جزوه پایتون

بصورت خلاصه!

WWW.PERSIANOVIN.COM

خلاصه پایتون

| خلاصه ای از سینتکس پایتون در یک نگاه:

متغیرها و انواع داده

- تعریف متغیر
- عدد صحیح (integer)
- عدد اعشاری (Float)
- مقادیر بولی (Bool)
- استرینگ (String)
- تاپل (Tuple)
- لیست (List)
- دیکشنری (Dictionary)
- ست (Set)

```
my_var = 5
5
5.1
True, False
"Hello"
(1, 2, 3, 4, 5)
[1, 2, 3, 4, 5]
{"A":1, "B":2}
{1, 2, 3, 4, 5}
```

عملگرهای حسابی

- جمع
- تفریق
- ضرب
- تقسیم
- جز صحیح تقسیم
- باقی مانده
- توان

```
5 + 2
5 - 2
5 * 2
5 / 2
5 // 2
5 % 2
5 ** 2
```

لیست ها

- ایندکسینگ (Indexing)
- طول لیست
- برش زدن (Slicing)
- اضافه کردن به انتها
- حذف کردن از لیست

```
list[index]
len(list)
list[start:end]
list.append(obj)
list.remove(obj)
```

کامنت ها

- تک خطی
- چند خطی

```
# This is a comment
"""multi-line"""
```

ورودی و خروجی کاربر

- ورودی کاربر
- چاپ خروجی

```
v = input("message")
print(v)
```

توابع

- تابع معمولی
- تابع لامبدا

```
def func(a, b): a+b
lambda a,b: a+b
```

PERSIANOVIN

مفاهیم پایه

پرینت کردن:

عبارت دلخواه را در خروجی چاپ میکند.

```
print(“Hello World”)
```

دریافت ورودی:

یک پیغام دلخواه در خروجی چاپ میکند و از کاربر میخواهد یک استرینگ را وارد کند.

```
input(“Your age?”)
```

کامنت گذاری:

با اضافه کردن # در ابتدای یک عبارت ایجاد میشود که پایتون آن را اجرا نمیکند.

```
#This is a comment
print(“This is code”)
```

متغیرها:

نام دلخواهی است که به داده مورد نظر میدهیم و از طریق آن میتوان به آن داده دسترسی داشت.

```
my_name = “Sara”
my_age = 12
```

