

جزوه کامل دوره‌ی پایتون آکادمی سبزلرن

مدرس: رضا دولتی

راه‌های ارتباطی:

اینستاگرام: [@rezadolati01](https://www.instagram.com/rezadolati01)

گردآوری: نعمان ریگی

راه‌های ارتباطی:

nomaan07.dev@gmail.com

تلگرام: [@nomaan07dev](https://t.me/nomaan07dev)

فهرست فصل‌ها

۳	فصل صفرم: مقدمات برای مبتدی‌ها
۱۵	فصل اول: نصب و پیاده سازی
۱۶	فصل دوم: مبانی و دستور نحو
۲۸	فصل سوم: انواع داده (سطح یک)
۷۳	فصل چهارم: دستورات کنترلی (تصمیم)
۷۹	فصل پنجم: دستورات کنترلی (تکرار)
۱۰۰	فصل ششم: تابع
۱۵۵	فصل هفتم: انواع داده (سطح یک)
۱۸۲	فصل هشتم: مازول‌ها و بسته‌ها
۱۹۹	فصل نهم: فایل‌ها، ورودی و خروجی
۲۳۲	فصل دهم: کلاس
۳۵۰	فصل یازدهم: مدیریت خط
۳۶۷	فصل دوازدهم: مباحث تکمیلی
۳۸۲	فصل سیزدهم: مباحث پیشرفته [همزمانی]

✖ درس ۱ تا درس ۴ ✖

درس ۵: زبان برنامه نویسی سطح بالا، پایین و میانی

زبان برنامه نویسی:

- سطح بالا: به زبان انسان نزدیکتر، سرعت کمتر، امنیت بالاتر. مانند: Python, JS, Ruby
- سطح میانی: شامل ویژگی هایی از هردو زبان. مانند: Java, C
- سطح پایین: به زبان ماشین نزدیکتر، امنیت پایین، سرعت بالاتر. مانند: باینری (01010001)

درس ۶: زبان های همه منظوره و خاص منظوره

زبان های همه منظوره (General Purpose Language): در هر سطح و موضوعی کاربرد دارند. مانند: Python – گیم

– وب

زبان های خاص منظوره (Domain Specific Language): فقط مختص یک موضوع است.

درس ۷: زبان های مفسری و کامپایلری

مفسری (interpreter): بررسی خط به خط و مشخص کردن خطاهای (سرعت کمتر). مانند: Python

کامپایلری (compiler): تولید فایل و اجرای دلخواه فایل و بررسی کلی (سرعت بالاتر). مانند: C#

درس ۸: زبان های استاتیک، داینامیک، قوی و ضعیف

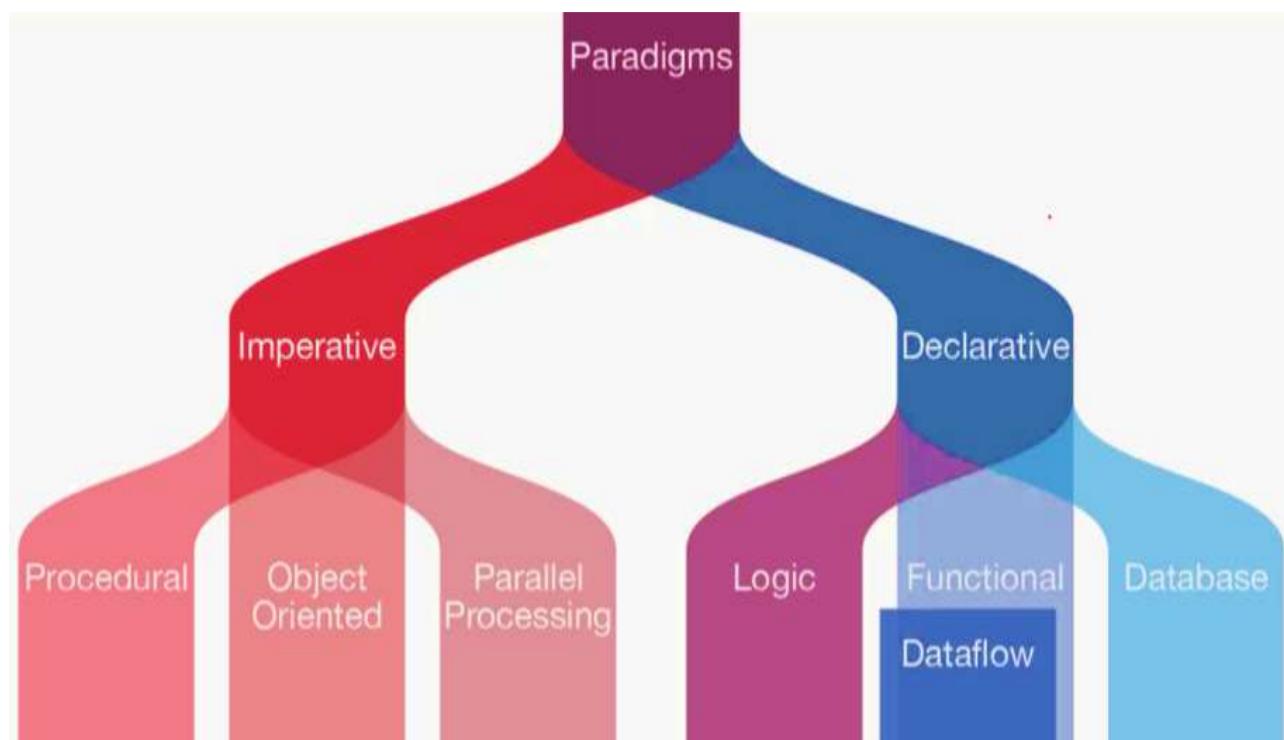
زبان برنامه نویسی استاتیک: باید نوع متغیر مشخص شود و خودش قادر به این کار نیست. مانند: C – C++

زبان برنامه نویسی داینامیک: خودش نوع متغیر را مشخص می کند. مانند: Python

زبان های تایپ قوی: حساسیت روی نوع داده ها و کار کردن با این انواع داده ها حساس است. مانند: Python

زبان های تایپ ضعیف: حساسیتی روی نوع داده وجود ندارد و کارهایی را پیش فرض انجام می دهد. مانند: C

درس ۹: پارادایم برنامه نویسی (الگوهای برنامه نویسی)



پارادایم برنامه نویسی دستوری(**Imperative**): مرحله به مرحله قدم‌های رسیدن به آن باید مشخص گردد. مراحل مهم است.

- برنامه نویسی پروسه‌ای(**Procedural**): قدم به قدم کارها مشخص است. مانند: C - C++ - Java
- برنامه نویسی شی‌گرا(**Object Oriented**): دارای کلاس Object، متود و ... خصوصیت: بیشتر تمرکز روی داده‌ها (شبیه سازی از دنیای واقعی). مانند: Python - Java - C++ - C#
- رایانش موازی(**Parallel**): اجرای همزمان یک برنامه روی چند پردازنده برای سرعت بیشتر و تبدیل تک task (task) های بزرگ به تک‌های کوچک.

پارادایم برنامه نویسی اعلانی یا اخباری(**Declarative**): خواسته را بیان می‌کند و نتیجه حاصل می‌شود. مراحل مهم نیست.

- منطقی(**Logical**): استوار بر منطق ریاضی (پازل - مجموعه).
- تابعی(**Functional**): تقسیم برنامه به توابع کوچک.
- پایگاه داده(**Database**): تمام تمرکز بر روی داده‌هاست.

* زبان‌ها نباید براساس پارادایم تقسیم شوند، زیرا اکثر زبان‌ها از تعداد زیادی پارادایم پشتیبانی می‌کنند.

درس ۱۰: پایتون(Python) و ویژگی‌های آن

ویژگی‌های پایتون:

- قدرتمندترین زبان
- آسان‌ترین زبان برای یادگیری
- زبانی همه منظوره (GPL)
- متن باز (Open source) است
- دارای کتابخانه‌های متنوع و زیاد برای بیشتر زمینه‌ها
- از نظر تایپ یک زبان داینامیک و تایپ قوی
- یک زبان سطح بالا.
- محدود به یک پارادایم خاص نیست
- یک زبان مفسری
- یک زبان برنامه نویسی قابل حمل (در هر سیستم عاملی قابل اجرا می‌باشد و وابستگی به سیستم عامل ندارد.)
- یک زبان توسعه پذیر
- case sensitive ($A \neq a$)
- یک زبان تعاملی

✖ درس ۱۱: داستان پایتون (تاریخچه) ✖

درس ۱۲: کاربردهای پایتون



درس ۱۳: نسخه‌های پایتون

A.B.C : 10.2.3

نحوه‌ی نمایش یک نسخه:

A: تغییرات بزرگ و اساسی B: تغییرات کوچک مهم هر نسخه C: رفع باگ‌ها و اشکالات

* از نسخه‌های end life استفاده نشود زیرا دیگر پشتیبانی نمی‌شوند.

درس ۱۴ و ۱۵: الگوریتم چیست؟

* به تعریف گام به گام یک عملیات برای حل یک مسئله با تقدم و تأخیر مشخص (اولویت مراحل) که متناهی باشد و از یک نقطه شروع و در نقطه‌ی دیگر به پایان برسد و ممکن است چندین بار تکرار شود، الگوریتم گفته می‌شود.

* حل کردن مسئله مهم است اما چگونه حل کردن مهم‌تر است.

* الگوریتم در کنار برنامه نویسی یادگرفته شود، کسی که الگوریتم یاد دارد، برنامه نویس بهتری است.

خصوصیت‌های الگوریتم: * ورودی * خروجی * قطعیت * محدودیت

نکات مهم در نوشتن الگوریتم:

* مراحل به ترتیب و پشت سرهم نوشته شود.

* قدم‌های ضروری باید در الگوریتم نوشته شود.

* از بیان جزئیات بیهوده صرف نظر و مراحل ساده و در عین حال کامل نوشته شود.

* هر الگوریتم فقط یک نقطه شروع دارد ولی ممکن است چند نقطه پایان داشته باشد.

* الگوریتم باید جامع باشد، به گونه‌ای که در حالت‌های خاص نتیجه‌ی مناسبی دهد.

$\xrightarrow{\quad}$ $\times \div -$

* اولویت عملگرهای ریاضی در نظر گرفته شود.

$() * / -$

* ترجیحاً از نمادهای موجود در صفحه کلید استفاده شود.

بخش‌های مهم در نوشتن الگوریتم:

* دستورات ورودی

* دستورات خروجی

* دستورات محاسباتی

* دستورات شرطی

* دستورات تکرار

دستورات ورودی و خروجی



مثال: الگوریتم دریافت یک عدد از کاربر و نمایش آن:

.1. شروع.

.2. n را دریافت کن.

.3. n را چاپ کن.

.4. پایان

دستورات محاسباتی:

* متغیر

* عملگرهای محاسباتی(ضرب، جمع، تقسیم، باقیمانده و ...)

* عملوندهای محاسباتی(اعداد، عبارات، متغیرها و ...)

مثال: الگوریتم دریافت یک عدد از کاربر و نمایش دو برابر آن:

.1 شروع.

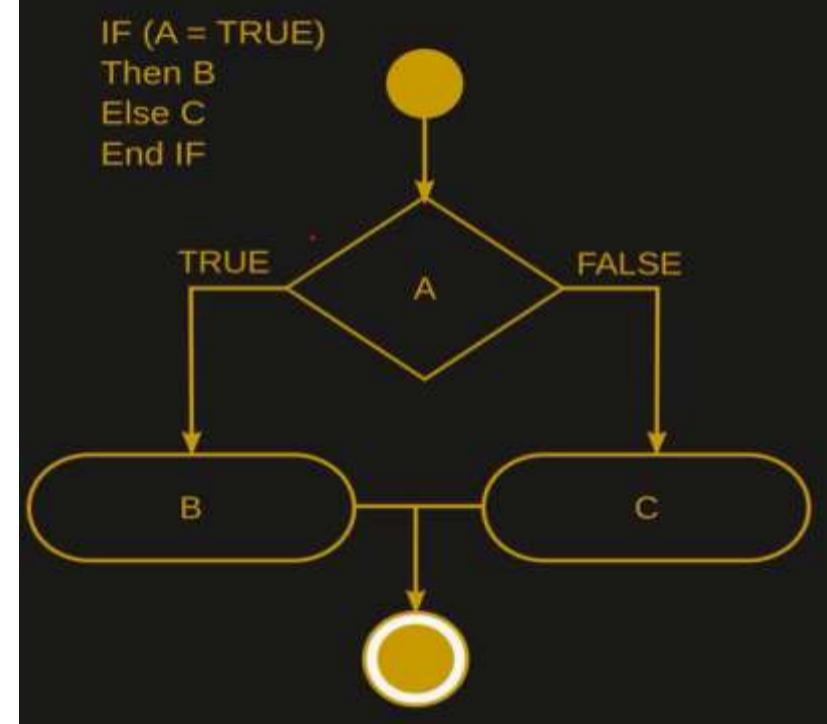
.2 n را دریافت کن.

X = n * 2 .3

.4 X را چاپ کن.

.5 پایان.

دستورات شرطی



مثال: الگوریتم تشخیص زوج بودن یک عدد:

.1 شروع.

.2 n را دریافت کن.

.3 اگر $n \% 2 == 0$ آنگاه "زوج" را چاپ کن در غیر اینصورت "فرد" را چاپ کن.

.4 پایان.

در دنیای برنامه نویسی:

/ : تقسیم •

* : ضرب •

% : باقی‌مانده •

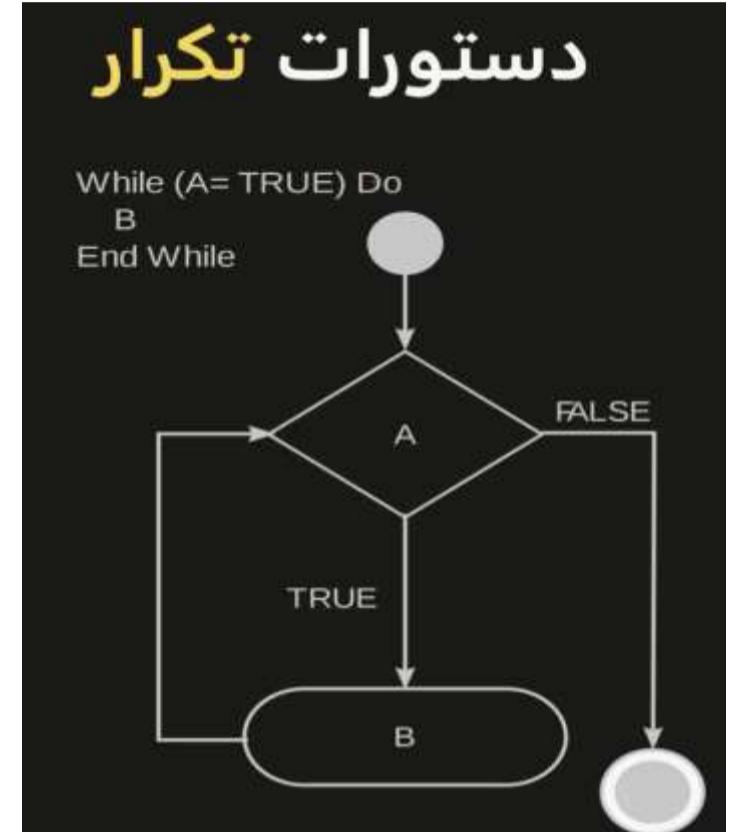
== : مقایسه •

دستورات تکرار

While (A= TRUE) Do

B

End While



مثال: الگوریتم چاپ تمام اعداد یک رقمی

.1 شروع

i = 1 .2

.3 کارهای زیر را تا زمانی که $10 < i$ تکرار کن:

• i را چاپ کن

• $i = i + 1$

.4 پایان

برخی از الگوریتم های مهم

برنامه نویسی پویا

- مبانی و اصول
- Edit Distance
- Edit Distance Alignment
- طولانی ترین زیردنباله مشترک
- طولانی ترین زیردنباله صعودی
- مینیمم پارتیشن
- راه هایی برای پوشش مسافت
- زمانبندی Assembly Line
- طولانی ترین مسیر در ماتریکس
- مسئله جمع زیرمجموعه ها
- استراتژی بهینه برای یک بازی
- مسئله کوله پشتی

گراف

- گراف وزنی
- درخت پوشان
- اتصال گراف
- اصطلاحات اساسی
- گراف های جهت دار
- Representation
- جستجوی عمق اول
- جستجوی سطح اول
- مرتب سازی توپولوژیکی
- الگوریتم فلوید وارshall
- الگوریتم Lee
- الگوریتم جانسون
- الگوریتم دایکسترا
- الگوریتم بلمن فورد
- الگوریتم FloodFill
- الگوریتم Prim و Kruskal

تحلیل الگوریتم: هدف از یادگیری الگوریتم این است که دنبال الگوریتم بهینه‌تر باشیم.

درس ۱۶: فلوچارت چیست؟

* برای درک بهتر الگوریتم از فلوچارت (روند نما) استفاده می‌کنیم.

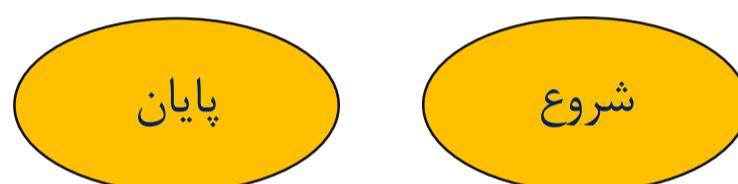
روش‌های بیان الگوریتم:

- بیان الگوریتم با جملات فارسی
- بیان الگوریتم با زبان ریاضی
- بیان الگوریتم توسط شکل‌ها

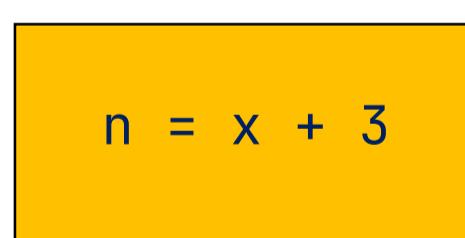
مهارت‌های مهم در رسم فلوچارت:

- ترمیناتور (Terminator)
- خط جریان (Flowline)
- فرایند (Process)
- تصمیم (Decision)
- داده (Data)

: (Terminator)

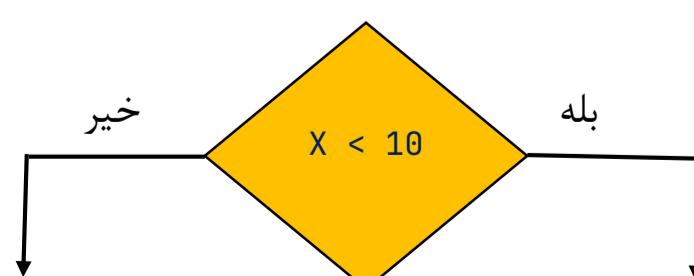


: (Flowline)



: (Process)

: (Decision) برای نمایش شرط‌ها



: (Data)

N را بخوان

N را چاپ کن

* برای رسم فلوچارت می‌توان از نرم افزارها و وبسایت‌های مختلف استفاده کرد.

مثال:

الگوریتمی که سه مقدار عددی را از ورودی خوانده، میانگین آنها را چاپ نماید.



الگوریتمی که دو مقدار را از ورودی خوانده و مقدار بزرگتر را چاپ کند.



درس ۱۸: مهارت‌های نرم در برنامه نویسی

مهارت‌های سخت(Hard skills): مهارت‌های فنی الزامی(کدنویسی و ...).

مهارت‌های نرم(Soft skills): در ظاهر الزامی نیستند، به خلق و خو مربوطند (مهارت‌های حل مسئله، کارکردن با تیم و ...).

مهارت‌های نرم ضروری:



و سایر مهارت‌ها همچون مهارت نوشتن قرارداد، مهارت مدیریت زمان، مهارت رزومه سازی، مهارت اعتماد به نفس، مهارت سرسختی، مهارت وقت شناسی و ...

پایان فصل صفرم

✖ درس ۱ تا درس ۹ ✖

درس ۱۰: آشنایی با محیط پایی چارم - بخش دوم - کلیدهای میانبر ۱

ctrl + r : جایگزینی:

shift + shift : دوبار زدن شیفت

ctrl + shift + ↑ ↓

ctrl + shift + ← →

alt + انتخاب کردن خط:

ctrl + alt + L

ctrl + w

ctrl + d

ctrl + /

سچ داخل فایل:

سچ در همه فایلها: (کم کاربرد)

جابجایی کدها بین خط

انتخاب کردن:

تایپ همزمان در چند خط:

مرتب کردن:

پیمایش به سمت بالا:

تکرار خط:

کامنت کردن:

درس ۱۱: آشنایی با محیط پایی چارم - بخش سوم - کلیدهای میانبر ۲

ctrl + space

پیشگویی:

alt + enter

کتابخانه: (رفع خطای کتابخانه با وارد کردن آن کتابخانه)

ctrl + alt + o

حذف کتابخانه ای که نیاز نداریم:

ctrl + shift + i

توضیحات یک تابع: (view سمت

ctrl + q

توضیحات کلی و کامل و عملکرد تابع:

ctrl + .

مخفي کردن چند بخش باهم:

ctrl + ~

دسترسی سریع به ظاهر و تمها:

ctrl + shift + a

سچ در گزینهها:

ctrl + b

نمایش توضیحات تابع در کدها:

shift + enter

رفتن به خط بعدی از هرجای کد:

ctrl + shift + z

رفتن به بعد: (پس از ctrl + z و بر عکس آن)

ctrl + ← →

پیمایش:

فصل دوم: مبانی و دستور نحو

درس ۱: مفهوم syntax و سطرها

به مجموعه قواعد و دستورات نگارشی یک زبان گفته می‌شود.

سطر فیزیکی که با عدد نمایش داده می‌شود.

آنچه مفسر تحت عنوان بک سطر قبول دارد (سطر منطقی).

* زمانی که دو متغیر را در یک خط می‌نویسیم گذاشتن سمی‌کالن (;) الزامی است.

```
f = 5; m = 7
```

* زمانی که می‌خواهیم یک خط کد منطقی را در یک خط فیزیکی دیگر قرار دهیم گذاشتن بک اسلش (\) الزامی است.

```
f = 'hello' \
    'guys'
```

درس ۲: کامنت (Comment) و داک استرینگ (Doc string)

* قوانین pep8 توصیه می‌کند که پس از کامنت یک خط فاصله گذاشته شود.

* از # برای کامنت کردن استفاده می‌کنیم، کامنت توسط مفسر نادیده گرفته می‌شود.

```
# this is a comment
```

* از Docstring در ابتدای یک تابع و یا کلاس برای توضیح دادن استفاده می‌شود، توسط مفسر نادیده گرفته می‌شود.

```
"""This is a docstring"""
''' This is a docstring '''
```

* در پایتون کتیشن (') و دابل کتیشن (") فرقی ندارند.

* کامنت برای خود برنامه نویس است.

* داک استرینگ برای کاربری است که استفاده می‌کند.

* داک استرینگ برای توابع و کلاس‌هاست.

درس ۳: تورفتگی و بلاک بندی

* برای تورفتگی باید از **Tab** فاصله استفاده شود و بهتر است از **Tab** استفاده نشود، مگر اینکه کدها از شخص دیگریست و او از تب استفاده کرده است.

```
x = 0
if x == 0:
    print('x')
```

درس ۴: آشنایی با ورودی و خروجی

* برای نمایش اطلاعات از دستور **print** استفاده می‌کنیم.

```
print('hi') # -> hi
```

* برای نوشتتن رشته‌ها از کتیشن (') یا دابل کتیشن (") استفاده می‌کنیم و گذاشتن آن الزامی است.

```
s = 's'
x = "s"
```

* برای جدا سازی در **print** از دستور **sep** استفاده می‌کنیم.

```
print('ali', 'reza', sep='*') # -> ali*reza
```

* برای مشخص کردن پایان دستور در **print** از **end** استفاده می‌کنیم.

```
print('ali', 'reza', end='*') # -> ali reza*
```

* برای گرفتن ورودی از کاربر از دستور **input** استفاده می‌کنیم.

```
s = input('x: ')
print(s)
```

```
x: 5
5
```

* برای رفتن به خط بعدی در **print** از "\n" استفاده می‌کنیم.

```
print('This is an example', '\n', 'for print')
print(20 * '-')
print("This is an example\nfor print")
```

```
This is an example
for print
-----
This is an example
for print
```

درس ۵: متغیر (variable)

str : کلمه‌ها، متن و ...

False, True : bool

int : اعداد صحیح

- * در پایتون نیاز به مشخص کردن نوع متغیر نیست.
- * متغیرها در واقع مکان‌هایی از حافظه هستند که نوعی داده خاص را در خود ذخیره می‌کنند.
- * برای مشخص کردن آدرس داده در حافظه از دستور id استفاده می‌کنیم.

```
a = 5  
print(id(a)) # -> 140705652073384
```

- * برای پاک کردن یک متغیر از دستور del استفاده می‌کنیم.

```
a = 5  
del a  
print(a)
```

```
NameError: name 'a' is not defined
```

درس ۶: شناسه (identifier)

- * به اسمی که برای متغیر و اشیای دیگر تعریف می‌کنیم شناسه identifier گفته می‌شود.
- * در مثال زیر a و MyClass شناسه هستند.

```
a = 5  
  
class MyClass:  
    pass
```

- * شناسه‌ی یک متغیر فقط می‌تواند با یکی از حروف انگلیسی (a-z, A-Z) و یا با underscore _ شروع شود.

```
_a = 4 ✓      #a = 3 ✗  
ab2 = 6 ✓      2a = 5 ✗
```

- * پایتون case sensitive است پس a با A فرق دارد.

- * برای طول یک شناسه هیچ محدودیتی وجود ندارد.

- * در شناسه از کاراکترهای خاص نمی‌شود استفاده کرد (فاصله، \$، !، /، # و ...)

* بهتر است برای شروع شناسه از `(_)` استفاده نکنیم زیرا جز شناسه‌های خصوصی مازول و کلاس می‌باشد.

`_Ali = (_` شناسه خصوصی مازول: شروع با `(_`)

`__Ali = (__` شناسه خصوصی کلاس: شروع با `(__`)

* در پایتون از شناسه‌های از پیش تعریف شده پایتون نمی‌شود استفاده کرد.

```
def = 5
```

SyntaxError: invalid syntax

* بهتر است از `l` (ال کوچک) و `I` (آی بزرگ) استفاده نشود زیرا در بعضی IDE‌ها مثل همانند.

* بهتر است کلاس‌ها **پاسکال کیس** (Pascal Case) نام‌گذاری شوند. (اول هر کلمه حرف بزرگ).

```
class MyClass:  
    pass
```

* برای توابع، متدها و متغیرها بهتر است از **حروف کوچک انگلیسی** استفاده شود و با `(_)` کلمات مختلف جدا شوند.

```
binary_search = 4
```

```
def func():  
    ...
```

درس ۷: کلمات کلیدی

* از کلمات کلیدی (کلماتی که مختص پایتون است) نباید به عنوان شناسه استفاده کرد.

```
if = 4
```

SyntaxError: invalid syntax

* برای نمایش کلمات کلیدی می‌توان از دستور `help` استفاده کرد.

```
help('keywords')
```

False	class	from	or
None	continue	global	pass
True	def	if	raise
and	del	import	return
as	elif	in	try
assert	else	is	while
async	except	lambda	with
await	finally	nonlocal	yield
break	for	not	

* برای استفاده از کلمات کلیدی می‌شود آن‌ها را تغییر داد ولی بهتر است این کار را نکنیم و از کلمات دیگر استفاده کنیم.

```
def_ = 5  
Def = 5
```

* برای اینکه مشخص کنیم کلمه کلیدی هست یا نه ابتدا کتابخانه `keyword` را فرا می‌خوانیم و سپس از `iskeyword` استفاده می‌کنیم.

```
import keyword  
  
print(keyword.iskeyword('def')) # -> True  
print(keyword.iskeyword('string')) # -> False
```

درس ۸: عملگرهای حسابی

ضرب: *	تفریق: -	جمع: +	توان: **	باقی‌مانده: %
تقسیم: /	تقسیم بدون قسمت اعشار: //			

درس ۹: عملگرهای مقایسه ای

* از این عملگرها بیشتر در شرط‌ها، حلقه‌ها و ... استفاده می‌شود.
* نتیجه این عملگرها یا `True` می‌باشد و یا `False`.

عملگر `==`: مقایسه انجام می‌دهد که آیا دو مقدار دو متغیر برابرند یا خیر. این عملگر با مساوی (`=`) تفاوت دارد.

```
x = 5  
print(x == 4) # -> False  
print(x == 5) # -> True
```

عملگر `!=` (نامساوی): برعکس `==` می‌باشد.

```
x = 5  
print(x != 4) # -> True  
print(x != 5) # -> False
```

عملگر `<>` (کوچکتر بزرگتر):

```
x = 5  
print(3 > 4) # -> True  
print(3 < 5) # -> False
```

عملگر \geq (بزرگ‌تر مساوی) :

```
x = 5
print(3 >= 4) # -> False
print(6 >= 5) # -> True
print(5 >= 5) # -> True
```

عملگر \leq (کوچک‌تر مساوی) :

```
x = 5
print(3 <= 4) # -> True
print(6 <= 5) # -> False
print(5 <= 5) # -> True
```

درس ۱۰: عملگرهای انتساب

عملگر $=$ (مساوی) : نسبت دادن دو عملوند چپ و راست به هم.

```
x = 5
y = x
z = 2 + 6
print(x) # -> 5
print(y) # -> 5
print(z) # -> 8
```

عملگر $+=$: ابتدا جمع و سپس مقداردهی به عملوند سمت چپ.

* عملگرهای $+=, -=, *=, /=, %=, //=, **=, \=, \&=, ^=, >>=, <<=$ نیز به همین صورت عمل می‌کنند.

```
x = 6
x += 2 # -> x = x + 2 -> x = 6 + 2 -> 8
```

درس ۱۱: عملگرهای منطقی

* عمده کاربرد این عملگرها در عبارت‌های شرطی و حلقه‌هاست و برای بررسی همزمان دو یا چند شرط با یکدیگر استفاده می‌شوند و نتیجه این عملگرها یا **True** است یا **False**.

عملگر **and** (و) : باید همه شرط‌ها درست باشند تا نتیجه **True** باشد.

```
x = 4
y = 5

print(x < 5 and y > 4) # -> True
print(x < 5 and y < 5) # -> False
```

عملگر **or** (یا): فقط یکی از شرط‌ها درست باشد کافیست تا نتیجه **True** باشد.

```
x = 4  
y = 5  
  
print(x < 5 or y < 5) # -> True  
print(x > 5 or y < 5) # -> False
```

عملگر **not** (برعکس): باید شرط غلط باشد تا نتیجه **True** باشد.

```
x = 4  
y = 5  
  
print(not (x < 3)) # -> True  
print(not (x > 3)) # -> False
```

درس ۱۲: عملگرهای عضویت

* برای تست عضویت استفاده می‌شوند (آیا چیزی داخل چیزی می‌باشد؟) و نتیجه این عملگرها یا **True** است یا **False**.

عملگر **in**: اگر عضو باشد نتیجه **True** است.

عملگر **not in**: اگر عضو نباشد نتیجه **True** است.

```
d = [1, 2, 3, 9, 'm']  
print(3 in d) # -> True  
print('z' not in d) # -> False
```

درس ۱۳: عملگرهای هویت

* برای بررسی بودن یکی بودن دو تا شی استفاده می‌شود و با برابری متفاوت است، مثلاً دو تا هزار تومانی برابرند ولی یکسان نیستند.

عملگر **is**: اگر یکی باشند نتیجه **True** است.

عملگر **not is**: اگر یکی نباشد نتیجه **True** است.

```
x = 5  
y = 5  
z = 7  
print(x is y) # -> True  
print(x is not z) # -> True
```

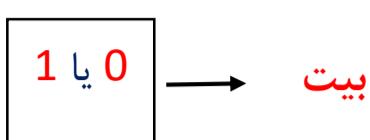
* لیست‌ها با مقادیر یکسان در خانه‌های متفاوتی از حافظه ذخیره می‌شوند، پس یکی نیستند.

```
x = [1, 2, 4]  
y = [1, 2, 4]  
print(x is y) # -> False
```

درس ۱۴: عملگرهای بیتی

* این عملگرها روی بیت‌ها کار می‌کنند.

* به مکان‌هایی در حافظه که فقط مقدار ۰ و ۱ می‌گیرند را بیت می‌گویند.



* هشت بیت = ۱ بایت

1	0	1	1	1	0	0	1
---	---	---	---	---	---	---	---

* برای تبدیل عدد باینری به دسیمال از روش زیر استفاده می‌کنیم.

$$\begin{array}{cccccccc} 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ \downarrow * & \downarrow * \\ 2^7 & + 2^6 & + 2^5 & + 2^4 & + 2^3 & + 2^2 & + 2^1 & + 2^0 \\ 0 & + 64 & + 0 & + 0 & + 8 & + 0 & + 0 & + 1 = 73 \end{array}$$

* در پایتون می‌توانیم عدد را به صورت باینری بنویسیم، به این صورت که قبل از آن باید **0b** گذاشت.

```
x = 0b000111
```

* برای این‌که مشخص کنیم باینری یک عدد چیست از تابع **bin** استفاده می‌کنیم.

```
print(bin(5)) # -> 0b101
```

* عملگر **&** (and): اگر هردو بیت یک باشند، یک بیت تعیین می‌کند.

```
x = 3      # 0b011
y = 4      # 0b100
# x & y = 0b000
print(x & y) # -> 0

x = 11    # 0b1011
y = 13    # 0b1101
# x & y = 0b1001
print(x & y) # -> 9
```

* عملگر **|** (or): اگر حداقل یک بیت یک باشد، یک بیت تعیین می‌کند.

```
x = 50    # 0b110010
y = 29    # 0b011101
# x | y = 0b111111
print(x | y) # -> 63
```

* عملگر **^** (**xor**): در صورتی که فقط یکی از بیت‌ها یک باشد، یک بیت تعیین می‌کند.

```
x = 50 # 0b110010
y = 29 # 0b011101
# x ^ y = 0b101111
print(x ^ y) # -> 47
```

* عملگر **<<** (**shift to left**): به سمت چپ هل (شیفت) می‌دهد.

```
x = 5      # 0b00000101
y = 4
# x << y = 0b01010000
print(x << y) # -> 80
```

* راه حل ذهنی: $x * (2 ** y)$

```
print(5 << 4) #-> 5 * (2 ** 4) = 80
```

* عملگر **>>** (**shift to right**): به سمت راست هل (شیفت) می‌دهد.

```
x = 15     # 0b1111
y = 2
# x >> y = 0b0011
print(x >> y) # -> 3
```

* راه حل ذهنی: $x // (2 ** y)$

```
print(15 >> 2) #-> 15 // (2 ** 2) = 3
```

* عملگر **~** (**Not**): همه‌ی بیت‌ها را برعکس می‌کند و مشخص می‌کند عدد به دست آمده منفی چه عددی است.

روش‌های رایج منفی ساز:

- نمایش مقدار علامت: اولین عدد از چپ را به عنوان علامت در نظر می‌گیرد (1 منفی است).
- روش متمم یک: همه‌ی صفرها به یک و همه‌ی یک‌ها به صفر تبدیل می‌شوند.
- روش متمم دو: همه‌ی اعداد را برعکس می‌کند ولی از سمت راست تا اولین 1 نباید تغییر داد و بقیه را برعکس کرد، منفی عدد به دست آمده پاسخ است.

```
print(~9) -> -10
# 9      ->  00001001
#       ->  11110110
#       - برعکس
#       - 00001010 -> 10 -> -10  روش متمم دو
```

* راه حل ذهنی: $-x - 1$

```
print(~9) # -> - 9 - 1 = -10
```

درس ۱۵: عملگر والروس

* این عملگر هم زمان مقدار دهی می کند و در شرطها، لیست و حلقه ها کاربرد دارد.

```
print(x := 5) # -> 5
```

```
print(x = 5)
```

```
TypeError: 'x' is an invalid keyword argument for print()
```

درس ۱۶: اولویت عملگرها

```
() -> f() -> [index:index] -> [] -> ** -> ~ -> + -> (*) (مثبت و منفی) -> (*, /, %)  
-> + -> (<<, >>) -> & -> ^ -> | -> (in, not in, is, is not, <, <=, >, >=, ==, !=) -> not -> and -> or -> lambda
```

درس ۱۷: عبارات و دستورات

* عبارت (**expression**): حداقل یک مقدار برای تولید می کند و باید ارزیابی شود و یک نتیجه می دهد، معمولاً چیزی تولید می کند (شناسه، لیترال (مقداری که به متغیر می دهیم)، عملگرهای حسابی و ...).

```
x + 7 * 9
```

* دستور (**statement**): از عبارت ها و کلمات کلیدی تشکیل می شود و یک وظیفه خاص را انجام می دهد و نیاز نیست ارزیابی شوند و می توانند چیزی تولید نکنند، دو نوع دستور ساده و مرکب داریم.

* دستورات ساده در یک سطر منطقی پیاده سازی می شوند.

```
print()
```

* دستورات مرکب معمولاً با یک کلمه کلیدی شروع می شوند و با **دو نقطه (:**) تمام می شوند و در ادامه بدن نوشته می شود و طبق قواعد فرورفتگی بدن باید جلوتر باشد.

* دستور یک بخشی مرکب:

```
def f():  
    pass
```

* دستور چند بخشی مرکب:

```
if x > 4:  
    print('bigger')  
elif x == 4:  
    print("even")
```

```
else:  
    print("smaller")
```

* عبارت را می‌شود به متغیر اختصاص داد و پرینت کرد.

```
x = 2 + 3  
print(x) # -> 5
```

* دستور را نمی‌شود به متغیر داد و نمی‌شود پرینت کرد.

```
x = if  
print(if)
```

```
TypeError: 'x' is an invalid keyword argument for print()
```

درس ۱۸: مروری بر مفاهیم شی گرایی، متدها و صفت‌ها

شی: یک نمونه (`instance`) از یک کلاس است.

متد (method) در شی گرایی: انجام دادن یک وظیفه‌ی خاص را برعهده دارد.

کلاس (class) در شی گرایی: برنامه‌ی ما از یک سری واحدها تشکیل می‌شود به نام کلاس که هر کدام یک از این کلاس‌ها تعدادی متده را که وظیفه‌ی خاصی انجام می‌دهند.

* همان طرح و نقشه است.

وراثت در شی گرایی: ایجاد یک کلاس جدید با ویژگی‌های کلاس قبلی و دارا بودن قابلیت‌های جدید را وراثت در شی گرایی گویند.

* از روی یک کلاس می‌شود به تعداد دلخواه شی ساخت.

* هر یک از این شی‌ها می‌تواند خصوصیت داشته باشد.

* متدها رفتار را نشان می‌دهند.

* خصوصیت‌ها توصیف می‌کنند.

✖ درس ۱۹: قوانین نگارشی (pep8) و ذن پایتون (pep20) ✖

درس ۲۰: تکالیف مبحث مبانی و دستور نحو

۱. دو متغیر X و U را تعریف کنید و به آن‌ها مقادیری اختصاص دهید. سپس، مقادیر X و U را بدون استفاده از متغیر سوم عوض کنید.
۲. برنامه‌ای بنویسید تا مساحت یک مستطیل را با استفاده از متغیرها محاسبه کند. طول و عرض را در متغیرها ذخیره کنید. سپس مساحت را محاسبه و چاپ کنید.
۳. برنامه‌ای ایجاد کنید که دما را از سانتی‌گراد به فارنهایت تبدیل کند. دما را بر حسب سانتی‌گراد در یک متغیر ذخیره کنید، تبدیل را انجام دهید و نتیجه را چاپ کنید و
۴. مساحت و محیط دایره را محاسبه کنید. شعاع را در یک متغیر ذخیره کنید و سپس مساحت و محیط را محاسبه و چاپ کنید.
۵. برنامه‌ای بسازید که مجموع، تفاوت، حاصلضرب و توان دو عدد را محاسبه کند. دو متغیر X و U را تعریف کنید، آن‌ها را در متغیرها ذخیره کنید، محاسبات را انجام دهید و نتایج را چاپ کنید.
۶. برنامه‌ای بسازید که میانگین سه عدد را محاسبه کند. سه متغیر X ، U و Z را تعریف کنید، آن‌ها را متغیرها ذخیره کنید، میانگین را محاسبه کنید و نتیجه را چاپ کنید.
۷. یک برنامه پایتون بنویسید تا مساحت یک مثلث را با توجه به قاعده و ارتفاع آن محاسبه کند. قاعده و ارتفاع را در دو متغیر ذخیره کنید، مساحت را محاسبه کنید و نتیجه را چاپ کنید.
۸. یک متغیر ایجاد کنید و سن خود را در آن بنویسید. سالی که در آن ۱۰۰ ساله خواهید شد را محاسبه کنید و نتیجه را چاپ کنید.

۹. نام متغیرها را در قطعه کد داده شده را مطابق با قراردادهای نام‌گذاری pep 8 تغییر دهید:

```
num1 = 42  
Num2 = 13  
ReSuLt = num1 Num2  
print(ReSuLt)
```

فصل سوم: انواع داده (سطح یک)

درس ۱: انواع داده در پایتون

- * با تابع `type` می‌توانیم نوع داده را مشخص کنیم.
- * اعداد در پایتون سه نوع اعداد صحيح (`integer`), اعداد اعشاري (`float`) و اعداد مختلط (`complex`) هستند.

```
print(type(5)) # -> <class 'int'>
print(type(5.1)) # -> <class 'float'>
print(type(3 + 2j)) # -> <class 'complex'>
```

- * اگر تابع داخل کلاس باشد به آن `متده` می‌گویند و معمولاً متده به صورت زیر نمایش داده می‌شود.

```
x.method()
```

- * نوع داده `دیکشنری` (`dictionary`) همانند دیکشنری عمل می‌کند و معنی و مفهوم را می‌رساند.

```
x = {'Ali': 'ugly', 'x': 5}
print(type(x)) # -> <class 'dict'>
```

- * نوع داده `بولین` (`boolean`) که یا `False` و یا `True` است.

```
x = True
y = False
print(type(x)) # -> <class 'bool'>
print(type(y)) # -> <class 'bool'>
```

- * نوع داده `مجموعه` (`set`) که در آن تکراری ها حذف می‌شوند.

```
x = {1, 2, 2, 3, 4, 4}
print(x) # -> {1, 2, 3, 4}
print(type(x)) # -> <class 'set'>
```

- * توالی‌ها عبارتند از `رشته` (`string`), `لیست` (`list`) و `تاپل` (`tuple`), که در توالی‌ها می‌توانیم بین عناصر بچرخیم به این معنی که می‌توانیم به عنوان مثال به عنصر بعدی و یا عناصر اول، دوم و ... دست یابیم.

```
s = 'Reza'
lst = [1, 2, 3, 4]
t = (1, 2, 3, 4)
print(type(s)) # -> <class 'str'>
print(type(lst)) # -> <class 'list'>
print(type(t)) # -> <class 'tuple'>
```

* در پایتون شمارش از صفر شروع می‌شود.

```
s = 'Reza Dolati'  
#     012345678910  
print(s[2: 6]) # -> za D
```

* **لیست** قابل تغییر است.

```
lst = [1, 2, 3, 4]  
lst[2] = 10  
print(lst) # -> [1, 2, 10, 4]
```

* تاپل قابل تغییر نیست.

```
t = (1, 2, 3, 4)  
t[2] = 10
```

```
TypeError: 'tuple' object does not support item assignment
```

* نوع داده **None** زمانی استفاده می‌شود که نمی‌دانیم چه نوع مقداری به متغیر بدهیم و همچنین زمانی که در توابع نمی‌خواهیم چیزی برگردانده شود از **None** استفاده می‌کنیم.

```
x = None  
print(type(x)) # -> <class 'NoneType'>
```

درس ۲: اعداد در پایتون

* اعداد در پایتون سه نوع اعداد صحیح (integer)، اعداد اعشاری (float) و اعداد مختلط (complex No.) هستند.

```
print(type(5)) # -> <class 'int'>
print(type(5.1)) # -> <class 'float'>
print(type(3 + 2j)) # -> <class 'complex'>
```

* اعداد مختلط معمولاً در پایتون استفاده‌ای ندارند.

* اعداد مبنای مختلفی دارند و اعدادی که در زندگی روزمره استفاده می‌کنیم بر مبنای ۱۰ هستند.

* حافظه‌ی کامپیوتر بر اساس مبنای ۲ (باینری = ۰، ۱) کار می‌کند.

* در کامپیوتر و وب مبناهای ۲، ۸ و ۱۶ بسیار معروف و پر استفاده هستند.

* در مبنای ۱۰ (۰ تا ۹) ده عدد استفاده می‌شود.

* در مبنای ۲ (۰ و ۱) دو عدد استفاده می‌شود.

* در مبنای ۸ (۰ تا ۷) هشت عدد استفاده می‌شود.

* در مبنای ۱۶ (۰ تا ۹ + a تا f) شانزده عدد استفاده می‌شود.

* در مبنای ۱۶ بعد از ۹ از حروف انگلیسی استفاده می‌کنیم.

```
a = 10      b = 11      c = 12      d = 13
e = 14      f = 15
```

* اعدادی که در کامپیوتر می‌نویسیم به صورت پیش‌فرض بر مبنای ۱۰ می‌باشند.

* برای این‌که بتوانیم عدد را بر مبنای ۲ (باینری) بنویسیم، باید ابتدای عدد از ۰b استفاده کنیم.

```
x = 0b1011
print(x) # -> 11
```

* برای این‌که بتوانیم عدد را بر مبنای ۸ بنویسیم، باید ابتدای عدد از ۰o استفاده کنیم.

```
x = 0o7235
print(x) # -> 3741
```

* برای این‌که بتوانیم عدد را بر مبنای ۱۶ بنویسیم، باید ابتدای عدد از ۰x استفاده کنیم.

```
x = 0xf6e2a
print(x) # -> 1011242
```

* از تابع bin برای به‌دست آوردن مبنای ۲ یک عدد استفاده می‌کنیم.

```
x = 10
print(bin(x)) # -> 0b1010
```

* از تابع `oct` برای به دست آوردن مبنای 8 یک عدد استفاده می کنیم.

```
x = 10  
print(oct(x)) # -> 0o12
```

* از تابع `hex` برای به دست آوردن مبنای 16 یک عدد استفاده می کنیم.

```
x = 10  
print(hex(x)) # -> 0xa
```

* برای تبدیل عدد صحیح به عدد اعشاری از تابع `float` استفاده می کنیم.

```
x = 10  
x = float(x)  
print(x) # -> 10.0
```

* برای تبدیل عدد اعشاری به عدد صحیح از تابع `int` استفاده می کنیم.

```
x = 1.9  
x = int(x)  
print(x) # -> 1
```

* برای تبدیل به عدد کامپلکس از تابع `complex` استفاده می کنیم.

```
x = 1  
x = complex(x)  
print(x) # -> (1+0j)
```

* اگر رشته فقط از عدد تشکیل شده باشد می توان آن را به عدد صحیح تبدیل کرد.

```
x = '234'  
print(int(x)) # -> 234
```

* در حافظه کامپیوتر اعداد به صورت باینری ذخیره می شوند و ممکن است گاهی در محاسبات اعشاری دقت کافی حاصل نشود.

```
print(0.6 - 0.4) # -> 0.1999999999999996
```

* این مشکل در حلقه ها و شرطها خودش را نشان می دهد.

```
print(0.6 - 0.4 == 0.2) # -> False
```

* برای حل این مشکل می توانیم از `decimal` که آن را باید از ماژول `decimal` وارد کنیم، استفاده کنیم.

```
from decimal import Decimal  
  
print(Decimal('0.6') - Decimal('0.4')) # -> 0.2
```

* برای این که بتوانیم اعداد اعشاری را به صورت کسری نمایش بدهیم، می‌توانیم از **Fraction** که آن را باید از ماژول **fractions** وارد کنیم، استفاده کنیم.

```
from fractions import Fraction  
  
print(Fraction(1.5)) # -> 3/2
```

* برای جداسازی ارقام بزرگ می‌توانیم از **_** (underscore) استفاده کنیم.

```
x = 1_000_000_000
```

* عدد اعشاری که بسیار بزرگ باشد به نحوه‌ی دیگری نمایش داده می‌شود.

```
print(1e+7) # -> 10000000.0  
# 1e+7 = 1 * 10 ** 7  
print(1e-7) # -> 0.0000001  
# 1e-7 = 1 * 10 ** -7
```

* به معنی بی‌نهایت و **-inf** به معنی منفی بی‌نهایت می‌باشد.

* اعداد صحیح محدودیت ندارند اما اعداد اعشاری محدودیت دارند.

```
print(2e+400) # -> inf  
  
f = float('inf')  
print(f) # -> inf
```

* برای نمایش بخش حقیقی اعداد **real** از **complex** و برای نمایش بخش مجھول از **imag** استفاده می‌کنیم.

```
c = 1 + 3j  
print(c.real) # -> 1.0  
print(c.imag) # -> 3.0
```

* برای نمایش مزدوج اعداد **complex** از متده **conjugate** استفاده می‌کنیم.

```
c = 1 + 3j  
print(c.conjugate()) # -> (1-3j)  
  
p = 2 - 3j  
print(p.conjugate()) # -> (2+3j)
```

* برای گرد کردن اعداد اعشاری از تابع **round** استفاده می‌کنیم، در ورودی اول عدد را ارسال می‌کنیم و در ورودی دوم مشخص می‌کنیم تا چند رقم اعشار باید رند شود.

```
c = 3.2564543565  
print(round(c, 2)) # -> 3.26
```

* برای به دست آوردن قدر مطلق یک عدد از تابع **abs** استفاده می‌کنیم.

```
print(abs(3.25)) # -> 3.25  
print(abs(-4.32)) # -> 4.32
```

* برای به توان رساندن از تابع **pow** استفاده می‌کنیم، در ورودی اول عدد را ارسال می‌کنیم و در ورودی دوم توان را می‌نویسیم.

```
print(pow(3, 2)) # -> 9
```

درس ۳: حل مثال برای مبحث اعداد

۱. برنامه‌ای بنویسید که وزن را بر اساس کیلوگرم دریافت کند و بر اساس گرم نمایش دهد.

```
kg = int(input('Enter kg: '))
g = kg * 100
print('g:', g)
```

```
Enter kg: 25
g: 2500
```

* در دستور `print` با `sep` می‌توانیم مشخص کنیم چه چیزی در میان عناصر قرار گیرد.

```
print(25, 'Ali', 'car', sep='/') # -> 25/Ali/car
```

۲. برنامه‌ای بنویسید که مساحت یک مثلث را حساب کند.

```
height = int(input('height: '))
base = int(input('base: '))
area = 1 / 2 * height * base
print('The area of triangle is:', area)
```

```
height: 4
base: 6
The area of triangle is: 12.0
```

۳. طراحی یک ماشین حساب ساده که جمع، تفریق، ضرب و تقسیم را انجام می‌دهد.

```
x = int(input('x: '))
y = int(input('y: '))
print(x, '+', y, '=', x + y)
print(x, '-', y, '=', x - y)
print(x, '*', y, '=', x * y)
print(x, '/', y, '=', x / y)
```

```
x: 3
y: 8
3 + 8 = 11
3 - 8 = -5
3 * 8 = 24
3 / 8 = 0.375
```

۴. یک عدد را از کاربر گرفته و ارقامش را جدا کنیم و چاپ کنیم (به فرض این که می‌دانیم عدد چند رقمی است).

* هر عددی باقیمانده‌اش به ۱۰ می‌شود رقم آخرش.

```
print(45903 % 10) # -> 3
```

* هر عددی را به صورت تقسیم بدون اعشار به ۱۰ تقسیم کنیم، رقم آخرش حذف می‌شود.

```
print(45903 // 10) # -> 4590
```

```
x = int(input('x: '))
y = x % 10
print(y)
x = x // 10
y = x % 10
print(y)
x = x // 10
y = x % 10
print(y)
```

```
x: 347
7
4
3
```

۵. یک زاویه را برحسب درجه بگیریم و آن را تبدیل به رادیان کنیم.

```
degree = int(input('degree: '))
radians = (degree * 3.14) / 180
print('radians:', radians)
```

```
degree: 90
radians: 1.57
```

درس ۴ و ۵: تکالیف مبحث اعداد و حل آنها

۱. برنامه‌ای بنویسید که شعاع دایره را از کاربر گرفته و محیط و مساحت دایره را حساب کند.

```
p = 3.14
r = float(input('r: '))
print(environment:', round(2 * p * r, 3))
print(area:', round(p * (r ** 2), 2))
```

۲. برنامه‌ای بنویسید که یک عدد از کاربر گرفته و مربع (توان دوم) و مکعب (توان سوم) آن را چاپ کند.

```
x = int(input('x: '))
print('x ^ 2 =', x ** 2)
print('x ^ 3 =', x ** 3)
```

۳. برنامه‌ای بنویسید که دو عدد از کاربر گرفته، اولی را به توان دومی برساند و نتیجه را چاپ کند.

```
x = int(input('x: '))
y = int(input('y: '))
print('x ^ y =', x ** y)
```

۴. برنامه‌ای بنویسید که سه عدد گرفته و میانگین آن‌ها را محاسبه و چاپ کند.

```
x = int(input('x: '))
y = int(input('y: '))
z = int(input('z: '))
ave = (x + y + z) / 3
print('ave:', round(ave, 2))
```

درس ۶: رشته‌ها در پایتون بخش (اول)

* برای بررسی این که مقدار متغیر از نوع خاصی هست یا خیر از تابع `isinstance` استفاده می‌کنیم، ورودی اول مقدار و ورودی دوم نوع می‌باشد.

```
x = 2
print(isinstance(x, int)) # -> True
print(isinstance(x, str)) # -> False
```

- * برای نمایش متن، جملات و تک کاراکترها از رشته‌ها استفاده می‌کنیم.
- * در پایتون رشته‌ها را داخل کتیشن (') و یا دابل کتیشن ("") قرار می‌دهیم.
- * رشته می‌تواند حروف انگلیسی، حروف و نمادهای سایر زبان‌ها، عدد، کاراکترهای خاص مانند #، & و ... باشد.
- * اعداد هم بهتر است به صورت رشته دخیره شوند مگر در مواردی که نیاز به انجام محاسبات است.
- * اگر یک رشته را با کتیشن آغاز کنیم باید با کتیشن ببندیم و همچنین اگر با دابل کتیشن آغاز کنیم باید با دابل کتیشن ببندیم.

```
s = 'reza'
z = "ali"
```

* زمانی که رشته را داخل کتیشن یا دابل کتیشن قرار دادیم، می‌توانیم داخل آن نیز از کتیشن یا دابل کتیشن استفاده کنیم اما باید دقیق کنیم که اگر رشته داخل کتیشن باشد باید داخل رشته از دابل کتیشن استفاده کنیم و اگر رشته داخل دابل کتیشن باشد، باید داخل رشته از کتیشن استفاده کنیم.

