Python 3.6

شهـــــروز صفـــــری

مقدمه

زبان برنامه نویسی Python یک زبان چند منظوره‌ی سطح بالاست که در سال ۱۹۸۹ توسط Guido van Rossum طراحی و در سال ۱۹۹۱ معرفی شد .

اهداف طراحی Python

- محاسبات ریاضی Mathematics
- توسعه نرم افزار Software Development
- توسعه برنامه های تحت وب Web Development
- برنامه نویسی سیستمی System Scripting

زبانهای برنامه نویسی موجود از دو روش برای تبدیل زبان سطح بالا (زبانی که نزدیک به زبان محاوره ای است) به زبان ماشین (صفر و یک) استفاده می کنند

- Compiler (مترجم) : تبدیل کل Source Code به یک فایل باینری و سپس اجرای آن
- Interpreter (مفسر) : اجرای خط به خط برنامه تا رسیدن به انتها

زبانهای خانواده‌ی C (مانند C++ و C# و . . .) زبان Java و Xamarin و . . . از روش Compile استفاده می کنند و زبان های Python و Jscript وPHP و Ruby و . . . از روش Interpret عمل می کنند .

هر کدام از این دو روش معایب و مزایای خود را دارند .

مزیت های Python

- Cross-Platform : کدهای مستقل از سیستم عامل هستند
- Simple Readability : خوانایی منبع کدها
- Fewer Lines of Code : کدهای خلاصه تر
- Quick Prototyping : نمونه سازی سریع تر و ساده تر
- Procedural Programing : برنامه نویسی فرآیند گرا
- OOP Programing : برنامه نویسی شیئ گرا

نصب و راه اندازی IDE

هر زبان برنامه نویسی علاوه بر نصب Compiler یا Interpreter روی سیستم عامل نیاز به یک محیط توسعه یکپارچه (Integrated Development Environment) دارد .

بعضی از کاربردی ترین IDE های زبان Python در ادامه آمده‌اند :

- Python IDLE (Python .org)
- PyCharm (Jetbrains.com)
- Jupyter Notebook (Jupyter.org)
- VS Code (Microsoft.com)
- repl.it (Online IDE)
- Anaconda (Scientific Computing Distribution of Python)

تعریف متغیرها و نوع داده‌ها در Python

تعریف متغیر Variable Declaration

اعلان متغیر در Python اجباری نیست یعنی برای استفاده از متغیرها نیازی به تعریف آنها در خطوط قبلی کد نیست و هر متغیر با نسبت دادن یک مقدار به آن تعریف شده و نوع آن مشخص می شود .

نام متغیر می تواند شامل :

- حروف الفبا Letters (A-Z و a-z)
- اعداد Numbers
- خط زیر Underscore

باشد .

❋ نکته : حرف اول نمی‌تواند عدد باشد

❋ نکته : حروف کوچک و بزرگ (Caps) در Python متفاوت هستند .

❋ نکته : نام متغیر نمی‌تواند یکی از کلمات کلیدی یا رزرو شده باشد

نوع داده ها (Data Type) در Python

- نوع داده‌های عددی Numeric
- int (Integer) : اعداد صحیح با بازه تقریبا نامحدود
- float : اعداد اعشاری
- complex : اعداد موهومی (که به صورت a+bj نمایش داده می شوند)
- نوع داده‌ی متنی یا رشته‌ای
- str (String) : برای انتساب متن از " (Double Quotation) یا ' (Single Quotation) استفاده می کنیم .

- نوع داده دوحالته (Boolean) : این نوع فقط شامل True یا False می شود .

(توجه داشته باشید T و F حروف بزرگ هستند)

* نکته : بعضی از نوع داده‌ها قابل تبدیل به نوع داده‌های دیگر هستند که توسط توابع تبدیل انجام می شود و به این عملیات Casting می گوئیم .

int() float() str() bool()

bin() hex() oct()

مثال :

int(2.8) → 2

int("4") → 4

float(2) → 2.0

hex(15) →0xf

عملگرها در Python

عملگرها به هفت دسته تقسیم میشوند :

دسته اول : عملگرهای محاسباتی (ریاضی) Arithmetic Operators

+	-	*	/	**	%	//
Add	Subtract	Multiply	Divide	Power	Modulus	Floor Divide

دسته دوم : عملگرهای مقایسه‌ای Comparison Operators

>	<	>=	<=	==	!=

دسته سوم : عملگرهای منطقی Logical Operators

and	or	not

دسته چهارم : عملگرهای انتساب — Assignment Operators

=	+=	-=	*=	/=	%=
//=	**=	&=	|=	^=	
>>=	<<=				

دسته پنجم : عملگرهای عضویت — Membership Operators

in	not in

دسته ششم : عملگرهای همسانی (تطابق) — Identity Operators

is	is not

دسته هفتم : عملگرهای دستکاری بیت‌ها — Bitwise Operators

&	|	^	~	>>	<<
And	Or	Xor	Not	Right Shift	Left Shift
				Signed Fill	Zero Fill

چند علامت پر کاربرد در Python

( ) : Parentheses or Round Bracket

[ ] : Bracket or Square Bracket

{ } : Braces or Curly Bracket

< > : Angle Bracket

\ : Backslash (برای نوشتن ادامه یک دستور در چند سطر)

; : Semicolon (برای نوشتن چند دستور در یک سطر)

UPPERCASE : نوشتن یک یا چند حرف به صورت حروف بزرگ

lowercase : نوشتن یک یا چند حرف به صورت حروف کوچک

camelCase : نوشتن کلمات بدون فاصله و با ترکیب حروف کوچک و بزرگ

Capitalize : بزرگ نوشتن حرف اول

مجموعه‌های تکرار پذیر Iterable Collections

لیست ها Lists

لیست شامل مجموعه ای از داده‌ها است که به صورت یک آرایه در کنار هم قرار می گیرند .

لیست ها دارای ترتیب هستند و قابلیت تغییر دارند و هر عضو یک شاخص دارد .

Ordered – Mutable –Indexable

نحوه تعریف List :

List = [Item1,Item2,…]

مثال :

L1= [1, 2, 3, 4, 5]

L2= [2, 3,"gfdfghdfg", 4.5]

L3= ['Reza','Pooya','Sara']

❋ نکته : شاخص داده اول صفر است یعنی L3[0] برابر Reza است .

❋ نکته : اعضای لیست قابل تغییر هستند L2[2]="Payam"

❋ نکته : امکان دسترسی به چند عضو فراهم است L1[1:3] (List Slicing)

List [a:b:c]

a : شروع b: پایان c : گام

متدهای List

متد	عملکرد	متد	عملکرد
List.append(item)	اضافه کردن یک عضو به انتهای لیست	List1.extend(List2)	ادغام دو لیست
List.insert(index,item)	اضافه کردن یک عضو در محل دلخواه	List.count(item)	تعداد یک عنصر
List.pop(index)	حذف یک عضو با index	List.copy()	ایجاد نسخه جدید از لیست
List.remove(item)	حذف یک عضو با مقدار	List.reverse()	معکوس کردن عناصر لیست
List.clear()	حذف تمام اعضا	List.sort()	مرتب سازی صعودی
List.index(item,start,end)	جستجوی یک عنصر	List.sort(reverse=True)	مرتب سازی نزولی

چندتایی‌ها Tuples

چندتایی مشابه لیست است با این تفاوت که بعد از ایجاد شدن قابلیت تغییر ندارند

Ordered – Immutable – Indexable

نحوه تعریف Tuple :

Tuple = (Item1,Item2,…)

مثال :

T1= (1, 2, 3, 4, 5)

T2= (1, 2,'Reza')

❋ نکته : برای دسترسی به اعضای Tuple از [ ] استفاده می کنیم

T1[0]=1

❋ نکته : چندتایی امکان اضافه کردن مستقیم اعضا را ندارد ولی با تعریف مجدد می توانیم عضو اضافه کنیم

T1=T1+('Ali',)

کاما در آخر ضروری است .

متدهای Tuple

متد	عملکرد	متد	عملکرد
Tuple.count(item)	تعداد یک عنصر	Tuple.index(item,start,end)	جستجوی یک عنصر

مجموعه ها Sets

مجموعه ها هم جزوی از تکرار شونده ها هستند با این تفاوت که اعضا Index ندارند و صورت Random جابجا می شوند

Unordered – Mutable - Unindexable - Unique

نحوه تعریف Set :

Set = {Item1,Item2,…}

مثال :

S1 = {1,2,3,4,5}

S2= {'Reza','Pooya','Ai'}

❋ نکته : در Set امکان تکراری بودن اعضا وجود ندارد برای همین در آمارگیری و شمارش کاربرد دارند

❋ نکته : دسترسی به اعضا با Index امکان پذیر نیست S1[3] = error

❋ نکته : تغییر یک عضو امکان پذیر نیست ولی حذف و اضافه کردن عضو امکان پذیر هست

متدهای Set

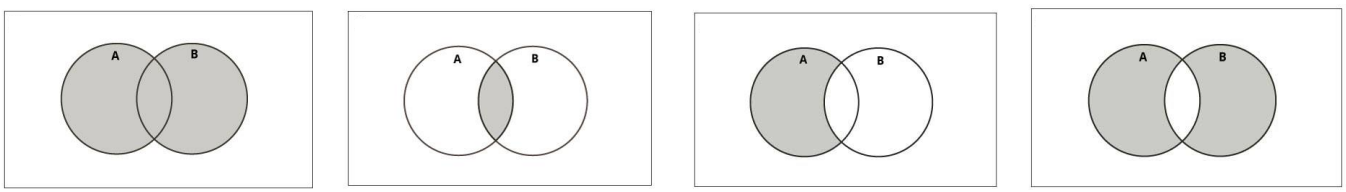
متد	عملکرد	متد	عملکرد
Set.add(item)	اضافه کردن یک عضو به انتهای set	Set.clear()	حذف تمام اعضا
Set.pop()	حذف یک عضو	Set1.update(Set2)	ادغام دو set
Set.remove(item)	حذف یک عضو با مقدار	Set.copy()	ایجاد نسخه جدید از set
Set.discard(item)	حذف یک عضو با مقدار ، در صورت عدم وجود عنصر خطا اتفاق نمی‌افتد		

❋ نکته : برای جستجوی یک عنصر در set متدی وجود ندارد و برای اینکار از عملگر in استفاده می‌کنیم .

item in set → True or False

علاوه بر متدهای بالا امکان ایجاد اجتماع و اشتراک در Set ها وجود دارد :

