**به نام خدای رنگین کمان**

**در این قسمت با مفاهیمی که لازمن برای ادامه تست نویسی آشنا میشیم مثله :**

* **توابع لامبدا**
* **متد islice از ماژول itertools**
* **Generator**
* **مفهوم iterate و ساخت آبجکت های iterable**
* **ماژول functools**
* **آشنایی عمیق با function**

**لامبدا :**

در هنگام تعریف این توابع که به توابع ناشناس معروف هستند ، از کلمه کلیدیlambda استفاده میکنیم به جای def.

لامبدا روشی سریع برای ایجاد تابع است. تابع لامبدا باید به ساده ترین شکل ممکن و به دور از هرگونه پیچیدگی نوشته شوند.

عبارت lambda هر تعدادی که بخواهید میتواند آرگومان داشته باشد اما فقط یک manipulation خواهد داشت. در قسمتmanipulation میایم برای argsهامون یه بلایی سرش میاریم.

lambda (arguments): manipulation(arguments)

مثال :

add = lambda x, y: x + y

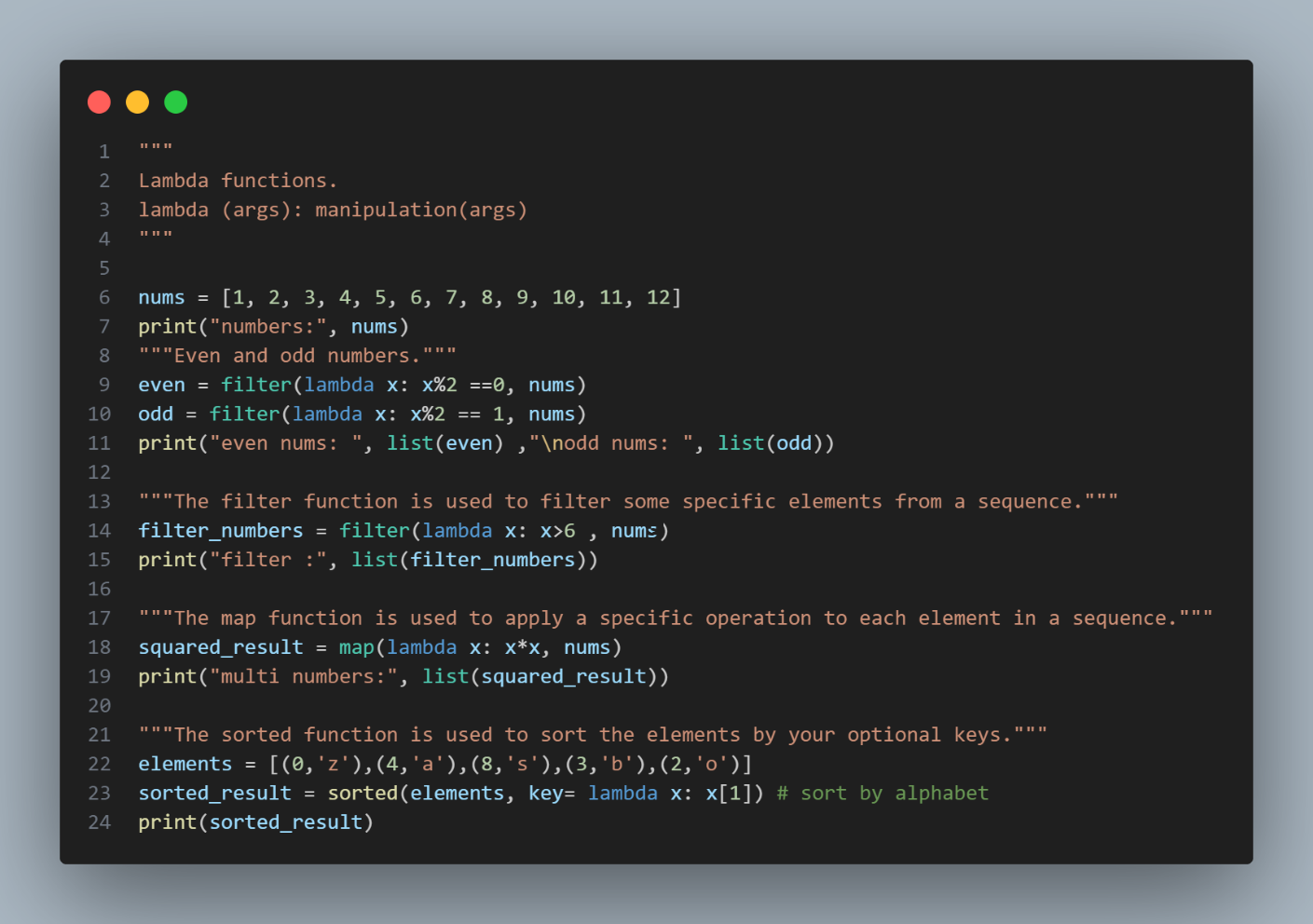
print( add(4, 6) )

در مثال بالا یک عبارت لامبدا ساختیم که دو مقدار x و y رو میگیره و اونها رو با هم جمع میکنه. و در آخر لامبدا رو داخل یک متغیر ذخیره میکنیم تا بتونیم بعدا ازش استفاده کنیم. در خط آخر هم لامبدا رو صدا زدیم و دو عدد رو بهش فرستادیم. اگه کد بالا رو اجرا کنید مقدار ۱۰ رو به شما برمیگردونه. **دقت کنید که لامبدا به صورت اتوماتیک مقدار نهایی رو return میکنه.**

یکی از مهمترین موارد استفاده رایج برای lambda، برنامه نویسی فانکشنال است چون پایتون از این پارادایم(سبک) پشتیبانی میکند.

