**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 (قابل تغییر) مثل list, dict, set وجود داره و آبجکت هایی که immutable (غیرقابل تغییر) 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.

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

**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.  
# 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)

# 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)

# به آبجکت هایی که بتونن آخرین وضعیت خودشون رو حفظ کنن میگیم iterator . یا به زبانی دیگرآبجکت 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']

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

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

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) )

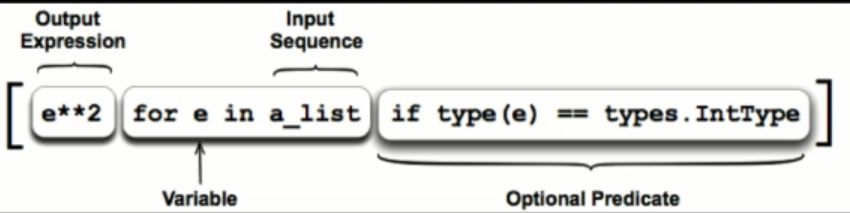
print( next(inums) )

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

حالا لیست inums یک Iterator هستش.

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

**list comprehension :**

  
list 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.

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 :**

(پیش نیاز ===> list comprehension و iterate )

# یک روش برای ساخت آبجکت هایی که خاصیت iterator دارند میباشد (آبجکت هایی که بتوانیم روی آن‌ها پیمایش انجام دهیم و حالت آخرین وضعیت را در خود ذخیره میکند )

# دو روش برای ساخت generator ها وجود دارد : function و expression (comprehension)

# برای ساخت generator از طریق function ما باید به جای استفاده از return میبایست yield استفاده شود

# یکی از تفاوتهایی که generator با function دارد این است که در function میبایست فقط یک بار از return استفاده شود اما در generator به هر تعداد که لازم باشد میتوانیم از yield استفاده کنیم

# میتوانیم از ()next استفاده کنیم و به مقادیر generator یک به یک دسترسی پیدا کنیم

# در‌واقع generator ها خاصیت iterator را دارند به این صورت که حالت آخرین وضعیت را ذخیره میکنند

# مفهوم lazy evaluation یا ارزیابی تنبل در generator ها وجود دارد به این صورت که تا مجبور نباشن مقداری را برای شما حساب نمیکنند و نمایش نمیدهند پس به خاطر این ویژگی در منابع ما صرفه جویی می‌شود و به همان میزان که لازم است برای ما محاسبات انجام می‌شود

# کاربرد اصلی generator ها هنگام کار با دیتاهای بسیار بزرگ میباشد چرا که در منابع ما صرفه جویی می‌شود

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

**args and kwargs :**

در پایتون از args و kwargs برای هندل کردن آرگومان های پیش بینی نشده استفاده میشود. این آرگومانها میتوانند به شکل ساده یا دیکشنری ارسال شوند

به مثال زیر دقت کنید

def show(name):

print( f'Hello {name}' )

show('amir', 'ali')

در مثال بالا یه دونه فانکشن داریم به اسم show که انتظار داره یک مقدار رو بهش بفرستیم و اون یک مقدار رو بگیره و چاپ کنه. اما ما دوتا مقدار رو فرستادیم و این کد به ما ارور میده.

اینجا میتونید از \*args استفاده کنید. args به شما اجازه میده که پارامترهای اضافی رو بگیرید.

def show(name, \*args):

print( f'Hello {name}' )

show('amir', 'ali')

اگه کد بالا رو اجرا کنید دیگه به شما ارور نمیده چون args اومده و پارامترهای اضافی رو داخل خودش نگهداشته. این مقادیر اضافی را به صورت تاپل در خودش ذخیره میکند

حالا اگه من به فانکشن show به صورت key-value مقدار بدم چه اتفاقی میوفته؟

show('amir', 'ali', age=20)

کد بالا رو اگه اجرا کنید باز هم به شما ارور میده. اینجا دیگه args نمیتونه برای شما کاری انجام بده.

برای گرفتن مقدارهای اضافی که به شکل key-value میان میتونید از \*\*kwargs استفاده کنید:

def show(name, \*args, \*\*kwargs):

print( f'Hello {name}' )

show('amir', 'ali', age=20)

بازم اگه کد بالا رو اجرا کنید دیگه ارور نخواهید داشت چون kwargs اونها رو داخل خودش ذخیره کرده.این مقادیر اضافی را به صورت دیکشنری در خودش ذخیره میکند

(( توجه شود که حتماً لازم نیست که از اسم های args و kwargs استفاده شود فقط \* و \*\* قبل از اسم‌ها استفاده شود ))

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

**آشنایی عمیق با functionها :**

# arguman و parameter این دو کلمه معمولاً به جای هم استفاده می‌شوند اما یک تفاوت کوچکی باهم دارن . اون مقداری که ارسال میکنیم به یک function به آن arguman گفته می‌شود اما اون مقداری که از function میگیرید parameter گفته می‌شود

# نکته ی بسیار مهم : هر چیزی که در پایتون وجود دارد یک کلاس است قسمت مهم این نکته این است که هر چیزی که در پایتون وجود دارد رفتار یکسانی دارند .منظور از objects همان کلاس میباشد

# هنگامیکه functionی (تابع) ساخته می‌شود میتوان با آن همانند object رفتار کرد یعنی میتوان آن را به یک متغییر داد اما باید دقت داشت که پرانتز ها را نگذاریم   
(( پرانتز ها به این معنی است که فانکشن مورد نظر را صدا بزن و آن را اجرا کن ))

# نکته ی بعدی این است که object اون function با متغییری که به آن دادیم کاملاً متفاوت است یعنی اگر مثلاً show حذف شود اما x بازهم کار میکند اما اگر show صدا زده شود error خواهد داد

