

Python

Get started:

پایتون زبان برنامه نویسی که از قاعده OOP پیروی میکند. که تو زمینه سرور، توسعه نرم افزار ها، ریاضیات، ... کاربرد دارد.

میتوان روی سرور ها برای ایجاد وب اپلیکیشن ها و کار با دیتا بیس ها ایجاد و تغییر فایل ها و برای هندل کردن بیگ دیتا ها و ریاضیات های پیچیده...

پایتون از نظر ساختار دستوری با زبان های دیگر تفاوت دارد، برخلاف زبان های مثل, java, c, c++، سمتیکالن ندارد و کروشه بلاک ندارد، و بر اساس فاصله تو رفتگی رفتار میکند و کد ها مشخص میشوند. پایتون به شدت نزدیک ساختار زبان انگلیسی محاوره ای است.

Syntax:

دستور زبانی پایتون با فرورفتگی مشخص میشود که دیگر بستگی به خود برنامه نویس دارد از یک فاصله تا چند تا ولی استاندارد چیزی که رعایت میشود ۴ تا فاصله یا همان یک تب است. و اینکه در حد جای کد مثل شرط ها یا حلقه ها یا کلاس ها یا فانکشن ها توى خط اول هر میزان فرورفتگی ای استفاده کردید برای باقی کد های اون مربوطه باید به همان میزان باشد. کد زیر ارور میخورد:

```
if 5 > 2 :  
    print("test")  
    print("test2")
```

Variables in python:

وریبل ها متغیر هایی هستند که آن ها رو مساوی یک مقداری قرار میدهیم. مانند اعداد طبیعی که int میگویند. در واقع متغیر ها ظرفی برای نگهداری داده های ما هستند! دستور خاصی وجود ندارد اسم وریبل علامت مساوی و مقدار.

```
x = 4
```

در پایتون نیازی نیست که هنگام ایجاد متغیر ها نوع انها رو مشخص کنیم مثلا بگوییم استرینگ هستند یا اینتجر یا دابل..... حتی ممکن است بعد از ایجاد و مقدار دهی یک متغیر که مثل اینتجر بوده نوع آن بعدا تغییر کند و استرینگ شود.

```
x = 4  
"x = "String
```

ولی اگر بخواهیم که متغیری رو مشخص کنیم دقیق از چه نوعی است یا به چه نوعی میخواهیم تغییرش دهیم باید از Casting استفاده کنیم

```
x = str(3)  
y = int(3)
```

اگر خواستیم تایپ یک متغیر رو دریافت کنیم که از چه تایپی است از Type() استفاده میکنیم.

```
print(type(x))
```

توى متغیرهای استرینگ تفاوتی نداره که دابل کوت استفاده بشه یا سینگل کوت اما برای اینکه در پسری از جملات از سینگل کوت استفاده میشه و برای اینکه سیستم قاطی نکنه از دابل کوت استفاده میشه معمولا

```
x = "i'm writing the test sentence"
```

نکته ای که وجود داره اینه که متغیر ها در پایتون به بزرگ یا کوچیکی حروف حساس هستند.
یعنی این قطعه دو تا متغیر جدا از هم ایجاد میکند:

```
a = 4  
A = "Amir"
```

Variable Names:

برای تعریف متغیر های پایتون یکسری قوانین وجود داره که باید رعایت بشوند:

۱. متغیر ها در پایتون باید با یک حرف یا _ شروع شود.
۲. متغیر ها در پایتون نمیتوانند با عدد شروع شوند.
۳. متغیر ها در پایتون فقط میتوانند شامل اعداد و حروف و اندرلاین شوند.
۴. متغیر ها در پایتون به کوچک و بزرگ حساس اند یعنی (این سه تا متغیر با یکدیگر تفاوت دارند age, Age, AGE)

۵. متغیر ها در پایتون نمیتوانند جز حروف های کلیدی باشند.
برای خوانا تر شدن کد و زیبایی کد سه روش برای نوشتن متغیر ها وجود داره:
camel case

به این صورت است که تمام کلمات به جز کلمه اول حرف اولشون با حرف بزرگ نوشته شود. مثال:
myVariableExample

Pascal Case

به این صورت است که تمام کلمات حرف اول انها با حرف بزرگ نوشته شود. مثال:
MyVariableExample

Snake Case

به این صورت است که تمام کلمات با _ از یکدیگر جدا میشوند. مثال:

در پایتون برای مقدار دهی متغیر ها نیازی نیست هر متغیر را در یک خط مقدار دهی کنید میتوانید به میزانی که دارید متغیر در یک خط تعریف میکنید بهش مقدار هم بدھید مثال:

```
x, y, z = "apple", "orange", "juice"
```

نکته: حواستان باشد به میزانی که متغیر وجود دارد به همان میزان باید مقدار هم وجود داشته باشد.

یا میتوانید به چند متغیر یک مقدار یکسان بدھید:

```
a = b = c = "blue"
```

Unpack a Collection:

اگر شما یه لیست تاپل داشته باشید در پایتون قابلیتی وجود دارد که به شما اجازه میدهد متغیر ها را به مقدار های داخل لیست تاپل مقدار دهی کنید مثال:

```

fruits = [ "orange" , "apple" , "cherry" ]
x =y =z = fruits
print(x)
print(y)
print(z)

```

و به ترتیب لیست متغیرها مقدار دهی شدند و چاپ میشوند.

Output Variables:

داخل تابع print() میتوانید متغیرها را چاپ کنید. برای چاپ متغیرها مختلف در کنار یکدیگر در یک تابع پرینت از ، باید استفاده کنید. مثال:

```

x = "python"
y = "is "
z = "greate"
print(x, y, z)

```

همچنان میشود با + هم چاپ کرد

```
print(a + y + z)
```

و باید توجه داشت که اگر جای استرینگ، اعداد بودند مثل ریاضی عمل میکرد و جمع میکرد اگر تفرقی باشد کسر میکرد.

نکته دیگری که وجود داره اینه که شما نمیتونی یه استرینگ و یک عدد رو با علامت + چاپ کنی و به خطابرمیخوری باید یا از کاما استفاده کنی که مهم نیست چه دیتا تایپ هایی رو داری با هم دیگه چاپ میکنی یا باید کست کنی توی تابع پرینت.

و نکته اخر اینکه وقتی با کاما چاپ میکنی استرینگ ها به یکدیگر نمیچسبند و یه فاصله دارند حتی اگر شما در نظر نگیرید اما اگر با علامت + چاپ کنید به یکدیگر میچسبند اگر در نظر نگیرید

Global Variables:

متغیرهایی که بیرون از توابع تعریف شوند را متغیرهای گلوبال میگویند و هم داخل تابع هم خارج از تابع قابل استفاده هستند.

نکته ای که وجود داره اینکه اگر شما یک متغیر به همان نام متغیر گلوبال ایجاد کنید داخل تابع تون، اون متغیر لوکال نامیده میشه و فقط داخل همان تابع اعتبار دارد و بیرون از آن تابع آن متغیر همون گونه میماند.

اگر بخواهید داخل یک تابع یک متغیر گلوبال تعریف کنید باید از کلید واژه global و استفاده کنید.

```

def myfunc():
    global x
    x = "fantastic"

myfunc()

```

```
print("python is " + x)
```

و همچنین اگر خواستید یک متغیر گلوبال رو داخل یک تابع عوض کنید مقدارش را از کلید واژه global استفاده کنید و مقدار جدید بهش بدهید.

Comments in python:

کامنت های برای خوانایی بیشتر بخشی از کد هستند که توضیحات اضافه کد را داخل بخش کامنت مینویسم تا اگر نفر بعدی یا ماه ها یا سال ها بعد اون قطعه کد همچنان خوانا باشد.

کامنت ها با علامت # شروع میشوند.
همچنین برای استفاده از کامنت ها توى چندین خط میتواند با تغريف سه تا کوت داخل ان بنویسید کامنت خود را:

```
"""
this is a sample comment!
"""
#another one!
```

با این صورت.
تا زمانی که استرینگ رو به وریبل ربط نديم پایتون اون رو میخونه اما نادیده میگیره.

Data Types:

نوع های مختلف دیتا در پایتون:

Text Type: str
Numeric Types: int, float, complex
Sequence Types: list, tuple, range
Mapping Type: dict
Set Types: set, frozenset
Boolean Type: bool
Binary Types: bytes, bytearray, memoryview
None Type: NoneType

نکته:
ترکیبی از اعداد و حروف است: complex

در پایتون برای مشخص کردن اینکه یک متغیر از چه تایپ است میتوان از تابع type استفاده کرد. مثال:
print(type(x))

Setting the Specific Data Type

Example

```
Data Type
x = str("Hello World")
str
x = int(20)
int
x = float(20.5)
float
x = complex(1j)
complex
x = list(("apple", "banana", "cherry"))
list
```

```

x = tuple(("apple", "banana", "cherry"))
tuple
x = range(6)
range
x = dict(name="John", age=36)
dict
x = set(("apple", "banana", "cherry"))
set
x = frozenset(("apple", "banana", "cherry"))
frozenset
x = bool(5)
bool
x = bytes(5)
bytes
x = bytearray(5)
bytearray
x = memoryview(bytes(5))
memoryview

```

Python Numbers:

سه گروه برای اعداد در پایتون وجود دارد : int, float , complex

دیدیم که چگونه میتوان تایپ انها را دیدی با تابع تایپ. وقتی یک متغیر را از نوع int تعریف میکنیم یعنی میتواند عدد منفی باشد یا مثبت یا به رقم های زیاد باشد بدون اعشار وقتی میگوییم عدد از جنس float است یعنی میتواند اعشار منفی یا مثبت باشد با هر رقم اعشار همچنین در استفاده از float ها میتوانیم از e. هم استفاده کنیم. که نشون دهنده توان ۱۰ در ان عدد است(چک شود)

وقتی هم میگوییم عدد از جنس complex است یعنی

Type Conversion:

شما میتوانید متغیری را که قبل مقدار دهی کردید با یک نوع مشخص را تغییر بدهید و اصطلاحاً کست کنید ان را. مثال

```

x = 1
y = 1j
z = 2.300

a = float(x)
b = int(z)
c = complex(x)

```

نکته: شما نمیتوانید اعدادی از جنس complex را به int, float تبدیل کنید!

Random Number:

در پایتون تابعی به اسم random (random) نداریم اما پایتون یک ماژول داخلی دارد که میتوان از آن استفاده کرد به وسیله ایمپورت کردن آن:

```
import random
```

```
print ( random.randrange(1,10))
```

Python Casting:

راجب کست کردن قبل از صحبت کردیم و گفتیم برای تبدیل دیتا تایپ هامون از این استفاده میکنیم و نوع استفاده شم هم توضیح دادیم اما این زیر باز هم مثال میزنیم تا متوجه شویم.

-int() - constructs an integer number from an integer literal, a float literal (by removing all decimals), or a string literal (providing the string represents a whole number)
-float() - constructs a float number from an integer literal, a float literal or a string literal (providing the string represents a float or an integer)
-str() - constructs a string from a wide variety of data types, including strings, integer literals and float literals

```
x = int(1) # x will be 1
y = int(2.8) # y will be 2
z = int("3") # z will be 3

x = float(1)      # x will be 1.0
y = float(2.8)    # y will be 2.8
z = float("3")    # z will be 3.0
w = float("4.2")  # w will be 4.2

x = str("s1") # x will be 's1'
y = str(2)     # y will be '2'
z = str(3.0)   # z will be '3.0'
```

Python Strings:

تویی پایتون سینگل کوتیشن با دابل کوتیشن تفاوتی ندارن و میتوانیم از انها در داخل یکدیگر استفاده کنیم. همچنین میتوانیم از استرینگ داخل تابع پرینت استفاده کنیم به راحتی.

برای قرار دادن چند خط استرینگ داخل یک متغیر از سه تا کوت استفاده میکنیم. مثال:

```
print("It's alright")
print("He is called 'Johnny'")
print('He is called "Johnny"')

multiline Strings:
a = """Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua."""
print(a)
```

یا بجای دابل کوت میتوانید از سینگل کوت استفاده کنید. و به همان گونه که وارد شده اند چاپ

میشوند.

در پایتون چیزی به اسم `char` نداریم. تک حرف همان رشته‌ای است که فقط یه ایندکس دارد. و برای چاپ یه ایندکس خاص از یک رشته میتوان از این استفاده کرد:

```
a = "Hello"
```

```
print(a[2])
```

و میتوانیم از حلقه‌ها داخل رشته‌ها استفاده کنیم، مثلاً تک تک حروف یک رشته رو داخلش سرچ کنیم یا چاپش کنیم.

```
for x in "banana":
```

```
    print(x)
```

و این به ما تک تک حروف رو در یک خط چاپ میکنه.

