**Lambda :**یکی از مهمترین موارد استفاده رایج برای lambda، برنامه نویسی فانکشنال است زیرا پایتون از پارادایم (یا سبک) برنامه نویسی که به عنوان برنامه نویسی فانکشنال شناخته می شود پشتیبانی می کند.  
لامبدا به شما این امکان را می دهد که یک تابع را به عنوان پارامتر به یک تابع دیگر (به عنوان مثال، در map، filter و غیره) ارائه دهید. در چنین مواردی، استفاده از lambda به لطف ایجاد یک تابع یک بارمصرف، مفید است.

## ****# استفاده از lambda با filter****

تابع filter برای فیلتر کردن برخی عناصر خاص از یک دنباله استفاده می شود. دنباله مورد استفاده در این تابع یک [آبجکت ‌iterable](https://www.mongard.ir/one_part/53/creating-iterable-objects-python/) مانند لیست ها ، مجموعه ها ، چندتایی ها و غیره است.

sequences = [10,2,8,7,5,4,3,11,0, 1]

filtered\_answer = filter (lambda x: x > 6, sequences)

print(list(filtered\_answer))

# OUTPUT

# [10, 8, 7, 11]

**# استفاده از lambda با map**

تابع map برای استفاده از یک عملیات خاص برای هر عنصر در یک دنباله استفاده می شود.

sequences = [10,2,8,7,5,4,11]

squared\_result = map (lambda x: x\*x, sequences)

print(list(squared\_result))

# OUTPUT

# [100, 4, 64, 49, 25, 16, 121]

توجه شود که خروجی توابعی همچون map و filter یک object از همان مدل میباشد برای نمایش مقادیر لازم است که یا از حلقه استفاده شود و یا از list استفاده شود

=======================================================================

**underscore :**اولین استفاده از underscoreها در پایتون زمانی هستش که شما میخواین از یک کلمه کلیدی به عنوان اسم یک آرگومان استفاده کنید. به کد زیر دقت کنید:

def show(name, class\_):

pass

مصرف بعدی underscoreها در پایتون برای زمانی هستش که شما میخواید یه عضو از کلاس رو به شکل private یا protected در بیارید:

class Person:

name = 'amir' #public

\_age = 10 #protected

\_\_height = 170 #private

زمانی که قبل از اسم عضوی از کلاس یک underscore قرار بدین اون عضو به شکل protected خواهد بود. اگر دوتا underscore قرار بدین به شکل private خواهد بود.

مصرف بعدی زمانی هستش که شما به قبل و بعد از اسم یک متد در کلاس دوتا underscore اضافه کنید:

class Person:

def \_\_init\_\_(self):

pass

متدهایی که به این شکل اسم گذاری شوند به عنوان special method در پایتون شناخته میشوند. تمامی متدهای builtin پایتون به این شکل نامگذاری میشوند. شما به هیچ وجه نباید اسم متدهاتون رو به این شکل قرار بدین.

آخرین مصرف underscoreها در پایتون زمانی هست که شما مقدار متغیری که دارید براتون مهم نیست:

for \_ in range(10):

print('Hello')

=================================================

**\_\_str\_\_ vs \_\_repr\_\_ :**

این دو متد برای زمانی هستش که شما میخواید یک آبجکت رو چاپ کنید:

class Person:

def \_\_init\_\_(self, name):

self.name = name

p1 = Person('amir')

print(p1)

زمانی که کد بالا رو اجرا میکنیم به شکل زیر نتیجه رو به ما نشون میده:

<\_\_main\_\_.Person object at 0x7f2772b0cf50>

پاسخی که پایتون برای ما آورده آنچنان مطلوب نیستش و نمیشه چیز زیادی ازش فهمید. برای تغییر دادن این رفتار پایتون از متد str استفاده میکنیم:

class Person:

def \_\_init\_\_(self, name):

self.name = name

def \_\_str\_\_(self):

return self.name

p1 = Person('amir')

print(p1)

خب حالا اگه کد بالا رو اجرا کنید نتیحه بهتری رو نمایش میده. در کد بالا به پایتون گفتیم زمانی که یک آبجکت رو چاپ کردیم مقدار name رو نشون بده:

amir

حالا متد repr چیه؟ متد repr هم دقیقا زمانی استفاده میشه که شما یک آبجکت رو صدا بزنید. اما تفاوتی که داره اینکه اگه کدتون رو از طریق ترمینال اجرا کنید متد repr فعال خواهد شد. درواقع متد repr برای زمانی هستش که شما قراره یک نتیجه رو به برنامه نویس دیگه نشون بدید اما متد str برای زمانی هستش که شما میخواید یک نتیجه رو به کاربر نشون بدین.

برای درک این موضوع داخل ترمنیال، با استفاده از دستور python وارد مفسر پایتون بشید.

حالا کد زیر رو وارد میکنیم:

>>> import datetime

>>> now = datetime.datetime.now()

>>> str(now)

'2019-8-15 10:29:34.786394'

>>> repr(now)

datetime.datetime(2019, 8, 15, 10, 29, 34, 786394)

در کد بالا ما زمان الآن رو ریختیم داخل متغیر now.

زمانی که با استفاده از متد str متغیر now رو صدا میزنیم میبینید که نتیجه به شکل یک زمان نشون داده میشه.

اما زمانی که متغیر now رو با متد repr صدا میزنیم نتیجه رو به شکل یک فانکشن میاره.

تفاوت این دو متد اینجا مشخص میشه. متد str برای نشون دادن یک نتیجه به کاربر عادی اما متد repr با هدف دیباگ کردن برنامه و نشون دادن یک پیغام به برنامه نویس دیگه استفاده میشه.