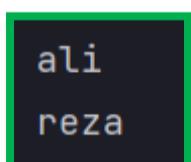
```
s = 'ali, "reza", ahmad'
z = "ali, 'reza', ahmad"
```

* اگر بخواهیم داخل رشته از کتیشن یا دابل کتیشن همانند کتیشن یا دابل کتیشنی که رشته در آن قرار دارد استفاده کنیم باید قبل از آن از **بک اسلش (\)** استفاده کنیم، **بک اسلش (\)** وظیفه‌ی لغو یک دستور را دارد.

```
s = "ali, \"reza\", ahmad"
z = 'ali, \'reza\', ahmad'
```

* در رشته‌ها با استفاده از \n می‌توانیم به خط بعدی برویم.

```
print('ali\nreza')
```



```
ali
reza
```

* برای لغو تاثیر **بک اسلش (\)** از یک بک اسلش دیگر قبلش استفاده می‌کنیم.

```
print('c:\Downloads\\new') # -> c:\Downloads\new
```

* در رشته‌ها با استفاده از \t می‌شود یک tab فاصله انداخت.

```
print('ali\treza') # -> ali      reza
```

* گاهی تعداد دستوراتی که باید لغو کنیم بسیار زیاد است، در این حالت اگر از **r** قبل از رشته استفاده کنیم، همه دستورات داخل رشته لغو می‌شوند.

```
print(r'c:\Download\new\telegram') # -> c:\Download\new\telegram
```

* در صورت طولانی شدن رشته می‌توانیم رشته را ببندیم، یک بک اسلش بگذاریم و ادامه رشته را در خط بعد بنویسیم.

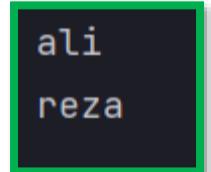
```
print('ali ' \
      'reza') # -> ali reza
```

* بهتر است بک اسلش را نگذاریم.

```
print('ali ' \
      'reza') # -> ali reza
```

* استفاده از سه کتیشن (**'''**) به جای یک کیتیشن، رشته را همانگونه که می‌نویسیم نمایش می‌دهد.

```
print('''ali
reza''')
```



* رشته‌ها را با **+** به یکدیگر متصل می‌کنیم.

```
print('ali' + 'reza') # -> alireza
```

* رشته‌ها را با ***** تکرار می‌کنیم.

```
print('ali-' * 3) # -> ali-ali-ali-
```

* اگر هردو رشته باشند می‌توان بدون کاما (**,**) و یا جمع آن‌ها را کنار هم قرار داد.

```
print('ali''reza') # -> alireza
```

درس ۷: رشته‌ها در پایتون بخش (دوم)

- * رشته‌ها چون توالی‌اند به راحتی می‌توان شماره گذاری کرد و عناصر مختلف آن را نمایش داد.
- * در شماره گذاری رشته‌ها، شمارش از **۰** آغاز می‌شود.
- * اگر بخواهیم از آخر شمارش کنیم، شمارش از **۱** آغاز می‌شود.

```
x = 'reza'  
print(x[3]) # -> a  
print(x[-3]) # -> r  
print(x[2]) # -> z  
print(x[0]) # -> r  
print(x[-2]) # -> z  
print(x[-4]) # -> r  
print(x[1]) # -> e
```

* در رشته‌ها برای مشخص کردن یک بخش از تکه بندی (**slicing**) استفاده می‌کنیم.

* در **slicing** اگر ابتدا خالی باشد یعنی از اول و اگر انتهای خالی باشد یعنی کل رشته را مشخص کند.

```
s = 'ali reza'  
#      ۰۱۲۳۴۵۶۷  
#      ...-۲-۱  
print(s[2: 3])          # -> i  
print(s[2:])            # -> i reza  
print(s[: 3])           # -> ali  
print(s[:])              # -> ali reza  
print(s[: 2] + s[2:])   # -> ali reza
```

* در **slicing** می‌توانیم گام مشخص کنیم.

```
s = 'ali reza'  
#      ۰۱۲۳۴۵۶۷  
print(s[2: 7: 2]) # -> irz
```

* رشته‌ها برخلاف لیست‌ها قابل تغییر نیستند و نمی‌توانیم یک کاراکتر را عوض کنیم.

```
s = 'ali reza'  
s[2] = 'q'
```

```
TypeError: 'str' object does not support item assignment
```

* از تابع **len** می‌توانیم برای مشخص کردن سایز رشته استفاده کنیم.

```
s = 'ali'  
print(len(s)) # -> ۳  
print(s[:len(s)]) # -> ali
```

* اگر در slicing گام را 1- بدهیم رشته را برعکس نمایش می‌دهد.

```
s = 'ali reza'  
print(s[::-1]) # -> azer ila
```

درس ۸: متدهای مهم رشته

* تابع `len` تعداد کاراکترهای یک رشته را مشخص می‌کند (فاصله نیز کاراکتر محسوب می‌شود.

```
s = 'ali #6'
print(len(s)) # -> 7
```

* متدهای `upper` حروف کوچک انگلیسی را به بزرگ تبدیل می‌کند.

```
s = 'aLi reza'
print(s.upper()) # -> ALI REZA
print(s) # -> aLi reza
```

* در این حالت تغییرات روی رشته اعمال نشده اند، برای اعمال تغییرات به صورت زیر می‌توانیم عمل کنیم.

```
s = 'aLi reza'
s = s.upper()
print(s) # -> ALI REZA
```

* متدهای `lower` حروف بزرگ انگلیسی را به کوچک تبدیل می‌کند.

```
s = 'ALI REZA'
s = s.lower()
print(s) # -> ali reza
```

* از متدهای `count` برای شمارش یک یا چند کاراکتر در یک رشته استفاده می‌کنیم.

```
s = 'ali reza'
print(s.count('a')) # -> 2
print(s.count('za')) # -> 1
```

* از متدهای `endswith` برای بررسی اینکه یک رشته با کاراکتر یا رشته‌ی خاصی به پایان رسیده است یا خیر، استفاده می‌کنیم.

```
s = 'ali ahmadi'
print(s.endswith('i')) # -> True
print(s.endswith('adi')) # -> True
print(s.endswith('h')) # -> False
```

* از متدهای `startswith` برای بررسی اینکه یک رشته با کاراکتر یا رشته‌ی خاصی آغاز شده است یا خیر، استفاده می‌کنیم.

```
s = 'ali ahmadi'
print(s.startswith('a')) # -> True
print(s.startswith('ali ')) # -> True
print(s.startswith('d')) # -> False
```

* از متدهای `find` برای پیدا کردن موقعیت یک کاراکتر در رشته استفاده می‌کنیم، اگر چند عدد از آن کاراکتر در رشته باشد این متدهای موقعیت اولین کاراکتر را از سمت چپ مشخص می‌کند.

```
s = 'ali mohammadi'
print(s.find('a')) # -> 0
print(s.find('m')) # -> 4
```

* از متدهای **rfind** برای پیدا کردن موقعیت یک کاراکتر در رشته از سمت راست استفاده می‌کنیم.

```
s = 'ali mohammadi'  
print(s.rfind('a')) # -> 10  
print(s.rfind('m')) # -> 9
```

* از متدهای **isalnum** برای بررسی این‌که یک رشته تماماً از عدد و حرف تشکیل شده باشد استفاده می‌کنیم.

```
s = 'alimohammadi'  
n = 'a li'  
m = 'a435li' بهار  
c = '$kb@v'  
print(s.isalnum()) # -> True  
print(n.isalnum()) # -> False  
print(m.isalnum()) # -> True  
print(c.isalnum()) # -> False
```

* از متدهای **isnumeric** و **isdigit** رای بررسی این‌که یک رشته فقط از عدد تشکیل شده باشد استفاده می‌کنیم.

```
x = '3535467'  
y = '45 445'  
z = '35h'  
print(x.isnumeric()) # -> True  
print(x.isdigit()) # -> True  
print(y.isdigit()) # -> False  
print(z.isdigit()) # -> False
```

* از متدهای **join** برای قرار دادن یک رشته در بین عناصر یک لیست و یا تاپل در صورتی‌که رشته باشند، استفاده می‌کنیم.

```
s = '#'  
lst = ['A', '5', 'me']  
print(s.join(lst)) # -> A#5#me
```

* از متدهای **split** برای جداسازی عناصر یک رشته براساس یک رشته‌ی خاص و قرار دادن آن در یک لیست استفاده می‌کنیم.

```
s = 'reza-ali-435'  
print(s.split('-')) # -> ['reza', 'ali', '435']
```

* از متدهای **replace** برای جایگزینی یک کاراکتر با یک کاراکتر استفاده می‌کنیم.

```
s = 'ali reza'  
print(s.replace('a', 'b')) # -> bli rezb
```

* از متدهای **strip** برای حذف یک کاراکتر از ابتدا و انتهای رشته استفاده می‌کنیم، مقدار پیش‌فرض آن فاصله است.

```
s = '      ali reza      '  
x = '**ali reza**'  
  
print(s.strip()) # -> ali reza  
print(x.strip('*')) # -> ali reza
```

* از متدهای حذف یک کاراکتر از ابتدای رشته و از `rstrip` برای حذف یک کاراکتر از انتهای رشته استفاده می‌کنیم، مقدار پیشفرض آن فاصله است.

```
s = '**ali reza**'

print(s.lstrip('*')) # -> ali reza**
print(s.rstrip('*')) # -> **ali reza
```

* با استفاده از متدهای `capitalize` حرف اول رشته بزرگ و بقیه کوچک می‌شوند.

```
s = 'aLI Reza'

print(s.capitalize()) # -> Ali reza
```

درس ۹: کاراکترهای ویژه و آشنایی با یونیکد

* هر کاراکتر با یک کد خاص در حافظه ذخیر می‌شود، که این کد می‌تواند اسکی (ASCII) و یا یونیکد (unicode) باشد.

* اسکی (ASCII) بسیار محدود است ام یونیکد (unicode) بسیار گسترده است.

* با استفاده از تابع `ord` می‌توانیم اسکی یا یونیکد یک کاراکتر را مشاهده کنیم

```
print(ord('%')) # -> 37
```

* با استفاده از تابع `chr` می‌توانیم مشخص کنیم که یک کد به چه کاراکتری اختصاص دارد.

```
print(chr(457)) # -> ئ
```

* اگر یونیکد بر مبنای ۱۰ (دسمال) باشد، با قرار دادن \۰ می‌توانیم مشخص کنیم که به چه کاراکتری اختصاص دارد.

```
print('\u0489') # -> ئ
print('\u2DBB') # -> ئى
```

* اگر یونیکد بر مبنای ۸ باشد، با قرار دادن \۰ می‌توانیم مشخص کنیم که به چه کاراکتری اختصاص دارد.

```
print('\052') # -> *
```

* اگر یونیکد بر مبنای ۱۶ باشد، با قرار دادن \x۰۵۲ می‌توانیم مشخص کنیم که به چه کاراکتری اختصاص دارد.

```
print('\x2a') # -> *
```

* از یونیکد \n می‌توان برای رفتن به خط بعد استفاده کرد، از یونیکد سایر موارد نیز می‌توان استفاده کرد، اما بهتر است از دستورات اختصاصی آن‌ها استفاده شود.

```
print('reza\u000Aali')
```



* اگر از \b در رشته استفاده کنیم، کاراکتر قبل از خودش را پاک می‌کند.

```
print('ali\breza') # -> alreza
```

* اگر از \r در رشته استفاده کنیم، همه‌ی کاراکترهای قبل از خودش را پاک می‌کند.

```
print('ali\rreza') # -> reza
```

درس ۱۰: فرمتدهی رشته با روش سنتی (روش اول)

* این روش نسبت به روش‌های دیگر قدیمی‌تر است، در آن از **%** استفاده می‌کنیم و مقادیر را باید به ترتیب ارسال کنیم.

```
x = 2
y = 3.6
print('x is: %i\ny is: %i\nz is: %i' % (x, y, 5))
```

```
x is: 2
y is: 3
z is: 5
```

* در این روش پس از **%** از این موارد استفاده می‌کنیم. (**type** اجباری می‌باشد و بقیه موارد اختیاری‌اند).

```
% [(key)] [flag] [W] [.p] type
```

* **type** نوع رشته را مشخص می‌کند و استفاده از آن اجباری می‌باشد.

* برای اعداد صحیح از **i** و یا **d** استفاده می‌کنیم و قسمت اعشار حذف می‌شود.

```
print('%i' % 2)    # -> 2
print('%i' % 4.554)  # -> 4
print('%d' % 6.34)   # -> 6
```

* اگر از **C** استفاده کنیم، کاراکتر یک یونیکد را نمایش می‌دهد و اگر یونیکد نباشد خود کاراکتر را نمایش می‌دهد.

```
print('%c' % 674)    # -> ظ
print('%c' % 'y')     # -> y
```

* از **s** برای رشته‌ها استفاده می‌کنیم.

```
print('%s' % 'reza')  # -> reza
```

* از **0** برای نمایش مبنای **8** یک عدد استفاده می‌کنیم.

```
print('%o' % 356)    # -> 544
```

* از **X** برای نمایش مبنای **16** یک عدد استفاده می‌کنیم، اگر از **X** استفاده کنیم، نتیجه نیز با حرف بزرگ نمایش داده می‌شود.

```
print('%x' % 436547567)  # -> 1a052fef
print('%X' % 436547567)  # -> 1A052FEF
```

* از **e** برای نمایش نماد علمی یک عدد استفاده می‌کنیم، اگر از **E** استفاده کنیم، نتیجه نیز با حرف بزرگ نمایش داده می‌شود.

```
print('%e' % 436547567)  # -> 4.365476e+08
print('%E' % 436547567)  # -> 4.365476E+08
```

* از **f** برای نمایش اعداد اعشار استفاده می‌کنیم، تا 6 رقم اعشار نمایش داده می‌شود.

```
print('%f' % 2.6)    # -> 2.600000
```

* از **r** برای نمایش رشته به کار می‌رود و بخلاف **s** رشته را همانگونه که هست و با کتیشن نمایش می‌دهد.

```
print('%r' % 'reza') # -> 'reza'
```

* از **[p.]** برای کنترل تعداد ارقام اعشار استفاده می‌کنیم، به این صورت که پس از **.** مشخص می‌کنیم تا چند رقم اعشار را نمایش دهد.

```
print('.%.2f' % 3.54355653545) # -> 3.54
```

* از **[w]** برای مشخص کردن طول میدان استفاده می‌کنیم، اگر طول میدان را بزرگ‌تر از تعداد کاراکترها مشخص کنیم، به جای طول میدان اضافی، قبل از کاراکترها فاصله گذاشته می‌شود.

```
print('%9.2f' % 3.5435) # -> 3.54
```

* از **+ در [flag]** نمایش علامت‌ها استفاده می‌کنیم.

```
print(' %+9.2f' % 3.5435) # -> +3.54
```

* از **- در [flag]** برای انتقال فضاهای خالی به سمت راست استفاده می‌کنیم.

```
print(' %-9.2f' % 3.5435) # -> 3.54
```

* از **0 در [flag]** برای پر کردن فضاهای خالی با **0** استفاده می‌کنیم، نمی‌توانیم **0** و **-** را در کنارهم به کار ببریم.

```
print(' %09.2f' % 3.5435) # -> 000003.54
```

* از **[(key)]** برای نمایش مقادیر کلیدها در دیکشنری استفاده می‌شود.

```
d = {'a': 2, 'b': 7}
print('%(a)f\n%(b)i' % d) # -> 4.365476E+08
```

```
2.000000
7
```

* اگر به جای موارد اختیاری از **ستاره (*)** استفاده کنیم، می‌توانیم آن موارد را قبل از متغیر و به ترتیب تعیین کنیم.

```
x = 3.8487320342358
print('%.8.*f' % (5, x)) # -> 3.84873
```

درس ۱۱: فرمتدهی رشته با متدهای `format` (روش دوم)

* نسبت به روش قبلی جدیدتر و دارای قابلیت‌های بیشتری است، در آن از `{}` استفاده می‌کنیم و مقادیر را به ترتیب به آن ارسال می‌کنیم.

```
'{} {}'.format(values)
```

```
x = 5  
y = 6.4354354  
print('x: {}\\ny: {}'.format(x, y))
```

```
x: 5  
y: 6.4354354
```

* اگر بخواهیم مقادیر به ترتیب نمایش داده نشوند، می‌توانیم از مکان مقادیر در `{}` استفاده کنیم.

```
x = 5  
y = 6.4354354  
print('x: {1}\\ny: {2}\\nz: {0}'.format(x, y, 4))  
#
```

```
x: 6.4354354  
y: 4  
z: 5
```

* می‌توانیم از مقادیر چند بار نیز استفاده کنیم.

```
x = 5  
y = 6.4354354  
print('x: {0}\\ny: {0}\\nz: {0}'.format(x, y, 4))
```

```
x: 5  
y: 5  
z: 5
```

* در **دیکشنری‌ها** استفاده از **کلید-مقدار** برای فرمت دهی بسیار طولانی است.

```
d = {'x': 5, 'y': 8, 'z': 2.34}  
print('x: {}\\ny: {}\\nz: {}'.format(x=d['x'], y=d['y'], z=d['z']))
```

```
x: 5  
y: 8  
z: 2.34
```

* برای حل این مشکل می‌توانیم به اجای ارسال مقادیر از دو ستاره (**) قبل از دیکشنری در هنگام فرمت دهی استفاده کنیم.

```
d = {'x': 5, 'y': 8, 'z': 2.34}
print('x: {} \ny: {} \nz: {}'.format(**d))
```

```
x: 5
y: 8
z: 2.34
```

* در **توالی‌ها** (لیست، رشته و تاپل) از یک ستاره (*) قبل از توالی در هنگام فرمت دهی استفاده می‌کنیم.

```
print('x: {} - y: {} \n'.format(*'re'))
print('x: {} - y: {} \n'.format(*[1, 2]))
print('x: {} - y: {} \n'.format(*(4, 8)))
```

```
x: r - y: e

x: 1 - y: 2

x: 4 - y: 8
```

ساختار کلی فرمت دهی با متدهای **format** (این مبحث با روش سوم مشترک است).

```
'{[field-name][!conversion][:format-spec]}'
```

* در **[field-name]** اسم، کلید و یا مکان متغیر را قرار می‌دهیم.

```
print('x: {} - y: {} \n'.format(*(4, 8))) # -> x: 8 - y: 4
```

* در **[!conversion]** پس از ! نوع متغیر را مشخص می‌کنیم، از **s** برای نمایش رشته، **r** برای نمایش رشته با کتیشن و **a** برای نمایش یونیکد کاراکتر استفاده می‌کنیم.

```
print('x: {}!s'.format('reza')) # -> x: reza
print('x: {}!r'.format('reza')) # -> x: 'reza'
print('x: {}!a'.format('≤')) # -> x: '\u2264'
```

* در **[format-spec]** پس از : نوع فرمت را تعیین می‌کنیم.

```
print('x: {:.2f}'.format(1.265356)) # -> x: 1.27
```

ساختار کلی [:format-spec]: (همه‌ی این موارد اختیاری می‌باشند.)

```
'{:[fill]align}[sign][#][0][width][grouping-option][.precision][type}'
```

* در [type] نوع را مشخص می‌کنیم.

* برای اعداد صحیح از d استفاده می‌کنیم.

```
print('{:d}'.format(3)) # -> 3
```

* اگر از c استفاده کنیم، کاراکتر یک یونیکد را نمایش می‌دهد..

```
print('{:c}'.format(1195)) # -> ©
```

* از s برای رشته‌ها استفاده می‌کنیم.

```
r print('{:s}'.format('reza')) # -> reza
```

* از b برای نمایش مبنای 2 (باینری) یک عدد استفاده می‌کنیم.

```
print('{:b}'.format(234)) # -> 11101010
```

* از o برای نمایش مبنای 8 یک عدد استفاده می‌کنیم.

```
print('{:o}'.format(234)) # -> 352
```

* از X برای نمایش مبنای 16 یک عدد استفاده می‌کنیم، اگر از X استفاده کنیم، نتیجه نیز با حرف بزرگ نمایش داده می‌شود.

```
print('{:x}'.format(234)) # -> ea
print('{:X}'.format(234)) # -> EA
```

* از e برای نمایش نماد علمی یک عدد استفاده می‌کنیم، اگر از E استفاده کنیم، نتیجه نیز با حرف بزرگ نمایش داده می‌شود.

```
print('{:e}'.format(234)) # -> 2.340000e+02
print('{:E}'.format(234)) # -> 2.340000E+02
```

* از f برای نمایش اعداد اعشار استفاده می‌کنیم، تا 6 رقم اعشار نمایش داده می‌شود.

```
print('{:f}'.format(2.1)) # -> 2.100000
```

* از % برای نمایش عدد کسری به صورت درصد استفاده می‌کنیم.

```
print('{:%}'.format(2/3)) # -> 66.666667%
```

* در [.precision] و پس از . مشخص می‌کنیم تا چند رقم اعشار نمایش داده شود.

```
print('{:.2f}'.format(4.5646543)) # -> 4.56
```

* در [grouping-option] برای جدا سازی اعداد بزرگ از کاما (,) یا اندراسکور (_) استفاده می‌کنیم.

```
print('{:.2f}'.format(3825987)) # -> 3,825,987.00  
print('{:_2f}'.format(3825987)) # -> 3_825_987.00
```

* در [width] طول میدان را مشخص می‌کنیم، اگر طول میدان را بزرگ‌تر از تعداد کاراکترها مشخص کنیم، به جای طول میدان اضافی، قبل از کاراکترها فاصله گذاشته می‌شود.

```
print('{:15,.2f}'.format(4543)) # -> 4,543.00
```

* در [0] جاهای خالی را با 0 پر می‌کنیم.

```
print('{:015,.2f}'.format(4543)) # -> 0,000,004,543.00
```

* در [#] اگر قبل از b از # استفاده کنیم، قبل از عدد باینری 0b نمایش داده می‌شود.

```
print('{:#b}'.format(43)) # -> 0b101011
```

* در [#] اگر قبل از o از # استفاده کنیم، قبل از مبنای 8 عدد، 0o نمایش داده می‌شود.

```
print('{:#o}'.format(43)) # -> 0o53
```

* در [#] اگر قبل از x از # استفاده کنیم، قبل از مبنای 16 عدد، 0x نمایش داده می‌شود.

```
print('{:#x}'.format(453)) # -> 0x1c5
```

* در [sign] اگر از + استفاده کنیم، علامت عدد را نشان می‌دهد.

```
print('{:+}'.format(43)) # -> +43  
print('{:+}'.format(-43)) # -> -43
```

* در [sign] اگر از - استفاده کنیم، اگر عدد منفی باشد علامت آن را نمایش می‌دهد.

```
print('{:-}'.format(43)) # -> 43  
print('{:-}'.format(-43)) # -> -43
```

* در [sign] اگر از فاصله استفاده کنیم، اگر عدد مثبت باشد، فاصله را نمایش می‌دهد و اگر عدد منفی باشد علامت آن را نمایش می‌دهد.

```
print('{: }'.format(43)) # -> 43  
print('{: }'.format(-43)) # -> -43
```

- * در [[fill]align] در قسمت align با استفاده از < فضاهای خالی را در سمت راست و انتهای رشته قرار می‌دهیم.
- * در [[fill]align] در قسمت align با استفاده از > فضاهای خالی را در سمت چپ و ابتدای رشته قرار می‌دهیم.
- * در [[fill]align] در قسمت align با استفاده از ^ رشته را در میان فضاهای خالی قرار می‌دهیم.

```
print('{:<15}'.format(43)) # -> 43
print('{:>15}'.format(43)) # -> 43
print('{:^15}'.format(43)) # -> 43
```

- * در [[fill]align] در قسمت fill مشخص می‌کنیم که فضاهای خالی با چه چیزی پر شوند، حتماً باید مشخص کنیم که فضاهای خالی در کجا قرار گیرند.

```
print('{:+<15}'.format(43)) # -> 43+++++++++++++
print('{:+>15}'.format(43)) # -> +++++++43+++++
print('{:+^15}'.format(43)) # -> ++++++43++++++
```

- * می‌توانیم فرمت را در بیرون از رشته و در متدهای format نیز مشخص کنیم.

```
print('{:{fill}{align}{width}}'.format(43, fill="+", align='^', width=15)) # -
> ++++++43++++++
```

درس ۱۲: فرمتدهی رشته با متدهای **f-string** (روش سوم)

- * این روش جدیدترین روش است و بهتر است از این روش استفاده کنیم.
- * این روش همانند روش قبل است فقط بهجای استفاده از متدهای **format** از **F** قبل از رشته استفاده می‌کنیم.

```
x = 14
print(f'{43:+^15} ---- {x ** 2:+}') # -> ++++++43++++++ ---- +196
print(F'{43:+^15} ---- {x ** 2:+}') # -> ++++++43++++++ ---- +196
```

- * اگر داخل رشته از **{}** استفاده کنیم و بخواهیم **{}** نمایش داده شود باید آن را داخل یک **{}** دیگر قرار دهیم.

```
print(f'{{x}} is {4}') # -> {x} is 4
```

- * اگر بخواهیم مقدار را در **{}** نمایش دهد، باید متغیر را داخل **{}** قرار دهیم.

```
print(f'{{x}} is {{{4}}}') # -> {x} is {4}
```

- * از **توابع** (فصل-۶) نیز می‌توانیم در **f-string** استفاده کنیم.

```
x = 'reza'

def up_case(s):
    return s.upper()

print(f'x: {up_case(x)}') # -> x: REZA
```

- * از **متدهای مختلف** نیز می‌توانیم در **f-string** استفاده کنیم.

```
x = 'reza'
print(f'x: {x.replace("a", "-")}') # -> x: rez-
```

- * برای این‌که بتوانیم چند رشته را پشت سرهم بنویسیم و فرمت دهی کنیم می‌توانیم آن‌ها را در یک متغیر قرار دهیم.

```
name = 'ali'
age = '23'
msg = (
    f'name: {name}\n'
    f'age: {age}'
)
print(msg)
```

```
name: ali
age: 23
```

* برای استفاده از تاریخ از مژول **datetime** استفاده می‌کنیم.

* برای مشخص کردن زمان کنونی از متدهای **now** و **today** مژول **datetime** استفاده می‌کنیم.

```
from datetime import datetime

print(datetime.today()) # -> 2023-07-24 18:28:20.815442
print(datetime.now()) # -> 2023-07-24 18:28:20.815442
```

* می‌توانیم تاریخ را نیز فرمت دهی کنیم، **%Y** نشان دهنده سال، **%m** نشان دهنده ماه، **%d** نشان دهنده روز، **%H** نشان دهنده ساعت در فرمت ۲۴ ساعته، **%M** نشان دهنده دقیقه، **%S** نشان دهنده ثانیه می‌باشند.

```
import datetime

today = datetime.datetime.today()
print(f'{today:%Y/%m/%d-%H:%M:%S}') # -> 2023/07/24-18:38:26
```

درس ۱۳: حل مثال برای مبحث رشته

۱. فرض کنیم یک رشته داریم و یک جمله در آن ذخیره شده است، برنامه‌ای بنویسیم که یک کاراکتر از کاربر بگیرد و مشخص کند آن رشته چند بار تکرار شده است.

```
s = 'this is an example for a string.'
x = input('x: ')
print(s.count(x))
```

۲. برنامه‌ای بنویسیم که یک جمله از کاربر بگیرد و آخرین کلمه‌اش را نمایش دهد.

```
s = input('enter string: ')
s = s.rstrip()
x = s.rfind(' ')
print(s[x + 1:])
```

۳. برنامه‌ای بنویسیم که دو رشته از کاربر گرفته و مشخص کند که رشته‌ی دوم در رشته‌ی اول هست یا خیر.

```
s1 = input('enter string1: ')
s2 = input('enter string2: ')
print(s2 in s1)
```

۴. برنامه‌ای بنویسیم که یک رشته از کاربر گرفته و همهٔ فاصله‌ها را حذف کند.

```
s = input('enter string: ')
print(s.replace(' ', '').replace('\t', ''))
```

۵. برنامه‌ای بنویسیم که شماره تلفن را از کاربر گرفته و ۰ اول را پاک کند.

```
s = input('enter number: ')
print(s.lstrip('0'))
```

درس ۱۴ و ۱۵: تکالیف مبحث رشته و حل آنها

۱. یک رشته را از کاربر بگیرید و سپس تعداد جمله‌ها، تعداد کلمه‌ها، تعداد کل کاراکترها (مثلاً حروف، فاصله، نقطه، ویرگول و...) و تعداد حروف انگلیسی (مثل a, b, c و ...) را چاپ کنید.

```
s = input('Enter your text: ')
sentences = s.count('.') + s.count('?') + s.count('!') + s.count(';')
words = s.count(' ') + 1
characters = len(s)
letters = characters - (s.count('.') + s.count('?') + s.count('!')
+ s.count(';') + s.count(',',) + s.count(':')
+ s.count('-') + s.count(' '))

print('number of sentences:', sentences)
print('number of words:', words)
print('number of characters:', characters)
print('number of letters:', letters)
```

۲. یک کاراکتر از کاربر بگیرید و یونیکد آن را چاپ کنید.

```
ch = input('Enter your char: ')
print(ord(ch))
```

۳. یک شماره همراه را از کاربر بگیرید و بررسی کنید که آیا تمام ارقام آن عدد است یا خیر؟

```
phone = input('Enter your phone number: ')
print(phone.isdigit())
```

درس ۱۶: لیست‌ها در پایتون (بخش اول)

* در لیست‌ها از هر نوع داده‌ای می‌توان استفاده کرد.

```
daneshjoo = ['reza', 'dolati', 10, 15, 3, 20]
print(daneshjoo) # -> ['reza', 'dolati', 10, 15, 3, 20]
```

* بهتر است از یک نوع داده در لیست استفاده کنیم.

* برای ایجاد لیست می‌توانیم عناصر را داخل براکت [] قرار دهیم آن‌ها را کاما (,) از هم جدا کینم.

```
x = [1, 3, 5, 7, 9]
```

* لیست را با استفاده از تابع list نیز می‌توانیم ایجاد کنیم.

```
s = 'reza32e5465'
print(list(s)) # -> ['r', 'e', 'z', 'a', '3', '2', 'e', '5', '4', '6', '5']
```

* با استفاده از متده split نیز می‌توانیم عناصر یک رشته را براساس کاراکتری خاص جدا کنیم و در یک لیست قرار دهیم.

```
s = 'reza-ali-ahmad-b-v'
print(s.split('-')) # -> ['reza', 'ali', 'ahmad', 'b', 'v']
```

* لیست‌ها نیز همانند رشته‌ها ایندکس و مکان مشخص دارند و می‌توانیم از slicing استفاده کنیم.

```
s = [1, 2, 4, 'Ali']
#    0   1   2   3

print(s[0])      # -> [1]
print(s[:3])     # -> [1, 2, 4]
print(s[1:])     # -> [2, 4, 'Ali']
print(s[:])      # -> [1, 2, 4, 'Ali']
print(s[1:3])    # -> [2, 4]
print(s[::2])    # -> [1, 4]
```

* لیست‌ها نیز همانند رشته‌ها از قابلیت تکرار با * و جمع با + پشتیبانی می‌کنند.

```
s = [1, 5]
d = [2, 6]
print(s * 2)    # -> [1, 5, 1, 5]
print(s + d + ['a', 'b', 3]) # -> [1, 5, 2, 6, 'a', 'b', 3]
```

* اشیا در پایتون یا تغییر پذیرند mutable (مانند لیست‌ها، یا تغییر ناپذیر immutable) مانند رشته‌ها، اعداد و تاپل.

```
s = [1, 5]
s[0] = 14
print(s) # -> [14, 5]
```

* در لیست‌ها می‌توانیم از عملگرهای مقایسه‌ای استفاده کنیم.

```
s = [1, 5]
x = [2, 7]
n = [1, 5]
print(s == x) # -> False
print(s == n) # -> True
```

* در لیست‌ها می‌توانیم از عملگرهای عضویت استفاده کنیم.

```
s = [1, 5]
print(2 in s) # -> False
print(5 in s) # -> True
```

درس ۱۷: لیست‌ها در پایتون (بخش دوم)

* در لیست‌ها برخلاف رشته‌ها و اعداد، اگر در لیست دوم که آن را مساوی لیست اول قرار دادیم تغییر اعمال کنیم، این تغییرات در لیست اول نیز اعمال می‌شوند.

```
s = [1, 5]
z = s
z[0] = 0
print(s) # -> [0, 5]
```

* برای این‌که تغییرات اعمال نشوند می‌توانیم از کپی لیست با مساوی قرار دادن لیست دوم با تمام عناصر لیست اول استفاده کنیم.

```
s = [1, 5]
z = s[:]
z[0] = 0
print(s) # -> [1, 5]
```

* می‌توانیم با استفاده از متدهای copy لیست را کپی کنیم.

```
s = [1, 5]
z = s.copy()
z[0] = 0
print(s) # -> [1, 5]
```

* برای دسترسی به عناصر لیست و یا رشته داخلی در هنگام slicing باید از یک براکت [] دیگر نیز استفاده کنیم.

```
x = [1, 5, ['ali', 'b']]
print(x[2][0]) # -> ali
print(x[2][0][1]) # -> l
```

* اگر یک لیست داخل لیست داشد و با استفاده از متدهای copy لیست را کپی کنیم و کپی لیست را تغییر بدھیم، تغییرات در لیست داخلی نیز اعمال می‌شوند، به این نوع کپی، کپی سطحی می‌گوییم که در این حالت عناصر داخلی کپی نمی‌شوند بلکه ارجاع داده می‌شوند.

```
x = [1, 5, ['ali', 'b']]
z = x.copy()
z[2][0] = 'reza'
print(x) # -> [1, 5, ['ali', 'b']]
```

* برای حل این مشکل می‌توانیم از کپی عمیق با استفاده از متدهای copy ماذول deepcopy استفاده کنیم که در آن عناصر داخلی نیز کپی می‌شوند.

```
from copy import deepcopy

x = [1, 5, ['ali', 'b']]
z = deepcopy(x)
z[2][0] = 'reza'
print(x) # -> [1, 5, ['ali', 'b']]
```

درس ۱۸: لیست‌ها در پایتون (بخش سوم)

* با استفاده از متدها می‌توانیم عنصر جدید به لیست اضافه کنیم.

```
x = [1, 5]
x.append(6)
print(x) # -> [1, 5, 6]
```

* در هنگام تغییر عناصر لیست می‌توانیم چند ایندکس را انتخاب کنیم و تغییر بدهیم.

```
x = [1, 5, 6, 7, 8]
x[2:5] = ['a', 'b']
print(x) # -> [1, 5, 'a', 'b']
```

* برای حذف یک یا چند عنصر از لیست می‌توانیم از `del` استفاده کنیم.

```
x = [1, 5, 6]
del x[2]
print(x) # -> [1, 5]
```

* برای مشخص کردن تعداد عناصر لیست می‌توانیم از تابع `len` استفاده کنیم.

```
x = [1, 5, 6]
print(len(x)) # -> 3
```

* در پایتون می‌توانیم از انتساب چندگانه استفاده کنیم، به این صورت که متغیرها را به ترتیب بنویسیم و مقداردهی کنیم.

```
a, b, c = [2, 3, 5]
print(a) # -> 2
print(b) # -> 3
print(c) # -> 5
```

* در هنگام انتساب چندگانه اگر تعداد مقادیر از متغیرها بیشتر بود، با گذاشتن **ستاره** (*) قبل از متغیر، آن متغیر مقادیر اضافی را می‌گیرد.

```
a, *b, c, = [2, 3, 5, 1, 6, 9]
print(a) # -> 2
print(b) # -> [3, 5, 1, 6]
print(c) # -> 9
```

درس ۱۹: چند تایی‌ها (تاپل) در پایتون

* در تاپل می‌توانیم هر نوع داده‌ای بنویسیم و بهتر است عناصر را داخل پرانتز () قرار دهیم و با کاما (،) آن‌ها را جدا کنیم.

```
x = (1, 2, 'reza', [5, 1])
```

* در لیست‌ها بهتر است از یک نوع داده استفاده کنیم و از تاپل‌ها بهتر است زمانی استفاده کنیم که چند نوع داده داریم.

* تاپل‌ها **تغییر ناپذیر (immutable)** هستند.

```
x = (1, 2, 3)  
x[1] = 5
```

```
TypeError: 'tuple' object does not support item assignment
```

* تاپل‌ها به دلیل تغییر ناپذیر بودن، در مصرف حافظه بهینه‌ترند.

* لیست‌های درون تاپل تغییر پذیرند.

```
x = (1, 2, [3, 4])  
x[2][1] = 6  
print(x) # -> (1, 2, [3, 6])
```

* برای تغییر دادن عناصر تاپل می‌توانیم ابتدا آن را به لیست تبدیل کنیم و بعد از اعمال تغییرات با استفاده از **tuple** دوباره لیست را به تاپل تبدیل کنیم.

```
x = (1, 2, 3)  
x = list(x)  
x[2] = 6  
x = tuple(x)  
print(x) # -> (1, 2, 6)
```

* هنگامی که می‌خواهیم فقط یک عدد را در تاپل قرار دهیم باید بعد از آن از کاما (،) استفاده کنیم چون در غیر این صورت نوع آن **int** می‌باشد.

```
x = 1,  
print(type(x)) # -> <class 'tuple'>
```

درس ۲۰: دیکشنری‌ها در پایتون (قسمت اول)

* برای دیکشنری از آکولاد `{}` استفاده می‌کنیم، دیکشنری از دو قسمت کلید و مقدار تشکیل شده است، ابتدا کلید قرار دارد و سپس مقدارش در جلوی آن و پس از `:` قرار دارد.

* کلیدها در یک دیکشنری نمی‌توانند تکراری باشند اما مقادیر می‌توانند.

ساختار دیکشنری:

Dictionary: `{key: value, key: value}`

```
d = {'a': 3, 'b': 7, 'c': 3}
```

* برای به‌دست آوردن عناصر دیکشنری نمی‌توانیم از ایندکس آن‌ها استفاده کنیم و از کلیدها استفاده می‌کنیم.

```
d = {'a': 3, 'b': 7, 'c': 3}
print(d['b']) # -> 7
```

* کلیدها می‌توانند هر نوع داهی تغییر ناپذیر باشند، تاپل اگر عضو تغییر پذیر مانند لیست داشته باشد نمی‌توان از آن به‌عنوان کلید دیکشنری استفاده کرد.

* مقادیر می‌توانند هر نوعی باشند.

```
d = {'a': 3, 'b': 7, 5: [1, 2], (3, 5): {'a': 2}}
```

* در دیکشنری‌ها برخلاف توالی‌ها (لیست، تاپل و رشته) ترتیب مهم نیست چون از ایندکس استفاده نمی‌کنیم، بلکه از کلید استفاده می‌کنیم.

* اگر از کلید تکراری استفاده کنیم مقدار کلید جایگزین می‌شود.

```
d = {'a': 3, 'b': 7, 'a': 9}
print(d) # -> {'a': 9, 'b': 7}
```

* مقادیر دیکشنری قابل تغییرند.

```
d = {'a': 3, 'b': 7, 'c': 9}
d['c'] = [1, 2]
print(d) # -> {'a': 3, 'b': 7, 'c': [1, 2]}
```

* می‌توانیم کلید و مقدار جدید به دیکشنری اضافه کنیم.

```
d = {'a': 3, 'b': 7}
d['m'] = 1,
print(d) # -> {'a': 3, 'b': 7, 'm': (1,)}
```

* با استفاده از متده `get` می‌توانیم مقدار یک کلید را به‌دست آوریم و در صورتی که کلید وجود نداشته باشد نتیجه `None` است.

```
d = {'a': 3, 'b': 7}
print(d.get('a')) # -> 3
print(d.get('w')) # -> None
```

* با استفاده از متدهای **keys** کلیدهای یک دیکشنری را به دست می آوریم.

```
d = {'a': 3, 'b': 7}
print(d.keys()) # -> dict_keys(['a', 'b'])
```

* با استفاده از متدهای **values** مقادیر یک دیکشنری را به دست می آوریم.

```
d = {'a': 3, 'b': 7}
print(d.values()) # -> dict_values([3, 7])
```

* با استفاده از متدهای **items** کلیدها و مقادیر یک دیکشنری را به دست می آوریم.

```
d = {'a': 3, 'b': 7}
print(d.items()) # -> dict_items([('a', 3), ('b', 7)])
```

* عناصر دیکشنری را می توانیم با استفاده از **list** به لیست تبدیل کنیم.

```
d = {'a': 3, 'b': 7}
print(list(d.keys())) # -> ['a', 'b']
print(list(d.values())) # -> [3, 7]
print(list(d.items())) # -> [('a', 3), ('b', 7)]
```

درس ۲۱: دیکشنری‌ها در پایتون (قسمت دوم)

* برای حذف یک عنصر از دیکشنری می‌توانیم از `del` استفاده کنیم.

```
d = {'a': 3, 'b': 7}
del d['b']
print(d) # -> {'a': 3}
```

* کلیدهای دیکشنری تغییر نمی‌کنند اما می‌توانیم ابتدا مقدار را بدر یک متغیر قرار دهیم، سپس کلید را حذف کنیم و درنهایت یک کلید جدید ایجاد کنیم و مقدار ذخیره شده از قبل را به آن بدهیم.

```
d = {'a': 3, 'b': 7}
x = d['b']
del d['b']
d['x'] = x
print(d) # -> {'a': 3, 'x': 7}
```

* از تابع `sorted` می‌توانیم برای مرتب سازی استفاده کنیم، اگر عناصر عدد باشند از کوچک به بزرگ و اگر حروف باشند بر اساس حروف الفبا مرتب می‌شوند.

```
a = [15, 1, 7]
b = ['c', 'a', 'm']
print(sorted(a)) # -> [1, 7, 15]
print(sorted(b)) # -> ['a', 'c', 'm']
```

* اگر بخواهیم تابع `sorted` مرتب سازی را بر عکس انجام دهد، باید مقدار `reverse` را `True` قرار دهیم.

```
a = [15, 1, 7]
b = ['c', 'a', 'm']
print(sorted(a, reverse=True)) # -> [15, 7, 1]
print(sorted(b, reverse=True)) # -> ['m', 'c', 'a']
```

* از تابع `sorted` در دیکشنری نیز می‌توانیم استفاده کنیم.

```
d = {'c': 1, 'a': 3, 'z': 2}
print(sorted(d.keys())) # -> ['a', 'c', 'z']
print(sorted(d.values())) # -> [1, 2, 3]
print(sorted(d.items())) # -> [('a', 3), ('c', 1), ('z', 2)]
```

* با استفاده از تابع `dict` می‌توانیم دیکشنری ایجاد کنیم، به این صورت که هر مقدار و کلید داخل یک تاپل باشند و همه عناصر داخل لیست باشند.

```
d = dict([('a', 5), ('b', 10)])
print(d) # -> {'a': 5, 'b': 10}
```

* اگر کلیدها رشته باشند می‌توانیم به جای استفاده از لیست و تاپل به کلیدها مقدار را بدهیم.

```
d = dict(a=5, b=10)
print(d) # -> {'a': 5, 'b': 10}
```

* با استفاده از حلقه **for** (توضیحات در فصل ۵) نیز می‌توانیم دیکشنری ایجاد کنیم.

```
d = {x: x ** 2 for x in range(5)}
print(d) # -> {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

* می‌توانیم از دیکشنری‌های تودر تو استفاده کنیم.

```
d = {
    'first': {'name': 'reza', 'age': 24},
    'second': {'name': 'ali', 'age': 27}
}

print(d) # -> {'first': {'name': 'reza', 'age': 24}, 'second': {'name': 'ali', 'age': 27}}
```

* از دیکشنری نمی‌توانیم به عنوان کلید استفاده کنیم چون تغییر پذیر است ، اما می‌توانیم به عنوان مقدار استفاده کنیم.

* با استفاده از تابع **len** می‌توانیم مشخص کنیم دیکشنری چند عنصر دارد.

```
d = {'a': 10, 'b': 30}
print(len(d)) # -> 2
```

* در دیکشنری برخلاف لیست، تاپل و رشته نمی‌توانیم از تکرار ***** و جمع **+** استفاده کنیم.

* در دیکشنری می‌توانیم از سایر عملگرها (**is**, **in**, **==** و ...) استفاده کینم.

* از تابع **zip** می‌توانیم برای زیپ کردن دو لیست به هم استفاده کنیم.

```
k = ['a', 'b', 'c']
v = [1, 2, 3]
z = zip(k, v)
print(list(z)) # -> [('a', 1), ('b', 2), ('c', 3)]

d = dict(zip(k, v))
print(d) # -> {'a': 1, 'b': 2, 'c': 3}
```

درس ۲۲: مجموعه‌ها در پایتون (قسمت دوم)

- * به اشیایی که دارای ویژگی‌ها و صفات مشترک هستند، مجموعه گفته می‌شود.
- * مجموعه‌ها در پایتون با آکولاد `{}` مشخص می‌شوند و عناصر با کاما `,` از یکدیگر جدا می‌شوند.

```
s = {1, 5, 6, 9}
```

- * مجموعه‌ها قابل تغییرند.

- * برای اضافه کردن یک عنصر جدید به مجموعه از متده استفاده می‌کنیم.

```
s = {1, 5}
s.add(8)
print(s) # -> {8, 1, 5}
```

- * مجموعه‌ها عناصر تکراری را حذف می‌کنند.

```
s = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 5}
```

```
print(s) # -> {1, 2, 5}
```

- * ترتیب در مجموعه اهمیت ندارد و نمی‌توانیم با ایندکس به عناصر مجموعه دست یابیم.

- * در مجموعه می‌توانیم از عملگرهای `is`, `in`, `==` و ... استفاده کینم.

- * آکولاد خالی `{}` تشکیل دیکشنری می‌دهد.

```
s = {}
print(s) # -> {}
print(type(s)) # -> <class 'dict'>
```

- * برای ایجاد مجموعه خالی باید از `set` استفاده کنیم.

```
s = set()
print(s) # -> set()
print(type(s)) # -> <class 'set'>
```

- * مجموعه‌ها نمی‌توانند به عنوان کلید دیکشنری قرار بگیرند، چون قابل تغییرند.

- * در مجموعه‌ها نمی‌توانیم از مقادیر قابل تغییر مانند لیست استفاده کنیم.

- * مجموعه‌ها هر نوع داده تغییر ناپذیر مانند عدد، رشته، تاپل را در خود ذخیره می‌کنند.

- * برای اضافه کردن چند عنصر به مجموعه از متده استفاده می‌کنیم.

```
s = set()
s.update([1, 2, 4], [3, 1, 4], [6, 7])
print(s) # -> {1, 2, 3, 4, 6, 7}
```

* برای حذف یک عنصر از مجموعه می‌توانیم از دو متده **discard** و **remove** استفاده کنیم، تفاوت این دو متده این است که اگر عنصر وجود نداشته باشد متده **remove** خطا می‌دهد اما متده **discard** خطا نمی‌دهد.

```
s = {1, 2, 3}
s.remove(3)
s.discard(2)
s.discard(5)
print(s) # -> {1}
```

* از تابع **len** برای شمردن تعداد عناصر مجموعه استفاده می‌کنیم، عناصر تکراری شمرده نمی‌شوند.

```
s = {1, 1, 1, 1, 2, 3}
print(len(s)) # -> 3
```

درس ۲۳: مجموعه‌ها در پایتون (قسمت دوم)

* **تفاضل دو مجموعه** عبارت است از همه‌ی مواردی که در مجموعه اول هستند اما در مجموعه دوم نیستند، برای تفاضل از **تفریق** (-) استفاده می‌کنیم، از متدهای **difference** نیز می‌توانیم برای بهدست آوردن تفاضل دو مجموعه استفاده کنیم.

```
p = {3, 9, 15, 12, 6, 18}
q = {4, 16, 10, 2, 8, 14, 12, 6, 18}

print(p - q) # -> {9, 3, 15}
print(p.difference(q)) # -> {9, 3, 15}
```

* **اجتماع دو مجموعه** عبارت است از همه‌ی مواردی که در دو مجموعه وجود دارد، برای اجتماع دو مجموعه از پایپ (|) استفاده می‌کنیم، از متدهای **union** نیز می‌توانیم برای بهدست آوردن اجتماع دو مجموعه استفاده کنیم.

```
p = {3, 9, 15, 12, 6, 18}
q = {4, 16, 10, 2, 8, 14, 12, 6, 18}

print(p | q) # -> {2, 3, 4, 6, 8, 9, 10, 12, 14, 15, 16, 18}
print(p.union(q)) # -> {2, 3, 4, 6, 8, 9, 10, 12, 14, 15, 16, 18}
```

* **اشتراک دو مجموعه** عبارت است از همه‌ی مواردی که در دو مجموعه مشترک است، برای اشتراک دو مجموعه از & استفاده می‌کنیم، از متدهای **intersection** نیز می‌توانیم برای بهدست آوردن اشتراک دو مجموعه استفاده کنیم.

```
p = {3, 9, 15, 12, 6, 18}
q = {4, 16, 10, 2, 8, 14, 12, 6, 18}

print(p & q) # -> {18, 12, 6}
print(p.intersection(q)) # -> {18, 12, 6}
```

* **تفاضل مقارن دو مجموعه** به معنی همه‌ی موارد دو مجموعه به جز اشتراک آن‌ها می‌باشد، برای تفاضل مقارن دو مجموعه از ^ استفاده می‌کنیم، از متدهای **symmetric_difference** نیز می‌توانیم برای بهدست آوردن تفاضل مقارن دو مجموعه استفاده کنیم.

```
p = {3, 9, 15, 12, 6, 18}
q = {4, 16, 10, 2, 8, 14, 12, 6, 18}

print(p ^ q) # -> {2, 3, 4, 8, 9, 10, 14, 15, 16}
print(p.symmetric_difference(q)) # -> {2, 3, 4, 8, 9, 10, 14, 15, 16}

print((q | p) - (q & p)) # -> {2, 3, 4, 8, 9, 10, 14, 15, 16}
print((q - p) | (p - q)) # -> {2, 3, 4, 8, 9, 10, 14, 15, 16}
```

* یک مجموعه زمانی زیرمجموعه‌ی یک مجموعه است که همه‌ی عناصر آن در مجموعه‌ی دیگر باشد.

* برای بررسی زیرمجموعه بودن یک مجموعه می‌توانیم از `<` و `>` و متدهای `issuperset` و `issubset` استفاده کنیم.

```
a = {10, 30, 60, 40, 50, 20}
b = {10, 30, 60}

print(b < a)    # -> True
print(a < b)    # -> False
print(a > b)    # -> True
print(a.issuperset(b))  # -> True
print(a.issubset(b))   # -> False
```

درس ۲۴: حل مثال برای مبحث دیکشنری، لیست و مجموعه

- * استفاده از لیست یا تاپل ندارد، اما بهتر است در مواردی که نیاز به تغییر است، از لیست استفاده شود.
۱. با استفاده از لیست‌ها، نام و نمرات سه درس یک دانشجو را دریافت کنیم و میانگین نمرات آن را نشان دهیم.

```
marks = []
name = input('name: ')
physics = int(input('physics mark: '))
marks.append(physics)
biology = int(input('biology mark: '))
marks.append(biology)
math = int(input('math mark: '))
marks.append(math)
ave = (marks[0] + marks[1] + marks[2]) / 3
print(f'marks: {marks}\naverage: {ave:.2f}')
```

۲. یک شماره تلفن یا کدملی را به صورت رشته از کاربر بگیریم و همهی اعداد آن را به صورت یک رشته در یک لیست ذخیره کنیم.

```
number = input('number: ')
print(list(number))
```

۳. دو لیست داریم، به جای آخرین عنصر لیست اول، لیست دوم را بگذاریم.

```
l1 = [1, 2, 3]
l2 = ['a', 'b', 'c']
l1[-1] = l2
print(l1) # -> [1, 2, ['a', 'b', 'c']]
```

۴. یک برنامه بنویسیم که یک رشته از کاربر بگیرد و تعداد کاراکترها را به جز کاراکترهای تکراری بشمارد.

```
s = input('enter string: ')
chs = set(s)
print(chs)
print(len(chs))
```

۵. اسم و نمره دو دانشجو را در یک دیکشنری ذخیره کنیم.