متد	عملکرد
Uni=Set1.union(Set2)	اجتماع دو مجموعه
Inter=Set1.intersection(Set2)	اشتراک دو مجموعه
Dif=Set1.difference(Set2)	تفاضل دو مجموعه
Dif=Set1. symmetric_difference(Set2)	تفاضل متقارن



واژه نامه‌ها Dictionaries

دیکشنری یکی دیگر از مجموعه های تکرار پذیر است که شامل دوتایی های کلید و مقدار می باشد که با : جدا می شوند

Key: Value

Unordered – Mutable – Indexable – Unique

نحوه تعریف دیکشنری :

Dict = {key1:value1,key2:value2, . . .}

مثال :

D={'Name':'Reza','Family':'Ahmadi','Age':25}

❋ نکته : کلیدها می توانند متنی یا عددی باشند ولی نباید تکراری باشند

❋ نکته : برای دسترسی به اعضا از نام کلید به همراه [ ] استفاده می کنیم

D['Family'] D.get('Family')

❋ نکته : برای اضافه کردن عضو جدید باید یک کلید و مقدار آن را تعریف کنیم

D['Code']=12856

متدهای Dictionary

متد	عملکرد	متد	عملکرد
Dict.get(key)	مقدار متناظر کلید	Dict.pop(key)	حذف یک کلید و مقدار آن
Dict.keys()	لیست کلیدها	Dict.popitem()	حذف آخرین کلید و مقدار آن و برگشت به صورت tuple
Dict.values()	لیست مقادیر	Dict.clear()	حذف تمام اعضا
Dict.items()	تبدیل dict به لیستی از tuple ها	Dict1.update(Dict2)	ادغام دو dict
Dict. setdefault(key,value)	اگر کلید وجود داشته باشد مقدار آنرا برمی‌گرداند در غیر اینصورت کلید : مقدار جدید ایجاد میکند		

رشته های متنی Strings

متغیرهای رشته ای نوعی Iterable هستند که می توان در بین کاراکترهای آن حرکت کرد

s="Hello, World"

s[0] → H

s[2:5] → llo

s[::-1] → dlroW ,olleH

متدهای string

متد	عملکرد	متد	عملکرد
s.split("string")	جداسازی متن با توجه به یک یا چند حرف	s.count("string")	شمارش تعداد تکرار
s.strip("string")	حذف یک یا چند حرف از ابتدا و انتهای متن	s.find("string",start,end)	جستجوی یک متن و برگشت محل اتفاق آن
s.lower()	تبدیل به حروف کوچک	s.index("string",start,end)	جستجوی یک متن و برگشت محل اتفاق آن
s.upper()	تبدیل به حروف بزرگ	s.replace("old","new",count)	جایگزین کردن یک متن با متن دیگر
s. capitalize()	بزرگ نوشتن حرف اول	s.startswith("string",start,end")	چک کردن شروع یک متن
s.title()	بزرگ نوشتن اول کلمات	s.endswith("string",start,end)	چک کردن انتهای یک متن
s.isnumeric()	True اگر کل متن عددی باشد	s.zfill(number)	پر کردن صفر پیش از رقم
s.isalpha()	True اگر کل متن حروف الفبا باشد	s.join(iterable)	اعضای iterable را با متن مشخص شده ترکیب می‌کند
s.isalnum()	True اگر متن فقط شامل عدد و حرف باشد	s.splitlines()	قراردادن سطرها در یک لیست (/n)

ساختارهای تصمیم Decision Control Flow

دستور if تک خطی

```
if شرط : Expr1
else: Expr2
```

مثال

```
a=5
if a>=8 : b=2
else: b=3
```

دستور if چند خطی

```
if شرط :
    Expr1
    Expr2
    …
else :
    Expr1
    Expr2
    …
```

دستور if تودرتو (Nested if)

```
if شرط :
    Expr1
    Expr2
    …
elif شرط :
    …
```

```
elif شرط :
        …
else :
        …
```

مثال

```
a=100
if a==10 :
        b=a+1
elif a==50 :
        b=a+2
elif a==90 :
        b=a+3
else :
        b=100
```

ساختارهای تکرار Loop Control Flow

حلقه for

for شمارنده in تکرار پذیر (Iterable) :

    Expr1
    Expr2
    …
    break / continue
    …
else:
    Expr1
    …

❋ نکته : شمارنده حلقه یک متغیر است که نیاری به اعلان ندارد

❋ نکته : تکرار پذیر می تواند از انواع List یا Tuple یا String یا سایر Iterable های Python باشد.

❋ نکته : دستور else در انتهای حلقه اجرا می‌شود مگر اینکه حلقه break شود

❋ نکته : برای ایجاد محدوده اعداد از تابع Range استفاده می کنیم

range(start,end,step)

range(50)

range(1,100)

range(1,500,2)

حلقه While

```
while شرط :
        Expr1
        Expr2
        …
        break / continue
        …
else:
        Expr1
        …
```

❋ نکته : در بدنه While باید دستوراتی نوشته شوند که شرط False شود در غیر اینصورت حلقه بی نهایت بار تکرار خواهد شد .

❋ نکته : دستور else در انتهای حلقه تکرار می‌شود مگر اینکه حلقه break شود .

مثال

```
a=1
while a<=20:
        print(a)
        a+=1
```

تکمیل لیست و مجموعه و دیکشنری List , Set , Dict Comprehension

با ترکیب حلقه for و شرط if می توانیم یک List یا Set یا Dict جدید بسازیم که با تغییرات روی منبع اصلی ایجاد می شود .

در Python به این عملیات List Comprehension یا Set Comprehension یا Dict Comprehension می گوئیم .

در ادامه مثالهایی از هر کدام آورده می شود

numbers=[1,2,3,4,5,6]

double_numbers= [n*2 for n in numbers] → [2,4,6,8,10,12]

format=[f ' numbers are {n} . ' for n in numbers] → ['numbers are 1 .',numbers are 2. ' , …]

❋ نکته : در ادامه و در قسمت Built-in Functions در مورد f strings توضیح خواهیم داد .

even=[n for n in numbers if n%2==0] → [2,4,6]

uni=[n for n in numbers if n in double_numbers] → [2,4,6]

همه‌ی مثال های بالا در مورد Set هم قابل اجراست

❋ نکته : تبدیل دو لیست به یک دیکشنری

students=['Ali','Reza','Sara','Maryam']

numbers=[18,17,19,15]

std_num={studentds[i]:numbers[i] for i in range(0,4)}

→ {'Ali':18,'Reza':17,'Sara':19,'Maryam':15}

❋ نکته : برای ترکیب دو List یا Tuple یا Set می توانیم از تابع zip استفاده کنیم .

این تابع اعضا را به صورت نظیر به نظیر تبدیل به Tuple می کند .

Std_num=dict(zip(students,numbers))

درج توضیح Comment

درج توضیح در زبانهای برنامه نویسی به دو دلیل انجام می شود

۱) مستند سازی Documentation

۲) افزایش خوانایی برنامه برای سایر استفاده کنندگان از برنامه و کدها

برای درج توضیح در Python دو روش وجود دارد :

۱) توضیح تک خطی Comment

برای درج توضیح تک خطی از علامت # استفاده می کنیم

۲) توضیح چند خطی Doc String

برای درج توضیح چند سطری از """ یا ''' استفاده می‌کنیم . (Triple Quotations)

Built-in Functions توابع استاندارد

توابع استاندارد یا توابع درونی مجموعه‌ای از توابع هستند که به صورت پایه در مفسر Python نصب شده‌اند و برای استفاده از آنها نیازی به Import کتابخانه نیست .

برای مشاهده لیست کامل توابع استاندارد می توانید به صفحه‌ی زیر مراجعه کنید

https://docs.python.org/3/library/functions.html

در ادامه چند تابع مهم را بررسی می کنیم

تابع Print

این تابع برای مشاهده مقدار متغیرها و متنها روی صفحه نمایش استفاده می شوند .

Print(value1,value2,…,sep="جدا کننده",end="متن انتهایی",file=open("آدرس و نام فایل","mode"),flush=True)

Value : مقادیری که در خروجی چاپ می شوند

Sep : یک متن دلخواه به عنوان جدا کننده بین مقادیر

end : متنی که در انتها چاپ خواهد شد

file : ارسال خروجی به یک فایل (فایل باید به تابع open ایجاد شود)

flush : مقدار True یا False برای تخلیه Buffer چاپ (پیش فرض = False)

مثال :

```
print("test")
print(5,7,9)          → 5 7 9          به صورت پیش فرض بین مقادیر یک فاصله قرار می گیرد
print(5,7,9,sep="-") → 5-7-9
```

❋ نکته : برای ایجاد سر از کاراکتر New Line از \n و برای پرش یک Tab به جلو از \t استفاده می کنیم

```
print("Enter your name :  \n ")
```

❋ نکته : برای مشاهده خروجی چند تابع print در یک سطر از آرگومان end="" استفاده می کنیم .

❋ نکته : در تابع print می توانیم از f string (مخفف Formatted String) استفاده کنیم .

در Python به ترکیب متن همراه با یک یا چند متغیر f string می گوئیم

```
Name="Reza"
Avg=20
print(f "The name is : {Name} and Average is : {Avg}")
```

❋ نکته : در f string می توانیم به تعداد کاراکتر دلخواه پرش ایجاد کنیم

```
print(f "The name is : {Name:5} and Average is : {Avg:5}")
```

در این حالت قبل از چاپ متغیر ۵ کاراکتر Space قرار می گیرد

برای پرش توابع rjust() و ljust() هم در دسترس هستند و برای قرار دادن صفرهای پیش از رقم از zfill(n) استفاده می کنیم .

```
print(f "The name is : {Name:5} and Average is : {Avg:5.2f}")
```

در این مثال متغیر Avg با دو رقم اعشار نمایش داده می شود .

{:d} → int {:f}→ float {:b} → binary

{:x} → hex {:o} → oct {;,} → 1000 Separator

تابع input

این تابع برای دریافت ورودی از صفحه کلید استفاده می شود

```
var=input(" Message ")
```

مقدار دریافت شده در متغیر var ذخیره می شود و باید توجه کرد که از نوع string است .