به صورت دیفالت داخل ترمینال متد repr صدا زده می‌شود

=======================================================================

**shallow and deep copy :**

shallow copy یعنی کپی سطحی و deep copy یعنی کپی عمیق.

نکته ای که همین اول کار باید بدونید اینکه این بحث فقط در مورد آبجکت های mutable مثل list, dict, set وجود داره و آبجکت هایی که immutable هستند مثل tuple ها این بحث درموردشون وجود نداره.

زمانی که شما سعی در کپی گرفتن از یک آبجکت mutable رو دارید اون چیزی که شما انتظار دارید با چیزی که پایتون قراره انجام بده متفاوت هستش. به مثال زیر دقت کنید:

a = [1, 2, 3, 4]

b = a

من در خط اول یک لیست دارم. برای کپی گرفتن از این لیست ساده ترین روشی که به ذهنمون میرسه اینکه اون لیست به یک متغیر دیگه منسوب کنیم، دقیقا مثل خط دوم. اما اینکار یک کپی از لیست نمیگیره و فقط یک اشاره گر جدید به اون لیست ایجاد میکنه.

اگر هم بخواید مطمئن بشید میتونید id های متغیرها رو چاپ کنید:

print( id(a) )

print( id(b) )

کد بالا رو که اجرا کنید میبینید که یک عدد رو نشون میده. به این میگیم shallow copy در پایتون. حالا من سعی میکنیم که لیست b رو تغییر بدم:

a = [1, 2, 3, 4]

b = a

b[0] = 11

print(a)

print(b)

در کد بالا من لیست b رو تغییر دادم اما لیست a هم تغییر خواهد کرد. چرا؟ چون b فقط یک اشاره گر بود به لیست a:

[11, 2, 3, 4]

[11, 2, 3, 4]

برای حل این مشکل ما سه تا متد داریم: list(), dict(), set()

b = list(a)

b[0] = 11

این سه تا متد deep copy میگیرن. حالا اگه لیست b رو تغییر بدین میبینید که لیست a تغییری نکرده.

[1, 2, 3, 4] #a

[11, 2, 3, 4] #b

اما این سه تا متد در زمانی که شما آبجکت های تو در تو دارید به درستی کار نخواهند کرد.

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

b = list(a)

b[4][0] = 55

print(a) # [1, 2, 3, 4, [55, 6]]

print(b) # [1, 2, 3, 4, [55, 6]]

همونطور که میبینید زمانی که لیست هامون تو در تو هستند باز هم هردو لیست تغییر میکنن. برای حل کامل مشکل میتونید از ماژول copy استفاده کنید:

import copy

a = [1, 2, 3, 4]

b = copy.copy(a) # shallow copy

c = copy.deepcopy(a) # deep copy

در این ماژول دوتا متد داریم که برامون هم shallow copy و هم deep copy میگیرن.

=================================================

**return None in pyhton :**

در پایتون زمانی که داخل فانکشن هاتون مقداری رو return نکنید، پایتون به صورت اتوماتیک None رو return خواهد کرد. به سه روش این کار انجام میشه.

روش اول زمانی هستش که شما به صورت صریح اعلام کنید که میخواید None رو return کنید:

def show(value):

if value:

return value

else:

return None

در مثال بالا اگه به فانکشن مقداری رو ارسال نکنید اون فانکشن برای شما None رو return خواهد کرد.

در روش دوم شما فقط کلمه return رو مینویسید:

def show(value):

if value:

return value

else:

return

در مثال بالا هم خود پایتون به صورت اتوماتیک مقدار None رو return میکنه.

و روش سوم زمانی هستش که شما بلاک else رو کلا نمینویسید:

def show(value):

if value:

return value

در مثال بالا هم اگه مقداری به فانکشن ارسال نشه پایتون به صورت اتوماتیک مقدار None رو return خواهد کرد.

=================================================

**json in pyhton :**

در این آموزش ، نحوه ذخیره داده ها در پایتون با استفاده از ماژول JSON را بررسی می کنیم. ما همچنین نحوه استفاده از متدهای json.dump () و json.dumps () ، متدهای json.load () و json.loads () و تفاوت آنها را یاد می گیریم. سرانجام، ما به نحوه serialize و deserialize کردن JSON به Object در پایتون خواهیم پرداخت.

## ****# چرا داده ها را در پایتون با استفاده از ماژول JSON ذخیره می کنیم؟****

1. توسعه دهندگان را قادر می سازد تا ساختارهای داده ساده را در یک فایل ریخته و در صورت نیاز بارگذاری کنند.

2. با استفاده از JSON می توان داده ها را بین برنامه های پایتون به اشتراک گذاشت.

3. فرمت JSON از پلتفرم یا زبان مستقل است. وقتی داده ها را با فرمت JSON ذخیره می کنید، می توانید به راحتی از آنها در سایر زبان های برنامه نویسی نیز استفاده کنید.

4. یادگیری آن ساده است و در قالب قابل حمل ارائه می شود.

## ****# استفاده از json.dump()****

برای استفاده از تابع json.dump () ابتدا ماژول json را وارد کنید. برای وارد کردن ماژول json ، از import json استفاده کنید. () json.dump به نوشتن داده ها در یک فایل JSON کمک می کند.

**Syntax:**

json.dump(data, file)

تابع () json.dump دو آرگومان دارد:

1. data: داده هایی که باید در یک فایل JSON نوشته شوند.

2. file: یک شیء فایل که می تواند برای ذخیره داده ها استفاده شود.