برای چاپ میزان طول یک رشته یا کلا استفاده از طول یک رشته از تابع `len()` استفاده میکنیم.

```
a = "Hello, World!"
```

```
print(len(a))
```

و همچنین میتوانیم چک کنیم که یک حرف خاص یا یک کلمه خاص در یک رشته وجود دارد یا خیر از طریق تابع `in`. اگر وجود داشته باشد به صورت بولین به ما برگردانده میشود.

```
txt = "The best things in life are free!"
```

```
print("free" in txt)
```

یا

```
txt = "The best things in life are free!"
```

```
if "free" in txt:
```

```
    print("Yes, 'free' is present.")
```

به همین صورت اگر بخواهیم چک کنیم که کلمه خاصی یا حرف خاصی وجود ندارد!! میتوانیم از تابع `not in` استفاده کنیم.

```
txt = "The best things in life are free!"
```

```
print("expensive" not in txt)
```

یا

```
txt = "The best things in life are free!"
```

```
if "expensive" not in txt:
```

```
    print("No, 'expensive' is NOT present.")
```

Slicing:

در پایتون میتوانیم با بخش بخش کردن یک رشته یه بخش خاصی از ان رشته را بازگردانیم به این صورت:

```
b = "Hello, World!"
```

```
print(b[2:5])
```

به این صورت است که عدد اول نشانه خانه شروع و عدد اخر نشانه خانه اخر است که خودش را شامل

نمیشود..!!

اگر عدد اول را نزاریم یعنی از اول شروع کن از ایندکس صفر.

```
b = "Hello, World!"  
print(b[:5])
```

اگر خانه اخر را خالی بگذاریم یعنی تا اخر کات کن
(برعکس اول)

```
b = "Hello, World!"  
print(b[2:])
```

نکته: در پایتون خلاف زبان های دیگر وقتی ایندکس منفی بدھیم یعنی از اخر لیست شروع کن و کار رو انجام بده و نکته مهم تر این است که خلاف اینکه ایندکس ها از اول از صفر شروع میشوند از اخر از منفی یک شروع میشوند.

```
b = "Hello, World!"  
print(b[-5:-2])
```

در اینجا ما داریم میگوییم که از پنجمین خونه از اخر تا دومین خونه از اخر چاپ کن. که orl رو چاپ میکند.

یکی دیگ از متدهایی که در استرینگ ها بکار میروند متدهای upper() و lower() هستند.

متدهای تمام رشته را به صورت حروف بزرگ بازمیگرداند و لور هم برعکس آن

```
a = "Hello, World!"  
print(a.upper())
```

```
a = "Hello, World!"  
print(a.lower())
```

و یکی از دیگر از متدهای استرینگ whitespace است که فضای خالی قبل و یا بعد از جمله رو از بین میبرد.

به این صورت:

```
a = " Hello, World! "  
print(a.strip()) # returns "Hello, World!"
```

و متدهای replace() و split() هم به ترتیب برای جایگزین کردن یک حرف خاص با یک حرف جدید در کل رشته است و اسپلیت هم بنا به کاراکتری که شما به ان میدهید از ان به بعد را به یک لیست و دو ایندکس جدا میکند:

```
a = "Hello, World!"  
print(a.replace("H", "J"))
```

9

```
a = "Hello, World!"  
print(a.split(",")) # returns ['Hello', ' World!']
```

String Concatenation:

برای چسباندن دو رشته به یکدیگر میتوانید از علامت + استفاده کنید. و برای گذاشتن فاصله بین انها

میتوانید از کوتیشن خالی استفاده کنید.

```
a = "Hello"  
b = "World"  
c = a + b  
print(c)
```

```
a = "Hello"  
b = "World"  
c = a + " " + b  
print(c)
```

F-String:

در پایتون دیدیم که نمیتوانیم رشته ها و اعداد یا دیگر دیتا تایپ ها رو به یکدیگر به وسیله + بچسبانیم و خطای میخورد. برای اینکار ما از F-String ها استفاده میکنیم.

برای استفاده قبل از رشته f میگذاریم و داخل رشته داخل برآخت ها {} متغیر را میگذاریم یا عمل ریاضیاتمان را داخل آن انجام میدهیم. و همچنین میتوانیم مقدار اعشار آن را تعیین کنیم و....

```
age = 36  
txt = f"My name is John, I am {age}"  
print(txt)
```

```
price = 59  
txt = f"The price is {price} dollars"  
print(txt)
```

```
price = 59  
txt = f"The price is {price:.2f} dollars"  
print(txt)
```

به {} placeHolder گفته میشود.

```
txt = f"The price is {20 * 59} dollars"  
print(txt)
```

Escape Character:

برای استفاده از دابل کوت، سینگل کوت، رفتن خط بعدی..... از کاراکتر بک اسلش استفاده میکنیم \
داخل رشته ها
به این صورت:

```
txt = "We are the so-called \"Vikings\" from the north."
```

انواع آن:

Code	Result
\'	Single Quote
\\"	Backslash
\n	New Line
\r	Carriage Return
\t	Tab

\b	Backspace
\f	Form Feed
\ooo	Octal value
\xhh	Hex value

String Methods:

Method	Description
<u>capitalize()</u>	Converts the first character to upper case
<u>casefold()</u>	Converts string into lower case
<u>center()</u>	Returns a centered string
<u>count()</u>	Returns the number of times a specified value occurs in a string
<u>encode()</u>	Returns an encoded version of the string
<u>endswith()</u>	Returns true if the string ends with the specified value
<u>expandtabs()</u>	Sets the tab size of the string
<u>find()</u>	Searches the string for a specified value and returns the position of where it was found
<u>format()</u>	Formats specified values in a string
<u>format_map()</u>	Formats specified values in a string
<u>index()</u>	Searches the string for a specified value and returns the position of where it was found
<u>isalnum()</u>	Returns True if all characters in the string are alphanumeric
<u>isalpha()</u>	Returns True if all characters in the string are in the alphabet
<u>isascii()</u>	Returns True if all characters in the string are ascii characters
<u>isdecimal()</u>	Returns True if all characters in the string are decimals
<u>isdigit()</u>	Returns True if all characters in the string are digits
<u>isidentifier()</u>	Returns True if the string is an identifier
<u>islower()</u>	Returns True if all characters in the

	string are lower case
<u>isnumeric()</u>	Returns True if all characters in the string are numeric
<u>isprintable()</u>	Returns True if all characters in the string are printable
<u>isspace()</u>	Returns True if all characters in the string are whitespaces
<u>istitle()</u>	Returns True if the string follows the rules of a title
<u>isupper()</u>	Returns True if all characters in the string are upper case
<u>join()</u>	Joins the elements of an iterable to the end of the string
<u>ljust()</u>	Returns a left justified version of the string
<u>lower()</u>	Converts a string into lower case
<u>lstrip()</u>	Returns a left trim version of the string
<u>maketrans()</u>	Returns a translation table to be used in translations
<u>partition()</u>	Returns a tuple where the string is parted into three parts
<u>replace()</u>	Returns a string where a specified value is replaced with a specified value
<u>rfind()</u>	Searches the string for a specified value and returns the last position of where it was found
<u>rindex()</u>	Searches the string for a specified value and returns the last position of where it was found
<u>rjust()</u>	Returns a right justified version of the string
<u>rpartition()</u>	Returns a tuple where the string is parted into three parts
<u>rsplit()</u>	Splits the string at the specified separator, and returns a list
<u>rstrip()</u>	Returns a right trim version of the

	string
<u>split()</u>	Splits the string at the specified separator, and returns a list
<u>splitlines()</u>	Splits the string at line breaks and returns a list
<u>startswith()</u>	Returns true if the string starts with the specified value
<u>strip()</u>	Returns a trimmed version of the string
<u>swapcase()</u>	Swaps cases, lower case becomes upper case and vice versa
<u>title()</u>	Converts the first character of each word to upper case
<u>translate()</u>	Returns a translated string
<u>upper()</u>	Converts a string into upper case
<u>zfill()</u>	Fills the string with a specified number of 0 values at the beginning

Python Booleans:

Boolean Values:

یک نوع دیتا تایپ است که دو نتیجه True, False را برمیگرداند. وقتی در توابع استفاده میکنید مثلا در شرطی مقدار بولین برمیگردد. و یا از طریق تابع bool() میتوانید برای متغیر ها استفاده کنید. و تقریبا هر چیزی که به ارزشی داشته باشه درست است. مثال: تمام اعداد به جز عدد صفر درست اند. تمام رشته ها درست اند. تمام لیست ها تاپل ها سرت ها دیکشنری ها درست اند مگر انکه خالی باشند!

```
bool("abc")
bool(123)
bool(["apple", "cherry", "banana"])
#All return True
```

عدد صفر، مقدار ارزش False، یا {}، []، ()، "" همگی False هستند.

```
bool(False)
bool(None)
bool(0)
bool("")
bool(())
bool([])
bool({})
```

شما میتوانید تابعی بسازید که مقداری که boolean میکند return میکند باشد.

```
def myFunction():
    return True
```

```

print(myFunction())

#####
def myFunction() :
    return True

if myFunction():
    print("YES!")
else:
    print("NO!")

```

پایتون مازول های داخلی زیادی دارد که جوابی که ریترن میکنند از نوع بولین هست مثل تابع `isinstance()` که چک میکند متغیر خاص از جنس دیتایی که ما بهش میگیم هست یا نه

```

x = 200
print(isinstance(x, int))

```

Python Operators:

در پایتون ما یکسری عملگر ها داریم که به دسته ها مختلف تبدیل میشوند:

Python divides the operators in the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Identity operators
- Membership operators
- Bitwise operators

Python Arithmetic Operators:

عملگر های ریاضیاتی هستند که روی اعداد صورت میگیرند.

Operator	Name	Example
+	Addition	$x + y$
-	Subtraction	$x - y$
*	Multiplication	$x * y$
/	Division	x / y
%	Modulus	$x \% y$
**	Exponentiation	$x ** y$
//	Floor division	$x // y$

Python Assignment Operators:

عملگر های برای انجام و مقدار دهن متغیر ها 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
:=	print(x := 3)	x = 3 print(x)

Python Comparison Operators:

برای مقایسه دو مقدار اند:

Operator	Name	Example
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Python Logical Operators:

برای ادغام شرط ها استفاده میشوند:

Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)

Python Identity Operators:

برای قیاس ابجکت ها است نه اینکه یکی هستند بلکه هویتیشن یکیه و به یکجا در حافظه اشاره میکنند!!

Operator	Description	Example
is	Returns True if both variables are the same object	x is y
is not	Returns True if both variables are not the same object	x is not y

Python Membership Operators:

برای اینکه ببیند ایا یک مقدار خاص در یک ابجکت خاص وجود دارد یا خیر:

Operator	Description	Example
in	Returns True if a sequence with the specified value is present in the object	x in y
not in	Returns True if a sequence with the specified value is not present in the object	x not in y

Python Bitwise Operators:

برای قیاس اعداد باینری استفاده میشوند:

Operator	Name	Description	Example
&	AND	Sets each bit to 1 if both bits are 1	x & y
	OR	Sets each bit to 1 if one of two bits is 1	x y
^	XOR	Sets each bit to 1 if only one of two bits is 1	x ^ y
~	NOT	Inverts all the bits	~x

<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off	x << 2
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off	x >> 2

Operator Precedence:

بحث اولویت ها داخل عملگر ها در پایتون مثل جاهای دیگ است:
نکته اگر دو عملگر با دارای اولویت یکسان وجود داشت به ترتیب از سمت چپ اولی اجرا میشود و بعد دومی.

Operator	Description
()	Parentheses
**	Exponentiation
+x -x ~x	Unary plus, unary minus, and bitwise NOT
* / // %	Multiplication, division, floor division, and modulus
+ -	Addition and subtraction
<< >>	Bitwise left and right shifts
&	Bitwise AND
^	Bitwise XOR
	Bitwise OR
== != > >= < <= is is not in not in	Comparisons, identity, and membership operators
not	Logical NOT
and	AND
or	OR

Python Lists:

در پایتون لیست ها اینگونه تعریف میشوند:

```
myList = [ "apple", "banana", "cherry"]
print(myList)
```

لیست ها تهیده شدنبرای نگهداری چندین مقدار در یک متغیر. ۴ نوع مازول داخلی برای نگهداری چندین دیتا در یک متغیر استفاده میشود. لیست ها یکی از اون ۴ مدل هستند ۳ تای دیگر به ترتیب تاپل، دیکشنری، ست هستند که هر کدام خاصیت های مشخص خود را دارند که بعداً به آن ها میپردازیم.

Python Collections (Arrays)

There are four collection data types in the Python programming language:

- **List** is a collection which is ordered and changeable. Allows duplicate members.
- **Tuple** is a collection which is ordered and unchangeable. Allows duplicate members.
- **Set** is a collection which is unordered, unchangeable*, and unindexed. No duplicate members.
- **Dictionary** is a collection which is ordered** and changeable. No duplicate members.

نکته هایی راجب لیست ها: لیست ها میتوانند مقدار های تکراری را نگذارند، قابل تغییر هستند، و به ترتیب هستند. لیست ها از هر جنس دیتا تایپی می توانند باشند.
از تابع `len()` برای طول لیست استفاده میکند. میتوانید از تابع `type()` برای تشخیص کالکشن نگهداری استفاده کنید.
برای اینکه یک لیست را مثل یک لیست چاپ کنید میتوانید از تابع `list()` استفاده کنید.

```
thislist = ["apple", "banana", "cherry"]
print(thislist)
#print the list items

thislist = list(("apple", "banana", "cherry")) # note the
double round-brackets
print(thislist)
```

```
thislist = ["apple", "banana", "cherry"]
print(len(thislist))
#print list length
```

نکته: از پایتون ۳.۶ به قبل دیکشنری ها به ترتیب نبودند اما از پایتون ۳.۷ به بعد دیکشنری ها به ترتیب هستند.