همچنین مثلا وقتی میخواهید یک تابعی را به یک تابع دیگه ای به عنوان پارامتر پاس بدهید، لامبدا میتونه مفید باشه.( لطف ایجاد یک تابع یک بارمصرف.)

در محیط production، **هرگز نباید توابع lambda پیچیده را بنویسید**، زیرا رمزگشایی برای کد نویسان که کد شما را حفظ می کنند بسیار دشوار است. اگر خودتان را در حال ساختن عبارات پیچیده تک خطی کشف کردید، بهتر است با def یک فانکشن کامل بنویسید.



**متد islice از ماژول itertools**

متود های بیشتری هم داره این ماژول که میتونید بخونید راجبش ولی ما الان فقط این رو بررسی میکنیم.

اما این متود برای ما چیکار میکنه ؟

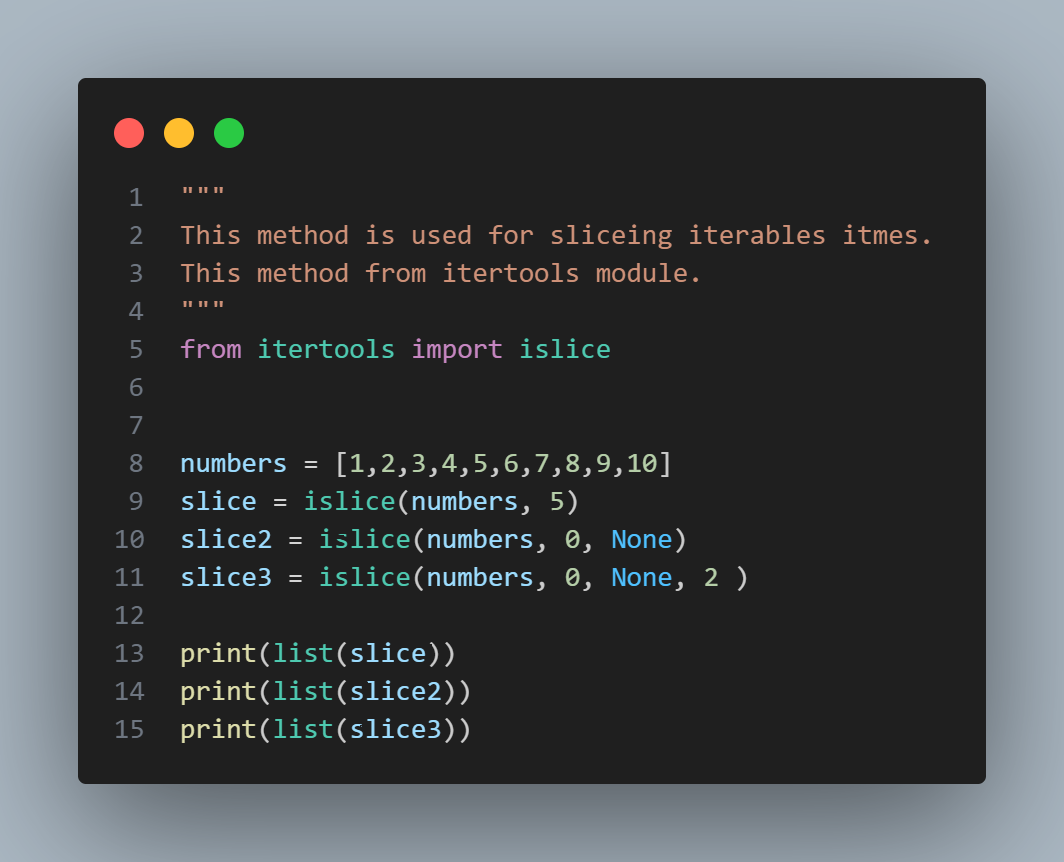
یک ایتربل میگیره و تیکه تیکش میکنه و اون قسمتی که میخای رو برات میاره.

اگه یک عدد بهش بدی ، از اول شروع میکنه تا اونجایی که عددو دادی و بعدش وای میسته.

اگه دوتا عدد بهش بدی، از از عدد اولت شروع میکنه و میاد تا عدد دومی که بهش دادی.

اگه سه تا عدد بهش بدی، مثل بالاییه ولی عدد سوم گام هارو مشخص میکنه.

مثال :



**: Generators**

بعضی وقتا مجبوریم که با داده هایی بزرگی کار کنیم که حافظه دستگاه مارا تحت تاثیر قرار میده،و 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 توابع ژنراتور ها و yield به کمک ما می آیند.

این توابع به ما یک آبجکت ایتراتور تنبل برمیگردانند.(lazy iterator)

اما این تنبل یعنی اینکه مثل بقیه آبجکت های ایتراتور(لیست و تاپل و....) میشه forروشون زد ولی بر خلاف اینها تکرار کننده های تنبل محتویات خود را در حافظه ذخیره نمی کنند.

مثلا شما اگه بخواهید یک فایل بزرگ را بخوانید که تعداد ستون های آن را بشمارید با استفاده از توابع عادی در پایتون، ممکنه به خطای مموری اررور برسید(تازه اگه سیستمون خفن باشه.....)

def csv\_reader(file\_name):

    file = open(file\_name)

    result = file.read().split("\n")

    return result

متود اوپن برای ما یک آبجکت تنبل برمیگردونه ولی توی خط بعدی ما میایم اون رو روی حافظه مینویسیم و کلا همه چی خراب میشه.

که اگه اینو ران کنید به این ارور میرسید:

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

پس باید از روش جایگزینی استفاده کنید(yeild)

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

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 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.

برای خلاصه کردن دستورات هم مثل list comprehension ، فقط جای [ ] از () استفاده میکنیم.