
    x = 5
    print(x)

func()
```

```
D:\pyt
5

Proces
```

الان اگر x = 5 پاک شود x = 10 چاپ میشود

```
x = 10

def func():  1 usage
    print(x)

func()
```

```
Run    test5
D:\python_sabz
10

Process finish
```

۴- Enclose :

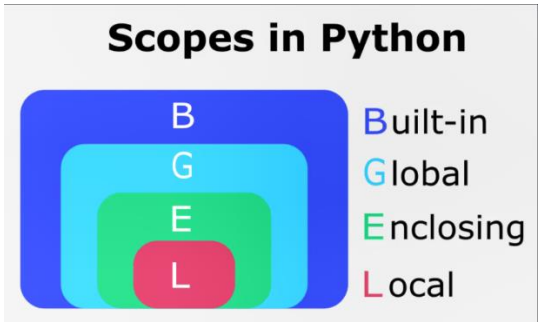
فرض کنیم توابع تو در تو داریم

نام هایی که درون تابع داخلی است local و نام هایی که درون تابع بیرونی است enclose است

```
x = 10
def func():  1 usage
    x = 20
    print(x)

    def func1():
        x = 30
        print(x)

func()
```

- Scope

میخواهیم در بخش قوانین حوزه ها را بررسی کنیم



ما هرچقدر در لایه های پایین تر باشیم به فضای نام لایه های بالاتر دسترسی داریم

هر چقدر اسم نزدیک تر به فضای نامی که هستیم باشد آن اسم چاپ میشود

مثلا در فضای نام local هستیم و میخواهیم x را پرینت کنیم . x هم در global است و هم در enclose . در اخر x در eclose چاپ میشود

```
x = 10
def func():  1 usage
    x = 20
    def func1():
        print(x)
    func1()

func()
```

```
Run
D:\py
20

Proce
```

برای دسترسی به x در global باید دستور زیر را بنویسیم

```
x = 10
def func():  1 usage
    x = 20
    def func1():
        global x
        print(x)
    func1()

func()
```

```
Run
D:\pyth
10

Process
```

اگر بخواهیم به x که در فضای نام enclose است درسترسی پیدا کنیم مینویسیم

```
x = 10
def func():  1 usage
    x = 20
    def func1():
        nonlocal x
        print(x)
    func1()

func()
```

```
Run
D:\py
20

Proce
```

- Pass by reference pass by value

Pass by value یعنی مقدار به تابع پاس داده شود و تغییرات روی کپی از آن صورت گیرد

Pass by reference یعنی ادرس داده میشود و تغییرات روی ادرس صورت میگیرد

پایتون یک زبان reference است وبه تابع ادرس داده میشود ولی چرا در کد زیر تغییرات رخ نمیدهد :

```
def func(x):  1 usage
    x +=1
    print(f"inside function {x}")

a = 1
func(a)
print(f"outside function {a}")
```

```
Run    test5
D:\python_sabzlern\1\.ven
inside function 2
outside function 1
```

دلیل این امر mutable , immutable است

بیایم یک لیست را در نظر بگیریم

```
def func(x):  1 usage
    x += [4, 5]
    print(f"inside function {x}")


a = [1, 2, 3]
func(a)
print(f"outside function {a}")
```

```
Run    test5
D:\python_sabzlern\1\.venv\Scripts\python.e
inside function [1, 2, 3, 4, 5]
outside function [1, 2, 3, 4, 5]

Process finished with exit code 0
```

میبینیم که تغییرات اعمال شده است

برای اینکه این اتفاق نیفتد میتوانیم از copy استفاده کنیم

دقت شود حتما باید به صورت += نوشته شود و اگر به صورت زیر نوشته شود انگار یک متغیر جدید تعریف شده است

```
def func(x):  1 usage
    x = x + [4, 5]
    print(f"inside function {x}")


a = [1, 2, 3]
func(a)
print(f"outside function {a}")
```

```
Run    test5
D:\python_sabzlern\1\.venv\Scripts\
inside function [1, 2, 3, 4, 5]
outside function [1, 2, 3]

Process finished with exit code 0
```

- Lamda

تابع lambda در پایتون یک نوع تابع بی‌نام (نامشخص) است که به شما این امکان را می‌دهد تا توابع کوچک را به صورت سریع و مختصر تعریف کنید. این تابع معمولاً برای تعریف توابع ساده و کوتاه استفاده می‌شود که در یک خط قابل نوشتن هستند.

```
a = lambda x, y: x + y
print(a( x: 1, y: 2))
```

```
D:\python_sabzl
3
```

کاربردها :

۱- map :

تابع map در پایتون یک تابع داخلی است که به شما این امکان را می‌دهد تا یک تابع را به هر عنصر از یک iterable (مانند لیست، تاپل و غیره) اعمال کنید. این تابع به شما کمک می‌کند تا به‌راحتی تغییراتی را روی مجموعه‌ای از داده‌ها اعمال کنید بدون اینکه نیاز به نوشتن حلقه‌های تکرار داشته باشید.

```
li1 = [1, 2, 3, 4, 5]
new = map(lambda x: x * 2, li1)
print(list(new))
```

```
D:\python_sabzlern\
[2, 4, 6, 8, 10]
```

۲- filter :

تابع filter در پایتون یک تابع داخلی است که به شما این امکان را می‌دهد تا عناصری از یک iterable (مانند لیست، تاپل و غیره) را بر اساس یک شرط خاص فیلتر کنید. این تابع فقط عناصری را که شرط مورد نظر را برآورده می‌کنند، نگه می‌دارد و بقیه را حذف می‌کند.

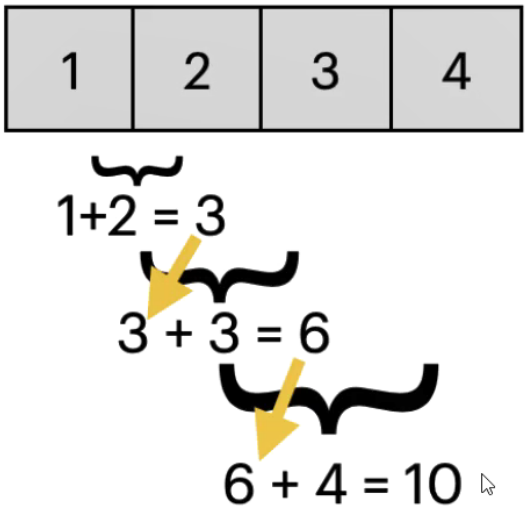
```
li1 = [1, 2, 3, 4, 5]
new = filter(lambda x: x >2 , li1)
print(list(new))
```

```
D:\python_sabz
[3, 4, 5]
```

۳- reduce

تابع reduce در پایتون یکی از توابع مفید است که در ماژول functools قرار دارد. این تابع به شما این امکان را می‌دهد که یک تابع دو آرگومانی را به صورت پیوسته بر روی عناصر یک iterable (مانند لیست یا تاپل) اعمال کنید و در نهایت یک نتیجه واحد به دست آورید.

```
reduce(lambda x, y: x + y, [1, 2, 3, 4])
```



```python
from functools import reduce

li1 = [1,2,3,4,5,6,7,8,9]
new = reduce(lambda x,y:x+y, li1)
print(new)
```

```
D:\python
45

Process
```

۴- sorted

قبلا با این تابع آشنا شدیم

این تابع یک ورودی دارد به نام key که میتوانیم به آن یک تابع پاس بدهیم و در نهایت براساس آن تابه مرتب سازی کنیم

```python
li1 = [1,2,3,4,5,6,7,8,9]
new = sorted(li1,key=lambda x:x%3)
print(new)
```

```
D:\python_sabzlern\1\.venv\Scr
[3, 6, 9, 1, 4, 7, 2, 5, 8]

Process finished with exit coc
```

- iterate

در برنامه‌نویسی و علوم کامپیوتر، مفاهیم "iterate"، "iteration"، "iterable" و "iterator" به طور گسترده‌ای مورد استفاده قرار می‌گیرند. در ادامه به توضیح هر یک از این مفاهیم می‌پردازم:

۱. Iterate (تکرار کردن):

• این واژه به معنای فرآیند تکرار یک مجموعه از عملیات یا مراحل است. به عبارت دیگر، وقتی می‌گوییم که یک الگوریتم یا تابع در حال "iterate" است، به این معناست که آن الگوریتم یا تابع به طور مکرر بر روی داده‌ها یا مجموعه‌ای از عناصر عمل می‌کند.

۲. Iteration (تکرار):

• "Iteration" به یک چرخه یا مرحله از تکرار اشاره دارد. در برنامه‌نویسی، هر بار که یک حلقه (loop) اجرا می‌شود و بر روی یک عنصر از مجموعه داده‌ها عمل می‌کند، یک "iteration" انجام شده است. به عنوان مثال، در یک حلقه for، هر بار که حلقه یک بار اجرا می‌شود، یک iteration جدید رخ می‌دهد.

۳. Iterable (قابل تکرار):

• یک "iterable" به هر شیءای گفته می‌شود که می‌توان آن را تکرار کرد. به عبارت دیگر، اگر یک شیء قابلیت تولید یک توالی از عناصر را داشته باشد که بتوان بر روی آن‌ها با استفاده از یک iterator حرکت کرد، آن شیء قابل

تکرار است. در زبان‌های برنامه‌نویسی مانند Python، لیست‌ها، دیکشنری‌ها و مجموعه‌ها (sets) از جمله iterableها هستند.

۴. Iterator (تکرارگر):

- "Iterator" یک شیء است که به شما اجازه می‌دهد تا بر روی عناصر یک iterable حرکت کنید. این شیء دارای دو متد اصلی است:
  - __iter__() که خود iterator را برمی‌گرداند.
  - __next__() که عنصر بعدی را از iterable برمی‌گرداند. اگر هیچ عنصر دیگری وجود نداشته باشد، این متد باید یک استثنا (exception) به نام StopIteration را پرتاب کند.

به طور خلاصه، مفهوم iterate به فرآیند تکرار اشاره دارد، iteration به هر بار تکرار اشاره دارد، iterable به اشیایی اشاره دارد که می‌توان آن‌ها را تکرار کرد و iterator ابزاری است برای دسترسی به عناصر یک iterable.

برای دیدن اینکه ایا iterator هست سا خیر میتوانیم از کد زر استفاده کنیم :

```
list1 = [1,2,3,4,5]
print("__iter__" in dir(list1))
print("__next__" in dir(list1))
```

```
D:\python_sabzlern\1\
True
False
```

همانطور که میتوانیم ببینیم که list، iterator نیست

برای اینکه تبدیل شود از iter() استفاده میکنیم

```
list1 = [1,2,3,4,5]
list1 = iter(list1)
print("__iter__" in dir(list1))
print("__next__" in dir(list1))
```

```
D:\python_sabzle
True
True
```

حال با زدن متد next میتوانیم روی آن راه برویم

```
list1 = [1,2,3,4,5]
list1 = iter(list1)
print(list1.__next__())
print(list1.__next__())
print(list1.__next__())
print(list1.__next__())
```

```
D:\pytho
1
2
3
4
```

- decorator

در پایتون، دکوراتورها (Decorators) ابزاری هستند که به شما این امکان را می‌دهند تا رفتار یک تابع یا متد را بدون تغییر کد اصلی آن، تغییر دهید. دکوراتورها معمولاً برای اضافه کردن ویژگی‌ها یا قابلیت‌های جدید به توابع یا متدها استفاده می‌شوند.

دکوراتور به صورت یک تابع تعریف می‌شود که یک تابع دیگر را به عنوان آرگومان می‌گیرد و معمولاً یک تابع جدید را برمی‌گرداند. به این ترتیب، شما می‌توانید از دکوراتور برای انجام کارهایی مانند لاگ‌کردن، مجوزدهی، زمان‌سنجی و غیره استفاده کنید.

برای مثال کد زیر را بررسی کنیم

```
def my_decorator(func):  1 usage
    def inner():
        print("*" * 20)
        func()
        print("*" * 20)

    return inner


@my_decorator  1 usage
def my_function():
    print("Hossein Ghorbani")


my_function()
```

```
Run    test5
D:\python_sabzlern\1\.v
********************
Hossein Ghorbani
********************

Process finished with e
```

در اینجا تابع my_function اصلاح شده است و قبل و بعد از آن ستاره چاپ شده است

حال اگر ورودی داشتیم به چه صورت بود . برای مثال میخواهیم تقسیم بر صفر را مدیریت کنیم

```
def my_decorator(func):  1 usage
    def inner(a:int,b:int):
        if b ==0 :
            return "not valid"
        return func(a,b)

    return inner


@my_decorator  1 usage
def my_function(a: int, b: int):
    return a / b


print(my_function( a: 2, b: 0))
```

```
Run    test5
D:\python_sabzl
not valid

Process finishe
```

- برای اینکه برای بعضی توابع تعداد ورودی و خروجی ها مشخص نیست و باید دکوراتور برای اکثر توابع جواب باشد میتوانیم از * و ** استفاده کنیم

```
def my_decorator(func):  1 usage
    def wrapper(*args, **kwargs):
        if args[1] ==0 :
            return "not valid"
        return func(*args, **kwargs)

    return wrapper


@my_decorator  1 usage
def my_function(a: int, b: int):
    return a / b


print(my_function( a: 2, b: 0))
```

```
Run    test5
D:\python_sabz
not valid

Process finish
```

- اعمال decorator روی decorator

```python
def plus(func):  1 usage
    def wrapper(*args, **kwargs):
        print("+"*20)
        func(*args, **kwargs)
        print("+"*20)
    return wrapper

def star(func):  1 usage
    def wrapper(*args, **kwargs):
        print("*"*20)
        func(*args, **kwargs)
        print("*"*20)
    return wrapper
@plus  1 usage
@star
def print_name():
    print("Hossein Ghorbani")

print_name()
```

```
Run    test5
D:\python_sabzlern\1\.ve
++++++++++++++++++++
********************
Hossein Ghorbani
********************
++++++++++++++++++++

Process finished with ex
```

ترتیب اجرا از پایین به بالا

- برای اینکه doc string هایی که روی تابع اصلی است تغییر نکند باید از کتابخانه functools استفاده کنیم

```python
from functools import wraps

def star(func):  1 usage
    @wraps(func)
    def wrapper(*args, **kwargs):
        print("*"*20)
        func(*args, **kwargs)
        print("*"*20)
    return wrapper
@star  1 usage
def print_name():
    """ print name"""
    print("Hossein Ghorbani")

print(print_name.__doc__)
```

```
Run    test5
D:\python_sabzlern\1\.venv\
print name

Process finished with exit
```

- Generator

جنراتور (Generator) در زبان‌های برنامه‌نویسی پایتون، ابزاری است که به شما امکان می‌دهد تا یک دنباله از مقادیر را به صورت تدریجی و در زمان نیاز تولید کنید. به عبارت دیگر، جنراتورها به شما این امکان را می‌دهند که به جای تولید تمام مقادیر در یک بار، آن‌ها را یکی یکی و به صورت "تنبل" (lazy) تولید کنید. این ویژگی می‌تواند در صرفه‌جویی در حافظه و زمان بسیار مفید باشد.

در هر بار که به yield برسیم برنامه متظر میماند تا متد next را صدا بزنیم

```
def func(x):  1 usage
    yield x
    print("HAHA")
    yield x+5
    print("niceeee!")
    yield x+10

f = func(1)
print(next(f))
print(next(f))
print(next(f))
```

```
Run    test5
HAHA
6
niceeee!
11

Process fi
```

میخواهیم دنباله ای از اعداد به توان دو بنویسیم

```
def my_generator():  1 usage
    for i in range(10):
        yield i**2


for i in my_generator():
    print(i)
```

```
Run    test5
D:\python_sab
0
1
4
9
16
25
36
```

اگر از عبارت هایی مانند list,max,min,... استفاده کنیم و متد next را صدا کنیم آنگاه خطا میخوریم دلیل آن این است با آن توابع یکبار طول آن را پیمایش کردیم

```python
def my_generator():  1 usage
    for i in range(10):
        yield i**2

g= my_generator()
print(max(g))
next(g)
```

```
Run    test5
D:\python_sabz
81
Traceback (mos
  File "D:\pyt
    next(g)
```

دو متد میخواهمیم معرفی کنیم

۱- close : اگر بزنیم در آن صورت generator بسته خواهد شد
۲- throw : برای تولید خطا و استثنا

- Coroutine

برای ارسال مقدار به generator میتوانیم از متد send() استفاده کنیم

```python
def my_generator():  1 usage
    while True:
        name = yield
        print(name)

g= my_generator()
next(g)
g.send("ali")
g.send("Hossein")
```

```
Run    test5
D:\python_
ali
Hossein

Process fi
```

برای بار اول حتما باید next() زده شود

میتوانیم یک decorator بنویسیم و دیگر لازم نباشد next را صدا بزنیم

```python
def coroutine(func):  1 usage
    def wrapper(*args, **kwargs):
        g = func(*args, **kwargs)
        next(g)
        return g
    return wrapper
@coroutine  1 usage
def my_generator():
    while True:
        name = yield
        print(name)

g= my_generator()
g.send("ali")
g.send("Hossein")
```

```
Run    test5
D:\python_
ali
Hossein

Process fi
```

- Function attribute

برای دیدن attribute های تابع میتوانیم از دستور dir استفاده کنیم

پیش از این با دوتا از attribute ها آشنا شدیم که در زیر باز تکرار شده است

```python
def func():  2 usages
    """ learn about __doc__ & __name__ """
    pass
print(func.__doc__)
print(func.__name__)
```

```
Run    test5
D:\python_sabzlern\1\.venv\Script
learn about __doc__ & __name__
func
```

- در ادامه میخواهیم خودمان یک سری attribute قرار دهیم

برای این کار راه اول :

Function name.attribute name = value

برای attribute ها میتوانیم از دد دستور زیر استفاده کنیم

```
def func():  4 usages
    """ learn about __doc__ & __name__
    pass

func.new_attribute = "value"
print(func.new_attribute)
print(dir(func))
print(func.__dict__)
```