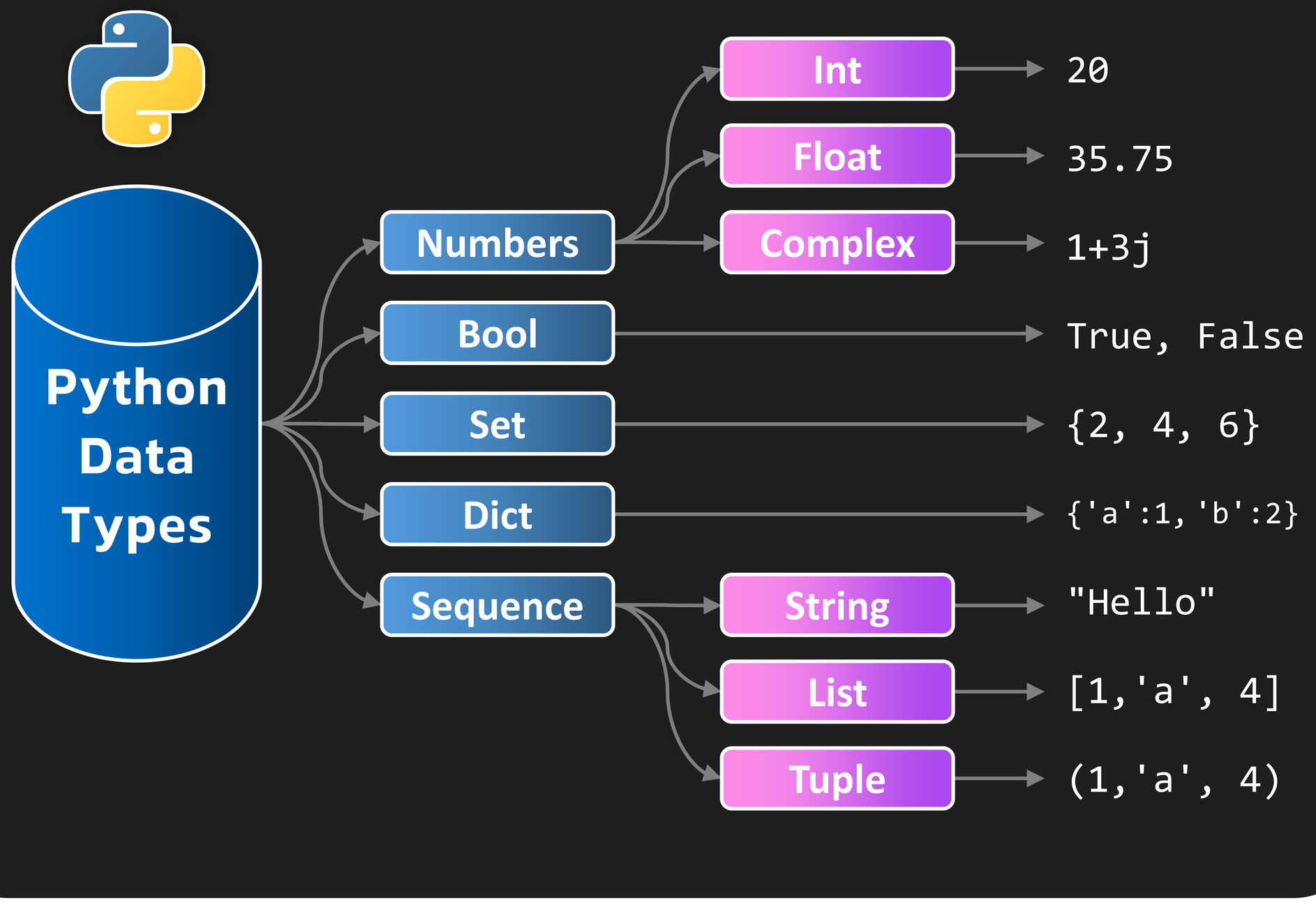
عملگر =+:

به زبان ساده میگوید: مقدار قبلی را بردار و به مقدار آن اضافه کن.
بطور مشابه میتوان از سایر عملگرهای حسابی به جای عملگر جمع استفاده کرد (مثلا =-)

```
age = 12
age += 4
#age is now 16
```

خلاصه پایتون

| انواع داده ها در پایتون که باید آنها را بشناسید:



کار با انواع داده

انواع داده:

اعداد صحیح (integer):

اعداد اعشاری (float):

اعداد مختلط (complex):

رشته ها (string):

مقدار بولی (bool):

```
my_number = 327
my_float = 3.14
My_complex = 2+3j
my_string = “Hello”
bool1 = True
bool2 = False
```

جمع کردن استرینگ ها:

استیرنگ ها وقتی جمع میشوند بهم میچسبند.

به این عمل میگویند concatenation.

```
“Hello” + “Sara”
#result: “HelloSara”
```

رد کردن کاراکتر:

عملگر \ باعث میشود کوتیشن بعد آن به عنوان کوتیشن انتهایی استرینگ در نظر گرفته نشده و مستقیما چاپ شود.

```
x = “She said \“Hi\””
print(x)
#prints: She said “Hi”
```

PERSIANOVIN

کار با انواع داده

استرینگ فرمت شده:

به شما کمک میکند تا یک مقدار دلخواه را درون یک استرینگ قرار دهید.

```
x = 7
print(f“x is {x}”)
#prints: x is 7
```

تبدیل نوع داده ها:

میتوانید یک نوع داده را به نوع دیگر تبدیل کنید.