```
marks = {}
name1 = input('name: ')
mark1 = float(input('mark: '))
marks[name1] = mark1
name2 = input('name: ')
mark2 = float(input('mark: '))
marks[name2] = mark2
print(marks)
```

درس ۲۵ و ۲۶: تکالیف مبحث دیکشنری و مجموعه و حل آنها

۱. فرض کنید شما قرار است یک دیکشنری واقعی بنویسید. با استفاده از نوع داده دیکشنری آن را پیاده سازی کنید. دقت کنید که کلمه اصلی باید کلید دیکشنری باشد و معانی آن میتوانند بیشتر از یک کلمه باشند. معانی را از کاربر بپرسید.

```
dictionary = {}
key = input('word: ')
meaning = input('meaning: ').split(',')
dictionary[key] = meaning

key = input('word: ')
meaning = input('meaning: ').split(',')
dictionary[key] = meaning

print(dictionary)
```

۲. حالا پس از انجام سوال قبلی، بخشی را به برنامه اضافه کنید تا یک کلمه را از کاربر گرفته و معانی آن را از دیکشنری خوانده و نمایش دهد.

```
word = input('your word: ')
print('meanings: ', dictionary[word])
```

۳. دو لیست مختلف از شماره تلفن های افراد مختلف داریم. ممکن است بعضی از شماره ها در لیست ها تکراری باشد (مثلآ ۰۹۳۹ در هر دو لیست وجود داشته باشد). لیست جدیدی ایجاد کنید که دو لیست قبلی را ترکیب میکند ولی تکراری ها در آن وجود ندارند.(از نوع داده مجموعه کمک بگیرید)

```
phones1 = ['0939', '0914', '0903']
phones2 = ['0938', '0919', '0903']
s = list(set(phones1 + phones2))
print(s) # -> ['0938', '0914', '0903', '0939', '0919']
```

درس ۲۷: بولین و `NoneType` در پایتون

* اگر متغیر، مقدار نداشته باشد می‌توانیم از `None` استفاده کنیم.

```
n = None
print(n) # -> None
print(type(n)) # -> <class 'NoneType'>
```

* بولین به مقادیر `True` و `False` گفته می‌شود که در واقع عدد هستند، `1` نمایانگر `True` و `0` نمایانگر `False` می‌باشد.

```
t = True
f = False
print(t) # -> True
print(type(t)) # -> <class 'bool'>
print(f) # -> False
print(type(f)) # -> <class 'bool'>
print(int(t)) # -> 1
print(int(f)) # -> 0
print(t + 1) # -> 2
print(f + 1) # -> 1
print(True == 1) # -> True
print(False == 0) # -> True
```

* در شرط‌ها `0` است، اما هر عددی به غیر از `0` درنظر گرفته می‌شود.

```
if 0:
    print(0)
if 1:
    print(1)
if -54.23:
    print(-54.23)
```

```
1
-54.23
```

* در شرط‌ها `False`، `None` درنظر گرفته می‌شود.

* در شرط‌ها تمام اشیای خالی (`False`، `''`، `()`، `{}`، `[]`، `set()`) درنظر گرفته می‌شوند.

* از تابع `bool` می‌توانیم برای تبدیل به بولین استفاده کنیم.

```
print(bool([])) # -> False
print(bool(24)) # -> True
```

درس ۲۸: عملگرهای منطقی با نوع داده

* در حالتی که دو طرف **and**, عدد یا نوع داده پر یا خالی داشته باشیم و هیچ عملگر مقایسه‌ای (`<=>`) نداشته باشیم از قاعده‌ی زیر استفاده می‌کنیم.

```
print(x and y)
if x is False -> x
if x is True -> y

print([] and 2) # -> []
print(5 and 10) # -> 10
print(0 and 4 < 10) # -> 0
print(5 and 4 < 10) # -> True
```

* در حالتی که دو طرف **or**, عدد یا نوع داده پر یا خالی داشته باشیم و هیچ عملگر مقایسه‌ای (`<=>`) نداشته باشیم از قاعده‌ی زیر استفاده می‌کنیم.

```
print(x or y)
if x is False -> y
if x is True -> x

print([] or 2) # -> 2
print(5 or 10) # -> 5
print(0 or 4 < 10) # -> True
print(5 or 4 < 10) # -> 5
```

پایان فصل سوم

فصل چهارم: دستورات کنترلی (تصمیم)

درس ۱: گرفتن چند ورودی در یک خط

* می‌توانیم چند ورودی در یک خط از کاربر بگیریم.

```
x, y, z = input('x, y, z: ').split(' ')
print('x:', x, '\ny:', y, '\nz:', z)
```

```
x, y, z: 2 8 1
x: 2
y: 8
z: 1
```

* باید تعداد متغیرها با تعداد ورودی‌ها برابر باشند و ورودی‌ها به صورت خودکار رشته هستند.

* برای اینکه بتوانیم تعداد زیادی ورودی بگیریم از لیست استفاده می‌کنیم، به اینصورت که یک رشته می‌گیریم و با کاما و یا ... جدا می‌کنیم.

```
x = input('Enter string: ').split(',')
print(x)
```

```
Enter string: this,34,ok,14
['this', '34', 'ok', '14']
```

* همچنین می‌توانیم از حلقه‌ها (در درس‌های آتی بررسی می‌شود) استفاده کنیم.

```
x = [int(x) for x in input('x: ').split(',')]
print(x)
```

```
x: 1,2,3,4,5,8
[1, 2, 3, 4, 5, 8]
```

درس ۲: ساختار تصمیم (بخش اول)

* برای بیان شرط از **if** استفاده می‌کنیم.

* ابتدا **if** را می‌نویسم و سپس شرط را می‌نویسم.

* **if** براساس **True** و **False** عمل می‌کند.

* بعد از شرط **دونقطه** (:) می‌گذاریم و دستورمان را با رعایت **فروفتگی** می‌نویسیم.

* بخش دیگر **if** دستور **else** است که معنی در غیر اینصورت می‌دهد.

* فقط یک **else** می‌توانیم استفاده کنیم و نوشتن **else** اختیاری است.

* ساختار کلی:

```
if True:  
    print(x)  
else:  
    print(y)
```

مثال: نمره‌ی یک دانشآموز را بگیرید و مشخص کنید که قبول شده است یا مردود.

```
x = int(input('mark: '))  
if x < 10:  
    print('failed!')  
else:  
    print('passed!')
```

* برای اینکه بتوانیم در **if** از شرط‌های دیگر استفاده کنیم، از **elif** استفاده می‌کنیم.

* بی‌نهایت **elif** می‌توانیم بنویسیم.

* فقط یک **if** و یک **else** می‌توانیم بنویسیم.

* **elif** باید بین **if** و **else** باشد.

مثال: نمرات یک دانشآموز را گرفته و براساس حروف الفبا رتبه بندی کنیم.

```
mark = int(input('mark: '))  
if 15 <= mark <= 20:  
    print('A')  
elif 10 <= mark < 15:  
    print('B')  
elif 5 <= mark < 10:  
    print('C')  
elif 0 <= mark < 5:  
    print('D')  
else:  
    print('invalid!')
```

درس ۳: ساختار تصمیم (بخش دوم + مثال)

* در ساختار **if** اگر یکی از شرط‌ها درست باشد، بقیه بررسی نمی‌شوند.

```
x = 7
if x > 5:
    print('yes') # -> yes
elif x == 7:
    print('of course')
```

* برای حل این مشکل می‌توانیم **if** های متعدد بنویسیم که به هم ربطی ندارند و هر کدام بررسی می‌شوند.

```
x = 7
if x > 5:
    print('yes') # -> yes
if x == 7:
    print('of course') # -> of course
if x < 12:
    print('absolutely') # -> absolutely
```

* اگر در **if** از تورفتگی استفاده کنیم و **if** دیگری بنویسیم، آن وقت پس از اینکه شرط اول درست بود، سراغ شرط دوم می‌رود.

```
mark = int(input('mark: '))
if mark > 10:
    print('passed')
    if 15 <= mark <= 20:
        print('A')
    elif 10 <= mark < 15:
        print('B')
else:
    print('failed')
```

* **if** را می‌توانیم در یک خط بنویسیم، در صورتی که فقط از **if** و **else** استفاده کنیم.

```
y = int(input('y: '))
x = 10 + 2 if y < 20 else 5
print(x)
```

۱. برنامه‌ای بنویسید که تشخیص دهد عددی که کاربر وارد کرده است زوج است یا فرد.

```
x = int(input('x: '))
if x % 2 == 0:
    print('even')
else:
    print('odd')
```

۲. برنامه‌ای بنویسید که تشخیص دهد بین دو عددی که کاربر وارد می‌کند، کدام کوچک‌تر است.

```
x = int(input('x: '))
y = int(input('y: '))
if x < y:
    print('min is x')
elif x == y:
    print('x equal to y')
else:
    print('min is y')
```

۳. برنامه‌ای بنویسید که تشخیص دهد بین سه عددی که کاربر وارد می‌کند، کدام کوچک‌تر است.

```
x = int(input('x: '))
y = int(input('y: '))
z = int(input('z: '))
min_digit = x
if y < min_digit:
    min_digit = y
if z < min_digit:
    min_digit = z

print(min_digit)
```

درس ۴ و ۵: تکالیف مبحث ساختار تصمیم و حل آنها

۱. یک عدد از کاربر بگیرید و بررسی کنید که آیا هم بر ۲ و هم بر ۵ بخش پذیر است یا نه.

```
x = int(input('x: '))
if x % 2 == 0 and x % 5 == 0:
    print('yes')
else:
    print('no')
```

۲. اضلاع یک مثلث را از کاربر گرفته و مشخص کنید که آیا این اضلاع تشکیل مثلث می‌دهند یا خیر. اگر جواب مثبت است، چه نوع مثلثی؟ (متساوی الساقین، مختلف الاضلاع، قائم الزاویه یا متساوی الاضلاع)

```
x = int(input('x: '))
y = int(input('y: '))
z = int(input('z: '))
if x + y > z and x + z > y and y + z > x:
    print('yes, it is a triangle.')
    if x == y == z:
        print('it is a equilateral triangle.')
    if x == y or x == z or y == z:
        print('it is a isosceles triangle')
    if x ** 2 + y ** 2 == z ** 2 or x ** 2 + z ** 2 == y ** 2 or y ** 2 + z ** 2 == x ** 2:
        print('it is a right triangle')
    if x != y != z:
        print('it is a different sided triangle.')
else:
    print('no, it is not a triangle.')
```

۳. یک کاراکتر از کاربر گرفته و مشخص کنید کاراکتر وارد شده عدد است، حروف انگلیسی است یا سایر نمادها. (از کد اسکی کمک بگیرید).

```
character = input('enter character: ')
if 97 <= ord(character) <= 122:
    print(f'{character} is a Small English letter.')
elif 65 <= ord(character) <= 90:
    print(f'{character} is a Capital English letter.')
elif 48 <= ord(character) <= 57:
    print(f'{character} is a number.')
else:
    print(f'{character} is a symbol or other languages letter.')
```

درس ۶: توابع `sum` و `min` و `max`

* برای مشخص کردن عدد کوچک‌تر از تابع `min` استفاده می‌کنیم.

```
print(min(2, 3, 1, 4, 7, 8, 2, 3, 6)) # -> 1
```

* اگر به ورودی تابع `min` لیست و یا تاپل خالی ارسال کنیم، می‌توانیم مقداردهی کنیم.

```
print(min([], default=567))
```

* برای مشخص کردن عدد بزرگ‌تر از تابع `max` استفاده می‌کنیم.

```
print(max(2, 3, 1, 4, 7, 8, 2, 3, 6)) # -> 8
```

* برای رشته و حروف انگلیسی نیز می‌توان از `min` و `max` استفاده کنیم، که براساس اولین حرف هر رشته و مطابق ترتیب حروف الفبا مرتب می‌کند.

```
print(min('reza', 'neda', 'sahel', 'ali')) # -> ali
print(max('reza', 'neda', 'sahel', 'ali')) # -> sahel
```

* برای جمع کردن از تابع `sum` استفاده می‌کنیم که ورودی آن باید **لیست**، **تاپل** یا **مجموعه** باشد.

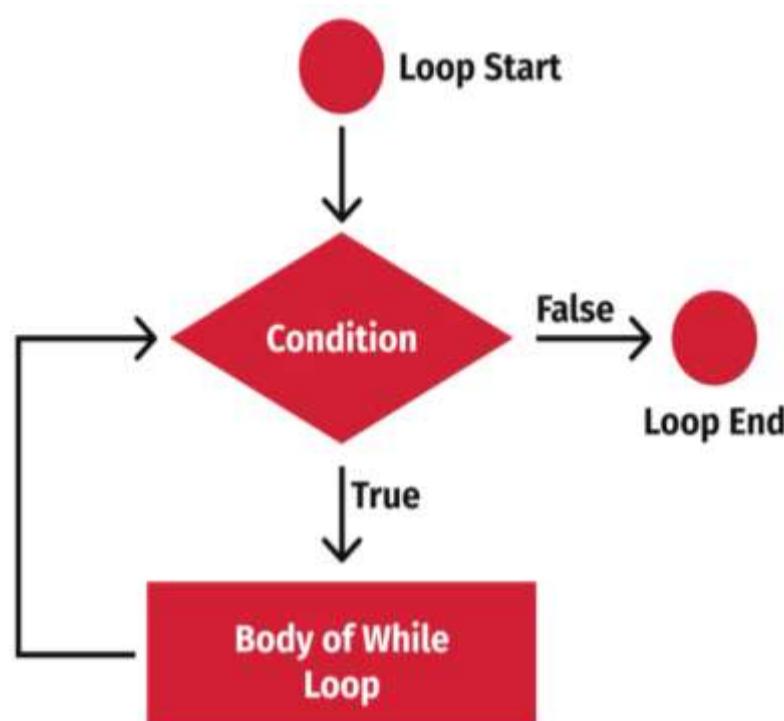
```
print(sum([1, 5, 6, 7])) # -> 19
print(sum((1, 5, 6, 7))) # -> 19
print(sum({1, 5, 6, 7})) # -> 19
```

* اگر در `sum` از `start` استفاده کنیم، نتیجه را با `start` جمع می‌کند.

```
print(sum({1, 5, 6, 7}, start=10)) # -> 19
```

پایان فصل چهارم

درس ۱: ساختار حلقه while



* برای تکرار می‌توانیم از حلقه‌ی **while** استفاده کنیم و تا زمانی که شرایط برقرار باشد حلقه تکرار می‌شود.

```
x = 0
while x < 3:
    print('repeat!')
    x += 1
```

* در این مثال در هر تکرار به **X** یکی اضافه می‌شود و تا زمانی که **X** کوچک‌تر از ۳ است حلقه تکرار می‌شود، پس حلقه سه بار تکرار می‌شود.

```
repeat!
repeat!
repeat!
```

* بعضی حلقه‌ها ممکن است بی‌نهایت اجرا شوند و باید مراقب این حلقه‌ها بود چون ممکن است سبب ایجاد مشکلات فراوانی شوند.

```
x = 0
while x < 3:
    print('repeat!')

while True:
    print('ok')
```

درس ۲: حل مثال برای حلقه while

۱. برنامه‌ای بنویسید که اعداد زوج دو رقمی را پیدا کند.

```
x = 10
while x < 100:
    if x % 2 == 0:
        print(x)
    x += 1
```

۲. برنامه‌ای بنویسید که اعداد سه رقمی که هم بر ۳ و هم بر ۷ بخش پذیرند را چاپ کند.

```
x = 100
while x < 1000:
    if x % 3 == 0 and x % 7 == 0:
        print(x)
    x += 1
```

۳. برنامه‌ای بنویسید که یک مثلث با یک شکلک خاص ایجاد کند و تعداد خطها را کاربر وارد کند.

```
r = 1
n = int(input('enter rows: '))

while r <= n:
    print('*' * r)
    r += 1
```

```
enter rows: 5
*
**
***
****
*****
```

۴. چاپ مثلث سوال قبل به صورت برعکس.

```
n = int(input('enter rows: '))

while n > 0:
    print('*' * n)
    n -= 1
```

```
enter rows: 5
*****
****
 ***
 **
 *
```

۵. مقسوم علیه‌های یک عدد را چاپ کنید. (مشخص کنید که آن عدد بر چه اعدادی بخش پذیر است).

```
n = int(input('enter a number: '))
x = 1

while n >= x:
    if n % x == 0:
        print(x)
    x += 1
```

۶. برنامه‌ای بنویسید که مشخص کند یک عدد کامل است یا خیر. (عددی که جمع مقسوم علیه‌هایش(جز خودش) با خودش برابر شودند، عددی کامل است.)

```
n = int(input('enter a number: '))
x = 1
divisor = 0
while n > x:
    if n % x == 0:
        divisor += x
    x += 1

if n == divisor:
    print('it is a perfect number.')
else:
    print('it is not a perfect number.')
```

۷. برنامه‌ای بنویسید که سری فیبوناچی را چاپ کند. (تا ۲۰ جمله و از ۰ شروع شود)

* سری **فیبوناچی** یک دنباله‌ای از اعداد است که هر عدد از جمع ۲ عدد قبل از خودش به دست می‌آید و این سری یا از ۰ و یا از ۱ شروع می‌شود. (۰, ۱, ۱, ۲, ۳, ۵, ۸, ۱۳, ۲۱, ۳۴, ...)

```
x = 0
y = 1
fibonacci = []
counter = 0
while counter <= 20:
    fibonacci.append(x)
    x, y = y, x + y # همزمان اجرا شوند
    counter += 1

print(fibonacci)
```

درس ۳: دستور while در حلقه break, continue, else (+ مثال)

* در while زمانی که بخواهیم از حلقه خارج شویم از break استفاده می‌کنیم.

* با break می‌توانیم حلقه‌های بینهايت را کنترل کنیم.

```
i = 1
while i <= 100:
    print(i, end=' - ') # -> 1-2-3-
    if i == 3:
        break
    i += 1
```

مثال: تعدادی عدد دلخواه از کاربر بگیریم و کوچک‌ترین آن‌ها را پیدا کنیم.

```
n = float(input('number: '))
minimum = n
while True:
    s = input('do you want to continue?(y/n) ')
    if s.lower() == 'n':
        break
    n = float(input('number: '))
    if n < minimum:
        minimum = n
print(n)
```

* اگر در حلقه‌ی while بخواهیم به اول حلقه برگرد و بقیه دستورات نادیده گرفته شود از continue استفاده می‌کنیم.

```
i = 0
while i <= 10:
    i += 1
    if i % 3 == 0:
        continue
    print(i)
```

* در while از دستور else می‌توانیم استفاده کنیم، اما زمانی اجرا می‌شود که حلقه به صورت نرمال به اتمام برسد و break اتفاق نیفتد.

```
i = 0
while i <= 10:
    i += 1
    if i % 3 == 0:
        continue
    print(i)
else:
    print('ok!')
```

مثال: یک عدد از کاربر بگیریم و تشخیص بدهیم که اول هست یا خیر. (عدد اول فقط بر خودش و یک بخش پذیر است).

```
number = int(input('enter a number: '))
i = 2

if number > 1:
    while i <= number/2:
        if number % i == 0:
            print(f'{number}' is not a prime number')
            break
        i += 1
    else:
        print(f'{number}' is a prime number')
else:
    print(f'{number}' is not a prime number')
```

درس ۴: حلقه while تو در تو (+ مثال)

* در حلقه while می‌توانیم حلقه‌های while دیگری نیز تعریف کنیم.

```
i = 0
while i <= 10:
    j = 1
    while j <= 10:
        print(j)
        j += 1
    print(i)
    i += 1
```

مثال: یک برنامه بنویسید که عناصر یک لیست را چاپ کند.

```
names = ['reza', 'neda', 'shadi', 'ali']

i = 0
while i < len(names):
    print(names[i])
    i += 1
```

مثال: یک برنامه بنویسید که یک عناصر یک لیست را چاپ کند به صورتی که حروف هر اسم یک در میان حرف بزرگ باشد.

```
names = ['reza', 'neda', 'shadi', 'ali']

i = 0
while i < len(names):
    name = names[i]
    j = 0
    while j < len(name):
        if j % 2 == 0:
            print(name[j].upper(), end=' ')
        else:
            print(name[j].lower(), end=' ')
        j += 1
    print()
    i += 1
```

مثال: یک برنامه بنویسید که یک جدول ضرب ده در ده چاپ کند.

```
i = 1
while i <= 10:
    j = 1
    while j <= 10:
        print(f'{i} * {j} = {i * j}', end='\t\t')
        j += 1
    print()
    i += 1
```

مثال: یک برنامه بنویسید که یک مثلث با یک نماد خاص چاپ کند. (با استفاده از حلقه‌های تو در تو)

```
row = int(input('enter row: '))
i = 1
while i <= row:
    j = 1
    while j <= i:
        print('*', end=' ')
        j += 1
    print()
    i += 1
```

درس ۵: ساختار حلقه `for` (+ مثال)

* زمانی که بخواهیم در توالی‌ها (لیست، رشته، تاپل)، دیکشنری و به طور کلی در تکرار شونده‌ها پیمایش کنیم از حلقه `for` استفاده می‌کنیم.

* شکل کلی:

For `target` in `obj`:
↓ ↓
متغیر تکرار شونده

```
numbers = [1, 2, 3, 4, 5, 6]
for i in numbers:
    print(i)

name = 'reza dolati'
for s in name:
    print(s)

t = (1, 4, 5)
for i in t:
    print(i)
```

مثال: یک لیست از کد اسکی‌های مختلف داریم. یک برنامه بنویسید که کاراکتر این کدها را چاپ کند.

```
codes = [67, 76, 34, 343, 234, 168, 935]
for c in codes:
    print(chr(c))
```

* در حلقه `for` باید قواعد تورفتگی رعایت شود.

* معمولاً زمانی از `for` استفاده می‌کنیم که تعداد تکرار مشخص است.

* معمولاً زمانی از `while` استفاده می‌کنیم که نمی‌دانیم چند بار باید اجرا شود.

* دستورات `for` و `else` و `continue`، `break` که در حلقه `while` وجود دارند، در حلقه `for` نیز با همان ویژگی‌ها قابل اجرا هستند.

```
codes = [67, 76, 34, 343, 234, 168, 935]
for c in codes:
    if chr(c) == 'r':
        break
    print(chr(c))
else:
    print('done')
```

* در حلقه `for` می‌توانیم `for`‌های تو در تو داشته باشیم.

* در `for` می‌توانیم حلقه `while` تعریف کنیم و بالعکس.

مثال: یک برنامه بنویسید که موارد مشترک دو لیست را چاپ کند.

```
list_1 = [1, 2, 8, 9, 6, 7, 3]
list_2 = [12, 8, 10, 3, 9, 7, 1, 0, 20]

for i in list_1:
    for j in list_2:
        if i == j:
            print(i)
```

* می‌توانیم برای **for** چند متغیر بنویسیم.

```
for a, b, c in [[1, 2, 3], [4, 5, 6], [7, 8, 9]]:
    print(a)
    print(b)
    print(c)
    print(20 * '*')
```

* اگر مقادیر اضافی نسبت به متغیر داشته باشیم، می‌توانیم قبل متغیر ***** بگذاریم، تا مقادیر اضافی را به صورت یک لیست به خود بگیرد.

```
for a, b, *c in [[1, 2, 3, 0], [4, 5, 6, 0]]:
    print(a)
    print(b)
    print(c)
    print(20 * '*')
```

* برای دیکشنری در حالت عادی، کلیدها را انتخاب می‌کند.

```
d = {'a': 1, 'b': 2, 'c': 3}
for k in d:
    print(k)
```

a
b
c

* برای این‌که **for**، کلید، مقدار و کلید – مقدار را انتخاب کند می‌توانیم از متدهای **key**، **values** و **items** استفاده کنیم.

```
d = {'a': 1, 'b': 2, 'c': 3}
for k in d.keys():
    print(k)
for v in d.values():
    print(v)
for i in d.items():
    print(i)
```

* اگر بخواهیم **for** در دیکشنری کلید – مقدار را نیز به صورت کلید – مقدار چاپ کند از **دو متغیر** و متدهای **items** استفاده می‌کنیم.

```
d = {'a': 1, 'b': 2, 'c': 3}
for k, v in d.items():
    print(f'{k} : {v}')
```

a : 1
b : 2
c : 3

درس ۶: آشنایی با `range` (+ مثال)

* برای تکرار در `for` از `range` استفاده می‌کنیم.

```
print(range(10)) # -> range(0, 10)
print(type(range(10))) # -> <class 'range'>
print(list(range(10))) # -> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
for i in range(5):
    print(i)
```

0
1
2
3
4

* در `range` می‌توانیم شروع را مشخص کنیم.

```
for i in range(2, 5):
    print(i)
```

2
3
4

* در `range` می‌توانیم گام تعیین کنیم.

```
for i in range(2, 10, 2):
    print(i)
```

2
4
6
8

* اگر گام را عددی منفی وارد کنیم، از راست به چپ حرکت می‌کند.

```
for i in range(10, 2, -2):
    print(i)
```

10
8
6
4

مثال: یک رشته را به عنوان ورودی بگیریم و ایندکس عناصر و عناصر آن را چاپ کنیم.

```
s = input('enter string: ')
for i in range(0, len(s)):
    print(f'{i} : {s[i]}')
```

مثال: فاکتوریل یک عدد را محاسبه کنیم.

```
n = int(input('number: '))
m = 1
for i in range(1, n + 1):
    m *= i

print(m)
```

مثال: یک برنامه بنویسید که ارقام یک عدد را از آخر به اول بنویسد.

```
n = input('number: ')
s = ''
for i in range(len(n) - 1, -1, -1):
    s += n[i]

print(s)
```

مثال: یک برنامه بنویسید که یک مثلث با عدد ایجاد کند.

```
n = int(input('number: '))

for i in range(1, n + 1):
    for j in range(1, i + 1):
        print(j, end=' ')
    print()
for i in range(n - 1, 0, -1):
    for j in range(1, i + 1):
        print(j, end=' ')
    print()
```

```
number: 4
1
12
123
1234
123
12
1
```

درس ۷: تکنیک حلقه سازی (enumerate, zip, reversed, sorted)

* برای شماره گذاری عناصر توالی می‌توانیم از تابع `enumerate` استفاده کنیم، این تابع شماره گذاری را از ۰ و مطابق ایندکس عناصر شروع می‌کند.

```
print(list(enumerate(['a', 'b', 'c']))) # -> [(0, 'a'), (1, 'b'), (2, 'c')]

my_list = ['a', 'b', 'c', 'd']
for i, j in enumerate(my_list):
    print(i, ':', j)
```

0 : a
1 : b
2 : c
3 : d

* در تابع `enumerate` می‌توانیم مشخص کنیم که شماره گذرای از چه عددی آغاز شود.

```
my_list = ['a', 'b', 'c', 'd']
for i, j in enumerate(my_list, 5):
    print(i, ':', j)
```

5 : a
6 : b
7 : c
8 : d

* برای زیپ کردن دو لیست می‌توانیم از تابع `zip` استفاده کنیم.

```
names = ['reza', 'ali', 'neda', 'mina']
ages = [20, 18, 25, 36]

for i, j in zip(names, ages):
    print(f'name: {i} | age: {j}')
```

name: reza age: 20
name: ali age: 18
name: neda age: 25
name: mina age: 36

* برای پیمایش از آخر می‌توانیم از تابع `reversed` استفاده کنیم.

```
y = [20, 18, 25, 36]

for i in reversed(y):
    print(i, end=' - ') # -> 36-25-18-20-
```

* برای مرتب کردن از تابع `sorted` استفاده می‌کنیم، این تابع اعداد را از کوچک به بزرگ و رشته را بر اساس حروف الفبا مرتب می‌کند.

```
s = ['reza', 'ali', 'neda', 'mina']
x = [20, 18, 25, 36]

for i in sorted(s):
    print(i, end=' - ') # -> ali-mina-neda-reza-
for i in sorted(x):
    print(i, end=' - ') # -> 18-20-25-36-
```

* اگر بخواهیم مرتب سازی را بر عکس انجام دهیم از تابع `sorted` به همراه `reversed` استفاده می کنیم.

روش اول:

```
x = [20, 18, 25, 36]

for i in reversed(sorted(x)):
    print(i, end=' - ') # -> 36-25-20-18-
```

روش دوم:

```
x = [20, 18, 25, 36]

for i in sorted(x, reverse=True):
    print(i, end=' - ') # -> 36-25-20-18-
```

درس ۸: تولید عدد تصادفی با ماثول random (+مثال)

* روش کلی استفاده از ماثول‌ها:

روش اول:

```
import random

print(random.random()) # -> 0.9447637291295037
```

روش دوم:

```
from random import random

print(random()) # -> 0.3249790419192793
```

* برای تولید عدد تصادفی در بازه‌ی ۰ تا ۱ می‌توانیم از تابع random استفاده کنیم.

```
from random import random

print(random()) # -> 0.3869024475083328
```

* برای تولید یک عدد تصادفی بر اساس یک عدد خاص و تولید دوباره همان عدد می‌توانیم از تابع seed استفاده کنیم.

* تابع seed به ما کمک می‌کند بفهمیم چه اعداد تصادفی تولید شده است.

```
from random import random, seed

seed(100)
print(random()) # -> 0.1456692551041303
```

* اگر دوباره از عدد 100 استفاده کنیم باز همین عدد تصادفی تولید می‌شود.

```
from random import random, seed

seed(100)
print(random()) # -> 0.1456692551041303
```

* برای تغییر بازه‌ی random از بازه‌ی ۰ و ۱ به بازه‌ی دلخواه می‌توانیم از روش زیر استفاده کنیم.

```
min + (random() * (max - min))

from random import random

print(5 + (random() * (10 - 5))) # -> 9.336956376988947
print(1 + (random() * (5 - 1))) # -> 2.95583323830563
```

* برای تغییر بازه‌ی **random** از بازه‌ی ۰ و ۱ به بازه‌ی دلخواه به جای روش قبلی می‌توانیم از تابع **uniform** استفاده کنیم.

```
from random import uniform

print(uniform(5, 10)) # -> 6.729439839222446
print(uniform(1, 5)) # -> 3.3936402248976587
```

* برای تولید عدد تصادفی صحیح می‌توانیم از تابع **randint** استفاده کنیم، اما بهتر است از تابع **int** استفاده کنیم.

* وقتی از **randint** استفاده می‌کنیم ممکن است ورودی دوم نیز به عنوان عددی تصادفی تولید شود.

```
from random import randint

print(randint(1, 5)) # -> 5
print(randint(1, 5)) # -> 3
```

* برای تولید عدد تصادفی با گام از تابع **randrange** استفاده می‌کنیم.

```
from random import randrange

print(randrange(1, 100, 2)) # -> 27
print(randrange(0, 100, 2)) # -> 74
```

* برای انتخاب یک عنصر به صورت تصادفی از یک لیست از تابع **choice** استفاده می‌کنیم.

```
from random import choice

numbers = [2, 3, 6, 8, 1, 5]
print(choice(numbers)) # -> 8
```

* برای تشکیل یک لیست با چند عنصر از یک لیست دیگر به صورت تصادفی و با تعداد دلخواه از تابع **sample** استفاده می‌کنیم.

```
from random import sample

numbers = [2, 3, 6, 8, 1, 5]
print(sample(numbers, 3)) # -> [6, 1, 5]
```

* برای بهم زدن ترتیب عناصر لیست از تابع **shuffle** استفاده می‌کنیم.

```
from random import shuffle

numbers = [2, 3, 6, 8, 1, 5]
shuffle(numbers)
print(numbers) # -> [6, 2, 1, 3, 8, 5]
```

مثال: یک برنامه بنویسید که اگر یک سکه را ۱۰۰۰ بار بیندازیم چند بار شیر و چند بار خط می‌آید.

```
from random import choice

coin = ['head', 'tail']
head = 0
tail = 0
for _ in range(1000):
    r = choice(coin)
    if r == 'head':
        head += 1
    else:
        tail += 1

print(head)
print(tail)
```

مثال: یک برنامه بنویسید که اگر یک سکه را ۱۰۰۰۰ بار بیندازیم هر وجه چند بار می‌آید.

```
from random import randint

toss = {1: 0, 2: 0, 3: 0, 4: 0, 5: 0, 6: 0}
for _ in range(10000):
    toss[randint(1, 6)] += 1

print(toss)
```

درس ۹ و ۱۰: تکالیف مبحث حلقه و اعداد تصادفی و حل آنها

۱. برنامه‌ای بنویسید که دو عدد از کاربر گرفته و اعداد مابین آن‌ها را نمایش دهد.

```
x = int(input('x: '))
y = int(input('y: '))

minimum = min(x, y)
maximum = max(x, y)

for i in range(minimum + 1, maximum):
    print(i)
```

۲. برنامه‌ای بنویسید که دو عدد صحیح را گرفته و مقسوم علیه‌های مشترکشان را نمایش دهد.

```
x = int(input('x: '))
y = int(input('y: '))
m = min(x, y)

for i in range(1, m + 1):
    if x % i == 0 and y % i == 0:
        print(i, end=' ')
```

۳. برنامه‌ای را بنویسید که دو عدد صحیح را گرفته و بزرگ‌ترین مقسوم علیه مشترکشان را نمایش دهد.

```
x = int(input('x: '))
y = int(input('y: '))
m = min(x, y)

for i in range(m, 0, -1):
    if x % i == 0 and y % i == 0:
        print(i)
        break
```

۴. برنامه‌ای را بنویسید که دو عدد صحیح را گرفته و کوچک‌ترین مضرب مشترکشان را نمایش دهد.

```
x = int(input('x: '))
y = int(input('y: '))
min_ = min(x, y)
max_ = max(x, y)

for i in range(1, min_ + 1):
    if (max_ * i) % min_ == 0:
        print(max_ * i)
        break
```

۵. برنامه‌ای بنویسید که عدد صحیحی را گرفته و تعداد رقم‌هایش را نمایش دهد. (عدد را به رشتہ تبدیل نکنید).

```
x = int(input('x: '))
i = 0
while x > 0:
    x //= 10
    i += 1

print(i)
```

۶. برنامه‌ای بنویسید که تعداد سطر را از کاربر گرفته و شکل زیر را رسم کند.

```
*
**
***
****
```

```
r = int(input('r: '))

for i in range(1, r + 1):
    print(' ' * (r - i), end=' ')
    print('*' * i)
```

۷. ما یک لیست دهتایی از اسم‌های مختلف داریم. برنامه‌ای بنویسید که کاربر یک اسم را در ذهن خود بگیرد و برنامه حدس بزند آن اسم چیست. سپس از کاربر بپرسد که درست حدس زده است یا خیر. اگر اشتباه بود اسم دیگری حدس بزند. برنامه نباید اسمی را که قبلاً کاربر گفته است اشتباه است، دوباره نشان دهد.

مثلاً لیست ما شامل ["A", "B", "C"] است. کاربر در ذهن خود A را انتخاب می‌کند. برنامه حدس می‌زند حرف انتخابی B است و می‌پرسد آیا درست حدس زده‌ام؟ کاربر می‌گوید نه. برنامه دوباره حدس می‌زند A و می‌پرسد درست حدس زده‌ام؟ کاربر می‌گوید بله و برنامه تمام می‌شود.

```
from random import choice
names = ['fatemeh', 'reza', 'neda', 'hassan', 'ahmad', 'leyla', 'mohsen',
'mohammad', 'yasin', 'sina']
print(names)
input('choose a name, then I guess it (press enter)')
names_cp = names.copy()
while True:
    if len(names_cp) == 0:
        print('I guessed them all')
        break
    cmp_choice = choice(names_cp)
    ans = input(f'is your choice "{cmp_choice}"?(y/n) ')
    if 'y' in ans.lower():
        print('I won')
        break
    names_cp.remove(cmp_choice)
```

درس ۱۱: مینی پروژه: رمزنگاری و رمزگشایی ساده

```
while True:
    print('select your option:\n\t1)Encrypt\n\t2)Decrypt\n\t3)Exit')
    choice = input('your choice: ')
    if choice == '1':
        plain_text = input('text: ')
        encrypted_text = ''
        for i in plain_text:
            n = chr(ord(i) * 2) + 67
            encrypted_text += n
        print(encrypted_text)
        break
    elif choice == '2':
        encrypted_text = input('text: ')
        decrypted_text = ''
        for i in encrypted_text:
            m = chr((ord(i) - 67) // 2)
            decrypted_text += m
        print(decrypted_text)
        break
    elif choice == '3':
        print('see ya :)')
        break
    else:
        print('wrong option!')
        continue
```

درس ۱۲: مینی پروژه: پسورد ساز

```
from random import choice

import string

a = string.ascii_lowercase
A = string.ascii_uppercase
number = '0123456789'
symbols = '~!@#$%^&*()_+=][{}><?/'
pas = a + A + number + symbols
final_password = ''
while True:
    print('select your option:\n\t1)create a password\n\t2)Exit')
    s = input('your choice: ')
    if s == '1':
        r = int(input('how many characters? '))
        for i in range(r):
            chr_choose = choice(pas)
            final_password += chr_choose
        print('your password is:', final_password, '\n', 20 * '*')
    if s == '2':
        print('bye! \n', 20 * '*')
        break
    else:
        print('select 1 or 2\n', 20 * '*')
        continue
```

درس ۱۳: مینی پروژه: تایмер ساده

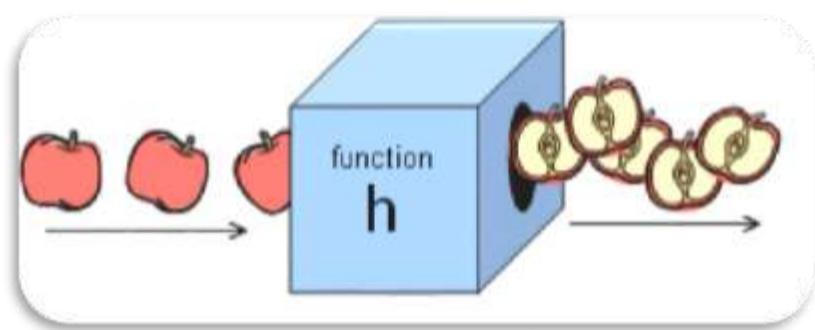
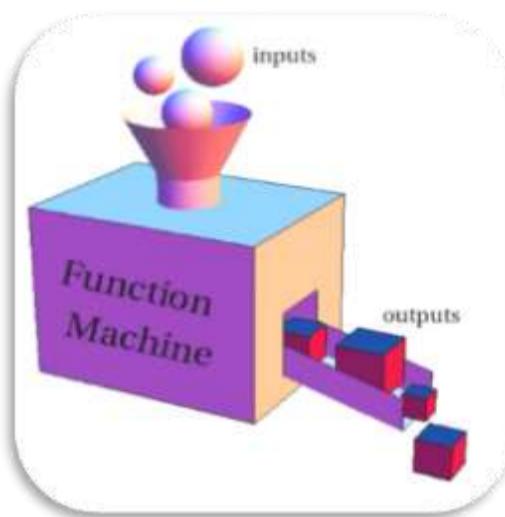
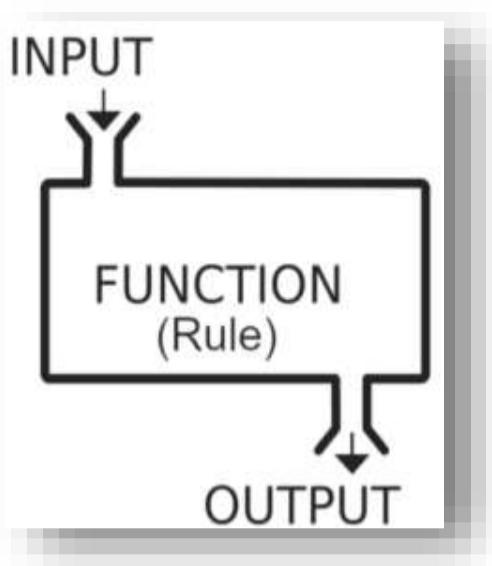
```
import time

while True:
    print('Do you want to start the timer?\n\t-yes\n\t-no')
    s = input('your option: ')
    if s.lower() == 'yes':
        hours = int(input('enter hours: '))
        minutes = int(input('enter minutes: '))
        seconds = int(input('enter seconds: '))
        total = (hours * 60 * 60) + (minutes * 60) + seconds
        for i in range(total, -1, -1):
            print(i)
            time.sleep(1)
        print('\n', 20 * '-')
        break
    if s.lower() == 'no':
        print('ok!\n', 20 * '-')
        break
    else:
        print('choose yes or no\nlets try it again')
        continue
```

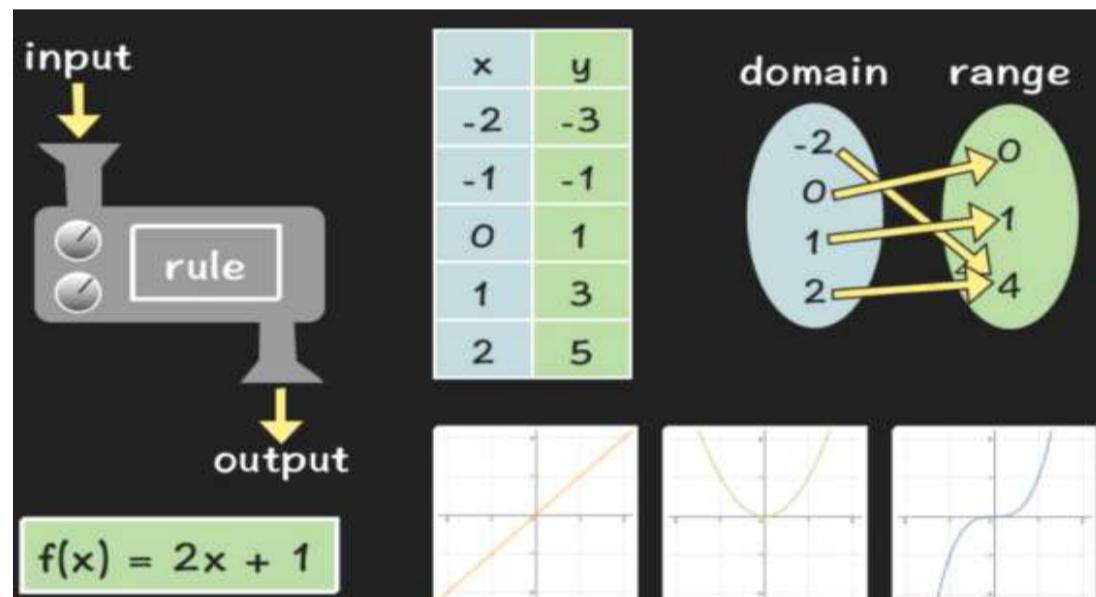
پایان فصل پنجم

درس ۱: مفهوم تابع (در زندگی، ریاضیات و برنامه نویسی)

* تابع در زندگی ورودی می‌گیرد و با انجام یک سری عملیات و پردازش خروجی را تحویل می‌دهد.



* تابع در ریاضی:



* در برنامه نویسی کدهایمان را در تابع می‌نویسیم و هر وقت به آن کدها نیاز داشتیم تابع را صدا می‌زنیم.

* قواعد تو رفتگی در تابع صدق می‌کند.

* برای تعریف تابع از کلمه کلیدی **def** استفاده می‌کنیم.

* برای نوشتن خروجی از کلمه کلیدی **return** استفاده می‌کنیم.

* برای فراخوانی تابع اسمش را می‌نویسیم و در صورت نیاز به آن ورودی می‌دهیم.

```

def f(x):
    return 2 * x + 1

print(f(-2))  # -> -3
  
```

درس ۲: سینتکس تابع و دستور pass

- * در پایتون نیاز نیست نوع ورودی و خروجی را تعیین کنیم.
- * اسم تابع بهتر است با حروف کوچک نوشته شود و کلمات با آندراسکور (_) جدا شوند.
- * بهتر است اسم تابع معنادار باشد و به کارکردش مربوط باشد.
- * در تابع ابتدا **def** را می‌نویسیم، سپس اسم تابع و در نهایت **پرانتز** می‌نویسیم و دو نقطه (:) می‌گذاریم و دستورات دیگر را در خط بعدی می‌نویسیم و قواعد **تو رفتگی** نیز باید رعایت شود و در انتهای پس از نوشتن دستورات، خروجی را به وسیله‌ی **return** مشخص می‌کنیم.
- * تابع تا زمانی که آن را صدا نزنیم، اجرا نمی‌شود.

```
def cub(x):
    if x > 3:
        return x ** 3

print(cub(4)) # -> 64
```

- * در پایتون ممکن است تابعی نیاز داشته باشیم که ورودی نیاز نداشته باشد.

```
def cube():
    x = int(input('x: '))
    return x ** 3

print(cube())
```

- * لزومی ندارد تابع خروجی داشته باشد.

```
def cube():
    x = int(input('x: '))
    print(x ** 3)

cube()
```

- * اگر برای تابع **return** ننویسیم و خروجی مشخص نکنیم، وقتی بخواهیم خروجی را ذخیره کنیم **None** پاسخ می‌دهد یا این‌که خودمان نیز می‌توانیم **return None** بنویسیم به معنی این‌که خروجی ندارد.

```
def cube():
    x = int(input('x: '))
    print(x ** 3)

n = cube()
print(n)
```

- * در بدنه تابع هرچقدر که دوست داریم می‌توانیم دستور بنویسیم.

* بعد از **return** هرچقدر دستور بنویسیم فایده ندارد چون تابع زمانی که به **return** برسد به پایان می‌رسد.

```
def cube():
    x = int(input('x: '))
    print('ok')
    x += 1
    return x ** 3
    print('ali')
    print('reza')
    print('mohammad')
```

```
n = cube()
print(n)
```

```
x: 3
ok
64
```

* به ورودی تابع پارامتر(**parameter**) گویند و زمانی که آن‌ها را صدا می‌زنیم و ورودی می‌دهیم به آن آرگومان (**argument**) گویند.

* لزومی ندارد اسم پارامتر و آرگومان یکسان باشد.

```
def cube(x):
    return x ** 3
```

```
y = int(input('y: '))
n = cube(y)
print(n)
```

* باید سعی شود تابع کوچک باشد مثلاً حداکثر ۲۰ خط باشد و هر تابع یک وظیفه‌ی مشخص داشته باشد.

* بهتر است تعداد پارامترها کم باشد.

* بهتر است تابع بی هدف نوشته نشود و مفید واقع شود.

* زمانی که نمی‌خواهیم بدنہ تابع را بنویسیم، برای این‌که برنامه اجرا شود و خطأ ندهد و ما بتوانیم بعداً بیاییم و بدنہ‌ی آن را بنویسیم، از دستور **pass** استفاده می‌کنیم.

* از **pass** می‌شود در سایر دستورات نیز استفاده کرد.

```
def cube(x):
    pass
```

```
if True:
    pass
```

درس ۳: حل مثال برای مبحث تابع

مثال: یک تابع بنویسیم که دو عدد بگیرد و مشخص کند عدد دوم چند بار در عدد اول تکرار شده است.

```
def repeat():
    x = int(input('x: '))
    y = int(input('y: '))
    count = 0
    while x > 0:
        if x % 10 == y:
            count += 1
        x //= 10
    return count

print(repeat())
```

مثال: حاصل عبارت $1! + 2! + 3! + \dots + n!$ را با استفاده از تابع به دست آورید.

```
def fact(x):
    f = 1
    for i in range(1, x + 1):
        f *= i
    return f

def plus(x):
    p = 0
    for i in range(1, x + 1):
        p += fact(i)
    return p

n = 7
print(plus(n)) # -> 5913
```

مثال: تابعی بنویسید که سه عدد بگیرد و بزرگترینشان را نشان دهد.

```
def maximum():
    x = float(input('x: '))
    y = float(input('y: '))
    z = float(input('z: '))
    print(max(x, y, z))

maximum()
```

درس ۴: سینتکس نوشتن آرگومان‌ها

* سینتکس **نرمال** که در آن به ازای هر پارامتر باید یک آرگومان تعریف کنیم و ترتیب نیز مهم است.

```
def max3(a, b, c):  
    print(max(a, b, c))  
  
max3(2, 4, 3) # -> 4
```

* سینتکس **نام – مقدار** (**name – value**) که در آن مشخص می‌کنیم هر آرگومان به چه پارامتری اختصاص دارد و چون مشخص می‌کنیم هر پارامتری چه مقداری دارد و اسم می‌دهیم پس ترتیب مهم نیست.

```
def func(a, b, c):  
    print('a: ', a)  
    print('b: ', b)  
    print('c: ', c)  
  
func(b=3, a=4, c=1)
```

a:	4
b:	3
c:	1

* سینتکس **ترکیبی نرمال و نام – مقدار** که در این سینتکس چون هردو آرگومان به کار رفته‌اند، ابتدا باید آرگومان‌های **نرمال** که ترتیب در آن‌ها مهم است نوشته شوند و سپس آرگومان‌های **نام – مقدار** نوشته شوند.

```
def func(a, b, c):  
    print('a: ', a)  
    print('b: ', b)  
    print('c: ', c)  
  
func(1, c=9, b=5)
```

a:	1
b:	5
c:	9

* سینتکس **set, str, tuple, list, ...)*iterable** که باید قبل آن ***** بگذاریم.

```
def func(a, b, c):  
    print('a: ', a, '\nb: ', b, '\nc: ', c)  
  
x = [7, 9, 10]  
func(*x)
```

a:	7
b:	9
c:	10

* سینتکس **دیکشنری** که باید از **دو ستاره (**)** قبل از آرگومان استفاده کنیم و ترتیب مهم نیست چون مقدار هر کلید مشخص است.

```
def func(a, b, c):  
    print('a: ', a, '\nb: ', b, '\nc: ', c)  
  
d = {'c': 4, 'a': 5, 'b': 9}  
func(**d)
```

a:	5
b:	9
c:	4

درس ۵: سینتکس نوشتن پارامترها

* سینتکس **نرمال** که در آن به ازای هر آرگومان باید یک پارامتر نوشته می‌شود و ترتیب نیز مهم است.

```
def func(a, b, c):  
    print('a: ', a, '\nb: ', b, '\nc: ', c)  
  
func(1, 2, 3)
```

a:	1
b:	2
c:	3

* سینتکس **default - value** که در این سینتکس به پارامتر یک مقدار پیش‌فرض می‌دهیم که اگر تعداد آرگومان‌ها از پارامترها کم‌تر بود، پارامترهای اضافی آن مقدار پیش‌فرض را به خود گیرند و مشکلی ایجاد نشود.

```
def func(a=2, b=6, c=3):  
    print('a: ', a, '\nb: ', b, '\nc: ', c)  
  
func(8)
```

a:	8
b:	6
c:	3

* سینتکس ترکیبی **نرمال و default - value** که از هردو سینتکس همزمان استفاده می‌کنیم ولی ابتدا باید پارامترهای نرمال را بنویسیم و سپس **default - value** را بنویسیم.

```
def func(a, b=6, c=3):  
    print('a: ', a, '\nb: ', b, '\nc: ', c)  
  
func(8)
```

a:	8
b:	6
c:	3

* سینتکس ***name** که در آن اگر قبل از پارامتر ستاره ***** بگذاریم، آرگومان‌های اضافی را به شکل **تاپل** به خود می‌گیرد.

```
def func(a, b, *c):  
    print('a: ', a, '\nb: ', b, '\nc: ', c)  
  
func(6, 9, 10, 5, 8)
```

a:	6
b:	9
c:	(10, 5, 8)

* بعد از پارامتر ستاره‌دار به شرطی می‌توانیم پارامتر دیگری بنویسیم که در آرگومان‌های آن به صورت **نام - مقدار** بنویسیم.

* پارامتر ستاره‌دار قابل ترکیب با سینتکس‌های دیگر است.

```
def func(a, b, *c, d):  
    print('a: ', a, '\nb: ', b, '\nc: ', c, '\nd: ', d)  
  
func(6, 9, 10, 5, 8, d=9)
```

a:	6
b:	9
c:	(10, 5, 8)
d:	9

* پارامتر ستاره‌دار نمی‌تواند آرگومان نام – مقدار به خود گیرد.

```
def func(a, b, *c):
    print('a: ', a, '\nb: ', b, '\nc: ', c, '\nd: ', d)

func(6, 9, c=2)
```

```
Traceback (most recent call last):
  File "F:\python\pythonProject\test.py", line 5, in <module>
    func(6, 9, c=2)
TypeError: func() got an unexpected keyword argument 'c'
```

* سینتکس **دیکشنری** که باید از دو ستاره (******) قبل از پارامتر استفاده کنیم.

* سینتکس دیکشنری قابل ترکیب با سینتکس‌های دیگر است.

```
def func(**a):
    print('a: ', a)

func(a=2, b=5, c=9)
```

```
a: {'a': 2, 'b': 5, 'c': 9}
```

درس ۶: پارامترهای نشانگر

پارامتر ستاره (*****): زمانی که ستاره گذاشتیم، پارامترهای بعد از آن آرگومان‌هایشان باید به صورت **نام – مقدار** نوشته شود.

```
def func(a, *, b, c):
    print(a, b, c)

func(1, b=2, c=3)
```

پارامتر اسلش (/): زمانی که اسلش گذاشتیم، پارامترهای قبل از آن آرگومان‌هایشان باید به صورت **نرمال و براساس موقعیتشان** نوشته شوند.

```
def func(a, b, /, c):
    print(a, b, c)

func(1, 2, c=3)
```

```
def func(a, b, /, c, d, *, e, f):
    print(a, b, c, d, e, f)

func(1, 2, 3, d=4, e=5, f=6)
```

(DocString) درس ۷: رشته مستند سازی

- * برای توضیح عملکرد یک تابع از داک استرینگ استفاده می کنیم.
- * داک استرینگ را در اولین خط بعد از تابع و قبل از دستورات می نویسیم و با سه دابل کتیشن (""") شروع و به پایان می رسانیم.
- * بهتر است حرف اول متن داک استرینگ با **حرف بزرگ انگلیسی** شروع شود و در پایان با **نقطه** به پایان رسد.
- * با استفاده از اtribیوت **__doc__** می توانیم به استرینگ تابع دسترسی پیدا کنیم.