Access List Items:

ایتم های یک لیست از ایندکس صفر شروع میشوند. شما میتوانید از طریق ایندکس های یک لیست به ایتم های ان دسترسی داشته باشید. و قبل از گفتیم که با استفاده از ایندکس های منفی شما میتوانید از آخر به لیست خود دسترسی داشته باشید.

```
thislist = ["apple", "banana", "cherry"]
```

```

print(thislist[1])

thislist = ["apple", "banana", "cherry"]
print(thislist[-1])

شما می‌توانید رنج مشخص کنید از چه ایندکسی تا چه ایندکسی لیست شما رو برگرداند که با اینکار
لیستی که بازگردانده می‌شود یک لیست جدید با ایتم هایی که مشخص کردید است.

thislist =
["apple", "banana", "cherry", "orange", "kiwi", "melon",
"mango"]
print(thislist[2:5])

thislist =
["apple", "banana", "cherry", "orange", "kiwi", "melon",
"mango"]
print(thislist[:4])

thislist =
["apple", "banana", "cherry", "orange", "kiwi", "melon",
"mango"]
print(thislist[2:])

thislist =
["apple", "banana", "cherry", "orange", "kiwi", "melon",
"mango"]
print(thislist[-4:-1])

```

به همین صورت اگر رنج اول رو مشخص نکنیم به صورت پیش فرض از ایندکس صفر لیست شروع می‌شود تا جایی که مشخص شده است و یا اگر رنج اخر رو مشخص نکنیم از ایندکسی که مشخص شده است شروع می‌شود و تا اخر لیست ادامه پیدا می‌کند.

اگر هم بخواهیم از اخر لیست شروع کند میتوانیم از اعداد منفی استفاده کنیم.

و همانند رشته ها میتوانیم از تابع `in` استفاده کنیم و بررسی کنیم که ایا مقدار خاصی که مد نظر داریم در لیست وجود دارد یا خیر.

```

thislist = ["apple", "banana", "cherry"]
if "apple" in thislist:
    print("Yes, 'apple' is in the fruits list")

```

Change List Items:

برای تغییر یک ایتم خاص از طریق ایندکس ان باید اقدام کنیم.

```

thislist = ["apple", "banana", "cherry"]
thislist[1] = "blackcurrant"
print(thislist)

```

همچنین میتوانیم یک رنج رو مشخص کنیم و تمام کلمات داخل ان رنج را تغییر دهیم و به ترتیب مقدار های مشخص شده برای جایگزینی این عمل اتفاق میافتد.

```
thislist =  
["apple", "banana", "cherry", "orange", "kiwi", "mango"]  
thislist[1:3] = ["blackcurrant", "watermelon"]  
print(thislist)
```

اگر بیشتر از دامنه ای که تعریف کرده اید مقدار بدید برای جایگزینی ادامه مقدارهایی که مشخص کرده اید در ادامه هم رنج می‌ایند و بقیه ایتم‌ها در ادامه‌ان. و قاعده‌تا طول لیست تغییر میکند.

```
thislist = ["apple", "banana", "cherry"]  
thislist[1:2] = ["blackcurrant", "watermelon"]  
print(thislist)
```

و اگر کمتر از رنج مشخص شده ایتم بدھید آن‌هایی که هستند جایگزین می‌شوند و بقیه کلماتی که داخل رنج بوده‌اند و باید جایگزین می‌شدند ولی مقداری داده نشده حذف می‌شوند و بقیه در ادامه‌ان می‌ایند.

```
thislist = ["apple", "banana", "cherry"]  
thislist[1:3] = ["watermelon"]  
print(thislist)
```

اگر بخواهیم در ایندکس مشخصی یک ایتم را اضافه کنیم و نه ان را جایگزین کنیم! برای اینکار از تابع insert() استفاده می‌کنیم.

```
thislist = ["apple", "banana", "cherry"]  
thislist.insert(2, "watermelon")  
print(thislist)
```

Add List Items:

برای اضافه کردن ایتم به لیست‌ها چند راه داریم، یکی از راه‌ها استفاده از تابع append() است برای اضافه کردن یک ایتم به آخر یک لیست است.

```
thislist = ["apple", "banana", "cherry"]  
thislist.append("orange")  
print(thislist)
```

و راجب تابع insert() هم که صحبت کردیم. که در ایندکس خاص ایتم خاصی را اضافه می‌کنیم. و جایگزین نمی‌شود!!

```
thislist = ["apple", "banana", "cherry"]  
thislist.insert(1, "orange")  
print(thislist)
```

و یکی از راه‌های دیگری که داریم اینه که ما میتوانیم ایتم‌های موجود در ۴ نوع لیستی که داریم رو به لیست خود اضافه کنیم با استفاده از تابع extend()

```
thislist = ["apple", "banana", "cherry"]
tropical = ["mango", "pineapple", "papaya"]
thislist.extend(tropical)
print(thislist)
```

با این کار ما داریم تمام ایتم های لیست تروپیکال رو به اخر لیست دیس لیست اضافه میکنیم.

Remove List Items:

برای حذف یک مقدار خاص از تابع remove() استفاده میکنیم. و اگر از ان مقدار خاص چند تا بود اولین به ترتیب از اول لیست پاک میشود.

```
thislist = ["apple", "banana", "cherry", "banana", "kiwi"]
thislist.remove("banana")
print(thislist)
```

برای پاک کردن یک ایندکس مشخص از pop() استفاده میکنیم. اگر ایندکس رو مشخص نکنیم به صورت پیش فرض اخیرین ایتم رو پاک میکند.

```
thislist = ["apple", "banana", "cherry"]
thislist.pop(1)
print(thislist)
```

همچنین از کلید واژه del هم میتوان برای پاک کردن یک ایتم لیست به کار برد یا کلا برای پاک کردن کل لیست

```
thislist = ["apple", "banana", "cherry"]
del thislist[0]
print(thislist)
```

```
thislist = ["apple", "banana", "cherry"]
del thislist
```

برای خالی کردن یک لیست هم از clear() استفاده میشود. لیست همچنان وجود دارد اما خالی است.

```
thislist = ["apple", "banana", "cherry"]
thislist.clear()
print(thislist)
```

Loop Lists:

میتوانیم از حلقه ها داخل لیست ها استفاده کنیم برای دسترسی به کل ایتم های یک لیست

```
thislist = ["apple", "banana", "cherry"]
for x in thislist:
    print(x)
```

```
thislist = ["apple", "banana", "cherry"]
for i in range(len(thislist)):
    print(thislist[i])
```

یا از حلقه while استفاده کنیم:

```
thislist = ["apple", "banana", "cherry"]
i = 0
while i < len(thislist):
    print(thislist[i])
    i = i + 1
```

یک روش دیگر که کوتاه تر و اسان تر است برای دسترسی به لیست ها وجود دارد.

```
thislist = ["apple", "banana", "cherry"]
[print(x) for x in thislist]
```

کوتاه شده for

Sort Lists:

برای مرتب سازی یک لیست چند متده وجود دارد که میتوانیم از آن ها استفاده کنیم. یه صورت پیش فرض متده sort() دارای حساسیت بزرگ و کوچکی است. یعنی حروف بزرگ رو قبل از کوچک مرتب میکند. برای اینکه بتوانید طبق خواسته تان مرتب سازی رو انجام دهید میتوانید از تابع های داخلی پایتون استفاده کنید.

```
thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]
thislist.sort()
print(thislist)
```

```
thislist = ["banana", "Orange", "Kiwi", "cherry"]
thislist.sort(key = str.lower)
print(thislist)
```

با استفاده از این تابع به حروف الفبا به صورت ترتیب الفبا مرتب میشوند و اعداد هم به ترتیب بزرگ و کوچکی.

```
thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]
thislist.sort(reverse = True)
print(thislist)
```

با ایتفاذه از این کلید واژه برای مرتب سازی همچنین میتوانیم نوع مرتب سازی خود را کاستوم کنیم. یعنی یک تابعی رو تعریف کنیم به عنوان نوع مرتب سازی و از آن داخل متده مرتب سازی استفاده کنیم.

```
def myfunc(n):
    return abs(n - 50)

thislist = [100, 50, 65, 82, 23]
thislist.sort(key = myfunc)
print(thislist)
```

و تابع () reverse() برعکس میکند.

```
thislist = ["banana", "Orange", "Kiwi", "cherry"]
thislist.reverse()
print(thislist)
```

Copy Lists:

برای کپی کردن یک لیست نمیتوانید صرفاً آن را مساوی یک لیست دیگر قرار دهید چون در واقع دارید به همون لیست اشاره میکنید و هویت آن در حافظه رو در بر میگیرد و یعنی هر تغییری در لیست ۱ انجام شود در لیست ۲ هم اعمال میشود. برای کپی کردن لیست ها چند متده داریم که تابع های داخلی خود پایتون هستند.

```
thislist = ["apple", "banana", "cherry"]
mylist = thislist.copy()
print(mylist)
```

```
thislist = ["apple", "banana", "cherry"]
mylist = list(thislist)
print(mylist)
```

slice Operator:

```
thislist = ["apple", "banana", "cherry"]
mylist = thislist[:]
print(mylist)
```

Join Lists:

برای اضافه کردن دو لیست به یکدیگر یا بیشتر میتوان چند راه را استفاده کرد. راه اول که ساده ترین آن است استفاده از + است که بسیار ساده است. یک راه دیگر استفاده از حلقه for است برای اضافه کردن تک تک ایتم های یک لیست به لیست دیگر. و راه اخر استفاده از extend() است.

```
list1 = ["a", "b", "c"]
list2 = [1, 2, 3]
```

```
list3 = list1 + list2
print(list3)
```

```
list1 = ["a", "b" , "c"]
list2 = [1, 2, 3]
```

```
list1.extend(list2)
print(list1)
```

```

list1 = ["a", "b" , "c"]
list2 = [1, 2, 3]

for x in list2:
    list1.append(x)

print(list1)

```

List Methods:

نگاه کلی به متدهای لیست ها:

Method	Description
<u>append()</u>	Adds an element at the end of the list
<u>clear()</u>	Removes all the elements from the list
<u>copy()</u>	Returns a copy of the list
<u>count()</u>	Returns the number of elements with the specified value
<u>extend()</u>	Add the elements of a list (or any iterable), to the end of the current list
<u>index()</u>	Returns the index of the first element with the specified value
<u>insert()</u>	Adds an element at the specified position
<u>pop()</u>	Removes the element at the specified position
<u>remove()</u>	Removes the item with the specified value
<u>reverse()</u>	Reverses the order of the list
<u>sort()</u>	Sorts the list

Python Tuples:

یک نوع دیگری از کالکشن ها tuple است که در پایتون استفاده میشود. ما در مبحث قبلی با لیست ها آشنا شدیم. تاپل ها غیر قابل تغییر هستند و با پرانتز تعریف میشوند. در تاپل ها اجازه داده تکراری وجود دارد. برای طول تاپل همانند لیست ها از len() استفاده میکنیم. اگر میخواهید تاپل رو با یک ایتم مختصی بسازید باید بعد از آن یک ، بزارید و گرنه پایتون آن را استرینگ میشناسد. در تاپل ها دیتا تایپ های مختلفی میتوانید استفاده کنید. و از type() میتوانید برای تشخیص نوع آن استفاده کنید. برای چاپ تاپل به صورت تاپل میتوانید از tuple استفاده کنید.

```
thistuple = ("apple", "banana", "cherry")
print(thistuple)
```

```
thistuple = ("apple", "banana", "cherry")
print(len(thistuple))
```

```
thistuple = ("apple",)
print(type(thistuple))
```

```
#NOT a tuple
thistuple = ("apple")
print(type(thistuple))
```

```
mytuple = ("apple", "banana", "cherry")
print(type(mytuple))
```

```
thistuple = tuple(("apple", "banana", "cherry")) # note the double round brackets
print(thistuple)
```

Access Tuple Items:

برای دسترسی به تاپل ها همانند لیست ها هستند:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[1])
```

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[-1])
```

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon",
"mango")
print(thistuple[2:5])
```

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon",
"mango")
print(thistuple[:4])
```

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon",
"mango")
print(thistuple[2:])
```

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon",
"mango")
print(thistuple[-4:-1])
```

```
thistuple = ("apple", "banana", "cherry")
if "apple" in thistuple:
    print("Yes, 'apple' is in the fruits tuple")
```

Update Tuples:

تایپل ها چون غیر قابل تغییر هستند برای تغییر دادن تایپل ها باید ان ها رو به لیست برگردانیم و اعمال لیست رو انجام بدیم و دوباره ان ها رو به تایپل برگردانیم.
البته یکسری تابع ها هستند برای تایپل ها مثلًا میتوانیم دو تایپل را با + با یکدیگر جمع کنیم یعنی مقدار های موجود در دو تایپل رو یکی میکنند و یا با کلمه del کل تایپل رو پاک کنیم.

```
x = ("apple", "banana", "cherry")
y = list(x)
y[1] = "kiwi"
x = tuple(y)

print(x)
```

```
thistuple = ("apple", "banana", "cherry")
y = list(thistuple)
y.append("orange")
thistuple = tuple(y)
```

```
thistuple = ("apple", "banana", "cherry")
y = ("orange",)
thistuple += y

print(thistuple)
```

```
thistuple = ("apple", "banana", "cherry")
del thistuple
print(thistuple) #this will raise an error because the tuple no longer
exists
```

Unpack Tuples:

برای استخراج اطلاعات از تاپل ها ان ها را مساوی یک مقداری قرار میدهیم و از انها استفاده میکنیم:

```
fruits = ("apple", "banana", "cherry")

(green, yellow, red) = fruits

print(green)
print(yellow)
print(red)
```

نکته باید تعداد با تعداد مقدار ها همسان باشند اگر نباشد باید از استریکس استفاده کنید. در واقع اخرين تمام ايندکس هاي باقى مانده رو شامل ميشود. و اگر در اخري استفاده نشود و در هر کدام استفاده شود اين اتفاق برای ان مقدار میافتد.

```
fruits = ("apple", "banana", "cherry", "strawberry", "raspberry")

(green, yellow, *red) = fruits

print(green)
print(yellow)
print(red)
```

```
fruits = ("apple", "mango", "papaya", "pineapple", "cherry")

(green, *tropic, red) = fruits

print(green)
print(tropic)
print(red)
```

Loop Tuples:

همانند لیست ها در تاپل ها نیز شما میتوانید از حلقه ها استفاده کنید تا به اعضای تاپل دسترسی داشته باشید:

```
thistuple = ("apple", "banana", "cherry")
for x in thistuple:
    print(x)
```

```
thistuple = ("apple", "banana", "cherry")
for i in range(len(thistuple)):
    print(thistuple[i])
```

```
thistuple = ("apple", "banana", "cherry")
i = 0
while i < len(thistuple):
    print(thistuple[i])
    i = i + 1
```

Join Tuples:

برای ادغام دو یا چند تاپل از + میتوان استفاده کرد و یا میتوان چند برابر کرد اعضای یک تاپل را:

```
tuple1 = ("a", "b", "c")
tuple2 = (1, 2, 3)
```

```
tuple3 = tuple1 + tuple2
print(tuple3)
```

```

fruits = ("apple", "banana", "cherry")
mytuple = fruits * 2

print(mytuple)

```

Tuple Methods:

برخی از متد های تاپل ها:

Method	Description
<u>count()</u>	Returns the number of times a specified value occurs in a tuple
<u>index()</u>	Searches the tuple for a specified value and returns the position of where it was found

Python Sets:

ست ها نوعی دیگر از کالکشن ها هستند که برای ذخیره اطلاعات در پایتون استفاده میشوند. ست ها غیر قابل تغییر و بدون ترتیب هستند. در ست ها شما نمیتوانید داده تکراری داشته باشید. همچنین در ست ها دادهای 1, true, 0, یکسان و false محسوب میشوند و اگر در یک ست بیانند تکراری محسوب میشوند. و به محض اینکه ست ایجاد شود اعضای آن قبل تغییر نیستند اما میتوان به آن عضو جدید اضافه کرد.

ست ها را میتوانید ریموو کنید یا اد کنید بهشون. از تابع len() در ست ها میتوانید استفاده کنید. و انواع مختلف دیتا تایپ ها رو میتوانید استفاده کنید در ست. و همچنین از تابع type() میتوانید برای تشخیص تایپ استفاده کنید. و از طریق کانسٹراکتور ست میتوانید خود ست را پرینت کنید مثل حالت سینتکسیش.

```

thisset = {"apple", "banana", "cherry"}
print(thisset)

```

```

thisset = {"apple", "banana", "cherry", True, 1, 2}

print(thisset)

```

`False` and `0` is considered the same value:

```
thisset = {"apple", "banana", "cherry", False, True, 0}  
print(thisset)
```

```
thisset = {"apple", "banana", "cherry"}  
print(len(thisset))
```

Using the `set()` constructor to make a set:

```
thisset = set(("apple", "banana", "cherry")) # note the double round-brackets  
print(thisset)
```

```
myset = {"apple", "banana", "cherry"}  
print(type(myset))
```

Access Items:

در لیست هایی از جنس ست نمی توان به طور مستقیم از طریق ایندکس ها به اعضای آن دسترسی داشت اما میتوان از طریق `in`, `loop`, `in` به آن دسترسی پیدا کرد.

```
thisset = {"apple", "banana", "cherry"}  
  
for x in thisset:  
    print(x)
```

```
thisset = {"apple", "banana", "cherry"}  
  
print("banana" in thisset)
```

```
thisset = {"apple", "banana", "cherry"}
```

```
print("banana" not in thisset)
```

Add Set Items:

برای اضافه کردن عضو جدید به لیست ست از `add()` استفاده می‌توان کرد. یا برای اضافه کردن یک لیست دیگر به لیست ست می‌توان به `update()` دسترسی داشت و مهم هم نیست جنس آن لیست از چه باشد.

```
thisset = {"apple", "banana", "cherry"}
```

```
thisset.add("orange")
```

```
print(thisset)
```

```
thisset = {"apple", "banana", "cherry"}
```

```
tropical = {"pineapple", "mango", "papaya"}
```

```
thisset.update(tropical)
```

```
print(thisset)
```

```
thisset = {"apple", "banana", "cherry"}
```

```
mylist = ["kiwi", "orange"]
```

```
thisset.update(mylist)
```

```
print(thisset)
```

Remove Set Items:

برای دیلیت کردن یک از اعضای یک لیست ست ها می‌توان از متده `remove()` استفاده کرد. و همچنین می‌توان از متده `pop()` استفاده کرد، با استفاده از این متده مقدار بازگشته چاپ شده بعد از چاپ دیلیت می‌شود از لیست و دیگر در لیست نخواهد بود. لیست ست ها رندوم هستند و هیچ ترتیبی ندارند، پس وقتی از متده `pop()` استفاده می‌کنیم نمیدانیم کدام عضو را دیلیت می‌کنم (در ایندکس چندم است). همچنین می‌توان از متده `discard()` استفاده کرد. فرق آن با متده `remove()` این است که اگر مقداری که داده

شده تا حذف شود وجود نداشته باشد، در ریموزو به اور بر می خورد اما در ان اینگونه نیست. از clear هم برای خالی کردن لیست استفاده می شود و از del برای کلا دیلیت کردن یک لیست استفاده می شود.

```
thisset = {"apple", "banana", "cherry"}  
  
thisset.remove("banana")  
  
print(thisset)
```

```
thisset = {"apple", "banana", "cherry"}  
  
thisset.discard("banana")  
  
print(thisset)
```

```
thisset = {"apple", "banana", "cherry"}  
  
x = thisset.pop()  
  
print(x)  
  
print(thisset)
```

```
thisset = {"apple", "banana", "cherry"}  
  
thisset.clear()  
  
print(thisset)
```

```
thisset = {"apple", "banana", "cherry"}  
  
del thisset  
  
print(thisset)
```

Loop Sets:

همچنین شما می‌توانید از طریق حلقه‌ها دسترسی داشته باشید به لیست ست‌ها.

```
thisset = {"apple", "banana", "cherry"}  
  
for x in thisset:  
    print(x)
```

Join Sets:

برای جوین دادن دو یا چند لیست ست با یکدیگر یا لیست‌های دیگر با لیست ست چندین راه وجود دارد.

یکی از راه‌ها استفاده از union است که با استفاده از این متده مجموع اعضای دو یا چند لیست را باهم ادعا می‌کند و در یک لیست ست جدید برمی‌گرداند. به جای کلید واژه ان می‌توان از | استفاده کرد که همان کار را می‌کند. اما نکته این است که این فقط برای لیست‌های ست است و برای تاپل به سمت یا جنس‌های دیگر باید از کلید واژه استفاده کرد.

```
set1 = {"a", "b", "c"}  
set2 = {1, 2, 3}  
  
set3 = set1.union(set2)  
print(set3)
```

```
set1 = {"a", "b", "c"}  
set2 = {1, 2, 3}  
  
set3 = set1 | set2  
print(set3)
```

برای ادغام کردن چندین لیست ست فقط کافیه بینشون کاما بزاریم یا چند تا علامت بزاریم

```
set1 = {"a", "b", "c"}  
set2 = {1, 2, 3}  
set3 = {"John", "Elena"}  
set4 = {"apple", "bananas", "cherry"}  
  
myset = set1 | set2 | set3 | set4  
print(myset)
```

```
set1 = {"a", "b", "c"}  
set2 = {1, 2, 3}  
set3 = {"John", "Elena"}  
set4 = {"apple", "bananas", "cherry"}  
  
myset = set1.union(set2, set3, set4)  
print(myset)
```

متدهایی که کارش این است که تمام ایتم‌های یک لیست ست را داخل ست دیگری بزند، متدهای update است. این متدهایی برای برنامه‌گرداندن همان اصلی را تغییر میدهد.

```
set1 = {"a", "b", "c"}  
set2 = {1, 2, 3}  
  
set1.update(set2)  
print(set1)
```

هر دوی متدهای union و update اعضای تکراری برنامه‌گردانند. برای نگهداری اعضای فقط تکراری‌ها باید از intersection استفاده کرد. که یک ست جدید برنامه‌گرداند که فقط شامل اعضای های تکراری است. می‌توان به جای این عبارت از & استفاده کرد که البته مثل مورد قبلی فقط برای لیست‌های ست است این عبارت.

متدهای intersection_update همان کار را می‌کنند با این تفاوت که دیگر لیست جدیدی برنامه‌گرداند همان لیست اصلی را تغییر میدهد. گفتیم که دو مقدار True و False یکسان در نظر گرفته می‌شود. پس در تکراری‌ها هم حساب می‌شوند.

```
set1 = {"apple", "banana", "cherry"}  
set2 = {"google", "microsoft", "apple"}  
  
set3 = set1.intersection(set2)  
print(set3)
```

```
set1 = {"apple", "banana", "cherry"}  
set2 = {"google", "microsoft", "apple"}  
  
set3 = set1 & set2  
print(set3)
```

```
set1 = {"apple", "banana", "cherry"}  
set2 = {"google", "microsoft", "apple"}  
  
set1.intersection_update(set2)  
  
print(set1)
```

```
set1 = {"apple", 1, "banana", 0, "cherry"}  
set2 = {False, "google", 1, "apple", 2, True}  
  
set3 = set1.intersection(set2)  
  
print(set3)
```

متدهای intersection و intersection_update می‌شوند. intersection این است که نشان دهنده ایتم‌هایی است که در سه اول وجود دارد اما در بقیه لیست‌ها وجود ندارد. می‌توان از استفاده کرد البته که مانند قبل فقط برای سه امکان پذیر است. با استفاده از difference_update دیگر سه جدیدی برگردانده نمی‌شود بلکه همان سه اصلی تغییر می‌کند.

```
set1 = {"apple", "banana", "cherry"}  
set2 = {"google", "microsoft", "apple"}  
  
set3 = set1.difference(set2)  
  
print(set3)
```

```
set1 = {"apple", "banana", "cherry"}  
set2 = {"google", "microsoft", "apple"}  
  
set3 = set1 - set2  
print(set3)
```

```
set1 = {"apple", "banana", "cherry"}  
set2 = {"google", "microsoft", "apple"}  
  
set1.difference_update(set2)  
  
print(set1)
```

حالا اگر میخواهیم ایتم هایی که در هیچ کدام از لیست ها وجود ندارند رو برگردانیم از استفاده symmetric_difference میتوانیم از عبارت ^ استفاده کنیم که البته همانند قبل فقط بین ست ها مجاز است. متده symmetric_difference همان کار را میکند با تفاوت اینکه دیگر ست جدید برنمیگرداند بلکه همان اصلی را تغییر میدهد.

```
set1 = {"apple", "banana", "cherry"}  
set2 = {"google", "microsoft", "apple"}  
  
set3 = set1.symmetric_difference(set2)  
  
print(set3)
```

```

set1 = {"apple", "banana", "cherry"}
set2 = {"google", "microsoft", "apple"}

set1.symmetric_difference_update(set2)

print(set1)

```

Set Methods:

برخی از متدهای سمت:

Method	Shortcut	Description
<code>add()</code>		Adds an element to the set
<code>clear()</code>		Removes all the elements from the set
<code>copy()</code>		Returns a copy of the set
<code>difference()</code>	-	Returns a set containing the difference between two or more sets
<code>difference_update()</code>	-=	Removes the items in this set that are also included in another, specified set
<code>discard()</code>		Remove the specified item
<code>intersection()</code>	&	Returns a set, that is the intersection of two other sets
<code>intersection_update()</code>	&=	Removes the items in this set that are not present in other, specified set(s)
<code>isdisjoint()</code>		Returns whether two sets have a intersection or not
<code>issubset()</code>	<=	Returns whether another set contains this set or not
	<	Returns whether all items in this set is present in other, specified set(s)
<code>issuperset()</code>	>=	Returns whether this set contains another set or not
	>	Returns whether all items in other, specified set(s) is present in this set
<code>pop()</code>		Removes an element from the set
<code>remove()</code>		Removes the specified element
<code>symmetric_difference()</code>	^	Returns a set with the symmetric differences of two sets
<code>symmetric_difference_update()</code>	^=	Inserts the symmetric differences from this set and another
<code>union()</code>		Return a set containing the union of sets
<code>update()</code>	=	Update the set with the union of this set and others

Python Dictionaries:

دیکشنری ها نوع دگیری از کالکشن ها هستند که دیتا در خود نگه میدارند و اخیرین نوعی که ما اینجا راجب شون صحبت میکنیم. دیکشنری ها یک key، و یک value دارند که متعلق به key هستند. داخل دیکشنری ها میتوان از هر نوع دیتا تایپ استفاده کرد؛ لیست، استرینگ، عدد،.... از پایتون ۳.۷ به بعد دیکشنری ها هم به ترتیب شدند. دیکشنری ها قابل تغییر هستند اما نمیتوان مقدار key تکراری به انها داد. برای استفاده از مقدار از کلید آن استفاده میکنند و key ان را صدا میزنند و value ان چاپ میشود.