تبدیل به float: float( )

تبدیل به integer: int( )

تبدیل به string: str( )

تبدیل به bool: bool( )

```
n = 354
new_n = float(n)
print(new_n)
#prints: 354.0
```

چک کردن نوع داده:

برای چک کردن نوع داده میتوان از تابع type استفاده کرد.

```
x = 3.14
type(x)
#result: float
```

ریاضیات

عملگرهای حسابی:

جمع کردن:
```
3 + 2   #5
```
تفریق کردن:
```
4 - 1   #3
```
ضرب کردن:
```
2 * 3   #6
```
تقسیم کردن:
```
5 / 2   #2.5
```
جز صحیح تقسیم:
```
5 // 2 #2
```
به توان رساندن:
```
5 ** 2 #25
```
محاسبه باقی مانده:
```
5 % 2   #1
```

توابع ریاضی داخل پایتون:

پیدا کردن کمترین مقدار:
```
min(5, 10, 25) #5
```
پیدا کردن بیشترین مقدار:
```
max(5, 10, 25) #25
```
محاسبه قدرمطلق:
```
abs(-7.2)      #7.2
```
به توان رساندن:
```
pow(5, 2)      #25
```
رند کردن:
```
round(3.2)     #3
```

توابع کتابخانه Math:

```
import math
```
محاسبه جذر:
```
math.sqrt(64)   #8
```
رند به بالا:
```
math.ceil(1.4)  #2
```
رند به پایین:
```
math.floor(1.4) #1
```
عدد پی:
```
math.pi         #3.14
```
توابع بیشتر در داکیومنت کتابخانه Math

خلاصه پایتون

| توابع مهم کتابخانه Math (ریاضی) در پایتون با مثال:

توابع کتابخانه ریاضی	خروجی	توضیحات
• math.ceil(1.03)	2	رند به بالا
• math.sqrt(81)	9.0	محاسبه جذر
• math.exp(2)	7.38905609893065	عدد e به توان 2
• math.fabs(-7)	7.0	محاسبه قدر مطلق
• math.floor(1.95)	1	رند به پایین
• math.log(8, 2)	3.0	لگاریتم 8 بر مبنای 2
• math.log10(100)	2.0	لگاریتم 100 بر مبنای 10
• math.pow(4, 2)	16.0	محاسبه 4 به توان 2
• math.pi	3.141592653589793	عدد پی
• math.radians(180)	3.141592653589793	تبدیل درجه به رادیان
• math.degrees(math.pi))	180.0	تبدیل رادیان به درجه
• math.sin(math.pi/2)	1.0	محاسبه سینوس پی دوم
• math.cos(math.pi)	-1.0	محاسبه کسینوس پی
• math.tan(math.pi/4)	0.1	محاسبه تانژانت پی چهارم

ارورهای مهم

Syntax Error:

زمانی ایجاد میشود که کد را با سینتکس (گرامر) اشتباه بنویسیم. مثلا وقتی برای تابع پرینت پرانتز نذاریم:

```
print “Hello”
```

```
File "test.py", line 1
    print "Hello"
    ^^^^^^^^^^^^^
SyntaxError: Missing
parentheses in call to
'print'
```

Name Error:

زمانی ایجاد میشود که متغیر، تابع یا کلاسی را فراخوانی کنیم که قبلا تعریف نشده است:

```
number = 5
Number += 1
```

نکته: پایتون به بزرگی و کوچیکی حروف حساس است.

```
File "test.py", line 2
    Number += 1
NameError: name 'Number' is
not defined
```

Zero Division Error:

زمانی ایجاد میشود که عددی را بر صفر تقسیم کنیم:

```
5 % 0
```

```
File "test.py", line 1
    5 % 0
ZeroDivisionError: integer
division or modulo by zero
```

توابع

ساخت تابع:

این اصول ساخت یک تابع بسیار ســـاده است. با کمک توابع میتوانید قطعه کـدهای دلخـواه را از طریق صدا زدن تابع به دفعات مختلف در برنامه اجرا کنید، بدون آنکه نیاز باشد آن قطعه کدها را تکرار کنید.

```
def my_function():
    name = input(“name:”)
    print(“Hello”)
    print(name)
```

صدا زدن تابع:

با نوشتن نام تابع به همراه پرانتز بعد آن میتوان آن را صـــدا زد. صـــدا زدن تابع یعنی اجـرا کردن کدهـای بدنه آن. هر بار کـه تابعی را صـدا بزنید، کدهای نوشته شده در بدنه آن اجرا میشود.

```
my_function()
my_function()
#my_function will
#run twice.
```

تابع با ورودی:

میتوانید برای توابع خـود ورودی تعـریف کنید تا موقع اجرا آن، به ازای ورودی هـای مختلف نتایج متفاوت و مطلوبی بگیرید.

```
def add(n1, n2):
    print(n1 + n2)

add(2, 3) #5
add(4, 7) #11
```

توابع

تابع با خروجی:

علاوه بر ورودی میتوانید برای توابعتون خروجی هم تعـریف کنید. خـــروجی توابـع با return مشـخص میشـود و این امکان را به شـما میدهد تا خروجـی حاصل از تابع را در یک متغیر ذخیره کنید.

```
def add(n1, n2):
    return n1 + n2

result = add(2, 3)

print(result) #5
```

محدوده متغیرها:

متغیرهایی که در محـدوده تابـع تعـریف میشـوند local هســـتند. یعـنی خـارج از تابـع در دسـترس نیستند و بعد از اجرا شدن تابع از بین میروند.

```
n = 2
def my_function():
    n = 3
    print(n)

my_function() #3
print(n) #2
```

ورودی های keyword:

موقع اجـــرا کردن توابع میتوانید ورودی های تابع را بصورت keyword وارد کنید. در این حالت مشخص میشـــود چه مقـــداری را برای هـــر پارامتر ورودی مشـــخص میکنید. و همچنین در حالت keyword ترتیب ورودی هایی که به تابع میدهید مهم نیست.

```
def divide(x, y):
    print(x / y)

divide(10, 5) #2.0
divide(x=10, y=5) #2.0
divide(y=10, x=5) #0.5
```

شرط ها

If:

با if میتوان شـرط دلخواه تعریف کرد. اگر شـرط برقرار باشـد کدهای بدنه if اجرا میشـــود. برای مثال در اینجا اگر n بزرگتر از 2 باشـد، دســـتور پرینت اجرا میشود.

```
n = 5
if n > 2:
    print(“Larger”)
#prints: Larger
```

Else:

با else میتوان مشخص کرد اگر شرط برقرار نبود چه چیزی اجرا شود.

```
age = 15
if age > 18:
    print(“Can drive”)
else:
    print(“Don’t drive”)
#prints: Don’t drive
```

Elif:

با elif میتوان علاوه بر شرط if شـــرط های دیگر تعریف کرد. در این حالت شرط ها به ترتیب از بالا به پایین چک میشـــوند و هر شـــرطی که برقرار باشد بدنه آن شرط اجرا شـده و بقیه شـــرط ها دیگر بررسی نمیشوند.

```
rating = 5
if rating > 5:
    print(“good”)
elif rating == 5:
    print(“medium”)
elif 0 < rating < 5:
    print(“bad”)
else:
    print(“very bad”)
#prints: medium
```

شرط ها

:and

اگر هر دو شـرط دو طرف آن برقرار باشـد، خروجی آن True شده و دستور آن اجرا میشود.

```
n = 3
if n > 0 and n < 5:
    print(“OK”)
#prints: OK
```

:or

اگر حـداقل یکی از شرط های دو طـرف آن برقرار باشد، خروجی آن True شـده و دســتور آن اجرا میشود.

```
age = 12
if age < 18 or age > 90:
    print(“Don’t drive”)
#prints: Don’t drive
```

:not

اگر خروجی شـرطی True باشـد آن را False، و اگر خروجی شرط False باشد آن را True میکند.

```
if not 1 > 3:
    print(“OK”)
#prints: OK
```

عملگرهای مقایسه ای:

با این عملگرها میتوان دو مقدار را مقایســه کرد. اگر عملگر برای مقادیر دو طرف آن برقرار باشـــد، خروجی عملگر True و در غیر اینصـــورت False میشود.

```
>     # بزرگتر
<     # کوچیکتر
>=    # بزرگتر مساوی
<=    # کوچکتر مساوی
==    # مساوی
!=    # نابرابر
```