# به اون functionهایی که پارامتر ورودیشان یک function دیگر است *higher-order گفته می‌شود   
خوده پایتون یک فانکشن heigher-order به نام map دارد*

*# توجه شود که* در فانکشن های تودرتو لازم است که در آخر فانکشن داخلی return شود

# نکته مهمی که درباره فانکشن های تودرتو وجود دارد این است که نمیتوان به طور مستقیم به فانکشن داخلی دسترسی داشت . به فانکشن های داخلی inner و به فانکشن های خارجی outer گفته می‌شود

# فانکشن های فرزند میتوانند به پارامترهای ورودی پدر دسترسی داشته باشند

# به این فانکشن های فرزند که به این صورت به پارامترهای پدر دسترسی دارند lexical closure گفته می‌شود

# با استفاده از متد \_\_call\_\_ میتوانیم objectهایمان را callable کنیم

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

**argument unpacking :**

در پایتون امکان این وجود دارد که به صورت همزمان چندین پارامتر را ارسال کرده تا فانکشن به صورت خودکار آنها را دریافت و استفاده کند. آرگومان ها میتوانند به شکل یک لیست یا دیکشنری ارسال شوند

هنگامیکه از دیکشنری برای ارسال آرگومان ها استفاده میکنیم توجه داشته باشید که نام keyها در دیکشنری با نام پارامترهای ورودی تابع برابر باشد اما ترتیب آن مهم نیست

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

**decorator :**

# برای ساخت decoratorها باید از فانکشن های تودرتو استفاده کرد

# هنگامیکه بالای یک فانکشن یک decorator قرار میدهیم به طور خودکار function را به عنوان آرگومان به function مربوط به decorator ارسال میکند

# نکته بسیار مهمی که باید به آن توجه کرد این است که decoratorها از پایین به بالا اجرا می‌شوند

# اگر functionی که بر روی آن decorator قرار گرفته باشد خود پارامتر ورودی داشته باشد این پارامتر ورودی در فانکشن های inner فرستاده می‌شود (( خود فانکشن به فانکشن outer فرستاده می‌شود ))  
در‌واقع در مثال t همان text است

# functools.wraps تمام متا دیتا هایی که گم شده‌اند رو پیدا میکند و نمایش میدهد

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

**assert :**

# کلمه کلیدی assert برای تست کردن کد توسط برنامه نویس استفاده میشود. (‌ assert ها به برنامه نویس کمک میکنند که یک باگ یا مشکلی رو داخل برنامه پیدا و برطرف کنند )( نه نشون دادن ارور به کاربر )  
#از assert به هیچ عنوان نباید برای اعتبارسنجی داده های کاربران یا نشان دادن یک پیغام به کاربر استفاده شود.

# دلیلی که نباید از assert برای اعتبار سنجی داده‌های کاربر استفاده کرد این است که میتوان در پایتون assert را غیر فعال کرد ( python -o filename.py )

# هنگامیکه شرط جلوی assert را داخل tuple قرار دهیم در این صورت همیشه true میباشد

# از raise برای نمایش یک exception استفاده میشه اما از assert برای بررسی یک شرط

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

**“\_\_if \_\_name\_\_ : “\_\_main :**

# یک روشی است برای تشخیص اینکه ماژول (فایل) مورد نظر به صورت مستقیم اجرا شده است یا در ماژول دیگر ایمپورت شده و در آن ماژول اجرا می‌شود

# زمانی که یک ماژول پایتونی میسازید و میخواهید آن را اجرا کنید قبل از اینکه کدهای پایتون اجرا شود خوده پایتون به صورت اتوماتیک یکسری special variable را ست میکند یکی از این special variableها \_\_name\_\_ میباشد

# اگر ماژول بصورت مستقیم اجرا شود مقدار \_\_name\_\_ برابر با \_\_main\_\_ میباشد

# اگر بصورت غیر مستقیم یعنی به وسلیه import در ماژول دیگر اجرا شود مقدار \_\_name\_\_ برابر با اسم فایل مورد نظر میباشد

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

**encapsulation :**

# encapsulation یا کپسوله سازی در پایتون به این معنی است که اجازه دسترسی مستقیم به attributeها رو ندیم و در عوض با استفاده از متدهای setter, getter, deleter, این کار رو انجام میدیم

# این مفاهیم مواقعی مورد استفاده قرار میگیرند که ما attribute را به صورت private یا protect تعریف کرده باشیم

# برای تعریف attribute به صورت private و protect و public به صورت زیر عمل میکنیم :

class Person:

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

self.address = address # public

self.\_name = name # protect

self.\_\_age = age #private

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

**context manger :**

فرض کنید که ما یک عمل اصلی داریم و قصد داریم قبل و بعد از این عمل اصلی یکسری کار برای ما انجام شود مثلاً فرض کنید که ما یکسری منابع محدود داریم که باید از آن‌ها به درستی استفاده کنیم

( هنگامیکه از فایلی شروع به خواندن میکنیم در آخر باید مطمئن باشیم که اون فایل بسته می‌شود یا هنگامیکه به سوکتی متصلی می‌شویم در آخر باید از قطع ارتباط مطمئن شویم )

در مثال اول مشکلی که وجود دارد به این صورت است که برای بستن فایل ما به صورت دستی این عمل را انجام داده‌ایم که این خود ایراد دارد چرا که اگر در حین خواندن از فایل یا عملیات روی آن به exception بخوریم فایل بسته نخواهد

در مثال دوم with مطمئن می‌شود که در آخر فایل بسته می‌شود

برای نوشتن یک context manger لازم است متدهای \_\_enter\_\_ و \_\_exit\_\_ نوشته شود