```
def max3(x, y, z):  
    """Receives three number and returns the largest."""  
    return max(x, y, z)
```

```
print(max3.__doc__)
```

```
Receives three number and returns the largest.
```

- * بهتر است در توابع از داک استرینگ استفاده کنیم.
- * بهتر است در داک استرینگ به جای کتیشن (') از دابل کتیشن (") استفاده کنیم.
- * از تابع **help** نیز می توانیم برای دسترسی به استرینگ تابع کمک بگیریم.

```
def max3(x, y, z):  
    """Receives three number and returns the largest."""  
    return max(x, y, z)
```

```
print(help(max3))
```

```
Help on function max3 in module __main__:  
  
max3(x, y, z)  
    Receives three number and returns the largest.  
  
None
```

- * از داک استرینگ می شود در کلاس ها نیز استفاده کرد.
- * کامنت برای برنامه نویس هاست و با استفاده از آن می توان توضیحات زیادی درباره تابع، عملکرد، جزئیات و ... نوشت و مفسر آن را نادیده می گیرد اما به وسیله ای داک استرینگ فقط وظیفه ای تابع را می نویسیم.
- * داک استرینگ را می توان **تک خطی** (که در آن به طور خلاصه عملکرد تابع را می نویسیم) و یا **چندخطی** نوشت.

* اگر داک استرینگ را **چندخطی** بنویسیم در خط اول خلاصه‌ی کارکرد تابع را می‌نویسیم، سپس با یک خط فاصله پارامترها را می‌نویسیم و با یک خط فاصله خروجی را می‌نویسیم.

```
def max3(x, y, z):
    """Receives three number and returns the largest

Parameters:
    x (int): A decimal integer
    y (int): A decimal integer
    z (int): A decimal integer

Returns:
    max_int (int): largest of three numbers
    """
    return max(x, y, z)
```

درس ۸: یادداشت تابع (function annotation)

* زمانی که بخواهیم قبل از اجرا، اشتباه را مشخص کنیم و نوع پارامتر را تعیین کنیم از یادداشت استفاده می کنیم به این صورت که پس از پارامتر دو نقطه (:) می گذاریم و نوع آن را می نویسیم و وقتی اشتباه بنویسیم در محیط پایچارم هایلایت می شود.

```
def func(x: int, y: int, z: int):  
    print(x, y, z)
```

```
func(1, 2, '3')
```

* می شود برای خروجی هم نوع تعیین کرد به این صورت که پس از پارامتر ها > - بگذاریم و نوع خروجی را مشخص کنیم.

```
def func(x, y, z: int = 3) -> int:  
    return x + y + z
```

* برای مشخص کردن یادداشت و نوع ورودی و خروجی می توانیم از اtribut __annotations__ استفاده کنیم.

```
def func(x, y, z: int = 3) -> int:  
    return x + y + z
```

```
print(func.__annotations__) # -> {'z': <class 'int'>, 'return': <class 'int'>}
```

درس ۹: مفهوم تابع first-class

* به اشیایی first-class می‌گویند که محدودیت نداشته باشند.

x = 3

* تابع first-class نیز به تابعی می‌گویند که محدودیت نداشته باشد و می‌توان به متغیر دیگر اختصاص داد و یا به یک تابع دیگر به عنوان آرگومان ارسال کرد.

* همه‌ی توابع در پایتون first-class هستند.

* همه‌ی کلاس‌ها در پایتون first-class هستند.

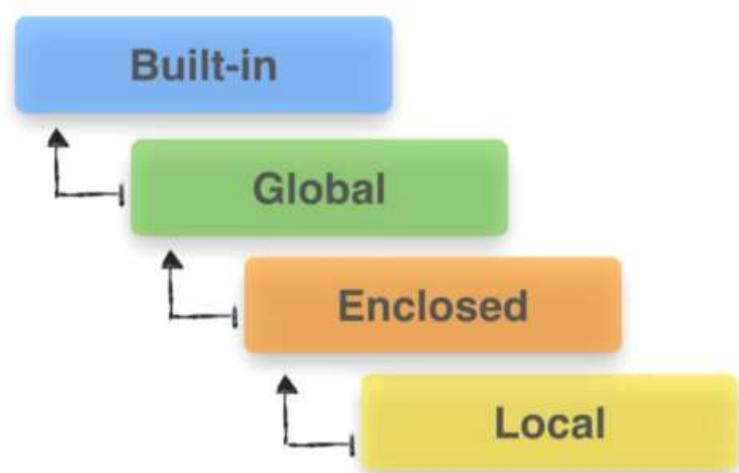
:first-class ویژگی‌های تابع

- می‌تواند به صورت پویا ایجاد یا نابود شود. (مانند شرط‌ها که اگر شرط برقرار باشد ایجاد و گرنه ایجاد نمی‌شود.)
- می‌تواند به متغیر اختصاص داشته باشد.
- می‌شود به عنوان آرگومان به یک تابع دیگر ارسال گردد.
- می‌شود به عنوان نتیجه یک تابع برگردانده شود.
- می‌تواند متده داشته باشد.

درس ۱۰: فضای نام و حوزه (بخش اول)

- * به مجموعه اسم‌هایی که در برنامه نویسی وجود دارد **فضای نام (name-space)** می‌گوییم.
- * در پایتون هر تابع دارای فضای نام جدا می‌باشد. مثلاً در دوتابع ممکن است اسم X وجود داشته باشد اما متفاوتند.
- * در **حوزه (scope)** چند فضای نام وجود دارد که مشخص می‌کند منظور کدام اسم است.

فضای نام‌ها در پایتون:



- * فضای نام **Built-in** بزرگ‌ترین فضای نام است و اسم‌هایی که در این فضای نام تعریف شده اند، در همه مژول‌ها شناخته می‌شوند.

```
test.py
1 len('ali')

test2.py
1 len('ali')
```

- * با استفاده از تابع **dir** می‌توانیم مشخص کنیم چه اسم‌هایی در فضای نام **Built-in** تعریف شده است.

```
print(dir(__builtins__))
```

```
['ArithmetError', 'AssertionError', 'AttributeError', 'BaseException', 'BaseExceptionGroup', 'Blo
```

- * فضای نام **Built-in** را خود پایتون تعریف می‌کند.

- * فضای نام **Global** که از فضای نام **Built-in** کوچک‌تر است و در هریک از مژول‌ها شناخته می‌شود و هر کدام از مژول‌ها دارای فضای نام **Global** جداگانه هستند که باهم تداخلی ندارند.

```
test.py
1 x = 2

test2.py
1 x = 5
```

- * با دستور **globals** می‌توانیم مشخص کنیم در مژول‌ها چه اسم‌هایی تعریف شده است.

```
x = 2
print(globals())
```

```
ns' (built-in)>, '__file__': 'F:\\python\\pythonProject\\test.py', '__cached__': None, 'x': 2}
```

* فضای نام **Global** برخلاف فضای نام **Built-in** به پایتون تعلق ندارد و مربوط به ماژول است و می‌توانیم تغییرش دهیم.

* فضای نام **Local** که کوچک‌ترین فضای نام است و متعلق به توابع و کلاس‌هاست و هر کدام فضای نام مستقل دارند که به توابع دیگر ربطی ندارند.

```
def ab():
    x = 10
    y = 5
```

```
def bc():
    x = 9
    y = 6
```

* با دستور **locals** می‌توانیم مشخص کنیم چه اسامی‌ای در فضای نام **Local** تعریف شده‌اند، این دستور را باید داخل تابع مربوطه بنویسیم.

```
def ab():
    x = 10
    y = 5
    print(locals())
```



```
ab() # -> {'x': 10, 'y': 5}
```

* فضای نام **Enclosed** که کوچک‌تر از **Global** و بزرگ‌تر از **Local** می‌باشد و زمانی ایجاد می‌شود که تابع **تودرتو** داشته باشیم، در تابع داخلی فضای نام **Local** می‌باشد و فضای نام تابع اصلی **Enclosed** می‌باشد.

```
def ab():
    x = 10 # Enclosed   فضای نام
    y = 5  # Enclosed   فضای نام

    def ab2():
        x = 5 # Local   فضای نام
```

* زمانی که بخواهیم از فضای نام یک ماژول دیگر استفاده کنیم از **import** استفاده می‌کنیم.

```
import math

print(math.pi) # -> 3.141592653589793
```

درس ۱۱: فضای نام و حوزه (بخش دوم)

- * برای تشخیص این که کدام فضای نام مدنظر می‌باشد از قوانین **حوزه** (scope) استفاده می‌کنیم.
- * ساختار حوزه و فضای نام در پایتون شبیه به هم است.

Scopes in Python



- * همهی پروژه‌ها و مازول‌ها در حوزه‌ی **Built-in** قرار دارند.
- * هر مازول دارای حوزه می‌باشد که به آن **Global** می‌گوییم.
- * هر تابع دارای حوزه‌ی **local** می‌باشد.
- * در توابع تو در تو، تابع داخلی حوزه‌ی **local** و تابع اصلی حوزه‌ی **Enclosing** می‌باشد.
- * حوزه‌ی **local** به فضای نام همهی حوزه‌ها دسترسی دارد ولی ابتدا فضای نام حوزه‌ی خودش را برای اسم مورد نظر جستجو می‌کند و اگر یافت نشد حوزه‌ی **Enclosing** را و سپس **Global** و در نهایت **Built-in** را جستجو می‌کند و اگر یافت نشد خطای دهد، اما بر عکس این عمل صادق نیست، یعنی حوزه‌ی بزرگ‌تر فضای نام حوزه‌های کوچک‌تر را بررسی نمی‌کند و اگر اسم در آن حوزه و یا حوزه‌های بالاتر یافت نشد خطای دهد.

```
x = 0

def ab():
    x = 5

    def bc():
        x = 10
        print(x)

    bc()

ab() # -> 10
print(x) # -> 0
```

* اگر بخواهیم در حوزه‌های داخلی بتوانیم به اسم حوزه‌های بزرگ‌تر دسترسی داشته باشیم و تغییرش دهیم از دستور **global** برای دستیابی به حوزه‌ی **nonlocal** و از دستور **Global** برای دستیابی به حوزه‌ی **Enclosing** استفاده می‌کنیم.

```
x = 0

def ab():
    x = 2

    def bc():
        global x
        x += 1
        print(x)

    bc()

ab() # -> 1
y = 0

def de():
    y = 2

    def ef():
        nonlocal y
        y += 1
        print(y)

    ef()

de() # -> 3
```

درس ۱۲: ارسال با مقدار و ارسال با ارجاع

* هنگامی که با مقدار ارسال می‌کنیم تغییرات در بیرون از تابع روی متغیر اعمال نمی‌شوند.

```
def func(x):
    x += 1
    x *= 5
    print(x)

a = 1
func(a) # -> 10
print(a) # -> 1
```

* در پایتون همه چیز ارسال با ارجاع است و اگر در مثال بالا تغییرات اعمال نشد علتیش به تغییر پذیر یا تغییر ناپذیر بودن مربوط است و اعداد، رشته، بولین، تاپل و ... تغییر ناپذیرند.

* در اشیاء تغییر پذیر، تغییرات اعمال می‌شوند.

```
def func(x):
    x += [1, 3, 5]
    x[0] = 0
    print(x)

a = [1, 2, 3]
func(a) # -> [0, 2, 3, 1, 3, 5]
print(a) # -> [0, 2, 3, 1, 3, 5]
```

```
def func(x):
    x['c'] = 3
    print(x)

a = {'a': 1, 'b': 2}
func(a) # -> {'a': 1, 'b': 2, 'c': 3}
print(a) # -> {'a': 1, 'b': 2, 'c': 3}
```

* اگر بخواهیم تغییرات در اشیاء تغییرپذیر اعمال نشوند، می‌توانیم از کپی آن شی استفاده کنیم.

```
import copy

def func(x):
    x['c'] = 3
    print(x)

a = {'a': 1, 'b': 2}
func(copy.deepcopy(a)) # -> {'a': 1, 'b': 2, 'c': 3}
print(a) # -> {'a': 1, 'b': 2}
```

* اگر بخواهیم تغییرات در اشیاء تغییرناپذیر اعمال شوند، می‌توانیم از روش زیر استفاده کنیم.

```
def func(x):
    x += 1
    print(x)
    return x

a = 4
a = func(a) # -> 5
print(a) # -> 5
```

درس ۱۳ و ۱۴: تکالیف مبحث تابع و حل آنها

۱. تابعی بنویسید که کار تابع داخلی `len` را انجام دهد.

```
def len2(it):
    counter = 0
    for _ in it:
        counter += 1
    return counter

b = [3, 2, 1]
print(len2(b)) # -> 3
```

۲. تابعی بنویسید که کار تابع داخلی `max` یا `min` را انجام دهد.

```
def max2(*args):
    m = args[0]
    for i in args:
        if i > m:
            m = i
    return m

print(max2(4, 2, 11, 23, 1)) # -> 23
```

۳. تابعی بنویسید که کار تابع داخلی `sum` را انجام دهد.

```
def sum2(it):
    a = 0
    for i in it:
        a += i
    return a

print(sum2((2, 1, 4, 1))) # -> 8
```

۴. تابعی بنویسید که یک عدد به عنوان ورودی گرفته و تشخیص دهد عدد مربع است یا خیر. (مثلا عدد ۸۱ مربع است، چون از حاصل ضرب 9×9 به دست می آید).

```
def square(x):
    for i in range(1, x):
        if i ** 2 == x:
            print(f'{x} is a square number: {i} * {i} = {x}')
            break
    else:
        print(f'{x} is not a square number.')

square(34)
```

۵. تابعی بنویسید که قیمت کالا و درصد تخفیف را گرفته و قیمت پس از تخفیف را محاسبه کند.

```
def discount_price(price, percent):  
    price_percent = 100 - percent  
    final_price = (price_percent * price) / 100  
    return final_price  
  
print(discount_price(20000, 10)) # -> 18000.0
```

۶. تابعی بنویسید که یک کاراکتر را خوانده و مشخص کند کاراکتر یک رقم، حرف بزرگ، حرف کوچک و یا سایر نمادها است.

```
def character_type(c):  
    if 97 <= ord(c) <= 122:  
        print(f'{c}' " is a small English letter.")  
    elif 65 <= ord(c) <= 90:  
        print(f'{c}' " is a capital English letter.")  
    elif 48 <= ord(c) <= 57:  
        print(f'{c}' " is a number.")  
    else:  
        print(f'{c}' " is a different symbol.")
```

درس ۱۵: لامبدا و کاربرد آن (sorted, map, filter, reduce)

- * برای توابع یک خطی و کم کاربرد از تابع **lambda** استفاده می‌کنیم.
- * توابع لامبدا اسم ندارند، کوچک و یک خطی‌اند و هر چقدر که لازم باشد می‌توانند ورودی بگیرند و دستور **return** برای آن‌ها نوشته نمی‌شود.
- * برای نوشتمن این توابع از دستور **lambda** استفاده می‌کنیم. سپس ورودی را می‌نویسیم و با **کاما (,** آن‌ها را جدا می‌کنیم، سپس **دو نقطه (:)** می‌گذاریم و در جلوی آن بدنہ را می‌نویسیم.

```
def func(x):  
    return x ** 2
```

تابع بالا با لامبدا اینگونه نوشته می‌شود .
lambda x: x ** 2

- * برای ذخیره شدن این تابع می‌توانیم آن را در یک متغیر قرار بدهیم که این کار توصیه نمی‌شود.

```
a = lambda x: x ** 2
```

- * در لامبدا نمی‌توانیم دستورات چند خطی مانند **while** و ... بنویسیم.

- * تابع **map** یک تابع و یک لیست می‌گیرد و وظیفه‌ی این تابع این است که یک به یک عناصر لیست را به تابع می‌فرستد، اصلاح می‌کند و برای استفاده باید آن را به لیست تبدیل کنیم.

```
def func(x):  
    return x ** 2  
  
my_list = [1, 2, 3, 4]  
new = map(func, my_list)  
  
print(my_list) # -> [1, 2, 3, 4]  
print(list(new)) # -> [1, 4, 9, 16]
```

- * به جای این‌که در بیرون از **map**، تابع تعریف کنیم، می‌توانیم از لامبدا استفاده کنیم.

```
my_list = [1, 2, 3, 4]  
new = map(lambda x: x ** 2, my_list)  
  
print(my_list) # -> [1, 2, 3, 4]  
print(list(new)) # -> [1, 4, 9, 16]
```

* تابع **filter** یک تابع و یک لیست می‌گیرد و عناصر لیست را به تابع می‌فرستد و براساس **True** یا **False** نتایج را برمی‌گرداند و فیلتر می‌کند، برای استفاده از این تابع باید آن را به لیست تبدیل کنیم.

```
def func(x):
    return x > 5

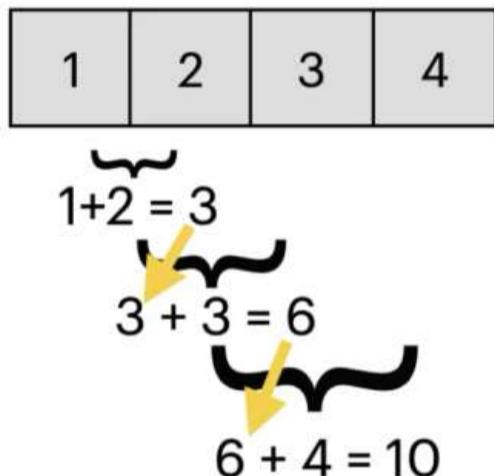
li = [1, 9, 10, 5, 0, 4, 2, 3, 4, 5, 6, 7]
new = filter(func, li)
print(li) # -> [1, 9, 10, 5, 0, 4, 2, 3, 4, 5, 6, 7]
print(list(new)) # -> [9, 10, 6, 7]
```

* به جای این که در بیرون از **filter**، تابع تعریف کنیم، می‌توانیم از لامبدا استفاده کنیم.

```
li = [1, 9, 10, 5, 0, 4, 2, 3, 4, 5, 6, 7]
new = filter(lambda x: x > 5, li)
print(li) # -> [1, 9, 10, 5, 0, 4, 2, 3, 4, 5, 6, 7]
print(list(new)) # -> [9, 10, 6, 7]
```

* تابع **reduce** یک تابع و یک لیست می‌گیرد و عملیات تابع را روی دو عنصر اول لیست انجام می‌دهد و سپس عملیات را روی حاصل دو عنصر اول و عنصر سوم انجام مسدهد و به همین ترتیب الی آخر، این دستور باید از ماژول **functools** وارد شود.

```
reduce(lambda x, y: x + y, [1, 2, 3, 4])
```



* از تابع **sorted** برای مرتب سازی استفاده می‌کنیم و اگر در این تابع از **key** استفاده کنیم، مرتب سازی را براساس تابعی که می‌گیرد انجام می‌دهد.

```
li = [4, 1, 2, 3, 4, 8, 7, 1, 25, 9]
s = sorted(li, key=lambda x: x % 3)
print(s) # -> [3, 9, 4, 1, 4, 7, 1, 25, 2, 8]
```

درس ۱۶ و ۱۷: تکالیف مبحث لامبدا و حل آنها

۱. برنامه‌ای بنویسید که با استفاده از لامبда، اعداد زوج و فرد را در یک لیست از اعداد صحیح بشمارد.

```
li = [4, 1, 2, 3, 4, 8, 7, 1, 25, 9]
odd = len(list(filter(lambda x: x % 2 != 0, li)))
even = len(list(filter(lambda x: x % 2 == 0, li)))

print(odd) # -> 6
print(even) # -> 4
```

۲. لیستی از تاپل‌ها به شکل `[('Reza', 65), ('Ali', 93), ...]` داریم. با استفاده از لامبدا برنامه‌ای بنویسید که این لیست را بر اساس اعداد موجود در تاپل مرتب کند.

```
lst = [('sahel', 25), ('ali', 93), ('reza', 65), ('ali', 87)]

lst.sort(key=lambda x: x[1], reverse=True)

print(lst) # -> [('ali', 93), ('ali', 87), ('reza', 65), ('sahel', 25)]
```

۳. لیستی از دیکشنری‌ها به شکل `[{'weight': 65, 'apple': 'red'}, ...]` داریم. با استفاده از لامبدا برنامه‌ای بنویسید که این لیست را بر اساس رنگ موجود در دیکشنری مرتب کند.

```
lst = [{'name': 'apple', 'weight': 50, 'color': 'red'},
        {'name': 'banana', 'weight': 60, 'color': 'yellow'},
        {'name': 'coconut', 'weight': 20, 'color': 'brown'},
        {'name': 'orange', 'weight': 30, 'color': 'orange'}]

lst.sort(key=lambda x: x['color'])

print(lst)
```

۴. برنامه‌ای بنویسید تا لیستی از اعداد صحیح را با استفاده از لامبدا به زوجها و فردها فیلتر کند.

```
lst = [1, 3, 4, 5, 7, 8, 9]
odd = list(filter(lambda x: x % 2 != 0, lst))
print('odd list: ', odd) # -> odd list: [1, 3, 5, 7, 9]
even = list(filter(lambda x: x % 2 == 0, lst))
print('even list: ', even) # -> even list: [4, 8]
```

۵. با استفاده از لامبدا یک برنامه‌ای بنویسید تا هر عدد را در لیستی از اعداد صحیح مربع و مکعب کند.

```
lst = [1, 2, 3, 4, 5, 7, 8, 9]
square = list(map(lambda x: x ** 2, lst))
cube = list(map(lambda x: x ** 3, lst))

print(square) # -> [1, 4, 9, 16, 25, 49, 64, 81]
print(cube) # -> [1, 8, 27, 64, 125, 343, 512, 729]
```

۶. با استفاده از لامبدا برنامه‌ای بنویسید تا بفهمید که آیا یک رشته داده شده با یک کاراکتر مشخص شروع می‌شود یا خیر.

```
x = 'this is a string'

starts_with = lambda s: True if x.startswith('t') else False

print(starts_with(x)) # -> True
```

۷. با استفاده از لامبدا برنامه‌ای بنویسید تا بررسی کند که آیا یک رشته داده شده عددی است یا خیر. (توجه داشته باشید که می‌خواهیم رشته‌های اعشاری همچون ۴.۵ هم تشخیص داده شوند).

```
n = 4.5

is_num = lambda s: s.replace('.', '', 1).isdigit()

print(is_num('af'))
```

درس ۱۸: ایتراتور (iterator)

* به معنی تکرار کردن است.

* به معنی تکرار است که رفتار تکرار را نشان می‌دهد (عمل تکرار کردن).

* در پایتون `for` و `while` رفتار تکرار را انجام می‌دهند.

* به معنی قابل تکرار (تکرار پذیر) است.

* همه شی‌هایی که بتوانیم روی آن‌ها عمل تکرار را انجام دهیم `iterable` هستند.

* همه شی‌هایی که بتوانیم آن‌ها را به `iterator` تبدیل کنیم `iterable` هستند.

* لیست، تاپل، رشته، دیکشنری، ست، رنج، زیپ و فایل `iterable` هستند.

```
for i in [1, 2, 3]:  
    print(i)
```

* برای این‌که بفهمیم یک شی `iterator` هست یا خیر از تابع `dir` استفاده می‌کنیم و اگر در نتیجه `__iter__` را پیدا کنیم یعنی شی `iterable` می‌باشد.

```
print('__iter__' in dir([1, 2, 3])) # -> True
```

* اعداد و بولین `iterable` نیستند.

* با دستور `next` می‌توانیم عناصر یک `iterator` را به ترتیب به دست آوریم.

* برای این‌که بتوانیم از عناصر یک `iterable` استفاده کنیم باید با استفاده از دستور `iter` آن را به `iterator` تبدیل کنیم.

```
i = [1, 2, 3]  
i = iter(i)  
while True:  
    print(next(i))
```

```
1  
2  
3  
Traceback (most recent call last):  
  File "F:\python\pythonProject\test.py", line 4, in <module>  
    print(next(i))  
      ^^^^^^  
StopIteration
```

* وقتی عناصر پایان می‌یابند با خطأ مواجه می‌شویم که برای حل این مشکل می‌توانیم از استثنای (در فصل‌های بعد آشنا می‌شویم) استفاده کنیم.

```
i = [1, 2, 3]  
i = iter(i)  
while True:  
    try:  
        print(next(i))  
    except StopIteration:  
        break
```

```
1  
2  
3
```

* حلقه `for` خودش خودکار `iterable` را به `iterator` تبدیل می‌کند و `next` ار چاپ می‌کند.

```
lst = [1, 2, 3]
for i in lst:
    print(i)
```

1
2
3

* می‌توانیم `iterator` هایی ایجاد کنیم که به پایان نمی‌رسند و یا قبل ایجاد شده‌اند.

```
from itertools import count

c = count()
print(next(c)) # -> 0
print(next(c)) # -> 1
print(next(c)) # -> 2
```

* از نوع `lazy` (تبل) می‌باشد و عناصر را زمانی چاپ می‌کند که به آن نیاز داشته باشیم و هر بار که بعدی را می‌خواهیم همان لحظه می‌سازد و می‌توانیم مشخص کنیم شمارش را از چند آغاز کند.

```
test.py x
1 from itertools import count
2
3 c = count(10)
4 for i in c:
5     print(i)
6     if i == 20:
7         break
```

The screenshot shows a PyCharm interface with a code editor containing the above Python script and a run tool window on the right. The run tool window has a title bar 'Run' and 'test'. It lists several runs with their results: 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, and 20. This demonstrates that the `count` iterator generates values lazily, starting at 10 and continuing up to 20.

درس ۱۹: دکوراتور (decorator) – بخش اول

* برای استفاده از **decorator** باید به توابع **تودرتو** و ارسال تابع به عنوان آرگومان (فصل ۶ – درس ۹ – مفهوم تابع **first-class**) باید به توابع **تودرتو** و ارسال تابع به عنوان آرگومان (فصل ۶ – درس ۹ – مفهوم تابع **first-class**) مسلط باشیم.

* یک تابع دیگر را به نوعی دچار تغییر می‌کند و عملکردهایش را بهبود می‌بخشد.

* در تابع دکوراتور از یک تابع داخلی استفاده می‌کنیم و به جای صدا زدن تابع داخلی آن را در تابع با استفاده از **return** بازگشت می‌دهیم.

* دکوراتور یک تابع به عنوان ورودی می‌گیرد.

```
def dec(func):
    def inner():
        print('*' * 20)
        func()
        print('*' * 20)

    return inner

def f():
    print('reza')

d = dec(f)
d()
```

```
*****  
reza  
*****
```

* بهتر است برای این‌که دکوراتور روی تابع اعمال شود به جای روش بالا قبل از تابع **@** بگذاریم و سپس **اسم دکوراتور** را بنویسیم.

```
def dec(func):
    def inner():
        print('*' * 20)
        func()
        print('*' * 20)

    return inner

@dec
def f():
    print('reza')

f()
```

```
*****  
reza  
*****
```

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

f(5, 0)
```

```
Traceback (most recent call last):
  File "F:\python\pythonProject\test.py", line 5, in <module>
    f(5, 0)
  File "F:\python\pythonProject\test.py", line 2, in f
    print(x / y)
           ~~^~~
ZeroDivisionError: division by zero
```

* در مثال بالا با خطا مواجه می‌شویم، چون تقسیم عدد بر صفر ممکن نیست، برای حل این مشکل می‌توانیم از دکوراتور استفاده کنیم.

```
def dec(func):
    def inner(x, y):
        if y == 0:
            print('it is not possible.')
        else:
            func(x, y)

    return inner

@dec
def f(x, y):
    print(x / y)

f(5, 0) # -> it is not possible.
```

درس ۲۰: دکوراتور (decorator) – بخش دوم

- * به دکوراتور **wrapper** (پوشاننده، پوشش دهنده) نیز گفته می‌شود.
- * در بعضی از دکوراتورها تعداد ورودی معلوم نیست و راه حل این است که پارامتر را به صورت **تاپل** و **دیکشنری** بگیریم.

```
def dec(func):  
    def inner(**x, **y):  
        print('*' * 20)  
        return func(**x, **y)  
        print('*' * 20)  
  
    return inner  
  
  
@dec  
def f(x, y, z):  
    return x * y * z['A']  
  
print(f(5, 9, {'A': 2}))
```

```
*****  
90
```

- * بعد از **return** دستورات نادیده گرفته می‌شوند و در مواقعي که بعد از تابع باید **print** بیاوریم، بهتر است به جای **return** از **print** استفاده کنیم.

```
def dec(func):  
    def inner(**x, **y):  
        print('*' * 20)  
        print(func(**x, **y))  
        print('*' * 20)  
  
    return inner  
  
  
@dec  
def msg():  
    return 'this is a text'  
  
msg()
```

```
*****  
this is a text  
*****
```

- * اگر بعد از `return` قرار نیست `print` شود، بهتر است تابع را در یک متغیر دیگر قرار دهیم و سپس آن را `return` کنیم.
 - * روی یک تابع می‌شود چند دکوراتور اعمال کرد.

```
def star(func):
    def inner(*x, **y):
        print('*' * 20)
        f = func(*x, **y)
        return f

    return inner

def plus(func):
    def inner(*x, **y):
        print('+'.join(['*' * 20] * len(x)))
        f = func(*x, **y)
        return f

    return inner

@plus
@star
def msg():
    return 'This is a text'

print(msg())
```

```
+++++  
*****  
This is a text
```

- * روش متداول اعمال چند دکوراتور برای یک تابع به صورت زیر است.

```
@plus  
@star  
def msg():  
    return 'This is a text'
```

- * اما به روش زیر نیز می‌توان نوشت.

```
printer = plus(star(msg))
```

- * گاهی ممکن است دکوراتورها ورودی داشته باشند، در آن صورت باید دکوراتور را داخل یک تابع دیگر قرار داد.

```
def star(a):
    def inner(func):
        def inner2(*x, **y):
            print('*' * a)
            func(*x, **y)
        return inner2
    return inner

@star(5)
def msg(name):
    print('I am ' + name)

msg('Reza')
```

I am Reza

* اگر برای تابع و دکوراتور داک استرینگ بنویسیم و بخواهیم توضیحات و اسم تابع را نشان دهد، توضیحات و اسم تابع داخلی دکوراتور را نشان می‌دهد، چون تابع کاملاً باز نویسی شده است.

```
def star(func):
    def inner(**x, **y):
        """this is a decorator."""
        print('*' * 20)
        func(*x, **y)

    return inner

@star
def msg(a):
    """Print a message"""
    print(a)

print(msg.__doc__) # -> this is a decorator.
print(msg.__name__) # -> inner
```

* برای حل این مشکل می‌توانیم از `wraps` که از `functools` وارد کردیم استفاده کنیم.

```
import functools

def star(func):
    @functools.wraps(func)
    def inner(**x, **y):
        """this is a decorator."""
        print('*' * 20)
        func(*x, **y)

    return inner

@star
def msg(a):
    """Print a message"""
    print(a)

print(msg.__doc__) # -> Print a message
print(msg.__name__) # -> msg
```

فرمول و روش کلی ساخت دکوراتور (decorator):

```
import functools

def decorator(func):
    @functools.wraps(func)
    def wrapper_decorator(*args, **kwargs):
        # Do something before
        value = func(*args, **kwargs)
        # Do something after
        return value

    return wrapper_decorator
```

درس ۲۱: حل مثال برای دکوراتور

مثال: دو تابع بنویسیم که یکی نام کاربری (username) و رمز (password) را از دیکشنری چاپ کند و دیگر رمز را تغییر دهد و یک بلک لیست هم داریم و یک تابع دکوراتور دیگر بنویسیم که اگر نام کاربری در بلک لیست بود، تغییرات اعمال نشود.

```
from functools import wraps

passwords = {'ali': 2432, 'reza': 8237, 'neda': 3546}
black_list = {'neda'}

def ban(func):
    @wraps(func)
    def inner(*args, **kwargs):
        if args[0] in black_list:
            print('this user is blocked!')
        else:
            func(*args, **kwargs)

    return inner

@ban
def print_passwords(username):
    print(f'{username} password is: {passwords[username]}')

@ban
def change_password(username, new_password):
    passwords[username] = new_password
    print(f'{username} new password is: {new_password}'')
```

مثال: یک تابع دکوراتور بنویسیم که مدت زمان یک تابع را حساب کند.

```
from functools import wraps
from time import perf_counter

def time_calculator(func):
    @wraps(func)
    def inner(*args, **kwargs):
        start_time = perf_counter()
        value = func(*args, **kwargs)
        end_time = perf_counter()
        run_time = end_time - start_time
        print(f'time of "{func.__name__}" is: {run_time}')
        return value

    return inner
```

درس ۲۲: ژنراتور(generator) – بخش اول

- * ژنراتور کمک می‌کند برنامه‌ها و کدهای ما بهینه تر نوشته شود.
- * رفتاری که با ژنراتور ایجاد می‌کنیم، همان رفتاری است که در ایتراتور(**iterator**) داشتیم.
- * در ژنراتور با صدا زدنتابع آن را اجرا نمی‌کند و یک شی برگردانده می‌شود که نوع آن ژنراتور است و می‌توان در آن با دستور **next** به عناصر بعدی دست یافت.
- * ژنراتور ها **Lazy** (تنبل) هستند یعنی همه‌ی عناصر را ایجاد نمی‌کند و زمان اضافی صرف نمی‌کند و فقط زمانی عناصر را ایجاد می‌کند که به آن‌ها نیاز داشته باشیم و این سبب می‌شود مصرف حافظه بهینه تری داشته باشیم.
- * با استفاده از دستور **yield** می‌شود تابع را به ژنراتور تبدیل کنیم.
- * در ژنراتورها به جای **yield** از **return** استفاده می‌کنیم.
- * زمانی که **next** را اجرا کنیم تا زمانی که به **yield** برسد و سپس توقف می‌کند تا **next** بعدی را اجرا کنیم و پس از آن تا **next** بعدی اجرا می‌کند.

```
def func(a):  
    print('reza')  
    yield a ** 2  
    print('hello')  
    yield 5  
    print('ok')  
    yield a - 2  
  
x = func(5)  
print(next(x))  
print(next(x))  
print(next(x))
```

```
reza  
25  
hello  
5  
ok  
3
```

* اگر **next** را **print** نکنیم مقدار **yield** چاپ نمی‌شود.

```
def func(a):  
    print('reza')  
    yield a ** 2  
  
x = func(5)  
next(x) # -> reza
```

* اگر در آخر **yield** نداشته باشیم بعد از اجرای دستورات خطای **StopIteration** می‌دهد.

```
def func(a):
    print('reza')
    yield a ** 2
    print('hello')
```

```
x = func(5)
print(next(x))
print(next(x))
```

```
reza
25
hello
Traceback (most recent call last):
  File "F:\python\pythonProject\test.py", line 9, in <module>
    print(next(x))
      ^^^^^^^^
StopIteration
```

* بعضی از ژنراتورها به **yield** بسیاری نیاز دارند که با استفاده از حلقه این مشکل را برطرف می‌کنیم.

```
def my_generator():
    for i in range(4):
        yield i ** 2
```

```
g = my_generator()
print(next(g))
print(next(g))
print(next(g))
print(next(g))
print(next(g))
```

```
0
1
4
9
Traceback (most recent call last):
  File "F:\python\pythonProject\test.py", line 11, in <module>
    print(next(g))
      ^^^^^^^^
StopIteration
```

* برای اینکه پس از پایان ژنراتور به ماتخاطی StopIteration برای مدیریت آن استفاده کنیم.

```
def my_generator():
    for i in range(4):
        yield i ** 2
```

```
g = my_generator()
for i in g:
    print(i)
```

0
1
4
9

درس ۲۳: ژنراتور(generator) – بخش دوم

مثال: یک تابع بنویسیم که کار تابع range پایتون را انجام دهد.

* ابتدا بدون استفاده از ژنراتور می‌نویسیم.

```
def list_range(start, end, step=1):
    new_range = []
    while start < end:
        new_range.append(start)
        start += step
    return new_range

r = range(10, 80, 9)

Lr = list_range(10, 80, 9)

print('range: ', list(r)) # -> range: [10, 19, 28, 37, 46, 55, 64, 73]
print('list_range: ', Lr) # -> list_range: [10, 19, 28, 37, 46, 55, 64, 73]
```

* سپس با استفاده از ژنراتور:

```
def gen_range(start, end, step=1):
    while start < end:
        yield start
        start += step

gr = list_range(10, 80, 9)

print('gen_range: ', gr) # -> gen_range: [10, 19, 28, 37, 46, 55, 64, 73]
```

=

```

*: مقایسه زمان و بهینه بودن دو تابعی که ساختیم، یکی ژنراتور (gen_range) و دیگری تابع معمولی (list_range):
from time import perf_counter

start = perf_counter()
s = 0
for i in list_range(1, 100000000):
    if i == 3:
        break
    s += i
end = perf_counter()

# ----

start2 = perf_counter()
s = 0
for i in gen_range(1, 10000000):
    if i == 3:
        break
    s += i
end2 = perf_counter()
print(end - start) # -> 19.03773509999155
print(end2 - start2) # -> 6.210000719875097e-05

```

* همانطور که می‌بینیم مدت زمان اجرا در ژنراتور بسیار کمتر است زیرا نیازی نیست تا آخر اجرا شود، چون ژنراتور از نوع **Lazy** (تنبل) است و فقط تا زمانی که ما نیاز داشته باشیم اجرا می‌شود.

* می‌توانیم از توابع مختلف (**sum**, **list**, **max**, **min**) برای ژنراتور استفاده کنیم.

```

def my_generator(r=10):
    for i in range(r):
        yield i ** 2

g = my_generator()
print(list(g)) # -> [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
print(list(g)) # -> [] Generator is finished
y = my_generator() # for another use we must define it again
print(list(y)) # -> [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

s = my_generator()
print(sum(s)) # -> 285
s = my_generator()
print(min(s)) # -> 0
s = my_generator()
print(max(s)) # -> 81

```

ژنراتور سه متد مهم دارد:

- `close`: هرجا که صدای زنیم ژنراتور را می‌بندد. (انگار به پایان رسیده است).
- `send`: در جلسات بعدی توضیح داده می‌شود.
- `throw`: می‌تواند سبب رخدادن یک استثنای شود.

```
def my_generator(r=10):
    for i in range(r):
        yield i ** 2

g = my_generator()
g.close()
print(next(g))
```

```
Traceback (most recent call last):
  File "F:\python\pythonProject\test.py", line 8, in <module>
    print(next(g))
               ^
StopIteration
```

```
def my_generator(r=10):
    for v in range(r):
        yield v ** 2

g = my_generator()
for i in g:
    if i == 16:
        g.throw(ValueError('Error!'))
    print(i)
```

```
0
1
4
9
Traceback (most recent call last):
  File "F:\python\pythonProject\test.py", line 9, in <module>
    g.throw(ValueError('Error!'))
  File "F:\python\pythonProject\test.py", line 3, in my_generator
    yield i ** 2
ValueError: Error!
```

* می توانیم از توابع تودر تو در ژنراتور استفاده کنیم و یک ژنراتور را به عنوان ورودی یک ژنراتور دیگر اختصاص بدهیم.

```
def square_gn(nums):
    for i in nums:
        yield i ** 2

def even_gn(nums):
    for i in nums:
        if i % 2 == 0:
            yield i

print(sum(square_gn(even_gn([1, 2, 3, 4, 7, 9, 1, 5, 2, 6]))) # -> 60
```

درس ۲۴ و ۲۵: تکالیف مبحث ژنراتور و حل آنها

۱. رفتار `enumerate` را با استفاده از ژنراتور پیاده سازی کنید.

```
def my_enumerate(sequence, start=0):
    c = start
    for r in sequence:
        yield c, r
        c += 1
```

```
li = ['ali', 'reza', 'neda']
```

```
for i, j in my_enumerate(li, 5):
    print(i, j)
```

```
5 ali
6 reza
7 neda
```

۲. یک ژنراتور برای تولید دنباله‌ی فیبوناچی بنویسید.

```
# 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ... : fibonacci numbers
```

```
def fibonacci():
    f1 = 0
    yield f1
    f2 = 1
    yield f2
    while True:
        f3 = f1 + f2
        yield f3
        f1 = f2
        f2 = f3
```

```
fib = fibonacci()
for _ in range(10):
    print(next(fib))
```

```
0
1
1
2
3
5
8
13
21
34
```

۳. ژنراتوری بنویسید که یک لیست گرفته و جمع عناصر آن را برگرداند. (هربار عنصر جدید جمع شود.)

مثال: لیست `[1, 2, 3, 4, 5]` را داریم، ژنراتور باید اولین بار ۱، دومین بار ۲ ۱، سومین بار ۳ ۲ ۱ و ... را برگرداند.

```
def sum_gen(lst):
    s = 0
    for i in lst:
        s += i
        yield s
```

```
sg = sum_gen([1, 2, 3, 4, 5])
for i in sg:
    print(i)
```

```
1
3
6
10
15
```

۴. ژنراتوری بنویسید که یک رشته گرفته و معکوس آن را برگرداند. (هربار یک کاراکتر).

```
def rev_str(s):
    a = len(s)
    for i in range(a-1, -1, -1):
        yield s[i]

sg = rev_str('abcd')
for ch in sg:
    print(ch)
```

d
c
b
a

۵. یک ژنراتور بی‌نهایت از اعداد زوج یا فرد بنویسید. زوج یا فرد بودن توسط کاربر تعیین می‌شود.

```
def my_gen(even_or_odd="e"):
    c = 0
    if even_or_odd == 'o':
        c = 1
    while True:
        yield c
        c += 2

e = my_gen('e')
for _ in range(5):
    print(next(e))
o = my_gen('o')
for _ in range(5):
    print(next(o))
```

۶. ژنراتوری ایجاد کنید که در هر مرحله، خروجی‌های زیر را پیدا کند.

بار اول: ۱، بار دوم: ۲۳، بار سوم: ۳۳۳، و ...

```
def num_gen():
    c = 1
    while True:
        s = ''
        for i in range(1, c+1):
            s += f'{c}\t'
        yield s
        c += 1

n = num_gen()
for j in range(5):
    print(next(n))
```

درس ۲۶: رفتار کروتین (coroutine)

* کروتین دستورالعملی است که اتصال یک سری از ورودی ها به خروجی ها را رقم می‌زند. (انگار که همزمان ورودی و خروجی خواهیم داشت).

* `yield` را می‌توانیم به متغیر اختصاص بدهیم و مقداری که در آن ذخیره می‌شود `None` است.

```
def my_gen():
    print('start ')
    while True:
        name = yield
        print('my name is: ', name)
```

```
g = my_gen()
next(g)
next(g)
next(g)
```

```
start
my name is: None
my name is: None
```

* گاهی می‌شود متغیر `yield` مقدار بگیرد، برای این کار از متده استفاده می‌کنیم.

```
def my_gen():
    print('start ')
    while True:
        name = yield
        print('my name is: ', name)
```

```
g = my_gen()
next(g)
next(g)
g.send('reza')
```

```
start
my name is: None
my name is: reza
```

* قبل از `next` باید از `send` استفاده کنیم.

```
def my_gen():
    print('start ')
    while True:
        name = yield
        print('my name is: ', name)
```

```
g = my_gen()
g.send('reza')
```

```
Traceback (most recent call last):
  File "F:\python\pythonProject\test.py", line 9, in <module>
    g.send('reza')
TypeError: can't send non-None value to a just-started generator
```

* پس از استفاده از `send` برای نمایش مقدار باید از `next` به جای استفاده کنیم.

```
def my_gen():
    print('start!')
    for i in range(10):
        name = yield i
        print('my name is: ', name)

g = my_gen()
print(next(g))
print('-' * 20)
print(g.send('reza'))
print('-' * 20)
print(next(g))
print('-' * 20)
print(g.send('neda'))
```

```
start!
0
-----
my name is: reza
1
-----
my name is: None
2
-----
my name is: neda
3
```

* برای اینکه قبل از `next`، `send` خودکار اجرا شود می‌توانیم از دکوراتورها استفاده کنیم.

```
from functools import wraps

def coroutine(func):
    @wraps(func)
    def start(*args, **kwargs):
        gn = func(*args, **kwargs)
        next(gn)
        return gn

    return start
```

```
@coroutine
def my_gen():
    print('start!')
    for i in range(10):
        name = yield i
        print('my name is: ', name)

g = my_gen()
print(g.send('reza'))
print(g.send('neda'))
print(g.send('sahel'))
```

```
start!
my name is: reza
1
my name is: neda
2
my name is: sahel
3
```

درس ۲۷: حل مثال برای رفتار کروتین

مثال: یک سانسور کننده بنویسیم که کلماتی که ممکن است نامناسب باشند را سانسور کنیم.

```
def cen_gen(words):
    print('Start!')
    w = None
    while True:
        word = yield w
        if word not in words:
            w = word
        else:
            w = '*' * len(word)
```

```
g = cen_gen(['donkey', 'cow', 'monkey'])
next(g)
print(g.send('reza'))
print(g.send('neda'))
print(g.send('donkey'))
print(g.send('ali'))
print(g.send('cow'))
print(g.send('monkey'))
print(g.send('sahel'))
```

```
Start!
reza
neda
*****
ali
***
*****
sahel
```

مثال: یک ژنراتور بنویسیم که کار `split` را انجام دهد. (کاراکترهایی که داخل یک رشته هستند را براساس یک کاراکتر خاص جدا می‌کند و داخل لیست قرار دهد.)

```
def spl_gen(delimiter=' '):
    print('Start!')
    s = None
    while True:
        line = yield s
        s = line.split(delimiter)
```

```
g = spl_gen('-')
next(g)
print(g.send('ali-reza-neda-sahel'))

f = spl_gen(' ')
next(f)
print(f.send('reza neda'))
```

```
Start!
['ali', 'reza', 'neda', 'sahel']
Start!
['reza', 'neda']
```

درس ۲۸: صفات تابع

* هر تابعی می‌تواند صفاتی داشته باشد.

* با استفاده از `dir` می‌توانیم مشخص کنیم که یک شی چه متدها و صفاتی را دارد.

```
def func(x=0):
    """doc..."""
    pass

print(dir(func))
```

* با استفاده از `__name__` و `__doc__` می‌توانیم داک استرینگ و اسم تابع را چاپ کنیم.

```
def func(x=0):
    """this is a docstring"""
    pass

print(func.__doc__) # -> this is a docstring
print(func.__name__) # -> func
```

* ما می‌توانیم صفات و اtribووت‌های دلخواه خودمان را به تابع اضافه کنیم، با استفاده از نقطه `(.)` و نوشتن اسم صفت و دادن مقدار می‌توانیم این کار را انجام دهیم.

```
def ave(li):
    return sum(li) / len(li)

def ave2(li):
    return sum(li) / len(li)

ave.school_name = 'danesh'
ave2.school_name = 'honar'

print(ave.school_name) # -> danesh
print(ave2.school_name) # -> honar
```

* با استفاده از تابع `setattr` می‌توانیم اtribووت دلخواهمان اضافه کنیم، این تابع سه ورودی می‌گیرد، ابتدا اسم شی که می‌خواهیم به آن اtribووت بدهیم، سپس اسم دلخواه اtribووت و در آخر مقدار را قرار می‌دهیم.

```
def ave(li):
    return sum(li) / len(li)

setattr(ave, 'school_name', 'danesh')
print(ave.school_name) # -> danesh
```

* تابع `__dict__` یک دیکشنری از اtribووت‌هایی که به تابع اختصاص دادیم به همراه مقدارشان را نمایش می‌دهد.

```
def ave(li):
    return sum(li) / len(li)

setattr(ave, 'school_name', 'danesh')
print(ave.__dict__) # -> {'school_name': 'danesh'}
```

* با استفاده از تابع `getattr` می‌توانیم مقدار اtribووت مورد نظرمان در تابع را چاپ کنیم، این تابع دو ورودی می‌گیرد، ابتدا اسم شی که می‌خواهیم مقدار اtribووت آن را مشاهده کنیم و سپس اسم اtribووت را قرار می‌دهیم و اگر آن اtribووت را نداشتمیم می‌توان مقدار دلخواه به این تابع بدهیم.

```
def ave(li):
    return sum(li) / len(li)

setattr(ave, 'school_name', 'danesh')

print(getattr(ave, 'school_name')) # -> danesh
print(getattr(ave, 'name', None)) # -> None
print(getattr(ave, 'name', 'no attribute')) # -> no attribute
```

* با استفاده از تابع `hasattr` می‌توانیم بررسی کنیم که آیا تابع اtribووت را دارد یا خیر، نتیجه `True` یا `False` است.

```
def ave(li):
    return sum(li) / len(li)

setattr(ave, 'school_name', 'danesh')

print(hasattr(ave, 'school_name'))
if hasattr(ave, 'school_name'):
    print(getattr(ave, 'school_name'))
print(hasattr(ave, 'name'))
if hasattr(ave, 'name'):
    print(getattr(ave, 'name'))
```

True
danesh
False

* برای حذف اtribووت می‌توانیم از `delattr` و یا `del` استفاده کنیم.

```
def ave(li):
    return sum(li) / len(li)

setattr(ave, 'school_city', 'shiraz')
setattr(ave, 'school_name', 'danesh')
print(ave.__dict__) # -> {'school_city': 'shiraz', 'school_name': 'danesh'}

delattr(ave, 'school_name')
del ave.school_city
print(ave.__dict__) # -> {}
```

درس ۲۹: تابع بازگشتی

```
def recursive():
    ...
    if stop_condition:
        ...
        return ...
    ...
    recursive()
```

```
recursive()
```

- * توابع بازگشتی زمانی بیشتر به درد میخورد که به الگوریتم مسلط باشیم.
- * توابع بازگشتی هیچ دستور و کلمه کلیدی جدید ندارد.
- * هر تابعی که خودش را داخل خودش صدا بزند تابع بازگشتی است.

مثال: فاکتوریل یک عدد را با تابع بازگشتی به دست بیاورید.