Message یک متن دلخواه است که قبل از دریافت ورودی نمایش داده می شود و درج آن اجباری نیست .۳

```
A=int(input('Enter a Number'))
```

تابع type

مشاهده نوع داده یک متغیر

a=5

print(type(a)) → class "int"

تابع len

مشاهده طول یک Iterable

تابع range

ایجاد یک محدوده از اعداد که به صورت یک Iterable برگشت می شود .

range(start,end,step)

❋ نکته : حد بالا در متغیر ذخیره نمی‌شود

توابع Casting

توابع casting به توابعی گفته می شود که برای تبدیل یک نوع متغیر به نوع دیگر استفاده می شوند

int()	float()	str()	complex()	bin()	hex()	oct()
list()	tuple()	set()	dict()			

تابع sum

sum(Iterable,init)

مثال

A=[4,6,8,7]

sum(A,20) →45

توابع min و max

max(Iterable)

min(Iterable)

تابع dir

مشاهده متدهای یک Object

a=[]

dir(a)

تابع id

مشاهده آدرس نگهداری متغیر در حافظه

تابع ord و chr

مشاهده کد اسکی یک حرف و مشاهده حرف متناظر با کد اسکی

ord('A') → 65

chr(65) → A

تابع zip

این تابع دو Iterable را به دوتایی های Tuple تبدیل می کند . تکرار شونده ها می توانند از انواع مختلف باشند ، مثلا یک List و یک Set و حتی نیاز نیست طول آنها برابر باشد .

a=[5,6,3,9]

b={1,2,3,4}

c=list(zip(a,b)) → [(5,1),(6,2),(3,3),(9,4)]

User-Defined Functions تعریف تابع

یک تابع مجموعه ای دستورات Python است که فرآیند مشخصی را پیاده سازی می کند

هر تابع سه قسمت دارد :

1) Return value 2) Function Name 3) Parameters

برای تعریف یک تابع از کلمه کلیدی def استفاده می کنیم

```
def Fname(param1,param2,…):
    Expr1
    Expr2
    …
    return return_value
```

مثال

```
def mul(x,y):
    m=x*y
    return m
```

❋ نکته : در هنگام فراخوانی یک تابع متغیرهای به جای پارامترها قرار می گیرند

در هنگام تعریف Parameters در هنگام فراخوانی Arguments

❋ نکته : تعداد پارامترها از نسخه 3.5 به بعد محدودیت ندارد

❋ نکته : پارامترها ۴ نوع هستند :

- اجباری Required یا Non-Default یا Positional
- اختیاری Optional یا Default
- تعداد متغیر Non-Keyworded Variable Length
- تعداد متغیر با فراخوانی نام Keyworded Variable Length

پارامترها به صورت پیش فرض اجباری هستند برای اختیاری شدن آرگومانها کافیست یک مقدار اولیه به آنها نسبت دهیم

```
def test(a,b,c=0)
```

* نکته : بعد از یک آرگومان اختیاری نمی‌توانیم آرگومانهای اجباری تعریف کنیم .

```
def test(a,b,c=" ",d)
```
→ خطا

آرگومانهای تعداد متغیر بدون نام

برای تعریف آرگومانهای با تعداد متغیر (تعداد پارامترها از ابتدا معلوم نیست) و بدون نام از * (Asterisk) استفاده می کنیم .

```
def func(*args)
```

در این حالت args به صورت یک Tuple به تابع ارسال می شود

```
def test(*a):
    for i in a:
        print(i**2)
----------------------------
test(2,4,7,6,9)
```

آرگومانهای تعداد متغیر با نام

آرگومانهای با تعداد متغیر و دارای نام با ** تعریف می شوند و در این حالت در هنگام فراخوانی تابع آرگومانها به صورت یک Dict به پارامتر تابع ارسال می شوند .

```
def func(**kwargs)
```

مثال :

```
def test(**a) ;
    for x in a.values():
        print(x**2)
------------------------------
test(a=5,b=6,c=7)
```

در هنگام استفاده از kwargs** متدهای values(). و keys(). در بدنه تابع پر استفاده خواهند بود تا بتوانیم کلیدها و مقادیر Dict را جداگانه پردازش کنیم .

در حالت کلی کلیدهایی که در هنگام فراخوانی برای تعریف مقادیر استفاده می شوند مهم نیستند و Dict با همان کلیدها ساخته می شود . برای اینکه کلیدها نامهای مشخصی داشته باشند باید در بدنه تابع کلیدها چک شوند

```
def test(**kwargs):
    mysrgs=['oper','check','do']
    if myargs!=list(kwargs.keys()) : raise Exception('Wrong Arguments')
    return sum(kwargs.values())
```

```
test(oper=9,check=10,do=5)
```

در مثال دستور raise Exception برای ایجاد یک خطای دلخواه و توقف ادامه برنامه استفاده شده است .

❋ نکته : برای ارسال یک Tuple به تابع در هنگام فراخوانی از ❋ استفاده می کنیم

❋ نکته : برای ارسال یک Dict به تابع در هنگام فراخوانی از ❋❋ استفاده می کنیم

❋ نکته : آرگومانهای اجباری و اختیاری هم امکان فراخوانی با نام را دارند و در واقع Key Worded هستند .

```
def test(a,b,c=0):
    pass
```

```
test(a=2,b=6,c=7)
```

or

```
test(b=5,a=2,c=8)
```

قلمرو متغیرها

برای متغیرهایی که درون و بیرون تابع ها استفاده می شوند قلمرو تعریف می شود

در حالت کلی دو قلمرو وجود دارد

- عمومی (جهانی) Global Variables
- خصوصی (محلی) Local Variables

متغیرهایی که در برنامه اصلی تعریف می شوند از نوع Global هستند . به همین جهت نیازی به استفاده از کلمه کلیدی global قبل از نام آنها نیست

```
def test():
    print(a+5)
------------------------
a=5
print(a)        → 5
test()          → 10
```

متغیرهای Global در همه‌ی تابع ها قابل استفاده هستند ولی اگر درون یک تابع متغیری با همان نام تعریف شود متغیر درون تابع اولویت دارد .

```
def test():
    a=10
    print(a+5)
------------------------
a=5
print(a)        → 5
test()          → 15
```

❋ نکته : متغیرهای Global در درون تابع ها قابل استفاده هستند ولی قابل تغییر نیستند ، یعنی اگر در تابع مقداری به آنها نسبت داده شود برنامه با خطا روبرو خواهد

حال اگر بخواهیم یک متغیر Global را درون تابع تغییر دهیم باید درون تابع با کلمه کلیدی global آنرا تعریف کنیم .

مثال :

```
def test():
    global a
    a=a+5
    print(a)
------------------------
a=5
print(a)        → 5
test()          → 10
print(a)        → 10
```

❋ نکته : کلمه کلیدی local نداریم و متغیرهای درون تابع ها به صورت خودکار Local تعریف می شوند و پس از پایان تابع حذف می شوند .

متغیرهای غیر محلی

متغیرهای غیر محلی فقط در توابع تودرتو (Nested Functions) کاربرد دارند و با کلمه کلیدی nonlocal تعریف می شوند .

```
def out():
    a=5
    print(a)            →5
    def in():
        nonlocal a
        a=a+5
        print(a)        →10
```

تعریف تابع یک خطی Lambda

lambda یک تابع تک خطی است که فقط یک خط دستور در آن می توان نوشت و یک مقدار برگشتی به متغیر نسب داده شده به آن برگشت می شود ولی تعداد پارامترهای آن محدودیتی ندارد .

نحوه تعریف

var=lambda para1,para2,… : Expr

در هنگام فراخوانی از نام متغیر استفاده می کنیم

var(arg1,arg2,…)

مثال

```
mul=lambda x,y :x*y
print(mul(7,9))      → 63
```

کاربرد اصلی lambda برای تعریف یک تابع درون تابع دیگر است که عملیات درونی را انجام می دهد .

First Class Function : تابعی که آرگومان تابع دیگر است

High Order Function : تابعی که تابع دیگر را به عنوان آرگومان قبول می کند .

Modules نصب و استفاده از ماژولها

ماژول یک فایل py. است که شامل کدهای نوشته شده به زبان Python است که یکی از ویژگیهای اصلی و قدرتمند در زبان Python به شمار می رود .

هر ماژول در بر گیرنده توابع و متغیرها و دستورات کلیدی است .

در هر پروژه Python می توانیم از ماژولهای استاندارد و یا تعریف شده توسط کاربر استفاده کنیم . این ماژولها برای سرعت بخشیدن به پروژه جاری و دوری از نوشتن کدهای طولانی برای فرآیندها ، بسیار موثر هستند .

Modules
- Standard Modules
- User-Defines Modules

ماژولهای استاندارد Python در هنگام نصب Interpreter به صورت خودکار نصب می شوند و در فولدر Lib در محل نصب Python (در Windows معمولا در C:\Program Files (x86)) قرار می گیرند .

ماژولهای تعریف شده توسط کاربر در محلی که فایلهای پروژه جاری قرار دارند ذخیره می شوند

نحوه استفاده از یک ماژول

```
import MaduleName
```

❋ نکته : نیازی به تایپ py. در انتهای نام ماژول نیست .

در بدنه برنامه برای دسترسی به عناصر ماژول (توابع ، متغیرها و . . .) از نام ماژول و علامت . (Dot) استفاده می کنیم .

```
import math


a=math.pi
b=math.gcd(500,150)
```

نام ماژول در سه محل جستجو می شود و اگر وجود نداشته باشد خطا اتفاق می افتد

۱) در فولدر جاری (جایی که فایل فراخوان دهنده وجود دارد)

۲) در فولدر Lib در محل نصب Python

۳) فولدرهایی که در مسیر پیش فرض سیستم عامل تعریف شده‌اند (Environment Variables)

❋ نکته : برای استفاده راحت تر از ماژول در برنامه اصلی می توانیم برای آن یک نام مستعار تعریف کنیم

```
import ModuleName  as Alias
```

مثال

```
import numpy as np
print(np.sin(3.14))
```

در بعضی موارد لازم است یک یا چند تابع مشخص را از یک ماژول Import کنیم . در این حالت به روش زیر عمل می کنیم .

```
from ModuleName import Function/Variable
```

```
from math import pi
```

در حالت برای فراخوانی تابع یا متغیر نیازی به درج نام ماژول نیست

```
print(pi)
```

❋ نکته : برای استفاده از همه‌ی توابع و متغیرهای یک ماژول از ❋ استفاده می کنیم

```
from ModuleName import *
```