زمانی که در \_\_enter\_\_ مقداری را return میکنید به متغیری که جلوی as قرار دارد فرستاده

exc\_type و exc\_val و exc\_tb برای هندل کردن ارور هایی است که ممکن است با آن مواجه شوید

هنگامیکه سه مقدار بالا را print میکنید اروری از طرف پایتون برای شما نمایش داده می‌شود برای اینکه این ارور نمایش داده نشود لازم است که در آخر \_\_exit\_\_ حتماً return True نوشته شود

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

**contextlib :**

( پیش نیاز ===> context manager )

در پایتون ماژول contextlib به ما کمک میکنه که بتونیم از context managerها به صورت ساده تر و سریع تر استفاده کنیم (‌contextlib جزو standard library های پایتون می‌باشد)

اولین مورد دکوریتور contextmanger میباشد که یک فانکشن معمولی را به context manger تبدیل میکند (‌هر کدی که قبل از yield باشد بخش enter میباشد و هر کدی که بعد از yield باشد بخش exit میباشد ) (‌هر مقداری هم که قرار است به عنوان as استفاده شود باید به وسیلیه yield فرستاده شود )

دومین مورد دکوریتور ContextDecorator میابشد که یک کلاس را به دکوریتور تبدیل میکند

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

**\_\_new\_\_:**

در پایتون متد new مسئول ساخت یک کلاس جدید هست. متد new تنها متدیست که قبل از متد init اجرا شده و میتواند مانع فعال شدن متد init شود به زبان دیگر اگر بخواهید نحوه ی ایجاد شدن کلاس را تغییر دهید از \_\_new\_\_ استفاده میکنید

خیلی جاها گفته شده که \_\_init\_\_ سازنده ی کلاس هست اما این جمله اشتباه میباشد چرا که سازنده ی کلاس \_\_new\_\_ میباشد .

\_\_init\_\_ فقط initializer (آغازگر) میباشد و کارهای اولیه را انجام میدهد

توجه شود که \_\_new\_\_ اولین مقداری که دریافت میکند cls میباشد یعنی خود کلاس بر خلاف \_\_init\_\_ که اولین مقداری که میگیرد self میباشد

دقت کنید که بعد از اینکه تغییراتی در \_\_new\_\_ به وجود آورید حتماً باید فانکشن new کلاس پدر را return کنید هماننده کد زیر :

return super().\_\_new\_\_(cls, \*args, \*\*kwargs)

به طور کلی نحوه ی کارکردن \_\_new\_\_ به این شکل میباشد:زمانی که نمونه‌ای از کلاس در حال ایجاد شدن میباشد \_\_new\_\_ سریع فعال می‌شود و کلاس خودش را به عنوان object به cls ارسال میکند ( به خودش ارسال میکند ) و در ادامه میتوانیم استفاده هایی را که مد نظر داشتیم را بکنیم

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

**\_\_call\_\_:**

در پایتون متد call به ما اجازه میده که نمونه هایی که از کلاس ها ایجاد میکنیم رو بتونیم مثل یک فانکشن صدا بزنیم به زبان دیگر هنگامیکه نمونه‌ای که از کلاس ساخته شده است به صورت فانکشن صدا زده شود این متد فراخوانی می‌شود

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

**memory management :**

# منظور از memory در‌واقع همان ram میباشد که پایتون از آن برای ذخیره سازی اطلاعات مربوطه و لازم استفاده میکند

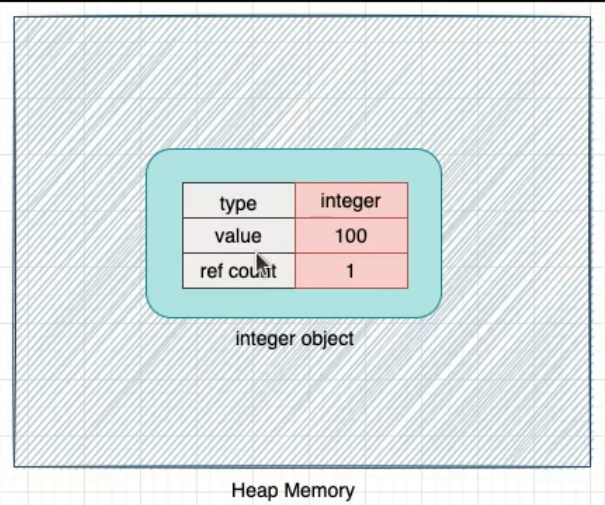
# ما دو نوع حافظه داریم stack و heap :  
 stack : حافظه های ثابت هستند ( سایز آن‌ها ثابت است و کم یا زیاد نمیشود )   
 heap : بر اساس نیاز برنامه میتوانند کم یا زیاد شوند پس به صورت داینامیک مقادیر را ذخیره میکند   
پایتون از نوع heap استفاده میکند

# پیاده‌سازی اصلی زبان پایتون cpython میباشد .CPython به زبان C نوشته شده است و بایت کد پایتون را تفسیر می کند.   
الگوریتم ها و ساختارهای مدیریت حافظه در کد CPython هستند. ( از ابزار های زبان c برای مدیریت حافظه استفاده میکند