* ابتدا با یک تابع عادی فاکتوریل را به دست می‌آوریم:

```
def fact(x):
    f = 1
    for i in range(1, x + 1):
        f *= i
    return f

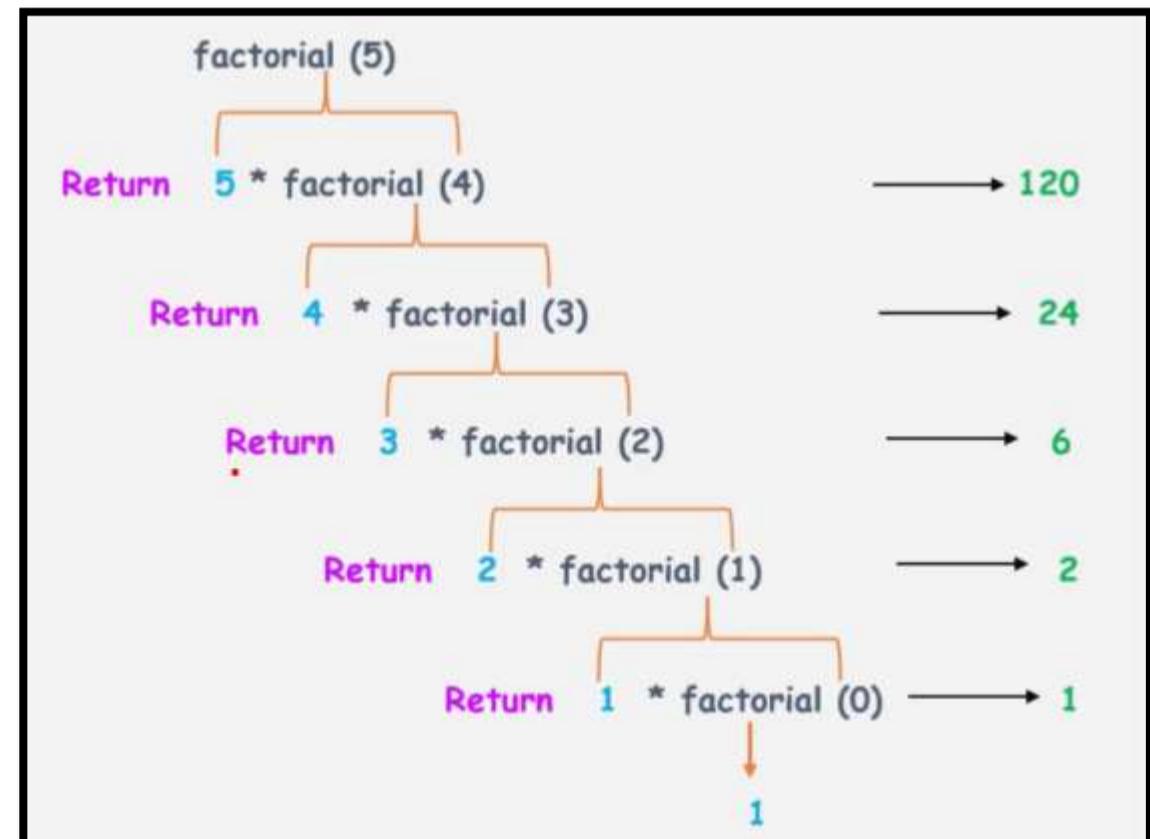
print(fact(5)) # -> 120
```

* سپس با استفاده از تابع برگشتی فاکتوریل را به دست می‌آوریم:

```
# 5! = 5 * 4 * 3 * 2 * 1
# 5! = 5 * 4!
# 5! = 5 * 4 * 3!
# 5! = 5 * 4 * 3 * 2!

def rec_fact(x):
    if x == 0 or x == 1:
        return 1
    return x * rec_fact(x - 1)

print(rec_fact(5)) # -> 120
```



درس ۳۰: حل مثال تابع بازگشتی

مثال: یک تایمر با استفاده از تابع بازگشتی بنویسید.

```
from time import sleep

def reverse(x):
    if x <= 0:
        return None
    sleep(1)
    print(x)
    reverse(x - 1)

reverse(10)
```

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

مثال: یک تابع برگشتی بنویسید که یک عدد بگیرد و حاصل ضرب ارقام بزرگتر مساوی از ۵ را به دست بیاورد.

```
def mul(n):
    if n == 0:
        return 1
    elif n % 10 < 5:
        return mul(n // 10)
    else:
        return n % 10 * mul(n // 10)

print(mul(1245367124657)) # -> 44100
```

مثال: یک تابع برگشتی بنویسیم که یک عدد بگیرد و اگر عدد و عدهای کوچکتر از آن مضرب ۳ باشد را چاپ کند.

```
def func(n):
    if n < 3:
        return
    elif n % 3 == 0:
        print(n)
    func(n - 1)

func(12)
```

```
12
9
6
3
```

مثال: یک تابع برگشتی بنویسیم که سری فیبوناچی را چاپ کند.

```
# 0, 1, 1, 2, 3, 5, 8, 13 ...
# 0, 1, 2, 3, 4, 5, 6, 7, ...

def fib(n):
    if n == 0 or n == 1:
        return n
    return fib(n - 1) + fib(n - 2)

print(fib(7)) # -> 13
```

درس ۳۱ : تکالیف مبحث تابع بازگشتی

۱. یک تابع بازگشتی برای محاسبه مجموع عناصر در یک لیست بنویسید.
 ۲. یک تابع بازگشتی برای محاسبه توان یک عدد بنویسید.
 ۳. یک تابع بازگشتی برای معکوس کردن یک رشته بنویسید.
 ۴. یک تابع بازگشتی برای شمارش تعداد تکرار یک عنصر در یک لیست بنویسید.
 ۵. یک تابع بازگشتی برای بررسی این‌که آیا رشته **palindrome** است یا خیر بنویسید. (از هر دو طرف به یک شکل خوانده می‌شود).
 ۶. یک تابع بازگشتی برای محاسبه مجموع ارقام در یک عدد صحیح مثبت بنویسید.
-

درس ۳۲: دکوراتور و ژنراتور بازگشتی و عمق بازگشتی

- * برای تابع بازگشتی می‌توانیم دکوراتور و ژنراتور بنویسیم ولی زیاد کاربرد ندارد.
- * دکوراتور در تمام فراخوانی‌های تابع بازگشتی صدا زده می‌شود.
- * قبل از اینکه تابع در دکوراتور تمام شود خودش را صدا می‌زند و قسمت قبل از تابع اجرا می‌شود و پس از اینکه تابع بازگشتی به اتمام رسید آن وقت قسمت بعد از تابع به تعداد صدای زدن تابع بازگشتی اجرا می‌شود.

```
from functools import wraps
from time import sleep

def dec(func):
    @wraps(func)
    def start(*args, **kwargs):
        print("*" * 40)
        value = func(*args, **kwargs)
        print("+" * 40)
        return value

    return start

@dec
def reverse(x):
    if x <= 0:
        return None
    sleep(0.5)
    print(x)
    reverse(x - 1)

reverse(5)
```

```
*****
5
*****
4
*****
3
*****
2
*****
1
*****
+++++
+++++
+++++
+++++
+++++
+++++
```

* در ژنراتور ممکن است نتیجه با آنچه انتظار داریم متفاوت باشد.

```
from time import sleep

def reverse(x):
    if x <= 0:
        return
    sleep(0.5)
    print(x)
    reverse(x - 1)

reverse(5) # -> 5
           # -> 4
           # -> 3
           # -> 2
           # -> 1

def g_rev(x):
    if x <= 0:
        return
    sleep(0.5)
    yield x
    g_rev(x - 1)

g = g_rev(5)
print(list(g)) # -> [5]      # انتظار داشتیم که نتیجه یک لیست از ۰ تا ۵ باشد!

def g_rev(x):
    if x <= 0:
        return
    sleep(0.5)
    yield x
    for i in g_rev(x - 1): # برای حل این مشکل می‌توانیم به این صورت بنویسیم:
        yield i

g = g_rev(5)
print(list(g)) # -> [5, 4, 3, 2, 1]
```

* در توابع بازگشتی با هر بار صدای زدن تابع قبلی در حافظه (پشته) ذخیره می‌شود، که به صورت پیشفرض تقریباً ۱۰۰۰ می‌باشد که با می‌توانیم این محدودیت را مشاهده کنیم و اگر بیشتر از این تابع ذخیره شود خطای می‌دهد.

```
import sys

print(sys.getrecursionlimit()) # -> 1000

def reverse(x):
    if x <= 0:
        return
    print(x)
    reverse(x - 1)

reverse(1000)
```

```
File "F:\python\pythonProject\test.py", line 9, in reverse
    print(x)
RecursionError: maximum recursion depth exceeded while calling a Python object
```

* با استفاده از `setrecursionlimit` می‌توانیم این محدودیت را به دلخواه تغییر دهیم.

```
import sys

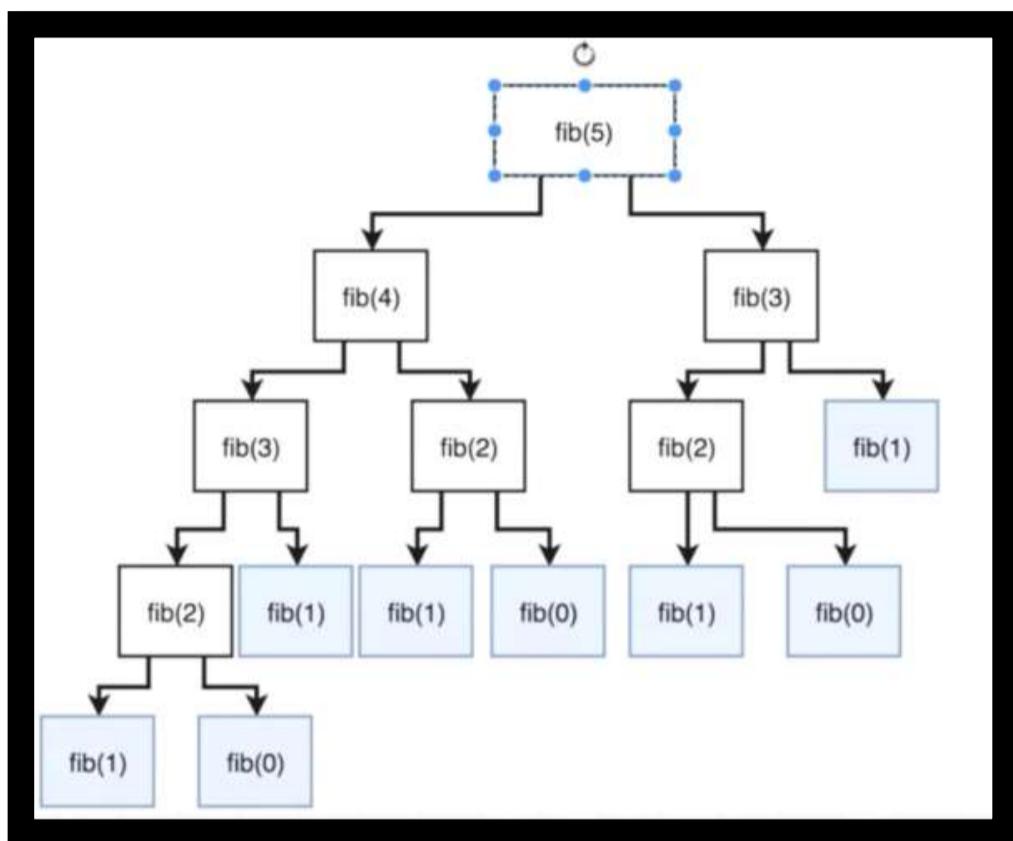
sys.setrecursionlimit(1000000000)

def reverse(x):
    if x <= 0:
        return
    print(x)
    reverse(x - 1)

reverse(1001)
```

درس ۳۳: تکنیک به خاطر سپاری (Memoization)

- * تابع بازگشتی معمولاً کندتر و مصرف حافظش بیشتر است، چون تابع را چندین بار صدای زند و اجرا می‌کند.
- * به عنوان مثال در تابع بازگشتی فیبوناچی چندین تابع تکراری را حساب می‌کند، که می‌توانیم برای سریع‌تر شدن یک دکوراتور بنویسیم و در آن یک دیکشنری قرار دهیم و ورودی‌های تابع را به آن بدهیم و جواب آن ورودی‌ها را که اگر قبلًا حاصل پیدا شده و در دیکشنری نوشته شده است نیاز به اجرای مجدد نیست و مقدار را بدهد و در غیر این صورت تابع را حساب کند.



```
# 0, 1, 1, 2, 3, 5, 8, 13 ...
# 0, 1, 2, 3, 4, 5, 6, 7, ...
from functools import wraps

def memoize(func):
    memory = {}

    @wraps(func)
    def wrapper_decorator(n):
        if n not in memory:
            memory[n] = func(n)
        return memory[n]

    return wrapper_decorator

@memoize
def fib(n):
    if n == 0 or n == 1:
        return n
    return fib(n - 1) + fib(n - 2)

print(fib(100)) # 354224848179261915075 # برنامه خیلی سریع اجرا می‌شود.
```

درس ۳۴: مینی پروژه: بازی دوز یا tic tac toe

```
from colorama import Fore

board = list(range(1, 10))
winners = ((0, 1, 2), (3, 4, 5), (6, 7, 8), (0, 3, 6), (1, 4, 7), (2, 5, 8),
(0, 4, 8), (2, 4, 6))
moves = ((5,), (1, 3, 7, 9), (2, 4, 6, 8))

def print_board():
    j = 1
    for i in board:
        end = ' '
        if j % 3 == 0:
            end = '\n\n'
        if i == 'X':
            print(Fore.LIGHTBLUE_EX + f'[{i}]', end=end)
        elif i == '0':
            print(Fore.LIGHTYELLOW_EX + f'[{i}]', end=end)
        else:
            print(Fore.LIGHTWHITE_EX + f'[{i}]', end=end)
        j += 1

def make_move(brd, plr, mve, undo=False):
    if can_move(brd, mve):
        brd[mve - 1] = plr
        win = is_winner(brd, plr)
        if undo:
            brd[mve - 1] = mve
        return True, win
    return False, False

def can_move(brd, mve):
    if mve in range(1, 10) and isinstance(brd[mve - 1], int):
        return True
    return False

def is_winner(brd, plr):
    win = True
    for tup in winners:
        win = True
        for j in tup:
            if brd[j] != plr:
                win = False
                break
        if win:
            break
    return win
```

```

def has_empty_space():
    return board.count('X') + board.count('O') != 9

def computer_move():
    mv = -1
    # Is computer can win?
    for i in range(1, 10):
        if make_move(board, computer, i, True)[1]:
            mv = i
            break
    # If player can win, stop the player.
    if mv == -1:
        for j in range(1, 10):
            if make_move(board, player, j, True)[1]:
                mv = j
                break
    # Move!
    if mv == -1:
        for tup in moves:
            for m in tup:
                if mv == -1 and can_move(board, m):
                    mv = m
                    break
    return make_move(board, computer, mv)

player, computer = 'X', 'O'
print(Fore.LIGHTWHITE_EX + ' Player: ', Fore.LIGHTBLUE_EX + 'X')
print(Fore.LIGHTWHITE_EX + 'Computer: ', Fore.LIGHTYELLOW_EX + 'O\n')
while has_empty_space():
    print_board()
    move = int(input(Fore.LIGHTWHITE_EX + 'Choose your move: '))
    moved, won = make_move(board, player, move)
    if not moved:
        print(Fore.LIGHTRED_EX + 'Invalid number! Try again')
        continue
    if won:
        print(Fore.LIGHTGREEN_EX + 'You won!')
        break
    elif computer_move()[1]:
        print(Fore.LIGHTRED_EX + 'You lost!')
        break

print_board()

```

فصل هفتم: انواع داده (سطح دو)

درس ۱: متدهای اعداد

* تابع `divmod` دو متغیر می‌پذیرد که ابتدا حاصل تقسیم متغیر اول بر متغیر دوم و سپس باقیمانده متغیر اول به متغیر دوم.

```
a = divmod(10, 3) # (x // y, x % y)
print(a) # -> (3, 1)
```

* تابع `pow` توان را حساب می‌کند و اگر متغیر سوم بدهیم، باقیمانده حاصل توان را به متغیر سوم حساب می‌کند.

```
a = pow(2, 4)
print(a) # -> 16

b = pow(2, 4, 3) # 2 ** 4 = 16, 16 % 3 = 1
print(b) # -> 1
```

* تابع `round` عدد را گرد می‌کند.

```
print(round(2.3675645, 2)) # -> 2.37
```

* تابع `abs` قدر مطلق عدد را به دست می‌آورد.

```
print(abs(234)) # -> 234
print(abs(-67)) # -> 67
```

* متدهای `as_integer_ratio` نسبت را پیدا می‌کند.

```
x = 0.75
print(x.as_integer_ratio()) # -> (3, 4)
y = 0.25
print(y.as_integer_ratio()) # -> (1, 4)
z = 5
print(z.as_integer_ratio()) # -> (5, 4)
```

* متدهای `bit_count` نشان می‌دهد یک عدد باینری چند بیت ۱ دارد.

```
x = 10 # 1010
print(x.bit_count()) # -> 2
```

* متدهای `denominator` مخرج یک کسر اعداد صحیح را نشان می‌دهد و حاصل همیشه ۱ می‌باشد.

```
x = 5425
print(x.denominator) # -> 1
```

* متد numerator صورت یک کسر اعداد صحیح (خود عدد) را نشان می‌دهد.

```
x = 5425
print(x.numerator) # -> 5425
```

* متد to_bytes یک عدد صحیح را به آرایه‌ای از بایت‌ها تبدیل می‌کند.

این متد سه ورودی دارد:

- ورودی اول طول آرایه را مشخص می‌کند.
- ورودی دوم byteorder است که مشخص می‌کند با ارزش‌ترین در ابتدای آرایه باشد و یا در انتهای، "big" مشخص می‌کند که در ابتدای باشد و "little" مشخص می‌کند که باید در انتهای باشد.
- ورودی سوم signed است که مشخص می‌کند متمم دو باشد یا خیر و دو مقدار False و True به خود می‌گیرد.

```
• x = 16
  print(x.to_bytes(3, byteorder='big', signed=False)) # -> b'\x00\x00\x10'
  print(x.to_bytes(3, byteorder='little', signed=False)) # -> b'\x10\x00\x00'
  print(x.to_bytes(3, byteorder='big', signed=True)) # -> b'\x00\x00\x10'
```

* متد to_bytes بر عکس from_bytes است که یک آرایه را به عدد تبدیل می‌کند.

```
print(int.from_bytes(b'\x00\x00\x10', byteorder='big')) # -> 16
```

* متد hex مبنای ۱۶ یک عدد اعشاری را نشان می‌دهد.

```
f = 16.5
print(f.hex()) # -> 0x1.080000000000p+4
```

* متد hex بر عکس fromhex عمل می‌کند.

```
print(float.fromhex('0x1.080000000000p+4')) # -> 16.5
```

* متد is_integer مشخص می‌کند که عدد اعشاری، صحیح هست یا خیر.

```
f = 16.5
print(f.is_integer()) # -> False

x = 17.00
print(x.is_integer()) # -> True
```

* متد bit_length مشخص می‌کند چه تعداد بیت برای نوشتگی یک عدد نیاز داریم.

```
x = 10
print(bin(x)) # -> 0b1010
print(x.bit_length()) # -> 4
```

درس ۲: متدهای لیست

* از متدهای append برای اضافه کردن یک عنصر جدید به لیست استفاده می‌کنیم.

```
li = [1, 2, 3]
li.append(5)
print(li) # -> [1, 2, 3, 5]
```

* از روش زیر نیز می‌توانیم یک عنصر جدید به لیست اضافه کنیم.

```
li = [1, 2, 3]
li[len(li):] = [5]
print(li) # -> [1, 2, 3, 5]
```

* از متدهای clear برای پاک کردن عناصر لیست استفاده می‌کنیم.

```
li = [1, 2, 3]
li.clear()
print(li) # -> []
```

* از متدهای count برای شمردن یک عنصر خاص در لیست استفاده می‌کنیم.

```
li = [1, 2, 3, 2, 2]
print(li.count(2)) # -> 3
```

* از متدهای copy برای گرفتن کپی سطحی استفاده می‌کنیم.

```
li = [1, 2, 3]
li2 = li.copy()
li2[0] = 0
print(li) # -> [1, 2, 3]
print(li2) # -> [0, 2, 3]
```

* از متدهای extend برای توسعه دادن به لیست استفاده می‌کنیم.

```
li = [1, 2, 3]
li.extend([4, 5, 6])
print(li) # -> [1, 2, 3, 4, 5, 6]
```

* از متدهای index برای پیدا کردن ایندکس یک عنصر استفاده می‌کنیم.

```
li = [7, 5, 6, 4, 1]
print(li.index(5)) # -> 1
```

* از متدهای insert برای قرار دادن یک عنصر در ایندکس مورد نظرمان استفاده می‌کنیم.

```
li = [7, 4, 6, 4, 5]
li.insert(0, 'a')
print(li) # -> ['a', 7, 4, 6, 4, 5]
```

* میتوانیم مشخص کنیم **index** تا کدام ایندکس برای پیدا کردن عنصر مورد نظر جستجو کند.

```
li = [7, 4, 6, 4, 5]
print(li.index(5, 2, 4))
```

```
Traceback (most recent call last):
  File "F:\python\pythonProject\test.py", line 2, in <module>
    print(li.index(5, 2, 4))
    ^^^^^^^^^^^^^^^^^^^^^^
ValueError: 5 is not in list
```

* از متدهای **pop** برای حذف کردن و نمایش عنصر حذف شده استفاده می‌کنیم و اگر مشخص نکنیم کدام عنصر حذف شود، عنصر آخر را حذف می‌کند، ورودی این تابع ایندکس عنصر مورد نظر می‌باشد.

```
li = [6, 5, 9]
print(li.pop()) # -> 9
print(li.pop(0)) # -> 6
```

* از متدهای **remove** برای حذف کردن یک عنصر استفاده می‌کنیم و اگر عنصری باشد که چند بار تکرار شده است فقط عنصر اول را حذف می‌کند.

```
li = [1, 4, 3, 4, 5, 6, 7]
li.remove(4)
print(li) # -> [1, 3, 4, 5, 6, 7]
```

* از متدهای **reverse** برای برعکس کردن عناصر لیست استفاده می‌کنیم.

```
li = [1, 4, 3, 4, 5, 6, 7]
li.reverse()
print(li) # -> [7, 6, 5, 4, 3, 4, 1]
```

* از متدهای **sort** برای مرتب کردن عناصر لیست استفاده می‌کنیم.

```
li = [8, 4, 2, 4, 1, 6, 6]
li.sort()
print(li) # -> [1, 2, 4, 4, 6, 6, 8]
```

درس ۳: متدهای تاپل و مجموعه

* از متدهای count برای شمردن یک عنصر خاص در تاپل استفاده می‌کنیم.

```
t = (1, 2, 3, 2, 2)
print(t.count(2)) # -> 3
```

* از متدهای index برای پیدا کردن ایندکس یک عنصر در تاپل استفاده می‌کنیم.

```
t = (7, 5, 6, 4, 1)
print(t.index(5)) # -> 1
```

* از متدهای clear برای پاک کردن عناصر مجموعه استفاده می‌کنیم.

```
s = {1, 2, 3}
s.clear()
print(s) # -> set()
```

* از متدهای copy برای گرفتن کپی سطحی استفاده می‌کنیم.

```
s = {1, 2, 3}
s2 = s.copy()

print(s) # -> {1, 2, 3}
print(s2) # -> {1, 2, 3}
```

* از متدهای difference برای مشخص کردن اختلاف دو مجموعه استفاده می‌کنیم و این تغییرات روی خود مجموعه اعمال نمی‌شوند.

```
s = {1, 2, 3, 4}
x = {3, 4, 5, 6}
print(s.difference(x)) # -> {1, 2}
print(s) # -> {1, 2, 3, 4}
```

* اگر بخواهیم این تغییرات روی خود مجموعه اعمال شوند از difference_update استفاده می‌کنیم.

```
s = {1, 2, 3, 4}
x = {3, 4, 5, 6}
s.difference_update(x)
print(s) # -> {1, 2}
```

* برای مشخص کردن اینکه دو مجموعه اشتراکی دارند یا خیر از isdisjoint استفاده می‌کنیم و نتیجه در صورتی True می‌باشد که اشتراکی نداشته باشند.

```
s = {1, 2, 3, 4}
x = {3, 4, 5, 6}
z = {8, 9, 7, 5}

print(s.isdisjoint(x)) # -> False
print(s.isdisjoint(z)) # -> True
```

* از متدهای **pop** برای حذف کردن عنصر آخر مجموعه و نمایش آن استفاده می‌کنیم.

```
s = {6, 5, 9}
print(s.pop()) # -> 9
print(s) # -> {5, 6}
```

درس ۴: متدهای دیکشنری

* از متدهای clear برای پاک کردن عناصر دیکشنری استفاده می‌کنیم.

```
d = {'a': 1, 'b': 2}  
d.clear()  
print(d) # -> {}
```

* از متدهای copy برای گرفتن کپی سطحی استفاده می‌کنیم.

```
d = {'a': 1, 'b': 2}  
c = d.copy()  
c['c'] = 3  
print(d) # -> {'a': 1, 'b': 2}  
print(c) # -> {'a': 1, 'b': 2, 'c': 3}
```

* برای ساختن دیکشنری با کلید از متدهای fromkeys استفاده می‌کنیم.

```
print(dict.fromkeys(['a', 'b'], 7)) # -> {'a': 7, 'b': 7}
```

* برای نمایش مقدار یک کلید در دیکشنری از متدهای get استفاده می‌کنیم و می‌توانیم به آن مقدار اولیه بدهیم.

```
d = {'a': 1, 'b': 2}  
print(d.get('a')) # -> 1  
print(d.get('c')) # -> None  
print(d.get('c', 0)) # -> 0
```

* از متدهای pop برای حذف یک آیتم و نمایش مقدار آن استفاده می‌کنیم، این متدهای کلید را به عنوان ورودی می‌گیرد و می‌توانیم به آن مقدار بدهیم.

```
d = {'a': 1, 'b': 2}  
print(d.pop('a')) # -> 1  
print(d.pop('c', 0)) # -> 0  
print(d) # -> {'b': 2}
```

* از متدهای popitem برای حذف یک آیتم از آخر دیکشنری و نمایش آن آیتم استفاده می‌کنیم.

```
d = {'a': 1, 'b': 2}  
print(d.popitem()) # -> ('b', 2)  
print(d) # -> {'a': 1}
```

* از متدهای setdefault برای نمایش مقدار یک کلید استفاده می‌کنیم و اگر آن کلید وجود نداشته باشد، None را نشان می‌دهد و همچنان می‌توانیم به آن مقدار بدهیم.

```
d = {'a': 1, 'b': 2}  
print(d.setdefault('a')) # -> 1  
print(d.setdefault('c')) # -> None  
print(d.setdefault('d', 0)) # -> 0  
print(d) # -> {'a': 1, 'b': 2, 'c': None, 'd': 0}
```

* از متدهای **update** برای اضافه کردن آیتم به دیکشنری استفاده می‌کنیم و به چند صورت می‌توانیم این کار را انجام دهیم.

```
d = {'a': 1, 'b': 2}
d.update({'a': 0, 'c': 5, 'd': 7})
print(d) # -> {'a': 0, 'b': 2, 'c': 5, 'd': 7}
```

```
d = {'a': 1, 'b': 2}
d.update([('a', 0), ('c', 5), ('d', 7)])
print(d) # -> {'a': 0, 'b': 2, 'c': 5, 'd': 7}
```

```
d = {'a': 1, 'b': 2}
d.update(a=0, c=5, d=7)
print(d) # -> {'a': 0, 'b': 2, 'c': 5, 'd': 7}
```

* برای تغییر یک کلید در دیکشنری از روش زیر استفاده می‌کنیم.

```
d = {'a': 1, 'b': 2}
d['x'] = d.pop('b')
print(d) # -> {'a': 1, 'x': 2}
```

درس ۵: کار با عملکر والروس

مثال: یک حلقه بنویسید که به یک لیست اسم اضافه کند و تا زمانی که کاربر بخواهد ادامه داشته باشد.

```
x = []
s = input('name(q for quit): ')
while s.lower() != 'q':
    x.append(s)
    s = input('name(q for quit): ')

print('names:', x)
```

* می‌توانیم مثال بالا را بهینه تر حل کنیم.

```
x = []
while True:
    s = input('name(q for quit): ')
    if s.lower() == 'q':
        break
    x.append(s)

print('names:', x)
```

* می‌توانیم با **عملگر والروس (=)** مثال بالا را بازهم بهینه تر حل کنیم.

```
x = []
while (s := input('name(q for quit): ')).lower() != 'q':
    x.append(s)

print('names:', x)
```

* در شرط‌ها نیز می‌توانیم از **عملگر والروس** استفاده کنیم.

```
a = [1, 2, 3, 4]
if (n := len(a)) > 3:
    print(n) # -> 4
```

* در توابع نیز می‌توانیم از **عملگر والروس** استفاده کنیم.

```
def func(x):
    print(x ** 2)

func(z := 5) # -> 25
print(z) # -> 5
```

* بهتر است از **عملگر والروس** با پرانتز () استفاده کنیم.

مثال: یک دیکشنری بنویسید که کلید اولش اندازه‌ی لیست، کلید دومش جمع عناصر لیست و کلید سومش میانگین عناصر لیست باشد.

```
x = [1, 2, 3, 4]
d = {'l': len(x), 's': sum(x), 'a': sum(x) / len(x)}

print(d) # -> {'l': 4, 's': 10, 'a': 2.5}
```

* می‌توانیم مثال بالا را بهینه تر حل کنیم.

```
x = [1, 2, 3, 4]
li = len(x)
s = sum(x)
d = {'l': li, 's': s, 'a': s / li}

print(d) # -> {'l': 4, 's': 10, 'a': 2.5}
```

* می‌توانیم با **عملگر والروس** مثال بالا بازهم بهینه تر حل کنیم.

```
x = [1, 2, 3, 4]
d = {
    'l': (li := len(x)),
    's': (s := sum(x)),
    'a': s / li
}

print(d) # -> {'l': 4, 's': 10, 'a': 2.5}
```

درس ۶: تکالیف مبحث عملگر والروس

۱. یک رشته بگیرید و طول آن را در صورتی که بیشتر از ۱۰ باشد چاپ کنید. (از عملگر والروس در شرط استفاده کنید.)
۲. در صورتی که یک لیست بیشتر از ۵ عنصر داشته باشد، جمع دو عنصر اول آن را چاپ کنید. (از عملگر والروس استفاده کنید.)
۳. برنامه‌ای بنویسید که جمع اعدادی که کاربر وارد می‌کند را محاسبه کند. برنامه تا زمانی که کاربر یک رشته خالی نزدیک است ادامه پیدا کند. (از عملگر والروس استفاده کنید.)
۴. بررسی کنید که آیا یک عدد خاص (مثلاً ۵) در یک لیست وجود دارد یا خیر. در صورت وجود ایندکس آن را چاپ کنید. (از عملگر والروس استفاده کنید.)
۵. برنامه‌ای بنویسید که اعداد تصادفی بین یک تا ۱۰۰ را تولید می‌کند. اگر عدد تولید شده بزرگ‌تر از ۸۰ باشد برنامه متوقف می‌شود. تعداد اعداد تولید شده باید چاپ شود. (از عملگر والروس استفاده کنید.)
۶. یک جمله از کاربر بگیرید و با استفاده از عملگر والروس، تعداد کلمات آن را چاپ کنید.

درس ۷: خلاصه سازی یا comprehension (بخش اول)(+مثال)

* با **خلاصه سازی** می‌توانیم لیست را ایجاد و دستورات مربوط به لیست را بنویسیم.

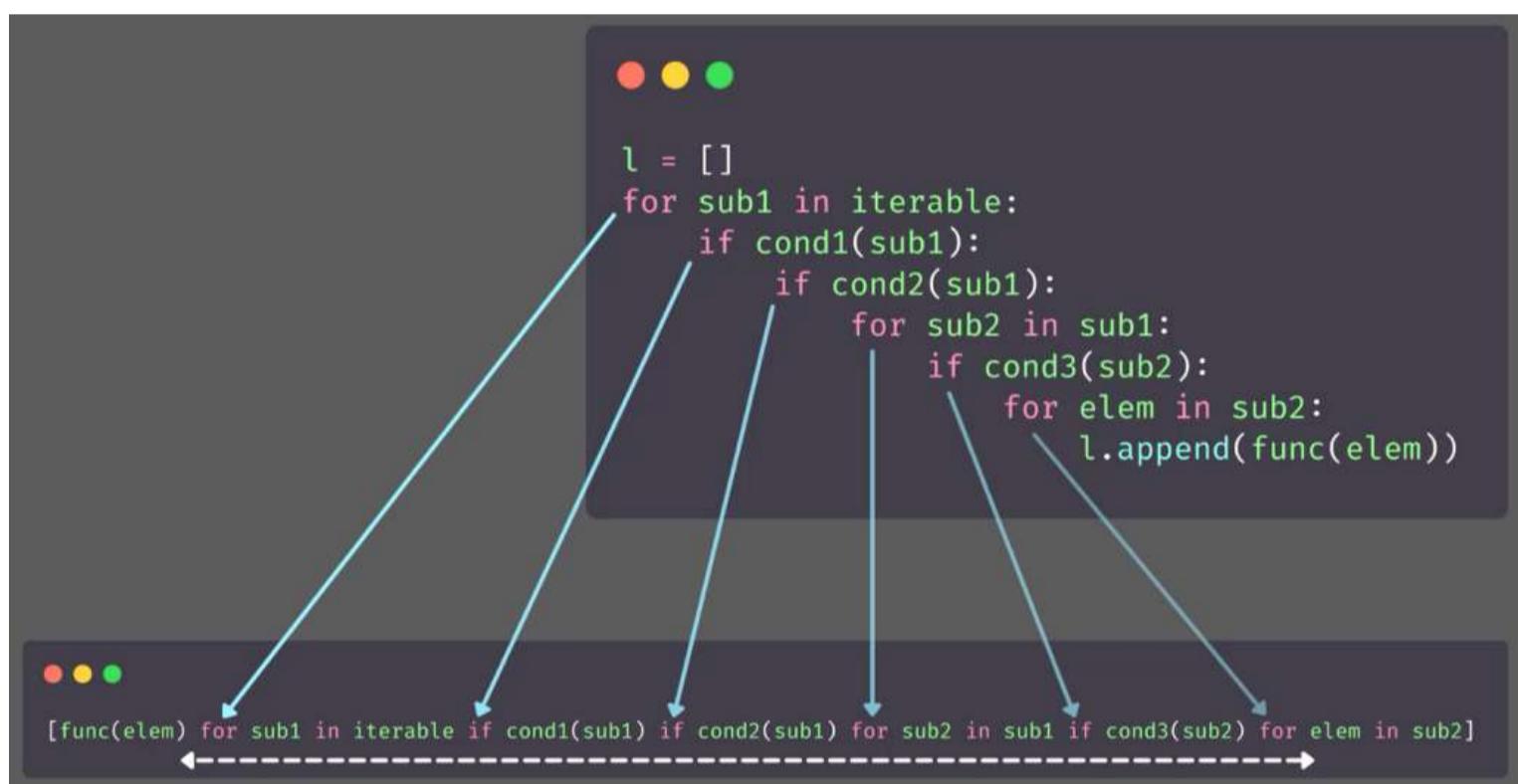
مثال: یک لیست بنویسیم که شامل توان دوم اعداد ۰ تا ۹ باشد.

```
# روش نرمال
s = []
for i in range(10):
    s.append(i ** 2)

print(s) # -> [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
# -----
# روش خلاصه سازی
x = [i ** 2 for i in range(10)]
print(x) # -> [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

ساختار کلی خلاصه سازی:

* ابتدا مقداری که قرار است به لیست اضافه شود را می‌نویسیم و سپس بقیه دستورات را پشت سرهم و بدون دو نقطه می‌نویسیم.



```
# روش نرمال
s = []
for i in range(100):
    if i % 2 == 0:
        if i > 10:
            if i < 90:
                if i % 3 == 0:
                    s.append(i - 1)

print(s) # -> [11, 17, 23, 29, 35, 41, 47, 53, 59, 65, 71, 77, 83]
# -----
# روش خلاصه سازی
x = [i - 1 for i in range(100) if i % 2 == 0 if i > 10 if i < 90 if i % 3 == 0]
print(x) # -> [11, 17, 23, 29, 35, 41, 47, 53, 59, 65, 71, 77, 83]
```

مثال: یک لیست بنویسیم شامل توان دوم اعداد ۰ تا ۹ در صورتی که زوج باشند.

```
# روش نرمال
s = []
for i in range(10):
    if i % 2 == 0:
        s.append(i ** 2)

print(s) # -> [0, 4, 16, 36, 64]
# -----
# روش خلاصه سازی
x = [i ** 2 for i in range(10) if i % 2 == 0]
print(x) # -> [0, 4, 16, 36, 64]
```

مثال: عناصر دو لیست را به ترتیب در یک تاپل قرار دهیم و اگر عناصر تکراری بودند، آن ها را قرار ندهیم (عنصر اول از لیست اول با عنصر اول از لیست دوم و عنصر اول از لیست اول با عنصر دوم از لیست دوم ...)

```
# روش نرمال
s1 = [1, 2, 3]
s2 = [4, 2, 3]
s = []
for i in s1:
    for j in s2:
        if i != j:
            s.append((i, j))

print(s) # -> [(1, 4), (1, 2), (1, 3), (2, 4), (2, 3), (3, 4), (3, 2)]
# -----
# روش خلاصه سازی
s1 = [1, 2, 3]
s2 = [4, 2, 3]
x = [(i, j) for i in s1 for j in s2 if i != j]
print(x) # -> [(1, 4), (1, 2), (1, 3), (2, 4), (2, 3), (3, 4), (3, 2)]
```

مثال: یک لیست بنویسیم که هر عنصرش به ترتیب یکی از ارقام عدد پی باشد. (تا ۴ رقم اعشار)

```
# روش نرمال
from math import pi

s = []
for i in range(5):
    s.append(str(round(pi, i)))
print(s) # -> ['3.0', '3.1', '3.14', '3.142', '3.1416']
# -----
# روش خلاصه سازی

x = [str(round(pi, i)) for i in range(5)]
print(x) # -> ['3.0', '3.1', '3.14', '3.142', '3.1416']
```

مثال: در یک ماتریس جای سطرها و ستونها را عوض کنیم.

```
# روش نرمال
matrix = [
    [1, 2, 3, 4],
    [5, 6, 7, 8],
    [9, 10, 11, 12]
]

t = []
for i in range(4):
    t_row = []
    for row in matrix:
        t_row.append(row[i])
    t.append(t_row)

for j in t:
    print(j)
# -----
# روش خلاصه سازی
print('-' * 20)
b = ([row[i] for row in matrix] for i in range(4))

for j in b:
    print(j)
```

```
[1, 5, 9]
[2, 6, 10]
[3, 7, 11]
[4, 8, 12]
-----
[1, 5, 9]
[2, 6, 10]
[3, 7, 11]
[4, 8, 12]
```

درس ۸: خلاصه سازی یا comprehension (بخش دوم)

* برای ساخت **ژنراتور** نیز می‌توانیم از خلاصه سازی استفاده کنیم (از پرانتز به جای **براکت** استفاده می‌کنیم).

```
c = (x for x in (1, 2, 3))
print(next(c)) # -> 1
print(next(c)) # -> 2
print(next(c)) # -> 3
```

* نباید خلاصه سازی بیهوده انجام بدھیم.

* برای مدیریت خطأ و استثنای بهتر است از خلاصه استفاده نکنیم.

* برای ساخت **مجموعه** نیز می‌توانیم از خلاصه سازی استفاده کنیم (از آکلاد **{}** به جای **براکت** استفاده می‌کنیم).

```
s = {i for i in range(10) if i % 2 != 0}
print(s) # -> {1, 3, 5, 7, 9}
```

* برای ساخت **دیکشنری** نیز می‌توانیم از خلاصه سازی استفاده کنیم (از آکلاد **{}** به جای **براکت** استفاده می‌کنیم).

```
d = {'reza': 25, 'ali': 30}
s = {key: value for key, value in d}
print(s) # -> {'ali': 30, 'reza': 25}
```

مثال: یک لیست بنویسید که اعداد فرد از یک لیست را بردارد و به جای اعداد زوج صفر بگذارد.

```
# روش نرمال
x = [1, 2, 5, 4, 7, 8, 10]
y = []

for i in x:
    y.append(i if i % 2 != 0 else 0)

print(y) # -> [1, 0, 5, 0, 7, 0, 0]
# -----
# روش خلاصه سازی

x = [1, 2, 5, 4, 7, 8, 10]
z = [i if i % 2 != 0 else 0 for i in x]
print(z) # -> [1, 0, 5, 0, 7, 0, 0]
```

مثال: یک لیست بنویسیم که اعداد بزرگتر از ۱۰۰ را از یک تابع رندوم نشان دهد.

```
from random import randrange

def func():
    return randrange(50, 150)

x = [c for _ in range(10) if (c := func()) > 100]
print(x)
```

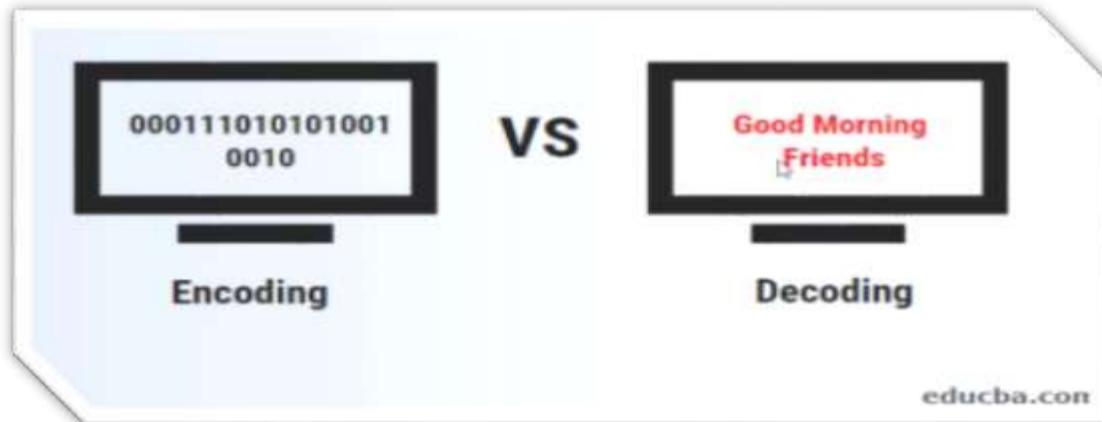
مثال: یک لیست از اسامی دانشجو داریم، یک دیکشنری بنویسید که نمره‌ی آن‌ها به صورت پیش‌فرض صفر باشد.

```
names = ['reza', 'ali', 'neda']
d = {name: [0 for _ in range(3)] for name in names}
print(d) # -> {'reza': [0, 0, 0], 'ali': [0, 0, 0], 'neda': [0, 0, 0]}
```

درس ۹: انکدینگ، دیکدینگ، اسکی و یونیکد

- * به جایگزین کردن کاراکترها و رشته‌های مورد استفاده‌ی ما با کاراکترها و رشته‌هایی که انتقال دادن آن‌ها یا ذخیره سازی آن‌ها برای یک سیستم بهینه باشد **انکدینگ** می‌گوییم و به تغییر دوباره این فرمت به همان کاراکترها و رشته‌های اولیه **دیکدینگ** می‌گوییم.
- * **انکدینگ** باید قابل برگشت باشد، یعنی اگر یک متن را **انکد** کردیم باید بتولاینم آن را **دیکد** کنیم.
- * برای **دیکد** نباید نیاز به کلید داشته باشیم و با فهمیدن ساختار **انکدینگ** به راحتی بتوان آن را **دیکد** کرد.
- * برای این‌که بتولاینم حروف و نمادها را در سیستم ذخیره کنیم به آن‌ها یک عدد نسبت داده اند که البته استانداردهای متفاوتی وجود دارد که ممکن است مطابق با آن استاندارد عدد نسبت داده شده متفاوت باشد.

```
print(ord("A")) # -> 65
```



- * از **اسکی (ASCII)** می‌توانیم برای نمایش ۱۲۸ کاراکتر استفاده کنیم.
- * چون **اسکی** محدود بود استاندارد **یونیکد (Unicode)** ساخته شد، یونیکد محدود به یک بایت نیست و می‌تواند تعداد بایت‌های مختلفی را داشته باشد.
- * **پایتون** از یونیکد استفاده می‌کند و همه‌ی کاراکترهای همه‌ی زبان‌ها را پشتیبانی می‌کند.
- * **یونیکد** دارای استانداردهای متفاوتی مثل **UTF-8**, **UTF-16**, **... و ...** می‌باشد.
- * از دستور **ord** برای نمایش یونیکد یک کاراکتر استفاده می‌کنیم.

```
print(ord("]")) # -> 93
```

- * از دستور **chr** برای نمایش کاراکتر یک یونیکد استفاده می‌کنیم.

```
print(chr(7252)) # -> ⚡
```

درس ۱۰: تکالیف مبحث انکدینگ، دیکدینگ، اسکی و یونیکد

۱. رشته "Hello, World" را encode کنید (با استفاده از انکدینگ UTF-8) و آن را چاپ کنید.

۲. بایت‌های زیر را با استفاده از UTF-8 دیکد (decode) و رشته decode شده را چاپ کنید.

b'My name is M\xc3\xb6bius'

۳. لیستی از کاراکترها داریم، هر کاراکتر را به مقدار ASCII مربوطه تبدیل کنید و نتایج را در یک لیست جدید ذخیره کنید.

۴. کاراکترهای یک جمله را از کاربر بگیرید و سپس مجموع مقادیر ASCII تمام کاراکترهای جمله را محاسبه و چاپ کنید.

۵. لیست زیر شامل Unicode code point آن‌ها است. با استفاده از انکدینگ UTF-16 آن‌ها را به بایت encode کنید و نتیجه را چاپ کنید.

[1024, 5678, 234, 987]

۶. بایت‌های زیر را با استفاده از انکدینگ UTF-16 دیکد کنید و رشته decode شده را چاپ کنید.

b'\xff\xfeM\x00y\x00\x00N\x00a\x00m\x00e\x00'

۷. تابعی بنویسید که رشته‌ای را به عنوان ورودی می‌گیرد و نمایش هگزادسیمال بایت‌های انکد شده UTF-8 را بر می‌گرداند.

درس ۱۱: نوع داده بایت (bytes و bytearray) (بخش اول)

در پایتون سه نوع رشته داریم:

- رشته‌های معمولی (اسکی و یونیکد) (str)
- داده‌های باینری (bytes)
- داده‌های باینری تغییرپذیر (bytearray)

* اگر قبل از رشته از **b** استفاده کنیم بایت‌های آن رشته را نشان می‌دهد و آن رشته تبدیل به بایت می‌شود.

```
name = b'reza'  
print(name) # -> b'reza'  
print(type(name)) # -> <class 'bytes'>
```

* بایت فقط می‌تواند شامل کاراکترهای اسکی باشد.

```
name = b'reza\x' # SyntaxError  
print(name)
```

```
File "F:\python\pythonProject\test.py", line 1  
  name = b'reza\x'  
          ^^^^^^  
SyntaxError: bytes can only contain ASCII literal characters
```

* برای حل این مشکل می‌توانیم از دستور **bytes** استفاده کنیم که دو ورودی می‌گیرد، ورودی اول رشته و ورودی دوم نوع اینکدینگ می‌باشد و به جای نشان دادن کاراکتر، یونیکد آن را بر مبنای ۱۶ نمایش می‌دهد.

```
name = bytes('reza\x', 'utf-8')  
print(name) # -> b'reza\xca\xac'  
print(type(name)) # -> <class 'bytes'>
```

درس ۱۲: نوع داده بایت (bytes و bytearray) (بخش دوم)

* برای تبدیل داده‌های عددی به بایت بهتر است آن داده‌ها را در **لیست** قرار دهیم.

```
numbers = bytes([1, 7, 9, 56, 250])
print(numbers) # -> b'\x01\x07\x09\x38\xfa
print(type(numbers)) # -> <class 'bytes'>
```

* اگر داده‌های عددی را در لیست قرار ندهیم، تبدیل به بایت نمی‌کند بلکه به تعداد آن عدد بایت **0** نشان می‌دهد.

```
numbers = bytes(6)
print(numbers) # -> b'\x00\x00\x00\x00\x00\x00'
print(type(numbers)) # -> <class 'bytes'>
```

* برای این‌که خود کاراکتر را به جای یونیکد آن نمایش دهد از **decode** استفاده می‌کنیم که با این‌کار بایت تبدیل به رشته می‌شود.

```
name = bytes('reza\u062f', 'utf-8')
print(name) # -> b'reza\xe0\xxa4\x92'
print(type(name)) # -> <class 'bytes'>

print(name.decode()) # -> reza\u062f
print(type(name.decode())) # -> <class 'str'>
```

* برای تبدیل بایت به رشته علاوه بر **decode** می‌توانیم از **str** نیز استفاده کنیم.

```
name = bytes('reza\u062f', 'utf-8')
print(str(name, 'utf-8')) # -> reza\u062f
print(type(str(name, 'utf-8'))) # -> <class 'str'>
```

* برای تبدیل رشته به بایت علاوه بر **b** و **byte** می‌توانیم از **encode** استفاده کنیم.

```
name = 'reza\u062f'.encode('utf-8')
print(name) # -> b'reza\xe0\xxa4\x92'
print(type(name)) # -> <class 'bytes'>
```

* بایت‌ها تغییر پذیر نیستند.

```
name = b'reza'
print(name[0])
name[0] = 120
```

```
114
Traceback (most recent call last):
  File "F:\python\pythonProject\test.py", line 3, in <module>
    name[0] = 120
    ~~~~^~~^
TypeError: 'bytes' object does not support item assignment
```

* برای این که بایت تغییر پذیر شود از **bytearray** استفاده می کنیم.

```
name = bytearray('reza', 'utf-8')
print(name) # -> bytearray(b'reza')
print(name[0]) # -> 114
name[0] = 126
print(name) # -> bytearray(b'~eza')
print(name[0]) # -> 126
print(type(name)) # -> <class 'bytearray'>
```

* دستور **bytes** علاوه بر رشته و نوع اینکدینگ یک ورودی دیگر می گیرد که مشخص می کند اگر در تبدیل یک کاراکتر به بایت دچار خطا شدیم، چه چیزی را به جایش نشان دهد.

```
name = bytes('rezaಇ', 'ascii')
print(name)
```

```
Traceback (most recent call last):
  File "F:\python\pythonProject\test.py", line 1, in <module>
    name = bytes('rezaಇ', 'ascii')
           ^^^^^^^^^^^^^^^^^^^^^^^^^^
UnicodeEncodeError: 'ascii' codec can't encode character '\u0c88' in position 4: ordinal not in range(128)
```

* اگر از **strict** استفاده کنیم همان خطا را می دهد.

```
Traceback (most recent call last):
  File "F:\python\pythonProject\test.py", line 1, in <module>
    name = bytes('rezaಇ', 'ascii', 'strict')
           ^^^^^^^^^^^^^^^^^^^^^^^^^^
UnicodeEncodeError: 'ascii' codec can't encode character '\u0c88' in position 4: ordinal not in range(128)
```

* اگر از **ignore** استفاده کنیم کاراکتر را نادیده می گیرد.

```
name = bytes('rezaಇ', 'ascii', 'ignore')
print(name) # -> b'reza'
```

* اگر از **replace** استفاده کنیم کاراکتر را با علامت سوال (?) جایگزین می کند.

```
name = bytes('rezaಇ', 'ascii', 'replace')
print(name) # -> b'reza?'
```

* اگر از **backslashreplace** استفاده کنیم کاراکتر را با بک اسلش (\) و یونیکدش **?** جایگزین می کند.

```
name = bytes('rezaಇ', 'ascii', 'backslashreplace')
print(name) # -> b'reza\\u0c88'
```

* اگر از **namereplace** استفاده کنیم اسم کاراکتر را جایگزین می کند.

```
name = bytes('rezaಇ', 'ascii', 'namereplace')
print(name) # -> b'reza\\N{KANNADA LETTER II}'
```

درس ۱۳: تکالیف مبحث نوع داده `bytes` و `bytearray`

۱. یک شی بایت ایجاد کنید و نمایش هگزادسیمال آن را چاپ کنید.
 ۲. یک رشته هگزا دسیمال را به `bytearray` تبدیل کنید.
 ۳. یک `bytearray` ایجاد کنید و عناصر آن را ویرایش کنید.
 ۴. دو شی `byte` را به هم متصل کنید.
 ۵. بررسی کنید آیا مقدار `byte` خاصی (مثلًا 20) در `bytearray` وجود دارد یا خیر.
 ۶. یک `bytearray` را به لیستی از اعداد صحیح تبدیل کنید.
 ۷. ایندکس مقدار یک `byte` خاص را در `bytearray` پیدا کنید.
-

درس ۱۴: متدهای رشته (بخش اول)

* از متدهای تبدیل حروف انگلیسی بزرگ رشته به حروف کوچک استفاده می‌کنیم.

```
s = 'ReZa DoLaTi'  
print(s.casefold()) # -> reza dolati
```

* متدهای `casefold` از قدرت و نفوذ بیشتری نسبت به `lower` برخوردار است.

```
s = 'ß'  
print(s.lower()) # -> ß  
print(s.casefold()) # -> ss
```

* از متدهای `center` برای مرتب کردن یک متن استفاده می‌شود، ابتدا طول رشته را مشخص می‌کنیم و سپس رشته را در وسط قرار می‌دهد و قبل و بعد آن فاصله می‌گذارد.

* اگر از `repr` استفاده کنیم رشته را داخل کتیشن (') قرار می‌دهد.

```
s = 'reza'  
print(repr(s.center(10))) # -> '     reza     '
```

* می‌توانیم مشخص کنیم که قبل و بعد رشته چه کاراکتری قرار گیرد، به این صورت که بعد از طول رشته آن کاراکتر را بنویسیم.

```
s = 'reza'  
print(s.center(10, '+')) # -> +++reza+++
```

* متدهای `expandtabs` تب های رشته را با فاصله جایگزین می‌کند، می‌توانیم مشخص کنیم چند فاصله بگذاریم.

```
s = 'reza\t\dolati'  
print(s) # -> reza      dolati  
print(s.expandtabs(12)) # -> reza              dolati
```

* با متدهای `format_map` می‌توانیم به مقدار کلیدهای دیکشنری در رشته مان دست یابیم.

```
x = {'x': 'john', 'y': 'smith'}  
print('my name is {x} {y}'.format_map(x)) # -> my name is john smith
```

* متدهای `index` و `rindex` یک کاراکتر را در رشته مشخص می‌کنند و برای جستجو از سمت راست رشته از `index` استفاده می‌کنیم.

```
s = 'reza dolati'  
print(s.index('a')) # -> 3  
print(s.rindex('a')) # -> 8
```

* اگر متدهای `index` یک کاراکتر را پیدا نکند خطا می‌دهد اما `find` خطا نمی‌دهد بلکه `-1` را نشان می‌دهد.

```
s = 'reza'  
print(s.find('y'))  
print(s.index('y'))
```

```
-1  
Traceback (most recent call last):  
  File "F:\python\pythonProject\test.py", line 3, in <module>  
    print(s.index('y')) # -> ValueError: substring not found  
    ^^^^^^^^^^^^^^  
ValueError: substring not found
```

* برای مشخص کردن این که همه کاراکترهای رشته حروف الفبا هستند یا خیر از متدهای `isalpha` استفاده می‌کنیم.

```
s = 'reza'  
print(s.isalpha()) # -> True  
m = 'reza/'  
print(m.isalpha()) # -> False
```

* برای مشخص کردن این که همه کاراکترهای رشته اسکی هستند یا خیر از متدهای `isascii` استفاده می‌کنیم.

```
s = 'reza'  
print(s.isascii()) # -> True  
m = 'رم'  
print(m.isascii()) # -> False
```

* برای مشخص کردن این که همه کاراکترهای رشته عدد هستند یا خیر از متدهای `isdigit` و `isnumeric` و `isdecimal` استفاده می‌کنیم.

```
s = '3422354'  
print(s.isdecimal()) # -> True  
print(s.isnumeric()) # -> True  
print(s.isdigit()) # -> True
```

* برای مشخص کردن این که اسم یک متغیر قابل استفاده هست یا خیر از متدهای `isidentifier` استفاده می‌کنیم.

```
s = 'x_66'  
m = '= 66'  
print(s.isidentifier()) # -> True  
print(m.isidentifier()) # -> False
```

* برای مشخص کردن این که آیا همه کاراکترهای رشته حرف کوچک هستند یا خیر از متدهای `islower` استفاده می‌کنیم.

```
s = 'reza'  
m = 'rEza'  
print(s.islower()) # -> True  
print(m.islower()) # -> False
```

* برای مشخص کردن این که آیا همهی کاراکترهای رشته قابل چاپ هستند یا خیر از متده استفاده می کنیم.