بیایید یک برنامه سریع برای ذخیره مجموعه ای از اعداد در یک فایل JSON ایجاد کنیم. برای ذخیره مجموعه اعداد ، از تابع json.dump () استفاده می کنیم:

import json

numbers = [10, 20, 30, 70, 191, 23] #create a set of numbers

filename = 'numbers.json' #use the file extension .json

with open(filename, 'w') as file\_object: #open the file in write mode

json.dump(numbers, file\_object) # json.dump() function to stores the set of numbers in

numbers.json file

در این برنامه، مجموعه اعداد را در numbers.json ذخیره می کنیم. پسوند .json نشان می دهد که فایل حاوی داده هایی با فرمت JSON است.

سپس به فایل در حالت 'w' (حالت نوشتن) دسترسی پیدا می کنیم تا بتوانیم داده ها را در یک فایل JSON نوشت. سرانجام ، تابع json.dump () مجموعه اعداد را در فایل number.json ذخیره می کند.

این برنامه خروجی ترمینال ندارد، اما وقتی پرونده number.json را باز می کنیم، داده های زیر را می بینیم:

[10, 20, 30, 70, 191, 23]

**# استفاده از json.dumps()**

متد () json.dumps می تواند برای تبدیل یک شی پایتون به یک رشته JSON استفاده شود.

**Syntax:**

json.dumps(data)

تابع () json.dumps یک پارامتر می گیرد ، یعنی داده هایی که باید به رشته JSON تبدیل شوند.

بیایید به مثال زیر نگاهی بیندازیم:

import json

data = {

'Name' : 'Felix',

'Occupation' : 'Doctor'

}

dict\_1 = json.dumps(data) # converting dictionary to JSON

print(dict\_1) # {'Name' : 'Felix','Occupation' : 'Doctor'}

## ****# تفاوت بین dump و dumps****

1. متد dump () دو پارامتر (داده و فایل) ، در حالی که متد dumps () تنها یک پارامتر (داده) می گیرد.

2. برخلاف متد dumps، متد dump با عملیات فایل ترکیب می شود.

**# استفاده از json.load()**

ما از تابع json.load برای خواندن یک فایل JSON استفاده می کنیم. تابع json.load () یک آرگومان می گیرد که شیء فایل است.

**Syntax:**

json.load(file\_object)

فرض کنید، ما یک فایل JSON به نام student.json داریم که حاوی اشیاء JSON است.

{

"name": "Felix",

"Subjects": ["English", "Political Science"]

}

بیایید یک کد برای خواندن داده های ذخیره شده در پرونده student.json با استفاده از تابع json.load بنویسیم.

import json

with open(r,'student.json') as file\_object:

data = json.load(file\_object)

print(data) # {"name": "Felix", "Subjects": ["English", "Political Science"]}

تابع json.load () فایل JSON را تجزیه می کند و یک دیکشنری به نام data را برمی گرداند.

**# استفاده از json.loads()**

ما از متد json.loads () برای تجزیه یک رشته JSON و بازگشت یک شی پایتون مانند دیکشنری استفاده می کنیم. متد json.loads () محتویات فایل را به عنوان یک رشته می گیرد.

**Syntax:**

json.loads(json\_string)

**Example:**

import json

# JSON string:

dict\_1 = {

"Name": "Felix Maina",

"Contact Number": 0712345678,

"Email": "fely@gmail.com",

}

# parse dict\_1:

y = json.loads(dict\_1)

# the result is a Python dictionary:

print(y) #{ "Name": "Felix Maina", "Contact Number": 0712345678,"Email": "fely@gmail.com", }

در اینجا ، رشته dict\_1 با استفاده از متد () json.loads تجزیه می شود که یک دیکشنری به نام y را برمی گرداند.

توجه: تفاوت اصلی بین json.loads () و json.load () این است که json.loads () رشته ها را می خواند در حالی که json.load () برای خواندن فایل ها استفاده می شود.

## ****# سریال سازی داده های JSON در پایتون****

Serialization فرآیند تبدیل یک نوع داده بومی به فرمت JSON است.

ماژول JSON یک شیء دیکشنری پایتون را به یک شی JSON تبدیل می کند. متدهای json.dump () و json.dumps () برای سریال سازی داده های پایتون به فرمت JSON استفاده می شود.

بیایید به یک مثال با استفاده از متد json.dump () نگاهی بیندازیم:

import json

# Data to be written

details = {

"name": "Felix Maina",

"years": 21,

"school": "Makerere"

}

# Serializing JSON and writing JSON file

with open("details.json", "w") as file\_object:

json.dump(details, file\_object) # {"name": "Felix Maina", "years": 21, "school": "Makerere"}

در اینجا، ما یک دیکشنری پایتون را به یک فایل فرمت JSON با نام details.json تبدیل می کنیم.

متد json.dumps () یک شی پایتون را به یک رشته JSON تبدیل می کند ، همانطور که در زیر نشان داده شده است:

import json

# Data to be written

details = {

"name": "Felix Maina",

"years": 21,

"school": "Makerere"

}

# Serializing JSON

json\_string = json.dumps( details )

print( json\_string ) #{"name": "Felix Maina", "years": 21, "school": "Makerere"}

## ****# تغییر JSON به Object در پایتون****

Deserialization فرآیند تبدیل داده های JSON به یک نوع داده بومی است. در اینجا، ما داده های JSON را به یک دیکشنری در پایتون تبدیل می کنیم.

ما از متد json.loads () برای تغییر داده های JSON به یک شی پایتون استفاده می کنیم. روش json.load () نیز برای تغییر یک قالب فرمت JSON به یک شی پایتون استفاده می شود.