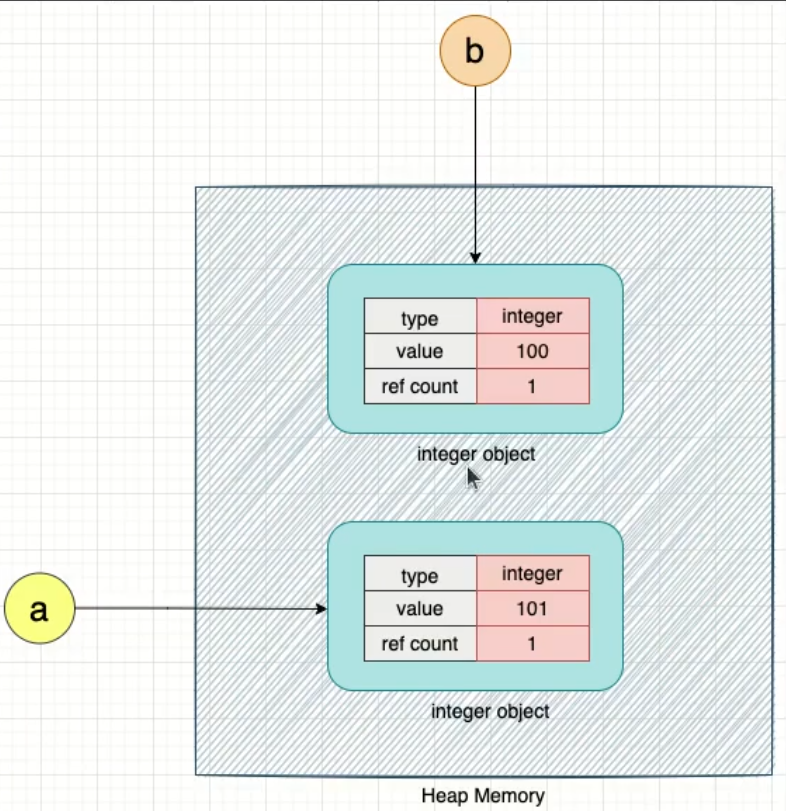
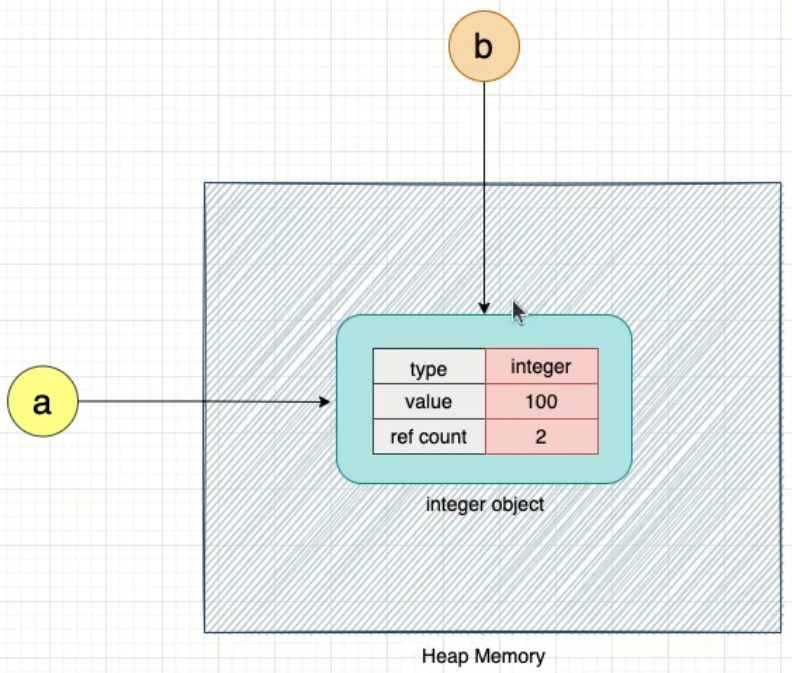
# در زبان‌هایی ماننده c برنامه نویس باید حافظه را مدیریت کند اما در پایتون این مدیریت در اختیار خوده زبان برنامه نویسی و مفسر آن میباشد ( زمان های خیلی کمی وجود دارد که برنامه نویس پایتون لازم باشد خود به مدیریت حافظه بپردازد )

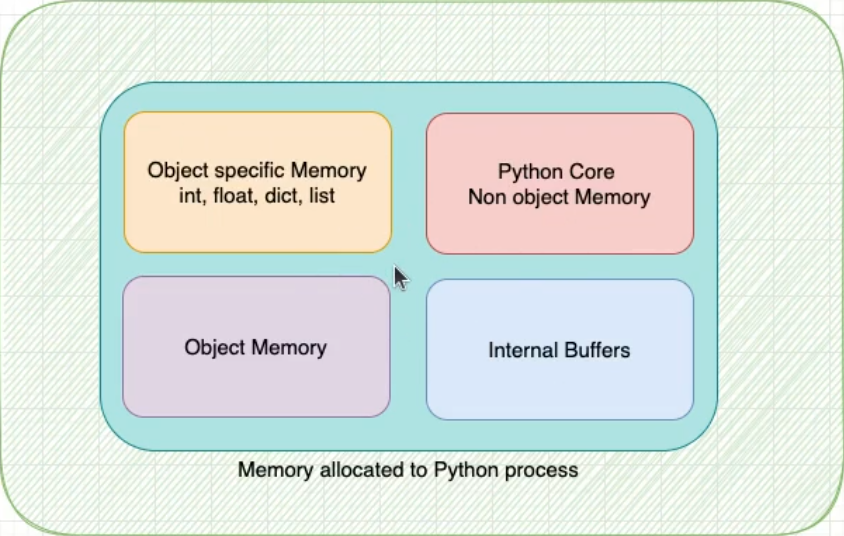
# در زبان برنامه نویسی c دو ابزار وجود دارد برای مدیریت حافظه :  
 malloc : برای زمانی است که شما بخواهید بخشی از حافظه را دریافت و اشغال کنید   
 free : برای آزاد سازی آن بخشی از حافظه که اشغال شده است   
خوده پایتون از این متدها برای مدیریت حافظه استفاده میکند