برای فهمیدن طول دیکشنری از تابع `len()` استفاده میشود. و همچنین برای فهمیدن نوع لیست از

(`type`) می‌توان استفاده کرد. همچنین می‌توان از قابلیت کانسٹراکتور خود دیکشنری استفاده کرد تا بتوان خود دیکشنری را چاپ کرد.

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}
```

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
print(thisdict)
```

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
print(thisdict["brand"])
```

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964,  
    "year": 2020  
}  
print(thisdict)
```

```
print(len(thisdict))
```

```
thisdict = {  
    "brand": "Ford",  
    "electric": False,  
    "year": 1964,  
    "colors": ["red", "white", "blue"]  
}
```

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
print(type(thisdict))
```

```
thisdict = dict(name = "John", age = 36, country = "Norway")  
print(thisdict)
```

Accessing Items:

میتوان به لیست دیکشنری به مقدارهایشون از طریق کلید ها دسترسی پیدا کرد. یا از طریق صدا زدن کلید یا از طریق متدهای `get`. همچنین با صدا زدن متدهای `keys` لیستی از تمام کلید های دیکشنری برگردانده میشود. و با صدا زدن `values` لیستی از تمام مقدار های دیکشنری برگردانده میشود. متدهای داریم به اسم `item` که تمام ایتم های دیکشنری رو به صورت لیست تاپل برگرداند. و میتوان از `in` در شرط استفاده کرد که چک کنیم کلید خاصی که مد نظر داریم در دیکشنری وجود دارد یا خیر.

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
x = thisdict["model"]
```

```
x = thisdict.get("model")
```

```
x = thisdict.keys()
```

```
car = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}
```

```
x = car.keys()
```

```
print(x) #before the change
```

```
car["color"] = "white"
```

```
print(x) #after the change
```

```
x = thisdict.values()
```

```
car = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
  
x = car.values()  
  
print(x) #before the change  
  
car["year"] = 2020  
  
print(x) #after the change
```

```
car = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
  
x = car.values()  
  
print(x) #before the change  
  
car["color"] = "red"  
  
print(x) #after the change
```

```
x = thisdict.items()
```

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
if "model" in thisdict:  
    print("Yes, 'model' is one of the keys in the thisdict dictionary")
```

```
car = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}
```

```
x = car.items()  
  
print(x) #before the change  
  
car["year"] = 2020  
  
print(x) #after the change
```

Change Dictionary Items:

برای تغییر مقدار در دیکشنری می‌توان از دو متده استفاده کرد! با صدا زدن کلید آن و دادن مقدار جدید یا با متده update

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
thisdict["year"] = 2018
```

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
thisdict.update({"year": 2020})
```

Add Dictionary Items:

برای اضافه کردن ایتم به دیکشنری مانند قسمت قبل عمل میکنیم با این تفاوت که قسمت قبل برای تغییر مقداری که وجود داشت اغدام میکردیم اما همان کار را انجام میدهیم و اگر مقدار و کلید وجود نداشتند به لیست اضافه می‌شوند.

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
thisdict["color"] = "red"  
print(thisdict)
```

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
thisdict.update({"color": "red"})
```

Remove Dictionary Items:

برای حذف کردن ایتمی از دیکشنری کافیست از چند متده که وجود دارد استفاده کنید. یکی از متدها clear است که کل دیکشنری را خالی میکند. مورد بعدی del است که هم می‌تواند کل دیکشنری را پاک کند و هم می‌تواند یک ایتم به خصوص را حذف کند. یکی از متدهای دیگر popitem است که از پایتون ۳.۶ به بعد اخرين کلید و مقدار رو که اضافه شدند رو دیلیت میکند ولی از ۳.۶ به قبل به صورت رندوم دیلیت میکند. متدهم که با کلید مخصوص که مشخص میشود دیلیت میکند ان ایتم را.

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
thisdict.pop("model")  
print(thisdict)
```

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
thisdict.popitem()  
print(thisdict)
```

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
del thisdict["model"]  
print(thisdict)
```

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
del thisdict  
print(thisdict) #this will cause an error because "thisdict" no longer  
exists.
```

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
thisdict.clear()  
print(thisdict)
```

Loop Dictionaries:

دیکشنری ها میتوان از حلقه ها برای چاپ کلید ها یا مقدار ها یا هردو با یکدیگر استفاده کرد. که نحوه های استفاده رو پایین مذکوریم. میتان از keys یا values استفاده کرد. اگر هم بخواهیم هر دو را باهم برگردانیم از items استفاده میکنیم.

```
for x in thisdict:  
    print(x)
```

در اینجا به تمام کلید ها برگردانده میشوند.
اما در قسمت پایین مقدار ها برگردانده میشوند.

```
for x in thisdict:  
    print(thisdict[x])
```

```
for x in thisdict.values():  
    print(x)
```

```
for x in thisdict.keys():  
    print(x)
```

```
for x, y in thisdict.items():  
    print(x, y)
```

Copy Dictionaries:

برای اینکه یک دیکشنری رو کپی کنیم داخل متغیر دیگری همانند چیزی که قبلا در لیست ها و تاپل ها گفتیم و در صورت کلی نمیتوانیم مساوی مقدار قرار دهیم چون به اون صورت داریم رفرنسی ازش ایجاد میکنیم که به اون خونه حافظه لیست اولیه اشاره میکنه و در نتیجه تغییراتی که روی لیست

اول اعمال میشود همیشه روی لیست دوم نیز اعمال میشود. برا همین میتوان از `()copy()` , `dict` استفاده کرد.

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
mydict = thisdict.copy()  
print(mydict)
```

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
mydict = dict(thisdict)  
print(mydict)
```

Nested Dictionaries:

ما مفهومی داریم به نام دیشکنری های تو در تو یا همان nested dictionaries که میتوان به وسیله این قابلیت ما داخل یک دیکشنری چندین دیکشنری دیگر نیز داشته باشیم.
راه اول این است که داخل دیکشنری اصلی ما چندین دیکشنری ایجاد کنیم به همراه کلید و مقدار یا اینکه بیرون از ان چند دیکشنری دیگر را به یک دیکشنری اضافه کنیم که از طریق حلقه ها به ان دسترس خواهیم داشت.

```
myfamily = {  
    "child1" : {  
        "name" : "Emil",  
        "year" : 2004  
    },  
    "child2" : {  
        "name" : "Tobias",  
        "year" : 2007  
    },  
    "child3" : {  
        "name" : "Linus",  
        "year" : 2011  
    }  
}
```

```

child1 = {
    "name" : "Emil",
    "year" : 2004
}
child2 = {
    "name" : "Tobias",
    "year" : 2007
}
child3 = {
    "name" : "Linus",
    "year" : 2011
}

myfamily = {
    "child1" : child1,
    "child2" : child2,
    "child3" : child3
}

```

برای چاپ کردن ایتم ها باید اسم دیکشنری بیرونی رو اول بگیم و بعد داخلی مورد نظر :

```
print(myfamily["child2"]["name"])
```

برای چاپ تمام ایتم های موجود :

```

for x, obj in myfamily.items():
    print(x)

    for y in obj:
        print(y + ':', obj[y])

```

Dictionary Methods:

برخی متدهای دیکشنری :

Method	Description
<code><u>clear()</u></code>	Removes all the elements from the dictionary
<code><u>copy()</u></code>	Returns a copy of the dictionary
<code><u>fromkeys()</u></code>	Returns a dictionary with the specified keys and value
<code><u>get()</u></code>	Returns the value of the specified key
<code><u>items()</u></code>	Returns a list containing a tuple for each key value pair
<code><u>keys()</u></code>	Returns a list containing the dictionary's keys
<code><u>pop()</u></code>	Removes the element with the specified key
<code><u>popitem()</u></code>	Removes the last inserted key-value pair
<code><u>setdefault()</u></code>	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
<code><u>update()</u></code>	Updates the dictionary with the specified key-value pairs
<code><u>values()</u></code>	Returns a list of all the values in the dictionary

Python If ... Else:

پایتون یکسری شرط هایی رو ساپورت میکند که بیشتر در موقعیت های شرطی یا داخل حلقه ها بکار میروند:

- Equals: `a == b`
- Not Equals: `a != b`
- Less than: `a < b`
- Less than or equal to: `a <= b`
- Greater than: `a > b`
- Greater than or equal to: `a >= b`

برای عبارات شرطی یکسری قواعد وجود دارد. برای نوشتن شرطی از if و elif و else و استفاده میشود. توى شرط اول if نوشته میشود و else بعد از آن گذاشته میشود به ان معنا که اگر فلان شرط درست بود فلان کار رو انجام بده در غیر این صورت که وارد بلوك الس میشود کاري که گفته شده رو انجام بده. اگر چندین شرط داشتیم میتوانیم if بنویسیم چندین elif ها استفاده کنیم. همچنین میتوان شرط را در یک خط نوشت اگر کوتاه است کار و شرط ما و یا if else... if رو هم به همانگونه میتوان در یک خط نوشت. یا میتوان از عبات سه شرطی استفاده کرد که سه تا شرط if elif else در ان بررسی میشوند. از عبارات and , or , not میتوان در شرط ها استفاده کرد. همانگونه که در دیکشنری ها تو در

تو داشتیم در شرط ها هم nested if داریم. یک عبارتی هم داریم به اسم pass که به هر دلیل خواستیم شرطی وجود داشته باشد اما هیچ کاری نکند این عبارت را داخل این شرط مینویسیم. در پایتون دقیق داشته باشید که بلوک ها وجود ندارند و پایتون بر اساس فرو رفتگی برخورد میکند پس در شرط ها و حلقه ها و فانکشن ها حواستان به فرو رفتگی ها باشد!!

```
a = 33
b = 200
if b > a:
    print("b is greater than a")
```

```
a = 33
b = 33
if b > a:
    print("b is greater than a")
elif a == b:
    print("a and b are equal")
```

```
a = 200
b = 33
if b > a:
    print("b is greater than a")
elif a == b:
    print("a and b are equal")
else:
    print("a is greater than b")
```

:دقت در فرو رفتگی ها:

```
a = 33
b = 200
if b > a:
    print("b is greater than a") # you will get an error
```

```
a = 200
b = 33
if b > a:
    print("b is greater than a")
else:
    print("b is not greater than a")
```

عبارت شرطی در یک خط به صورت کوتاه:

```
if a > b: print("a is greater than b")
```

عبارت دو شرطی در یک خط به صورت کوتاه:

```
a = 2
b = 330
print("A") if a > b else print("B")
```

و عبارت سه شرطی که راجبش صحبت کردیم:

```
a = 330
b = 330
print("A") if a > b else print("=") if a == b else print("B")
```

استفاده از کلید واژه های و یا نات :

```
a = 200
b = 33
c = 500
if a > b and c > a:
    print("Both conditions are True")
```

```
a = 200
b = 33
c = 500
if a > b or a > c:
    print("At least one of the conditions is True")
```

```
a = 33
b = 200
if not a > b:
    print("a is NOT greater than b")
```

شرط های تو در تو:

```
x = 41

if x > 10:
    print("Above ten,")
    if x > 20:
        print("and also above 20!")
    else:
        print("but not above 20.")
```

عبارت pass برای رد کردن شرط خالی:

```
a = 33
b = 200

if b > a:
    pass
```

Python For Loops:

در پایتون نوعی از حلقه ها for است که با ویژگی های ان اشنا میشویم. برای انجام عملیات رو تک تک اعضای یک لیستی یا متغیر ها مثل استرینگ و... از حلقه for استفاده میکنیم. داخل حلقه ها میتوان از عبارات مختلفی استفاده کرد. یکی از دستور های حلقه ها break است که وقتی به این عبارت برسد حلقه تمام میشود و از حلقه خارج میشود برنامه. عبارت دیگر continue است که وقتی به این عبارت برسد در واقع پرش میکند از ان مرحله از حلقه.تابع دیگری که میتوان در حلقه ها استفاده کرد range است که قبلا با ان اشنا شدیم. میتوانیم بعد از حلقه else بیاریم به این معنا که هر وقت حلقه ما تمام شد این عبارت در else اجرا میشود؛ مگر انکه به عبارت break برخورد کند. در پایتون nested loops هم داریم حلقه های تو در تو. و همانند شرطی که pass داشتیم اینجا هم داریم.