```
s = 'reza'
m = 'reza\t'
print(s.isprintable()) # -> True
print(m.isprintable()) # -> False
```

* برای مشخص کردن این که آیا همهی کاراکترهای رشته فاصله هستند یا خیر از متده استفاده می کنیم.

```
s = '\t\n\n'
m = 'reza '
print(s.isspace()) # -> True
print(m.isspace()) # -> False
```

* متده حرف اول همهی کلمات رشته را به حرف بزرگ تبدیل می کند و بقیهی حرفها را به حرف کوچک.

```
s = 'tHis Is a TexT'
print(s.title()) # -> This Is A Text
```

* برای مشخص کردن این که حرف اول همهی کلمات داخل رشته حرف بزرگ و بقیهی حرفها حرف کوچک هستند یا خیر از متده استفاده می کنیم.

```
s = 'This Is A Text'
m = 'tHis Is a TexT'
print(s.istitle()) # -> True
print(m.istitle()) # -> False
```

* برای مشخص کردن این که همهی حروف رشته، حرف بزرگ هستند یا خیر از متده استفاده می کنیم.

```
s = 'THIS IS A TEXT'
m = 'This is a Text'
print(s.isupper()) # -> True
print(m.isupper()) # -> False
```

درس ۱۵: متدهای رشته (بخش دوم)

* از متدهای **ljust** برای مرتب کردن یک متن استفاده می‌شود، ابتدا طول رشته را مشخص می‌کنیم و سپس رشته را در سمت چپ قرار می‌دهد و بعد از آن فاصله می‌گذارد.

* اگر از **repr** استفاده کنیم رشته را داخل کتیشن (') قرار می‌دهد.

```
s = 'reza'  
print(repr(s.ljust(10))) # -> 'reza'           '
```

* می‌توانیم مشخص کنیم که بعد رشته چه کاراکتری قرار گیرد، به این صورت که بعد از طول رشته آن کاراکتر را بنویسیم.

```
s = 'reza'  
print(s.ljust(10, '+')) # -> reza++++++
```

* متدهای **rjust** همانند عمل می‌کند اما رشته را در سمت راست و فاصله و یا کاراکتری که تعیین کردیم را قبل از رشته قرار می‌دهد.

```
s = 'reza'  
print(s.rjust(10, '+')) # -> ++++++reza
```

* متدهای **maketrans** یونیکد کاراکترهای مورد نظرمان را نشان می‌دهد و کاراکتر اول جایجا می‌کند و کاراکتر سوم را حذف می‌کند و برای اعمال این تغییرات از متدهای **translate** استفاده می‌کنیم و به عنوان ورودی به آن **maketrans** را می‌دهیم.

```
s = 'rezab'  
table = str.maketrans('b', 'β', 'a')  
print(s) # -> rezab  
print(table) # -> {98: 223, 97: None}  
print(s.translate(table)) # -> rezβ
```

* در متدهای **maketrans** می‌توانیم به عنوان ورودی دیکشنری بدهیم.

```
d = {'e': '-', 'a': '+', 'r': None}  
s = 'reza'  
table = str.maketrans(d)  
print(s.translate(table)) # -> -z+
```

* متدهای **partition** یک رشته را با توجه به رشته‌ای که به این متدهایی که به سه قسمت قبل و خود این رشته و بعد آن تقسیم می‌کند و اگر بخواهیم از آخر دنبال رشته‌ی مورد نظرمان بگردد از **rpartition** استفاده می‌کنیم.

```
s = 'this is a text'  
print(s.partition('s')) # -> ('thi', 's', ' is a text')  
print(s.rpartition('s')) # -> ('this i', 's', ' a text')
```

* متدهای **removeprefix** از اول رشته، رشته‌ی مورد نظر ما را حذف می‌کند.

* متدهای **removesuffix** از آخر رشته، رشته‌ی مورد نظر ما را حذف می‌کند.

```
s = '++this is a text++'  
print(s.removeprefix('++')) # -> +this is a text++  
print(s.removesuffix('++')) # -> ++this is a text
```

* تفاوت متدهای `lstrip` و `rstrip` با متدهای `lstrip` و `removeprefix` و `removesuffix` و `lstrip` تعداد مهم نیست.

```
s = '++this is a text++'
print(s.removeprefix('++')) # -> +this is a text++
print(s.removesuffix('++')) # -> ++this is a text
print(s.lstrip('+')) # -> this is a text++
print(s.rstrip('+')) # -> ++this is a text
```

* متدهای `lstrip` و `rstrip` یک لیست می‌سازد که عناصرش را بر اساس ورودی که می‌دهیم جدا می‌کند.

```
s = 'reza,ali,mohsen'
print(s.split(',')) # -> ['reza', 'ali', 'mohsen']
```

* متدهای `rsplit` جداسازی را از سمت راست انجام می‌دهد.

```
s = 'reza,ali,mohsen'
print(s.split(',', 1)) # -> ['reza', 'ali,mohsen']
print(s.rsplit(',', 1)) # -> ['reza,ali', 'mohsen']
```

* متدهای `splitlines` یک لیست می‌سازد که عناصر را بر اساس \n جدا می‌کند، اگر مقدار `True` بتوانیم \n را هم نشان می‌دهد.

```
s = 'reza\nali\nmohsen'
print(s.splitlines()) # -> ['reza', 'ali', 'mohsen']
print(s.splitlines(True)) # -> ['reza\n', 'ali\n', 'mohsen']
```

* متدهای `swapcase` تمام حروف کوچک را به حروف بزرگ و تمام حروف بزرگ را به حروف کوچک تبدیل می‌کند.

```
s = 'reza - MOHSEN'
print(s.swapcase()) # -> REZA - mohsen
```

* با متدهای `zfill` می‌شود به تعداد دلخواه قبل از رشته ۰ گذاشت.

```
h = '11'
m = '3'
print(h.zfill(2), ':', m.zfill(2)) # -> 11 : 03
```

فصل هشتم: مازول‌ها و بسته‌ها

درس ۱: نیم نگاهی به برنامه نویسی مازول‌لار

* به تقسیم یک پروژه‌ی بزرگ به تسك‌ها و مازول‌های کوچک‌تر جهت مدیریت بهتر، برنامه نویسی مازول‌لار می‌گویند.

درس ۲: مفهوم اسکریپت، مازول، لایبرری و فریمورک

* به همه‌ی فایل‌هایی با فرمت **py**. که داخلش کدهایی می‌نویسیم و قرار است **مستقیماً** اجرا شود و برای ما کارهایی انجام دهد، **اسکریپت** (script) می‌گوییم.

```
def ab():
    print('hi')

ab() # -> hi
```

* **مازول** (module) همانند اسکریپت یک فایل **py**. است، ولی تفاوتش این است که قرار نیست مستقیماً اجرا شود، بلکه توسط اسکریپت‌ها یا فایل‌های **py**. دیگر **import** و استفاده می‌شود.

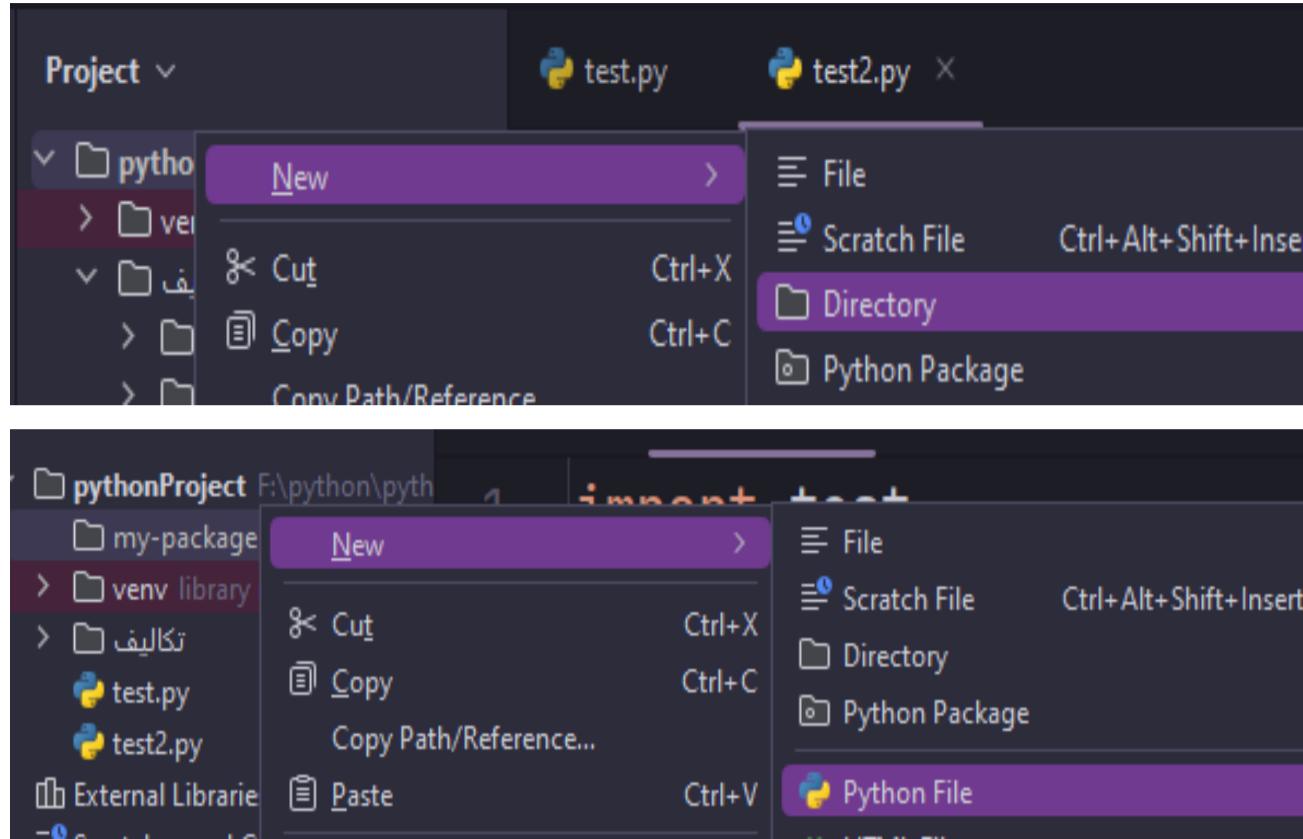
```
import random

def ab():
    print('hi')

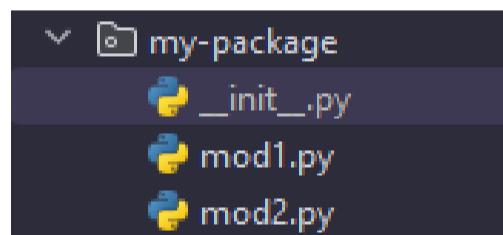
ab()
print(random.randrange(10))
```

* تمام فایل‌های **py**. **مازول** هستند، حتی **اسکریپت‌ها** نیز **مازول** هستند.

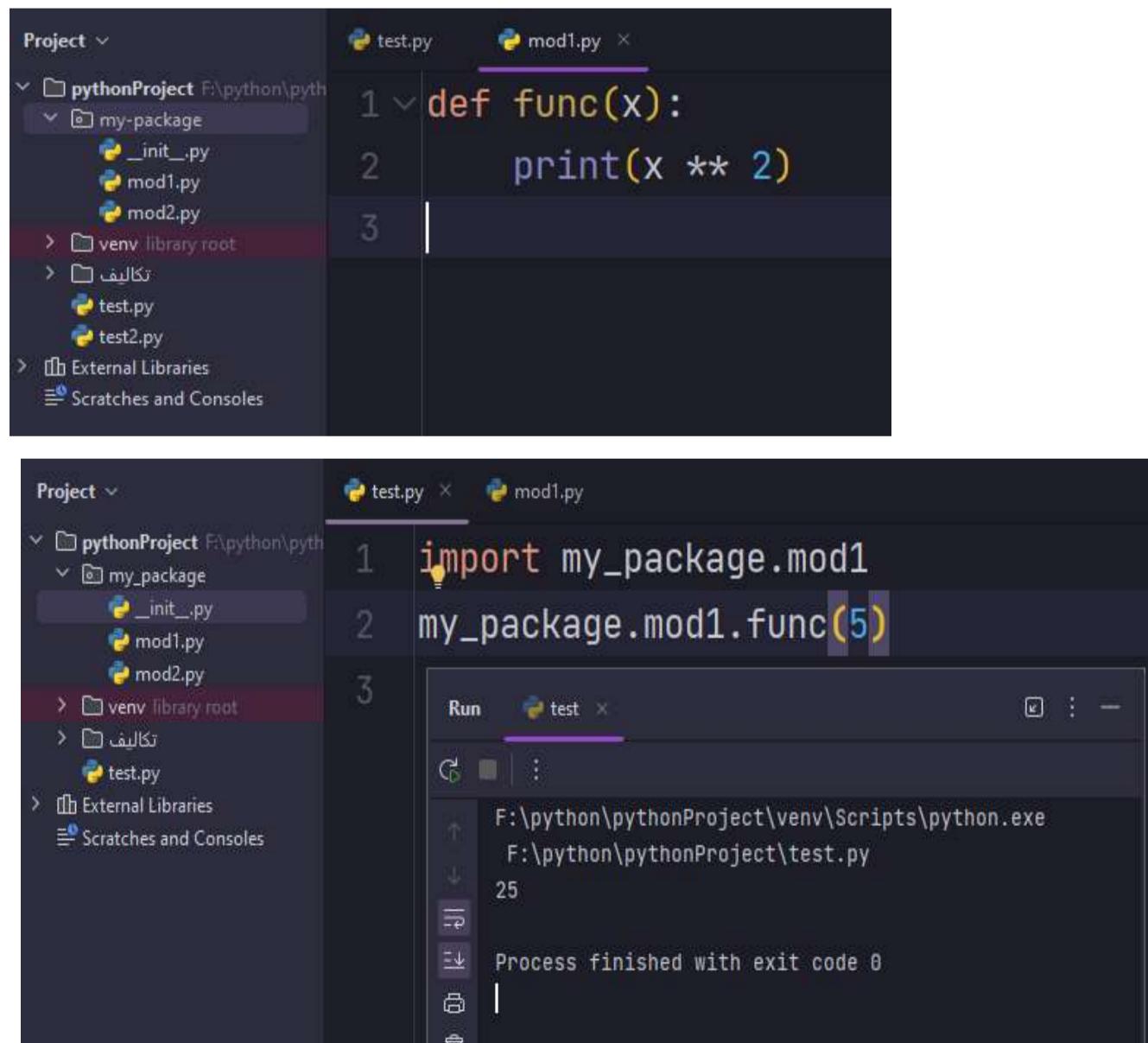
* اگر مجموعه‌ای از ماثول‌های مرتبط باهم که یک وظیفه‌ی خاص را انجام می‌دهند، داخل یک دایرکتوری - پوشه (Directory) جمع‌آوری کنیم، به آن پکیج (package) گفته می‌شود.



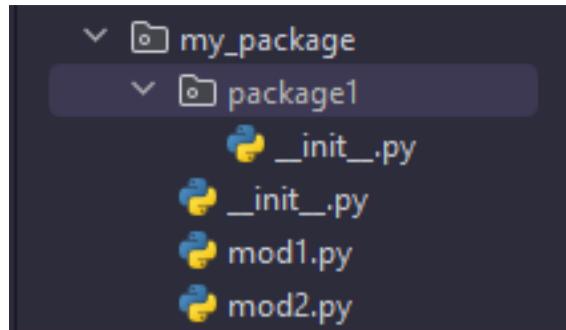
* برای تبدیل این دایرکتوری به پکیج باید یک فایل `py`. دیگر ایجاد کرد با نام `__init__.py` ایجاد کرد.



* برای استفاده از این پکیج به صورت زیر عمل می‌کنیم.



* داخل پکیج می‌توان پکیج‌های دیگر ایجاد کرد.



* به ترکیب چند پکیج باهم **لایبرری** (library) گفته می‌شود.

* در پایتون **لایبرری** و **پکیج** دارای یک مفهوم‌اند، لایبرری بزرگ‌تر از پکیج است.

در پایتون دو نوع مژول داریم:

- مژول‌های عادی و نرمال پایتون با فرمت **.py**. که به آنها **مژول خالص** (pure module) می‌گویند.
- **مژول‌هایی که از زبان‌های دیگر** (...، **c**, **java**, **c#**, ...) در پایتون استفاده می‌کنیم، که به آنها **مژول‌های توسعه** (Extension) می‌گویند.

* به چارچوب و قوانینی که کدهای ما را کنترل می‌کنند **فریمورک** (framework) می‌گویند.

* ما کدهایی را که لازم داریم از **لایبرری** وارد می‌کنیم، اما **فریمورک** کدهای ما را وارد خودش و کنترل می‌کند.

* **فریمورک** می‌تواند شامل پکیج‌ها و لایبرری‌های متعددی باشد.

درس ۳: ساختار پروژه‌ها در پایتون

- * هیچ اجباری برای قرار دادن فایل‌های مختلف در پروژه‌ی ما نیست و می‌توانیم یک پروژه ایجاد کنیم که فقط دارای یک مژول باشد.
- * برای پیکربندی پروژه از فایل‌های `setup.cfg` و `setup.py` استفاده می‌کنیم.
- * توصیف و توضیحات مربوط به پروژه را در فایل `readme` قرار می‌دهیم.
- * برای معرفی کردن فایل‌های غیر پایتونی که در پروژه وجود دارد از فایل `manifest` استفاده می‌کنیم.
- * پیش‌نیازهای پروژه را در فایل `requirements` قرار می‌دهیم.
- * پروانه انتشار پروژه را در فایل `license` قرار می‌دهیم.
- * راهنمای آموزش‌ها و ... در فایل `docs` قرار می‌گیرند.
- * تست‌های پروژه را در فایل `tests` قرار می‌دهیم.

درس ۴: ساخت و استفاده از مژول (بخش اول)

- * برخی از پکیج‌ها و مژول‌ها به صورت پیش‌فرض در پایتون وجود دارند، مانند `datetime`, `random` و ...

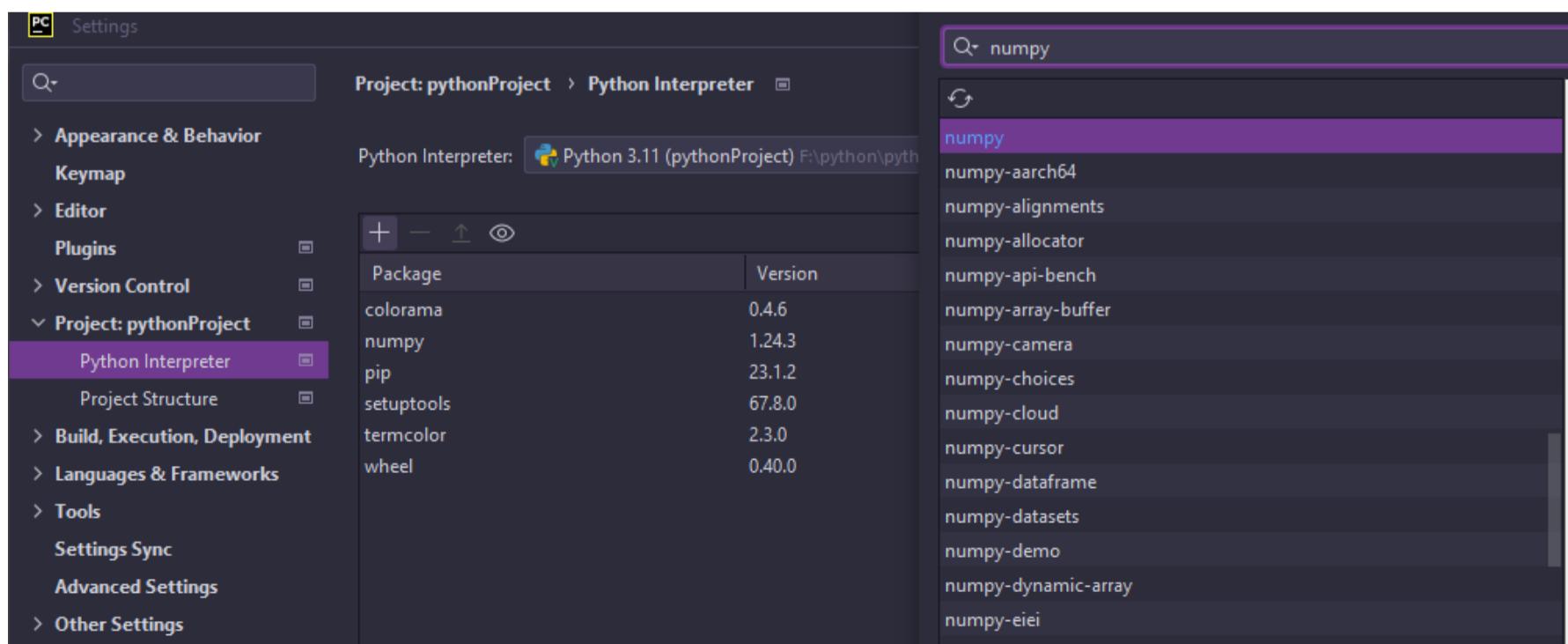
```
import random
import datetime

print(random.randrange(20, 50))
print(datetime.date.today())
```

- * برخی از پکیج‌ها و مژول‌ها توسط برنامه نویسان دیگر نوشته شده‌اند اما کلی نیستند و همه به آن‌ها نیاز ندارند پس به صورت پیش‌فرض در پایتون وجود ندارند و اگر به آن‌ها نیاز داشتیم می‌توانیم آن‌ها را نصب کنیم.
- * برای نصب این مژول‌ها در محیط پایچارم به صورت زیر عمل می‌کنیم.

File -> Settings -> Project: pythonProject -> Python Interpreter

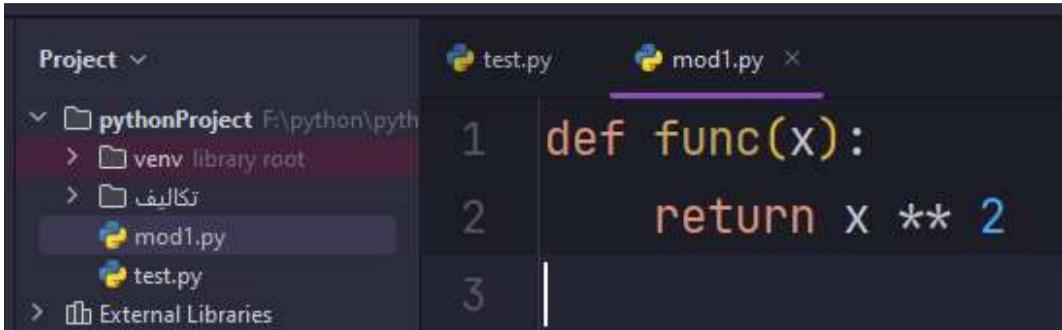
- * پس از وارد شدن به این بخش مژول مورد نظرمان را جستجو و آن را نصب می‌کنیم.



* در محیط‌های غیر از پایچارم می‌توانیم از ترمینال استفاده کنیم.

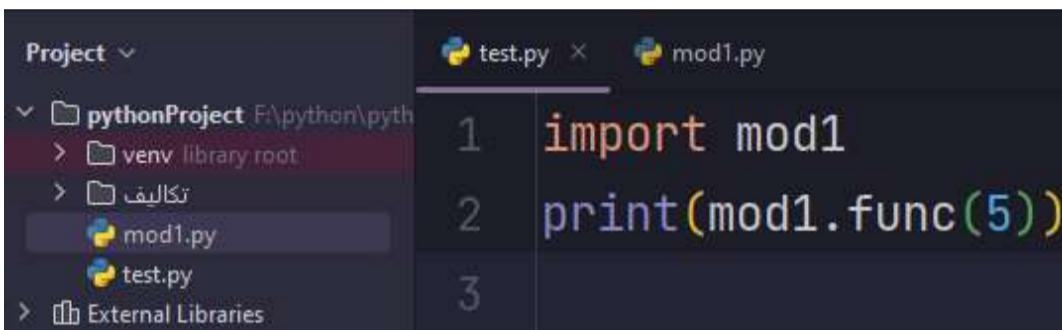
```
(venv) PS F:\python\pythonProject> pip install numpy
```

* می‌توانیم **ماژول اختصاصی** خودمان را بسازیم.



```
Project < PythonProject >
  test.py
  mod1.py
```

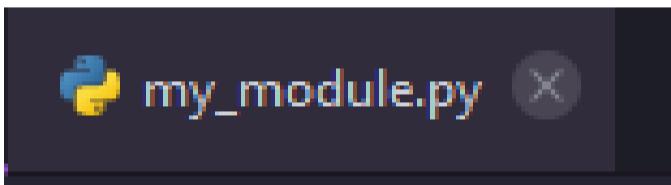
```
def func(x):
    return x ** 2
```



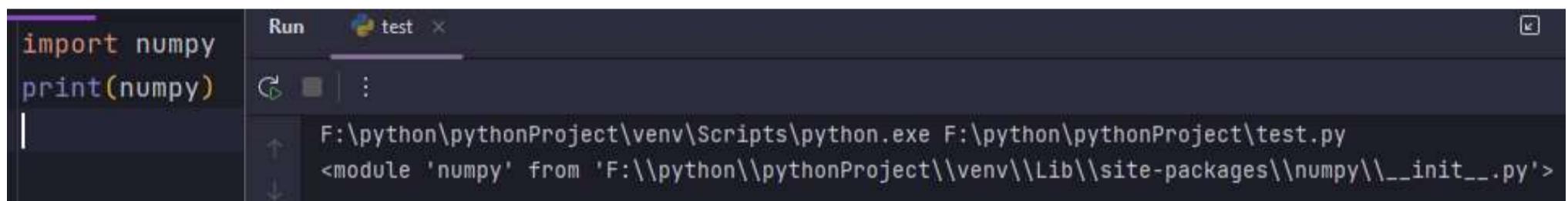
```
Project < PythonProject >
  test.py
  mod1.py
```

```
import mod1
print(mod1.func(5))
```

* اسم ماژول‌ها را باید با **حروف کوچک** بنویسیم و کلمات را با **اندراسکور** (**_**) جدا کنیم.



* با دستور **print** می‌توانیم به آدرس ماژول دست یابیم.



```
import numpy
print(numpy)
```

```
F:\python\pythonProject\venv\Scripts\python.exe F:\python\pythonProject\test.py
<module 'numpy' from 'F:\python\pythonProject\venv\lib\site-packages\numpy\__init__.py'>
```

* می‌توانیم به ماژول **اسم مستعار** بدهیم، به این صورت که بعد از اسم ماژول **as** می‌گذاریم و سپس اسمی که می‌خواهیم را به آن می‌دهیم.

```
import numpy.random as npr
from random import randrange as r

print(npr.rand())
print(r(5, 10))
```

درس ۵: ساخت و استفاده از ماژول (بخش دوم)

* با وارد کردن (`import`) ماژول اسم‌های آن وارد نمی‌شود. برای استفاده از اسم‌های ماژول باید ابتدا ماژول را وارد کنیم، سپس اسم ماژول را بنویسیم و بعد از آن نقطه (.) بگذاریم و سپس اسم مورد نظرمان را بنویسیم.

```
import mod1  
mod1.func2('Ali')
```

* دستور `__name__` اسم ماژول را نشان می‌دهد.

```
print(mod1.__name__) # -> mod1
```

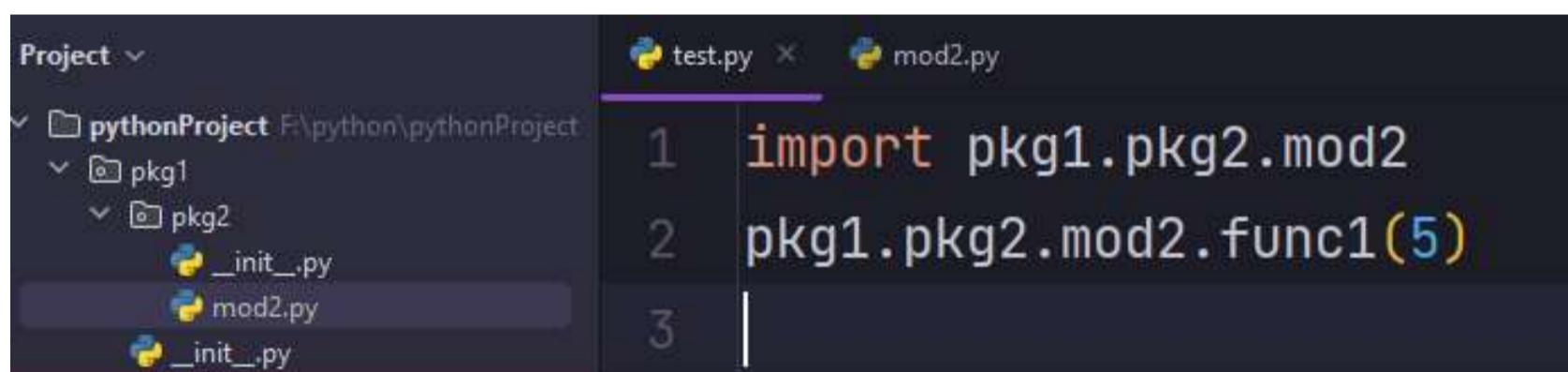
* برای استفاده از اسم‌های یک ماژول می‌توانیم آن‌ها را در یک متغیر ذخیره کنیم.

```
import mod1  
  
f = mod1.func2  
f('ReZA')
```

* ماژول‌ها می‌توانند شامل متغیر نیز باشند.

```
import math  
  
print(math.pi) # -> 3.141592653589793  
print(type(math.pi)) # -> <class 'float'>
```

* اگر ماژول داخل پکیج باشد، با نوشتن **اسم پکیج** و سپس گذاشتن **نقطه (.)** و سپس نوشتن **اسم ماژول** می‌توانیم از آن استفاده کنیم.



* به جای وارد کردن **ماژول** می‌توانیم **مستقیم** اسم‌ها را وارد کنیم.

```
from pkg1.pkg2.mod2 import func1  
from mod1 import func2  
  
func1(5)  
func2('Reza')
```

* می‌توانیم چند اسم همزمان از ماژول مدنظرمان وارد کنیم، به این صورت که پس از هر اسم یک **کاما (,** می‌گذاریم.

```
from mod1 import func2, func1  
  
func1(5)  
func2('Reza')
```

* با گذاشتن **ستاره** (*) به جای اسم‌های مازول می‌توان همه‌ی آن‌ها را وارد کرد.

```
from mod1 import *

func1(5)
func2('Reza')
```

* وقتی مستقیم اسم‌های مازول را وارد می‌کنیم، خود مازول شناخته شده نیست.

```
from mod1 import func2, func1

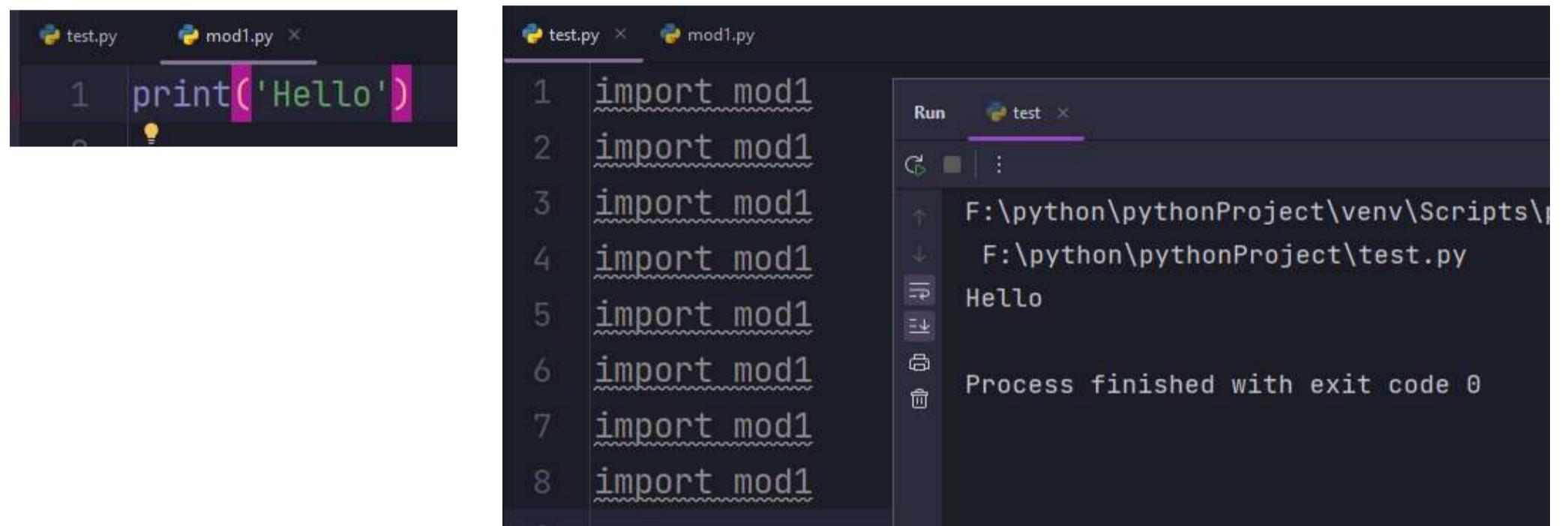
print(mod1) # -> NameError: name 'mod1' is not defined
```

* اگر در مازول اسمی باشد که با **اندراسکور** (_) شروع شده است، در صورت استفاده از **ستاره** آن اسم وارد نمی‌شود.

```
from mod1 import *

print(_age) # -> NameError: name '_age' is not defined.
```

* مازول را هرچقدر وارد کنیم فقط یک بار اجرا می‌شود.



```
test.py
1 print('Hello')

mod1.py
1 import mod1
2 import mod1
3 import mod1
4 import mod1
5 import mod1
6 import mod1
7 import mod1
8 import mod1

Run test
F:\python\pythonProject\venv\Scripts\python.exe F:\python\pythonProject\test.py
Hello
Process finished with exit code 0
```

* برای حل این مشکل می‌توانیم از مازول **reload** و دستور **importlib** استفاده کنیم.

```
test.py
1 import mod1
2 import importlib
3 importlib.reload(mod1)
4
5

Run test
F:\python\pythonProject\venv\Scripts\python.exe F:\python\pythonProject\test.py
Hello
Hello
Process finished with exit code 0
```

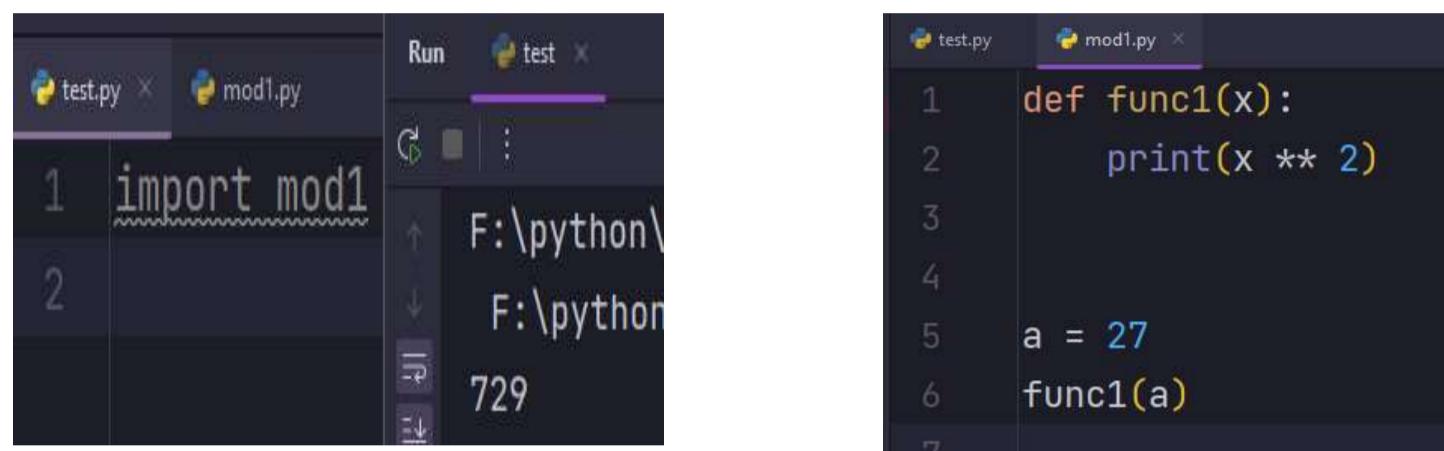
درس ۶: ساخت و استفاده از مژول (بخش سوم)

* اگر از `__name__` داخل خود مژول استفاده کنیم، نتیجه‌ای که نشان می‌دهد اسم مژول نیست بلکه `__main__` است.

```
print(__name__) # -> __main__
```

* و زمانی که این مژول را در فایل دیگر وارد کنیم، نتیجه‌ای که نشان می‌دهد اسم مژول است.

* اگر داخل مژول دستورات قابل اجرا باشند و این مژول را وارد فایل دیگر کنیم، این دستورات نیز اجرا می‌شوند.



```
test.py
1 import mod1
2
mod1.py
1 def func1(x):
2     print(x ** 2)
3
4
5 a = 27
6 func1(a)
7
```

* اگر بخواهیم دستورات داخل خود مژول اجرا شوند اما اگر مژول در فایل دیگر وارد شود اجرا نشوند از شرط در مژول استفاده می‌کنیم که اگر `__name__` برابر با `__main__` بود دستورات اجرا شوند.



```
test.py
1 def func1(x):
2     print(x ** 2)
3
4
5 a = 27
6 if __name__ == '__main__':
7     func1(a)
8
```

* برای استفاده از یک مژول مسیر آن مژول باید شناخته شده باشد، برای مشخص کردن مسیرهای پایتون از مژول `sys` و دستور `path` استفاده می‌کنیم.

```
import sys
print(sys.path)
```

* برای استفاده از یک مژول که مسیرش شناخته شده نیست، با استفاده از روش زیر می‌توانیم مسیر را به پایتون بدهیم.

```
import sys

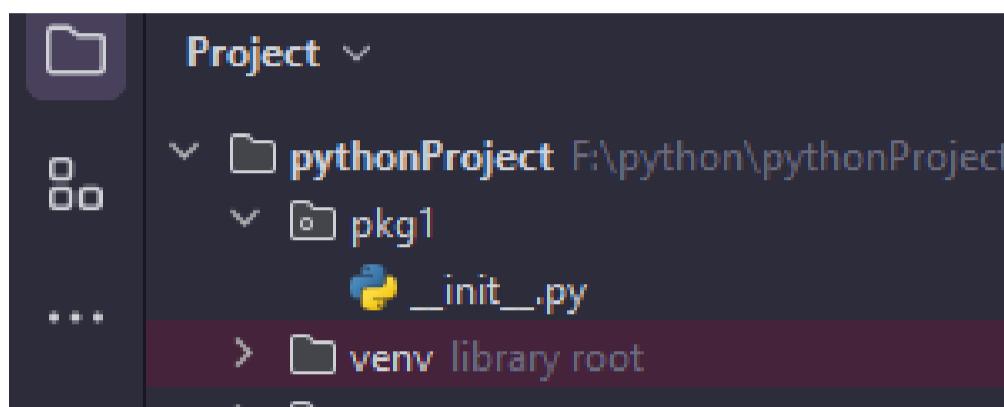
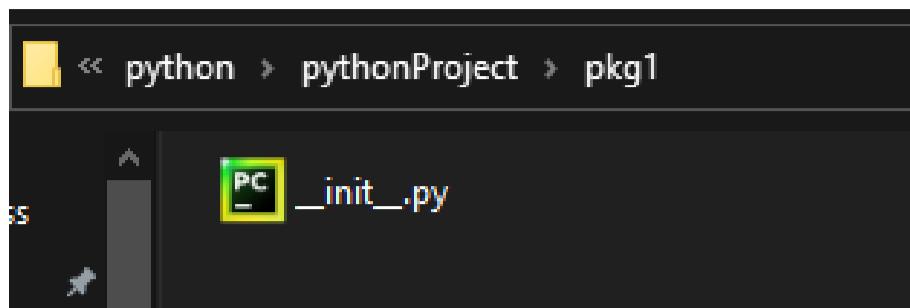
sys.path.append(r'D:\TST')
print(sys.path)
```

* برای مشخص کردن مسیر یک مژول، از دستور `__file__` استفاده می‌کنیم.

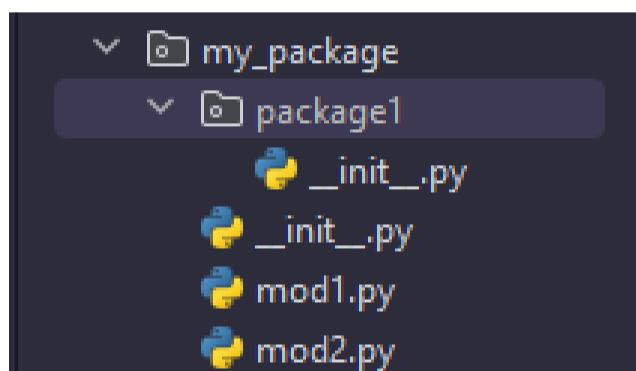
```
import mod1
print(mod1.__file__)
```

درس ۷ : ساخت و استفاده از پکیج (بخش اول)

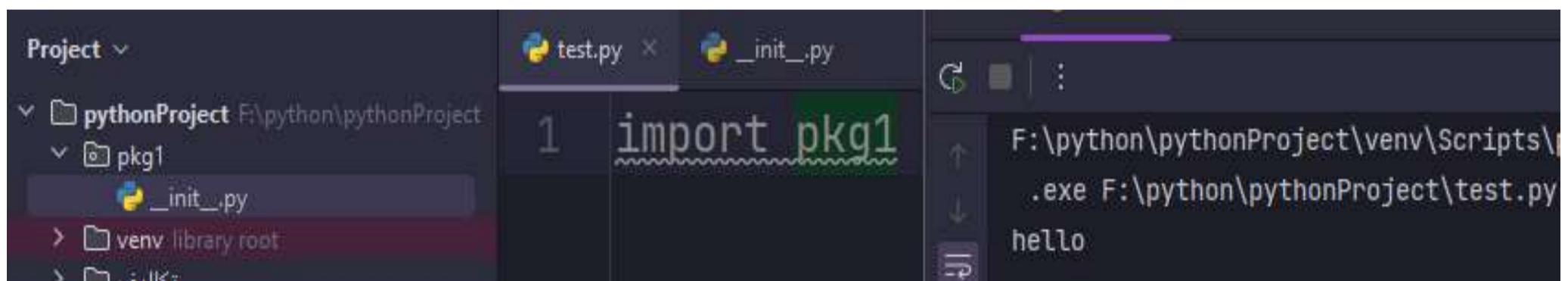
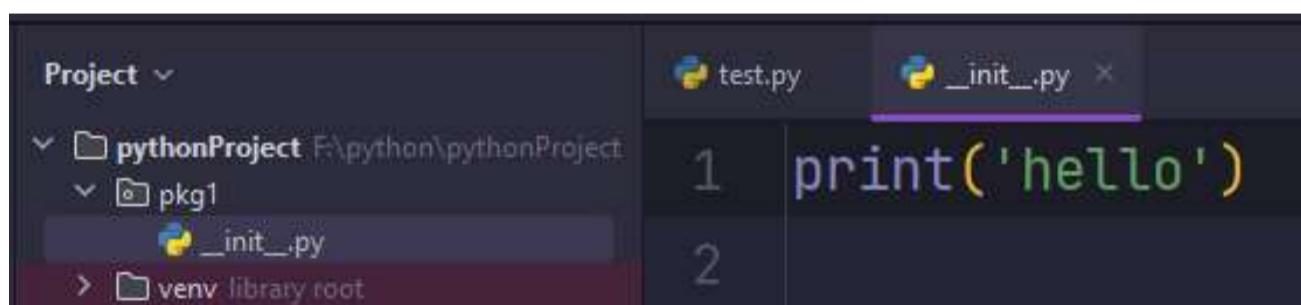
* برای ساخت پکیج باید یک پوشه ایجاد کنیم و در آن نیز یک فایل با نام `__init__.py` ایجاد کنیم.



* داخل پکیج می‌توان پکیج‌های دیگر ایجاد کرد.

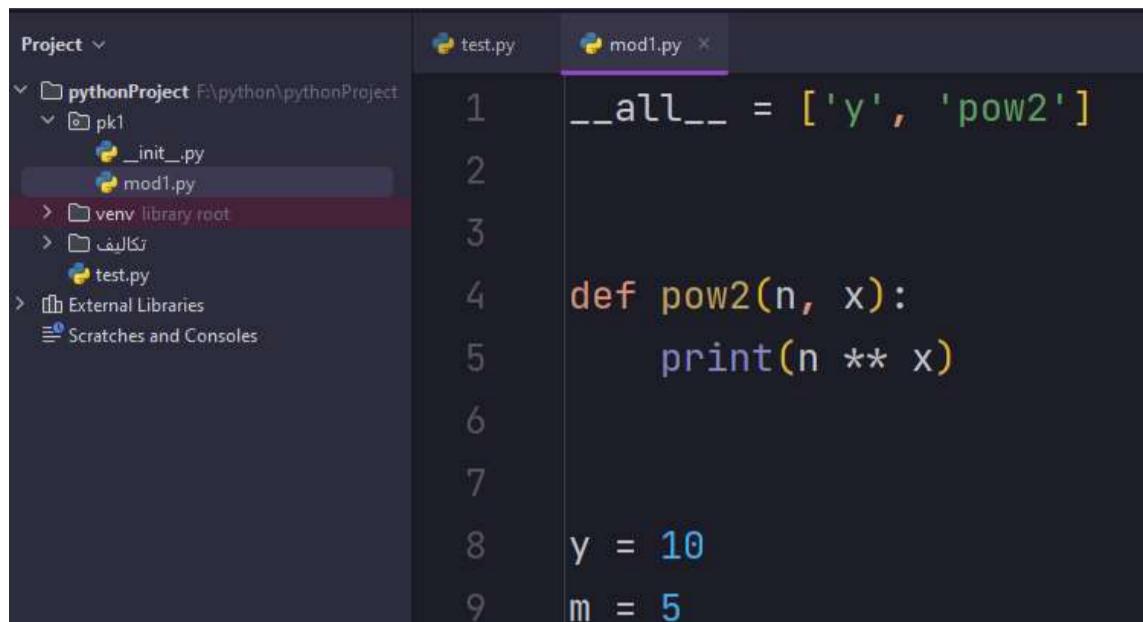


* دستورات داخل فایل `__init__.py` درهنگام وارد شدن پکیج اجرا خواهد شد.

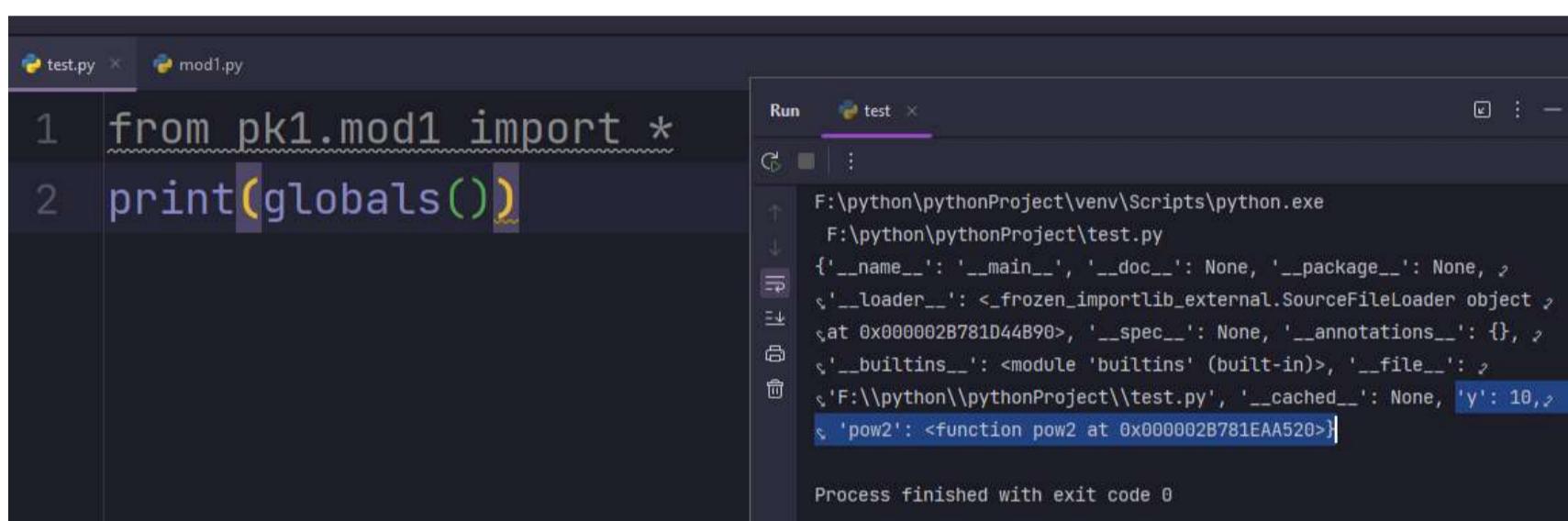


درس ۸: ساخت و استفاده از پکیج (بخش دوم)

* اگر از ستاره (*) استفاده کنیم، همه‌ی اسم‌های داخل مژول وارد می‌شوند، اما اگر بخواهیم فقط اسم‌های مدنظرمان وارد شود می‌توانیم داخل مژول از `__ali` استفاده کنیم و اسم‌های مدنظرمان را داخلش بنویسیم.



```
1 __all__ = ['y', 'pow2']
2
3
4 def pow2(n, x):
5     print(n ** x)
6
7
8 y = 10
9 m = 5
```



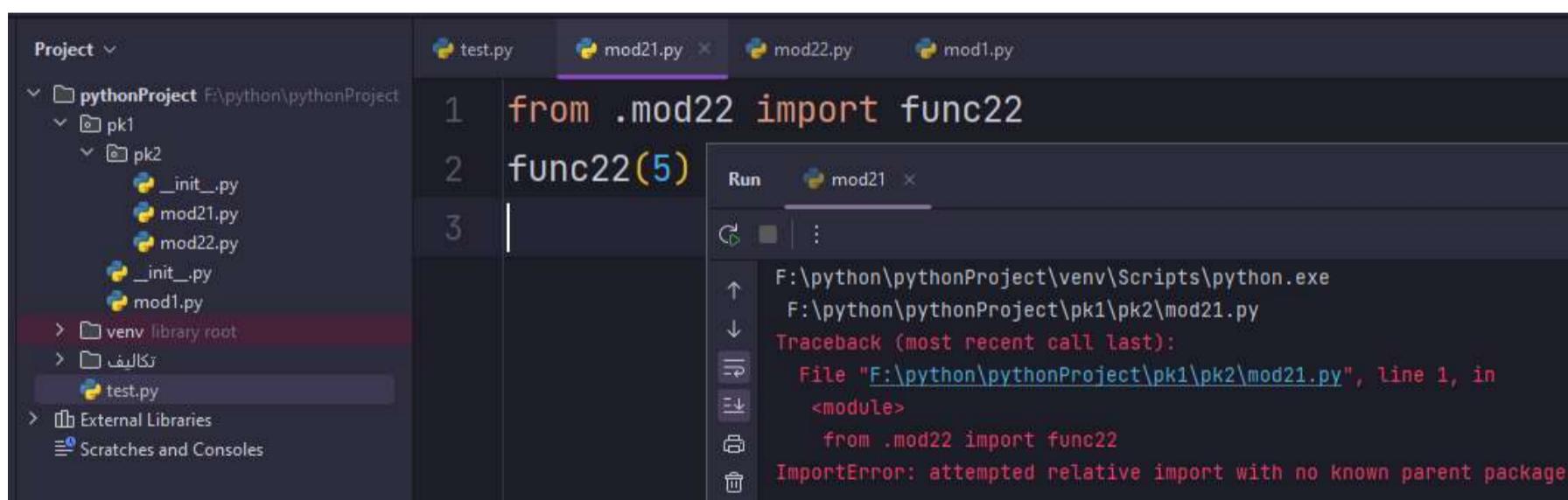
```
1 from pk1.mod1 import *
2 print(globals())
```

Run test

```
F:\python\pythonProject\venv\Scripts\python.exe
F:\python\pythonProject\test.py
{'__name__': '__main__', '__doc__': None, '__package__': None,
 '__loader__': <_frozen_importlib_external.SourceFileLoader object at 0x000002B781D44B90>, '__spec__': None, '__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>, '__file__': 'F:\\python\\pythonProject\\test.py', '__cached__': None, 'y': 10, 'pow2': <function pow2 at 0x000002B781EAA520>}
```

Process finished with exit code 0

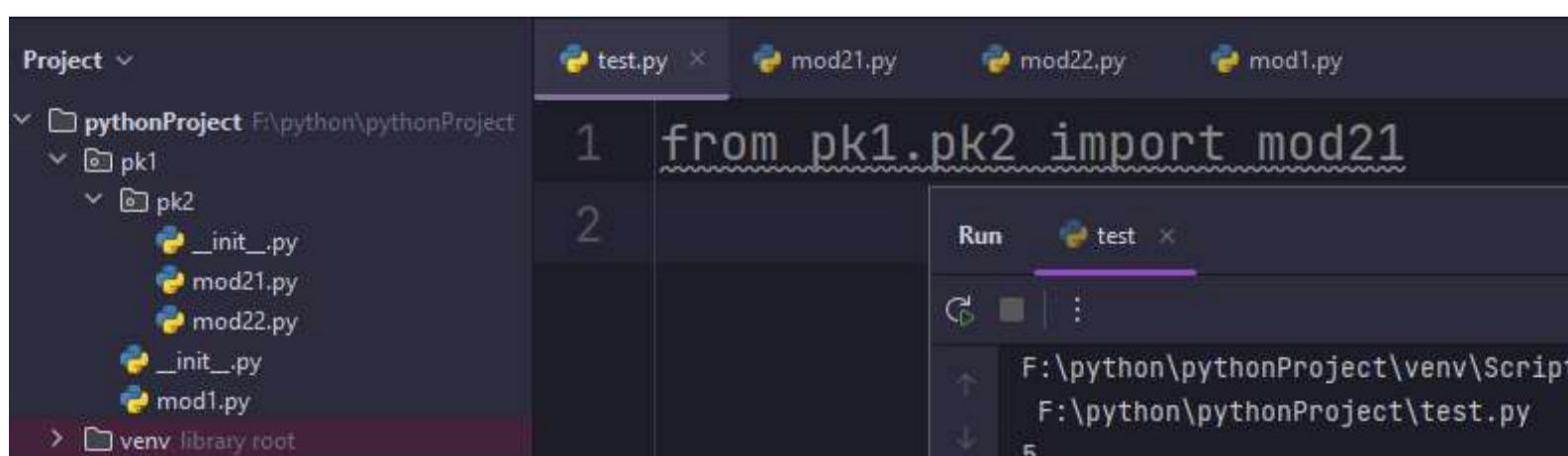
* درهنگام وارد کردن نام مژول اگر از **یک نقطه** (.) استفاده کنیم منظور همان **پکیج فعلی** می‌باشد اگر این دستور را در مژول اجرا کنیم **خطا** می‌دهد و اگر این مژول را در فایلی دیگر استفاده کنیم، بدون خطا اجرا می‌شود.