# پایتون برای ذخیره کردن مقداری در حافظه همراه آن دو مقدار دیگر هم ذخیره میکند

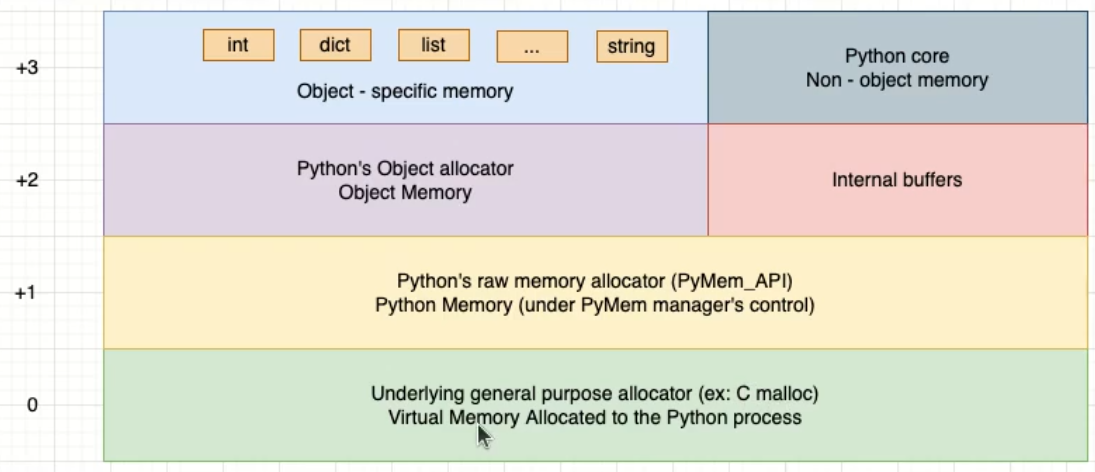


type که نوع مقدار را مشخص میکند , value که مقدار مربوطه میباشد و ref count که تعداد متغیر هایی که به این مقدار اشاره میکنند

  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
# پایتون برای مدیریت بهینه حافظه , حافظه را به ۴ دسته بندی مختلف تقسیم میکند :



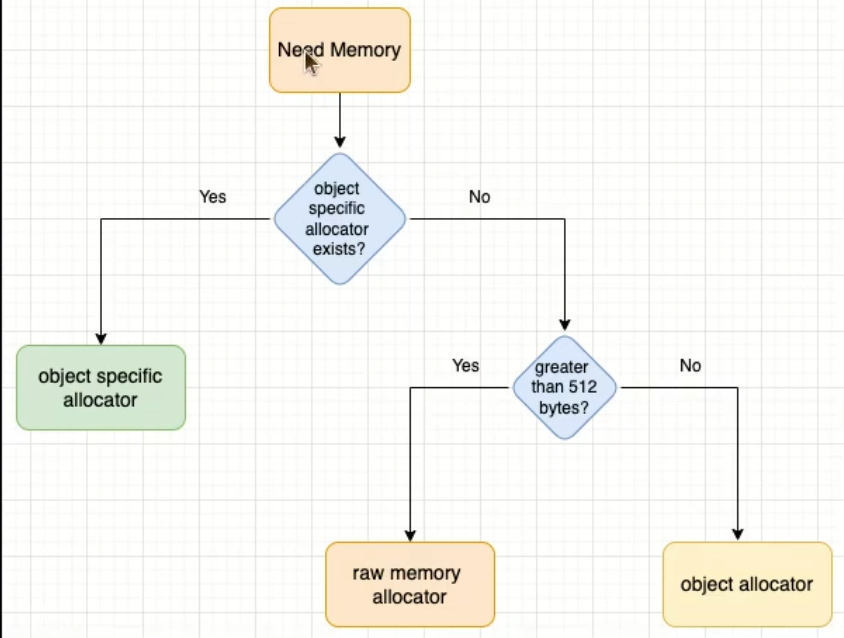
در عکس هر ۴ قسمت اندازه یکسانی دارند که در اصل اینطور نیست

  
# فقط لایه صفر با سیستم عامل کار میکند و دیگر لایه ها همچین اجازه ای را ندارند

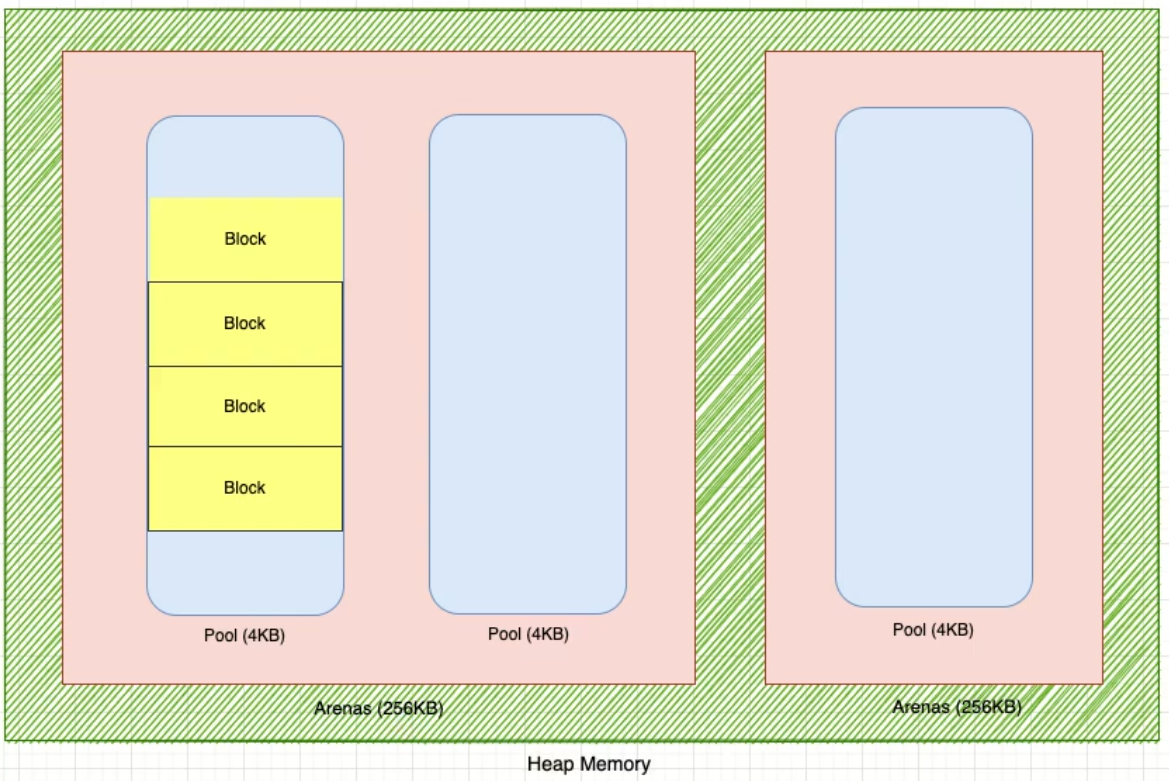
# لایه ۱ یا raw memory allocator برای زمانی استفاده می‌شود که مقدار شما از ۵۱۲ بایت بیشتر باشد