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    print(x)
```

```
for x in "banana":
    print(x)
```

استفاده break در پایتون:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    print(x)
    if x == "banana":
        break
```

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    if x == "banana":
        break
    print(x)
```

استفاده continue:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    if x == "banana":
        continue
    print(x)
```

نحوه کاربرد range:

```
for x in range(2, 30, 3):
    print(x)
```

استفاده else در حلقه ها:

```
for x in range(6):
    print(x)
else:
    print("Finally finished!")
```

nested loops:

```
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]

for x in adj:
    for y in fruits:
        print(x, y)
```

عبارت pass برای حلقه های خالی:

```
for x in [0, 1, 2]:
    pass
```

Python While Loops:

دومین و اخرين حلقه داخل پايتون حلقه while است که هم تقربيا همانند حلقه for است. حلقه while تا زمانی که شرطی که بهش ميدهيم درست باشد ادامه پيدا ميکند و حلقه را اجرا ميکند و شامل عبارت های continue , break ميشود. همچنين ميتواند از else استفاده کند.

```
i = 1
while i < 6:
    print(i)
    i += 1
```

```
i = 1
while i < 6:
    print(i)
    if i == 3:
        break
    i += 1
```

```
i = 0
while i < 6:
    i += 1
    if i == 3:
        continue
    print(i)
```

```
i = 1
while i < 6:
    print(i)
    i += 1
else:
    print("i is no longer less than 6")
```

Python Functions:

تابع ها در پایتون چیز هایی هستند که در خود دیتا نگهداری میکنند و یکسری کار خاصی که ما تعیین میکنیم انجام میدهند و با صدا زدن ان ها اجرا می شوند. تابع ها میتوانند پارامتر داشته باشند یا ارگومان داشته باشند. فرق پارامتر و ارگومان در این است که پارامتر متغیری است که داخل پرانتز جلوی تابع تعریف میشود. ارگومان مقداری است که پاس میشود به تابع. با کلیدواژه def تعریف میشود. باید به مقداری که ارگومان تعریف میشود در تابع به همان میزان هم پاس شود. در موقعی که نمیدانید چند پارامتر باید پاس کنید به تابع میتوانید از * استفاده کنید. همچنین میتوانید بر اساس keyword پارامتر رو ارسال کنید. یعنی مهم نیست اول بفرستید یا اخر پارامتر در تابع اول هست یا اخر مهم اینه که دقیق اسما ارگومان ذکر شود. در * args نباید اسم پارامتر رو بنویسیم فقط مقدار ان را مینویسیم (ارزش ان را) اما داخل ** kwargs باید هم اسم پارامتر و هم مقدار ان را بنویسیم. هم چنین میتوانیم برای پارامتر هامون یک مقدار دیفالت بزاریم که اگر به عنوان ارگومان چیزی ارسال نشد مشکلی به وجود نیاید. برای اینکه تابعی مقداری را بازگرداند باید از return استفاده کنیم. و همانند حلقه ها و شرطی ها اینجا هم عبارت pass میتوانیم استفاده کنیم.

یکی از مفاهیم دیگ ای که در تابع ها وجود دارد این است که شاید ما بخواهیم دقیقا بر اساس جایگاه

ارگومان ها و ترتیب انها و بدون ذکر نام ارگومان ها رو پاس کنیم. برای استفاده از این باید بعد از ارگومان / ، بگذاریم.

یکی دیگر از مفاهیم Keyword Arguments است که یعنی باید مقدار را با اسم ارگومان بنویسیم. اما دقیقاً بر عکس این موضوع Keyword-Only Arguments یعنی حتماً باید نام ارگومان ذکر بشود تا پاس بشود به تابع و برای استفاده از آن باید از قبل از ارگومان * بگذاریم.

و میتوانیم این دو مفهوم را با یکدیگر ترکیب کنیم.

یکی از روش های نوشتن تابع برای حل مسله recursion است که تابعی است که خودش را صدا میزند. و سعی میکند هر سری مسله رکوچک تر کند و به نقطه شرط توقف نزدیکی تر میشود.

یه نکته خیلی مهم: توابع رو با متدها اشتباہ نگیریم!!!!

توابع خارج از کلاس ها تعریف می‌شوند و وابسته به کلاس یا ابجکتی نیستند و مستقل هستند.

اما متدها داخل کلاس ها تعریف می‌شوند و برای استفاده از آنها باید یک شی از کلاس ساخته بشود و به اطلاعات داخل کلاس دسترسی دارند.

```
def my_function():
    print("Hello from a function")
```

```
my_function()
```

```
def my_function(fname):
    print(fname + " Refsnes")
```

```
my_function("Emil")
my_function("Tobias")
my_function("Linus")
```

:args

```
def my_function(*kids):
    print("The youngest child is " + kids[2])
```

```
my_function("Emil", "Tobias", "Linus")
```

:Keyword Arguments

```
def my_function(child3, child2, child1):
    print("The youngest child is " + child3)

my_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")
```

**kwargs:

```
def my_function(**kid):
    print("His last name is " + kid["lname"])

my_function(fname = "Tobias", lname = "Refsnes")
```

```
def my_function(country = "Norway"):
    print("I am from " + country)

my_function("Sweden")
my_function("India")
my_function()
my_function("Brazil")
```

```
def my_function(x):
    return 5 * x

print(my_function(3))
print(my_function(5))
print(my_function(9))
```

Dunder:

متدهای ویژه در پایتون با `__` شروع میشوند و پایان میابند که هر کدام یکسری وظایف خاص خود را دارند.
همانند `...., ()__init__`

Generator:

همانند تابع است با یکسری تفاوچت ها. اولاً که در تابع شما فقط میتوانید یک خروجی داشته باشید با دستور `return` ولی در `generator` ها از دستور `yield` به جای آن استفاده میشود و همچنین میتوان هر چند تا که مورد نیاز استفاده کرد و به زبان دیگر میتوان چندین خروجی داشت. همچنین هر جا که کد متوقف شود سری بعدی که اجرا شود برعکس تابع ها که از اول

شروع میشوند در اینجا از ادامه فرایند ادامه پیدا میکند.

```
def myGenerator():
    #Code to execute
    yield something
```

```
1 def counter():
2     i=1
3     while(i<=10):
4         yield i
5         i+=1
6
7 for i in counter():
8     print(i)
```

output:

```
1
2
3
4
5
6
7
8
9
10
```

```
1 def myGenerator(number):
2     yield number
3     yield number + 1
4
5 generator = myGenerator(6)
6
7 print(next(generator))
8 print(next(generator))
```

```
6
7
```

Generator expression:

همانند List comprehension است با تفاوت اینکه در اینجا به جای [] از () استفاده میکنیم و همچنین خروجی که به ما میدهد یک generator است. و از نظر اشغال فضا بشدت کاربردی است که بسیار حافظه کمی را اشغال میکند. و همانند لیست ها میتوان روی ایتم های آن پیمایش کرد.

(Expression for item in collection)

:مثال

```
List_comprehension = [number for number in range(5)]  
Generator_Expression = (number for number in range(5))
```

```
print(List_comprehension)  
print(Generator_Expression)
```

output:

```
[0, 1, 2, 3, 4]  
<generator object <genexpr> at 0x00BCC9C8>
```

اگر سایز هم قیاس کنیم:

```
from sys import getsizeof
```

```
list_comprehension = [number for number in range(1000)]  
Generator_Expression = (number for number in range(1000))
```

```
print(getsizeof(list_comprehension))  
print(getsizeof(Generator_Expression))
```

output:

```
4508  
56
```

paradigms:

نوعی از تفکر برنامه نویسی است که نوع های مختلف وجود دارد که در پایتون سه تا استفاده میشود.

- Object Oriented programming paradigms
- Procedure Oriented programming paradigms
- Functional programming paradigms

در واقع از خود مسله برای حل جواب استفاده میکنیم در کلاس ها و فانکشن ها

...9

در POP paradigms برنامه خود را به صورت عادی مینویسم و خیلی بیسیک و ساده.

در Functional programming paradigms برنامه خود را به صورت ریاضی خالص مینویسم با شرط ها و عبارات ریاضیاتی.

:مثال ها

OOP:

```
# class Emp has been defined here  
class Emp:  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age  
  
    def info(self):  
        print("Hello, % s. You are % s old." % (self.name,
```

```
    self.age))  
  
# Objects of class Emp has been  
# made here  
Emps = [Emp("John", 43),  
        Emp("Hilbert", 16),  
        Emp("Alice", 30)]
```

```
# Objects of class Emp has been  
# used here  
for emp in Emps:  
    emp.info()
```

output:

```
Hello, John. You are 43 old.  
Hello, Hilbert. You are 16 old.  
Hello, Alice. You are 30 old.
```

POP:

```
# Procedural way of finding sum  
# of a list
```

```
mylist = [10, 20, 30, 40]
```

```
# modularization is done by  
# functional approach  
def sum_the_list(mylist):  
    res = 0  
    for val in mylist:  
        res += val  
    return res
```

```
print(sum_the_list(mylist))
```

output:

```
100
```

Functional:

```
# Functional way of finding sum of a list  
import functools
```

```
mylist = [11, 22, 33, 44]
```

```
# Recursive Functional approach
```

```
def sum_the_list(mylist):

    if len(mylist) == 1:
        return mylist[0]
    else:
        return mylist[0] + sum_the_list(mylist[1:])

# lambda function is used
print(functools.reduce(lambda x, y: x + y, mylist))
```

output:

110

context manager:

```
def myfunction():
    pass
```

Python Lambda:

لامبدا در پایتون میتواند هر مقدار ارگومانی که میخواهد بگیرد اما فقط یک نتیجه میدهد. از لامبда در فانکشن ها میتوان استفاده کرد مثلا در جایی که میخواهیم محاسبات ریاضی یا برای کاری که به صورت کوتاه مدت میخواهیم استفاده کنیم.

```
x = lambda a, b : a * b
print(x(5, 6))
```

```
x = lambda a : a + 10
print(x(5))
```

```
x = lambda a, b, c : a + b + c
print(x(5, 6, 2))
```

استفاده در فانکشن وقتی میخواهیم برای عددی که نمیدونیم کار کنیم:

```
def myfunc(n):
    return lambda a : a * n
```

```
def myfunc(n):
    return lambda a : a * n

mydoubler = myfunc(2)

print(mydoubler(11))
```

```
def myfunc(n):
    return lambda a : a * n

mytrippler = myfunc(3)

print(mytrippler(11))
```

```
def myfunc(n):
    return lambda a : a * n

mydoubler = myfunc(2)
mytrippler = myfunc(3)

print(mydoubler(11))
print(mytrippler(11))
```

Python Arrays:

پایتون برای Array ها مازول های داخلی ندارند. داخلی پایتون لیست ها همون Array ها هستند.
Access the Elements of an Array:

می توان با اشاره به شماره ایندکس ها به مقدار اون ایندکس دسترسی پیدا کرد.

```
x = cars[0]
```

برای مقدار دهی ایندکس خاص:

```
cars[0] = "Toyota"
```

همانند دیتا کالکشن هایی که در گذشته دیدیم اینجا هم می‌توانیم از `len` استفاده کنیم.

Looping Array Elements:

میتوان از حلقه `for in` استفاده کرد تا به تمام ایندکس های یک ارایه (لیستی که همانند ارایه باهاش برخورد می‌شود) دسترسی پیدا کرد.

```
for x in cars:  
    print(x)
```

Adding Array Elements:

با استفاده از `append` میتوان به ارایه مقداری را اضافه کرد.

```
cars.append("Honda")
```

Removing Array Elements:

برای حذف کردن یک مقدار از ارایه توسط ایندکس ان از `pop` استفاده می‌شود و برای حذف کردن بر اساس مقدار ان از `remove` استفاده می‌شود.

```
cars.pop(1)
```

```
cars.remove("Volvo")
```

Array Methods:

Method	Description
<u>append()</u>	Adds an element at the end of the list
<u>clear()</u>	Removes all the elements from the list
<u>copy()</u>	Returns a copy of the list
<u>count()</u>	Returns the number of elements with the specified value
<u>extend()</u>	Add the elements of a list (or any iterable), to the end of the current list
<u>index()</u>	Returns the index of the first element with the specified value
<u>insert()</u>	Adds an element at the specified position
<u>pop()</u>	Removes the element at the specified position
<u>remove()</u>	Removes the first item with the specified value
<u>reverse()</u>	Reverses the order of the list
<u>sort()</u>	Sorts the list

Python Classes and Objects:

همانگونه که میدانیم پایتون یک زبان شی گرایی است! و با مفهوم OOP قبل اشنا شدیم، که هر چیزی در این نوع برنامه نویسی یکی شی است یعنی در واقع ما در حل مسله تلاش میکنیم تا صورت سوال رو به جواب مسله در برنامه نویسی نزدیک تر کنیم.