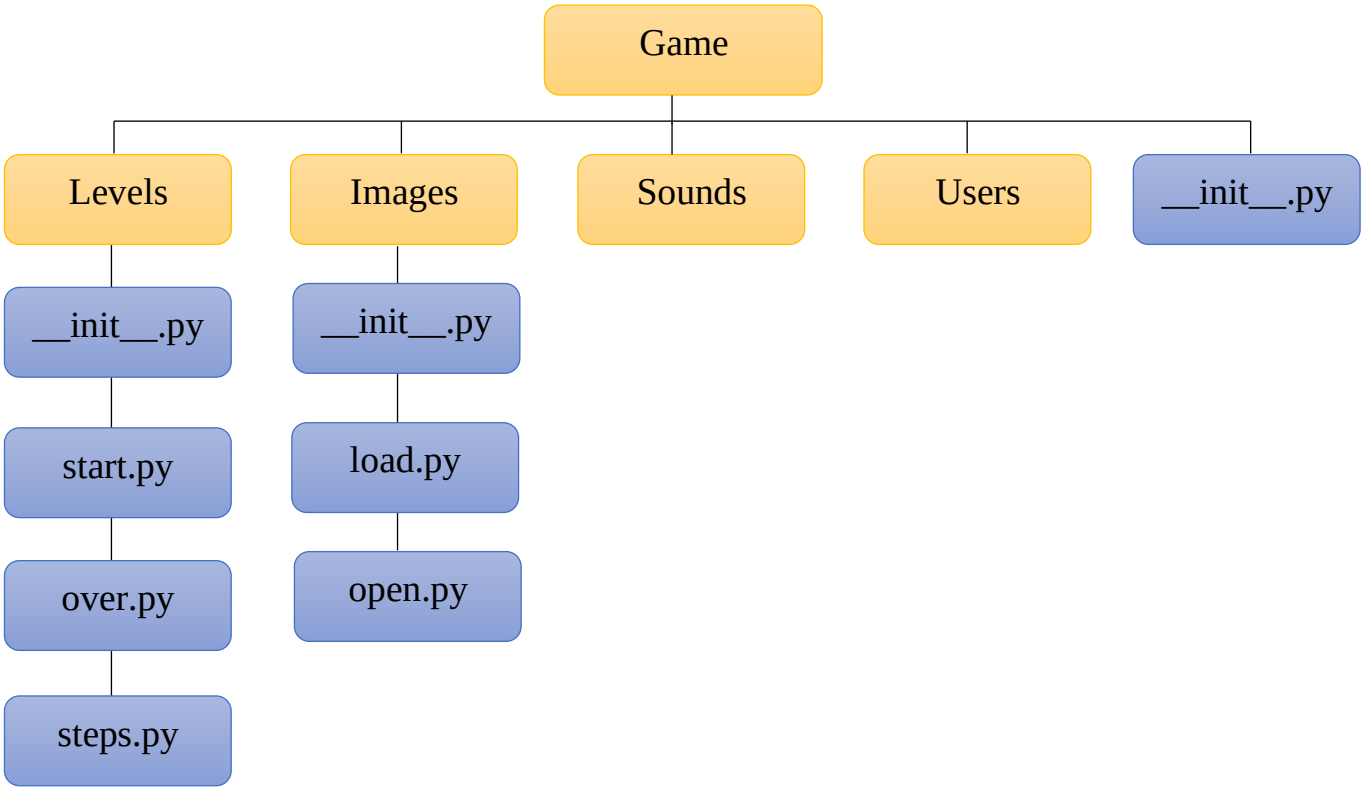
اگر از روش بالا استفاده کنیم برای استفاده از توابع و متغیرها نیازی به .ModuleName نیست

این موضوع باعث می شود تا فراخوانی تابعها آسانتر باشد ولی از خوانایی برنامه کاسته می شود .

❋ نکته : در هنگام استفاده از چند ماژول که به صورت والد و فرزند تعریف شده‌اند باید مراقب باشیم که یک ماژول والد در فرزند خودش import نشود .

Python Packages مدیریت بسته ها

بسته یا Package یک فولدر و زیر فولدرهای آن هستند که ماژولها را در خود ذخیره کرده‌اند .



شکل بالا یک مثال از ایجاد یک Package برای بازی است .

همانطور که مشاهده می شود فولدر اصلی شامل چند زیر فولدر است و هر فولدر دربرگیرنده چند ماژول خواهد بود .

❋ نکته : هر فولدر شامل یک فایل __init__.py است که عملیات شروع اجرای بسته (Initialization) را بر عهده دارد. این فایل می تواند خالی باشد.

نحوه استفاده از Module های ذخیره شده در Package

```
import Game.Levels.start
Game.Levels.start.defficalty(2)
```

برای راحت تر شدن استفاده می توانیم از روش زیر را به کار بگیریم

```
from Game.Levels import start
start.defficalty(2)
```

نصب Package های استاندارد با pip

برای نصب Package هایی که به صورت پیش فرض نصب نیستند از دستور pip استفاده می کنیم

برای مشاهده آخرین نسخه‌ی Package ها و بسته های جدید به PyPi.org مراجعه می کنیم

نصب یک بسته

```
pip install PackageName
```

برای حذف بسته از Uninstall استفاده می کنیم

```
pip uninstall PackageName
```

❋ نکته : مشاهده نسخه جاری pip

```
pip --version
```

❋ نکته : به روز رسانی pip

```
pip install --upgrade pip
```

❋ نکته : مشاهده لیست بسته های نصب شده

```
pip list
```

❋ جستجو در بسته های موجود

```
pip search متن
```

چند بسته پر کاربرد در Python

```
pip install jupyter
```

بسته jupyter یک توزیع از Python است که روی Browser اجرا می شود و شامل امکاناتی مانند رسم نمودار است و محیط گرافیکی دارد .

برای استفاده از Jupyter پس از نصب

```
jupyter  notebook
```

بسته PyInstaller برای ایجاد فایل EXE. در Windows استفاده می شود

```
pip install pyinstaller
```

ایجاد فایل اجرایی از یک فایل

pyinstaller file.py

pyinstaller --onefile file.py

بسته OS : در این بسته توابع دسترسی به فایلها و فولدرها قرار دارند

بسته NumPy یک بسته کاربردی در Python است که برای محاسبات عددی پیشرفته و مدیریت ماتریسهای همگن چند بعدی و حل معادلات خطی استفاده می شود

pip install numpy

بسته SciPy : در این بسته توابع راضی پیشرفته و محاسبات علمی و فنی قرار دارند

بسته MatPlotLib : این بسته برای رسم نمودارها و خروجی گرافیکی استفاده می شود

بسته TKinter و wxPython : این بسته ها شامل توابع و عناصر ایجاد فرمها و کنترل ها هستند و برای ایجاد رابط گرافیکی (GUI) استفاده می شوند .

بسته Pandas : ایجاد جدول و مدیریت Data Structure ها

بسته Pillow : برای پردازش تصویر

بسته های Scrapy و Socket : برای مدیریت شبکه و دستکاری داده ها

مدیریت فایلها File I/O

برای ذخیره دائمی مقادیر متغیرها و استفاده در دفعات بعدی باید آنها را در یک فایل ذخیره کنیم

ایجاد یک فایل

var=open("Address\\File Name","Mode","Encoding")

در متغیر var اشاره گر فایل ذخیره می شود

❋ نکته : اگر آدرس فایل درج نشود فایل در آدرس جاری ایجاد می شود .

❋ نکته : روش درج آدرس

c:\\test\\test.txt

c:/test/test.txt

حالتهای Mode

r : (read) فقط برای خواندن

w : (write) فقط برای نوشتن (در صورت وجود فایل آنرا بازنویسی می کند)

a : (append) اضافه کردن متن به انتهای فایل

x : (Exclusive Creation) فقط برای نوشتن (در صورت وجود فایل خطا اتفاق می افتد)

t : (Text Mode) باز کردن در حالت متنی (حالت پیش فرض)

b : (Binary Mode) باز کردن در حالت Bite Like (برای فایلهای Image و Exe و...)

+ : برای به روز رسانی (قابل استفاده با r و w)

حالتهای Encoding

این آرگومان برای تعیین روشهای کد گذاری استفاده می شود . روش کد گذاری به سیستم عامل بستگی دارد

Windows → cp1252

Linux → utf-8

مثال

```
f=open("test.txt","r")
f=open("Image.jpg","rb")
f=open("c:\\test.test.txt","wt",encoding="utf-8")
f=open("c:/run/prog.exe","r+b")
```

❋ نکته : در حالتهای r و w اشاره گر (Pointer) فایل در ابتدای فایل قرار می گیرد و در حالت a در انتهای آن

متدهای File

File.read(size)

آرگومان Size برحسب Byte است و در صورت عدم درج همه‌ی فایل خوانده می شود و در یک str قرار می گیرد

File.readline()

خواندن یک سطر تا رسیدن به \n (شامل \n) و برگشت به صورت str

File.readlines()

خواندن همه‌ی سطرها و قراردادن در یک list

File.write("String")

نوشتن متن درون فایل (برای ایجاد سطر باید \n درج شود)

File.writelines(list)

File.close()

File.seek(number)

پرش به یک موقعیت مشخص در فایل بر حسب بایت . اگر مقدار معلوم نشود موقعیت فعلی برگشت می شود .

❋ نکته : اگر در هنگام خواندن و نوشتن فایل خطایی اتفاق بیافتد ممکن است File.close() اجرا نشود و در نتیجه فایل باز بماند و ذخیره نشود

برای حل این مشکل از with استفاده می کنیم

```
with open("d:/test.txt","x") as File:
    content=File.read()
```

عملیات روی فایلها و فولدرها Directories and Files

برای اجرای عملیات روی فایلها در سطح سیستم عامل از ماژول os استفاده می کنیم

```
import os
```

متدهای os

os.getcwd()	آدرس فولدر جاری به صورت رشته
os.getcwdb()	آدرس فولدر جاری به صورت باینری
os.chdir("Address")	تغییر مسیر
os.listdir()	لیست فایلها و فولدرهای در یک لیست
os.mkdir("Name")	ایجاد فولدر
os.rename("old","new")	تغییر نام
os.remove("File Name")	حذف فایل
os.rmdir("Folder Name")	حذف یک فولدر خالی

❋ نکته : برای حذف فولدر دارای محتویات از ماژول shutil استفاده می کنیم

```
import shutil
shutil.rmtree("Folder Name")
```

Errors and Exceptions رفع خطا

خطاها در زبانهای برنامه نویسی به ۳ دسته تقسیم می شوند :

۱) Syntax Error خطای نوشتاری

۲) Runtime Error خطای زمان اجرا

۳) Semantic Error خطای معنایی

خطای نوع سوم توسط Interpreter قابل تشخیص نیست و باید الگوریتم برنامه و روش حل مسئله چک شود

در هنگام اتفاق افتادن خطاهای Syntax و Runtime زبان Python یک عملیات raise Exception را اجرا می کند و ادامه اجرای برنامه متوقف خواهد شد .