# لایه ۲ یا object allocator برای مقادیر استفاده می‌شود که کوچکتر مساوی ۵۱۲ بایت باشد

# لایه ۳ یا object -specific memory که برای آبجکت های خاص پایتون استفاده می‌شود

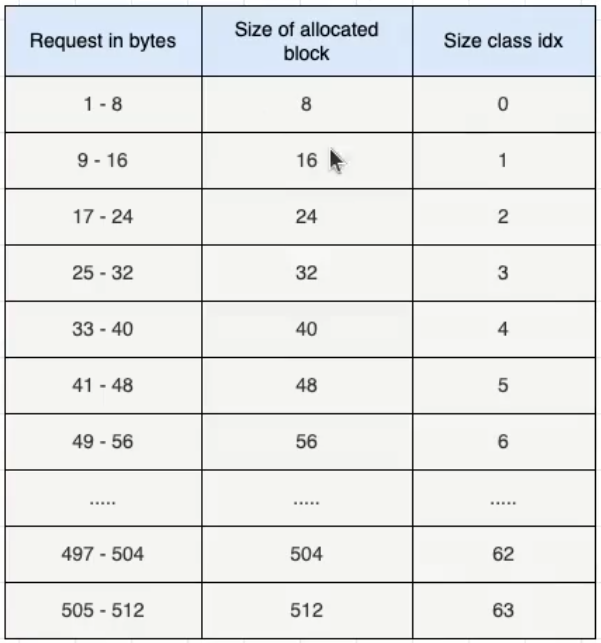


# پایتون heap memory را به صورت زیر قسمت بندی میکند و از آن استفاده میکند :



هنگامیکه قرار است آبجکتی ذخیره شود پایتون ۲۵۶ کیلو بایت از حافظه را اختصاص میدهد که به آن arena گفته می‌شود   
در ادامه به ۶۴ قسمت تقسیم میکند که به آن pool گفته می‌شود و سایر آن‌ها ۴ کیلو بایت میباشد ( میتواند سایز pool کم یا زیاد شود )   
در ادامه pool را دسته بندی و تقسیم‌بندی میکند که به آن‌ها block گفته می‌شود و برای ذخیره مقادیر از این block ها استفاده می‌شود ( داخل هر بلاک فقط و فقط یک آبجکت ذخیره می‌شود) (‌همینطور یک آبجکت داخل یک بلاک ذخیره می‌شود حال هرچه قدر مقدار آبجکت بزرگ باشد )