**Example:**

# importing the module

import json

# creating the JSON data as a string

data = '{"Name" : "Felix", "status" : "married"}'

print("data before deserailizing")

print(data) #json string

# deserailizing the data

h = json.loads(data)

print("data after deserailizing")

print(h) #python dictionary

**Output**:

# data before deserailizing

{"Name" : "Felix", "status" : "married"}

#data after deserailizing

{'status': 'married', 'Name': 'Felix'}

اجازه دهید یک فایل ایجاد کرده و نام آن را cars.json بگذارید. این فایل باید دارای اطلاعات زیر باشد:

{

"name": "Suzuki",

"year": 2001,

"model": "GDF10"

}

حالا بیایید با استفاده از تابع load () این فایل را از حالت طبیعی خارج کنیم:

import json

# opening the JSON file

data = open('cars.json','r')

print("Datatype before deserialization : ")

print(data) # prints the contents of the file

# deserailizing the data

h = json.load(data)

print("Datatype after deserialization : ")

print(h) # prints a python dictionary

=================================================

**underscore :**

اولین استفاده از underscoreها در پایتون زمانی هستش که شما میخواین از یک کلمه کلیدی به عنوان اسم یک آرگومان استفاده کنید. به کد زیر دقت کنید:

def show(name, class):

pass

همونطور که میدونید کلمه  class یک کلمه کلیدی رزرو شده پایتون هستش و شما نمیتونید ازش به عنوان اسم آرگومان استفاده کنید. به همین دلیل میتونید با اضافه کردن یک underscore به آخر کلمه اون رو از حالت رزرو خارج کنید.

def show(name, class\_):

pass

مصرف بعدی underscoreها در پایتون برای زمانی هستش که شما میخواید یه عضو از کلاس رو به شکل private یا protected در بیارید:

class Person:

name = 'amir' #public

\_age = 10 #protected

\_\_height = 170 #private

زمانی که قبل از اسم عضوی از کلاس یک underscore قرار بدین اون عضو به شکل protected خواهد بود. اگر دوتا underscore قرار بدین به شکل private خواهد بود.

مصرف بعدی زمانی هستش که شما به قبل و بعد از اسم یک متد در کلاس دوتا underscore اضافه کنید:

class Person:

def \_\_init\_\_(self):

pass

متدهایی که به این شکل اسم گذاری شوند به عنوان special method در پایتون شناخته میشوند. تمامی متدهای builtin پایتون به این شکل نامگذاری میشوند. شما به هیچ وجه نباید اسم متدهاتون رو به این شکل قرار بدین.

آخرین مصرف underscoreها در پایتون زمانی هست که شما مقدار متغیری که دارید براتون مهم نیست:

for i in range(10):

print('Hello')

مثال بالا رو اگه اجرا کنید مقدار Hello ده بار براتون اجرا میشه. اما اگه دقت کنید متغیر i که در حلقه for قرار دادید اصلا کاربرد نداره. پس به جاش میتونید به شکل زیر کار کنید:

for \_ in range(10):

print('Hello')

=================================================

**iterate :**

در این بخش میخوایم در مورد سه فعل مهم در پایتون صحبت کنیم: Iterable, Iterator, Iteration

هر سه این کلمات از کلمه Iterate مشتق شدن. در زبان انگلیسی کلمه Iterate به معنی تکرار کردن است.

اول مفهوم Iteration میگم. در پایتون به عمل تکرار کردن میگیم Iteration. ما در زبان پایتون دو نوع Iteration داریم. اولی حلقه for و دومی حلقه while. به مثال زیر دقت کنید:

for i in range(1, 11):

print(i)

در مثال بالا ما یک نمونه از Iteration رو داریم. پس به خود عمل تکرار کردن میگیم Iteration.

حالا Iterable رو میگیم. به آبجکت هایی که بتونیم روی اونها عمل تکرار رو انجام بدیم میگیم Iterable.

یا به عبارت دیگه به هر آبجکتی که بتونیم روش عمل Iteration رو انجام بدیم میگیم Iterable.

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

for num in nums:

print(num)

در مثال بالا میبینید که ما داخل لیست nums حلقه زدیم و تونستیم به مقدارهاش دسترسی داشته باشیم. الآن لیست nums یک آبجکت Iterable هستش.

مورد بعدی Iterator هستش. به آبجکت هایی که بتونن آخرین وضعیت خودشون رو حفظ کنن میگیم iterator

آبجکت هایی که بشه روشون متد next رو اجرا کرد میتونن آخرین وضعیت خودشون رو حفظ کنن.

در حال حاضر متد next روی لیستی که در بالا ساختیم قابل اجرا نیستش.

print( dir( nums ) )

اگه کد بالا رو اجرا کنید تمام اتریبیوت ها و متدهایی که در nums در دسترس هستش رو به ما نشون میده:

['\_\_add\_\_', '\_\_class\_\_', '\_\_contains\_\_', '\_\_delattr\_\_', '\_\_delitem\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getitem\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_iadd\_\_', '\_\_imul\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_iter\_\_', '\_\_le\_\_', '\_\_len\_\_', '\_\_lt\_\_', '\_\_mul\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_reversed\_\_', '\_\_rmul\_\_', '\_\_setattr\_\_', '\_\_setitem\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']

در نتیجه ای که پایتون به ما نشون داده میبینید که متد \_\_next\_\_ وجود نداره.

اما به جاش متد \_\_iter\_\_ رو داریم که میتونه لیست nums رو به یک Iterator تبدیل کنه

inums = iter(nums)  
or  
inums = nums.\_\_iter\_\_()

در کد بالا با استفاده از متد iter لیست nums رو به یک Iterator تبدیل کردیم.