```
Run   test5
D:\python_sabzlern\1\.venv\Scripts\p
value
['__annotations__', '__builtins__',
{'new_attribute': 'value'}
```

راه دوم استفاده از setattr و برای دیدن آن getattr

```
def func():  2 usages
    """ learn about __doc__ & __name__ """
    pass

setattr(func, 'new_attr', 'new_value')
print(getattr(func, 'new_attr',None))
```

```
Run   test5
D:\python_sabzlern\1
new_value
```

میتوانیم برای getattr یک مقدار default قرار بدهیم

hasattr چک میکند که آیا چنین attribute وجود دارد یا خیر

برای پاک کردن یک attribute دو راه زیر وجود دارد

```
def func():  4 usages
    """ learn about __doc__ & __name__ ""
    pass

setattr(func, 'new_attr', 'new_value')
del func.new_attr
delattr(func, 'new_attr')
print(getattr(func, 'new_attr'))
```

- Recursive function

الگوی توابع بازگشتی به صورت زیر است:

```
def recursive()
    ...
    if stop_condition:
        ...
        return ...
    ...
    recursive()

recursive()
```

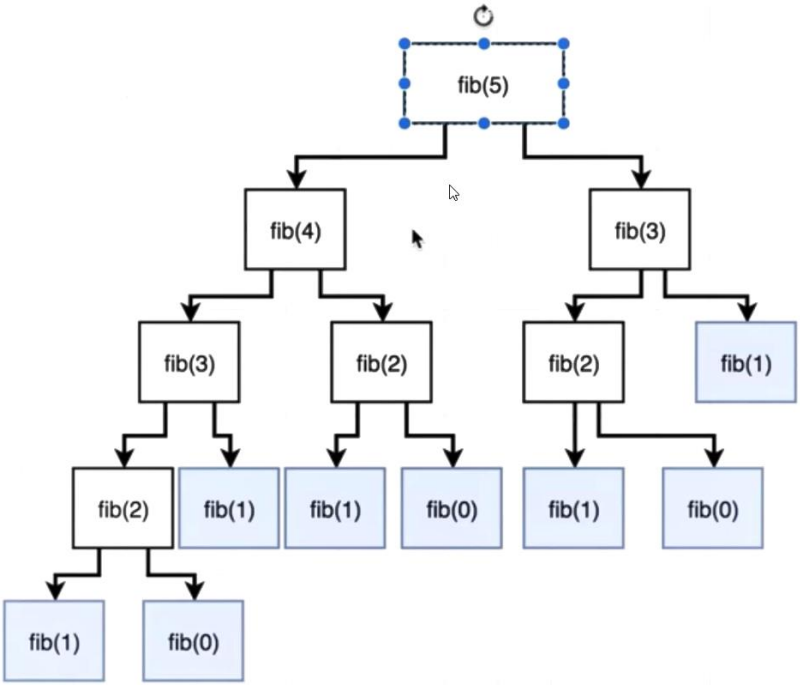
برای مثال برای به دست اورن Factorial میتوانیم از کد زیر استفاده کنیم

```
def factorial(n):  2 usages
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)

print(factorial(5))
```

```
Run    test5
D:\python_sabz
120

Process finish
```

- یک تکنیک برای افزایش سرعت برای محاسبه دنباله های بازگشتی این است که اگر قبلا محاسبه شده بود دوباره محاسبه نشود. نام این تکنیک memorization است

در شکل زیر میبینیم برای محاسبه fibo(5) چندبار fibo(2) حساب شده است



کد مربوط به آن در زیر آمده است :

```
from functools import wraps
def dec_factorail(func):  1 usage
    memery = {}
    @wraps(func)
    def wrapper(*args, **kwargs):
        if args[0] not in memery.keys():
            memery[args[0]] = func(args[0])
        return memery[args[0]]
    return wrapper

@dec_factorail  2 usages
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)

print(factorial(200))
```

فصل هفتم: انواع داده (سطح دو)

- Number method

در ادامه معرفی متدهای توابع

۱- divmode() : به طور خلاصه جواب تقسیم روبه‌رو را میدهد : (x//y,x%y)

```
print(divmod(10,3))
```

```
(3, 1)
```

۲- pow() : عدد اول را به توان عدد دوم میرساند. اگر سه ورودی داشته باشیم پس از به توان رساندن باقی مانده آن را نیز حساب میکند

```
print(pow(2, 4, 3))
```

```
D:\python
1
```

۳- as_integer_ratio() : نشان میدهد یک عدد اعشار به صورت کسری چگونه است

```
x = 3.5
print(x.as_integer_ratio())
```

```
D:\python_sabzlern\1\.venv
(7, 2)
```

۴- bit_count : برای شمارش تعداد ۱ ها در حالت باینری

۵- Bit_lengh : برای شمارش طول بیت

```
x = 5 #101
print(x.bit_length())
print(x.bit_count())
```

```
D:\pyth
3
2
```

۶- to_bytes : برای نمایش اعداد به صورت هگز که ورودی اول خود عدد، ورودی دوم میگوید پر ازرش سمت راست باشد یا سمت چپ ، ورودی سوم میگوید عدد علامت باشد یا خیر

```
x = 16
print(x.to_bytes(x, byteorder='big',signed=False))
```

```
n   test6 ×
venv\Scripts\python.exe D:\python_sabzlern\1\test6.p
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x10'
```

۷- from_bytes : برعکس کار بالا را انجام میدهد

۸- hex : برای اعداد اعشاری است و تبدیل به hex میکند

۹- from_hex : عکس بالا را انجام میدهد

۱۰- is_integer : چک میکند آیا عدد صحیح است یا خیر

```
x = 16.000
print(x.is_integer())
```

```
D:\python_sabzle
True
```

- list method

۱- append : برای اضافه کردن یک عنصر به آخر لیست

```
x = [1, 2, 3, 4, 5]
x.append(6)
print(x)
```

```
D:\python_sabzlern\1\.ven
[1, 2, 3, 4, 5, 6]
```

۲- clear : عناصر داخل لیست را پاک میکند

۳- count : تعداد عناصر لیست را میدهد

```
x = [1, 2, 3, 4, 1]
print(x.count(1))
```

```
D:\pyth
2
```

۴- extend : یک دنباله را میتوانیم به لیست خود اضافه کنیم

```
x = [1, 2, 3, 4,]
x.extend((5,6,7))
print(x)
```

```
D:\python_sabzlern\1\.v
[1, 2, 3, 4, 5, 6, 7]
```

۵- index : عضوی را میدهیم و میگوید index چند است . میتوانیم برای آن محدوده معلوم کنیم برای مثال از جای index ۲ تا index ۱۰

```
x = [1, 2, 3, 4,]
print(x.index( __value: 3, __start: 1, __stop: 5))
```

```
D:\python_sabzlern
2
```

۶- insert : برای اضافه کردن یک عنصر در جایگاه خاص

```
x = [1, 2, 3, 4,]
x.insert( __index: 5, __object: 2)
print(x)
```

```
D:\python_sabzlern\1\.v
[1, 2, 3, 4, 2]
```

۷- pop : برای پاک کردن یک عنصر و ورودی جایگاه آن را میگیرد . همچنین عنصر پاک شده را برمیگرداند

```
x = [1, 2, 3, 4,]
print(x.pop(0))
print(x)
```

```
D:\python_sabzlern\
1
[2, 3, 4]
```

۸- remove : مقدار عنصر مورد نظر را براساس مقدار پاک میکند . اگر دوتا بود اولین از سمت راست پاک میکند

```
x = [1, 2, 3, 4,]
x.remove(2)
print(x)
```

```
D:\python_s
[1, 3, 4]
```

۹- reverse : لیست را برعکس میکند

۱۰-sort :لیست را مرتب میکند . همچنین قبلا با کار کرد reverse,lambda آشنا شدیم

```
x = [1, 2, 3, 4,]
x.sort(reverse = True,key = lambda x:x%3)
print(x)
```

```
D:\python_sabz
[2, 1, 4, 3]
```

- Tuple and set method

برای تاپل‌ها دوتا متد بیشتر نداریم

۱- index : جایگاه مقدار مورد نظر را نمایش میدهد
۲- count : تعداد عناصر در تاپل نمایش میدهد

برای set ها قبلا با اکثر آن‌ها آشنا شدیم . همانطور گفته شد setها باید اعضای آن غیر قابل تغییر باشند

۱- add : یک عنصر به مجموعه ما اضافه میکند
۲- copy : مانند کپی در لیست‌ها است
۳- pop :یک عنصر را به صورت شانسی پاک میکند
۴- remove : برای پاک کردن یک عنصر خاص
۵- قبلا با دستورات اجتماع و اشتراک و... آشنا شدیم . حال اگر در آخر آن update باشد در آن صورت روی مجموعه اصلی هم اعمال میشود
۶- isdisjoint : اگر اشتراکی نباشد True برگردانده میشود

```
s1 = {1,2,3}
s2 = {4,5,6}
print(s2.isdisjoint(s1))
```

```
D:\python_sabzle
True
```

- dictionary method

۱- clear : برای پاک کردن دیکشنری
۲- fromkeys : برای ساخت دیکشنری که فقط کلیدها را میگیرد و میتوان مقدار defaut برای آن معلوم کرد.اگر مقدار default نداشته باشد در آن صورت None درنظرمیگیرد

```
print(dict.fromkeys( __iterable: [1,2,3,4], __value: 1))
```

```
{1: 1, 2: 1, 3: 1, 4: 1}
```

۳- get : با دادن کلید میتوانیم value را بگیریم ۰ اگر نباشد none میدهد و همچنین میتوانیم مقدار default بدهیم
۴- pop : برخلاف pop های دیگر باید به آن مقدار کلید را بدهیم
۵- popitem : معمولا از آخر حذف میکند و نیاز به مقدار کلید نیست
۶- setdefault : یک کلید به آن میدهیم . اگر وجود داشت مقدار آن را برمیگرداند اگر وجود نداشت خودش به دیکشنری اضافه میکند با مقدار none. مقدار default هم میتوانیم برای آن قرار دهیم

```python
d = {"a": 1, "b": 2, "c": 3}
print(d.setdefault("c", 4))
print(d)
print(d.setdefault("f", 4))
print(d)
```

```
D:\python_sabzlern\1\.venv\Scripts\py
3
{'a': 1, 'b': 2, 'c': 3}
4
{'a': 1, 'b': 2, 'c': 3, 'f': 4}

Process finished with exit code 0
```

۷- update : میتوانیم یک dictionary را ابدیت کنیم به سه روش

اول : اینکه یک دیکشنری به ورودی بدهیم اگر کلیدها تکراری بودند در آن صورت مقدار جدید قرار میگیرد

دوم : با استفاده از جفت tuple

سوم:با استفاده از مقدار = کلید

```python
d = {"a": 1, "b": 2, "c": 3}
d.update({"d": 4, "e": 5, "f": 6})
d.update([("g", 7), ("h", 8)])
d.update(i=9,j=10)
print(d)
```

```
Run  test6
Scripts\python.exe D:\python_sabzlern\1\test6.py
'f': 6, 'g': 7, 'h': 8, 'i': 9, 'j': 10}

code 0
```

- برای اینکه نام یک کلید را عوض کنیم از تکنیک زیر استفاده میکنیم

```python
d = {"a": 1, "b": 2, "c": 3}
d["e"]=d.pop("c")
print(d)
```

```
D:\python_sabzlern\1\.venv\
{'a': 1, 'b': 2, 'e': 3}
```

- Walrus

همانطور قبلا از این هم خواندیم برای اینکه همزمان داخل بدنه یک تابع مقدار دهی شود و اجرا شود از این عملگر استفاده میکنیم :=

حال مثالی در نظر بگیریم که میخواهیم وقتی کاربر q را وارد کرد دیگر برنامه تمام شود

```python
li = []
while (x := input("enter q to close program: ")) != "q":
    li.append(x)

print(li)
```

```
D:\python_sabzlern\1\.venv\Scripts
enter q to close program: ali
enter q to close program: reza
enter q to close program: q
['ali', 'reza']
```

به پرانتزها باید دقت شود

مثالی دیگر برای if

```
l1 = [1, 2, 3, 4]
if (x := len(l1)) > 3:
    print(x)
```

```
D:\python_sabzlern
4
```

مثال اخر استفاده آن در دیکشنری

```
l1 = [1, 2, 3, 4]
dic1 = {
    "len" : (l:=len(l1))
    ,"sum": (s:=sum(l1))
    ,"avg" : (s/l)
    ,
}
print(dic1)
```

```
Run    test6
D:\python_sabzlern\1\.venv\Scripts\p
{'len': 4, 'sum': 10, 'avg': 2.5}

Process finished with exit code 0
```

- Comprehension

برای خلاصه سازی استفاده میشود به مثال زیر دقت شود :

```
l1 = [1, 2, 3, 4]
l2 = [i**2 for i in l1]
print(l2)
```

```
D:\python_sabzler
[1, 4, 9, 16]
```

به طور خلاصه به صورت زیر نوشته میشود :

```
l = []
for sub1 in iterable:
    if cond1(sub1):
        if cond2(sub1):
            for sub2 in sub1:
                if cond3(sub2):
                    for elem in sub2:
                        l.append(func(elem))
```

```
[func(elem) for sub1 in iterable if cond1(sub1) if cond2(sub1) for sub2 in sub1 if cond3(sub2) for elem in sub2]
```

- برای درک بهتر مثال دیگر همراه با if مینویسیم

```
l1 = [1, 2, 3, 4]
l2 = [j for i in l1 if (j:=i**2)>5]
print(l2)
```

```
D:\python_s
[9, 16]
```

- مثالی دیگر میخواهیم حروف کلماتی که در لیست هستند را نمایش دهیم

```
names = ["ali","reza","sahel"]
x = [char.upper() for name in names for char in name]
print(x)
```

```
D:\python_sabzlern\1\.venv\Scripts\python.exe D:\pyt
['A', 'L', 'I', 'R', 'E', 'Z', 'A', 'S', 'A', 'H', '
```

- مثالی برای پیدا کردن معکوس لیست :

```
matrix = [
    [1,2,3,4]
    ,[5,6,7,8]
    ,[9,10,11,12]
    ,
]
l = [[i[j] for i in matrix] for j in range(len(matrix))]
print(l)
```

```
Run    test6
D:\python_sabzlern\1\.venv\Scripts\
[[1, 5, 9], [2, 6, 10], [3, 7, 11]]

Process finished with exit code 0
```

راه جالب دیگر استفاده از zip است

```
matrix = [
    [1,2,3,4]
    ,[5,6,7,8]
    ,[9,10,11,12]
    ,
]
x = zip(*matrix)
print(list(x))
```

```
Run    test6
D:\python_sabzlern\1\.venv\Scripts\python.exe D:\p
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]

Process finished with exit code 0
```

- اگر به جای [ ] بیایم و () قرار دهیم در آن صورت تاپل ساخته نمیشود بلکه یک generator ساخته میشود

```
x = (i for i in range(10))
print(type(x))
```

```
D:\python_sabzlern\1\.venv\Scrip
<class 'generator'>
```

- برای set ها هم میتوانیم استفاده کنیم
- برای دیکشنری ها هم میتوانیم استفاده کنیم . برای مثال میخواهیم دو لیست که در یکی اسم ها و در دیگری سن قرار دارد دیکشنری بسازیم

```
names = ["neda", "reza", "ali"]
ages = [12, 15, 22]
dict1 = {name:age for name, age in zip(names, ages)}
print(dict1)
```

```
D:\python_sabzlern\1\.venv\Scripts\python.ex
{'neda': 12, 'reza': 15, 'ali': 22}

Process finished with exit code 0
```

- اگر else داشتیم به چه صورتی بنویسیم

```
numbers = [1,2,3,4,5,6,7,8,9]
x = [i if i %2 == 0 else 0 for i in numbers]
print(x)
```

```
D:\python_sabzlern\1\.venv\Scripts
[0, 2, 0, 4, 0, 6, 0, 8, 0]
```

- میتوانیم از توابع نیز استفاده کنیم

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]


def func1(number: int):  1 usage
    if number % 2 == 0:
        return number
    else:
        return 0


x = [func1(i) for i in numbers]
print(x)
```

```
Run    test6
D:\python_sabzlern\1\.venv\Scrip
[0, 2, 0, 4, 0, 6, 0, 8, 0]

Process finished with exit code
```

- میخواهیم ده عدد رندوم بینه ۵۰و۱۰۰ قرار دهیم . باید از walrus استفاده کنیم

```
from random import randrange

x = [j for _ in range(10) if (j:=randrange(200))<100]
print(x)
```

```
D:\python_sabzlern\1\.ven
[24, 4, 5, 86]
```

- Decoding and Encoding

Asci ها قبلا با آشنا شدیم شامل ۷ بیت هستند که بسیار محدود هستند

رایج ترین utf8 است که همه زبان ها را پشتیبانی میکند

برای دیدن شماره میتوانیم از دستور ord استفاده کنیم

```
print(ord('㋀'))
```

```
12800
```

برای دیدن کارکتر میتوانیم از دستور chr() استفاده کنیم

```
print(chr(12800))
```

```
㋀
```

علاوه براین میتوانیم به صورت زیر با استفاده از Unicode چاپ کنیم . دقت شود اگر U بزرگ بود پیش از چاپ 0000 میخواهد

```
print("\U000031F0")
print("\u31F0")
```

```
D:\python_sa
ㄱ
ㄱ
```

میتوانیم برای چاپ از هگز و اکتاو و باینری استفاده کنیم

- Bytes & bytearray

علاوه بر نوع داده str برای رشته نوع داده byte و bytes و bytearray هم داریم که در ادامه معرفی میکنیم

- Byte : این نوع داده فقط از اسکی ها پشتیبانی میکند

```
x = b'Hossein'
print(x)
print(type(x))
```

```
D:\python_sabzlern\1\.
b'Hossein'
<class 'bytes'>
```