* نکته : برای ایجاد یک خطای دلخواه در هر جای برنامه از کد زیر استفاده می کنیم :

```
raise Exception(' متن دلخواه')
```

مدیریت خطا و استثنا

برای مدیریت خطاها در Python از دستور try استفاده می کنیم :

```
try:
    Expr1
    Expr2
    . . .
except:
    دستوراتی که در اثر
    خطا اجرا خواهند شد
finally:
    دستوراتی که در هر صورت اجرا خواهند شد
```

می توانیم در جلوی دستور except نوع خطا را مشخص کنیم . مثال :

```
except(ValueError):
    . . .
```

برای این منظور باید خطاهای استاندارد را بدانیم . خطاها به صورت ترکیبی از حروف کوچک و بزرگ (CamelCase) نوشته می شوند

خطاهای استاندارد Python (Built-in Exceptions)

NameError	KeyError
TypeError	RuntimeError
SyntaxError	ImportError
ValueError	TabError
IndentError	OSError
IndexError	DeprecationWarning
AttributeError	

مثال :

```
try:
    a=int(input('Enter a Number : '))
    b=a**2
    print(f ' Square of the Number is : {b} ')
except(ValueError,NameError,TypeError):
    print('Wrong Number')
```

❋ نکته : در صورتی که جلوی except هیچ خطایی درج نشود با اتفاق افتادن هر خطایی کدهای زیر except اجرا خواهند شد .

❋ نکته : برای مشاهده کلاس خطا و علت آن می توانیم از متد exc_info() در ماژول sys استفاده کنیم .

مثال :

```
import sys
a=[4,5,6,3,'b',9,8,'e']


for x in a:
        try:
                print(f 'The Square of  x is {x**2}')
        except:
                print(sys.exc_info())
```

ایجاد خطای سفارشی (User-Defined Exception)

برای ایجاد یک خطای دلخواه باید یک کلاس ایجاد کنیم که مشتق کلاس Exception باشد .

در مثال زیر نحوه تعریف و استفاده از خطاهای سفارشی مشاهده می شود

مثال :

```
class ValueIsSmall(Exception):
   pass
class ValueIsLarge(Exception):
   pass

while True:
   try:
      n=int(input('Enter a Numer : '))
      if n<100 : raise ValueIsSmall
      if n>100 : raise ValueIsLarge
      print('You Guessed Correct')
      break
   except ValueIsSmall:
      print('Value is Small')
   except ValueIsLarge:
      print('Value is Large')
  except :
      print('Wrong Number , Try Again . . . ')
```

برنامه نویسی شئی گرا به کمک کلاسها و اشیاء
Object-Oriented Programming

روش برنامه نویسی شئی گرا به کمک ایجاد کلاسها و اشیا باعث خلاصه شدن کدها و اجتناب از ایجاد کدهای تکراری و طولانی می شود . در واقع کدهایی می نویسیم که بارها تکرار می شوند بدون نیاز به اضافه شدن خطوط برنامه .

هر کلاس دارای دو نهاد است :

۱) صفتها یا خاصیت ها Attributes , Properties

۲) رفتارها یا فرآیندها Behaviors

Characteristics of a Class
- Attributes (Properties)
- Behaviors (Methods)

در Python صفتها بوسیله متغیرها (Variables) و فرآیندها بوسیله توابع (Functions) پیاده سازی می شوند .

در هنگام تعریف یک کلاس به آن Class و در هنگام استفاده به آن نمونه (Instance) یا شئی (Object) می‌گوئیم.

روش تعریف کلاس :

```
class ClassName:
    Class Attributes
    …
    def __init__(self,var1,var2, … ):
        self.var1=var1              (Instance Attributes)
        …
    def method(self,parameter1, parameter2, … ):
        …
    Dunder Methods
    def __del__(self):              (Destructor)
```

در تعریف کلاس

Class Name : نام کلاس است که از قوانین نامگذاری متغیرها پیروی می‌کند.

Class Attributes : متغیرهای عمومی کلاس هستند که برای همه‌ی اشیاء (نمونه ها) کلاس مشترک هستند

__init__ : تابع سازنده است که در هنگام نمونه سازی از کلاس به صورت خودکار فراخوانی می‌شود پارامتر اول تابع سازنده همواره self است که به خود کلاس اشاره می‌کند. سایر پارامترها برای مقدار دهی به متغیرهای کلاس استفاده می‌شوند .

Instance Attributes : متغیرهای درونی کلاس هستند که خواص هر نمونه را نگهداری می‌کنند. این متغیرها به ازاء هر شئی (نمونه) متفاوت هستند .

Methods : برای ایجاد هر فرآیند مورد نیاز کلاس یک تابع تعریف می‌کنیم که به این توابع Method می‌گوئیم. هر تابعی که درون کلاس تعریف می‌شود باید پارامتر self را داشته باشد که اشاره گر به کلاس است .

Dunder Methods : هر کلاس شامل متدهای از پیش تعریف شده است که با Double Underscore نامگذاری شده‌اند . به این توابع به صورت مخفف Dunder و یا Magic Methods می گوئیم .

مانند : __init__ و __repr__ و __str__ و __len__ و . . .

این متدها برای انجام عملیات های مختلف مربوط به کلاس تعریف شده‌اند و یکی از پرکاربردترین آنها __init__ است که عملیات تنظیمهای اولیه (Initialization) را انجام می دهد .

در ادامه مطلب با چند Dunder مهم دیگر آشنا خواهیم شد .

__del__ : تابع مخرب کلاس است که در هنگام حذف نمونه از حافظه به صورت خودکار فراخوانی می‌شود . حذف نمونه از حافظه با پایان یافتن برنامه و یا دستور del اتفاق می‌افتد .

ایجاد نمونه از کلاس (Instance - Object)

پس از تعریف کلاس برای استفاده از خواص و متدها باید نمونه یا شئی از کلاس ایجاد کنیم .

اگر تابع سازنده (__init__) بدون پارامتر باشد ایجاد نمونه به صورت زیر انجام می‌شود

```
object_name = ClassName()
```

و اگر تابع سازنده با پارامتر باشد در هنگام نمونه سازی (تعریف شئی) نام کلاس با آرگومان فراخوانی می‌شود

```
object_name = ClassName(var1,var2, … )
```

در این حالت آرگومانها به پارامترهای تابع سازنده ارسال می‌شوند و متغیرهای Instance بوسیله تابع سازنده مقدار دهی می‌شوند .

برای دسترسی به صفتها و متدهای کلاس و اشیاء از . (Dot) استفاده می کنیم .

مقدار دهی به متغیرهای شئی (Instance Attributes)

برای مقدار دهی به متغیرهای کلاس (صفتها) دو روش وجود دارد :

۱) فراخوانی کلاس با آرگومان ورودی در هنگام نمونه سازی

۲) مقدار دهی با . (Dot)

ترجیح برنامه نویسان از نسخه ۳ به بعد روش اول است

روش اول :

در این روش بوسیله تابع سازنده (Constructor یا Initiator) در هنگام ایجاد کلاس متغیرهای کلاس را مقدار دهی می‌کنیم .

تابع سازنده در Python یک متد Dunder با نام __init__ است .

مثال

```
class Car:
    Wheels=4
    def __init__(self,Brand,Model,Year,Price):
        self.Brand=Brand
        self.Model=Model
```

```
        self.Year=Year
        self.Price=Price
    def last_check(self,date):
        print(f "checkup complete in {date}")
    def leased(self,number_of_months):
        instalment=self.Price*1.1/number_of_months
        return instalment
    def __del__(self):
        print("Car deleted ! ")
```

در این مثال یک اتومبیل به صورت یک کلاس تعریف شده که دارای صفتها و متدهای مخصوص به خود است .

برای ایجاد نمونه (شئی) از دستور زیر استفاده می کنیم

```
car1=Car("Toyota","Camry",2015,22000)
```

حال می توانیم نمونه های متعدد از کلاس Cars ایجاد کنیم

```
car2=Car("Nissan","Maxima",2014,21000)
```

برای دسترسی به هر صفت یا متد از نام نمونه به همراه . (Dot) استفاده می شود .

```
print(car1.Price)
print(car2.Model)
print(car1.leased(36))
```

روش دوم :

```
class Car:
    def __init__(self):
        Brand=""
        Model=""
        Year=int()
        Price=int()
    def last_check(self,date):
```

```
        print(f "checkup complete in {date}")
    def leased(self,number_of_months):
        instalment=self.Price*1.1/number_of_months
        return instalment
```

در موقع نمونه سازی

```
car1=Car()
car1.Brand="Toyota"
car1.Model="Camry"
```

همانطور که در مثال مشاهده می شود تعریف متدها مشابه تعریف یک تابع است .

بررسی چند تابع Dunder مهم

__init__ : تابع سازنده کلاس است که در هنگام نمونه سازی از کلاس به صورت خودکار اجرا می‌شود و معمولا برای مقدار دهی به Instance Variable ها استفاده می‌شود .

__del__ : تابع مخرب کلاس است که در هنگام حذف نمونه از حافظه فراخوانی می‌شود

__str__ : در صورت چاپ کلاس با دستور Print مقداری که در این متد Return شده نمایش داده می‌شود

```
def __str__(self):
    print(self.Brand)
    print(self.Model)
    print(self.Year)
    print(self.Price)
    return ""
```

پس از تعریف این متد اگر در برنامه اصلی print(car1) را اجرا کنیم مشخصات اتومبیل نمایش داده می‌شود .

__repr__ : مشابه str ولی برای توسعه دهنده ها مناسب است و در هنگام چاپ تابع repr() را قرار می‌دهیم

```
print(repr(car1))
```

__len__ : اگر تابع len روی کلاس استفاده شود مقدار برگشتی این تابع نمایش داده می‌شود

__getitem__ : این متد یک پارامتر index دریافت می‌کند و با آن می‌توانیم عضو یک لیست داخل کلاس را نمایش دهیم در هنگام استفاده از کلاس می‌توانیم مانند Iterable ها index و مقدار آن را دریافت می‌کنیم

print(car1[2])

__setitem__ : این متد یک پارامتر index و یک مقدار دریافت می‌کند و امکان درج مقدار در یک لیست درون کلاس را فراهم می‌کند در هنگام استفاده از کلاس مقدار و index را به کلاس ارسال می‌کنیم

car1[2]=15000

توجه داشته باشید در دو متد اخیر باید یک Iterable درون کلاس تعریف شود و این دو متد به آن دسترسی داشته باشند .

__setattr__ : این متد مشابه setitem است با این تفاوت که در هنگام مقدار دهی به صفتها فراخوانی می‌شود

car1.Price=65000

__call__ : با ایجاد این متد کلاس قابلیت فراخوانی (callable) خواهد داشت و می‌توانیم نام کلاس را با آرگومان استفاده کنیم. این متد دارای آرگومانهای بدون نام با تعداد متغیر است و با آن می‌توانیم صفتهای جدید به کلاس اضافه کنیم و یا صفتهای موجود را تغییر دهیم .