حالا اگه یک بار دیگه کد زیر رو اجرا کنید میبینید که متد next در دسترس هستش

print( dir( inums ) )

['\_\_class\_\_', '\_\_delattr\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_iter\_\_', '\_\_le\_\_', '\_\_length\_hint\_\_', '\_\_lt\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_next\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_setstate\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_']

حالا میتونید از متد next استفاده کنید و به آیتم های داخل لیست inums به صورت تکی دسترسی پیدا کنید:

print( next(inums) )

print( next(inums) )

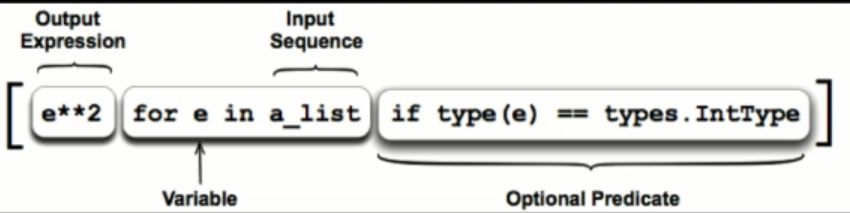
print( next(inums) )

print( next(inums) )  
or  
print( inums.\_\_next\_\_() )

حالا لیست inums یک Iterator هستش.

=================================================

**list comprehenstion :**

  
ist comprehension پایتون راهی برای ایجاد یک لیست بر اساس لیستی دیگر است. list comprehension معمولاً برای فیلتر کردن موارد از لیست یا تغییر مقادیر موجود در لیست استفاده می شود. list comprehension در داخل پرانتز قرار می گیرد. هنگامی که با لیست ها کار می کنید، ممکن است بخواهید یک لیست بر اساس محتویات یک دنباله موجود ایجاد کنید. به عنوان مثال، ممکن است بخواهید یک لیست بر اساس دنباله ای از کاراکترها ایجاد کنید. یا ممکن است بخواهید لیستی را ایجاد کنید که محتویات یک لیست دیگر را در دو ضرب کند.

اینجاست که list comprehension وارد می‌شود.

ساختار نگارش کلی list comprehension در پایتون به شکل زیر هستش:

[output expression forloop if sentence]

میخوام یک لیستی بسازم و از ۱ تا ۲۰ رو بریزم داخلش. برا اینکار میتونید از list comprehension استفاده کنید. به شکل زیر:

v = [i for i in range(1, 21)]

print(v)

تمامی list comprehensionها در پایتون داخل یک جفت براکت قرار میگیرن. داخل براکت یک حلقه for زدیم و نتیجه رو داخل متغیر v ذخیره کردیم.

حالا اگه بخوایم میتونیم شرط هم اضافه کنیم.

v = [i for i in range(1, 21) if i%2==0]

print(v)

همونطور که میبینید تونستم یک شرط اضافه کنم تا بتونم فقط اعدادی رو که زوج هستند ذخیره کنم.

=================================================

**generator :**

آیا تا به حال مجبور شده اید با مجموعه داده ای آنقدر بزرگ کار کنید که حافظه دستگاه شما را تحت تأثیر قرار دهد؟ یا شاید شما یک تابع پیچیده دارید که هر بار که فراخوانی می شود نیاز به حفظ یک حالت داخلی دارد، اما این تابع برای توجیه ایجاد کلاس خود، بسیار کوچک است. در این موارد و موارد دیگر، ژنراتورها و عبارت yield پایتون اینجا هستند تا به شما کمک کنند.

Generator ها روشی ساده‌تر برای ساختن object های iterable هستند

## ****# استفاده از generatorهای پایتون****

توابع ژنراتور که با PEP 255 معرفی شدند، نوع خاصی از تابع هستند که یک تکرار کننده تنبل(lazy iterator) را برمی گرداند. اینها اشیایی هستند که می توانید مانند یک لیست روی آنها حلقه بزنید. با این حال، بر خلاف لیست ها، تکرار کننده های تنبل محتویات خود را در حافظه ذخیره نمی کنند.

اکنون که ایده ای تقریبی از کاری که یک ژنراتور انجام می دهد دارید، ممکن است تعجب کنید که آنها در عمل چگونه به نظر می رسند. بیایید به دو مثال نگاهی بیندازیم. در قسمت اول، نحوه عملکرد ژنراتورها را به طور کلی بررسی خواهیم کرد. سپس، بزرگنمایی می کنید و هر مثال را با دقت بیشتری بررسی می کنید.

### ****+ مثال اول: خواندن فایل‌های بزرگ****

یک مورد معمول استفاده از ژنراتورها کار با جریان داده یا فایل های بزرگ مانند فایل های CSV است. این فایل های متنی داده ها را با استفاده از کاما از ستون ها جدا می کنند. این فرمت یک روش رایج برای اشتراک گذاری داده ها است. حال، اگر بخواهید تعداد ردیف های یک فایل CSV را بشمارید، چه؟ بلوک کد زیر یک راه برای شمارش آن ردیف ها را نشان می دهد:

csv\_gen = csv\_reader("some\_csv.txt")

row\_count = 0

for row in csv\_gen:

row\_count += 1

print(f"Row count is {row\_count}")

با نگاهی به این مثال، ممکن است انتظار داشته باشید که csv\_gen یک لیست باشد. برای پر کردن این لیست، ()csv\_reader یک فایل را باز می کند و محتویات آن را در csv\_gen بارگذاری می کند. سپس، برنامه روی لیست تکرار می شود و row\_count را برای هر ردیف افزایش می دهد.