- Bytes : این نوع داده میتواند از Unicode ها هم پشتیبانی کند . زمانی که پرینت را میزنیم عبارت هایی که در اسکی نیستند را به صورت هگز نمایش میدهد.

```
x = bytes("◎Hossein◎", "utf-8")
print(x)
print(type(x))
```

```
D:\python_sabzlern\1\.venv\Scripts\python
b'\xe2\x97\x8eHossein\xe2\x97\x8e'
<class 'bytes'>
```

برای اینکه در پرینت به درستی نمایش دهد میتوانیم از دستور زیر استفاده کنیم . به نوع داده دقت کنیم

```
x = bytes("◎Hossein◎", "utf-8")
print(x.decode("utf-8"))
print(type(x.decode("utf-8")))
```

```
D:\python_sabzl
◎Hossein◎
<class 'str'>
```

چگونه متوجه میشود تا چند هگز مربوط به یک کارکتر است ؟ از روی تعداد یک‌های اولیه. در مثال زیر سه هگز برای یک کارکتر است

U+25CE	◎	11100010 10010111 10001110	BULLSEYE

یک سری ورودی دارد که میگوید اگر تبدیل نشد چیکار کن :

```
x = bytes("◎Hossein◎", "ascii","replace")
x = bytes("◎Hossein◎", "ascii","ignore")
x = bytes("◎Hossein◎", "ascii","namereplace")
```

- Bytearray

مانند Bytes است با این فرق که قابل تغییر است

```
x = bytearray("◎Hossein◎", "utf-8")
x[0] = 0b11100010
x[1] = 0b10010110
x[2] = 0b10100000
print(x.decode("utf-8"))
print(type(x.decode("utf-8")))
```

```
Run  testo
D:\python_sabzlern
■Hossein◎
<class 'str'>
```

- حال میفهمی در 32 چرا تبدیل به space شده است :

```
print((32).to_bytes( length: 4, byteorder='big'))
```

```
D:\python_sabzler
b'\x00\x00\x00 '
```

- String method

۱- casefold : همه کارکترها را به حروف کوچک تبدیل میکند . نسبت به capitalize قوی تر است و روی همه زبان ها جواب میدهد

۲- center : رشته را درمرکز قرار میدهد . دیفالت آن space است

```
s = "Hossein"
print(s.center( __width: 10, __fillchar: "*"))
```

```
D:\python_sabzlern
*Hossein**
```

۳- Index , find : هر دو جایگاه اولین رشته را درون رشته مورد نظر پیدا میکنند . فقط find اگر پیدا نکند ۱- میدهد ولی index خطا

۴- rindex : از سمت راست جست و جو میکند

```
s = "Hossein"
print(s.find("f"))
print(s.index("f"))
```

```
D:\python_sabzlern
-1
Traceback (most re
```

۵- format_map : مانند format است با این فرق دیکشنری میگیرد

```
s = {"name":"Hossein","last_name":"Ghorbani"}
print("my name is {name},{last_name}".format_map(s))
```

```
D:\python_sabzlern\1\.venv\Scr
my name is Hossein,Ghorbani
```

۶- isnumeric,isdigitis,decimal:

By definition, `isdecimal()` ⊆ `isdigit()` ⊆ `isnumeric()`. That is, if a string is `decimal`, then it'll also be `digit` and `numeric`.

Therefore, given a string `s` and test it with those three methods, there'll only be 4 types of results.

isdecimal()	isdigit()	isnumeric()	Example
True	True	True	"038", "০৩৮", "０３８"
False	True	True	"⁰³⁸", "₀₃₈", "⓪③⑧"
False	False	True	"⅔⅛⅝", "ⅢⅢⅧ", "⑩⑬㊿", "壹貳參"
False	False	False	"abc", "38.0", "-38"

۷- isascii : چک میکند ایا همه اسکی هستند
۸- isalpha : چک میکند آیا همه حروف هستند
۹- isalnum : ایا همه عدد و حروف هستند
۱۰- isidentifier : آیا میتوان به عنوان نام متغیر استفاده کرد
۱۱- islower : ایا همه حروف کوچک هستند
۱۲- isupper : برعکس ۱۰
۱۳- isprintable : قابل چاپ هست یا نه برای مثال \n نمیتوان دید
۱۴- isspace : ایا همه رشته با \t, \n , space پر شده است
۱۵- istitle : آیا شروع کلمات با حرف بزرگ است
۱۶- expandtabs : به جای tab ها در رشته space قرار میدهد
۱۷- ljust, rjust : مانند center است فقط از سمت چپ یا راست منظم میکند

```
s = "Hossein"
print(s.rjust( _width: 20, _fillchar: "-"))
print(s.ljust( _width: 20, _fillchar: "-"))
```

```
D:\python_sabzlern\1\.ven
-------------Hossein
Hossein-------------
```

۱۸- title : حروف اول کلمات را بزرگ میکند
۱۹- swapcase : حروف بزرگ را کوچک و حروف کوچک را بزرگ میکند
۲۰- lstrip, removeprefix, removesuffix, rstrip : removeperfix پیشوندها را حذف میکند و removesufix پسوندها را حذف میکند برای درک بهتر به مثال زیر دقت شود. برای strip ها هرچی با آن کارکتر شروع شود پاک میکند

```
s = "--Hossein--"

print(s.removeprefix("-"))
```

```
D:\python_sab
-Hossein--
```

```
1   s = "--Hossein--"
2
3   print(s.lstrip("-"))
```

```
D:\python_sa
Hossein--
```

در strip ترتیب مهم نیست

```
s = "xyzxyzyyyzzzxxx+"

print(s.lstrip("xzy"))
```

```
D:\pytho
+

Process
```

۲۱-rsplit, split, splitlines : قبلا تا حدودی آشنا شدیم

فرق rstrip,strip زمانی معلوم میشود که ارگمان دوم را دهیم که تا چند بار split کند

```
s = "ali,hossein,reza"

print(s.rsplit( sep: ",", maxsplit: 1))
print(s.split( sep: ",", maxsplit: 1))
```

```
D:\python_sabzlern\1\.venv\Scr
['ali,hossein', 'reza']
['ali', 'hossein,reza']
```

Splitlines بر اساس \n جدا میکند

```
s = "ali\n,hossein\n,reza"

print(s.splitlines())
```

```
D:\python_sabzlern\1\.venv\Scri
['ali', ',hossein', ',reza']
```

فصل هشتم: ماژول ها و بسته ها

- script : فایل py. است که با اجرای آن یک خروجی میدهد و خارج از کلاس‌ها و توابع دستوری است که خروجی را برای ما تولید میکند
- Module : معمولا با واسطه استفاده میشود برای اجرا باید در یک فابل py. دیگر import شود
- Package : به مجموعه ای از ماژول‌ها میگوییم
- Library : کتابخانه ها گسترده تر از پکیج ها هستند

- Modul

ماژول ها سه نوع هستند :

۱- به صورت خودکار در پایتون نصب شده اند مانند random

```
import random
print(random.randint( a: 1,  b: 100))
```

۲- ماژول هایی که باید نصب شوند مانند numpy

برای نصب این بکیج ها میتوانیم مستقیم از تنظیمات نصب کنیم و یا به CMD رفته ادرس پروژه را بدهیم activate را بزنیم تا (venv) وارد شویم و سپس پکیج را نصب کنیم با دستور pip install [name package]

۳- ماژول هایی که خودمان میسازیم

برای این کار یک فایل py. میسازیم برای مثال نام آن را my_modul قرار میدهیم و درون آن یک تابع مینویسیم به نام my_func

```
Project
1 D:\python_sabzlern\1
.venv library root
my_modul.py
test5.py
```

```
test5.py   test6.py   my_modul.py
def my_func(x:int):
    return x**2
```

برای استفاده کردن میتوانیم ابتدا نام ماژول را صدا کنیم وبه صورت دلخواه میتوانیم یک نام کوتاه شده به آن دهیم و بعد به صورت زیر استفاده کنیم

```
import my_modul as my
print(my.my_func(5))
```

```
D:\python
25
```

راه دیگر این است همزمان تابع هم صدا کنیم به طور مثال

```
from my_modul import my_func as my
print(my(5))
```

```
D:\py
25
```

- برای دیدن محل نصب پکیج میتوانیم به صورت زیر عمل کنیم

```
import random
print(random)
```

```
D:\python_sabzlern\1\.venv\Scripts\python.exe D:\python_
<module 'random' from 'C:\\Users\\Admin\\AppData\\Local\
```

اگر به صورت from بنویسیم خود ماژول اضافه نمیشود و نمیتوان print کرد

- میتوانیم درون ماژول ها تابع،متغیر و کلاس نوشت و از آن استفاده کرد برای مثال متغیر pi

```
def my_func(x:int):
    return x**2
pi = 3.14
```

```
import my_modul
print(my_modul.pi)
```

```
D:\python_sa
3.14
```

اگر _ قبل از نام منغیر قرار دهیم یعنی این متغیر private است و تا موقعی به صورت صریح import نشود نمیتوانیم آن را ببینیم

```
from my_modul import _age
print(_age)
```

```
D:\pytho
24
```

- اگر همه توابع و متغیرها را بخواهیم از * استفاده میکنیم
- نکته بسیار مهم این است که ماژول ها یکبار import میشود و اگر در خط های بعدی تکرار کنیم بی فایده است

```
from my_modul import *
from my_modul import *
from my_modul import *
from my_modul import *
from my_modul import *
```

```
D:\python_sabzlern\1\.
hello world
```

برای اینکه دوباره اجرا شود و مقدار متغیرها به حالت اول برگردد میتوانیم از کتابخانه زیر استفاده کنیم

```
import my_modul
from my_modul import *
from importlib import reload;reload(my_modul)
from my_modul import *
```

```
D:\python_sabzlern\1\
hello world
hello world
```

- فرض کنیم در ماژول یک سری کد های اجرایی داریم و میخواهیم زمانی که آن را ساده میزنیم آن ها اجرا نشود برای این کار میتوانیم از متد __name__ استفاده کنیم به صورت زیر :

```
def my_func(x:int):  1 usage
    return x**2
pi = 3.14
_age = "24"

if __name__ == "__main__":
    print(my_func(5))
```

```
D:\python_sabzlern\
25

Process finished wi
```

دلیل این است که name درون خود ماژول __main__ است ولی در اسکریپت های دیگر نامی است که با پسوند py. مشخص کردیم

- برای دیدن فایل‌هایی که میتوانیم ماژول ایجاد کنیم و برای پایتون شناخته شده است میتوانیم از کد زیر استفاده کنیم

```
from sys import path
for i in path:
    print(i)
```

```
D:\python_sabzlern\1\.venv\Scripts\python.exe D:\python_sab
D:\python_sabzlern\1
D:\python_sabzlern\1
C:\Program Files\JetBrains\PyCharm 2025.1\plugins\python-ce
D:\python_sabzlern\1\.venv\Scripts\python313.zip
C:\Users\Admin\AppData\Local\Programs\Python\Python313\DLLs
C:\Users\Admin\AppData\Local\Programs\Python\Python313\Lib
C:\Users\Admin\AppData\Local\Programs\Python\Python313
D:\python_sabzlern\1\.venv
D:\python_sabzlern\1\.venv\Lib\site-packages
C:\Program Files\JetBrains\PyCharm 2025.1\plugins\python-ce
C:\Program Files\JetBrains\PyCharm 2025.1\plugins\python-ce
```

برای اینکه مسیر جدیدی اضافه کنیم میتوانیم از دستور path.append استفاده کنیم

- همانطور که قبلا هم معرفی شد با استفاده از dir میتوانیم نام‌ها تعریف شده برای مثال تابع ها ، متغیرها و... ببینیم . اگر dir خالی بزنیم در همان اسکریپت نمایش میدهد

```
import my_modul
print(dir(my_modul))
print(dir())
```

```
'__package__', '__spec__', '_age', 'my_func', 'pi']
er__', '__name__', '__package__', '__spec__', 'my_modul']
```

- برای دیدن مسیر ماژل نیز میتوانیم از دستور زیر استفاده کنیم :

```
import my_modul
print(my_modul.__file__)
```

```
D:\python_sabzlern\1\.venv\Scripts
D:\python_sabzlern\1\my_modul.py
```

- Package

مجموعه ای از ماژول ها که درون یک فایل مشخص هستند پکمیج نام دارد

برای ساخا پکیج یک فولدر میسازیم و درون آن فایل __init__.py قرار میدهیم

میتوان درون یک پکیج،پکیج دیگیر باشد که به صورت زیر صدا میزنیم

```
import pak1.pk2.pk3.modul
```

بهتر است اگر از from استفاده کردیم در اخر یک متد یا تابع و یا ماژول را import کنیم

اگر در فایل های __init__ کد اجرایی بنویسیم با صدا زدن آن اجرا میشود

- آشنایی با __all__ :

برای اینکه در ماژول ها زمانی که از * استفاده میکنیم بگوییم چه چیزهایی import شود از all استفاده میکنیم

```
__all__ = ["my_func","pi"]
def my_func(x:int):  1 usage
    return x**2
pi = 3.14
age = "24"
```

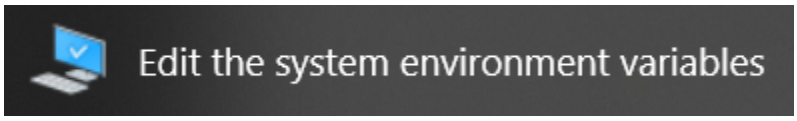
```
from my_modul import *
print(age)
```

```
D:\python_sabzlern\1\.venv\Scripts\python
Traceback (most recent call last):
  File "D:\python_sabzlern\1\test6.py", l
    print(age)
          ^^^
NameError: name 'age' is not defined
```

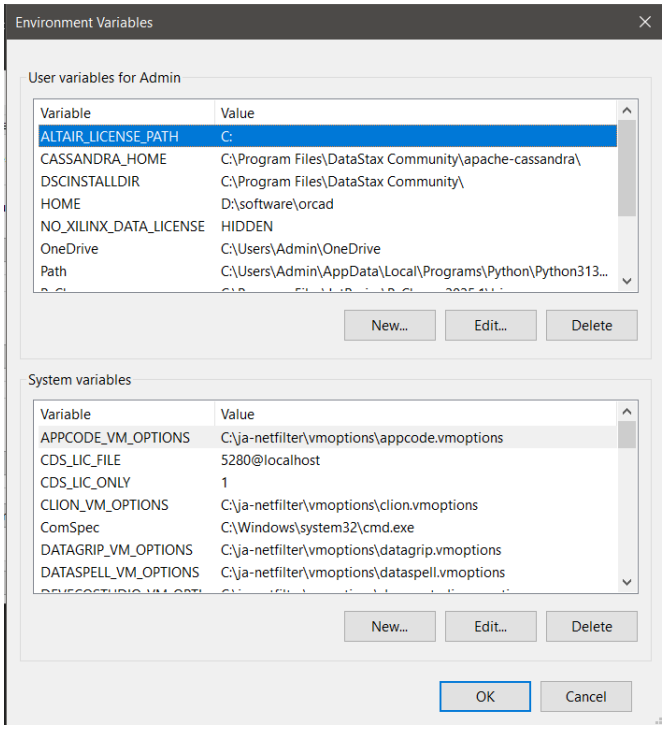
برای درون ماژول ها اختیاری است ولی برای درون پکیج ها درون init اجباری است که وقتی از * استفاده کردیم چه پکیج هایی و ماژول هایی دربر گیرد

- Path

برای اینکه path ها دسترسی داشته باشیم سرچ میکینم



پنجره ای که باز میشود روی گزینه environment variable میزنیم



حال دو محیط داریم سیستم که الویت بالا تری دارد و user

اگر بخواهیم نسخه ها را عوض کنیم اولویت که هرچقدر بالاتر باشد بیشتر است را تغییر میدهیم

برای تغییر نسخه در یک پروژه خاص میتوانیم در cmd بنویسیم

```
:\Users\Admin>py -3.10
```

- Venv

برای اینکه یک محیط مجازی در پایچارم بسازیم خیلی راحت است و قبلا آشنا شدیم

اگر بخواهیم از محیط مجازی استفاده کنیم که پروژه قبلا ساخته شده است از select existing ، وارد venv مورد نظر میشویم و script و python.exe را انتخاب میکنیم

اگر پایچارم در دسترس نبود میتوانیم از دستور زیر استفاده کنیم

```
C:\Users\Admin\Desktop>py -3.13 -m venv env
```

Env نام فولدر است

3.13 نام ورژن پایتون

- Pip

برای نصب پکیج از دستور pip install استفاده میکنیم

اگر ورژن خاصی مدنظر داریم از دستور pip install [package]==x.x.x

برای update کردن پکیج از دستور pip install --upgrade [package]

برای update کردن خود pip از دستور python -m pip install --upgrade pip

برای دیدن لیست پکیج ها از pip list استفاده میکنیم

برای دیدن پکیج هایی که نیاز به ابدیت دارند از pip list --outdated

برای دیدن جزییات پکیج pip show [package]

برای ساخت فایل requirement میتوانیم از کد pip freeze >requirement.txt

برای نصب پکیج ها در فایل requirement وجود دارد pip freeze > requirement.txt

---

فصل نهم: فایل ها، ورودی و خروجی

برای باز کردن فایل از دستور open استفاده میکنیم . ورودی اول آدرس فایل است :

```
f = open(file='test.txt',mode='w')
```

حال میخواهیم با انواع مدها آشنا شویم :

۱- r(rt) : برای خواندن فایل است . درصورت عدم وجود فایل خطا میدهد.اشاره گر از اول فایل است.
۲- w(wt) : برای نوشتن در فایل است . درصورت عدم وجود ایجاد میکند.اشارهگر در اول فایل است.محتویات داخل فایل را پاک میکند
۳- a(at) : برای نوشتن در فایل . اگر نباشد ابتدا ایجاد میشود .اشارهگر به آخر فایل اشاره دارد
۴- x(xt) : برای نوشتن در فایل است . در صورت عدم وجود فایل خطا میدهد.اشاره گر به اول فایل اشاره دارد.محتویات داخل فایل را پاک میکند
۵- rb,wb,ab,xb : ویژگی های بالا را دارد و برای فایل های باینری است
۶- r+(r+t/rt+) : برای خواندن ونوشتن در فایل . در صورت عدم وجود خطا میدهد . اگر از read استفاده نکنیم و از اول write کنیم کلامت شروع به جایگزین شدن میشود ولی برعکسش در آخر شروع به نوشتن میکند
۷- w+(w+t/wt+) : برای خواندن و نوشتن.متن قبلی پاک میشود. در صورت عدم وجود فایل ساخته میشود
۸- a+(a+t/at+) : هم برای نوشتن و خواندن در فایل است . اشاره گر در آخر فایل است . در صورت عدم وجود فایل آن را ایجاد میکند
۹- x+(x+t/xt+) : مانند w+ است ولی اگر فایل وجود داشته باشد خطا میدهد