حلقه های تکرار

حلقه while:

در این حلقه کدهای بدنه تا زمانی تکرار میشوند تا شرط حلقه False شود (یعنی شرط حلقه دیگر برقرار نباشد).

```
n = 1
while n < 10:
    n += 1
print(n) #10
```

حلقه for:

با حلقه for میتوان روی اعضای یک داده قابل پیمایش (مثل استرینگ، لیست، تاپل، دیکشنری و range) حرکت کرد و بدنه حلقه به تعداد اعضای داده قابل پیمایش اجرا میشود.

```
numbers = [2, 4, 3]
for number in numbers:
    print(number,end=“ ”)
#prints: 2 4 3
```

_ در حلقه for:

اگر مقداری که حلقه for در هر تکرار میخواند را نیازی ندارید، میتوانید از _ استفاده کنید. برای مثال فقط میخواهیم * را سه بار تکرار کنیم:

```
for _ in range(3):
    print(“*”, end=“”)
#prints: ***
```

break:

با تعریف شرط دلخواه و با کمک break میتوانید زودتر از حلقه خارج شوید. همچنین میتوانید برای حلقه else تعریف کنید که کدهای آن فقط زمانی اجرا میشود که حلقه با break متوقف نشود.

```
s = [2, 4, 1, 5]
for x in s:
    if x > 3:
        print(“stopped”)
        break
else:
    print(“completed”)
```

لیست ها

روش های ساخت لیست:

```
my_list1 = [1, 2, 3]
my_list2 = ["a", "b"]
my_list3 = list((1, 2))    #[1, 2]
my_list4 = list(range(3)) #[0, 1, 2]
my_list5 = list("Hi")      #['H', 'i']
```

اضافه کردن به لیست:

```
my_list = ["a", "b"]
my_list.append("c")
print(my_list)              #["a", "b", "c"]
my_list.insert(0, "#")
print(my_list)              #["#", "a", "b", "c"]
```

حذف کردن از لیست:

```
my_list = ["a", "b", "c", "d"]
my_list.remove("b")
print(my_list)              #["a", "c", "d"]
my_list.pop()
print(my_list)              #["a", "c"]
del my_list[1]
print(my_list)              #["a"]
```

لیست ها

ترکیب لیست ها:

```
my_list1 = [1, 2]
my_list2 = [3, 4]
result1 = my_list1 + my_list2
print(result1)                  #[1, 2, 3, 4]
result2 = [my_list1, my_list2]
print(result2)                  #[[1, 2], [3, 4]]
```

مرتب کردن لیست:

```
my_list = [2, 4, 1, 3]
my_list.sort()
print(my_list)                  #[1, 2, 3, 4]
my_list.sort(reverse=True)
print(my_list)                  #[4, 3, 2, 1]
```

برش زدن لیست:

```
my_list = ["a", "b", "c", "d"]
my_list[1:3]                    #['b', 'c']
```

لیست ها

خواندن و تغییر اعضای لیست:

```
my_list = ["a", "b", "c"]
my_list[1]                  #b
my_list[1] = 0
print(my_list)              #['a', 0, 'c']
```

حلقه زدن روی اعضای لیست:

```
my_list = ["a", "b"]

for item in my_list:
    print(item, end=" ")  #a b
```

با کمک تابع enumerate میتوانیم همزمان شماره اندیس اعضای لیست را هم بخوانیم:

```
for index, item in enumerate(my_list):
    print(index, item)
#0 a
#1 b
```

کپی کردن لیست:

```
my_list = ["a", "b"]
copy1 = my_list.copy()
copy2 = my_list[:]
copy3 = list(my_list)
```

لیست ها

شمارش اعضای لیست:

تابع len تعداد کل اعضای لیست و متد count تعداد یک عضو دلخواه را برمیگرداند:

```
my_list = ["a", "b", "c", "a"]
len(my_list)                #4
my_list.count("a")          #2
```

چک کردن عضویت در لیست:

متد index شماره اندیس عضو دلخواه را برمیگرداند. اگر عضو مورد نظر در لیست نباشد ارور میدهد.

```
my_list = ["a", "b", "c"]
"b" in my_list              #True
my_list.index("b")          #1
```

List Comprehension:

به تکنیک ساخت لیست در یک خط میگویند List Comprehension:

```
list1 = [i for i in range(3)]
print(list1)                    #[0, 1, 2]
list2 = [i for i in range(3) if i%2==0]
print(list2)                    #[0, 2]
```

خلاصه پایتون

| متدهای لیست در یک نگاه (توضیحات در کپشن):

متدهای لیست	تغییرات روی لیست	خروجی متد
`['A', 'B'].append('D')`	`['A', 'B', 'D']`	-
`['A', 'B'].clear()`	`[]`	-
`['A', 'B'].copy()`	-	`['A', 'B']`
`['A', 'B'].count('B')`	-	`1`
`['A', 'B'].index('A')`	-	`0`
`['A', 'B'].insert(1, 'D')`	`['A', 'D', 'B']`	-
`['A', 'B'].pop(0)`	`['B']`	`'A'`
`['A', 'B'].remove('B')`	`['A']`	-
`['A', 'B'].reverse()`	`['B', 'A']`	-
`['B', 'A'].sort()`	`['A', 'B']`	-
`['A', 'B'].extend([1])`	`['A', 'B', 1]`	-

تاپل ها

روش های ساخت تاپل:

تاپل ها مثل لیست ها هستند با این تفاوت که قابل تغییر نیستند.

```
my_tuple1 = (1, 2, 3)
my_tuple2 = ("a", "b")
my_tuple3 = tuple([1, 2])   #(1, 2)
```

ترکیب تاپل ها:

```
my_tuple1 = (1, 2)
my_tuple2 = (3, 4)
result1 = my_tuple1 + my_tuple2
print(result1)                  #(1, 2, 3, 4)
result2 = (my_tuple1, my_tuple2)
print(result2)                  #((1, 2), (3, 4))
```

مرتب کردن تاپل ها:

تاپل ها متدی برای مرتب شدن ندارند، اما با تابع sorted میتوان نسخه مرتب شده آنها را تولید کرد.

```
my_tuple = (2, 4, 1, 3)
result1 = tuple(sorted(my_tuple))
print(result1)                      #(1, 2, 3, 4)
result2 = tuple(sorted(my_tuple, reverse=True))
print(result2)                      #(4, 3, 2, 1)
```

دیکشنری ها

روش های ساخت دیکشنری:

```
my_dict1 = {"a":1, "b":2}
my_dict2 = dict(a=1, b=2)    #{'a': 1, 'b': 2}
my_dict3 = dict.fromkeys(["a", "b"], 1)
                             #{'a': 1, 'b': 1}
```

ترکیب دیکشنری ها:

```
dict1 = {"a":1, "b":2}
dict2 = {"c":3}
result1 = {**dict1, **dict2}
print(result1)               #{'a':1,'b':2,'c':3}
result2 = dict1 | dict2
print(result2)               #{'a':1,'b':2,'c':3}
dict2.update(dict1)
print(dict2)                 #{'c':3,'a':1,'b':2}
```

خواندن و تغییر دادن داده های دیکشنری:

```
my_dict = {"a":1, "b":2}
my_dict["a"]                 #1
my_dict["a"] = 7
print(my_dict)               #{"a":7, "b":2}
```

ست ها

روش های ساخت ست:

ست ها مجموعه هایی هستند که داده تکراری نمیپذیرند و تکرارها را حذف میکنند.

```
my_set1 = {1, 2, 3}
my_set2 = set([1, 2, 3, 3]) #{1, 2, 3}
my_set3 = set("Hello")      #{'H', 'o', 'l', 'e'}
```

عملیات های ریاضی با ست:

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
print(set1 - set2)          #{1, 2}
print(set1 & set2)          #{3}
print(set1 | set2)          #{1, 2, 3, 4, 5}
print(set1 ^ set2)          #{1, 2, 4, 5}
```