این یک توضیح منطقی است، اما اگر فایل بسیار بزرگ باشد، آیا این طرح همچنان کار می کند؟ اگر فایل از حافظه ای که در دسترس دارید بزرگتر باشد چه؟ برای پاسخ به این سوال، فرض کنید ()csv\_reader فقط فایل را باز می کند و آن را در یک آرایه می خواند:

def csv\_reader(file\_name):

file = open(file\_name)

result = file.read().split("\n")

return result

این تابع یک فایل مشخص را باز می کند و از file.read() همراه با .split() برای اضافه کردن هر خط به عنوان یک عنصر جداگانه به یک لیست استفاده می کند. اگر بخواهید از این نسخه csv\_reader() در بلوک کد شمارش ردیفی که در بالا مشاهده کردید استفاده کنید، خروجی زیر را دریافت خواهید کرد:

Traceback (most recent call last):

File "ex1\_naive.py", line 22, in <module>

main()

File "ex1\_naive.py", line 13, in main

csv\_gen = csv\_reader("file.txt")

File "ex1\_naive.py", line 6, in csv\_reader

result = file.read().split("\n")

MemoryError

در این حالت، open() یک شی generator را برمی‌گرداند که می‌توانید با تنبلی آن را خط به خط تکرار کنید. با این حال، file.read().split() همه چیز را به یکباره در حافظه بارگذاری می کند و باعث ایجاد خطای حافظه می شود.

قبل از اینکه این اتفاق بیفتد، احتمالاً متوجه خواهید شد که رایانه شما دچار کندی سرعت شده است. حتی ممکن است لازم باشد برنامه را با وقفه Keyboard متوقف کنید. بنابراین، چگونه می توانید این فایل های داده عظیم را مدیریت کنید؟ نگاهی به تعریف جدیدی از csv\_reader():

def csv\_reader(file\_name):

for row in open(file\_name, "r"):

yield row

در این نسخه، فایل را باز می‌کنید، آن را تکرار می‌کنید و یک ردیف ایجاد می‌کنید. این کد باید خروجی زیر را بدون خطای حافظه تولید کند:

Row count is 64186394

اینجا چه اتفاقی دارد میافتد؟ خب، شما اساساً ()csv\_reader را به یک تابع generator تبدیل کرده اید. این نسخه یک فایل را باز می کند، در هر خط حلقه می زند و به جای بازگرداندن هر سطر، آن را yield می کند.

شما همچنین می توانید یک عبارت generator (که به آن generator comprehension نیز گفته می شود) تعریف کنید که دارای نحو بسیار مشابه list comprehension است. به این ترتیب، می توانید از ژنراتور بدون فراخوانی تابع استفاده کنید:

csv\_gen = (row for row in open(file\_name))

این یک راه مختصرتر برای ایجاد لیست csv\_gen است. به زودی در مورد دستور yield پایتون بیشتر خواهید آموخت. در حال حاضر، فقط این تفاوت کلیدی را به خاطر بسپارید:

* استفاده از yield منجر به یک شی generator می شود
* استفاده از return تنها در خط اول فایل کار میکند

### ****+  مثال دوم: ساخت دنباله بینهایت****

بیایید دنده ها را عوض کنیم و به تولید دنباله بی نهایت نگاه کنیم. در پایتون، برای به دست آوردن یک دنباله محدود، range() را فراخوانی کرده و آن را در یک list ارزیابی می کنید:

>>> a = range(5)

>>> list(a)

[0, 1, 2, 3, 4]

با این حال، تولید یک دنباله بی نهایت نیاز به استفاده از یک ژنراتور دارد، زیرا حافظه رایانه شما محدود است:

def infinite\_sequence():

num = 0

while True:

yield num

num += 1

این بلوک کد کوتاه و شیرین است. ابتدا متغیر num را مقداردهی اولیه کرده و یک حلقه بی نهایت راه اندازی می کنید. سپس، بلافاصله num را yield میکنید تا بتوانید حالت اولیه را بگیرید. این عمل range() را تقلید می کند.

پس از yield، عدد را با 1 افزایش می دهید. اگر این را با حلقه for امتحان کنید، خواهید دید که واقعا بی نهایت به نظر می رسد:

>>> for i in infinite\_sequence():

... print(i, end=" ")

...

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29

30 31 32 33 34 35 36 37 38 39 40 41 42

[...]

6157818 6157819 6157820 6157821 6157822 6157823 6157824 6157825 6157826 6157827

6157828 6157829 6157830 6157831 6157832 6157833 6157834 6157835 6157836 6157837

6157838 6157839 6157840 6157841 6157842

KeyboardInterrupt

Traceback (most recent call last):

File "<stdin>", line 2, in <module>

این برنامه تا زمانی که آن را به صورت دستی متوقف نکنید به اجرای آن ادامه خواهد داد.

به جای استفاده از حلقه for، می‌توانید مستقیماً () next را روی generator پایتون فراخوانی کنید. این به ویژه برای آزمایش یک ژنراتور در کنسول مفید است:

>>> gen = infinite\_sequence()

>>> next(gen)

0

>>> next(gen)

1

>>> next(gen)

2

>>> next(gen)

3

در اینجا، شما یک generator به نام gen دارید که به صورت دستی با فراخوانی مکرر next () آن را تکرار می کنید. این به عنوان یک بررسی سلامت عقل عالی عمل می کند تا مطمئن شوید که ژنراتورهای شما خروجی مورد انتظار شما را تولید می کنند.