- method

- برای نوشتن در فایل‌ها میتوانیم از متد write استفاده کنیم . اگر خود write را پرینت کنیم تعداد کارکترهای نوشته شده را میدهد . \t \n هم کارکتر حساب میشود

```
f = open(file='test.txt',mode='w')
print(f.write("Hossein\tGhorbani\n"))
```

```
D:\pyt
17
```

- اگر بخواهیم کاکترهای Unicode را چاپ کنیم باید به صورت زیر نوشت :

```
f = open(file='test.txt',mode='w',encoding='utf-8')
print(f.write("㌀"))
```

```
D:\pyt
1
```

برای داده های بایتی تعداد را تعداد بایت در نظر میگیرد

```
f = open(file='test.txt',mode='wb')
b = bytes("ㅎ","utf-8")
print(f.write(b))
```

```
D:\python_sabzler
3
```

- متد writelines ورودی یک لیست یا دیکشنری(فقط کلیدها) میگیرد و پشت سرهم مینویسید
- برای خواندن میتوانیم از متد read استفاده کنیم . ورودی که میتوان به آن داد این است که چند کارکتر به جلو رود.بعد از آن نشانگر در آنجا قرار میگیرد
- میتوانیم برای جلوگیری از خطا از دو متد readable and writable استفاده کنیم

```
f = open(file='test.txt',mode='rb')
if f.readable():
    print(f.read(3))
```

- Readline متدی است که خط به خط میخواند

اگر درون readlineعدد قرار دهیم تعداد مشخص شده همان خط را برمیگرداند

اگر در خط بعد تکرار کنیم به خط بهد نمیرود بلکه از همان نشانگر ادامه میدهد تا زمانی که \n برسد

مثال زیر را برای دو اسم Alireza , Hossein در خط های جدا نوشته شده است دقت کنید

```
f = open(file='test.txt',mode='r')
print(f.readline(2),end="")
print(f.readline(2),end="")
print(f.readline(2),end="")
print(f.readline(10),end="")
print(f.readline(10),end="")
```

```
D:\python_sabzlern\1
Alireza
Hossein

Process finished wit
```

- readlines همه خط ها را میخواند و در لیست قرار میدهد

```
f = open(file='test.txt',mode='r')
print(f.readlines(),end="")
```

```
D:\python_sabzlern\1\.venv\Scripts\python.ex
['Alireza\n', 'Hossein\n', 'Mohammad\n']
Process finished with exit code 0
```

- اگر بخواهیم روی خط ها حرکت کنیم به راحتی میتوانیم for برنیم

```
for line in open(file='test.txt'):
    print(line, end='')
```

```
D:\python_sabzlern\1\.venv\
Alireza
Hossein
Mohammad
```

- Buffer

اول با close آشنا شویم . close باعث میشود که فایلی که باز کردیم بسته شده و دیتاها از بافر به هارد دیسک منتقل شود.

بافر حافظه ای است که باعث میشود cpu کمتر درگیر شود .

بیایم با یک مثال بررسی کنیم :

برای متوقف کردن برنامه از input استفاده کردیم

```
f = open('test.txt','w')
f.write('Hossein Ghorbani')
x = input("enter input:")
f.close()
```

```
D:\python_sabzl
enter input:
```

تا زمانی که ورودی x وارد نشود در فایل text چیزی نوشته نمیشود و داده ها در بافر هستند

حال برای خالی کردن بافر میتوانیم از متد flash استفاده کنیم که به صورت زیر است :

```
f = open('test.txt','w')
f.write('Hossein Ghorbani')
f.flush()
x = input("enter input:")

f.close()
```

حال از بافر داخلی به بافر سیستم عامل منتقل میشود

برای اینکه مطمئن شویم که ۱۰۰ درصد در هارد دیسک ما نوشته میشود و میتوانیم از دستور زیر از ماژول os استفاده کنیم

```
import os
f = open('test.txt','w')
f.write('Hossein Ghorbani')
os.fsync(f)
x = input("enter input:")
f.close()
```

برای open ارگمانی داریم به نام بافر که میتواند مقدار های مختلفی بگیرد

۱- 0 : برای حالت بایت است و هیچ چیزی در بافر ذخیره نمیکند

```
f = open('test.txt','wb',buffering=0)
f.write(b'Hossein Ghorbani')
x = input("enter input:")
f.close()
```

۲- 1 : برای رشته است و به صورت خط به خط بافر را خالی میکند

```
f = open('test.txt','w',buffering=1)
f.write('Hossein Ghorbani\n')
f.write('Farid KIabiri')
x = input("enter input:")
f.close()
```

```
1   Hossein Ghorbani
2
Run   test6
D:\python_sabzlern\1\.venv\Scripts\
enter input:
```

۳- 1- : حالت دیفالت است که به اندازه بافر سیستم مورد نظر ذخیره میکند . برای دیدن طول بافر میتوانیم کد زیر را بزنیم

```
import io
print(io.DEFAULT_BUFFER_SIZE)
```

```
D:\python_sabzler
8192
```

۴- عدد دلخواه: معمولا برای بایت استفاده شده و میتوانیم طول بافر را به صورت دلخواه مشخص کنیم

- متد readlines ورودی میگیرد و اگر تعداد بایت ها بیشتر از آن خط شود به خط بعدی میرود ولی در همان خط اندازه ۸۱۹۲ بایت محدودیت دارد

```
f = open('test.txt','r')
print(f.readlines(10))
```

```
1 Hossein
2 Farid
3 taha
```

```
Run  test6
D:\python_sabzlern\1\.venv\Scripts\python.exe D:\python_sab
['Hossein\n', 'Farid\n']
```

- Pointer

- برای فهمیدن موقعیت کرسر در فایل میتوانیم از دستور tell استفاده کنیم . tell تعداد بایت‌ها را به ما میگوید

```
f = open('test.txt','w')
f.write("Ali\n")
print(f.tell())
f.write("Reza")
print(f.tell())
```

```
D:\python
5
9
```

نکته بسیار مهم این است که \n در ویندوز درواقع \r\n است و دو کارکتر محسوب میشود

- برای تغییر موقعیت خود میتوانیم از seek استفاده کنیم

```
f = open('test.txt','w')
f.write("Ali\n")
print(f.tell())
f.write("Reza")
print(f.tell())
f.seek(0)
f.write("yap")
```

```
1 yap
2 Reza
```

```
Run  test6
D:\python_sabzlern\1\.venv\Scripts\python.exe D:
5
9
```

Seek دو ارگمان دارد.

اولی برای بایاس است دومی سه حالت دارد

۱- 0:ابتدای فایل
۲- ۱:موقعیت فعلی
۳- ۱-:انتهای فایل

برای حالت رشته باید یکی از این دو آرگمان صفر باشد

- With

With برای context manager های استفاده میشود

حالا context manager ها چی هستند ؟

یک سری دستور هستند که قبل و بعد از یک سری کد یک کاری انجام میدهد

برای بهتر فهمیدن آن یک context manager در زیر نوشته شده است

```
class test:  1 usage
    def __enter__(self):
        print("start")
    def __exit__(self, exc_type, exc_val, exc_tb):
        print("end")

with test():
    print("loading...")
```

```
Run   test6
D:\python_sabzlern\1\.venv
start
loading...
end
```

اگر enter چیزی را return کند میتوانیم با استفاده از عبارت as آن را ببینیم

```
class test:  1 usage
    def __enter__(self):
        print("start")
        return "test as"
    def __exit__(self, exc_type, exc_val, exc_tb):
        print("end")

with test() as a:
    print(a)
    print("loading...")
```

```
Run   test6
D:\python_sabzlern\1\.venv\
start
test as
loading...
end
```

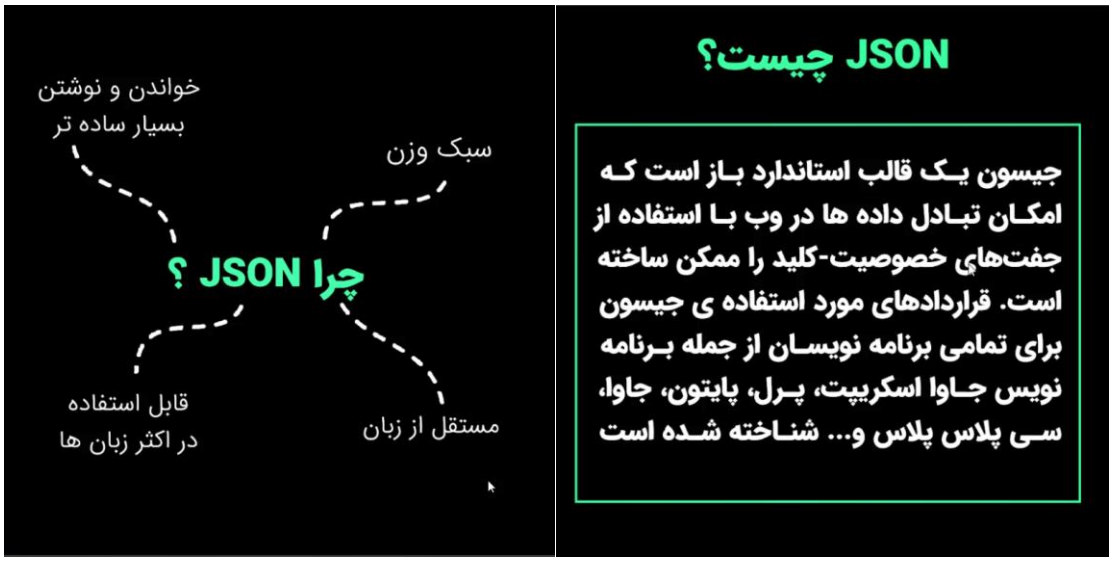
- حال به سراغ فایل ها میرویم . اگر به صورت with نوشته شود در enter آن خود فایلreturn میشود و در exit فایل به صورت خودکار بسته میشود . میتوانیم چندین فایل به صورت پشت سر هم نوشت

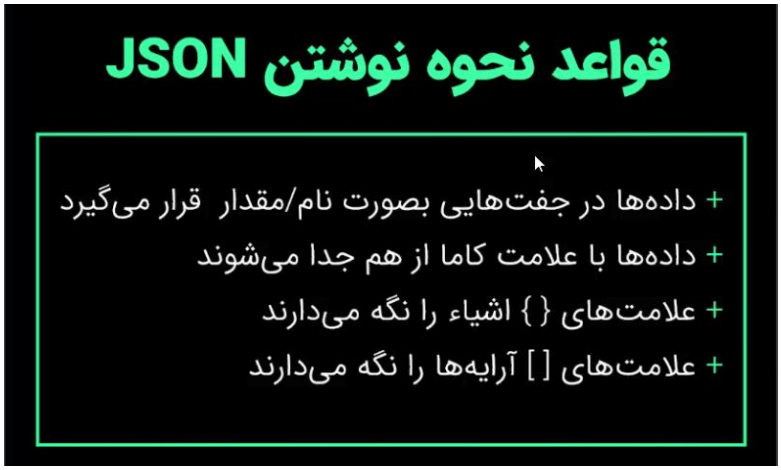
```python
with open("test.txt","w") as f1, open("test.txt","r") as f2:
    f1.write("test")
    f1.flush()
    f2.seek(0)
    print(f2.readlines())
```

```
Run    test6
D:\python_sabzlern\1\.venv\Scri
['test']
```

- Json

Java script object notation مخفف شده json است





python	Java script
Int/float	**number**
True/False	**true/false**
None	**null**
dict	**Object**

- در ادامه میخواهیم با دستورات json آشنا شویم

1- Loads : json string to python format

برای مثال عبارت‌های زیر را بررسی میکنیم

```
import json
js = 'true'
py = json.loads(js)
print(py,'\t',type(py))
```

```
Run    test6
D:\python_sabzlern\1\.venv\Scripts
True      <class 'bool'>
```

```
import json
js = '{"a":1,"b":2}'
py = json.loads(js)
print(py,'\t',type(py))
```

```
Run    test6
D:\python_sabzlern\1\.venv\Scripts\python.e
{'a': 1, 'b': 2}     <class 'dict'>
```

```python
import json

js = '[1 ,2 ,3 ,4]'
py = json.loads(js)
print(py, '\t', type(py))
```

```
Run    test6
D:\python_sabzlern\1\.venv\Scripts\pytho
[1, 2, 3, 4]     <class 'list'>
```

```python
import json

js = '4'
py = json.loads(js)
print(py, '\t', type(py))
```

```
Run    test6
D:\python_sabzlern\1\.venv\Scripts\pyt
4    <class 'int'>
```

```python
import json

js = '4.5'
py = json.loads(js)
print(py, '\t', type(py))
```

```
Run    test6
D:\python_sabzlern\1\.venv\Sc
4.5      <class 'float'>

Process finished with exit co
```

```python
import json
js = 'null'
py = json.loads(js)
print(py,'\t',type(py))
```

```
D:\python_sabzlern\1\.venv\Script
None     <class 'NoneType'>
```

2- load:json file to python type

برای مثال فایل json زیر را داریم

```
{
  "name": "Hossein",
  "lastname": "Ghorbani"
}
```

با زدن دستور load به نوع پایتنی تیدیل میشود

```python
import json

with open("test.json","r",encoding="utf-8") as f:
    data = json.load(f)
print(data)
```

```
D:\python_sabzlern\1\.venv\Scripts\python.exe
{'name': 'Hossein', 'lastname': 'Ghorbani'}

Process finished with exit code 0
```

3- dumps: python type to json string

عکس مثال های loads است که در زیر یک نمونه اوردم

```python
import json

py = True
js = json.dumps(py)
print(js,'\t',type(js))
```

```
Run    test6
D:\python_sabzlern\1\.venv\S
true     <class 'str'>
```

4- dump: python type to json file

```python
import json

py = {
    "name": "Hossein",
    "lastName": "Ghorbani",
    "age": 22,
    "marks": [20, 15, 16, 14]
}
with open("test.json", "w",encoding="utf8") as f:
    json.dump(py, f,indent=4)
```

```
{
    "name": "Hossein",
    "lastName": "Ghorbani",
    "age": 22,
    "marks": [
        20,
        15,
        16,
        14
    ]
}
```

- csv

این نوع فایل‌ها به صورت جدولی هستند

برای نوشتن در فایل های csv باید ابتدا شی write را ایجاد کنیم

سپس با استفاده از writerow میتوانیم به صورت تک سطر تک سطر نوشت و یا با استفاده از writerows میتوانیم به صورت چندسطر چندسطر نوشت

```
import csv

with open("test.csv", "w",newline="") as csvfile:
    csvwriter = csv.writer(csvfile)
    csvwriter.writerow(["A","B","C"])
    csvwriter.writerows([[1,2,3],[4,5,6],[7,8,9]])
```

```
A,B,C
1,2,3
4,5,6
7,8,9
```

- برای خواندن از فایل های csv میتوانیم از متد reader استفاده کنیم که یک obj، itrerator میدهد و روی آن میتوانیم for بزنیم

```
import csv

with open("test.csv", "r",newline="") as csvfile:
    data = csv.reader(csvfile)
    for row in data:
        print(row)
```

```
D:\python_sabzlern\1\.venv\Scr
['A', 'B', 'C']
['1', '2', '3']
['4', '5', '6']
['7', '8', '9']
```

- اگر به صورت دیکشنری بخواهیم میتوانیم از DictReader استفاده کنیم

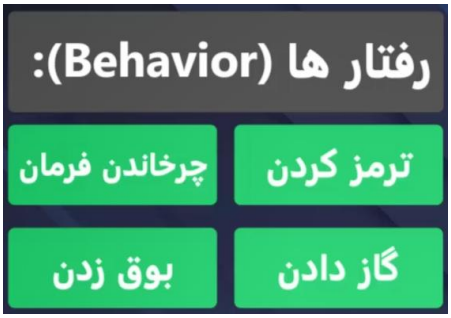
```
import csv

with open("test.csv", "r",newline="") as csvfile:
    data = csv.DictReader(csvfile)
    for row in data:
        print(row)
```

```
Run    test6
D:\python_sabzlern\1\.venv\Scripts\py
{'A': '1', 'B': '2', 'C': '3'}
{'A': '4', 'B': '5', 'C': '6'}
{'A': '7', 'B': '8', 'C': '9'}
```

فصل دهم: کلاس‌ها

در کلاس های باید با مفهوم شی گرایی آشنا شویم

- اولین مفهوم Behavior است که مجموعه ای متدها هستند که یک کار را برای ما انجام میدهد. برای مثال در کلاس ماشین چرخاندن فرمان،گاز دادن،ترمز گرفتن و ...
- ویژگی‌های کلاس attribute هستند برای مثال در همان کلاس ماشین میتوانیم به رنگ ماشین ، سال ساخت ماشین و... اشاره کنیم



صفات	رفتارها	کلاس
نام صاحب حساب شماره حساب میزان موجودی	واریز پول برداشت پول مشاهده موجودی	حساب بانکی
شماره دانشجویی تعداد واحد پاس شده رشته	محاسبه معدل انتخاب واحد	دانشجو