Create a Class:

برای ساخت کلاس از کلید واژه `class` استفاده میکنیم. ما کلاس ها را ایجاد میکنیم که داخل اون ها فانکشن ها و ویژگی های برنامه مون رو بنویسیم.

نکته: معمولاً ما برای یک پروژه فکر میکنیم چند بخش داریم و هر بخش چه ویژگی هایی دارند و یا چه اعمالی دارند، مثلاً ما یک کلاس انسان داریم که یکسری رفتار داره که این رفتار ها میشوند فانکشن و یکسری ویژگی دارد که این ها میشوند ویژگی ها(متغیر ها) اینگونه روش برنامه نویسی را شی گرایی میگویند.

```
class MyClass:
    x = 5
```

Create Object:

برای استفاده از کلاس ها در کلاس های دیگر و در فانکشن ها باید یک ابجکت جدید از روی اون کلاس یا اون شی بسازیم.

```
p1 = MyClass()  
print(p1.x)
```

The `__init__()` Function:

در پایتون به کانسٹراکتور میگویند `init`. که برای مقدار دهی استفاده میشود تا وقتی از بیرون داخل یک فانکشن دیگر یا یک کلاس دیگر مقدار دهی صورت گرفتن توسط شی جدیدی که ساخته شده است ان مقدار درست وارد کلاس شود و استفاده شود. در واقع هر زمان که شی ای ازش ساخته بشود اولین چیزی که اجرا میشود این فانکشن است.

```
class Person:  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age  
  
p1 = Person("John", 36)  
  
print(p1.name)  
print(p1.age)
```

The `__str__()` Function:

همانند متدهای `toString` در زبان های جاوا، جاوا اسکریپت است برای فرمت نحوه نمایش رشته است. که میگوییم رشته را به چه صورت نمایش بدهد با اطلاعاتی که ما بهش میدهیم.

```

class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return f"{self.name}({self.age})"

p1 = Person("John", 36)

print(p1)

```

Object Methods:

شی ها میتوانند متدهایی را در بر داشته باشند. (مثلی که بالا زدم) در کلاس ها میتوانیم متدهایی را ایجاد کنیم

```

class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def myfunc(self):
        print("Hello my name is " + self.name)

p1 = Person("John", 36)
p1.myfunc()

```

کلید واژه self همان مانند this عمل میکند. یعنی اشاره میکند به متغیر محلی در همان فانکشن برای جلوگیری از تداخل اسم ها با یکدیگر. با این تفاوت که در پایتون در پارامتر های میاریم. البته که میتوانیم از کلید واژه ای که دوست داریم استفاده کنیم اما استاندارد این هست.

Modify Object Properties:

برای مقدار دهی یک ویژگی ابجکت به اینگونه عمل میکنیم:

```
p1.age = 40
```

Delete Object Properties:

برای دیلیت کردن یک ویژگی ابجکت از `del` استفاده میکنیم.

```
del p1.age
```

Delete Objects:

با استفاده از `del` و نام خود ابجکت ان ابجکت را حذف میکنیم(ان ابجکتی از روی کلاسمان ساختیم).

```
del p1
```

The pass Statement:

همانند شرط و حلقه و توابع در کلاس ها هم `pass` را داریم که اگر به هر دلیلی خواستیم خالی باشے کلاسمان از این استفاده کنیم.

```
class Person:  
    pass
```

Python Inheritance:

در پایتون چون زبان شی گرا است و با OOP کار میکنیم، مفهومی وجود دارد به اسم ارث بری که یکی از ۴ ویژگی مهم زبان های شی گرا است. در ارث بری یک کلاس کلاس والد میشود و یک یا چندین کلاس دیگر فرزند این کلاس والد میشوند . یعنی چی؟! یعنی تمام ویژگی ها، متدها، رفتار های کلاس والد در کلاس فرزند نیز وجود دارند که یا همان هستند دقیقا یا کلاس فرزند میتواند آن را `override` کنید و ویژگی یا رفتار بیشتر به ان بدهید. البته یکسری نکات وجود دارد که باید رعایت شود مثل اینکه هر کلاسی میتواند کلاس والد باشد و هر کلاس میتواند فقط از یک کلاس ارث بری کند و در کانسٹراکتور کلاس فرزند باید کلاس والد ذکر بشه . داخل پرانتز کلاس فرزند اسم کلاس والد باشد. همانند کد ها:

```
class Person:  
    def __init__(self, fname, lname):  
        self.firstname = fname  
        self.lastname = lname  
  
    def printname(self):  
        print(self.firstname, self.lastname)  
  
#Use the Person class to create an object, and then execute the  
method:  
  
x = Person("John", "Doe")  
x.printname()
```

```
class Student(Person):  
    pass
```

```
x = Student("Mike", "Olsen")  
x.printname()
```

استفاده کانسٹراکتور پدر در فرزند:

```
class Student(Person):  
    def __init__(self, fname, lname):  
        Person.__init__(self, fname, lname)
```

```
class Student(Person):  
    def __init__(self, fname, lname):  
        super().__init__(fname, lname)
```

اضافه کردن ویژگی به کلاس فرزند:

```

class Student(Person):
    def __init__(self, fname, lname, year):
        super().__init__(fname, lname)
        self.graduationyear = year

x = Student("Mike", "Olsen", 2019)

```

اضافه کردن رفتار به کلاس فرزند:

```

class Student(Person):
    def __init__(self, fname, lname, year):
        super().__init__(fname, lname)
        self.graduationyear = year

    def welcome(self):
        print("Welcome", self.firstname, self.lastname, "to the class of",
              self.graduationyear)

```

Python Iterators:

ابجکت هایی هستند که میتوان روی مقداری ازها پیمایش انجام دادکه دارای دو متدهای `__iter__()` و `__next__()` هست.

اما `iterable` یعنی دیتا کالکشن هایی که قابلیت ایتریت کردن روشون وجود دارد و `iter()` متدهای `iterable` را دارا هست. استرینگ ها هم `iterable` هستند.

```

mytuple = ("apple", "banana", "cherry")
myit = iter(mytuple)

print(next(myit))
print(next(myit))
print(next(myit))

```

```
mystr = "banana"
myit = iter(mystr)

print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
```

Looping Through an Iterator:

ما برای ایتریت کردن از حلقه for هم میتوانیم استفاده کنیم.

```
mytuple = ("apple", "banana", "cherry")
```

```
for x in mytuple:
    print(x)
```

همچنین برای استرینگ ها:

```
mystr = "banana"

for x in mystr:
    print(x)
```

Create an Iterator:

برای ایجاد یک کلاس یا ابجکتی که ایتریتور باشد باید متدهای `__next__()` و `__iter__()` را ایمپلیمنت کند. متدهای `__init__()` مثل `iter` میتوان داخل مقدار دهی کرد و کارهای دیگر اما تفاوتش اینه که باید همیشه خودش را برگرداند. و هم `__next__()` همینطور ولی حتماً باید مقدار بعدی را برگرداند.

```
class MyNumbers:  
    def __iter__(self):  
        self.a = 1  
        return self  
  
    def __next__(self):  
        x = self.a  
        self.a += 1  
        return x  
  
myclass = MyNumbers()  
myiter = iter(myclass)  
  
print(next(myiter))  
print(next(myiter))  
print(next(myiter))  
print(next(myiter))  
print(next(myiter))
```

در مثال بالا ممکن است همینگونه تا بینهات ادامه پیدا کند برای اینکه در چنین ماردی به مشکل نخوریم متدی وجود دارد به اسم StopIteration این متد را میتوان جوری استفاده کرد در شرط که اگر به اون عدد دلخواه یا مقدار دلخواه تولید شد و کار انجام شد استپ کند و تمام شود برنامه.

```

class MyNumbers:
    def __iter__(self):
        self.a = 1
        return self

    def __next__(self):
        if self.a <= 20:
            x = self.a
            self.a += 1
            return x
        else:
            raise StopIteration

myclass = MyNumbers()
myiter = iter(myclass)

for x in myiter:
    print(x)

```

Decorators in Python:

از دکوریتور های زمانی استفاده میکنیم که یک تابع داریم و نمیخواهیم ساختار کلی تابع را تغییر دهیم ولی میخواهیم عملی به ان اضافه کنیم با استفاده از دکوریتور ها اینکار را نجام میدهیم. که بیشتر برای لگین یا تایید دسترسی یا تایمینگ انجام میشود. برای ساخت یک دکوراتور تابع مورد نظر خود را با تغییراتی که میخواهیم بمینویسیم و بعد بالا اون با علامت @ نام تابعی که میخواهیم درش اضافه کنیم رو مینویسیم.

```

def say_hello():
    print("Hello!")

def greet_decorator(func):
    def wrapper():
        print("Welcome!")
        func()
    return wrapper

@greet_decorator

```

```
def say_hello():
    print("Hello!")
```

Python Polymorphism:

یک دیگر از خاصیت های مهم OOP بحث پلی مورفیسم است. خود این کلمه به معنای شکل های زیاد هست. یعنی شما متدها/متغیرها/متانکشن را که اسمشان و کارشان یکی است را میتوانید در برنامه های مختلف استفاده کنید مثل `len` که در تاپل ها لیست ها دیکشنری هاو... استفاده میشوند. معمولا در کلاس ها استفاده میشوند مثل از یک متدهای `move` در کلاس های مختلف استفاده شود. ای در ارث بری مثلای کلاس وسیله نقلیه داشته باشیم و ۳ تا زیر مجموعه ماشین قایق هواپیما این متدها را صدا بزنیم چون هر سه این کلاس ها این متدها را دارند (در واقع این یک رفتار است که در اشتراک است با این ها) (نزدیک شدن مسله به روش حل). و میتوان این متدها رو `override` کرد.

پس اینکه یک شی میتواند متدهای داشته باشد که هم نام یک شی دیگر باشد و همان کار را بکند.

```
class Car:  
    def __init__(self, brand, model):  
        self.brand = brand  
        self.model = model  
  
    def move(self):  
        print("Drive!")  
  
class Boat:  
    def __init__(self, brand, model):  
        self.brand = brand  
        self.model = model  
  
    def move(self):  
        print("Sail!")  
  
class Plane:  
    def __init__(self, brand, model):  
        self.brand = brand  
        self.model = model  
  
    def move(self):  
        print("Fly!")  
  
car1 = Car("Ford", "Mustang")      #Create a Car class  
boat1 = Boat("Ibiza", "Touring 20") #Create a Boat class  
plane1 = Plane("Boeing", "747")     #Create a Plane class  
  
for x in (car1, boat1, plane1):  
    x.move()
```

```

class Vehicle:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    def move(self):
        print("Move!")

class Car(Vehicle):
    pass

class Boat(Vehicle):
    def move(self):
        print("Sail!")

class Plane(Vehicle):
    def move(self):
        print("Fly!")

car1 = Car("Ford", "Mustang") #Create a Car object
boat1 = Boat("Ibiza", "Touring 20") #Create a Boat object
plane1 = Plane("Boeing", "747") #Create a Plane object

for x in (car1, boat1, plane1):
    print(x.brand)
    print(x.model)
    x.move()

```

Python Scope:

وقتی یک متغیری را داخل یک تابع تعریف میکنیم ان متغیر لوکال فقط متعلق به ان تابع است و جای دیگری تعریف نشده است با ان مقدار به این سکوپ گفته میشود. حالا اگر از nested function ها استفاده کنیم متغیر تعریف شده در تابع اول در تابع های زیر قابل استفاده است اما متغیری که در تابع زیری تعریف شده است در تابع بالایی قابل استفاده نیست. اما اگر از کلید واژه nonlocal استفاده کنیم یعنی میتوانیم هم در تابع اول متغیر x داشته باشیم با یک مقدار خاص هم در تابع زیری داشته باشیم با یک مقدار که دادیم بهش. همچنین اگر در بدنه اصلی پایتون متغیری را تعریف کنیم به ان متغیر گلوبال گفته میشود که یعنی در تمام توابع و کلاس ها قابل استفاده است یا از کلید واژه global استفاده کنیم. یا اگر میخواهیم داخل یک تابع متغیری را تغییر دهیم از کلید واژه global استفاده کنیم:

```

def myfunc():
    x = 300
    def myinnerfunc():
        print(x)

```

```
( )myinnerfunc
```

```
myfunc()
```

تعريف گلوبال:

```
x = 300
```

```
def myfunc():
    x = 200
    print(x)
```

```
myfunc()
```

```
print(x)
```

تعريف گلوبال و تغيير ان در تابع:

```
x = 300
```

```
def myfunc():
    global x
    x = 200
```

```
myfunc()
```

```
print(x)
```

استفاده از نان لوکال:

```
def myfunc1():
    x = "Jane"
    def myfunc2():
        nonlocal x
        x = "hello"
    myfunc2()
    return x
```

```
print(myfunc1())
```

Python Modules:

مدل ها مثل کتابخانه ها می‌مانند. برای ساخت یک مدل کافیه یک فایل پایتون بنویسیم با کلاس ها یا تابع ها بنویسیم با توجه به خواسته هایمان و ان را سیو کنیم با پسوند .py. برای استفاده ان در فایل های خودمان import کنیم. و به صورت module_name.function_name. استفاده کنیم. در مدل ها هر چیزی میتوانیم بگذاریم. همچنین میتوانیم مدل را که ایمپورت میکنیم به نام دیگری سیو کنیم با استفاده از as. یکسری مدل های داخلی خود پایتون وجود دارد که میتوانید ان را ایمپورت کنید و از ان ها استفاده کنید. با استفاده از dir تمام فانکشن ها و متغیر ها رو برآتون داخل یک لیست نمایش میدهد.