__add__ : در این تابع عملیاتی را تعریف می‌کنیم که در صورت استفاده از عملگر + بین دو کلاس این عملیات انجام می‌شود . مقدار برگشتی از این تابع در هنگام جمع دو نمونه از کلاس به برنامه اصلی برگشت می‌شود در این متد پارامتر اول که self است به خود کلاس و پارامتر دوم به شئی دوم کلاس اشاره می‌کند .

__gt__ : تعریف عملیات بزرگتر در هنگام مقایسه دو نمونه. آرگومانها مشابه __add__ است

__lt__ : تعریف عملیات کوچکتر در هنگام مقایسه دو نمونه. آرگومانها مشابه __add__ است

__eq__ : تعریف عملیات مساوی در هنگام مقایسه دو نمونه. آرگومانها مشابه __add__ است

__dict__ : مشاهده دیکشنری صفتهای کلاس

__iter__ :

__set__ : Data Descriptor ؟؟

__get__ : Data Descriptor ؟؟

برنامه نویسی شئی گرا علاوه بر خلاصه کردن کدها و انجام عملیات ساده‌تر سه خاصیت مهم دیگر در برنامه نویسی را فراهم می کند

- وراثت Inheritance
- کپسوله سازی Encapsulation
- چند شکلی polymorphism

وراثت Inheritance

وراثت به این معناست که می‌توانیم یک کلاس را به صورت مشتق کلاس دیگر تعریف کنیم . در این حالت همه‌ی صفتها و متدهای کلاس والد (Parent یا Base) به کلاس فرزند (Child یا Derived) منتقل می‌شود .

کلاس فرزند می‌تواند خواص و متدهای مستقل هم داشته باشد .

نحوه تعریف :

```
class Child(Parent):
    . . .
```

در هنگام تعریف باید نام کلاس والد جلوی کلاس فرزند نوشته شود .

مثال :

```
class Motorcycle(Car):
    Wheels = 2
    def __init__(self, …):
        …
```

* نکته : اگر کلاس فرزند تابع __init__ نداشته باشد تابع سازنده والد برای آن اجرا خواهد ولی اگر در فرزند تابع سازنده تعریف کنیم تابع سازنده والد باطل خواهد شد و برای اجرای آن از super() استفاده می‌کنیم .

کپسوله سازی Encapsulation

کپسوله سازی برای حفاظت از صفتها و متدها استفاده می‌شود . وقتی بخواهیم یک یا چند صفت یا متد در خارج از کلاس قابل دسترسی نباشند از این خاصیت استفاده می‌کنیم .

برای کپسوله کردن یک صفت یا متد کافیست در ابتدای نام آن Underscore قرار دهیم .

مثال :

```
class Car:
    def __init__(self):
        self.__maxprice=25000
```

در این مثال متغیر maxprice__ فقط با یک تابع درون کلاس قابل تغییر است .

چند شکلی Polymorphism

چند شکلی در روش OOP یعنی بتوانیم یک متد (تابع) را با یک نام ثابت ولی پارامترهای متفاوت تعریف کنیم .

یعنی یک متد با نام مشخص با توجه به ورودی های متفاوت خروجی متفاوتی را برگشت دهد و این متد در کلاسهای مختلف کار متفاوتی انجام دهد .

مثال :

```
class Motorcycle(Car):
    def __init__(self, …):
        …
    def leased(self,number_of_months):
        instalment=self.Price*1.2/number_of_months
        return instalment
```

در مثال مشاهده می کنیم که متد leased هم در کلاس واحد (Car) و هم در کلاس فرزند (Motorcycle) تعریف شده است . اما در کلاس فرزند ضریب ۱.۱ به ۱.۲ تغییر یافته .

حال در نمونه سازی اگر همین متد (leased) اگر با کلاس والد فراخوانی شود نتیجه متفاوتی با متد مشابه در کلاس فرزند برگشت می‌شود .

مفهوم Decorator

دکوراتورها امکان تغییر (Modify) و گسترش (Develop) عملکرد توابع و متدها و کلاسها را فراهم می‌کنند .

در واقع دکوراتور خود یک تابع است که یک تابع دیگر را به عنوان آرگومان ورودی دریافت می‌کند و بعد از تغییر شکل دادن آن ، همان تابع را بر می‌گرداند .

باید توجه داشته باشید که توابع First Class Object هستند . یعنی امکان فراخوانی با یک Callable را دارند .

Callable ها در Python سه نوع هستند:

Function - Method - Class -

پس می‌توانیم با یک دکوراتور عملکرد یک Callable را تغییر دهیم یا بهبود ببخشیم .

برای تعریف دکوراتور به دو تابع تودرتو نیاز داریم ، اول تابع اصلی (High Order Function) که یک Callable را به عنوان آرگومان می‌پذیرد و دوم تابع درونی (Wrapper Function یا Inner Function) که عملیات تغییر و گسترش را به عهده دارد و در واقع تابعی است که تابع ارسال شده به دکوراتور را فراخوانی می‌کند و آنرا تغییر می‌دهد و سپس به تابع اصلی برگشت می‌دهد .

مثال :

```
def addline(func):
    def wrapper():
        print("------------------------")
        func()
        print("------------------------")
        return func
    return wrapper
```

در مثال بالا تابع اصلی (addline) که نام دکوراتور است یک تابع را به عنوان آرگومان دریافت می‌کند سپس تابع درونی (wrapper) یک سطر در ابتدا و یک سطر در انتهای خروجی تابع نمایش می‌دهد و آنرا به تابع اصلی بر می‌گرداند .

استفاده از دکوراتور

برای استفاده از دکوراتور از علامت @ استفاده می‌کنیم . یعنی در خط بالای تعریف تابع نام دکوراتور را درج می‌کنیم

```
@addline
```

```
def myfunc():
    print("Main Function Print")
```

با این دستورات تابع myfunc به صورت آرگومان به addline ارسال می‌شود و سپس تابع wrapper دستورات مورد نظر را اجرا می‌کند . در تابع wrapper خطی که تابع func() را فراخوانی می‌کند در واقع تابع myfunc را اجرا می‌کند.

حال با اجرای تابع myfunc در هر قسمت از برنامه اصلی خروجی به صورت زیر خواهد بود :

```
myfunc ()
```

```
-------------------------
Main Function Print
-------------------------
```

با این روش می‌توانیم عملکرد توابع را بدون دستکاری در بدنه آن تابع تغییر دهیم .

به عنوان یک مثال کاربردی تصور کنید ۲۰ گزارش داریم که به صورت توابع جداگانه ایجاد شده‌اند و هر تابع یک گزارش را به خروجی ارسال می‌کند حال می‌خواهیم امکان ایجاد گزارش فقط با استفاده از نام کاربری و رمز امکان پذیر باشد ، کافیست یک دکوراتور برای چک کردن User و Pass بسازیم و همه‌ی توابع را با این دکوراتور تغییر دهیم . این کار به سادگی اجرا می‌شود و نیازی به دستکاری در همه‌ی تابع‌ها نیست .

ایجاد دکوراتور برای تابع دارای آرگومان

در مثال بالا تابع myfunc هیچ آرگومانی ندارد حال اگر بخواهیم یک تابه دارای آرگومان را به دکوراتور ارسال کنیم تابع wrapper باید مشابه تابع ارسال شونده به دکوراتور دارای آرگومان باشد و این آرگومانها باید یکسان باشند .

مثال :

```
def addline(func):
    def wrapper(a):
        print("------------------------")
        func(a)
        print("------------------------")
        return func
    return wrapper
```

در موقع تعریف و فراخوانی :

```
@addline
def myfunc(n):
    print(f "The name is : {n}")

name="Reza"
myfunc(name)
```

⬇

```
------------------------
The name is Reza
------------------------
```

❋ نکته : اگر تابعی با تعداد آرگومانهای نامشخص داشته باشیم در تابع wrapper می‌توانیم از *args استفاده کنیم و همینطور برای ارجاع آرگومانهای با تعداد نامشخص و دارای نام می‌توانیم از **kwargs استفاده کنیم .

❋ نکته : متدهای یک کلاس به سه دسته تقسیم می‌شوند :

Class Method - Static Method - Instance Method -

Instance Method : متدهای معمول و پر استفاده هستند که به صورت عادی تعریف می‌شوند و آرگومان اول آنها خود کلاس است که در هنگام تعریف با کلمه self به عنوان آرگومان اول و به صورت اجباری ارجاع می‌شود . این متدها به Attribute ها و متدهای نمونه‌ی ایجاد شده از کلاس دسترسی دارند و می‌توانند آنها را تغییر دهند یا فراخوانی کنند .

این نوع متد از طریق نام کلاس قابل دسترسی نیست مگر اینکه خود کلاس به عنوان آرگومان به آنها ارسال شود . در اصل این متدها برای نمونه (Instance) ایجاد شده از کلاس ساخته می‌شوند و قرار نیست عناصر داخل کلاس را تغییر دهند .

مثال : در کلاس Car که قبلا تعریف کردیم متدهای last_check و leased از این نوع هستند .

Static Method : این نوع متدها برای ایجاد یک فرآیند که مربوط به کلاس است استفاده می‌شوند و خود کلاس به عنوان آرگومان به آنها ارسال نمی‌شود و مانند یک تابع که درون کلاس تعریف شده و به عنوان عضوی از کلاس شناخته می‌شود ولی به عناصر کلاس و هم عناصر نمونه کلاس دسترسی ندارد .

برای اعلام یک متد به عنوان Static Method در هنگام تعریف آن از دکوراتور @staticmethod استفاده می‌کنیم .

```
@staticmethod
def func(…):
    …
```

اینگونه متدها برای عملیات درونی کلاس که به نمونه‌ها مربوط نیست کاربرد دارند .

مثال :

فرض کنید در تاریخ مشخصی بخواهیم یک تخفیف عمومی برای اتومبیل‌ها اعلام کنیم . در این مورد به صفتها و متدهای کلاس نیازی نداریم و فقط کافیست تا تاریخ را به عنوان آرگومان به تابع تخفیف ارسال کنیم ، پس باید یک متد Static تعریف کنیم تا از طریق نام کلاس و بدون نیاز به آرگومان کلاس ، فراخوانی شود .

```
class Car:
    Wheels=4
    def __init__(self,Brand,Model,Year,Price):
        self.Brand=Brand
        self.Model=Model
        self.Year=Year
        self.Price=Price
    def last_check(self,date):
        print(f "checkup complete in {date}")
```

```
    def leased(self,number_of_months):
        instalment=self.Price/number_of_months
        return instalment


    @staticmethod
    def discount(date):
        print(f "All Cars Sale with 20% Discount Until {date}")
```

در این مثال برای اعلام تخفیف کافیست نام کلاس همراه تابع discount فراخوانی شود و نیازی به ساختن نمونه از کلاس نیست .

```
Car.discount("2018/05/15")
```

Class Method : این نوع متدها برای ایجاد تغییر و گسترش عناصر کلاس تعریف می‌شوند ، یعنی برخلاف متدهای Instance که برای نمونه استفاده می‌شوند متدهای Class تغییری در نمونه‌ها ایجاد نمی‌کنند و برای عملیات داخلی کلاس ساخته می‌شوند .