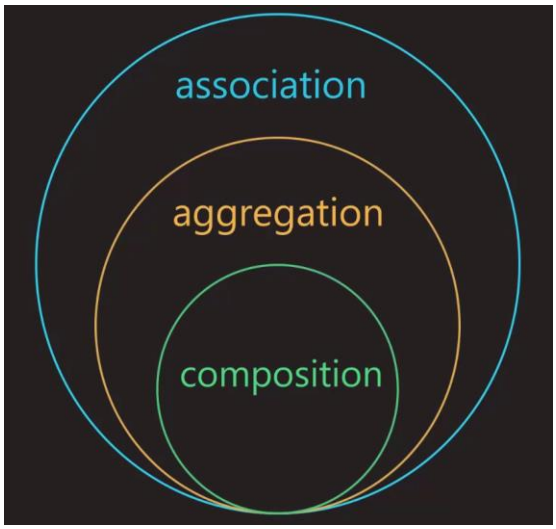
- Encapsulation : به معنی کپسوله سازی است و به مجموعه ای از متدها و صفات مرتبط که در کنار هم میشینند
- Data hiding : پنهان سازی داده به این معنی است که یک سری متدهای مهم و صفت‌های مهم که کاربر نباید به آن ها دسترسی داشته باشد را به صورت خصوصی نوشته. مانند موجودی حساب که کاربر نباید هبآن دسرسی داشته باشد
- Interface : یک سری متدهایی برای عملکرد یک تابع است و برای استفاده کاربر است. باید دقت شود که خود عملکرد درون interface نوشته نشده است و صرفا نحوه استفاده از آن است
- Abstraction : به معنی انتزاع است و تعریف ساده این است که جزییات را حذف کنیم. برای مثال انتزاع حیوان . یا انتزاع خودرو
- Concrete : یک نمونه واقعی از آن انتزاع است برای مثال گربه و یا پراید
- کلاس های انتزاع : کلاس‌هایی که در لول بالا تر هستند و یک سری صفات مشترک دارند برای مثال کلاس انتزاع حیوان ویژگی مشترک قد و وزن و راه رفتن را دارند

- کلاس های ارث بری شده : این نوع کلاس های از کلاس هایی که انتزاع هستند یک سری ویژگی های مشترک را به ارث میبرد . همچنین در بحث چند ریختی یک سری متدها هستند که در کلاس‌های پدر نوشته شده و در کلاس های فرزند overwrite میشود برای مثال کلاس حیوان متد حرکت کردن را دارد و برای کلاس ماهی دوباره نوشته شده و نوع حرکت شنا میشود
- انتزاع به دو صورت است :

۱- Is A : که همان وراثت است

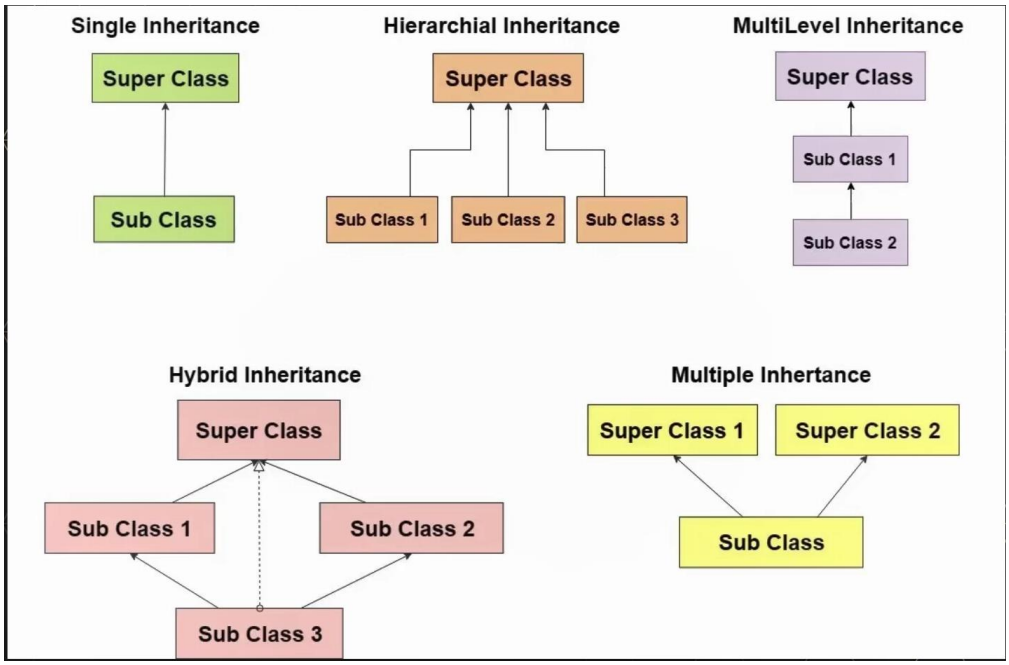
۲- Has A : که ترکیب یا composition است

ترکیب سطح های مختلفی دارد که در زیر آمده است :

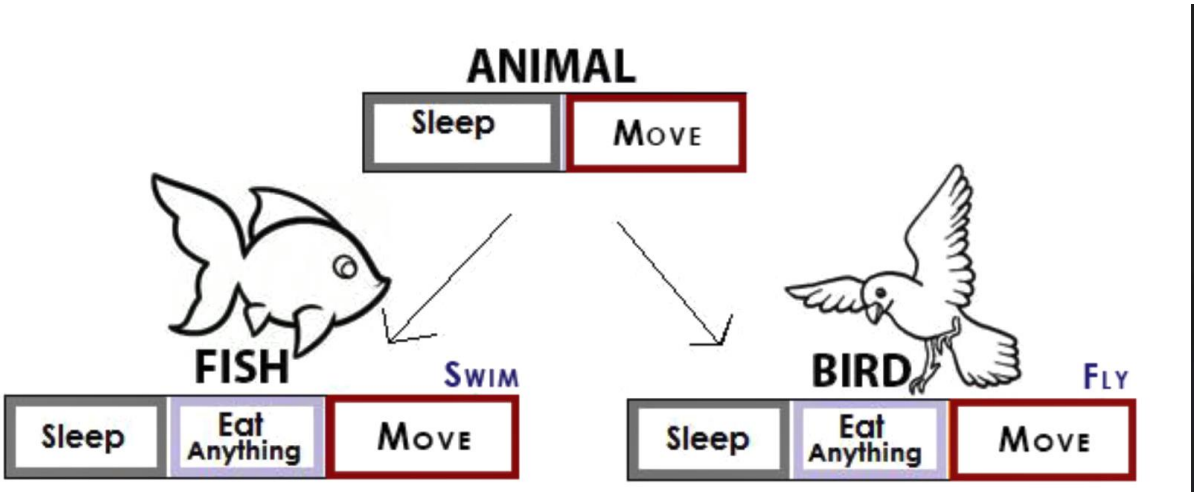


- Association : نوعی است که باهم ارتباط دارند ولی وابستگی به شدت کم . برای مثال دو صفحه اینستاگرام یکدیگر را فالو دارند ولی بود و نبود هر صفحه موجودیت صفحه دیگر را زیر سوال نمیبرد
- Aggregation : نوعی است که وابستگی بیشتری دارند و یک کلاس درون کلاس دیگر است ولی موجودیت مستقل هم هستند . برای مثال باتری موبایل درون گوشی
- Composition : در این مدل دو کلاس به شدت به یکدیگر وابسته هستند و وجود هریک به دیگری وابسته است برای مثال موتور و ماشین

- Inheritance : به معنای وراثت هست و رابطه IS A را توضیح میدهد. برای مثال پراید یک خودرو است و یا پرنده یک موجود است . انواع وراثت داریم که در زیر آمده است :



- به کلاس parent ، نیز super class میگویند
- به کلاس child ، نیز sub class میگویند
- نحوه جست و جو متد ها در کلاس ها از پایین به بالا و چپ به راست است
- Polymorphism : به معنی چندریختی است . برای مثال با در کلاس حیوان متد حرکت کردن را داریم ولی برای ماهی شنا کردن است و برای پرنده پرواز کردن



یا به طور مثال کلاس کارمند داریم و در این کلاس نوع محاسبه حقوق مدیر با منشی فرق دارد

مثال واقعی ما تابع len را داریم و این تابه میتواند طول str,list,tupl را به ما بدهد

یا + برای str یه نوع عملکرد دارد برای list عملکرد دیگر

- Duck typing : یعنی برای ما مهم نیست واقعا شی چی باشد و اگر صدای اردک و راه رفتن اردک را به ما نشان دهد در آن صورت اردک است . برای مثال متد len برایش مهم نیست که در واقع شی ما چیست و اگر بتواند طول آن را بشمارد این کار را میکند

اشنایی با یک سری قوانین :

۱- همه چیز در پایتون شی است
۲- هر شی باید حداقل نمونه ای از یک کلاس باشد
۳- یک متغیر یک ارجاع به شی است
۴- هر کلاسی خود یک شی از type است و همه کلاس ها از کلاسی به نام obj ارث بری میکنند

```
class Person:  2 usages
    def __init__(self, name, age):
        self.name = name
        self.age = age

print(type(Person))
print(Person.__base__)
```

```
Run    test6 ×
D:\python_sabzlern\1\.venv\Scri
<class 'type'>
<class 'object'>
```

۵- در ساختارهای جدید پایتون مفهوم کلاس برابر type درنظر گرفته شده است

- نحوه پیاده سازی

برای نوشتن کلاس از کلمه کلیدی class استفاده میکنیم

اسم کلاس‌ها باید pascalcase باشد یعنی حروف اول کلمه باید بزرگ باشد

اگر اسم کلاس را نوشته و مساوی با یک متغیر قرار دهیم در آن صورت یک شی یا instance ایجاد کردیم

برای اینکه attributeایجاد کنیم میتوانیم به صورت زیر عمل کنیم

```
class Point:  2 usages
    pass


p1 = Point()
p2 = Point()

p1.x = 5
p1.y = 10
p2.x = 3
p2.y = 7
```

برای نوشتن method باید مانند تابع نوشته و از self استفاده کنیم که اشاره به همان obj دارد . برای مثال متد حرکت دادن نقطه در صفحه دوبعدی در زیر آمده است :

```
class Point:  2 usages
    def move(self, x, y):  1 us
        self.x = x
        self.y = y
```

```
p1.move( x: 1, y: 1)
print(p1.x, p1.y)
```

```
D:\python_
1 1
```

نحوه دیگر صدا زدن متد روی یک obj به صورت زیر است :

```
Point.move(p1, x: 5, y: 3)
print(p1.x, p1.y)
```

```
5 3
```

- Doc string

کلاس ها مانند توابع لازم به مستند سازی دارند . قاعده کلی آن این است که در خط اول یک توضیح کوتاه نوشته شود ، در خط دوم مثال زده شود ، در خط سوم توضیم متد ها ، در خط دوم توضیح صفات و در خط آخر interface ها

مثال ها باید مانند حالت تعاملی پایتون باشد :

```
def __init__(self,x:int=0,y:int=0):
    """

    :param x:
    :param y:
    """
    self.x = x
    self.y = y
```

```
class Point:  2 usages
    """
    lorem ipsum dolor sit amet.
    >>> p1 = Point()
    >>> p2 = Point()
    >>> p1.move(10,10)
    >>> p1.x
    10
    """
```

- برای تست کردن هم باید از کتابخانه doctest استفاده کنیم

```
import doctest
doctest.testmod()
```

- برای دیدن مستندهای کلاس‌ها میتوانیم از help استفاده کنیم

- نیم نگاهی به مدیریت استثنا

برای مدیریت استثناها میتوانیم از ساختار try & except استفاده کنیم

از هر کجا در بدنه try خطا بخوریم از بدنه try خارج شده و وارد except شده

```
try:
    x= int(input("x: "))
    y= int(input("y: "))
    print(x/y)
    print("END!!!!!!!!!")
except ZeroDivisionError:
    print("Error")
```

```
Run    test6
D:\python_sabzl
x: 12
y: 0
Error
```

میتوانیم مانند elif، except های مختلف نوشت :

```
try:
    x= int(input("x: "))
    y= int(input("y: "))
    print(x/y)
    print("END!!!!!!!!!")
except ZeroDivisionError:
    print("Error1")
except ValueError:
    print("Error2")
```

```
Run    test6
D:\python_sabzler
x: 5
y: sdv
Error2
```

- اگر except خالی نوشت یعنی هر خطایی جر این ها بود بدنه درون خود را اجرا کن
- else : اگر به هیچ خطایی نخوریم داخل بدنه else اجرا میشود
- finally : در هر صورت به خطا بخوریم و یا نه داخل بدنه finally اجرا میشود

- Modul

فرض کنیم ماژول زیر را نوشته ایم و میخواهیم صدا کنیم

```
from modul1.modul2.inner import Animal
```

میخواهیم مسیر کوتاه‌تری رویم که به صورت زیر باشد :

```
from modul1 import Animal
```

برای این کار در فایل init مورد نظر کد زیر را نوشته :

```
from .modul2.inner import Animal
```

حال به هدف خود میرسیم

- در ادامه بعضی وقت ها لازم است که از یک کلاس فقط یک obj داشته باشیم (معمولا در پایگاه داده ها) و برای ساخت obj خیلی زمانبر است . برای این کار در ماژول مربوطه کد زیر را مینویسیم :

```
point = None
def get_point():
    global point
    if Point is  None:
        sleep(5)
        point = Point()
    return Point
```

حال برای استفاده از آن get_point را import میکنیم و تا زمانی که تابع را صدا نزنیم در آن صورت شی ساخته نمیشود

```
from Point import get_point
```

در ادامه گفتیم بهتر است تایپ های را معلوم کنیم ولی برای point ، None مشخص کردیم. چه کنیم که مشکل پیش نیاید. برای این کار از کلاس typing استفاده میکنیم که میگوییم که تایپ یا None است و یا Point

```
from typing import Optional
point : Optional["Point"] = None
```

- فرض کنیم یک اسکریپت داریم که هم برای اجرای کد است و هم برای استفاده به عنوان ماژول است

برای مدیریت آن باید برنامه های اجرایی را در یک تابع به نام main نوشته و با استفاده از __name__ آن را صدا بزنیم که به صورت زیر است :

```
def main():  1 usage
    print("nothing")
if __name__ == "__main__":
    main()
```

- Access modifies

سه سطح دسترسی داریم :

۱- Public : متد یا متغیر برای همه در دسترس است و بیرون و داخل کلاس میتوانیم استفاده کنیم
۲- Private : فقط داخل کلاس در دسترس است
۳- Protection : برای داخل کلاس و کلاس‌هایی که ارث بری کرده اند در دسترس است

در پایتون به صورت مستقیم سطح درسترسی پیاده سازی نشده است و فقط یک hint میدهد و باید _ استفاده کنیم

- کاربردهای _ :
۱- برای نام گذاری متغیری که هرگز کاربرد ندارد

```
a, _, _ = (12, 15, 14)
```

همچنین درون for دیدیم

۲- قبل اسم متغیر یا متد که یک protection میسازد

```
class User:  1 usage  new *
    def __init__(self, name, age):  new *
        self.name = name
        self.age = age
        self._User_id = id(self)

ali = User( name: 'ali',  age: 18)
print(ali._User_id)
```

```
Run    test ×
D:\python_sabzlern\1\.ver
1698078020608
```

درست است protection است ولی خارج از کلاس به آن درسترسی پیدا کردیم ولی زیر آن خط کشیده

- برای ماژول ها اگر از * استفاده کنیم آن را import نمیکند ولی اگر از __all__ استفاده کنیم و ذکر کنیم آنگاه import میشود
۳- اگر __ قبل از نام متغیر قرار دهیم آنگاه private میشود و دسترسی مستقیم به آن نداریم :

```
class User:  1 usage  new *
    def __init__(self, name, age):  new *
        self.name = name
        self.age = age
        self.__User_id = id(self)

ali = User( name: 'ali',  age: 18)
print(ali.__User_id)
```

```
Run   test
D:\python_sabzlern\1\.venv\Scripts\python.ex
Traceback (most recent call last):
  File "D:\python_sabzlern\1\test.py", line
    print(ali.__User_id)
          ^^^^^^^^^^^^^
AttributeError: 'User' object has no attribu
```

برای دسترسی به آن میتوانیم به صورت زیر استفاده کنیم :

```
ali = User( name: 'ali',  age: 18)
print(dir(ali))
print(ali._User__User_id)
```

```
D:\python_sabzlern\1\.v
['_User__User_id', '__c
1452832150528
```

در بحث ارث بری برای آنکه نام ها با یکدیگر قاطی نشود به صورت زیر نام گذاری میشود :

```
class User:  1 usage  new *
    def __init__(self, name, age):  new *
        self.name = name
        self.age = age
        self.__User_id = id(self)


class Account(User):  1 usage  new *
    def __init__(self, name, age):  new *
        super().__init__(name, age)
        self.__User_id = id(self)

ali = Account( name: 'ali',  age: 18)
print(dir(ali))
print(ali._User__User_id,'\t',ali._Account__User_id)
```

```
Run   test
D:\python_sabzlern\1\.venv\Scripts\pytho
['_Account__User_id', '_User__User_id',
2811049104384    2811049104384
```

- خیلی رایج است که getter , setter برای attribute ها نوشته شود

```python
def set_phone(self, phone):
    if len(phone) == 11 and phone.isdigit():
        self.__phone = phone

def get_phone(self):
    return self.__phone
```

- Repr & string

Repr معمولا برای برنامه نویس نوشته میشود و str برای کاربر نهایی

به طور مثال برای time و یا str

```python
print(datetime.datetime.now())
print(repr(datetime.datetime.now()))
```

```
2025-06-24 00:01:11.196382
datetime.datetime(2025, 6, 24, 0, 1, 11, 196418)
```

برای کلاس های خود میتوانیم repr و str داشته باشیم برای این کار باید از __str__ و __repr__ استفاده کنیم

```python
class User:  1 usage  new *
    def __init__(self, name, age):  new *
        self.name = name
        self.age = age
        self.__User_id = id(self)

    def __str__(self):  new *
        return f'{self.name}, {self.age}'
    def __repr__(self):  new *
        return f'{self.__class__.__name__} ({self.name} ,{self.age} ,{self.__User_id})'

user1 = User( name: 'John',  age: 18)
print(user1)
print(repr(user1))
```

```
Run   test
D:\python_sabzlern\1\.venv\Scripts\python.exe D:
John, 18
User (John ,18 ,2429003389584)
```