```
1 from .mod22 import func22
2 func22(5)
```

Run mod21

```
F:\python\pythonProject\venv\Scripts\python.exe
F:\python\pythonProject\pk1\pk2\mod21.py
Traceback (most recent call last):
  File "F:\python\pythonProject\pk1\pk2\mod21.py", line 1, in <module>
    from .mod22 import func22
ImportError: attempted relative import with no known parent package
```

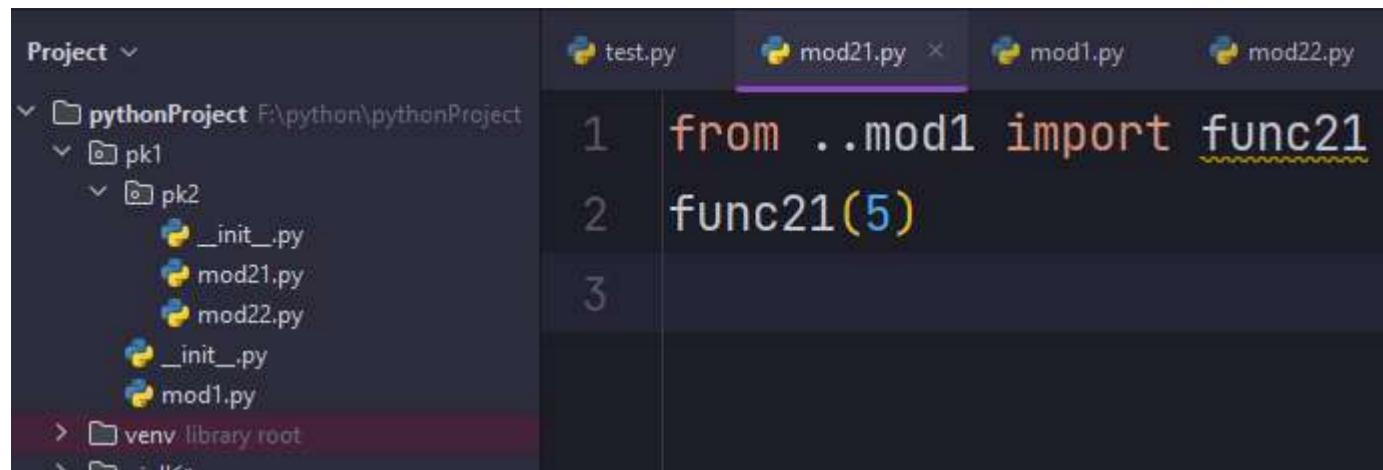


```
1 from pk1.pk2 import mod21
2
```

Run test

```
F:\python\pythonProject\venv\Scripts\python.exe
F:\python\pythonProject\test.py
5
```

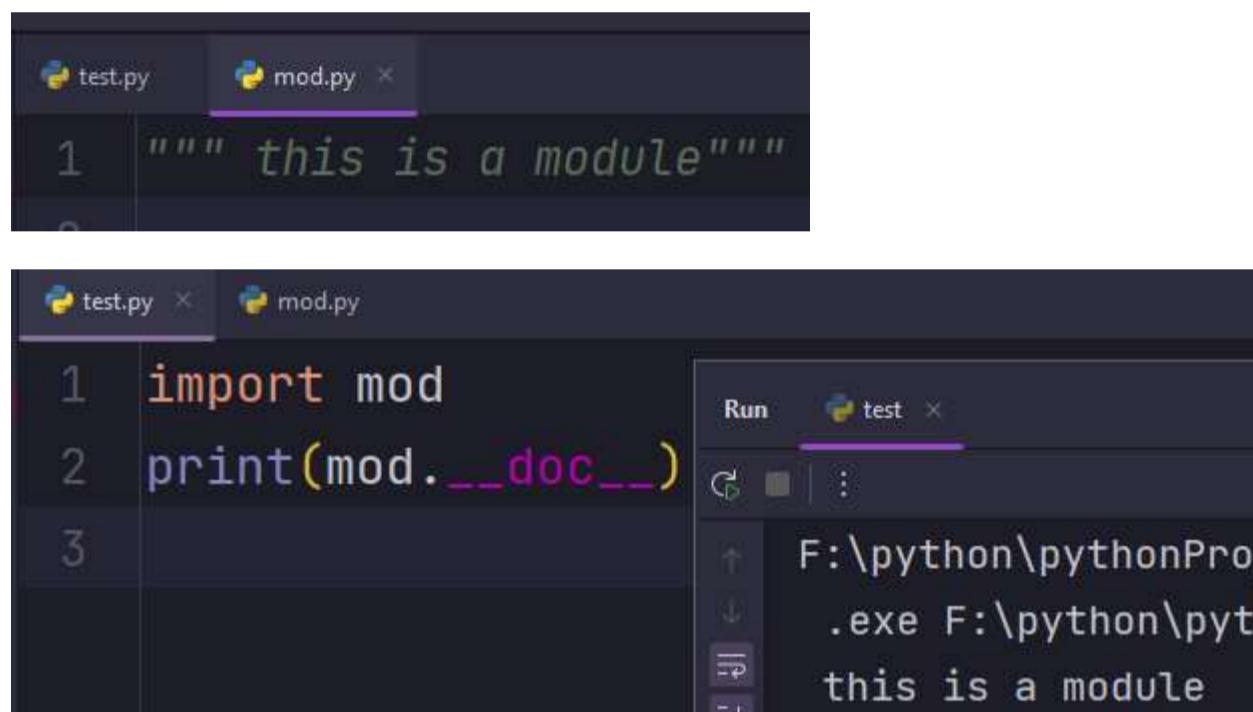
* درهنگام وارد کردن نام مژول اگر از دو نقطه (. .) استفاده کنیم، منظور پکیج پدر (قبلی) می باشد.



```
Project: pythonProject F:\python\pythonProject
  pk1
    _init_.py
    mod21.py
    mod22.py
  __init__.py
  mod1.py
  venv library root
```

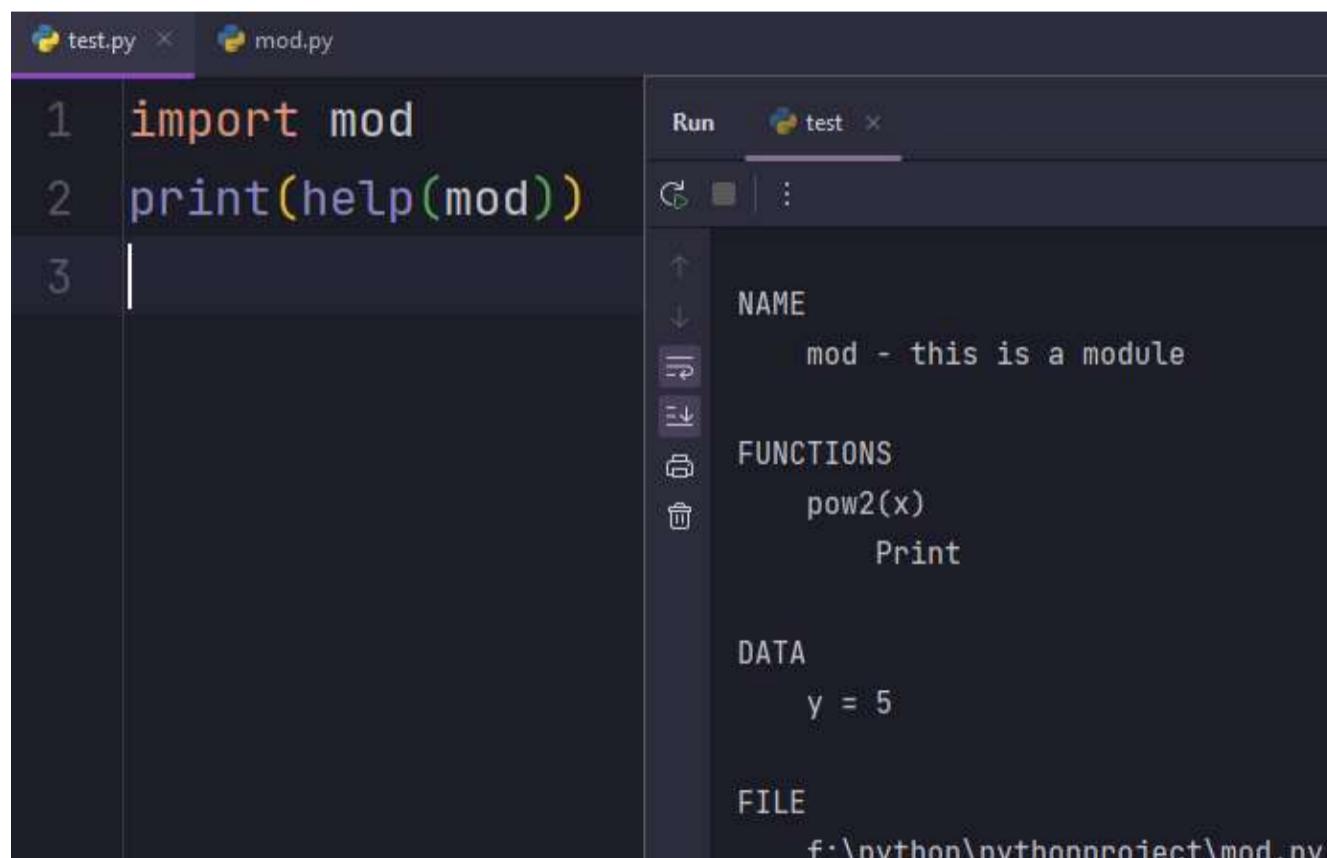
```
test.py mod21.py mod1.py mod22.py
1 from ..mod1 import func21
2 func21(5)
3
```

* مژول می تواند داک استرینگ داشته باشد.



```
test.py mod.py
1 """ this is a module"""
2
3 test.py mod.py
Run: test
F:\python\pythonProject
  .exe F:\python\pyt
  this is a module
```

* اگر از `__doc__` استفاده کنیم فقط داک استرینگ مژول را نشان می دهد اما اگر از `help` استفاده کنیم همه توضیحات مربوط به مژول و نامهای داخل آن را نشان می دهد.



```
test.py mod.py
1 import mod
2 print(help(mod))
3
Run: test
NAME
  mod - this is a module
FUNCTIONS
  pow2(x)
  Print
DATA
  y = 5
FILE
  f:\python\pythonproject\mod.py
```

✖ درس ۹: متغیر محیطی path و ورژن‌های پایتون ✖

درس ۱۰: سیستم مدیریت بسته (PyPi و pip)

- * برای نصب پکیج در پایتون می‌توانیم از `pip` استفاده کنیم، به این صورت که در ترمینال ابتدا `pip` را می‌نویسیم و سپس `install` و اسم پکیج را می‌نویسیم.

```
Terminal Local + ▾  
(venv) PS F:\python\pythonProject> pip install menta
```

- * با نوشتن اسم پکیج آخرین نسخه آن نصب می‌شود، همچنین ما می‌توانیم نسخه دلخواهمان را نصب کنیم، به این صورت که بعد از اسم پکیج `==` می‌گذاریم و ورژن آن را می‌نویسیم.

```
Terminal Local + ▾  
(venv) PS F:\python\pythonProject> pip install menta==0.0.1a1
```

- * برای حذف پکیج در پایتون با ترمینال، ابتدا `pip` را می‌نویسیم و سپس `uninstall` و اسم پکیج را می‌نویسیم.

```
Terminal Local + ▾  
(venv) PS F:\python\pythonProject> pip uninstall menta
```

- * برای ارتقای نسخه‌ی یک پکیج از دستور `upgrade` استفاده می‌کنیم، به این صورت که ابتدا `pip` را می‌نویسیم و سپس `install --upgrade` و اسم پکیج را می‌نویسیم.

```
Terminal Local + ▾  
(venv) PS F:\python\pythonProject> pip install --upgrade menta
```

- * برای ارتقای `pip` به صورت زیر عمل می‌کنیم.

```
Terminal Local + ▾  
(venv) PS F:\python\pythonProject> python -m pip install --upgrade pip
```

* برای نمایش لیست همه‌ی پکیج‌های نصب شده از `pip list` استفاده می‌کنیم.

```
(venv) PS F:\python\pythonProject> pip list
Package    Version
-----
cffi      1.15.1
colorama   0.4.6
```

* برای نمایش لیست پکیج‌هایی که به ارتقا نیاز دارند از `pip list --outdated` استفاده می‌کنیم.

```
(venv) PS F:\python\pythonProject> pip list --outdated
Package    Version  Latest  Type
-----
numpy      1.24.3  1.25.0  wheel
setuptools 67.8.0  68.0.0  wheel
```

* برای نمایش جزئیات یک پکیج از `show` استفاده می‌کنیم.

```
(venv) PS F:\python\pythonProject> pip show numpy
```

* اگر بخواهیم از پروژه روی یک سیستم دیگر استفاده کنیم، باید همه‌ی پکیج‌های مورد نیاز در آن‌جا نصب شود، برای این‌کار ابتدا **اسم پکیج‌ها** را در یک فایل `txt` ذخیره می‌کنیم و سپس از آن‌ها استفاده می‌کنیم.

* برای ذخیره اسم پکیج‌های مورد نیاز ابتدا **> freeze** را می‌نویسیم و سپس اسم فایلی که می‌خواهیم پکیج‌ها در آن ذخیره شوند را به **دلخواه می‌نویسیم**.

```
(venv) PS F:\python\pythonProject> pip freeze > requirements.txt
```

```
requirements.txt
1 cffi==1.15.1
2 colorama==0.4.6
3 numpy==1.24.3
4 pycparser==2.21
5 PyNaCl==1.5.0
6 termcolor==2.3.0
```

* برای نصب پکیج‌های مورد نیاز ابتدا **-r** را می‌نویسیم و سپس اسم فایلی که پکیج‌ها را در آن ذخیره کردیم را می‌نویسیم.

```
(venv) PS F:\python\pythonProject> pip install -r requirements.txt
```

درس ۱۱: محیط مجازی (virtualenv و venv)

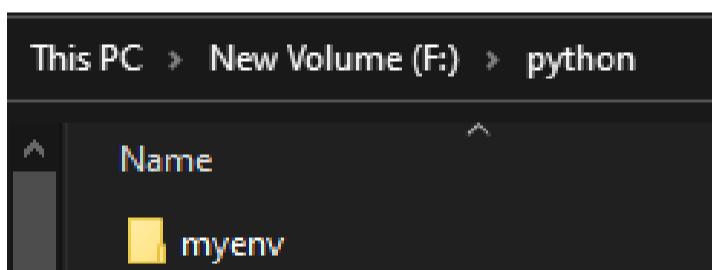
* برای ایجاد محیط مجازی باید **virtualenv** را نصب کنیم.

```
Terminal Local + ▾  
(venv) PS F:\python\pythonProject> pip install virtualenv
```

* سپس می‌توانیم مشخص کنیم که محیط مجازی در کجا ایجاد شود، به این صورت که ابتدا **virtualenv** را می‌نویسیم و سپس اسم پوشه‌ای که می‌خواهیم در آن ایجاد شود.

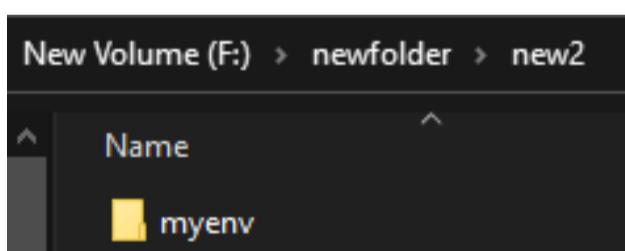
* در ترمینال از **cd** برای مشخص کردن یک مسیر استفاده می‌شود.
* در ترمینال از **..** برای رفتن به یک مسیر قبل‌تر استفاده می‌شود.

```
Terminal Local + ▾  
(venv) PS F:\python\pythonProject> cd ..  
(venv) PS F:\python> cd ..  
(venv) PS F:\python\pythonProject> cd ..  
(venv) PS F:\python> cd ..  
(venv) PS F:\> cd python  
(venv) PS F:\python> virtualenv myenv
```



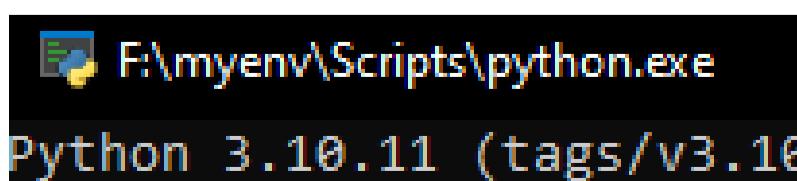
* می‌توانیم مسیر را به روش دیگر نیز مشخص کنیم، به این صورت که ابتدا **virtualenv** را می‌نویسیم و سپس مسیری را که می‌خواهیم محیط مجازی در آن ایجاد شود را می‌نویسیم.

```
Terminal Local + ▾  
(venv) PS F:\> virtualenv newfolder\new2\myenv
```

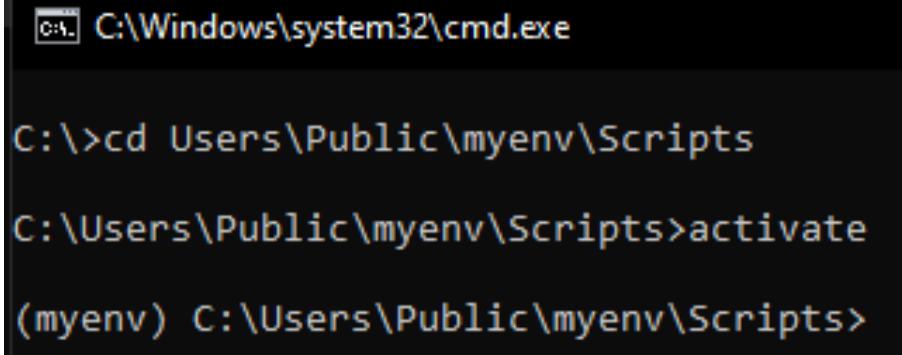


* می‌توانیم مشخص کنیم این محیط مجازی بر اساس چه نسخه‌ای از پایتون نصب شود.

```
Terminal Local + ▾  
(venv) PS F:\> virtualenv --python=python3.10 myenv
```

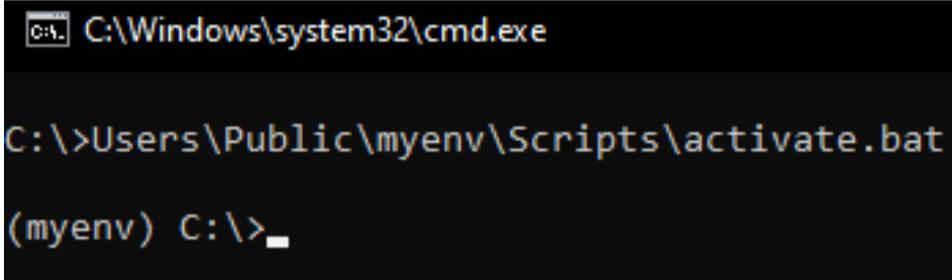


* برای فعالسازی محیط مجازی باید فایل **Scripts** که در پوشه **activate.bat** موجود دارد را اجرا کنیم و پس از آن هر پکیجی که نصب کنیم روی این محیط مجازی نصب می شود.



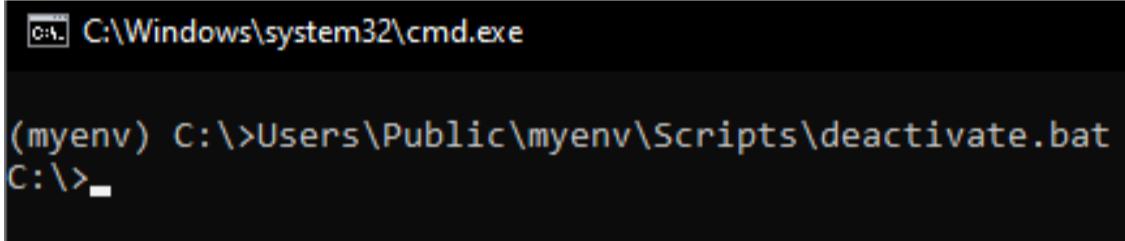
```
C:\>cd Users\Public\myenv\Scripts  
C:\Users\Public\myenv\Scripts>activate  
(myenv) C:\Users\Public\myenv\Scripts>
```

* با روش زیر نیز می توانیم فایل **activate.bat** را اجرا کنیم.



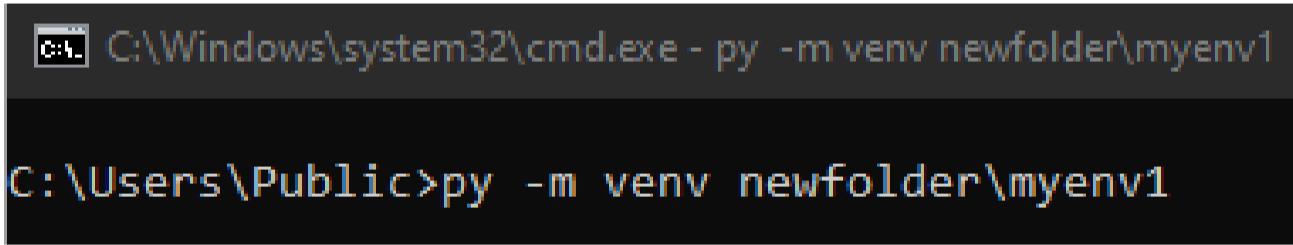
```
C:\>Users\Public\myenv\Scripts\activate.bat  
(myenv) C:\>_
```

* برای غیرفعال کردن محیط مجازی باید فایل **deactivate.bat** را اجرا کنیم.

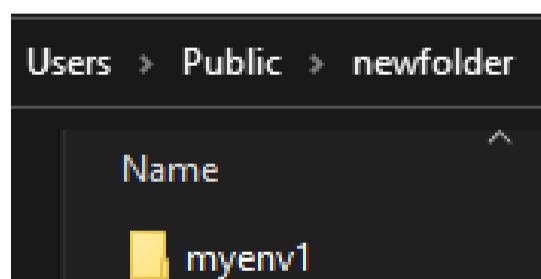


```
(myenv) C:\>Users\Public\myenv\Scripts\deactivate.bat  
C:\>_
```

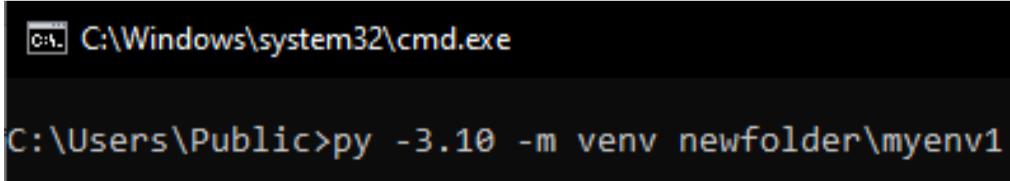
* در نسخه های جدید پایتون برای ایجاد محیط مجازی نیاز نیست که **virtualenv** را نصب کنیم و می توانیم با استفاده از **venv** محیط مجازی را ایجاد کنیم.



```
C:\>C:\Windows\system32\cmd.exe - py -m venv newfolder\myenv1  
C:\Users\Public>py -m venv newfolder\myenv1
```



* می توانیم نسخه دلخواه را نصب کنیم.

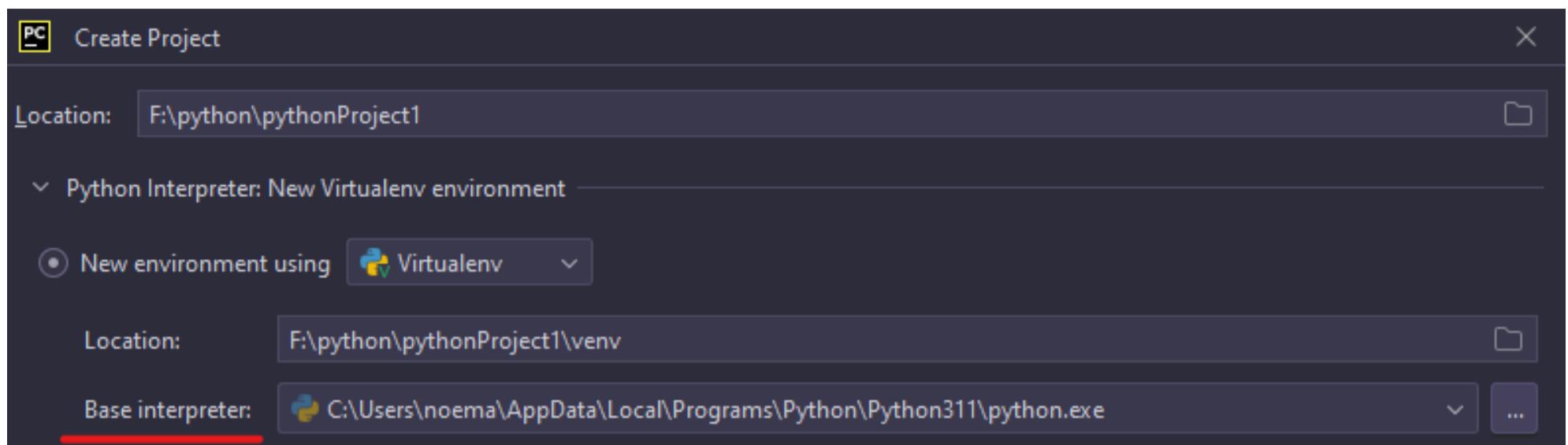


```
C:\>C:\Windows\system32\cmd.exe  
C:\Users\Public>py -3.10 -m venv newfolder\myenv1
```

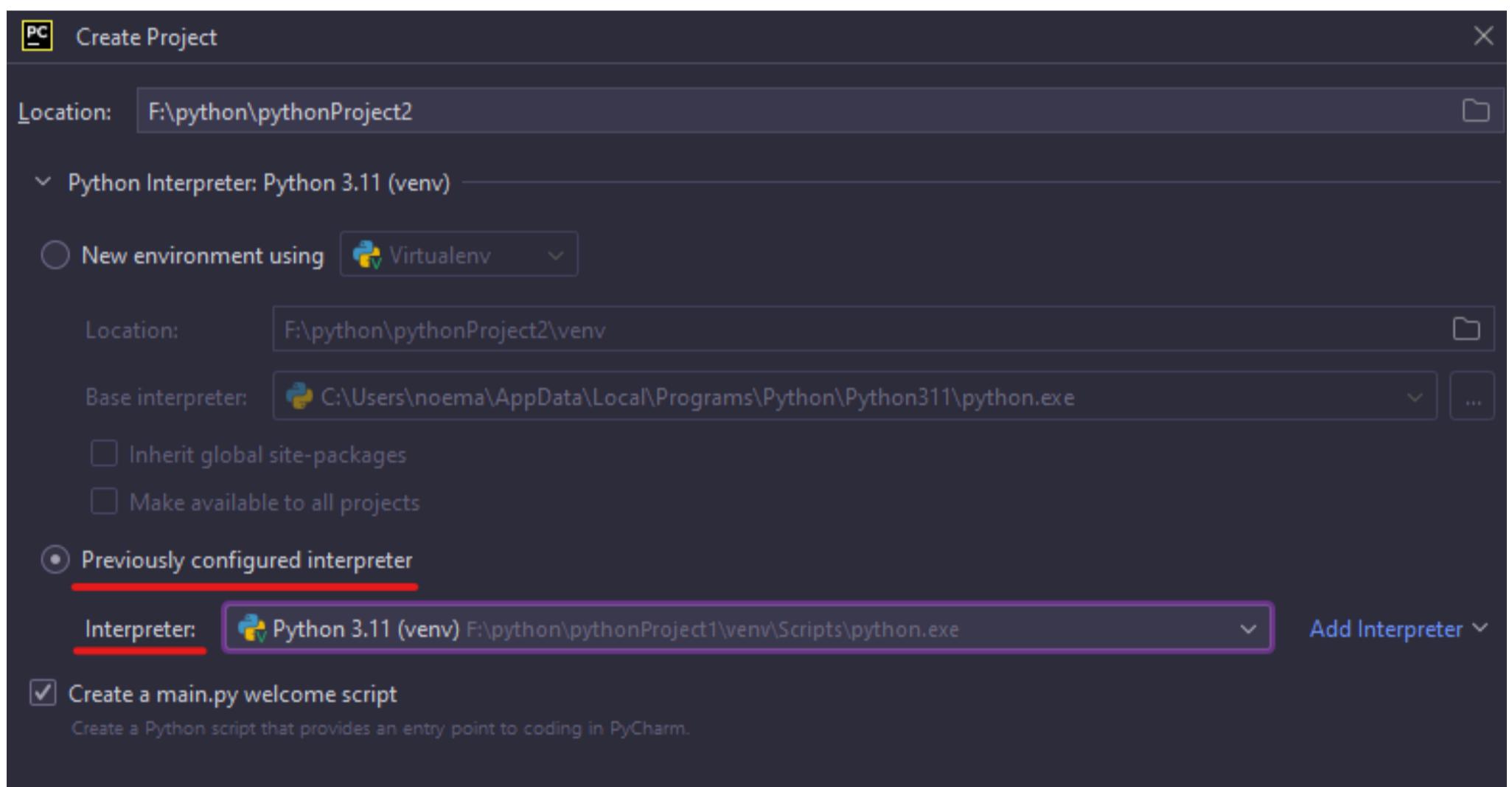
* برای فعال یا غیرفعال کردن نیز همانند قبل عمل می کنیم.

* در پایچارم با ایجاد پروژه می‌توانیم محیط مجازی را نیز ایجاد کنیم.

* از قسمت **base interpreter** می‌توانیم نسخه دلخواه را انتخاب کنیم.



* برای استفاده از یک محیط مجازی در پایچارم که قبل ایجاد شده است باید در هنگام ایجاد پروژه گزینه **Previously configured interpreter** را فعال کنیم و سپس فایل **python.exe** موجود در پوشه **Scripts** محیط مجازی را انتخاب کنیم.



درس ۱۲: تکالیف مبحث ماژول‌ها و بسته‌ها

۱. یک ماژول به نام "math_operations" ایجاد کنید که شامل توابعی برای عملیات ریاضی پایه است: جمع، تفریق، ضرب و تقسیم. سپس برنامه دیگری ایجاد کرده و این ماژول را در آن وارد کنید. از توابع نوشته شده در ماژول برای انجام عملیات روی دو عدد وارد شده توسط کاربر استفاده کنید.

۲. یک پکیج پایتون به نام "shapes" ایجاد کنید که شامل دو ماژول است: "rectangle" و "circle". هر ماژول باید توابعی برای محاسبه مساحت و محیط دایره و مستطیل داشته باشد. سپس برنامه دیگری ایجاد کرده و این ماژول را در آن وارد کنید و مساحت و محیط دایره و مستطیل را با استفاده از مقادیر وارد شده توسط کاربر محاسبه کنید.

۳. یک پکیج پایتون به نام "data_analysis" ایجاد کنید که حاوی ماژولی به نام "statistics" است. ماژول باید توابعی برای محاسبه میانگین، میانه و انحراف استاندارد بر روی لیستی از اعداد داشته باشد. سپس برنامه دیگری ایجاد کرده و این ماژول را در آن وارد کنید و این محاسبات آماری را بر روی لیستی از اعداد وارد شده توسط کاربر نشان دهید.

پایان فصل هشتم

درس ۱: مفهوم فایل و انواع آن

* برای ذخیره طولانی مدت داده‌ها از فایل‌ها استفاده می‌کنیم.

در پایتون دو نوع فایل داریم:

- باینری: فایل‌هایی که با `decode` کردن آن‌ها چیزی نمی‌فهمیم (موسیقی، رنگ، عکس، صدا و...)
- متنی: فایل‌هایی که برای انسان قابل درک و خواندن هستند و کدگذاری آن‌ها یا `Unicode` (هر کدام از بایت‌های ذخیره شده در آن نشان دهنده‌ی یک کاراکتر هستند).

* در واقع همه‌ی فایل‌ها باینری هستند، فایل متنی نیز در هنگام ذخیره شدن به شکل باینری ذخیره می‌شود.

درس ۲: تابع `open` و مدهای باز کردن فایل

* از دستور `open` برای باز کردن فایل استفاده می‌کنیم،

دستور `open` چند آرگومان دارد:

`(file, mode, buffering, encoding, errors, newline, closed, opener)`

* اگر آرگومان‌ها را به ترتیب مشخص کنیم نیاز نیست اسم آرگومان را بنویسیم.

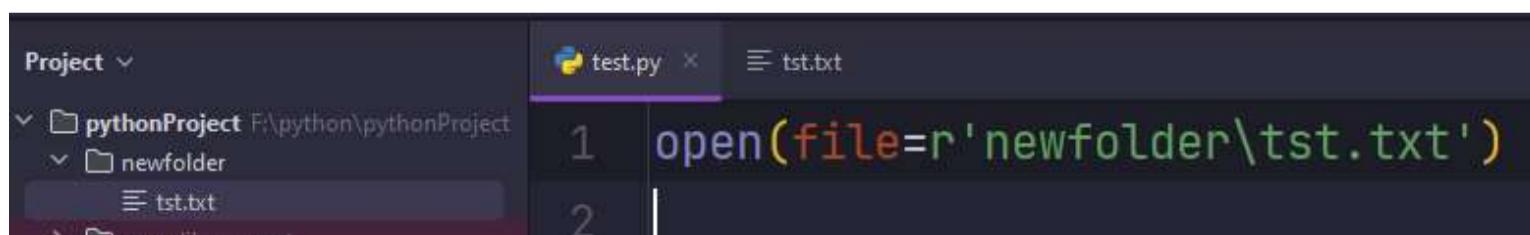
```
open('note.txt', 'r')
```

* با آرگومان `file` مشخص می‌کنیم چه فایلی باز شود.

```
open(file='picture.jpg')
```

* تنها آرگومان اجباری `open` آرگومان `file` است.

* اگر فایل در جایی که اسکریپت اجرا می‌شود نباشد، می‌توانیم آدرس فایل را بنویسیم.



* برای مشخص کردن آدرس پوشه فعلی از مژول `os` و سپس از دستور `getcwd` استفاده می‌کنیم.

```
import os  
print(os.getcwd())
```

* از آرگومان **mode** برای مشخص کردن نوع بازکردن فایل استفاده می‌کنیم.

* از مقدار **r** برای خواندن فایل‌های متنی استفاده می‌کنیم، اشاره‌گر نیز در ابتدای فایل است و مقدار پیش‌فرض **mode** نیز **r** می‌باشد.

```
open(file='note.txt', mode='r')
```

* برای خواندن از دستور **readline** استفاده می‌کنیم.

```
f = open('note.txt', 'r')  
print(f.readline())
```

* اگر فایل وجود نداشته باشد خطأ می‌دهد.

```
f = open('note2.txt', 'r')
```

```
f = open('note2.txt', 'r')  
^^^^^^^^^^^^^^^^^
```

```
FileNotFoundException: [Errno 2] No such file or directory: 'note2.txt'
```

* اگر برای **mode** از **rt** استفاده کنیم یعنی فایل متنی است و اگر **t** را نویسیم به صورت پیش‌فرض فایل متنی در نظر می‌گیرد.

* * اگر برای **mode** از **rb** استفاده کنیم یعنی فایل باینری است.

* از مقدار **w** (**wt**) برای **نوشتن** فایل‌های متنی استفاده می‌کنیم و اشاره‌گر نیز در ابتدای فایل است.

```
open('note.txt', 'w')
```

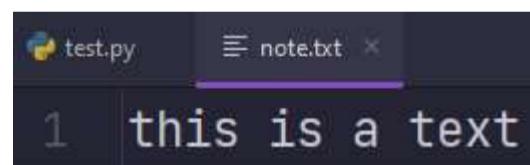
* اگر از **w** استفاده کنیم، نمی‌توانیم آن فایل را بخوانیم.

```
f = open('note.txt', 'w')  
print(f.readline())
```

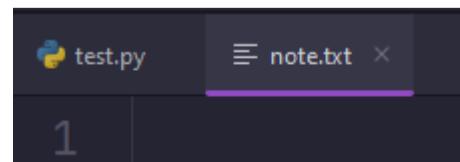
```
print(f.readline())  
^^^^^
```

```
io.UnsupportedOperation: not readable
```

* بعد از استفاده از **w** محتوای قبلی پاک می‌شود.



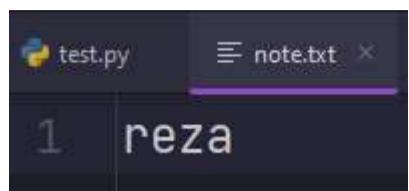
```
f = open('note.txt', 'w')
```



* اگر فایل وجود نداشته باشد اول ایجاد و سپس باز می‌شود و خطأ نمی‌دهد.

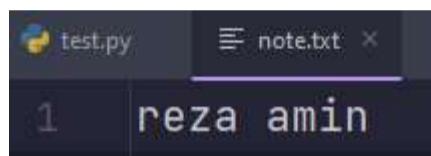
* برای نوشتن از دستور **write** استفاده می‌کنیم.

```
f = open('note.txt', 'w')
f.write('reza')
```



* از مقدار **a** (**at**) برای نوشتن در انتهای فایل متنی استفاده می‌کنیم، پس اشاره‌گر در انتهای فایل قرار دارد و محتوای قبلی را پاک نمی‌کند.

```
f = open('note.txt', 'a')
f.write(' amin')
```



* اگر فایل وجود نداشته باشد اول ایجاد و سپس باز می‌شود و خطای نمی‌دهد.

* از مقدار **x** (**xt**) برای نوشتن در ابتدای فایل متنی استفاده می‌کنیم، و اشاره‌گر در ابتدای فایل قرار دارد.

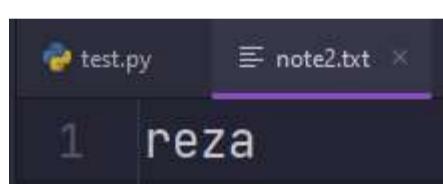
* اگر فایل وجود داشته باشد خطای نمی‌دهد.

```
f = open('note.txt', 'x')
f.write('reza')
```

```
f = open('note.txt', 'x')
^^^^^^^^^^^^^^^^^
FileExistsError: [Errno 17] File exists: 'note.txt'
```

* اگر فایل وجود نداشته باشد، اول ایجاد و سپس باز می‌شود و خطای نمی‌دهد.

```
f = open('note2.txt', 'x')
f.write('reza')
```



* اگر به جای **r**، **rt**، **w**، **wt**، **rwt**، **rb**، **ab**، **xb**، **wb**، **rb**، **xt** (**x**، **at**) استفاده می‌شود.

```
f = open('note2.txt', 'rb')
print(f.readline()) # -> b'reza'
```

* از مقدار **r+** (**r+t** یا **rt+**) برای خواندن و نوشتن استفاده می‌کنیم.

* اشاره‌گر در انتهای فایل وجود دارد و هنگام نوشتن در انتهای فایل نوشته می‌شود.

```
f = open('note2.txt', 'r+')
f.write('-amin')
f.write('-hassan')
print(f.readline()) # -> reza-amin-hassan
```

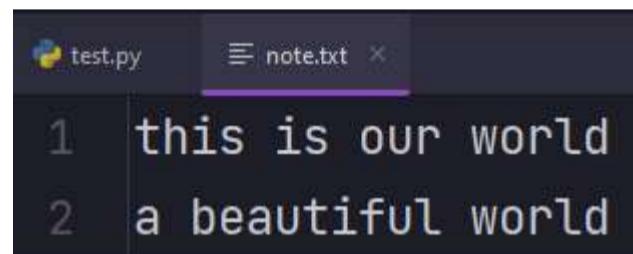
* اگر فایل وجود نداشته باشد خطای دهد.

```
f = open('note10.txt', 'r+')
f.write('reza')
```

```
f = open('note10.txt', 'r+')
^^^^^^^^^^^^^^^^^
```

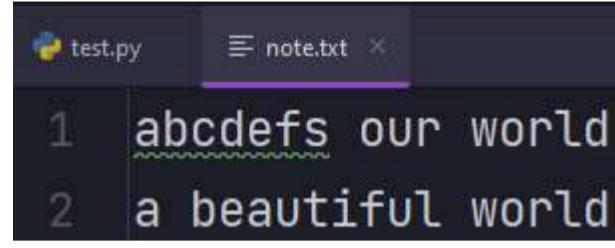
```
FileNotFoundException: [Errno 2] No such file or directory: 'note10.txt'
```

* اگر **readline** را بنویسیم و فقط **write** را بنویسیم اشاره‌گر در ابتدای فایل قرار می‌گیرد و کarakترها را جایگزین می‌کند.



```
test.py      note.txt
1 this is our world
2 a beautiful world
```

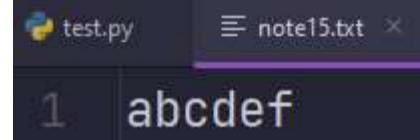
```
f = open('note.txt', 'r+')
f.write('abcdef')
```



```
test.py      note.txt
1 abcdefs our world
2 a beautiful world
```

* از مقدار **w+** (**w+t** یا **wt+**) برای خواندن و نوشتن استفاده می‌شود، متن قبلی پاک می‌شود و اگر فایل وجود نداشته باشد ایجاد و سپس باز می‌کند و خطای دهد.

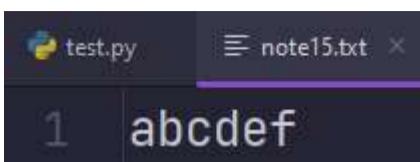
```
f = open('note15.txt', 'w+')
f.write('abcdef')
```



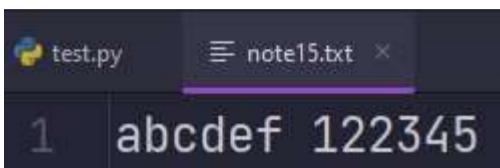
```
test.py      note15.txt
1 abcdef
```

* بعد از **write** اشاره‌گر به انتهای فایل می‌رود و اگر از **readline** استفاده کنیم، چیزی نمایش داده نمی‌شود.

* از مقدار **a+** (**a+t** یا **at+**) برای خواندن و نوشتن استفاده می‌شود، متن قبلی پاک نمی‌شود و از انتهای فایل می‌نویسد.



```
f = open('note15.txt', 'a+')
f.write(' 122345')
```



* بعد از **write** اشاره‌گر به انتهای فایل می‌رود و اگر از **readline** استفاده کنیم، چیزی نمایش داده نمی‌شود.

* از مقدار **x+** (**x+t** یا **xt+**) برای خواندن و نوشتن استفاده می‌شود، اگر فایل وجود نداشته باشد ایجاد و سپس باز می‌کند و خطای دهد و اگر فایل وجود داشته باشد خطای دهد.

```
f = open('note2.txt', 'x+')
f.write(' reza')
```

```
f = open('note2.txt', 'x+')
^^^^^^^^^^^^^^^^^
FileExistsError: [Errno 17] File exists: 'note2.txt'
```

* بعد از **write** اشاره‌گر به انتهای فایل می‌رود و اگر از **readline** استفاده کنیم، چیزی نمایش داده نمی‌شود.

* اگر به جای **r+** **rb+**، **ab+**، **wb+**، **rb+**، از **xt+** **x+**، **(at+)** **a+**، **(wt+)** **w+**، **(rt+)** **r+** استفاده می‌شود.

درس ۳: متدهای خواندن و نوشتن در فایل (بخش اول)

* برای نوشتن در یک فایل متنی از متدهای `write` استفاده می‌کنیم.

```
f = open('note.txt', 'w')
f.write('reza')
```

* اگر `print` را `write` کنیم، تعداد کاراکترهایی که می‌نویسد را نمایش می‌دهد.

```
f = open('note.txt', 'a')
print(f.write('reza')) # -> 4
```

* اگر یک کاراکتر غیر `ascii` بنویسیم خطای می‌دهد.

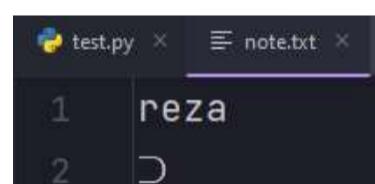
```
f = open('note.txt', 'a')
f.write('ߵ')
```

```
return codecs.charmap_encode(input, self.errors, encoding_table)[0]
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

```
UnicodeEncodeError: 'charmap' codec can't encode character '\u1450' in position 0: character maps to <undefined>
```

* برای رفع این خطای ابتدا باید در هنگام باز کردن فایل با آرگومان `encoding` نوع انکدینگ آن را مشخص کنیم.

```
f = open('note.txt', 'a', encoding='utf-8')
f.write('\nߵ')
```



* اگر فایل متنی شامل یک کاراکتر غیر `ascii` باشد و در هنگام باز کردن نوع انکدینگ آن را مشخص نکنیم، در هنگام خواندن خطای می‌دهد.

```
f = open('note.txt', 'a')
f.read()
```

```
f.read()
io.UnsupportedOperation: not readable
```

* اگر فایل را به صورت `باینری` باز کنیم، در هنگام `نوشتن` نمی‌توانیم به آن رشته بدهیم.

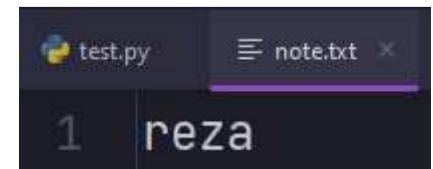
```
f = open('note.txt', 'ab')
s = 'reza'
print(f.write(s))
```

```
print(f.write(s))
^^^^^^^^^
```

```
TypeError: a bytes-like object is required, not 'str'
```

* برای حل این مشکل باید `رشته` را به `بایت` تبدیل کنیم.

```
f = open('note.txt', 'ab')
s = b'reza'
print(f.write(s)) # -> 4
```



درس ۴: متدهای خواندن و نوشتن در فایل (بخش دوم)

* برای این که مشخص کنیم که فایل را برای نوشتن باز کرده ایم یا خیر از متدهای `writable` استفاده می‌کنیم، که نتیجه یا `True` است یا `false`.

```
f = open('note.txt', 'r')
m = open('note.txt', 'w')
s = 'reza'
print(f.writable()) # -> False
print(m.writable()) # -> True
```

* اگر فایل را به صورت خواندن باز کنیم و از متدهای `write` استفاده کنیم خطای دهنده می‌توانیم از متدهای شرط استفاده کنیم.

```
f = open('note.txt', 'r')
s = 'reza'
if f.writable():
    print(f.write(s))
```

* برای این که مشخص کنیم که فایل را برای خواندن باز کرده ایم یا خیر از متدهای `readable` استفاده می‌کنیم، که نتیجه یا `True` است یا `false`.

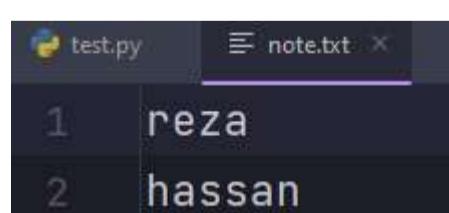
```
f = open('note.txt', 'r')
m = open('note.txt', 'w')
s = 'reza'
print(f.readable()) # -> True
print(m.readable()) # -> False
```

* اگر فایل را به صورت نوشتن باز کنیم و از متدهای `read` استفاده کنیم خطای دهنده می‌توانیم از متدهای شرط استفاده کنیم.

```
m = open('note.txt', 'w')
s = 'reza'
if m.readable():
    print(m.read(s))
```

* اگر بخواهیم یک `list` از رشته‌ها را در فایل متنی که وارد کردیم بنویسیم از متدهای `writelines` استفاده می‌کنیم.

```
f = open('note.txt', 'w', encoding='utf8')
lst = ['reza\n', 'hassan']
f.writelines(lst)
```



* با متدهای `writelines` علاوه بر `list` می‌توانیم دیکشنری (کلیدهای آن)، مجموعه، رشته و ... را به شرط این که نوع داده آن رشته باشد را در فایل متنی بنویسیم.

* برای خواندن محتوای یک فایل متنی از متده استفاده می کنیم.

```
f = open('note.txt', 'r', encoding='utf8')  
print(f.read())
```

```
reza  
hassan
```

* می توانیم مشخص کنیم که `read` چند کاراکتر را بخواند.

```
f = open('note.txt', 'r', encoding='utf8')  
print(f.read(3)) # -> rez
```

* اگر دوباره مشخص کنیم که `read` چند کاراکتر بخواند، فایل متنی را از اول نمی خواند بلکه از جایی می خواند که دستور قبلی خواندنش به پایان رسیده است.

```
f = open('note.txt', 'r', encoding='utf8')  
print(f.read(3))  
print(f.read(1))  
print(f.read(3))
```

```
rez  
a  
ha
```

* اگر از `readline` استفاده کنیم، یک خط را می خواند.

```
f = open('note.txt', 'r', encoding='utf8')  
print(f.readline(), end='') # -> reza  
print(f.readline(), end='') # -> hassan
```

* می توانیم مشخص کنیم که `readline` چند کاراکتر یک خط را بخواند.

```
f = open('note.txt', 'r', encoding='utf8')  
print(f.readline(2)) # -> re  
print(f.readline(2)) # -> za
```

* وقتی که `readline` به محدودیت تعداد کاراکتر خط برسد، با دستور بعدی خط بعدی را می خواند.

```
f = open('note.txt', 'r', encoding='utf8')  
print(f.readline(7)) # -> reza  
print(f.readline(2)) # -> ha
```

* دستور `readlines` همهی سطرها را می خواند و داخل یک لیست قرار می دهد.

```
f = open('note.txt', 'r', encoding='utf8')  
print(f.readlines()) # -> ['reza\n', 'hassan']
```

درس ۵: مدیریت بافر (بخش اول)

* یک حافظه‌ی موقت میانجی بین دو واحد مجزا است.

* با دستور **close** می‌توانیم فایل را ببندیم.

```
f = open('note.txt', 'w', encoding='utf8')
f.write('reza\nhassan')
f.close()
```

* سعی کنیم همیشه فایل را ببندیم.

* در مثال زیر تا زمانی که ورودی را مشخص نکنیم، داده‌های ما در فایل نوشته نمی‌شوند چون اطلاعات در **بافر** ذخیره شده‌اند.

The screenshot shows three stages of a Python script execution in PyCharm:

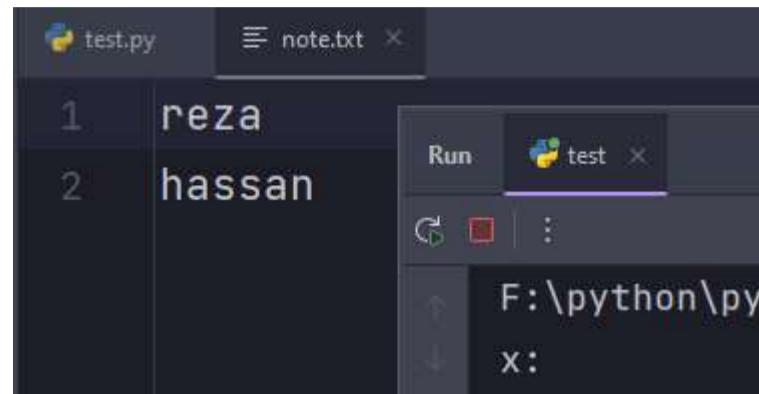
- Stage 1:** The code is being typed into the editor. It contains:

```
1 f = open('note.txt', 'w', encoding='utf8')
2 f.write('reza\nhassan')
3 input('x: ')
4 f.close()
```

The run tool window shows the script name and the current step: "Run test". The terminal pane is empty: "x: |".
- Stage 2:** The code has been run, and the user has entered "x: ". The run tool window shows the script name and the current step: "Run test". The terminal pane shows the input: "x: |".
- Stage 3:** The code has been run, and the user has entered "x: a". The run tool window shows the script name and the current step: "Run test". The terminal pane shows the input: "x: a" and the output: "Process finished".