**+  مثال سوم: تشخیص پالیندروم**

شما می توانید از دنباله‌های بی نهایت به طرق مختلف استفاده کنید، اما یکی از کاربردهای عملی آنها در ساخت آشکارسازهای پالیندروم است. یک آشکارساز پالیندروم تمام دنباله‌های حروف یا اعداد را که پالیندروم هستند پیدا می‌کند. اینها کلمات یا اعدادی هستند که به صورت یکسان به جلو و عقب خوانده می شوند، مانند 121. ابتدا، آشکارساز پالیندروم عددی خود را تعریف کنید:

def is\_palindrome(num):

# Skip single-digit inputs

if num // 10 == 0:

return False

temp = num

reversed\_num = 0

while temp != 0:

reversed\_num = (reversed\_num \* 10) + (temp % 10)

temp = temp // 10

if num == reversed\_num:

return num

else:

return False

در مورد درک ریاضیات اساسی در این کد زیاد نگران نباشید. فقط توجه داشته باشید که تابع یک عدد ورودی می گیرد، آن را برعکس می کند و بررسی می کند که آیا عدد معکوس شده با عدد اصلی یکسان است یا خیر. اکنون می توانید از ژنراتور توالی بی نهایت خود برای دریافت لیست در حال اجرا از تمام پالیندروم های عددی استفاده کنید:

>>> for i in infinite\_sequence():

... pal = is\_palindrome(i)

... if pal:

... print(i)

...

11

22

33

[...]

99799

99899

99999

100001

101101

102201

KeyboardInterrupt

Traceback (most recent call last):

File "<stdin>", line 2, in <module>

File "<stdin>", line 5, in is\_palindrome

در این حالت، تنها اعدادی که روی کنسول چاپ می شوند، همان اعدادی هستند که به جلو یا عقب هستند.

اکنون که یک مورد استفاده ساده برای یک generator توالی نامتناهی دیدید، بیایید عمیق‌تر به نحوه عملکرد ژنراتورها بپردازیم.

## ****#  درک generatorهای پایتون****

تا اینجا، شما در مورد دو روش اصلی ایجاد generatorها یاد گرفته اید: با استفاده از توابع ژنراتور و عبارات ژنراتور. حتی ممکن است درک شهودی از نحوه عملکرد ژنراتورها داشته باشید. بیایید یک لحظه وقت بگذاریم تا آن دانش را کمی واضح تر کنیم.

توابع ژنراتور درست مانند توابع معمولی به نظر می رسند و عمل می کنند، اما با یک مشخصه تعیین کننده. توابع ژنراتور به جای return از کلمه کلیدی yield پایتون استفاده می کنند. تابع generator را که قبلا نوشتید به یاد بیاورید:

def infinite\_sequence():

num = 0

while True:

yield num

num += 1

این به نظر یک تعریف تابع معمولی است، به جز عبارت yield پایتون و کدی که به دنبال آن است. yield نشان می دهد که یک مقدار به تماس گیرنده ارسال می شود، اما بر خلاف return، بعد از آن از تابع خارج نمی شوید.

در عوض، وضعیت(state) تابع به خاطر سپرده می شود. به این ترتیب، زمانی که next() بر روی یک generator فراخوانی می شود (به طور صریح یا ضمنی در یک حلقه for)، متغیر شماره قبلی افزایش یافته و سپس دوباره به دست می آید. از آنجایی که توابع generator شبیه توابع دیگر هستند و بسیار شبیه به آنها عمل می کنند، می توانید فرض کنید که عبارات generator بسیار شبیه سایر comprehensionهای موجود در پایتون هستند.

### ****+  ساخت عبارات ژنراتور پایتون(Generator Expressions)****

مانند list comprehensionها، عبارات genereator به شما اجازه می دهد تا به سرعت یک شی genereator را تنها در چند خط کد ایجاد کنید. آنها همچنین در موارد مشابهی که از list comprehensionها استفاده می شود، با یک مزیت اضافی مفید هستند: می توانید آنها را بدون ساختن و نگه داشتن کل شی در حافظه قبل از تکرار ایجاد کنید. به عبارت دیگر، هنگام استفاده از عبارات genereator، هیچ جریمه حافظه ای نخواهید داشت. این مثال از مربع کردن برخی اعداد را در نظر بگیرید:

>>> nums\_squared\_lc = [num\*\*2 for num in range(5)]

>>> nums\_squared\_gc = (num\*\*2 for num in range(5))

هر دو nums\_squared\_lc و nums\_squared\_gc اساساً یکسان به نظر می رسند، اما یک تفاوت اساسی وجود دارد. آیا می توانید آن را تشخیص دهید؟ وقتی هر یک از این اشیاء را بررسی می کنید، به اتفاقاتی که می افتد نگاه کنید:

>>> nums\_squared\_lc

[0, 1, 4, 9, 16]

>>> nums\_squared\_gc

<generator object <genexpr> at 0x107fbbc78>

شی اول از براکت برای ساختن یک لیست استفاده کرد، در حالی که شی دوم با استفاده از پرانتز یک عبارت generator ایجاد کرد. خروجی تأیید می کند که یک شی generator ایجاد کرده اید و از یک لیست متمایز است.

### ****+  مشخص کردن پرفورمنس ژنراتور پایتون****

قبلاً یاد گرفتید که ژنراتورها یک راه عالی برای بهینه سازی حافظه هستند. در حالی که یک ژنراتور دنباله نامتناهی یک مثال افراطی از این بهینه‌سازی است، بیایید نمونه‌های مربع‌سازی اعدادی را که به‌تازگی دیده‌اید تقویت کنیم و اندازه اشیاء حاصل را بررسی کنیم. می توانید این کار را با یک فراخوانی به sys.getsizeof() انجام دهید:

>>> import sys

>>> nums\_squared\_lc = [i \* 2 for i in range(10000)]

>>> sys.getsizeof(nums\_squared\_lc)

87624

>>> nums\_squared\_gc = (i \*\* 2 for i in range(10000))

>>> print(sys.getsizeof(nums\_squared\_gc))

120

در این مورد، لیستی که از list comprehension دریافت می کنید 87624 بایت است، در حالی که شی generator تنها 120 بایت است. این به این معنی است که لیست بیش از 700 برابر بزرگتر از شی generator است!