در ترمینال دیفالت print، repr است ولی در پایچارم str است

- Class attribute & instance attribute

Instance class : این مدل صفات به ازای هر شی منحصر به فرد است و معمولا با کلمه کلیدی self همراه است

Class attribute : این مئل صفات بین اشیا مشترک هستند که خارج از متدها نوشته میشود

مثال زیر را ببینیم :

```
class User:  6 usages  new *
    all_users:list["User"] = []
    def __init__(self, user_name:str, email:str, password:int):  new *
        self.user_name = user_name
        self.email = email
        self.password = password
        User.all_users.append(self)

    def __str__(self):  new *
        return f'{self.__class__.__name__}({self.user_name!r}, {self.email!r}, {self.password})'
    def __repr__(self):  new *
        return f'{self.user_name}'
```

برای آنکه rapr به صورت '' نمایش دهد از !r استفاده میکنیم

نکته : در ورژن‌های قبلی پایتون باید از ماژول typing برای list استفاده میکردیم

نکته : برای آنکه یک ویژگی در ورژن های بعدی است ولی سرور ما از ورژن های قبلی پشتیبانی میکند پیاده سازی کنیم میتوانیم از دستور زیر در خط اول استفاده کنیم تا ابدیت شود

```
from __future__ import annotations
```

حال میخواهیم چند کاربر اضافه کنیم

```
user1 = User( user_name: "ali123",  email: "ali123@gmail.com",  password: 654654)
user2 = User( user_name: "reza123",  email: "reza123@gmail.com",  password: 5343)
user3 = User( user_name: "sara123",  email: "sara123@gmail.com",  password: 12315)
```

برای تغییر نباید از نام instance استفاده کنیم و اینکار باعث میشود که یک attr جدید برای آن شی ساخته شود و باید از نام کلاس استفاده کنیم

نکته: برای دیدن attr میتوانیم از dir استفاده کنیم و متوجه میشویم که instance attr ها فقط برای یک سری شی ها وجود دارد ولی class attr ها برای همه شی ها

- Pprint را میتوانیم برای نمایش بهتر str ها استفاده کنیم

- Inheritance

قبلا با تئوری ارث‌بری آشنا شدیم حال میخواهیم با نحوه پیاده سازی آن آشنا شویم

برای اینکه از یک کلاس ارث بری کنیم در پرانتز نام آن را مینویسیم

```
class Seller(User):  1 usage  new *
    pass
```

در اینجا attr ها و متد ها ارث بری شده اند حتی class attr ها

- میتوانیم کلاس‌های داخلی خود پایتون را ارث بری کنیم و متد های آن را override کنیم و یا به آن ها متد اضافه کنیم . ولی بهتر است override نکنیم . برای مثال میخواهیم لیست را ارث بری کنیم و یک تایپ جدید که برای user است بازنویسی کنیم و یک متد برای سرچ کردن بنویسیم :

```
class ListUser(list["User"]):  1 usage  new *
    def search(self, name: str) -> list["User"]:  new *
        match_user: list["User"] = []
        for user in self:
            if name in user.user_name:
                match_user.append(user)
        return match_user


class User:  7 usages  new *
    all_users: list[User] = ListUser()
```

- برای آنکه یک متد را override کنیم نام آن متد را نوشته و دستورات جدید را درون آن مینویسیم
- برای __init__ که یک سری صفات از کلاس پدر امده از کلمه کلیدی super استفاده میکنیم . برای مثال __init__ و __repr__ را بازنویسی کردیم

```
class Buyer(User):  new *
    def __init__(self, user_name: str, email: str, password: int, phone_number: str):  new *
        super().__init__(user_name, email, password)
        self.phone_number = phone_number

    def __repr__(self):  new *
        return f'{self.__class__.__name__}({self.user_name!r}, {self.email!r}, {self.password}, {self.phone_number!r})'
```

- در مثالی دیگر میخواهیم برای تابه append به این صورت عمل کنیم اگر غیر user گرفت خطا دهد

```
class ListUser(list["User"]):  1 usage  new *
    def search(self, name: str) -> list["User"]:  new *
        match_user: list["User"] = []
        for user in self:
            if name in user.user_name:
                match_user.append(user)
        return match_user

    def append(self, user: User) -> None:  new *
        if not isinstance(user, User):
            raise TypeError("user is not a User")

        return super().append(user)
```

- Multiple inheritance

وراثت چندگانه معمولا توصیه نمیشود و بهتر است استفاده نکنیم

مثال زیر را در نظر بگیریم :

```
class Super1:  1 usage  new *
    def printer(self, ) -> None:  1 usage  new *
        print("super1")


class Super2:  1 usage  new *
    def printer(self) -> None:  new *
        print("super2")


class Sub(Super1, Super2):  1 usage  new *
    pass

def main():  1 usage  new *
    obj = Sub()
    obj.printer()

if __name__ == "__main__":
    main()
```

```
Run    test1 ×
D:\python_sabzlern\1\.venv\Scripts\p
super1

Process finished with exit code 0
```

در این مثال چرا super1 پرینت شد . زیرا اول نوشته شد ولی این راه حل درستی نیست

یا در مثال زیر ما میخواهیم برای super1 مقدار a1 را مقدار دهی کنیم ولی اصلا چنین attr ساخته نمیشود

```python
class Super1:  1 usage  new *
    def __init__(self, a1: int) -> None:  new *
        self.a1 = a1


class Super2:  1 usage  new *
    def __init__(self, a2: int) -> None:  new *
        self.a2 = a2


class Sub(Super2, Super1):  1 usage  new *
    def __init__(self, a1, a3) -> None:  new *
        super().__init__(a1)
        self.a3 = a3


def main():  1 usage  new *
    obj = Sub( a1: 1, a3: 2)
    obj.a1


if __name__ == "__main__":
    main()
```

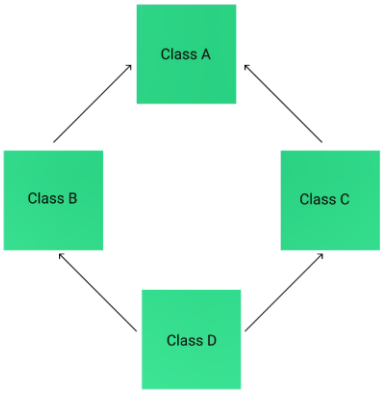
```
Run    test1
D:\python_sabzlern\1\.venv\Scripts\python.exe D:\python_sabzlern\1\test1.
Traceback (most recent call last):
  File "D:\python_sabzlern\1\test1.py", line 23, in <module>
    main()
    ~~~~^^
  File "D:\python_sabzlern\1\test1.py", line 19, in main
    obj.a1
AttributeError: 'Sub' object has no attribute 'a1'. Did you mean: 'a2'?

Process finished with exit code 1
```

برای حل این مشکل راه اول و ساده این است که به صورت زیر کد بزنیم :

```python
class Sub(Super2, Super1):  1 usage  new *
    def __init__(self, a1, a2, a3) -> None:  new *
        Super1.__init__(self, a1)
        Super2.__init__(self, a2)
        self.a3 = a3
```

- Diamond

میخواهیم ابتدا مشکل الماس را بررسی کنیم برای اینکار ابتدا ساختار زیر را در نظر بگیریم :



کلاس های بالا را پیاده‌سازی میکنیم ابتدا A :

```
class A :  4 usages  new *
    number_of_calls_A = 0
    def call_me(self):  4 usages  new *
        self.number_of_calls_A += 1
        print("class A Called")
```

دقت شود برای number_of_calls عبارت self قرار دادیم و برای هر obj یک attr به این نام ساخته میشود و وقتی میگوییم print(a1.number_of_calls) و print(a2.number_of_calls) تعداد آن ممکن است فرق داشته باشد و هرکدام یک attr جدا گانه برای obj است

: B

```
class B(A) :  1 usage  new *
    number_of_calls_B = 0
    def call_me(self):  1 usage  new *
        super().call_me()
        A.call_me(self)
        self.number_of_calls_B += 1
        print("class B Called")
```

: C

```
class C(A) :  1 usage  new *
    number_of_calls_C = 0
    def call_me(self):  1 usage  new *
        super().call_me()
        A.call_me(self)
        self.number_of_calls_C += 1
        print("class C Called")
```

: D

```
class D(B,C):  1 usage  new *
    number_of_calls_D = 0
    def call_me(self):  1 usage  new *
        super().call_me()
        self.number_of_calls_D += 1
        print("class D Called")
```

حال نتیجه را وقتی کلاس D را صدا میکنیم :

```
class A Called
class B Called
class A Called
class C Called
class D Called
```

همانطور که میتوان دید class A دوبار صدا زده شده است که مشکل ساز است

برای حل این مشکل از الگوریتم mro استفاده میکنیم که نحوه پیاده سازی آن در super است .

```
class A :  2 usages  new *
    number_of_calls_A = 0
    def call_me(self):  2 usages  new *
        self.number_of_calls_A += 1
        print("class A Called")

class B(A) :  1 usage  new *
    number_of_calls_B = 0
    def call_me(self):  1 usage  new *
        super().call_me()
        self.number_of_calls_B += 1
        print("class B Called")


class C(A) :  1 usage  new *
    number_of_calls_C = 0
    def call_me(self):  1 usage  new *
        super().call_me()
        self.number_of_calls_C += 1
        print("class C Called")

class D(B,C):  1 usage  new *
    number_of_calls_D = 0
    def call_me(self):  1 usage  new *
        super().call_me()
        self.number_of_calls_D += 1
        print("class D Called")
```

```
class A Called
class C Called
class B Called
class D Called
1 1 1 1
```

برای دیدن ترتیب اجرا الگوریتم mro میتوانیم کد زیر را اجرا کنیم :

```
def main():  1 usage  new *
    obj = D()
    obj.call_me()
    pprint(D.__mro__)
```

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

- برای مقدار دهی به کلاس های multiple inheritance بهتر است از keyargs** استفاده کنیم

Init های زیر را برای کلاس‌های A,B,C,D به ترتیب نوشته ایم

```
def __init__(self, a1, a2) -> None:  new *
    self.a1 = a1
    self.a2 = a2
```

```
def __init__(self, b1) -> None:  new *
    self.b1 = b1
```

```
def __init__(self, c1, c2, c3) -> None:  new *
    self.c1 = c1
    self.c2 = c2
    self.c3 = c3
```

```
def __init__(self, d1) -> None:  new *
    self.d1 = d1
```

میخواهیم به a1,a2,b1,c1,c2,c3,d1 مقدار دهی کنیم .

برای اینکار از **keyargs استفاده میکنیم برای نمونه کلاس D نوشته شده است :

```
def __init__(self, d1, **keyargs) -> None:  new *
    super().__init__(**keyargs)
    self.d1 = d1
```

نتیجه نهایی به صورت زیر است :

```
def main():  1 usage  new *
    obj = D(a1=1, a2=2, b1=3, c1=4, c2=5, c3=6, d1=7)
    obj.call_me()
    print(obj.a1, obj.a2, obj.b1, obj.c1, obj.c2, obj.c3, obj.d1)
```

```
D:\python_sabzlern\
class A Called
class C Called
class B Called
class D Called
1 2 3 4 5 6 7
```

- Polymorphism

در پایتون به صورت دیفالت چند ریختی پیاده‌سازی شده است. برای مثال دو کلاس گاو و گربه پیاده سازی کردیم که دو متد معرفی و صدا دارد که در زیر آمده است :

```
class Cat():  new *
    def __init__(self,name="",color=""):  new *
        self.name = name
        self.color = color

    def info(self):  new *
        print(f"i'm cat, my name is {self.name}, {self.color}")

    def sound(self):  new *
        print("meowwwww")
```

```python
class Cow():  new *
    def __init__(self,name="",color=""):  new *
        self.name = name
        self.color = color
    def info(self):  new *
        print(f"i'm cow,my name is {self.name}, {self.color}")

    def sound(self):  new *
        print("mowwww")
```

حال دو شی میسازیم و در حلقه for اجرا میکنیم :

```python
def main():  1 usage  new *
    cat = Cat()
    cow = Cow()
    for animal in [cat,cow]:
        animal.info()
        animal.sound()

if __name__ == "__main__":
    main()
```

```
Run    Animal
D:\python_sabzlern\1\.ver
i'm cat, my name is ,
meowwwww
i'm cow,my name is ,
mowwww

Process finished with ex
```

همانطور که میتوان دید که به ازای هر شی متد مورد نظر خود اجرا شده است

حال یک تابع مینویسیم . این تابع هم یه صورت خودکار به صورت جداگانه متد های مورد نظر را صدا میزند

```python
def main():  1 usage  new *
    cat = Cat()
    cow = Cow()
    def Animal(obj):  new *
        obj.info()
        obj.sound()
    Animal(cat)
    Animal(cow)
```

```
D:\python_sabzlern\1\.venv\Scripts
i'm cat, my name is ,
meowwwww
i'm cow,my name is ,
mowwww

Process finished with exit code 0
```

راه معمول و استاندارد چندریختی این است که یک کلاس super داشته باشیم و متد های آن را override کنیم

```python
class Animal:  2 usages  new *
    def __init__(self, name, color):  new *
        self.name = name
        self.color = color

    def info(self):  1 usage (1 dynamic)  new *
        print("i'm animal")

    def sound(self):  1 usage (1 dynamic)  new *
        print("i can make sound")
```

```python
class Cat(Animal):  1 usage  new *
    def __init__(self, name="", color=""):  new *
        super().__init__(name,color)

    def info(self):  1 usage (1 dynamic)  new *
        print(f"i'm cat, my name is {self.name}, {self.color}")

    def sound(self):  1 usage (1 dynamic)  new *
        print("meowwwww")
```

- Mixin

این مدل کلاس‌ها برای افزایش functionality کلاس‌ها است . ویژگی های آن عبارت است از :

۱- رابطه IS_A نیست
۲- از روی آن شی نمیسازند
۳- آخر نام کلاس عبارت mixin نوشته میشود

برای مثال یک سری کلاس داریم برای وسیله نقلیه است مانند کلاس هواپیما و خودرو و دوچرخه و...

حال دو کلاس mixin نوشته به نام های wifi , music :

```
class Vehicle:  2 usages  new *
    def __init__(self, model, year):  new *
        self.model = model
        self.year = year


class WifiMixin:  1 usage  new *
    def wifi_conected(self):  new *
        print("wifi conected")


class Car(Vehicle, WifiMixin):  new *
    def __init__(self, model, year):  new *
        super().__init__(model, year)


class Motor(Vehicle):  new *
    def __init__(self, model, year):  new *
        super().__init__(model, year)
```

در این مثال فرض شده است که موتور wifi ندارد .

- Duck typing

قبلا با مفهوم آن آشنا شدیم برای مثال تابع len روی رشته ، لیست ، تاپل کار میکند زیرا همه آن ها متد __len__ را دارند

مفهومی دیگر میخواهیم آشنا شویم :

LBYL: look before you leap

مثال زیر توجه شود :

```
x = 2.55

if hasattr(x,'__len__'):
    print(len(x))
else :
    print('sorry...')
```

EAFP: it’s easier to ask for forgiven than permission

```
x = 2.55

try:
    print(len(x))

except:
    print("error")
```

- حل مثال محاسبه مساحت و محیط

به بخش highlight شده که نحوه مقدار دهی است دقت کنیم :

```
class Shape:  1 usage  new *
    def __init__(self, **keyargs):  new *
        area = 0
        perimeter = 0
        for key, value in keyargs.items():
            setattr(self, key, value)

    def area(self):  new *
        pass

    def perimeter(self):  new *
        pass


class Rectangle(Shape):  1 usage  new *
    def __init__(self, **keyargs):  new *
        super().__init__(**keyargs)

    def area(self):  new *
        self.area = (self.width + self.lengh) * 2
        return self.area

    def perimeter(self):  new *
        self.perimeter = self.width * self.lengh
        return self.perimeter
```

- Call

یک سری کلاس ها را میتواند صدا زد مانند تابع ولی یک سری اینگونه نیست مانند عدد int

برای چک کردن آن میتوانیم از دستور callable استفاده کنیم

```python
def main():  1 usage  new *

    def fun():  new *
        pass
    x = 4
    print(callable(fun))
    print(callable(x))
```

```
D:\python_sabzlern\1\
True
False

Process finished with
```

شی ها به صورت دیفالت قابل فراخوانی نیستند

برای اضافه کردن این ویژگی میتوانیم به صورت زیر نوشت :

```python
class Rectangle(Shape):  1 usage  new *
    def __init__(self, **keyargs):  new *
        super().__init__(**keyargs)

    def area(self):  new *
        self.area = (self.width + self.lengh) * 2
        return self.area

    def perimeter(self):  new *
        self.perimeter = self.width * self.lengh
        return self.perimeter

    def __call__(self,  **kwargs):  new *
        super().__init__( **kwargs)
```

```
def main():  1 usage  new *
    r = Rectangle(lengh=5,width=10)
    r(lengh=3,width=3)
    print(r.perimeter())
```

- Types of method

اولین تایپ که رایج ترین است و تا الان با آن کار کردیم ورودی self میگیرد و به instance method معروف است

```
class Comment:  1 usage  new *
    def __init__(self, name, email, description):  new *
        self.name = name
        self.email = email
        self.date = datetime.now()
        self.description = description

    def show(self):  1 usage  new *
        print(f"name:{self.name},\n"
              f"mail:{self.email},\n"
              f"{self.description}")
```

دومین نوع class method ها است که در همه شی‌ها مشترک است و نیاز به پارمتر self ندارد و پارامتر cls میگیرد .