مشابه Static Method این نوع به عناصر نمونه دسترسی ندارند و برای عملیات داخلی کلاس طراحی شده‌اند در نتیجه برای استفاده از این نوع متد نیازی به نمونه سازی از کلاس نیست .

در این نوع هر متد یک آرگومان اجباری اول به نام cls دارد که مشابه self در Instance Method است و خود کلاس را در هنگام فراخوانی ارسال می‌کند . البته انتخاب نام cls اختیاری است و برای خوانایی بیشتر و تمایز با self معمولا از این نام استفاده می‌شود .

مثال :

فرض کنیم در کلاس Car می‌خواهیم تعداد اتومبیلهای فروش رفته را شمرده و با یک تابع آنرا چاپ کنیم .

تعداد متغیری است که به نمونه ها متصل نیست و درون کلاس با هر بار نمونه سازی (اجرای __init__) یک عدد اضافه می‌شود . برای چاپ آن باید تابعی بسازیم که دسترسی به عناصر کلاس داشته باشد ولی با نمونه ها کاری ندارد .

```
class Car:
    Wheels=4
    Count=0
    def __init__(self,Brand,Model,Year,Price):
        self.Brand=Brand
        self.Model=Model
        self.Year=Year
        self.Price=Price
        Car.Count+=1

    def last_check(self,date):
        print(f"checkup complete in {date}")
    def leased(self,number_of_months):
        instalment=self.Price/number_of_months
        return instalment
    @staticmethod
    def discount(date):
        print(f"All Cars Sale with 20% Discount Until {date}")

    @classmethod
    def  cars_count(cls):
        print("Number of sold cards is : ",cls.Count)
```

حال با فراخوانی Car.cars_count() تعداد نمونه‌های ساخته شده از کلاس Car نمایش داده خواهد شد .

❋ نکته : دکوراتور @property

با این دکوراتور می‌توانیم یک متد را به یک Attribute تبدیل کنیم .

بیشترین کاربرد این دکوراتور برای ایجاد محدودیت (Validation) در هنگام استفاده از Class Attribute یا Instance Attribute ها است . در واقع عملیات Getter و Setter با این دکوراتور امکان پذیر می‌شود .

در برنامه نویسی شئی گرا عملیات های Get و Set مواقعی استفاده می‌شود که لازم است یک Attribute در خارج از کلاس در دسترس نباشد (Private باشد) و فقط با توابع درونی کلاس امکان خواندن و تغییر آن فراهم شود و این کار Encapsulation را پیاده سازی می‌کند .

البته در Python همه‌ی Instance Attribute ها چه از نوع عمومی و چه از نوع خصوصی توسط نمونه ایجاد شده از کلاس قابل فراخوانی و تغییر هستند و عملیات Get و Set بیشتر برای کنترل داده ها استفاده می‌شود .

در هنگام ایجاد متد Getter از این دکوراتور استفاده می‌کنیم و باید توجه داشته باشیم که متد آرگومان نداشته باشد و فقط برای برگرداندن یک Attribute استفاده شود .

مثال :

فرض کنیم در کلاس Car می‌خواهیم قیمت خودروها بین ۲۰,۰۰۰ تا ۱۰۰,۰۰۰ محدود شود .

```
class Car:
    Wheels=4
    def __init__(self,Brand,Model,Year,Price):
        self. Brand=Brand
        self. Model=Model
        self. Year=Year
        self. _Price=Price
    @property
    def Price(self):
        return self. _Price
    @Price. setter
    def Price(self, value):
        if 20000<= value <=100000:
            self. _Price=value
```

```
        else:
            raise Exception("Price in not Valid ! ")
```

توجه داشته باشید متد زیر @property که getter است با نام Price تعریف شده و باید متد setter همنام متد getter باشد .

به نمونه سازی و استفاده از getter و setter توجه کنید :

```
car1=Car("Toyota","Camry",2017,20000)
print(car1.Price)    → 20000
car1.Price=45000
print(car1.Price)    → 45000
```

اگر در خط سوم مقداری کمتر از ۲۰ هزار یا بیشتر از ۱۰۰ هزار وارد کنیم متد setter باعث ایجاد خطا خواهد شد .

در این مثال در هنگام نمونه سازی شرط چک نخواهد شد چون در تابع __init__ قانونی برای Price تعریف نشده است یعنی اگر در خط اول برنامه آرگومان آخر ۱۵۰۰۰ درج شود خطایی دریافت نمی‌شود زیرا هنوز تابع setter فراخوانی نشده است

برای چک کردن شرط هم در موقع نمونه سازی و هم در موقع مقدار دهی به Attribute می‌توانیم در تابع سازنده تابع setter را فراخوانی کنیم .

برای این منظور در تابع سازنده self.Price=Price قرار می‌دهیم این کار باعث فراخوانی تابع setter خواهد شد و باید در تابع getter و setter متغیر برگشتی یک مقدار جدید (به جز Price) باشد .

```
@property
def Price(self):
    return self.p
@Price. setter
def Price(self, value):
    if 20000<= value <=100000:
        self.p=value
    else:
        raise Exception("Price in not Valid ! ")
```

اتصال به پایگاه داده‌ها در Python

برای ذخیره داده های با حجم بالا و مدیریت داده ها نیاز به ذخیره آنها در یک پایگاه داده (Database) داریم .

به همین جهت مدیریت داده ها در یک زبان برنامه نویسی دارای اهمیت است .

تعریف Database : مجموعه ای از داده ها و اطلاعات مربوط به هم که به صورت ساختار Table→Record→Field ذخیره می شوند .

برای مدیریت Database ها پراستفاده‌ترین زبان SQL است (Structured Query Language)

برای همین بسیاری از نرم‌افزارهای مدیریت داده مانند : SQL Server و SQLite و Access و Oracle و MySQL و PostgreSQL از این زبان برای ذخیره و دستکاری داده ها در یک Database استفاده می کنند

در Python امکان دسترسی به انوع نرم‌افزارهای مدیریت Database فراهم شده است . روش ذخیره سازی و همینطور دستورات SQL در آنها مشترک است ولی روش ایجاد و دسترسی متفاوت است .

نرم‌افزارهای پایگاه داده با دو روش داده ها را مدیریت می کنند :

- روش مبتنی بر فایل (SQLite و Access)
- روش مبتنی بر Client-Server (SQL Server و MySQL)

در ادامه روش اتصال به چند نرم‌افزار Database مشهور و پرکاربرد را بررسی می‌کنیم .

اتصال به SQLite

SQLite یک پایگاه داده مبتنی بر فایل است

ماژول اتصال import sqlite3

```
connection=sqlite3.connect("Filename. db")
cursor=connection.cursor()
cursor.execute("SQL Expression")
connection.commit()
connection.close()
```

دستور اول : ایجاد فایل Database

دستور دوم : ایجاد Pointer برای اجرای عملیات (مانند read و write)

دستور سوم : اجرای دستورات SQL

دستور چهارم : ذخیره Database

دستور پنجم : بستن اتصال

❋ نکته : با هر عملیات read یا write اشاره گر cursor به طور خودکار به رکورد بعدی اشاره خواهد کرد .

دستورهای SQL

دستورات زبان SQL به چهار دسته اصلی تقسیم می‌شوند :

- DDL (Data Definition Language) : دستوراتی که برای تعریف داده ها استفاده می‌شوند

CREATE INDEX – CREATE TABLE – CREATE DATABASE

ALTER TABLE – ALTER DATABSE

DROP INDEX – DROP TABLE

- DML (Data Manipulation Language) : دستوراتی که برای بازیابی و تغییر داده ها استفاده می‌شوند

INSERT INTO – DELETE – UPDATE – SELECT

- DCL (Data Control Language) : دستوراتی که برای ایجاد مجوز ها و کنترل دسترسی کاربران هستند

REVOKE - GRANT

- TCL (Transaction Control Language) : دستوراتی که برای انتقال داده ها استفاده می‌شوند

SAVEPOINT – ROLLBACK – COMMIT

❋ نکته : در بعضی از دسته بندیها دسته ای به نام DQL تعریف می شود که فقط شامل دستور SELECT است .

دستور ایجاد جدول

```
cursor.execute("CREATE TABLE Tablename(Field1 datatype,Field2 datatype, …)")
```

❋ نکته : نوع داده ها در SQLite به صورت زیر تعریف می شوند :

text - integer – real – blob - null

❋ نکته : برای تعریف کلید اولیه در SQLite از دستور PRIMARY KEY بعد از نوع داده استفاده می‌کنیم

مثال :

```
CREATE TABLE Students(
std_id INTEGER PRIMARY KEY,
first_name TEXT,
last_name TEXT,
average REAL,
) ;
```

❋ نکته : در دستورات SQL حروف کوچک و بزرگ انگلیسی یکسان هستند

❋ نکته : اگر دستور ایجاد جدول دو بار اجرا شود با خطای جدول تکراری روبرو خواهیم شد ، برای جلوگیری از ایجاد جدول در صورت وجود از دستور IF NOT EXISTS استفاده می کنیم

```
cursor.execute("CREATE TABLE IF NOT EXISTS Tablename(Field1 datatype,Field2 datatype, …)")
```

دستور درج یک یا چند رکورد

```
cursor.execute("INSERT INTO Tablename VALUES(?,?,?)",(Field1,Field2,Field3))
```

* نکته : در تابع VALUES به ازاء هر فیلد یک ? قرار می دهیم و در آرگومان بعدی فیلدها را به صورت Tuple یا List ارسال می‌کنیم . (هر بار فقط یک Tuple ارسال می‌شود)

* نکته : برای درج رکورد ها با استفاده از INSERT می توانیم از f string هم استفاده کنیم .

```
cursor.execute(f"INSERT INTO Tablename VALUES('{Field1}','{ Field2}','{ Field3}')")
```

ولی اینکار امن نیست و خطر SQL Injection را به همراه دارد زیرا می توان به جای مقدار فیلدها یک متن را ارسال کرد که آن متن دستورات دیگر SQL باشند . (مانند دستور حذف جدول)