با این حال یک چیز را باید در نظر داشت. اگر لیست کوچکتر از حافظه در دسترس ماشین در حال اجرا باشد، در این صورت list comprehension می تواند سریعتر از عبارت generator معادل ارزیابی شود. برای کشف این موضوع، اجازه دهید نتایج حاصل از دو comprehension بالا را جمع بندی کنیم. می‌توانید با cProfile.run() اینکار را انجام دهید:

>>> import cProfile

>>> cProfile.run('sum([i \* 2 for i in range(10000)])')

5 function calls in 0.001 seconds

Ordered by: standard name

ncalls tottime percall cumtime percall filename:lineno(function)

1 0.001 0.001 0.001 0.001 <string>:1(<listcomp>)

1 0.000 0.000 0.001 0.001 <string>:1(<module>)

1 0.000 0.000 0.001 0.001 {built-in method builtins.exec}

1 0.000 0.000 0.000 0.000 {built-in method builtins.sum}

1 0.000 0.000 0.000 0.000 {method 'disable' of '\_lsprof.Profiler' objects}

>>> cProfile.run('sum((i \* 2 for i in range(10000)))')

10005 function calls in 0.003 seconds

Ordered by: standard name

ncalls tottime percall cumtime percall filename:lineno(function)

10001 0.002 0.000 0.002 0.000 <string>:1(<genexpr>)

1 0.000 0.000 0.003 0.003 <string>:1(<module>)

1 0.000 0.000 0.003 0.003 {built-in method builtins.exec}

1 0.001 0.001 0.003 0.003 {built-in method builtins.sum}

1 0.000 0.000 0.000 0.000 {method 'disable' of '\_lsprof.Profiler' objects}

در اینجا، می‌توانید ببینید که جمع کردن همه مقادیر در list comprehension، تقریباً یک سوم زمان جمع‌آوری در generator را به خود اختصاص داده است. اگر سرعت مشکل است و حافظه مشکلی ندارد، احتمالاً list comprehension ابزار بهتری برای کار است.

به یاد داشته باشید، list comprehensions، لیست های کامل را برمی گرداند، در حالی که عبارات generator، ژنراتورها را برمی گرداند. ژنراتورها چه از یک تابع یا یک عبارت ساخته شده باشند یکسان کار می کنند. استفاده از یک عبارت فقط به شما امکان می دهد generatorهای ساده را در یک خط مشخص کنید، با بازدهی فرضی در پایان هر تکرار داخلی.

دستور yield پایتون مطمئناً پایه‌ای است که تمام عملکرد ژنراتورها بر آن استوار است، بنابراین بیایید به نحوه عملکرد yield در پایتون بپردازیم.

## ****#  درک دستور yield پایتون****

در کل، yield یک دستور نسبتاً ساده است. وظیفه اصلی آن کنترل جریان یک تابع generator به روشی شبیه به دستورات بازگشتی است. همانطور که در بالا به طور خلاصه ذکر شد، دستور yield پایتون چند ترفند در آستین خود دارد.

هنگامی که یک تابع generator را فراخوانی می کنید یا از یک عبارت generator استفاده می کنید، یک تکرار کننده خاص به نام generator را برمی گردانید. شما می توانید این generator را به یک متغیر اختصاص دهید تا از آن استفاده کنید. وقتی متدهای خاصی را روی ژنراتور فراخوانی می کنید، مانند next()، کد داخل تابع تا yield اجرا می شود.

وقتی دستور yield پایتون زده می‌شود، برنامه اجرای تابع را به حالت تعلیق در می‌آورد و مقدار بازده را به تماس‌گیرنده برمی‌گرداند. (در مقابل، return اجرای تابع را به طور کامل متوقف می کند.) هنگامی که یک تابع به حالت تعلیق در می آید، وضعیت آن تابع ذخیره می شود. این شامل هر گونه اتصال متغیر محلی به generator، اشاره گر دستورالعمل، پشته داخلی، و هرگونه رسیدگی به استثنا می شود.

این به شما امکان می دهد هر زمان که یکی از متدهای generator را فراخوانی کردید، اجرای تابع را از سر بگیرید. به این ترتیب، تمام ارزیابی عملکرد بلافاصله پس از yield، از سر گرفته می شود. با استفاده از چند عبارت yield پایتون می توانید این را در عمل مشاهده کنید:

>>> def multi\_yield():

... yield\_str = "This will print the first string"

... yield yield\_str

... yield\_str = "This will print the second string"

... yield yield\_str

...

>>> multi\_obj = multi\_yield()

>>> print(next(multi\_obj))

This will print the first string

>>> print(next(multi\_obj))

This will print the second string

>>> print(next(multi\_obj))

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

StopIteration

نگاهی دقیق تر به آخرین فراخوانی next() بیندازید. می توانید ببینید که اجرا با یک traceback متوقف شده است. این به این دلیل است که ژنراتورها، مانند همه تکرارکننده ها(iterators)، می توانند خسته شوند. مگر اینکه generator شما بی نهایت باشد، می توانید فقط یک بار از طریق آن تکرار کنید. هنگامی که همه مقادیر ارزیابی شدند، تکرار متوقف می شود و حلقه for خارج می شود. اگر از next() استفاده کردید، در عوض یک استثنا صریح StopIteration دریافت خواهید کرد.

yield را می توان به روش های بسیاری برای کنترل جریان اجرای ژنراتور استفاده کرد. تا آنجایی که خلاقیت شما اجازه می دهد، می توان از چندین عبارت yield پایتون استفاده کرد.