پیش از آن باید decorator، @classmethod را صدا زد

ویژگی های آن این است که فقط class attr ها را میپذیرد و برای هر instance متفاوت نیست

```python
class Comment:  2 usages  new *
    site_add = "lorem ipsum"
    def __init__(self, name, email, description):  new *
        self.name = name
        self.email = email
        self.date = datetime.now()
        self.description = description

    def show(self):  new *
        print(f"name:{self.name},\n"
              f"mail:{self.email},\n"
              f"{self.description}")

    @classmethod  1 usage  new *
    def site_name(cls):
        print(cls.site_add)


def main():  1 usage  new *
    c1 = Comment( name: 'ali',  email: 'ali@gmail.com',  description: 'good ')
    Comment.site_name()
```

```
Run    comment ×
D:\python_sabzlern\1\.ver
lorem ipsum
```

میتوانیم درون آن شی ایجاد کنیم

```python
    @classmethod  1 usage  new *
    def show_star(cls, name, email, description):
        des = "*" + description + "*"
        return cls(name, email, des)


def main():  1 usage  new *
    c2 =Comment.show_star( name: 'ali',  email: 'ali@gmail.com',  description: 'good ')
    c2.show()
```

```
Run    comment ×
D:\python_sabzlern\1\.venv\Scrip
name:ali,
mail:ali@gmail.com,
*good *
```

نوع آخر static method ها هستند که نه به شی احتیاج دارند و نه کلاس . این نوع متد ها فقط چون جایگاه مناسب برای کلاس دارند میتوانیم استفاده کنیم . فرقی با یک تابع معمولی ندارد .از @staticmethod استفاده میکنیم .

برای مثال میخواهیم زمان گذشته از کامنت گذاشته شده را حساب کنیم

```
    @staticmethod  1 usage  new *
    def time_elpsed(date):
        return date - datetime.now()


def main():  1 usage  new *
    c2 = Comment.show_star( name: 'ali',  email: 'ali@gmail.com',  description: 'good ')
    # sleep(5)
    print(c2.time_elpsed(c2.date))
```

- Property

در زبان‌های دیگر با setter , getter مقدار دهی میکنیم ولی در پایتون از از property استفاده میکنیم

برای پیاده سازی آن ابتدا آن را به صورت _getter, _setter نوشته و سپس از دستور property استفاده میکنیم و ورودی ها را تنظیم میکنیم

مثال برای کلاس color

داخل set , get باید attr ها به صورت _attr باشند

```
class Color:  1 usage  new *
    def __init__(self, name, rgb):  new *
        self.name = name
        self.rgb = rgb

    def _get_rgb(self):  1 usage  new *
        return self._rgb

    def _set_rgb(self, rgb):  1 usage  new *
        if rgb:
            self._rgb = rgb
        else:

            raise ValueError("rgb cannot be None")

    def _get_name(self):  1 usage  new *
        return self._name


    def _set_name(self, name):  1 usage  new *
        if name:
            self._name = name
        else:
            raise ValueError("name is required")

    name = property(_get_name, _set_name)
    rgb = property(_get_rgb, _set_rgb)
```

- ورودی ها property به ترتیب getter , setter , deleter است . علاوه براین میتوانیم doc string نوشت.

اگر درون property ، doc string نوشته نشود آنگاه از getter برمیدارد

برای نمونه متد دلیت زیر را نوشته :

```
def _del_name(self):  1 usage  new *
    del self._name


name = property(fget=_get_name, fset=_set_name,fdel=_del_name,doc="new property")
```

- راه بهتر برای نوشتن property استفاده از decorator منطقی تر است برای این کار ابتدا همه اسم ها را name قرار میدهیم و سپس به ترتیب get , set , del را نوشته

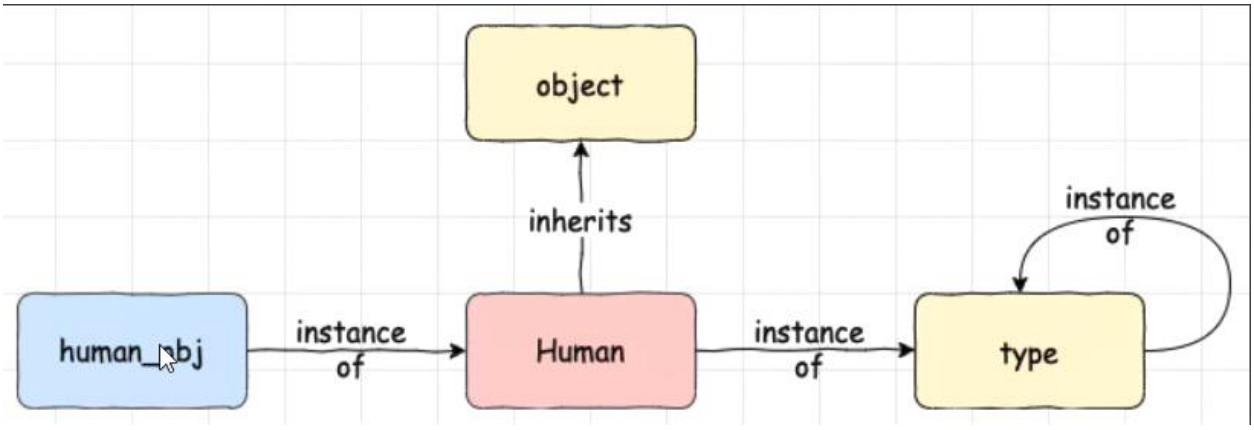
```
@property  3 usages  new *
def name(self):
    return self._name

@name.setter  1 usage  new *
def name(self, name):
    if name:
        self._name = name
    else:
        raise ValueError("name is required")

@name.deleter  1 usage  new *
def name(self):
    del self._name
```

- Composition

قبلا با مفهوم آن آشنا شدیم و حال میخواهیم مثال برگه و سوالات را بزنیم.

```python
class Quesstion:  1 usage  new *
    def __init__(self, question, answer):  new *
        self.question = question
        self.answer = answer

    def __str__(self):  new *
        return f'{self.question}: {self.answer}'


class Exam:  new *
    def __init__(self):  new *
        self.question = Quesstion( question: "what is your name?", answer: "my name is Hossein")

    def __str__(self):  new *
        return f'{self.question}'
```

- Meta class



هر کلاس از کلاس object ارث بری میکند

هرکلاس شی از type است

```python
class Human:  3 usages  new *
    pass

print(type(Human))
print(Human.__bases__)
print(isinstance(Human, object))
```

```
Run    exam
D:\python_sabzlern\1\.v
<class 'type'>
(<class 'object'>,)
True
```

به کلاس type، meta class گویند که میتوان meta class های خود را بسازیم

```
class HumanMeta(type):  1 usage  new *
    pass
class Human(metaclass=HumanMeta):  3 usages  new *
    pass
```

- Abstract class

همانطور که قبلا گفته شد در این نوع کلاس ها از گفتن جزییات پرهیز میشود وفقط متد های مورد نیاز نام آن ها نوشته میشود

برای استفاده از آن باید ABC را import کنیم

ویژگی ها :

۱- میتوان متدهای غیر انتزاعی هم اضافه کرد
۲- میتوان انواع متدها که قبلا با آن آشنا شدیم را پیاده سازی کرد مانند static method
۳- میتوانیم یک پیاده سازی دیفالت هم برای آن نوشت

```
class Vehicle(ABC):  new *

    @abstractmethod  new *
    def drive(self):
        """complete this method"""

    @classmethod  new *
    @abstractmethod
    def brake(cls):
        print("Brake")

    def class_name(self):  new *
        print(self.__class__.__name__)
```

کلاس های که از آن ارث میبرد میتواند به صورت زیر باشد :

۱- خود آن کلاس یک کلاس انتزاعی باشد

```
class Car(Vehicle):  new *

    @abstractmethod  new *
    def driver(self):
        """complete this method"""
```

۲- اگر همه متدهای کلاس انتزاعی پیاده سازی شد در آن صورت کلاس معمولی است
۳- میتوانیم از دیفالت ها به صورت زیر استفاده کنیم

```
class Car(Vehicle):  new *

    def drive(self):  new *
        print("Driving")

    def brake(self):  new *
        super().brake()
```

- Operator overloading

به طور مثال میخواهیم عملگر + را نقش آن را تغییر دهیم برای اینکار magic method مربوطه را تغییر میدهیم

برای مثال میخواهیم زمانی که از + استفاده کردیم سن دو شخص را جمع کند

Self برای شی قبل + و other برای شی بعد +

```python
class Person:  2 usages  new *
    def __init__(self,name,age):  new *
        self.name = name
        self.age = age

    def __add__(self,other):  new *
        return self.age + other.age

person1 = Person( name: "John", age: 20)
person2 = Person( name: "ali", age: 33)
print(person1 + person2)
```

```
Run    exam
D:\python
53
```

- Slot

اگر بخواهیم فقط یکسری attr های خاص مقدار دهی شود و نتوان attr های دیگر تعریف کرد از slot استفاده میکنیم

به مثال زیر توجه شود :

```python
class Test:  2 usages  new *
    __slots__ = ("a", "b", "c")

    def __init__(self, a, b, c):  new *
        self.a = a
        self.b = b
        self.c = c
```

الان میتوانیم فقط a b c را مقداردهی کنیم

- برای کلاس هایی که ارث بری میشود میتوان attr های جدید تعریف کرد
- اگر برای کلاس sub، slot تعریف کنیم در آن صورت هم slot های super class معتبر است و هم slot های sub class . در مثال زیر attr های معتبر برای sub class

```python
class Test:  1 usage  new *
    __slots__ = ("a", "b", "c")

    def __init__(self, a, b, c):  new *
        self.a = a
        self.b = b
        self.c = c


class Test2(Test):  new *
    __slots__ = ("d")
    def __init__(self, a, b, c, d):  new *
        super().__init__(a, b, c)
        self.d = d
```

A B C D

- Iterable & iterator

Iterable : به صوت خلاصه obj که بتوان روی آن for زد

```python
lst = [1,2,3]
print("__iter__" in dir(lst))
print("__next__" in dir(lst))
for item in lst:
    print(item)
```

```
D:\python_sabzl
True
False
1
2
3
```

Iterator : obj که بتوان روی آن next زد

```
n = iter(lst)
print(next(n))
print(next(n))
print(next(n))
```

```
D:\python
1
2
3
```

- به یک فرق مهم بین آن ها دقت شود :

در iterable ها همیشه حرکت از اول شروع میشود :

```
lst = [1,2,3,4,5]
for i in lst:
    if i <=3 :
        print(i)
print("-"*20)
for i in lst:
    print(i)
```

```
D:\python_s
1
2
3
----------
1
2
3
4
5
```

در iterator ها حرکت از ادامه شروع میشود :

```
lst = [1,2,3,4,5]
lst = iter(lst)
for i in lst:
    if i <=3 :
        print(i)
    else :
        break
print("-"*20)
for i in lst:
    print(i)
```

```
D:\python_s
1
2
3
----------
5

Process fin
```

حال میخواهیم برای کلاس خود iterator,iterable پیاده سازی کنیم :

: Iterable

```
class Test:  1 usage  new *
    def __init__(self):  new *
        self.lst = [1,2,3,4,5]

    def __iter__(self):  new *
        for i in self.lst:
            yield i

t= Test()
for i in t:
    print(i)
```

```
Run    exam
D:\python_sa
1
2
3
4
5

Process fin
```

: Iterator

```
class Test:  1 usage  new *
    def __init__(self):  new *
        self.lst = [1, 2, 3, 4, 5]
        self.x = 0
    def __iter__(self):  new *
        for i in self.lst:
            yield i

    def __next__(self):  new *
        self.x += 3
        return self.x
t = Test()
print(next(t))
print(next(t))
print(next(t))
```

```
Run    exam
D:\python_sabz
3
6
9

Process finish
```

- دقت شود اگر از تبدیل iter(t) استفاده کنیم در آن صورت next روی __iter__ اعمال میشود. پس برای باید این دو متد هماهنگ باشند برای این کار کد زیر را در نشر بگیریم :

```
class Pow2:  1 usage  new *
    def __init__(self,maxPow):  new *
        self.maxPow = maxPow
        self.pow = 0

    def __iter__(self):  new *
        return self

    def __next__(self):  new *
        if self.pow < self.maxPow:
            self.pow += 1
            return self.pow**2
        else:
            raise StopIteration

p = Pow2(5)
for i in p:
    print(i)
```

```
Run    exa
D:\pyt
1
4
9
16
25

Process
```

- Decorator

میتوانیم روی کلاس‌ها دکوریتر اعمال کنیم :

```python
from functools import wraps
def Test_Decorator(cls):  1 usage  new *
    @wraps(cls)  new *
    def wrapper(*args, **kwargs):
        print('-'*20)
        cls(*args, **kwargs)
        print('-'*20)
        return cls(*args, **kwargs)

    return wrapper

@Test_Decorator  1 usage  new *
class Test:
    def __init__(self, name, age):  new *
        self.name = name
        self.age = age

obj = Test( name: 'ali', age: 54)
print(obj.name)
```

```
Run    exam
D:\python_sabzlern\1\.venv\
--------------------
--------------------
ali

Process finished with exit
```

علاوه بر این دکوریتور میتواند به صورت کلاس باشد :

دقت شود که برای مستند ها از update_wrapper استفاده کردیم

```python
from functools import update_wrapper
class TestDecorator:  1 usage  new *

    def __init__(self, cls):  new *
        update_wrapper(self, cls)
        self.cls = cls
    def __call__(self, *args, **kwargs):  new *
        print("-"*20)
        print("*"*20)
        return self.cls(*args, **kwargs)

@TestDecorator  1 usage  new *
class Test:
    def __init__(self, name, age):  new *
        self.name = name
        self.age = age

obj = Test( *args: 'ali',54)
print(obj.name)
```

```
Run    exam
D:\python_sabzlern\1\.ven
--------------------
********************
ali

Process finished with exi
```

برای تابع هم به جای cls از func استفاده میکنیم

```
class MyClassDecorator:
    def __init__(self, func):
        update_wrapper(self, func)
        self.func = func

    def __call__(self):
        print("*" * 40)
        self.func()
        print("-" * 40)


@MyClassDecorator
def my_func():
    """MY Func"""
    print("reza")
    print(1 + 2)
```

- Descriptor

فرض کنیم که برای یک سری attr ها نیاز به اعتبار سنجی‌های شبیه هم هست و با property باعث میشود که تکرار کد داشته باشیم

برای این که از تکرار جلوگیری کنیم از کلاس های Descriptor استفاده میکنیم

```
class NameFiled:  3 usages  new *
    def __init__(self, name=None):  new *
        self.name = name

    def __get__(self, instance, owner):  new *
        return self.name

    def __set__(self, instance, value):  new *
        if 0<len(value)<15:
            self.name = value
        else:
            raise ValueError("value must be between 0 and 15")
class Person:  1 usage  new *
    name = NameFiled()
    father_name = NameFiled()
    mother_name = NameFiled()
    def __init__(self,name,father_name,mother_name):  new *
        self.name = name
        self.father_name = father_name
        self.mother_name = mother_name

p = Person( name: 'ali', father_name: 'reza', mother_name: 'neda')
print(p.name)
print(p.father_name)
print(p.mother_name)
```

- مشکلی که این کد دارد این است که اگر p dict را پرینت کنیم در آن صورت متوجه میشویم که خالی است و attr برای آن مشخص نشده است

```
p = Person( name: 'ali', father_name: 'reza', mother_name: 'neda')
print(p.name)
print(p.father_name)
print(p.mother_name)
print(p.__dict__)
```

```
D:\python_sabzle
ali
reza
neda
{}
```

- راه حل چیست ؟ استفاده از dict , instance , set_name

```
class NameFiled:  3 usages  new *
    def __set_name__(self, owner, name):  new *
        self.name = name

    def __set__(self, instance, value):  new *
        if 0<len(value)<10:
            instance.__dict__[self.name] = value
        else:
            raise TypeError("user is not a User")
    def __get__(self, instance, owner):  new *
        return instance.__dict__[self.name]

    def __delete__(self, instance):  new *
        del instance.__dict__[self.name]


class Person:  1 usage  new *
    name = NameFiled()
    father_name = NameFiled()
    mother_name = NameFiled()
    def __init__(self,name,father_name,mother_name):  new *
        self.name = name
        self.father_name = father_name
        self.mother_name = mother_name
```

- Context manager

برای اینکه یک سری کار ها قبل و بعد از یک سری دستورات انجام شود استفاده میکنیم

مثالی که قبلا دیدیم :

```python
class context:  1 usage  new *

    def __enter__(self):  new *
        print("start!!!")
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        print("exit!!!")

c = context()
with c as c:
    print("Hossein")
```

```
Run    exam
D:\python_sabzle
start!!!
Hossein
exit!!!

Process finishe
```

پارمتر های exit اولی نوع خطا ، دومی مقدار خطا و اخری traceback

```python
def __exit__(self, exc_type, exc_val, exc_tb):  new *
    print(exc_type,'\n', exc_val,'\n', exc_tb,'\n')
    print("exit!!!")
```

```
start!!!
<class 'ZeroDivisionError'>
 division by zero
 <traceback object at 0x0000020D7C27FAC0>
```

در هر صورت اگر به خطا هم بخوریم exit اجرا میشود

برای اینکه خطا ignore شود میتوانیم از تکنیک زیر استفاده کنیم که در آن صورت ادامه کد ها اجرا نمیشود :

```python
6
7       def __exit__(self, exc_type, exc_val, exc_tb):
8           print("exit!!!")
9           return True
10
11  c = context()
12  with c as c:
13      print(1)
14      print(10/0)
15      print(2)
16
```

```
Run    exam
D:\python_sabzlern\1\.venv\Scripts\pyt
start!!!
1
exit!!!

Process finished with exit code 0
```

- مثالی از کنترل فایل

```python
class FileManager:
    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode
        self.file = None

    def __enter__(self):
        print("The file was opened!")
        self.file = open(self.filename, self.mode)
        return self.file

    def __exit__(self, exc_type, exc_val, exc_tb):
        print("The file was closed!")
        self.file.close()
```

- Data class

از پایتون ۳/۷ اضافه شده است

در آن باید حتما نوع داده‌ها معلوم باشد

```python
from dataclasses import dataclass

@dataclass  new *
class Student:
    name: str
    age: int
```