شما میتوانید قسمتی از یک مدل را فقط ایمپورت کنید با ساتفاده از کلید واژه .from

```
def greeting(name):
```

```

print("Hello, " + name)
#save with .py

import mymodule

mymodule.greeting("Jonathan")  

                        استفاده از یک ابجکت خاص در مدل:  

person1 = {
    "name": "John",
    "age": 36,
    "country": "Norway"
}  

#save mymodule.py

import mymodule

a = mymodule.person1["age"]
print(a)  

                        سیو کردن مدل به یک اسم دیگر:  

import mymodule as mx

a = mx.person1["age"]
print(a)  

                        ایمپورت کردن یک قسمت خاص از مدل:  

def greeting(name):
    print("Hello, " + name)

person1 = {
    "name": "John",
    "age": 36,
    "country": "Norway"
}  

}

from mymodule import person1

print (person1["age"])

```

Python Datetime:

ما در پایتون می‌توانم datetime را ایمپورت کنیم و از آن مثل ابجکت‌هایی از جنس زمان باهاشون برخورد کنیم. در کانسٹراکتور خود این کلاس یکسری پارامتر ما می‌توانیم بدهیم بهش که بعضی هاش اپشنال هستند یعنی ضروری نیستند مثل میلی ثانیه یا ثانیه و... یکسری متدهایی هم داره این که بکار می‌ایند. میتوانیم تایم خود را درست کنیم یا می‌توانیم به صورت فرمت خاصی تایم رو برگردانیم.

```

import datetime

```

```

x = datetime.datetime.now()
print(x)

#output 2024-11-07 00:20:47.840660

import datetime

x = datetime.datetime(2020, 5, 17)

print(x)

import datetime

x = datetime.datetime(2018, 6, 1)

print(x.strftime("%B"))

```

رفنس های فرمت ها:

Directive	Description	Example
%a	Weekday, short version	Wed
%A	Weekday, full version	Wednesday
%w	Weekday as a number 0-6, 0 is Sunday	3
%d	Day of month 01-31	31
%b	Month name, short version	Dec
%B	Month name, full version	December
%m	Month as a number 01-12	12
%y	Year, short version, without century	18
%Y	Year, full version	2018
%H	Hour 00-23	17
%I	Hour 00-12	05
%p	AM/PM	PM
%M	Minute 00-59	41
%S	Second 00-59	08
%f	Microsecond ~~~~~ ~~~~~	548513

	0000000-9999999	
%Z	UTC offset	+0100
%z	Timezone	CST
%j	Day number of year 001-366	365
%U	Week number of year, Sunday as the first day of week, 00-53	52
%W	Week number of year, Monday as the first day of week, 00-53	52
%c	Local version of date and time	Mon Dec 31 17:41:00 2018
%C	Century	20
%x	Local version of date	12/31/18
%X	Local version of time	17:41:00
%%	A % character	%
%G	ISO 8601 year	2018
%u	ISO 8601 weekday (1-7)	1
%V	ISO 8601 weeknumber (01-53)	01

Python Math:

یکسری متدها وجود داره در پایتون که برای مسائل ریاضی استفاده می‌شوند. و همچنین یک مدل برای ریاضی وجود داره توی پایتون که اگر شما اون رو ایمپورت کنید میتوانید از متدهای ان استفاده کنید و مسائل خود را حل کنید. که در زیر برخی از این متدها را میاوریم:

```
x = min(5, 10, 25)
y = max(5, 10, 25)

print(x) # 5
print(y) # 25

x = abs(-7.25)

print(x) #make number positive

x = pow(4, 3)

print(x) # equal to 4*4*4
```

The Math Module:

```
import math
```

```

x = math.sqrt(64)

print(x) # 8

import math

x = math.ceil(1.4)
y = math.floor(1.4)

print(x) # returns 2
print(y) # returns 1

import math

x = math.pi

print(x) #3.14...

```

Python JSON:

در پایتون کار با فایل های حسون میتوان از مدل که خودش دارد استفاده کرد . جیسون در واقع فایل تکستی است که به زبان جاوا اسکریپت نوشته شده است باید توجه داشت وقتی که ابجکت های پایتون رو به جیسون تبدیل میکنیم ابجکت ها فرق میکنند که در ادامه ذکر میکنیم انها را. و اینکه برعکس هم این قضیه صدق میکند. متدهای مورد استفاده برای کار با فایل های جیسون رو در پایتون میاوریم.

برای استفاده از فایل استرینگ جیسون در پایتون:

```

import json

# some JSON:
x = '{ "name":"John", "age":30, "city":"New York"}'

# parse x:
y = json.loads(x)

# the result is a Python dictionary:
print(y["age"])

# a Python object (dict):
x = {
    "name": "John",
    "age": 30,
    "city": "New York"

```

برای تبدیل ابجکت پایتون به فایل جیسون:

```

}

# convert into JSON:
y = json.dumps(x)

# the result is a JSON string:
print(y)

```

You can convert Python objects of the following types, into JSON strings:

- dict
- list
- tuple
- string
- int
- float
- True
- False
- None

```

import json

print(json.dumps({"name": "John", "age": 30}))
print(json.dumps(["apple", "bananas"]))
print(json.dumps(("apple", "bananas")))
print(json.dumps("hello"))
print(json.dumps(42))
print(json.dumps(31.76))
print(json.dumps(True))
print(json.dumps(False))
print(json.dumps(None))

```

اشیا در پایتون و در جیسون:

Python	JSON
dict	Object
list	Array
tuple	Array
str	String
int	Number
float	Number
True	true
False	false
None	null

نمونه ای ابجکت پایتون به جیسون که دارای تمام دیتا تایپ های مجاز است:

```

import json

x = {
    "name": "John",
    "age": 30,
    "married": True,
    "divorced": False,
    "children": ("Ann", "Billy"),
    "pets": None,
    "cars": [
        {"model": "BMW 230", "mpg": 27.5},
        {"model": "Ford Edge", "mpg": 24.1}
    ]
}

```

```
print(json.dumps(x))
```

همچنین میتوانیم از جدا کننده ها استفاده کنیم که خوانا تر باشند:
`json.dumps(x, indent=4, separators=(". ", " = "))`

و یا ترتیب بدھیم:

```
json.dumps(x, indent=4, sort_keys=True)
```

Regular Expression:

بحث یا خیر. که از طریق مازول داخلی پایتون `re` استفاده میشود. متدهای `re` را در جدول پایین ذکر میکنیم که از آن ها برای جستجو استفاده میکنیم.

Function	Description
findall	Returns a list containing all matches
search	Returns a Match object if there is a match anywhere in the string
split	Returns a list where the string has been split at each match
sub	Replaces one or many matches with a string

Match Object:

ابجکتی است که دارای اطلاعاتی راجب سرچ و نتیجه ان را در بر دارد. اگر نباشد `None` برمیگرداند.

```

import re

txt = "The rain in Spain"
x = re.search("ai", txt)
print(x) #this will print an object

```

همچنین این ابجکت سه تا متده دارد که میتوانید از آن ها استفاده کنید:

: یک لیست تاپل برمیگرداند که دارای شروع و پایان موقعیت ایتم مج شده است.

```
import re

txt = "The rain in Spain"
x = re.search(r"\bS\w+", txt)
print(x.span())
```

: یک رشته برمیگرداند string.

```
import re

txt = "The rain in Spain"
x = re.search(r"\bS\w+", txt)
print(x.string)
```

: اون بخشی از رشته رو که در آن ایتمی مج شده است را برمیگرداند.

```
import re

txt = "The rain in Spain"
x = re.search(r"\bS\w+", txt)
print(x.group())
```

همچنین در این روند از کاراکتر هایی استفاده میکنی که معانی خاصی دارند به انها Metacharacters میگویند.

Character	Description	Example
[]	A set of characters	"[a-m]"
\	Signals a special sequence (can also be used to escape special characters)	"\d"
.	Any character (except newline character)	"he..o"
^	Starts with	"^hello"
\$	Ends with	"planet\$"
*	Zero or more occurrences	"he.*o"
+	One or more occurrences	"he.+o"

?	Zero or one occurrences	"he.?o"
{}	Exactly the specified number of occurrences	"he.{2}o"
	Either or	"falls stays"
()	Capture and group	

همچنین از \ استفاده میکنیم که در ادامه ان کاراکتر هایی میانند که معانی خاصی ایجاد میکنند، که به آنها Special Sequences میگویند.

Character	Description	Example
\A	Returns a match if the specified characters are at the beginning of the string	"\AThe"
\b	Returns a match where the specified characters are at the beginning or at the end of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string")	r"\bain" r"ain\b"
\B	Returns a match where the specified characters are present, but NOT at the beginning (or at the end) of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string")	r"\Bain" r"ain\B"
\d	Returns a match where the string contains digits (numbers from 0-9)	"\d"
\D	Returns a match where the string DOES NOT	"\D"

	the string DOES NOT contain digits	
\s	Returns a match where the string contains a white space character	"\s"
\S	Returns a match where the string DOES NOT contain a white space character	"\S"
\w	Returns a match where the string contains any word characters (characters from a to Z, digits from 0-9, and the underscore _ character)	"\w"
\W	Returns a match where the string DOES NOT contain any word characters	"\W"
\Z	Returns a match if the specified characters are at the end of the string	"Spain\Z"

و sets ها که داخل [] قرار میگیرند نیز مورد استفاده قرار میگیرند.

Set	Description
[arn]	Returns a match where one of the specified characters (a, r, or n) is present
[a-n]	Returns a match for any lower case character, alphabetically between a and n
[^arn]	Returns a match for any character EXCEPT a, r, and n
[0123]	Returns a match where any of the specified digits (0, 1, 2, or 3) are present
[0-9]	Returns a match for any digit between 0 and 9
[0-5][0-9]	Returns a match for any two-digit

	numbers from 00 and 59
[a-zA-Z]	Returns a match for any character alphabetically between a and z, lower case OR upper case
[+]	In sets, +, *, ., , (), \${} has no special meaning, so [+] means: return a match for any + character in the string

:re های متد ها

findall():

این متد یک لیست که دارای تمامی ایتم های مج شده است را برمیگرداند. ترتیب لیست به همان ترتیبی است که مج ها پیدا شدند. اگر هم ایتمی مج نشود و پیدا نشود لیست به صورت خالی بازگردانده میشود.

```
import re
```

```
txt = "The rain in Spain"
x = re.findall("ai", txt)
print(x)
```

search():

این متد در استرینگ دنبال ایتمی میگردد که طبق الگوی داده شده مج باشد و ان را به صورت match بازمیگرداند. اگر بیشتر از یک ایتم باشد فقط اولی را بازمیگرداند. اگر هم هیچی نباشد که none برمیگرداند.

```
import re
```

```
txt = "The rain in Spain"
x = re.search("\s", txt)

print("The first white-space character is located in
position:", x.start())
```

split():

این متد لیستی برمیگرداند که ایتم ها هر کجا بینشون whitespace باشد رو جدا میکند و هر کدام را به صورت یک ایتم لیست برمیگرداند. همچنین میتوان مشخص کرد که تا چند تا whitespace رو در لیست برگرداند.

```
import re
```

```
txt = "The rain in Spain"
x = re.split("\s", txt, 1)
print(x)
```

sub():

با این متدها میتوانید مجھهای پیدا شده را با مقداری که خودتان مشخص میکنید جایگزین کنید

```
import re
```

```
txt = "The rain in Spain"
x = re.sub("\s", "9", txt)
print(x)
```

pip:

یک پکیج است. پکیج تمام چیزهایی که برای استفاده از مدل‌ها نیاز داریم را شامل میشوند. استفاده از pip بسیار ساده است. برای اینکه بفهمیم نصب داریم یا خیر pip –version استفاده میکنیم. و برای نصب یک پکیج از pip install <package name> d میکنیم. برای انکه ببینیم چه پکیج‌هایی را نصب داریم از list استفاده میکنیم.

Pyunit:

الهام گرفته شده از junit است که برای تست نویسی استفاده میشود از کتابخانه unittest

```
class Calculator:
    def add(self, a, b):
        return a + b

    def subtract(self, a, b):
        return a - b

class TestCalculator(unittest.TestCase):
    def setUp(self):
        self.calc = Calculator()

    def test_add(self):
        result = self.calc.add(2, 3)
        self.assertEqual(result, 5)

    def test_subtract(self):
        result = self.calc.subtract(5, 2)
        self.assertEqual(result, 3)

if __name__ == '__main__':
    unittest.main()
```

Hash Table:

در هش تیبل‌ها برای تاپل‌ها و دیکشنری‌ها استفاده میشوند. که key را به صورت هش ذخیره میکند برای پیدا کردن سریع‌تر انها و دسترسی اسان‌تر. این هش کردن از طریق تابع هش صورت میگیرد.