# سایر بلاک ها براساس جدول زیر مشخص می‌شود :

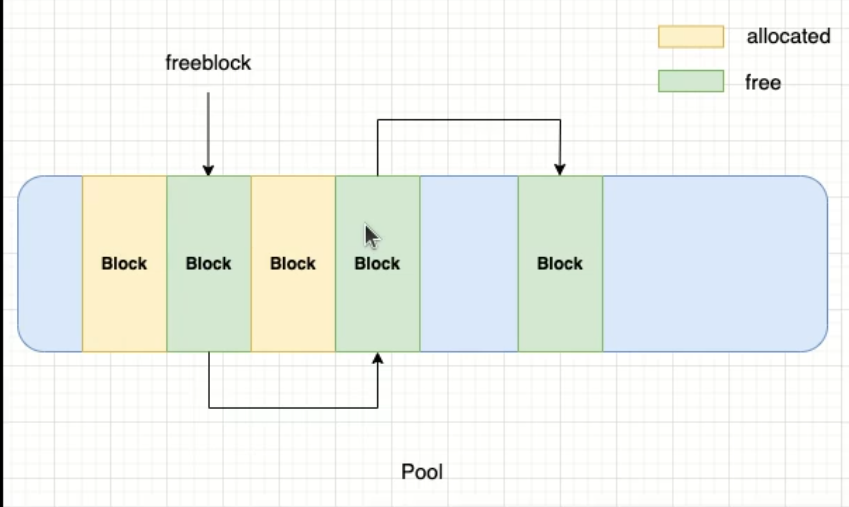


بر اساس جدول بالا حداقل سایز بلاک ۸ بایت و حداکثر ۵۱۲ بایت میباشد

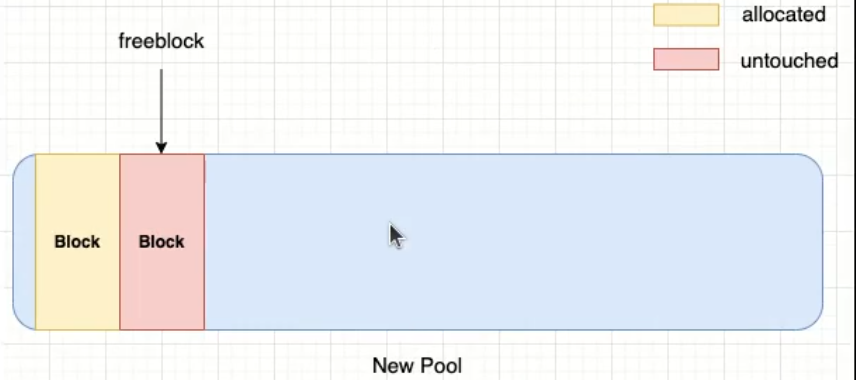
# نکته ی بسیار مهمی که باید به آن توجه کرد این است که block هایی که داخل pool ذخیره می‌شود باید یک سایز باشند

# تمام pool هایی که سایز کلاسشون(size class idx) (‌سایز block ها) یکسان میباشد به هم به وسیله doubly link list وصل هستند

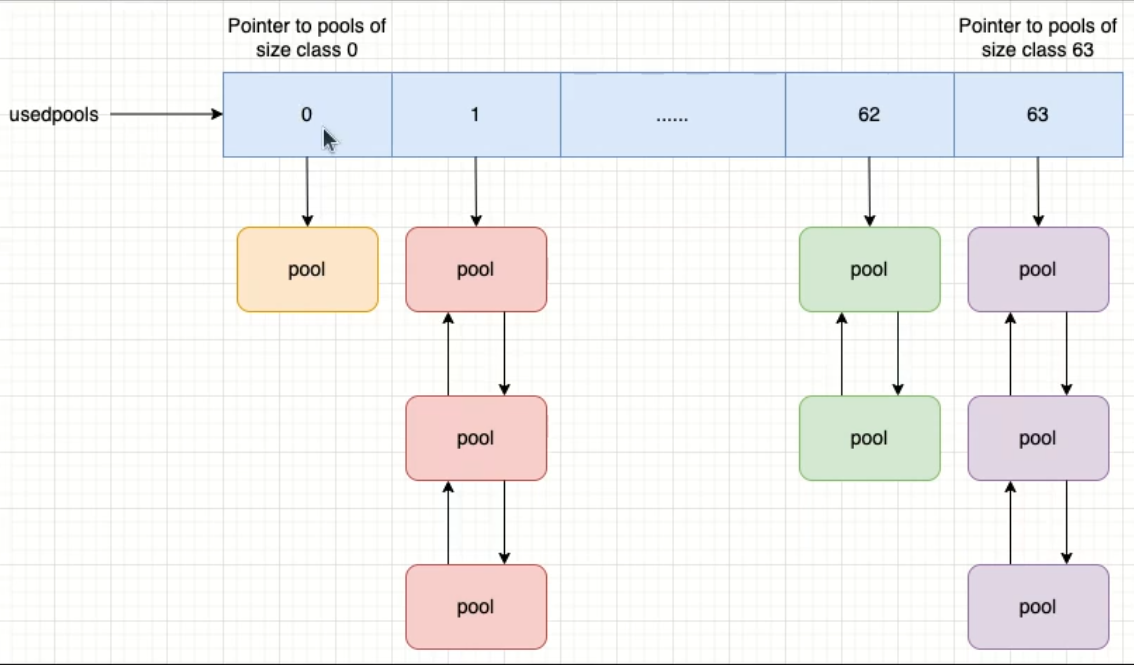
 # براساس تصویر زیر block ها مرتب می‌شوند یعنی بلاک هایی که قبلاً allocate بوده‌اند و الان free شده‌اند جابجا می‌شوند :



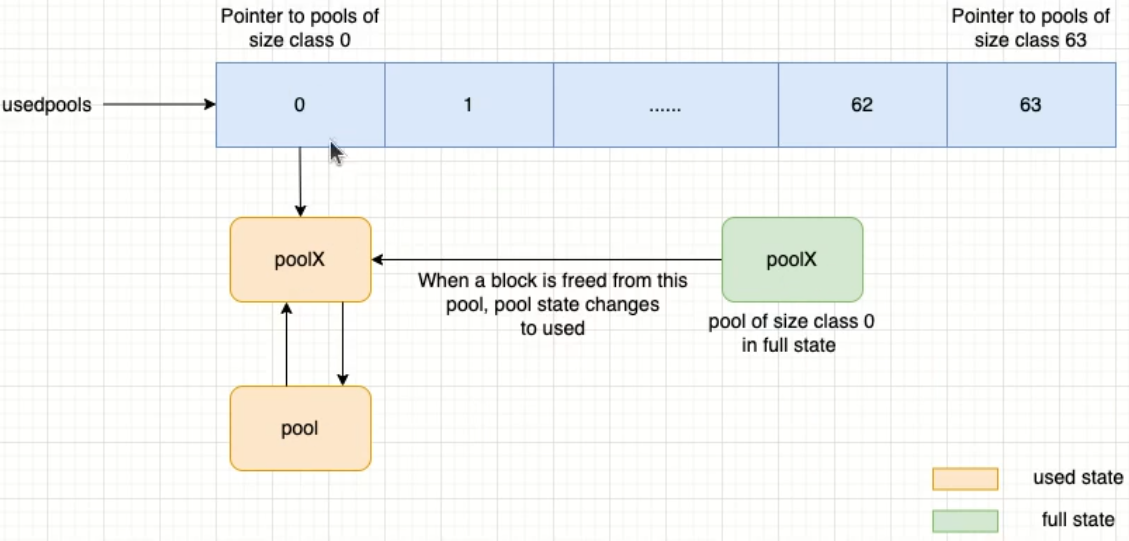
# زمانیکه pool مربوطه تمام می‌شود و تمام block هایی که میتوانست داشته باشد allocate می‌شود برنامه از arena درخواست pool جدید میکند این pool در همان ابتدا به block های مساوی تقسیم نمیشود بلکه تنها دو بلاک ساخته می‌شود بلاک اول برای آبجکتی که قصد ذخیره کدن آن را داشتید و بلاک بعدی ساخته می‌شود که بتوانید در آن آبجکت آینده خود را ذخیره کنید(از قبل بلاک بندی نمیکند):