در آن میتوانیم __post_inti__ قرار دهیم و یک سری شرط ها بذاریم

```
class Student:
    name: str
    age: int
    def __post_init__(self):  new *
        if self.age < 0:
            self.age = 0

St = Student("John", -18)
print(St)
```

```
Run    exam ×
D:\python_sabzlern\1\.venv\Scrip
Student(name='John', age=0)
```

میتوانیم مقدار دهی اولیه هم بکنیم که باید از اول به آخر باشد

فصل یازدهم: مدیریت خطا

قبلا با خطاها آشنا شدیم

```
while True:
    try:
        enter = int(input("Enter a number: "))
        print(enter)
        break

    except ValueError:
        print("Invalid input")
```

- میتوانیم انواع خطاها داشته باشیم و برای مدیریت آن ها میتوانیم except های متوالی نوشت و در آخر یک except default :

```
try:
    x = int(input("Enter a number: "))
    y = int(input("Enter another number: "))
    x = x / y + "error"
except ZeroDivisionError:
    print("You can't divide by zero")
except ValueError:
    print("You didn't enter a number")
except:
    print("Something went wrong")
```

- اگر مدیریت دو خطا شبیه هم بود میتوانیم به صورت زیر عمل کنیم :

```
try:
    x = int(input("Enter a number: "))
    y = int(input("Enter another number: "))
    x = x / y + "error"
except (ZeroDivisionError, ValueError):
    print("You can't divide by zero")
# except ValueError:
#     print("You didn't enter a number")
except:
    print("Something went wrong")
```

- میتوانیم شی خطا را بگیریم :

```
except ZeroDivisionError as e:
    print(e.__class__.__name__)
```

- همه خطاها از کلاس پدر Exception ارث بری میکنند و میتوانیم از آن استفاده کنیم :

```
try:
    x = int(input("Enter a number: "))
    y = int(input("Enter another number:
    x = x / y + "error"
except Exception as e:
    print(e)
```

```
D:\python_sabzlern\1\.venv\Sc
Enter a number: 5
Enter another number: 0
division by zero
```

- Else

زمانی اجرا میشود که کله بدنه try بدونه خطا اجر شود. بهتر است قسمت هایی که احتمال خطا میدهیم در try و بقیه را در else نوشت

```
try:
    x = int(input("Enter a number: "))
    y = int(input("Enter another number: "))
    x = x / y
except Exception as e:
    print(e)
else:
    print(x)
```

اگر return داشته باشیم در آن صورت else اجرا نمیشود

- Finally

در هر صورت چه با وجود خطا چه بدون وجود خطا و چه با return اجرا میشود

در صورت دوتا return، از finally برگردانده میشود

- میتوانیم به صورت تو در تو مدیریت استثنا داشته باشیم :

```
try:
    x = int(input("Enter a number: "))
    y = int(input("Enter another number: "))
    x = x / y
except Exception as e:
    try:
        print(int(wefwef))
    except Exception as e:
        print(e)
else:
    print(x)
```

```
Run    exam
D:\python_sabzlern\1\.venv\Scripts
Enter a number: 5
Enter another number: dsf
name 'wefwef' is not defined

Process finished with exit code 0
```

- میتوانیم کلاس های دلخواه برای error های خود بنویسیم

```python
class DigitError(Exception):  new *
    def __init__(self,message):  new *
        self.message = message
        super().__init__(self.message)
    def __str__(self):  new *
        return "digit error"
```

- Warning

برای نوشتن آن میتوانیم به صورت زیر عمل کنیم :

```python
from warnings import warn


def plus(x, y):  1 usage  new *
    if not isinstance(x, int) or not isinstance(x, int):
        warn("x and y should be int")
    print(x + y)

plus( x: "1", y: "2")
```

```
Run    exam
D:\python_sabzlern\1\.venv\Scripts\python.exe D:\python_sabzlern\1\exam.
D:\python_sabzlern\1\exam.py:6: UserWarning: x and y should be int
  warn("x and y should be int")
12
```

- Assert

معمولا برای دیباگ استفاده میشود اگر زمانی مطمئن بودیم اتفاقی میافتد و نیفتاد از برای ما خطا میدهد :

```python
def plus(x, y):  1 usage  new *
    assert  isinstance(x, int) or  isinstance(x, int)
    print(x + y)

plus( x: "1", y: "2")
```

```
D:\python_sabzlern\1\.venv\Scripts\python.ex
Traceback (most recent call last):
  File "D:\python_sabzlern\1\exam.py", line
    plus("1","2")
    ~~~~^^^^^^^^^
  File "D:\python_sabzlern\1\exam.py", line
    assert  isinstance(x, int) or  isinstanc
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError
```

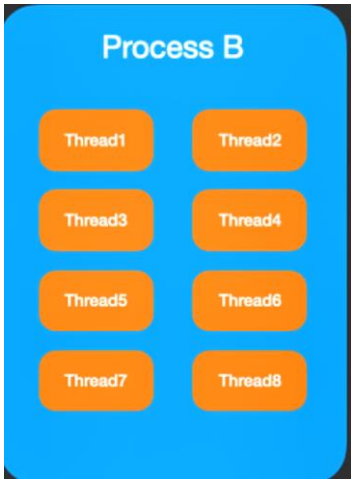
فصل دوازدهم: همزمانی

۱. Process

یک "process" (فرآیند) به یک نمونه از یک برنامه در حال اجرا اشاره دارد. هر فرآیند دارای فضای حافظه خاص خود است و می‌تواند به صورت مستقل از سایر فرآیندها اجرا شود. در پایتون، می‌توان از ماژول multiprocessing برای ایجاد و مدیریت فرآیندها استفاده کرد. این روش برای بهره‌برداری از چند هسته‌ای بودن CPU مناسب است.

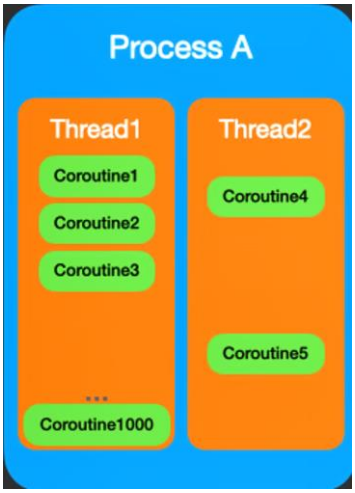
۲. Thread

یک "thread" (رشته) یک واحد اجرایی در یک فرآیند است. چندین رشته می‌توانند در یک فرآیند اجرا شوند و به اشتراک‌گذاری حافظه بپردازند. در پایتون، می‌توان از ماژول threading برای ایجاد و مدیریت رشته‌ها استفاده کرد. اما به دلیل Global Interpreter Lock (GIL)، رشته‌ها در پایتون معمولاً برای I/O-bound tasks مناسب‌تر هستند تا CPU-bound tasks.



۳. Coroutine

"Coroutine" (همکار) نوع خاصی از تابع است که می‌تواند در وسط اجرای خود متوقف شده و بعداً از همان نقطه ادامه یابد. این مفهوم به ویژه در برنامه‌نویسی غیرهمزمان (asynchronous programming) کاربرد دارد. در پایتون، می‌توان از کلمه کلیدی async برای تعریف coroutineها و از await برای صبر کردن بر روی نتیجه‌ی یک coroutine دیگر استفاده کرد.



- Multiprocessing

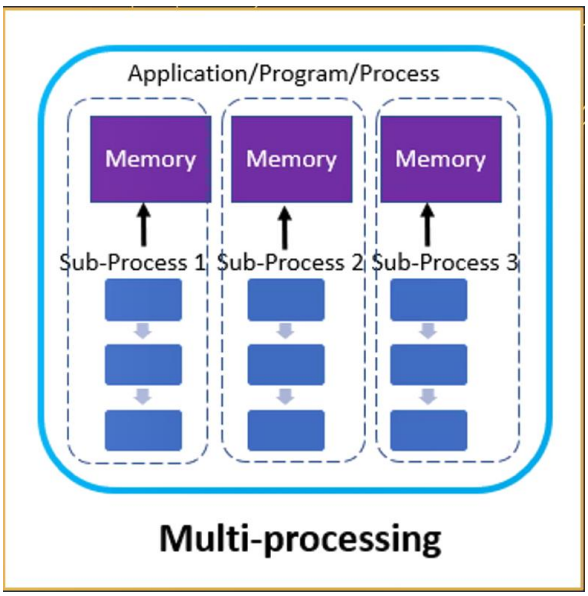
Multiprocessing به استفاده از چندین فرآیند مستقل برای اجرای همزمان برنامه اشاره دارد. هر فرآیند دارای فضای حافظه جداگانه‌ای است و می‌تواند به طور مستقل از سایر فرآیندها اجرا شود. این روش به ویژه برای برنامه‌هایی که نیاز به پردازش‌های سنگین دارند و می‌خواهند از تمام هسته‌های CPU استفاده کنند، مناسب است.

مزایا:

- قابلیت استفاده از چند هسته CPU.
- ایزوله بودن فرآیندها، که باعث می‌شود در صورت بروز خطا در یک فرآیند، سایر فرآیندها تحت تأثیر قرار نگیرند.

معایب:

- هزینه بالای ایجاد و مدیریت فرآیندها.
- ارتباط بین فرآیندها معمولاً پیچیده‌تر از رشته‌ها است.



- Multithreading

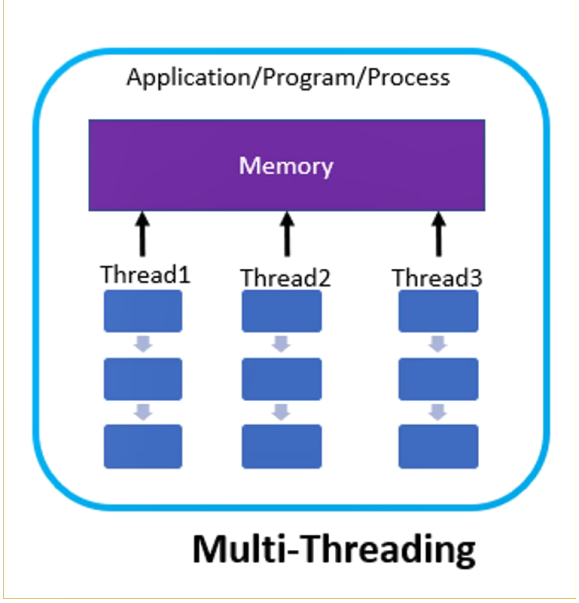
اشتراک‌گذاری حافظه بپردازند و معمولاً برای کارهای I/O-bound مناسب‌تر هستند. در پایتون، ماژول threading برای مدیریت رشته‌ها استفاده می‌شود.

مزایا:

- هزینه پایین‌تر نسبت به multiprocessing.
- ارتباط ساده‌تر بین رشته‌ها، زیرا آن‌ها فضای حافظه مشترکی دارند.

معایب:

- وجود Global Interpreter Lock (GIL) در پایتون که مانع از اجرای همزمان کد پایتون در چندین هسته می‌شود. بنابراین، برای کارهای CPU-bound ممکن است کارایی کمتری داشته باشد.



- Asynchronous Programming

Asynchronous programming (برنامه‌نویسی غیرهمزمان) یک الگوی برنامه‌نویسی است که به شما اجازه می‌دهد تا در هنگام انتظار برای I/O، کنترل را به سایر بخش‌های برنامه بدهید. در پایتون، با استفاده از کلمات کلیدی async و await، می‌توانید coroutineها را تعریف کنید.

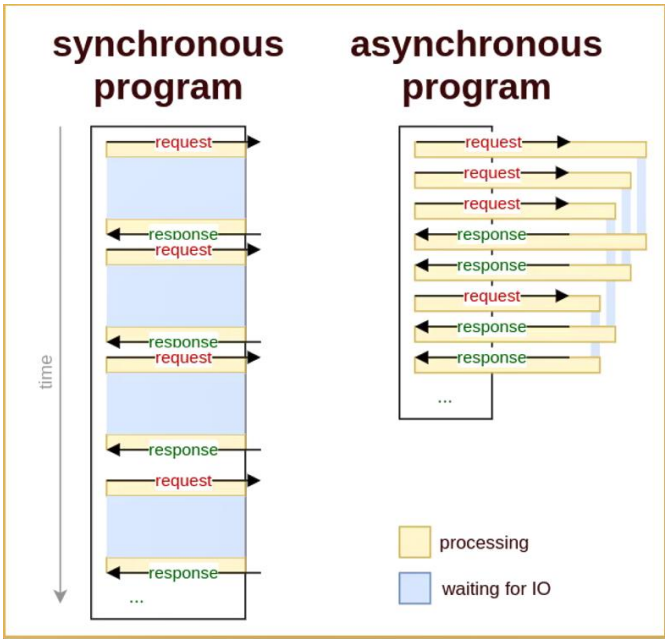
مزایا:

- کارایی بالا در کارهای I/O-bound.
- استفاده بهینه از منابع سیستم، زیرا در زمان انتظار برای I/O، CPU آزاد است.

معایب:

• پیچیدگی بیشتر در کد و نیاز به درک عمیق‌تر از مفهوم coroutine.

• ممکن است برای کارهای CPU-bound مناسب نباشد.



نتیجه‌گیری

انتخاب بین multiprocessing، multithreading و asynchronous programming بستگی به نوع وظایفی دارد که می‌خواهید انجام دهید:

• اگر با کارهای سنگین CPU مواجه هستید، multiprocessing گزینه مناسبی است.

• اگر با کارهای I/O-bound سروکار دارید و به سادگی نیاز دارید که همزمان چند کار را انجام دهید، multithreading یا asynchronous programming می‌تواند مفید باشد.

• برای برنامه‌های پیچیده با نیاز به مقیاس‌پذیری بالا و کارایی در I/O، asynchronous programming معمولاً بهترین انتخاب است.

- اولین برنامه multi threading

join برنامه را تا زمان اجرا thread ها متوقف میکند

```python
import threading
from time import sleep

print("start")

def multi_thread(i):  1 usage  new *
    print(f"multi thread:{i}")
    sleep(3)

threads = [threading.Thread(target=multi_thread, args=(i,)) for i in range(10)]
for t in threads:
    t.start()
for t in threads:
    t.join()
print("end")
```

```
Run    exam
D:\python_sabzlern\1\
start
multi thread:0
multi thread:1
multi thread:2
multi thread:3
multi thread:4
multi thread:5
multi thread:6
multi thread:7
multi thread:8
multi thread:9
end

Process finished with
```

- هندل کردن حافظه مشترک :

به دلیل آنکه از حافظه مشترک استفاده میشود ممکن است مشکل پیش آید و برای جلوگیری از این از lock استفاده میشود

```python
import threading

same_memory = 0
lock = threading.Lock()
def func():  1 usage  new *
    for i in range(1_000_000):
        global same_memory
        with lock:
            same_memory += 1

threads = []
for _ in range(3):
    t = threading.Thread(target=func)
    threads.append(t)

for t in threads:
    t.start()
for t in threads:
    t.join()

print(same_memory)
```

فصل سیزدهم: مباحث تکمیلی

- Union , pipe

اگر متغیر ما دو نوع داده میتوانست باشد میتوانیم از Union استفاده کنیم

```
from typing import Union

x:Union[int,str] = 5
print(isinstance(x,int))
x = "4"
print(isinstance(x,str))
```

```
Run    exam
D:\python_
True
True
```

میتوانیم از pipe استفاده کنیم

```
x:int|str = 5
print(isinstance(x,int))
x = "4"
print(isinstance(x,str))
```

```
D:\python_sabzle
True
True
```

- Match case

مانند همان switch case است و حال دقیق تر بررسی میکنیم :

مثال ماشین حساب :

در اخر برای آن یک حالت دیفالت نوشته شده است :

```python
x = int(input("Enter a number: "))
y = int(input("Enter another number: "))
op = (input("Enter operation: "))

match op:
    case "+":
        print("{} + {} = {}".format( *args: x,y,x+y))
    case "-":
        print("{} - {} = {}".format( *args: x,y,x-y))
    case "*":
        print("{} * {} = {}".format( *args: x,y,x*y))
    case "/":
        print("{} / {} = {}".format( *args: x,y,x/y))
    case _:
        print("errorD")
```

```
Run    exam
D:\python_sabzlern\1\.venv\Scripts\
Enter a number: 5
Enter another number: 4
Enter operation: /
5 / 4 = 1.25

Process finished with exit code 0
```

- میتوانیم به آن تاپل دهیم و همچنین در آن شرط بنویسیم برای مثال یک بازی با دو تاس داریم :

```python
x = int(input("Enter a number: "))
y = int(input("Enter another number: "))

match (x,y):
    case (6,6) if 0<x<7 and 0<y<7:
        print("win")
    case (1,1) if 0<x<7 and 0<y<7:
        print("lose")
    case (x,6) if 0<x<7 and 0<y<7:
        print("repeat")
    case (6,y) if 0<x<7 and 0<y<7:
        print("repeat")
    case  (x,y) if 0<x<7 and 0<y<7:
        print("next")
    case _ :
        print("error")
```

- برای دیکشنری :

```
match {"x":x, "y":y}:
    case {"x":6,"y":6} if 0<x<7 and 0<y<7:
        print("win")
    case {"x":1,"y":1} if 0<x<7 and 0<y<7:
        print("lose")
    case {"x":x,"y":6} if 0<x<7 and 0<y<7:
        print("repeat")
    case {"x":6,"y":y} if 0<x<7 and 0<y<7:
        print("repeat")
    case  {"x":x,"y":y} if 0<x<7 and 0<y<7:
        print("next")
    case _ :
        print("error")
```

اگر مقداری برای ما مهم نبود در تاپل از * و در دیکشنری از ** استفاده میکنیم

```
    print("repeat")
case {"x":6,**others} if 0<x<7 and 0<y<7:
    print("repeat")
```

- میتوانیم با as آن را بگیریم :

```
case {"x":6,"y":6} as a if 0<x<7 and 0<y<7:
    print(a)
case {"x":1,"y":1} if 0<x<7 and 0<y<7:
```

```
Enter a number: 6
Enter another number: 6
{'x': 6, 'y': 6}
```

منبع

https://sabzlearn.ir/course/python/