دستور استخراج یک یا چند رکورد

```
cursor.execute("SELECT Field1,Field2, . . . FROM Tablename")
```

* نکته : برای استخراج همه‌ی فیلدها به جای نام فیلدها * قرار می‌دهیم

* نکته : مقدار برگشتی از دستور SELECT یک List است که هر رکورد در آن به صورت Tuple ذخیره شده است

برای مشاهده رکوردها باید خروجی تابع تبدیل به لیست شود که بوسیله متدهای fetch این کار امکان پذیر است

```
Records=cursor.fetchall()          مشاهده همه‌ی رکوردها
Records=cursor.fetchmany(size)     مشاهده تعداد رکوردهای معین
Records=cursor.fetchone()          مشاهده یک رکورد
```

البته می توانیم خروجی متد execute را مستقیم به List تبدیل کنیم .

```
Records=list(cursor.execute("SELECT * FROM Tablename"))
```

* نکته : در بسیاری از موارد لیست همه‌ی رکوردها مورد نیاز نیست و دسترسی به رکوردهای دارای شرایط مشخص پر کاربرد تر است . برای این منظور در دستور SELECT از عبارت WHERE استفاده می‌کنیم

SELECT Field1,Field2, . . . FROM Tablename WHERE Field=شرط

مثال:

```
cursor.execute("SELECT * FROM Students WHERE std_id=902310")
cursor.execute("SELECT * FROM Students WHERE average>17")
```

❋ نکته : بوسیله دستور ORDER BY می‌توانیم رکوردها را مرتب کنیم

```
cursor.execute("SELECT * FROM Students WHERE average>17 ORDER BY last_name")
```

ترتیب مرتب سازی به صورت صعودی است و با دستورات ASC یا DESC بعد از نام فیلد می‌توانیم ترتیب را مشخص کنیم .

❋ نکته : با دستور LIMIT می‌توانیم تعداد رکوردها را مشخص کنیم

```
cursor.execute("SELECT * FROM Students ORDER BY lats_name LIMIT 50")
```

دستور به روز رسانی یک یا چند فیلد

cusrsor.execute("UPDATE Tablename SET Field=مقدار جدید WHERE Field=شرط")

مثال:

```
cursor.execute("UPDATE Students SET average=18.5")
```

در این مثال معدل همه‌ی دانشجویان به عدد ۱۸/۵ تغییر خواهد یافت که مطلوب نیست ، به همین دلیل همراه با دستور UDATE همواره از عبارت شرطی استفاده می‌کنیم

```
cursor.execute("UPDATE Students SET average=18.5 WHERE std_id=545985")
```

❋ نکته : برای انتقال به متغیرها به شرط یا دیگر قسمتهای عبارتهای SQL از f string و یا ? استفاده می‌کنیم .

همانطور که قبلا گفته شد روش ? مطمئن تر است

مثال :

```
std_num=input("Enter Student Number : ")
cursor.execute("UPDATE Students SET average=18.5 WHERE std_id=?",(std_num,))
```

در مثال مشاهده می‌شود که برای جایگزین کردن مقادیر ? باید از Tuple استفاده کرد .

دستور حذف یک یا چند رکورد

```
cursor.execute("DELETE FROM Tablename  WHERE Field=شرط")
```

توجه داشته باشید که اگر شرط درج نشود همه‌ی رکوردها حذف خواهند شد

```
std_num=input("Enter Student Number : ")
cursor.execute("DELETE FROM Students  WHERE std-id=?",[std_num])
```

در این مثال کد دانشجو به صورت List ارسال شده است .

اتصال به Microsoft Access

یکی دیگر از نرم افزارهای پر استفاده برای نگهداری پایگاه داده ها Access است .

برای ایجاد و اتصال به Access ابتدا باید دو Module را نصب کنیم

```
pip install pyodbc
pip install msaccessdb
```

ایجاد فایل Database

```
import msaccessdb
msaccessdb.create("آدرس و نام فایل")
```

البته می‌توانیم فایل accdb. را با استفاده از نرم افزار Access هم ایجاد کنیم تا نیاز به ماژول msaccessdb نباشد

❋ نکته : با هر بار اجرای دستور بالا Database جدید جایگزین Database قبلی خواهد شد

اتصال به Database

```
connection= pyodbc. connect(r"Driver={Microsoft Access Driver (*.mdb, *.accdb)};
DBQ=آدرس فایل داده")
cursor = connection.cursor()
```

برای اجرای صحیح کد بالا باید Microsoft Access Engine Driver نصب باشد

در صورت استفاده از نسخه ۶۴ بیتی Python این Driver به صورت پیش فرض نصب نیست .

با اجرای کد [x for x in pyodbc.drivers() if x.startswith('Microsoft Access Driver')] در IDLE می‌توانیم از نصب بودن Driver مطلع شویم .

اگر خروجی کد یک List خالی [ ] باشد یعنی Driver نصب نیست .

برای نصب Driver عبارت ACE driver را جستجو می‌کنیم و یا از لینک زیر فایلهای نصب را دانلود می‌کنیم

https://www.microsoft.com/en-us/download/details.aspx?id=54920

دستورات مربوط به ایجاد جدول و درج و حذف رکوردها و استخراج داده ها از Database مشابه SQLite است و می‌توانیم همان دستورات SQL را برای مدیریت داده‌ها در Access هم بکار ببریم .

اتصال به MySQL و MariaDB

پایگاه داده‌های MySQL و MariaDB پایگاه داده‌های مبتنی بر Client/Server هستند.

بدین معنی که فایل پایگاه داده درون یک Data Directory که روی Server قرار دارد ایجاد می شود .

MariaDB توسط گسترش دهنده های اصلی MySQL بعد از خریداری شدن MySQL توسط شرکت Oracle در سال ۲۰۱۰ ایجاد شد تا نسخه مشابه MySQL همچنان متن باز و مجانی باقی بماند.

برای دسترسی به Sever باید IP Address یا URL آنرا به برنامه‌ی فراخوان کننده پایگاه داده بدهیم علاوه بر آدرس نیاز به User و Password هم داریم .

برای ایجاد پایگاه داده می توانیم از برنامه MySQL و یا یک شبیه ساز Web Server مانند XAMPP استفاده کنیم.

نرم افزار XAMPP شامل Apache HTTP Server و MariaDB و PHP و Perl است .

پس از نصب و راه اندازی XAMPP امکان دسترسی به پایگاه داده MariaDB روی Local Host (آدرس 127.0.0.1) فراهم خواهد شد .

برای اتصال به پایگاه داده ابتدا باید ماژول pymysql را نصب کنیم :

```
pip install pymysql
```

اتصال به Database

```
import pymysql
connection=pymysql.connect(host="آدرس سرویس دهنده",user="نام کاربری",password="رمز",
db="نام پایگاه داده")
cursor=connection.cursor()
```

مثال :

```
connection=pymysql.connect(host="localhost",user="reza",password="123",
db="StudentDB")
cursor=connection.cursor()
```

دستورات مربوط به ایجاد جدول و درج و حذف رکوردها و استخراج داده ها از Database مشابه SQLite است و می‌توانیم همان دستورات SQL را برای مدیریت داده‌ها در MySQL هم بکار ببریم .

تنها تفاوت این است که در هنگام استفاده از INSERT INTO یا سایر دستورات برای ارسال مقادیر فیلدها به جای ? از %S استفاده می‌کنیم .

```
cursor.execute("INSERT INTO students VALUES(%s,%s,%s,%s,%s) ",(1234,"Ali","Ahmadi",22,18.5)
```

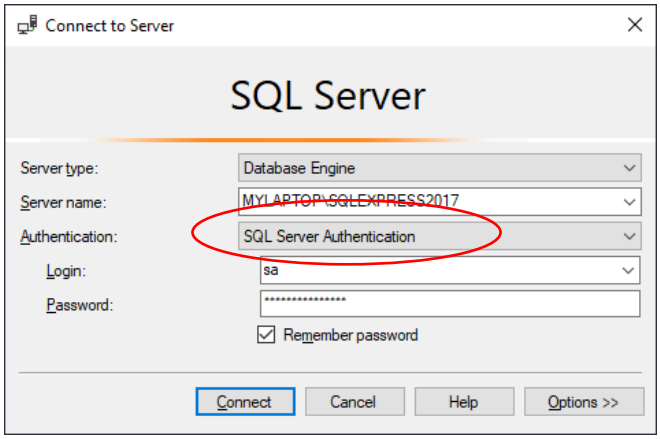
اتصال به Microsoft SQL Server

نرم‌افزار SQL Server هم مشابه MySQL یک پایگاه داده مبتنی بر Client\Server است .

برای همین در هنگام اتصال باید آدرس Server و یا IP Address آنرا داشته باشیم .

در SQL Server دو روش برای دسترسی به بانک اطلاعاتی وجود دارد :

- Windows Authentication
- SQL Server Authentication



در حالت اول باید با User و Pass حساب کاربری تعریف شده در Windows وارد شده باشیم یا اینکه User و Pass را داشته باشیم .

در حالت دوم با User و Pass تعریف شده در SQL Server وارد می شویم . این User و Pass در هنگام نصب Instance تعریف می‌شوند .

اتصال به Database

برای اتصال از ماژول pyodbc استفاده می‌کنیم

```
pip install pyodbc
```

مشابه روشهای قبل ابتدا باید اتصال به پایگاه داده و جدول مورد نظر در آن برقرار شود .

```
import pyodbc
connection=pyodbc.connect(r"Driver={SQL Server Native Client 11.0};Server= آدرس سرویس دهنده ;
Database=نام پایگاه داده;uid= نام کاربری ; pwd= رمز ")
cursor=connection.cursor()
```

❊ نکته : اگر در پایگاه داده حالت Windows Authentication تنظیم شده باشد و با Account ویندوز Login کرده باشیم نیازی به درج uid و pwd نیست .

مثال :

```
Database=pyodbc.connect(r"Driver={SQL Server Native Client 11.0};Server=MYLAPTOP\SQLEXPRESS2017;Database=students;uid=sa;pwd=123")

cursor=Database.cursor()
```

در مثال نام اتصال Database گذاشته شده و Server به صورت ComputerName/InstanceName تعریف می‌شود نام پایگاه داده students است و نام کاربری sa درج شده (این User پیش فرض SQL Server است که در هنگام نمونه سازی انتخاب می‌شود sa=system administrator)

دستورات مربوط به ایجاد جدول و درج و حذف رکوردها و استخراج داده ها از Database مشابه SQLite است و می‌توانیم همان دستورات SQL را برای مدیریت داده‌ها در SQL Server هم بکار ببریم .