# python یا درواقع cpython برای مدیریت بهت حافظه آرایه ای را ایجاد میکند که از szidx برای index آرایه استفاده میکند که در هر ایندکس poolهایی که هنوز کامل پر نشده‌اند را در آن ذخیره میکند :



# توجه شود poolهایی که قبلاً پر بوده‌اند و الان خالی شده‌اند دوباره در این آرایه اضافه می‌شوند و در بالا قرار میگیرند :



# زمانی که arena خالی شده باشد arena مربوطه دوباره در اختیار سیستم عامل قرار میگیرد

# در هر زبان برنامه نویسی مفهومی وجود دارد به نام garbage collector یا آشغال جمع کند که وظیفه آن حذف آبجکت هایی است که دیگر در حافظه کاربردی ندارند

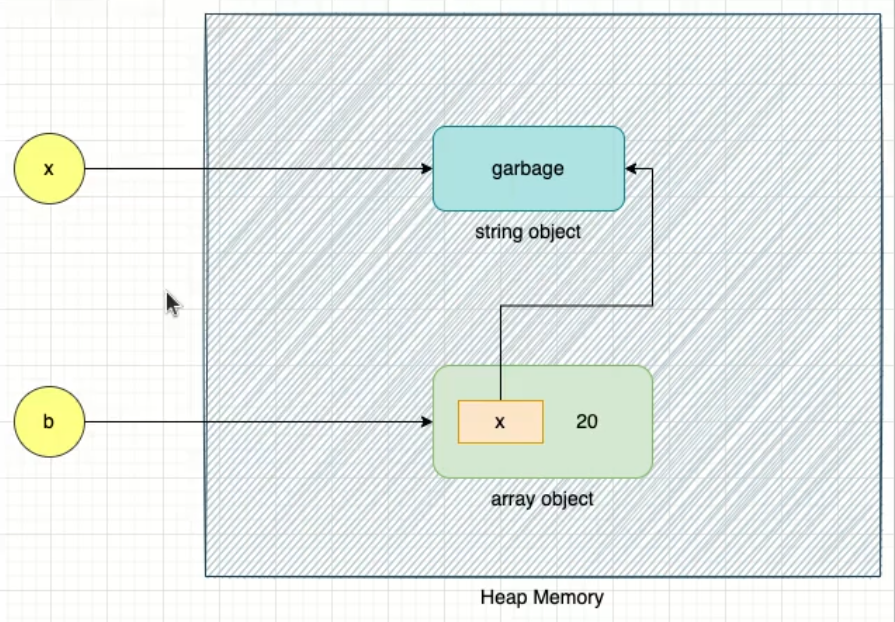
# برای مشاهده ref count یک مقدار میتوانیم از دستورات زیر استفاده کنیم :

import sys

a = ‘memory’

sys.getrefcount(a)

خروجی دستور sys.getrefcount در این مثال ۲ میباشد چرا که علاوه بر a خوده این دستور این مقدار را به عنوان var دریافت کرده است :



بر اساس تصویر بالا ref count متغیر x برابر با ۳ و لیست b برابر با ۲ میباشد

# برای حذف یک متغیر میتوانیم از دوحالت x = None یا del x استفاده کنیم

# توجه شود که فقط متغیر حذف می‌شود و مقدار تا زمانی که ref count برابر با صفر نشود حذف نمیشود زمانی که ref count یک مقدار برابر صفر شود به وسیله garbage collector حذف می‌شود

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

**aggregate and annotate in django :**

اگر تفاوت aggregate و annotate را درک کنیم میتوانیم خیلی راحتتر کوئری هایی با استفاده از این ۲ مورد بنویسیم :

aggregate : عملیاتی که aggregate انجام میدهد به این صورت است که روی تمام object های یک کوئری یک عملیات خاصی را انجام میدهد مثلاً مجموع قیمت یک کوئری ( فیلد قیمت تمام objectها باهم جمع شد ) (‌ نتیجه را بصورت یک dictionary برمیگرداند )

annotate : اما annotate با aggregate متفاوت است به این صورت که عملیاتی که انجام میدهد روی تک تک object های یک کوئری اعمال می‌شود مثلاً تعداد کتاب‌های هر نویسنده (‌درست است که annotate هم روی یک کوئری اعمال می‌شود اما هر object مقداری متفاوت دارد ) ( نتیجه را بصورت یک queryset برمیگرداند )

وقتی از annotate استفاده میکنید بدلیل اینکه خروجی annotate یک queryset میباشد میتوانیم بر روی آن filter یا order\_by استفاده کنیم اما زمانی که از aggregate استفاده میکنیم همچین کاری ممکن نیست

بدلیل نکته بالا ما میتوانیم روی annotate از aggregate استفاده کنیم اما برعکس این قضیه ممکن نیست

Book.objects.annotate(num\_authors=Count("authors")).aggregate(Avg("num\_authors"))

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