* زمانی که از **flush** استفاده کنیم، اطلاعات را از بافر پاک و در فایل ذخیره می‌کند.

```
f = open('note.txt', 'w', encoding='utf8')
f.write('reza\nhassan')
f.flush()
input('x: ')
f.close()
```



* زمانی که از **flush** استفاده می‌کنیم، اطلاعات از بافر داخلی پاک و به بافر سیستم عامل منتقل می‌شود و پس از آن از بافر سیستم عامل به فایل انتقال می‌یابد، اما ممکن است این اطلاعات به بافر سیستم عامل منتقل شوند اما در فایل ذخیره نشوند، برای حل این مشکل ابتدا **os** را وارد می‌کنیم و سپس بعد از **fsync** از **flush** استفاده می‌کنیم، که در این حالت قطعاً اطلاعات در فایل ذخیره می‌شوند.

```
import os
f = open('note.txt', 'w', encoding='utf8')
f.write('reza\nhassan')
f.flush()
os.fsync(f)
```

درس ۶: مدیریت بافر (بخش دوم)

* اگر مقدار آرگومان **buffering** صفر (0) باشد، به معنای این است که کار بافرینگ انجام نشود. (اطلاعات در حافظه موقت ذخیره نمی‌شوند و مستقیم در فایل ذخیره می‌شوند).

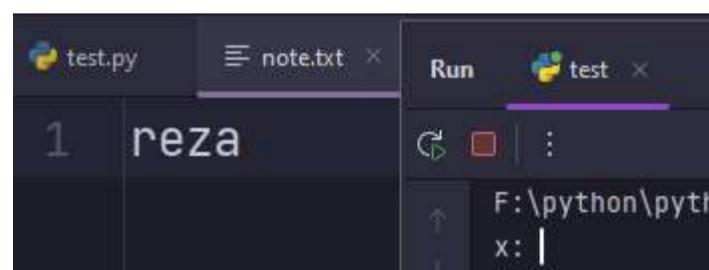
* مقدار صفر (0) برای فایل‌های متنی استفاده نمی‌شود.

```
f = open('note.txt', 'w', buffering=0)
f.write('reza')
```

```
f = open('note.txt', 'w', buffering=0)
^^^^^^^^^
ValueError: can't have unbuffered text I/O
```

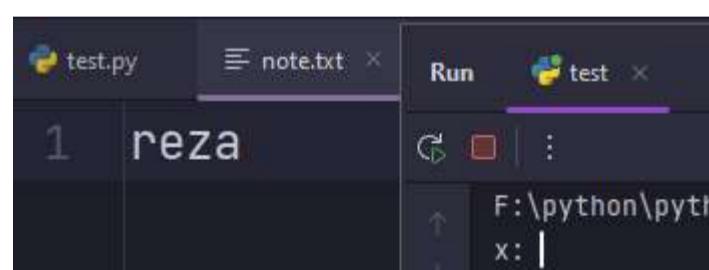
* مقدار صفر (0) برای فایل‌های باینری استفاده می‌شود.

```
f = open('note.txt', 'wb', buffering=0)
f.write(b'reza')
input('x: ')
```



* اگر به مقدار یک (1) بدهیم (**line buffering**)، هر خط را در بافر ذخیره می‌کند و هنگام رفتن به خط بعدی (با استفاده از \n در انتهای رشته)، آن خط را در فایل ذخیره می‌کند و خط بعدی را در بافر قرار می‌دهد.

```
f = open('note.txt', 'w', buffering=1)
f.write('reza\n')
f.write('hassan')
input('x: ')
```



* از **line buffering** فقط برای فایل‌های متنی استفاده می‌کنیم.

```
f = open('note.txt', 'wb', buffering=1)
f.write(b'reza\n')
```

```
E:\python\pythonProject\test.py:1: RuntimeWarning: line buffering (buffering=1) isn't supported in binary mode, the default buffer size will be used
  f = open('note.txt', 'wb', buffering=1)
```

* با وارد کردن `io` و سپس دستور `DEFAULT_BUFFER_SIZE` می‌توانیم مشخص کنیم که سایز بافر سیستم عامل چند کاراکتر است.

```
import io  
  
print(io.DEFAULT_BUFFER_SIZE) # -> 8192
```

* اگر به `buffering` مقدار منفی یک (-1) بدهیم به اندازه‌ی تعداد کاراکترهایی که بافر سیستم عامل می‌تواند ذخیره کند، کاراکترها در بافر ذخیره می‌شوند و در صورتی قبل از پایان برنامه در فایل ذخیره می‌شوند که تعداد کاراکترها از ظرفیت بافر سیستم عامل بیشتر شود.

* اگر به `buffering` مقدار دلخواه بدهیم، به تعداد آن مقدار در بافر ذخیره می‌کند و معمولاً در حالت [باینری](#) استفاده می‌شود.

```
f = open('note.txt', 'wb', buffering=10)  
f.write(b'123456789')
```

* اگر تعداد کاراکترها از مقداری که به `buffering` دادیم بیشتر باشد، همه‌ی آن در فایل ذخیره می‌شود.

درس ۷: تغییر موقعیت اشاره‌گر

* متدهای `tell` موقعیت اشاره‌گر را نشان می‌دهد.

```
test.py note.txt
1 reza
2 ali
```

```
f = open('note.txt', 'r', encoding='utf8')
print(f.tell())
print(f.readline(), end='')
print(f.tell())
print(f.readline(), end='')
```

```
0
reza
5
ali
```

* برای رفتن به خط بعدی از `\n` استفاده می‌کنیم، این دستور در ویندوز درواقع دو کاراکتر `\r\n` می‌باشد و به همین دلیل زمانی که موقعیت آن را تعیین می‌کنیم، دو کاراکتر محسوب می‌شود. (در لینوکس فقط کاراکتر `\n` است).

```
test.py note.txt
1 ali
2 reza
```

```
f = open('note.txt', 'rb')
print(f.tell()) # -> 0
print(f.read()) # -> b'ali\r\nreza'
print(f.tell()) # -> 9
```

* با دستور `os` از `linesep` و نمایش آن با `repr` می‌توانیم در پایتون مشخص کنیم که سیستم عامل از چه دستوری برای رفتن به خط بعدی استفاده می‌کند.

```
import os

print(repr(os.linesep)) # -> '\r\n'
```

* تعیین موقعیت اشاره‌گر در فایل‌های باینری دقیق‌تر است و در فایل‌های متنی ممکن است دقیق عمل نکند.

```
f = open('note.txt', 'r', encoding='utf8')
print(f.tell()) # -> 0
f.read(6)
print(f.tell()) # -> 6
f.seek(0)
print(f.tell()) # -> 0
```

* متدها **seek** یک آرگومان دیگر نیز می‌گیرد که مشخص می‌کند از ابتداء، جایی که هستیم یا انتهای تغییر مکان را شروع کنیم.

* اگر مقدار آرگومان دوم متدها **seek**، صفر (0) باشد، یعنی از ابتدای فایل باید مکان اشاره‌گر را تغییر دهد و به صورت پیش‌فرض مقدار این آرگومان صفر می‌باشد.

```
f = open('note.txt', 'rb')
print(f.tell()) # -> 0
print(f.read(5)) # -> b'ali\r\n'
f.seek(2, 0)
print(f.tell()) # -> 2
print(f.read(5)) # -> b'i\r\nre'
```

* اگر مقدار آرگومان دوم متدها **seek**، یک (1) باشد، یعنی از موقعیت فعلی باید مکان اشاره‌گر را تغییر دهد.

```
f = open('note.txt', 'rb')
print(f.tell()) # -> 0
print(f.read(5)) # -> b'ali\r\n'
f.seek(2, 1)
print(f.tell()) # -> 7
print(f.read(5)) # -> b'za'
```

* اگر مقدار آرگومان دوم متدها **seek**، دو (2) باشد، یعنی از انتهای فایل باید مکان اشاره‌گر را تغییر دهد.

```
f = open('note.txt', 'rb')
print(f.tell()) # -> 0
print(f.read(5)) # -> b'ali\r\nr'
f.seek(2, 2)
print(f.tell()) # -> 11
print(f.read(5)) # -> b''
```

* اگر مقدار آرگومان دوم متدها **seek**، دو (2) باشد، می‌توانیم برای تعیین مکان از عدد منفی استفاده کنیم که یعنی از آخر به عقب برمی‌گردد.

```
f = open('note.txt', 'rb')
print(f.tell()) # -> 0
print(f.read(5)) # -> b'ali\r\n'
f.seek(-2, 2)
print(f.tell()) # -> 6
print(f.read(5)) # -> b'za'
```

* هنگام استفاده از فایل‌های متنی یکی از آرگومان‌های متدها **seek**، باید صفر (0) باشد.

```
f = open('note.txt', 'r', encoding='utf8')
f.seek(2)
f.seek(0, 0)
f.seek(2, 0)
f.seek(0, 2)
f.seek(0, 1)
```

* هنگام استفاده از فایل‌های متنی در متدها **seek** نمی‌توانیم از عدد منفی استفاده کنیم.

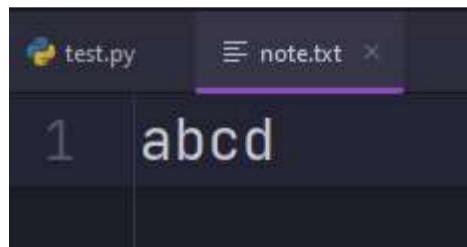
* اگر در هنگام استفاده از فایل‌های متنی یکی از آرگومان‌های متدها seek (۰)، صفر (۰) نباشد، با خطا مواجه می‌شویم.

```
f = open('note.txt', 'r', encoding='utf8')  
f.seek(2, 1)
```

```
f.seek(2, 1)  
io.UnsupportedOperation: can't do nonzero cur-relative seeks
```

* اگر در هنگام استفاده از فایل‌های متنی در هنگام باز کردن برای mode a استفاده کنیم، استفاده از متدها seek (۰) و mode a دیگر بی‌فااید است، چون در این حالت نوشتمن از انتهای و مکان اشاره‌گر قابل تغییر نیست.

```
f = open(file='note.txt', mode='a', encoding='utf8')  
f.write('ab')  
f.seek(0)  
f.write('cd')
```



درس ۸: دستور `with/as` و شی `context manager`

* در پایتون اشیایی هستند به نام `context manager` که دو متدهای `enter` و `exit` دارند. تعریف شده است که این اشیا قبل و بعد از دستورات یک سری کار انجام می‌دهند.

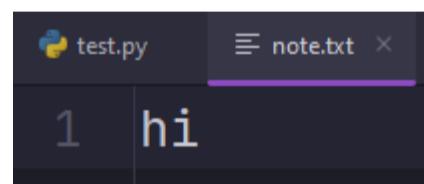
* همهی دستور `with` با دستور `context manager` فعال می‌شوند، به این صورت که `with` قبل از اجرای دستورات `enter` را صدا می‌زند و اجرا می‌کند و بعد از اتمام دستورات `exit` را صدا می‌زند و اجرا می‌کند.

```
class A:  
    def __enter__(self):  
        print('start!')  
  
    def __exit__(self, exc_type, exc_value, exc_tb):  
        print('end!')  
  
with A():  
    print('I am Reza')
```

```
start!  
I am Reza  
end!
```

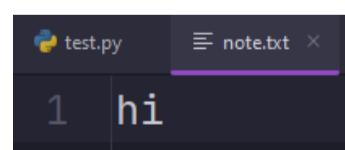
* فایل‌ها نیز `with` در فایل‌ها قبل از جرای دستورات فایل را باز می‌کند و پس از آن تحت هر شرایطی فایل را می‌بندد.

```
f = open('note.txt', 'w')  
with f:  
    f.write('hi')  
f.write('aa')  
  
    f.write('aa')  
ValueError: I/O operation on closed file.
```



* اگر فایل را در یک متغیر قرار ندهیم و در دستور `with` بخواهیم بنویسیم با خطای مواجه می‌شویم، برای حل این مشکل می‌توانیم از `as` استفاده کنیم، دستور `as` خود فایل را بر می‌گرداند.

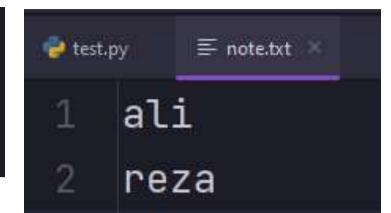
```
with open('note.txt', 'w') as f:  
    f.write('hi')
```



* بهتر است برای خواندن یک فایل از حلقه **for** استفاده کنیم.

```
with open('note.txt', 'r') as f:  
    for line in f:  
        print(line, end='')
```

```
ali  
reza
```

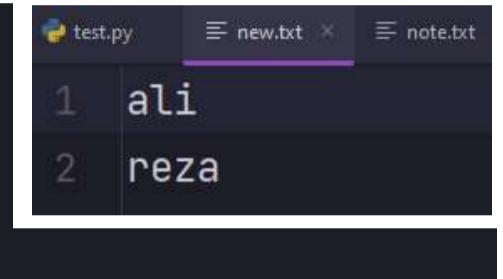


```
test.py note.txt  
1 ali  
2 reza
```

* از **with** می‌توانیم به صورت تودرتو استفاده کنیم.

```
with open('note.txt', 'r') as f:  
    with open('new.txt', 'w') as n:  
        for line in f:  
            print(line, end='')  
            n.write(line)
```

```
ali  
reza
```

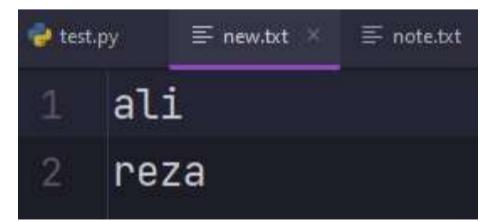


```
test.py new.txt note.txt  
1 ali  
2 reza
```

* به جای استفاده از **with** به صورت تودرتو بهتر است فایل‌ها را در یک خط باز کنیم و با **کاما (,**) جدا کنیم.

```
with open('note.txt', 'r') as f, open('new.txt', 'w') as n:  
    for line in f:  
        print(line, end='')  
        n.write(line)
```

```
ali  
reza
```



```
test.py new.txt note.txt  
1 ali  
2 reza
```

درس ۹: شیء فایل استاندارد

- * سه نوع شیء فایل توسط پایتون ایجاد می‌شود که به آن‌ها شیء فایل استاندارد می‌گویند.
- * این شیء‌ها `stdin`, `stdout`, `stderr` هستند.
- * با این شیء‌ها در صفحه نمایش می‌نویسیم و از کیبورد می‌خوانیم.
- * اشیاء فایل استاندارد را باید از `sys` وارد کنیم.
- * با `stdin` از کیبورد می‌خوانیم و در واقع متدهای `input` همان `stdin` می‌باشد.

```
from sys import stdin

f = stdin.readline()
print(f)
```



```
hi
hi
```

- * با `stdout` در صفحه نمایش می‌نویسیم و در واقع متدهای `print` همان `stdout` می‌باشد.

```
from sys import stdout

stdout.write('hello') # -> hello
```

- * با `stderr` در صفحه نمایش می‌نویسیم و برای استثناهای خطاهای استفاده می‌شود.

```
from sys import stderr

stderr.write('Error!')
```

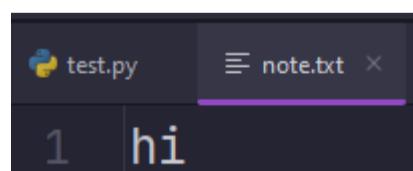


```
Error!
```

- * می‌توانیم `stdout` را تغییر دهیم تا به جای این که صفحه نمایش نوشته شود، در فایل نوشته شود.

```
from sys import stdout

stdout_temp = stdout
stdout = open('note.txt', 'w')
stdout.write('hi')
stdout.close()
stdout = stdout_temp
```



* هنگامی که `print` را در `sys.stdout` تغییر دهیم، `print` نیز در فایل می‌نویسد.

```
import sys

stdout_temp = sys.stdout
sys.stdout = open('note.txt', 'w')
sys.stdout.write('hi')
print('\nHello Reza')
sys.stdout.close()
sys.stdout = stdout_temp
print('bye') # -> bye
```

* را نیز می‌توانیم تغییر دهیم.

```
import sys

stdin_temp = sys.stdin
sys.stdin = open('note.txt', 'r')
f = sys.stdin.readline()
z = input()
print(z) # -> Hello Reza
sys.stdin.close()
sys.stdin = stdin_temp
```

* دستور `print` یک آرگومان به نام `file` می‌گیرد که پیش‌فرض آن `sys.stdout` می‌باشد و ما می‌توانیم یک فایل دیگر را به آن بدهیم.

```
f = open('note.txt', 'w')
print('reza', file=f)
```

* دستور `print` یک آرگومان به نام `flush` می‌گیرد مقدار آن `True` یا `False` می‌باشد، که اگر `True` باشد یعنی اطلاعات در بافر ذخیره نشوند و مستقیم به فایل انتقال یابند و به صورت پیش‌فرض مقدار آن `False` می‌باشد.

```
f = open('note.txt', 'w')
print('reza', file=f, flush=True)
input()
```

درس ۱۰: نکات تکمیلی در مورد فایل‌ها

* آرگومان `open` در `errors` مشخص می‌کند که اگر فایل با خطا مواجه شد چه چیزی را نشان دهد.

* مقدار پیش‌فرض آرگومان `None` می‌باشد که در واقع همان `strict` است و خطای عادی را نشان می‌دهد.

```
f = open('note1.txt', 'w')
f.write('בדיקה')

m = open('note2.txt', 'w', errors=None)
m.write('בדיקה')

v = open('note3.txt', 'w', errors='strict')
v.write('בדיקה')
```

```
return codecs.charmap_encode(input, self.errors, encoding_table)[0]
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

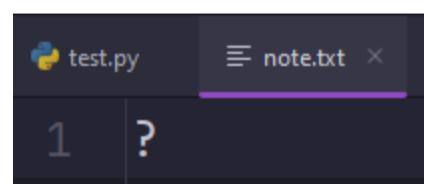
UnicodeEncodeError: 'charmap' codec can't encode character '\u147b' in position 0: character maps to <undefined>
```

* اگر به آرگومان `ignore` مقدار `errors` بدهیم، خطای نادیده می‌گیرد.

```
f = open('note1.txt', 'w', errors='ignore')
f.write('בדיקה')
```

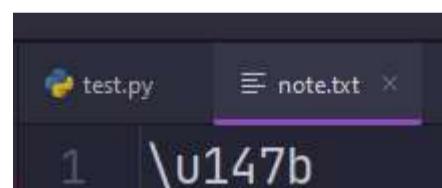
* اگر به آرگومان `replace` مقدار `errors` بدهیم، خطای نادیده می‌گیرد و به جای آن در فایل علامت سوال (?) می‌نویسد.

```
f = open('note.txt', 'w', errors='replace')
f.write('בדיקה')
```



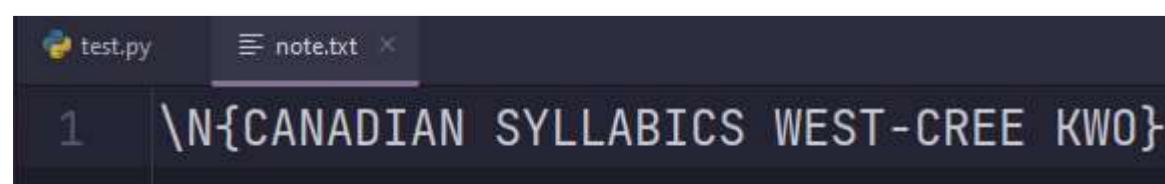
* اگر به آرگومان `backslashreplace` مقدار `errors` بدهیم، خطای نادیده می‌گیرد و به جای آن در فایل یونیکد آن را می‌نویسد.

```
f = open('note.txt', 'w', errors='backslashreplace')
f.write('בדיקה')
```



* اگر به آرگومان `namereplace` مقدار `errors` بدهیم، خطای نادیده می‌گیرد و به جای آن در فایل نام آن را می‌نویسد.

```
f = open('note.txt', 'w', errors='namereplace')
f.write('בדיקה')
```



* اگر از دستور **fileno** استفاده کنیم، توصیفگر فایل را نمایش می‌دهد.

```
f = open('note.txt', 'w')
print(f.fileno()) # -> 3
```

* با دستور **truncate** مشخص می‌کنیم که چند بایت از فایل را نگه دارد.

```
f = open('note.txt', 'w')
f.write('reza\ndolati')
f.truncate(3)
```



* با دستور **os.path.exists** از **os** می‌توانیم مشخص کنیم که یک فایل وجود دارد یا خیر و با استفاده از شرط‌ها از خطا جلوگیری کنیم.

```
from os import path

if path.exists('note2.txt'):
    print('yes')
```

* با دستور **os.remove** از **os** می‌توانیم یک فایل را حذف کنیم.

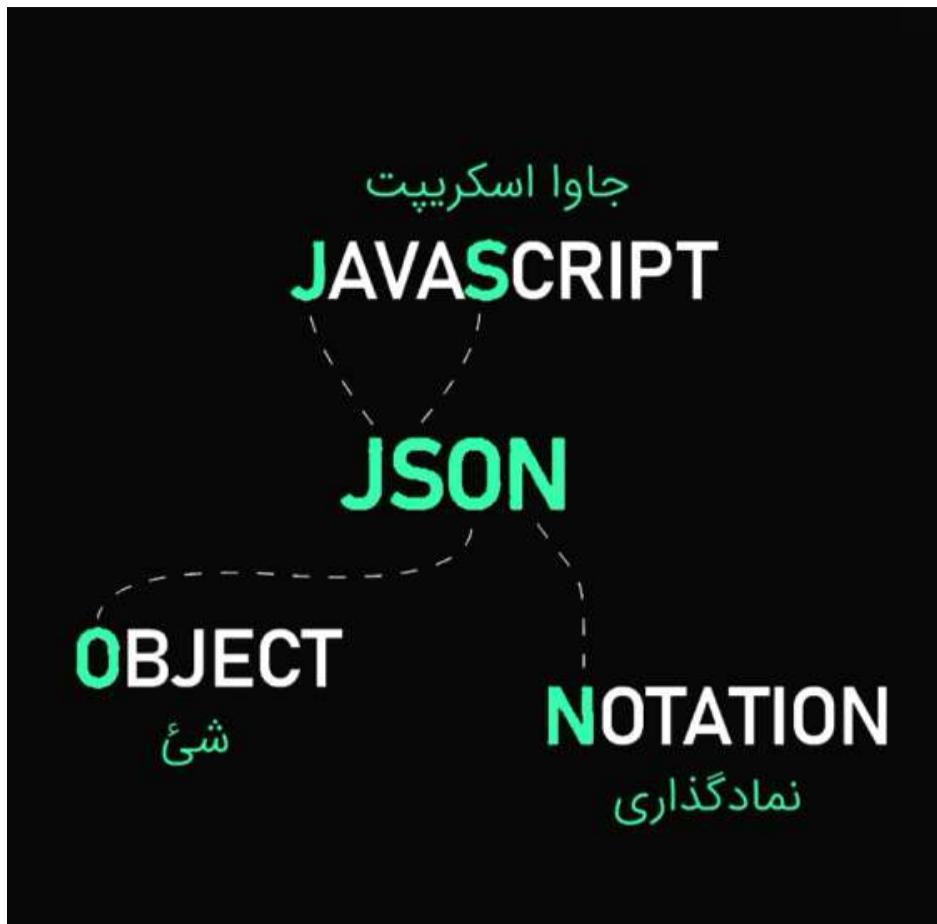
```
import os

os.remove('note.txt')
f = open('note.txt', 'r')
f.read()
```

```
f = open('note.txt', 'r')
^^^^^^^^^^^^^^^^^^^^^
FileNotFoundError: [Errno 2] No such file or directory: 'note.txt'
```

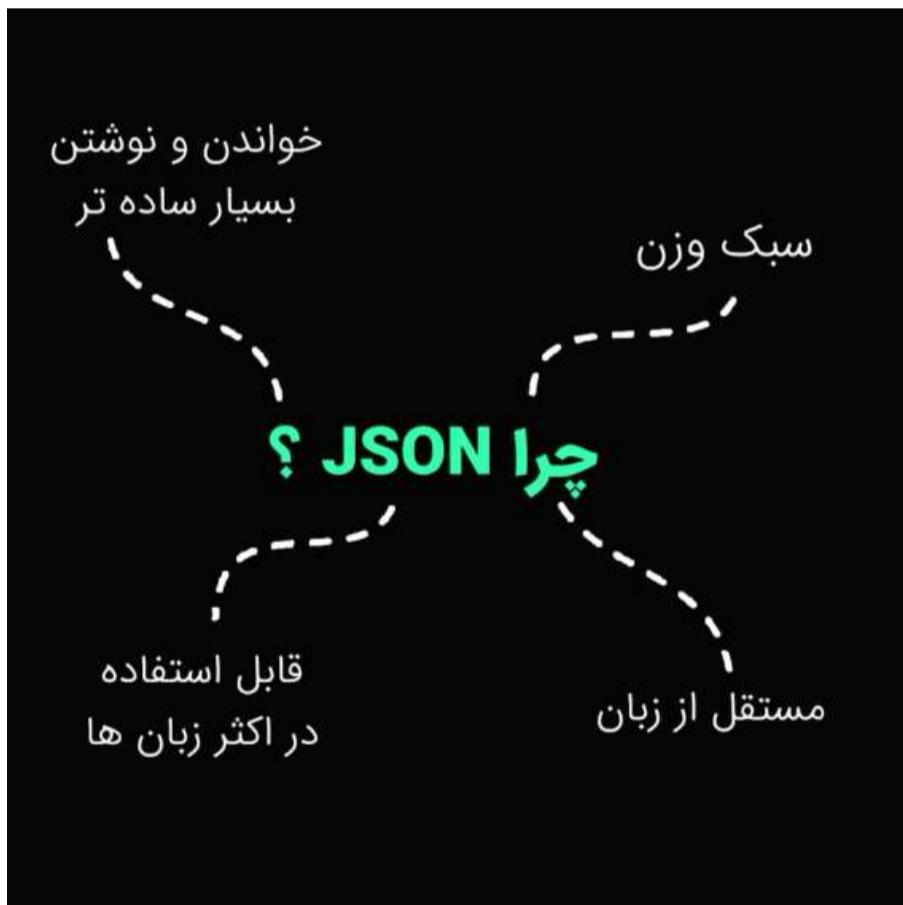
درس ۱۱: تکالیف مبحث فایل

۱. برنامه‌ای بنویسید که یک ورودی از کاربر می‌گیرد و در فایلی به نام "output.txt" ذخیره می‌کند. برنامه باید تا زمانی که کاربر کلمه exit را وارد نکرده است، به گرفتن ورودی ادامه دهد.
۲. یک برنامه پایتون بنویسید که محتویات یک فایل به نام "data.txt" را بخواند و هر خط را پس از حذف فاصله‌های اضافی بین کلمات چاپ کند.
۳. برنامه‌ای بنویسید که محتوای فایلی به نام "sample.txt" را بخواند و تکرار هر کلمه را بشمارد و سپس کلمه و تعداد آن را چاپ کنید. (برای این تمرین می‌توانید از اینترنت و مازول re کمک بگیرید.)
۴. برنامه‌ای بنویسید که محتویات یک فایل دیگری به نام "source.txt" را در فایل دیگری به اسم "destination.txt" کپی کند.
۵. برنامه‌ای بنویسید که از کاربر چندین خط متن را بگیرید و هر خط را به فایلی "notes.txt" اضافه کند. (فایل خالی نیست و از قبل دارای متن است). برنامه تا زمانی که کاربر یک خط خالی نکرده است باید ادامه پیدا کند.
۶. برنامه‌ای بنویسید که محتویات یک فایل به نام "numbers.txt" که شامل اعداد صحیح (در هر خط یک عدد) است را بخواند و مجموع و میانگین اعداد را محاسبه کند.
۷. برنامه‌ای بنویسید که محتویات فایلی به نام "document.txt" را بخواند و عملیات جستجو و جایگزینی را انجام دهد. برنامه باید از کاربر بخواهد که کلمه‌ای را برای جستجو و کلمه‌ای را برای جایگزینی آن وارد کند. پس از جایگزینی، محتوای به روز شده را در یک فایل جدید با نام "update_document.txt" ذخیره کنید.
۸. برنامه‌ای بنویسید که فایلی به نام "file_to_delete.txt" را حذف کند. قبل از حذف، باید بررسی کند که آیا فایل وجود دارد یا خیر.



JSON چیست؟

جیسون یک قالب استاندارد باز است که امکان تبادل داده ها در وب با استفاده از جفت های خصوصیت-کلید را ممکن ساخته است. قراردادهای مورد استفاده ای جیسون برای تمامی برنامه نویسان از جمله برنامه نویس جاوا اسکریپت، پرل، پایتون، جاوا، سی پلاس پلاس و ... شناخته شده است



نمونه ای از جیسون

exemple.json

```
{  
  "age": 19,  
  "name": "Hema",  
  "isWebDeveloper": true,  
  "igUser": "@ItsHemaPie",  
  "hobbies": ["Coding", "BasketBall"]  
}
```

* چون همهی زبان‌های برنامه نویسی قواعد نوشتن **JSON** را می‌دانند، این زبان‌ها به راحتی داده‌های ذخیره شده در **JSON** را به نوع‌های خاص خودشان تبدیل می‌کنند و بر عکس.

* در **جاوا اسکریپت** برخلاف پایتون انواع مختلف اعداد وجود ندارد.

python	-	javascript
int/float	-	number

* در **جاوا اسکریپت** برخلاف پایتون بولین با حرف کوچک **کوچک** نوشته می‌شود.

python	-	javascript
True/False	-	true/false

* در **جاوا اسکریپت** به جای **None** در پایتون از **null** استفاده می‌شود.

python	-	javascript
None	-	null

* اشیاء در **جاوا اسکریپت** همان **دیکشنری‌ها** در پایتون هستند.

python	-	javascript
dict	-	object

* آرایه‌ها در **جاوا اسکریپت** همان لیست‌ها یا تاپل‌ها در پایتون هستند.

python	-	javascript
list/tuple	-	array

* رشته‌ها در **جاوا اسکریپت** همان رشته‌ها در پایتون هستند.

درس ۱۳: کار با فایل جیسون (JSON)

* می‌توانیم **جیسون** را در پایتون وارد کنیم.

```
import json
```

* برای تبدیل یک رشته در **جیسون** به یک داده در پایتون از **loads** استفاده می‌کنیم.

```
import json

jn_str = '4'
py_type = json.loads(jn_str)

print(jn_str, '----', type(jn_str)) # -> 4 ---- <class 'str'>
print(py_type, '----', type(py_type)) # -> 4 ---- <class 'int'>
```

* باید قوانین نوشتمن جیسون را رعایت کنیم.

```
import json

jn_str = 'True'
py_type = json.loads(jn_str)
```

```
raise JSONDecodeError("Expecting value", s, err.value) from None
json.decoder.JSONDecodeError: Expecting value: line 1 column 1 (char 0)
```

* برای حل این خطا باید **بولین** را با حرف کوچک و مطابق قوانین جیسون بنویسیم.

```
import json

jn_str = 'true'
py_type = json.loads(jn_str)

print(jn_str, '----', type(jn_str)) # -> true ---- <class 'str'>
print(py_type, '----', type(py_type)) # -> True ---- <class 'bool'>
```

* برای تبدیل به **رشته** در پایتون باید رشته را در جیسون داخل دابل کتیشن ("") قرار دهیم.

```
import json

jn_str = '"hello"'
py_type = json.loads(jn_str)

print(jn_str, '----', type(jn_str)) # -> "hello" ---- <class 'str'>
print(py_type, '----', type(py_type)) # -> hello ---- <class 'str'>
```

* رشته در جیسون را به لیست در پایتون نیز می‌توانیم تبدیل کنیم.

```
import json

jn_str = '[1, 2, 3, 4]'
py_type = json.loads(jn_str)

print(jn_str, '----', type(jn_str)) # -> [1, 2, 3, 4] ---- <class 'str'>
print(py_type, '----', type(py_type)) # -> [1, 2, 3, 4] ---- <class 'list'>
```

* رشته در جیسون را به دیکشنری در پایتون می‌توانیم تبدیل کنیم.

```
import json

jn_str = '{"a": 5, "b": 8}'
py_type = json.loads(jn_str)

print(jn_str, '----', type(jn_str)) # -> {"a": 5, "b": 8} ---- <class 'str'>
print(py_type, '----', type(py_type)) # -> [{"a": 5, "b": 8} ---- <class 'dict'>
```

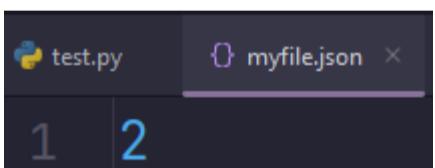
* در جیسون به جای None در پایتون از null استفاده می‌شود.

```
import json

jn_str = 'null'
py_type = json.loads(jn_str)

print(jn_str, '----', type(jn_str)) # -> null ---- <class 'str'>
print(py_type, '----', type(py_type)) # -> None ---- <class 'NoneType'>
```

* برای تبدیل یک فایل در جیسون به یک داده در پایتون از load استفاده می‌کنیم.



```
import json

with open('myfile.json', 'r') as jf:
    s = json.load(jf)

print(s) # -> 2
print(type(s)) # -> <class 'int'>
```

* برای تبدیل یک داده در پایتون به یک رشته در جیسون از `dumps` استفاده می‌کنیم.

```
import json

d = {'a': 5, 'b': 6}
j = json.dumps(d)

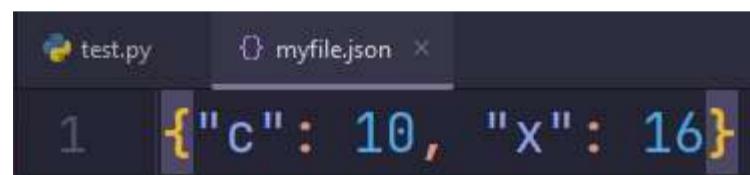
print(d, '----', type(d)) # -> {'a': 5, 'b': 6} ---- <class 'dict'>
print(j, '----', type(j)) # -> {"a": 5, "b": 6} ---- <class 'str'>
```

* برای تبدیل یک داده در پایتون به یک فایل در جیسون از `dump` استفاده می‌کنیم.

```
import json

d = {'c': 10, 'x': 16}
j = json.dumps(d)

with open('myfile.json', 'w') as jf:
    json.dump(d, jf)
```

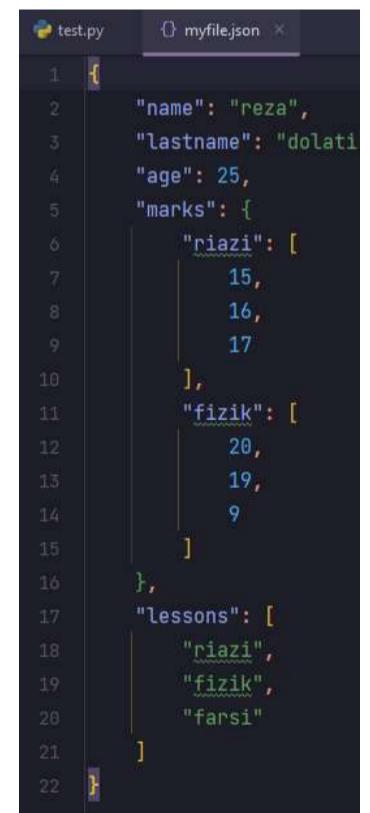


* برای استفاده از دندانه‌گذاری در جیسون باید از آرگومان `indent` در `dump` استفاده کنیم، مقدار `indent` فاصله دندانه‌ها را مشخص می‌کند که بهتر است 4 باشد.

```
import json

d = {'name': 'reza',
      'lastname': 'dolati',
      'age': 25,
      'marks': {'riazi': [15, 16, 17], 'fizik': [20, 19, 9]},
      'lessons': ['riazi', 'fizik', 'farsi']
     }

with open('myfile.json', 'w') as jf:
    json.dump(d, jf, indent=4)
```



* برای مرتب سازی کلیدها در جیسون باید از آرگومان `sort_keys=True` استفاده کنیم، مقدار آن اگر باشد کلیدها را براساس حروف الفبا مرتب می‌کند.

```
import json

d = {'name': 'reza',
      'lastname': 'dolati',
      'age': 25,
      'marks': {'riazi': [15, 16, 17], 'fizik': [20, 19, 9]},
      'lessons': ['riazi', 'fizik', 'farsi']
    }

with open('myfile.json', 'w') as jf:
    json.dump(d, jf, indent=4, sort_keys=True)
```

* برای جداسازی اشیا در جیسون باید از آرگومان `separators` در `dump` استفاده کنیم، که مقدار اول مشخص می‌کند که با چه چیزی اشیا را از هم جدا کند که پیش‌فرض آن `(,)` است و مقدار دوم مشخص می‌کند که با چه چیزی کلید و مقدار را از هم جدا کند که پیش‌فرض آن `دو نقطه (::)` است.

```
import json

d = {'name': 'reza',
      'lastname': 'dolati',
      'age': 25,
      'marks': {'riazi': [15, 16, 17], 'fizik': [20, 19, 9]},
      'lessons': ['riazi', 'fizik', 'farsi']
    }

with open('myfile.json', 'w') as jf:
    json.dump(d, jf, indent=4, separators='//,->')
```

* وقتی داده پایتون را به جیسون تبدیل می‌کنیم، به آن **serialize** می‌گویند.

* وقتی جیسون را به نوع داده پایتون تبدیل می‌کنیم، به آن **deserialize (pars)** می‌گویند.

* اگر بخواهیم چند داده را همزمان از پایتون به جیسون انتقال دهیم بهتر است که آن‌ها را داخل یک دیکشنری قرار دهیم.

```
import json

name = 'reza'
age = 25
marks = {'riazi': [15, 16, 17], 'fizik': [20, 19, 9]}
lessons = ['fizik', 'honor', 'riazi']
b = False

d = {'name': name, 'age': age,
      'marks': marks, 'lessons': lessons, 'b': b}
with open('myfile.json', 'w') as jf:
    json.dump(d, jf, indent=4)
```

```
1 "name": "reza",
2 "age": 25,
3 "marks": {
4     "riazi": [
5         15,
6         16,
7         17
8     ],
9     "fizik": [
10        20,
11        19,
12        9
13     ],
14 },
15 "lessons": [
16     "fizik",
17     "honor",
18     "riazi"
19 ],
20 },
21 "b": false
22 }
```

* اگر بخواهیم چند داده را همزمان از جیسون به پایتون انتقال دهیم بهتر است به صورت زیر عمل کنیم.

```
1 import json
2
3 with open('myfile.json') as jf:
4     p = json.load(jf)
5
6
7 name = p['name']
8 age = p['age']
9 marks = p['marks']
10 lessons = p['lessons']
11 b = p['b']
12
13
14 print(name)
15 print(age)
16 print(marks)
17 print(lessons)
18 print(b)
```

Run test

reza
25
{'riazi': [15, 16, 17], 'fizik': [20, 19, 9]}
['fizik', 'honor', 'riazi']
False

درس ۱۴: کار با فایل CSV

- * فایل‌های CSV (comma separated value) فایل‌هایی هستند که مقادیر آن با کاما (,) از یکدیگر جدا شده‌اند.
- * فایل‌های CSV معمولاً برای داده‌های ساختار یافته جدولی استفاده می‌شوند.



```
test.py itis.csv
1 "sepal.length","sepal.width","petal.length","petal.width","variety"
2 5.1,3.5,1.4,.2,"Setosa"
3 4.9,3,1.4,.2,"Setosa"
4 4.7,3.2,1.3,.2,"Setosa"
5 4.6,3.1,1.5,.2,"Setosa"
6 5,3.6,1.4,.2,"Setosa"
7 5.4,3.9,1.7,.4,"Setosa"
8 4.6,3.4,1.4,.3,"Setosa"
9 5,3.4,1.5,.2,"Setosa"
10 4.4,2.9,1.4,.2,"Setosa"
11 4.9,3.1,1.5,.1,"Setosa"
12 5.4,3.7,1.5,.2,"Setosa"
13 4.8,3.4,1.6,.2,"Setosa"
14 4.8,3,1.4,.1,"Setosa"
15 4.3,3,1.1,.1,"Setosa"
```

* می‌توانیم CSV را در پایتون وارد کنیم.

```
import csv
```

* برای خواندن اطلاعات CSV در پایتون می‌توانیم از reader استفاده کنیم که نتیجه لیست iterator می‌باشد، می‌توانیم از اطلاعات با استفاده از for یا next استفاده کنیم.

```
import csv

with open('iris.csv') as cf:
    data = csv.reader(cf)
    print(next(data))
    print(next(data))
```

```
['sepal.length\tsepal.width\tpetal.length\tpetal.width\tvariety']
['5.1\t3.5\t1.4\t2\tSetosa']
```

* برای خواندن اطلاعات CSV اگر از **DictReader** استفاده کنیم، اطلاعات را به صورت دیکشنری نمایش می‌دهد.

```
import csv

with open('iris.csv') as cf:
    data = csv.DictReader(cf)
    for i in data:
        print(i)
```

```
{'sepal.length\tsepal.width\tpetal.length\tpetal.width\tvariety': '5.1\t3.5\t1.4\t.2\tSetosa'}
{'sepal.length\tsepal.width\tpetal.length\tpetal.width\tvariety': '4.9\t3\t1.4\t.2\tSetosa'}
{'sepal.length\tsepal.width\tpetal.length\tpetal.width\tvariety': '4.7\t3.2\t1.3\t.2\tSetosa'}
{'sepal.length\tsepal.width\tpetal.length\tpetal.width\tvariety': '4.6\t3.1\t1.5\t.2\tSetosa'}
{'sepal.length\tsepal.width\tpetal.length\tpetal.width\tvariety': '5\t3.6\t1.4\t.2\tSetosa'}
{'sepal.length\tsepal.width\tpetal.length\tpetal.width\tvariety': '5.4\t3.9\t1.7\t.4\tSetosa'}
{'sepal.length\tsepal.width\tpetal.length\tpetal.width\tvariety': '4.6\t3.4\t1.4\t.3\tSetosa'}
['sepal.length\tsepal.width\tpetal.length\tpetal.width\tvariety': '15\t3.7\t1.5\t.2\tSetosa']
```

* برای نوشتن داده‌های پایتون در فایل CSV ابتدا باید یک شی **writer** ایجاد کنیم.

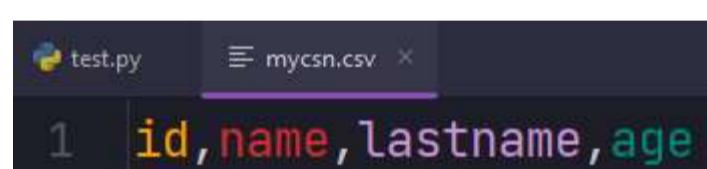
```
import csv

with open('mycsn.csv', 'w') as cf:
    writer = csv.writer(cf)
```

* با استفاده از **writerow** می‌توانیم یک سطر در فایل CSV بنویسیم

```
import csv

with open('mycsn.csv', 'w') as cf:
    writer = csv.writer(cf)
    writer.writerow(['id', 'name', 'lastname', 'age'])
```



* با استفاده از `writerow` می‌توانیم چند سطر در فایل CSV بنویسیم.

```
import csv

with open('mycsn.csv', 'w') as cf:
    writer = csv.writer(cf)
    writer.writerow(['id', 'name', 'lastname', 'age'])
    data = [['ali27', 'Ali', 'Ahmadi', '27'],
            ['#mammad', 'Mohammad', 'Rezaei', '24'],
            ['hasssaaaan', 'Hassan', 'Hassani', '22'],
            ['dfj;ls', 'Reza', 'Moradi', '29'],
            ]
    writer.writerows(data)
```

```
test.py mycsn.csv
1 id,name,lastname,age
2
3 ali27,Ali,Ahmadi,27
4
5 #@mammad,Mohammad,Rezaei,24
6
7 hasssaaaan,Hassan,Hassani,22
8
9 dfj;ls,Reza,Moradi,29
10
```

* برای حذف فاصله می‌توانیم از `open` در `newline` استفاده کنیم.

```
import csv

with open('mycsn.csv', 'w', newline='') as cf:
    writer = csv.writer(cf)
    writer.writerow(['id', 'name', 'lastname', 'age'])
    data = [['ali27', 'Ali', 'Ahmadi', '27'],
            ['#mammad', 'Mohammad', 'Rezaei', '24'],
            ['hasssaaaan', 'Hassan', 'Hassani', '22'],
            ]
    writer.writerows(data)
```

```
test.py mycsn.csv
1 id,name,lastname,age
2 ali27,Ali,Ahmadi,27
3 #@mammad,Mohammad,Rezaei,24
4 hasssaaaan,Hassan,Hassani,22
```

درس ۱۵: تکالیف مبحث CSV و JSON

۱. تابعی بنویسید که رشته JSON داده شده را `parse` کند و دیکشنری شامل داده را برگرداند.
۲. تابعی بنویسید که یک رشته JSON و یک کلید را به عنوان ورودی می‌گیرد و مقدار مربوط به آن کلید را برابر می‌گرداند.
۳. یک تابع پایتون بنویسید که یک رشته JSON و یک جفت کلید – مقدار را به عنوان ورودی می‌گیرد و JSON را با به روز رسانی یا اضافه کردن جفت کلید – مقدار تغییر می‌دهد.
۴. یک تابع پایتون بنویسید که یک رشته JSON را می‌گیرد یک دیکشنری حاوی تعداد هر مقدار منحصر به فرد در JSON را برمی‌گرداند.

مثال: ورودی:
'[{"color": "red"}, {"color": "blue"}, {"color": "red"}, {"color": "green"}]'

خروجی:
{'red': 2, 'blue': 1, 'green': 1}
۵. یک تابع پایتون بنویسید که دو رشته JSON را به عنوان ورودی می‌گیرد و یک رشته جدید JSON را که ترکیبی از هر دو ورودی است برمی‌گرداند.
۶. یک تابع پایتون بنویسید که یک رشته JSON را به عنوان ورودی می‌گیرد و آن را به فرمت CSV تبدیل می‌کند.
۷. یک تابع پایتون بنویسید که لیستی از دیکشنری‌ها را می‌گیرد و داده‌ها را در یک فایل CSV می‌نویسد.
۸. یک تابع پایتون بنویسید که مسیر فایل CSV و نام ستون را به عنوان ورودی می‌گیرد و مجموع تمام مقادیر عددی آن ستون را محاسبه می‌کند.

درس ۱: نیم نگاهی به مدیریت استثنای فصل بعد

* خطاهای ساختاری (SyntaxError) قبل از اجرای برنامه رخ می‌دهند و مربوط به رعایت نکردن قواعد نوشتاری پایتون می‌باشند.



A screenshot of the Visual Studio Code interface. The left pane shows a Python file named 'test.py' with the following code:

```
1 while True
2     print('hi')
```

An arrow points to the closing brace of the first line. The right pane shows the 'Run' tab is active, and the output window displays the following:

```
F:\python\pythonProject\venv\Scripts\python.exe -i "F:\python\pythonProject\test.py"
while True
^
SyntaxError: expected ':'
```

* به خطاهایی که پس از اجرای برنامه رخ می‌دهند، استثنا (RuntimeError) می‌گویند.

مثال: برنامه زیر از نظر سینتکس کاملا درست است اما در صورتی که کاربر یک عدد را بر صفر تقسیم کند، با خطای `ZeroDivisionError` مواجه می‌شویم که یک استثنای است (در برنامه نویسی تقسیم عدد بر صفر ممکن نیست).

```
x = int(input('x: '))
y = int(input('y: '))
print('x / y =', x / y)
```

مثال: برنامه زیر از نظر سینتکس کاملا درست است اما در هنگام اجرا با خطای **NameError** مواجه می‌شویم زیرا متغیر تعریف نشده است.

```
print(x + 5)
```

مثال: برنامه زیر از نظر سینتکس کاملا درست است اما در هنگام اجرا با خطای **TypeError** مواجه می‌شویم زیرا جمع رشته با عدد ممکن نیست.

```
print('2' + 3)
```

```
Traceback (most recent call last):
  File "F:\python\pythonProject\test.py", line 1, in <module>
    print('2' + 3)
               ~~~~~^~~
TypeError: can only concatenate str (not "int") to str
```

مثال: برنامه زیر از نظر سینتکس کاملا درست است اما در صورتی که کاربر یک رشته غیر عددی را وارد کند، با خطای **ValueError** مواجه می‌شویم چون برنامه باید رشته را به عدد تبدیل کند.

```
x = int(input('x: '))
print(x + 5)
```

```
x: 5
Traceback (most recent call last):
  File "F:\python\pythonProject\test.py", line 1, in <module>
    x = int(input('x: '))
               ^^^^^^^^^^^^^^^^^^
ValueError: invalid literal for int() with base 10: 's'
```

* اگر با استثنای مواجه شویم برنامه متوقف می‌شود اما می‌توانیم این استثنای را مدیریت کنیم که در صورت رخداد آنها بازهم برنامه متوقف نشود و اجرا شود.

* با دستور **try** و **except** می‌توانیم استثنای را مدیریت کنیم، پس از **try** دستورات را می‌نویسیم و پس از **except** مشخص می‌کنیم که اگر با استثنای مواجه شد چگونه عمل کند.

```
x = int(input('x: '))
y = int(input('y: '))

try:
    print('x / y =', x / y)
except:
    print('Error!')
print('End!')
```

```
x: 4
y: 0
Error!
End!
```

* اگر در بدنی **try** استثنا رخ دهد، بقیه‌ی دستورات بدن را اجرا نمی‌کند و دستورات **except** را اجرا می‌کند.

```
x = int(input('x: '))
y = int(input('y: '))

try:
    print('x / y =', x / y)
    print('+'*15)
except:
    print('Error!')
print('End!')
```

```
x: 4
y: 0
Error!
End!
```

* اگر بدانیم که چه استثنایی ممکن است رخ دهد، در **except** می‌توانیم استثنا را بنویسیم به این صورت که اگر استثنا رخ داد چگونه عمل کند.

```
x = int(input('x: '))
y = int(input('y: '))

try:
    print('x / y =', x / y)
except ZeroDivisionError:
    print('ZeroDivisionError!')
    y = int(input('y: '))
    print('x / y =', x / y)
print('End!')
```

```
x: 4
y: 0
ZeroDivisionError!
y: 2
x / y = 2.0
End!
```

* می‌توانیم به تعداد دلخواه **except** بنویسیم و چند استثنا را همزمان مدیریت کنیم.

```
x = int(input('x: '))
y = int(input('y: '))

try:
    print('x / y =', x / y)
except ZeroDivisionError:
    print('ZeroDivisionError!')
    y = int(input('y: '))
    print('x / y =', x / y)
except ValueError:
    print('ValueError!')
except:
    print('UnknownError!')
print('End!')
```

* می‌توانیم برای چند استثنا یک **except** بنویسیم.

```
x = int(input('x: '))
y = int(input('y: '))

try:
    print('x / y =', x / y)
except (ZeroDivisionError, ValueError):
    print('Error!')
```

* می‌توانیم پس از **except** دستور **else** را نیز بنویسیم که زمانی اجرا می‌شود که با خطا مواجه نشویم.

```
x = int(input('x: '))
y = int(input('y: '))

try:
    print('x / y =', x / y)
except (ZeroDivisionError, ValueError):
    print('Error!')
else:
    print('End!')
```

```
x: 4
y: 2
x / y = 2.0
End!
```

* می‌توانیم با دستور **raise** استثنا بنویسیم، به این صورت که بعد از آن اسم استثنایی که می‌خواهیم رخ بدهد را می‌نویسیم.

```
x = int(input('x: '))
y = int(input('y: '))

try:
    print('x / y =', x / y)
    if y == 10:
        raise ValueError('raised Error!')
except (ZeroDivisionError):
    print('Error!')
```

```
x: 4
y: 10
x / y = 0.4
Traceback (most recent call last):
  File "F:\python\pythonProject\test.py", line 7, in <module>
    raise ValueError('raised Error!')
ValueError: raised Error!
```

* استثناء اتفاق بيفتد يا خير دستورات داخل **`finally`** در هر صورت اجرا می‌شوند.

```
x = int(input('x: '))
y = int(input('y: '))

try:
    print('x / y =', x / y)
    if y == 10:
        raise ValueError
except ZeroDivisionError:
    print('Error!')
finally:
    print('+' * 10)
```

```
x: 4
y: 10
x / y = 0.4
Traceback (most recent call last):
  File "F:\python\pythonProject\test.py", line 7, in <module>
    raise ValueError
ValueError
++++++
```

درس ۲: مفهوم شی گرایی

* شی (object) وظیفه و رفتار (behavior) خاصی دارد و دارای خصوصیاتی (attributes) نیز می‌باشد.



* هر شی از روی یک طرح ایجاد شده است که به آن کلاس (class) می‌گوییم.

* کلاس (class) یک طرح است که از روی آن می‌توانیم اشیا مختلف را بسازیم.



* برای پیاده سازی رفتار شی، داخل کلاس از تابع استفاده می‌کنیم که به آن متده (method) گفته می‌شود.

* برای پیاده سازی خصوصیات شی، داخل کلاس از متغیرها استفاده می‌کنیم.

* شی (object) یک نمونه (instance) از کلاس (class) است.

کلاس	رفتارها	صفات
حساب بانکی	واریز پول برداشت پول مشاهده موجودی	نام صاحب حساب شماره حساب میزان موجودی
دانشجو	محاسبه معدل انتخاب واحد	شماره دانشجویی تعداد واحد پاس شده رشته

کلاس	رفتارها	صفات
	چیدن از درخت گرفتن آب میوه	رنگ وزن اندازه

* در پایتون همه داده‌ها شی هستند، هر شی شناسه، نوع و مقدار دارد.

```
x = 5
y = 10
print(type(x)) # -> <class 'int'>
print(type(y)) # -> <class 'int'>
```

* شناسه اشیا غیرقابل تغییر است و با دستور `id` می‌توانیم آن را مشخص کنیم.

```
x = 5
print(id(x)) # -> 140722096235432
```

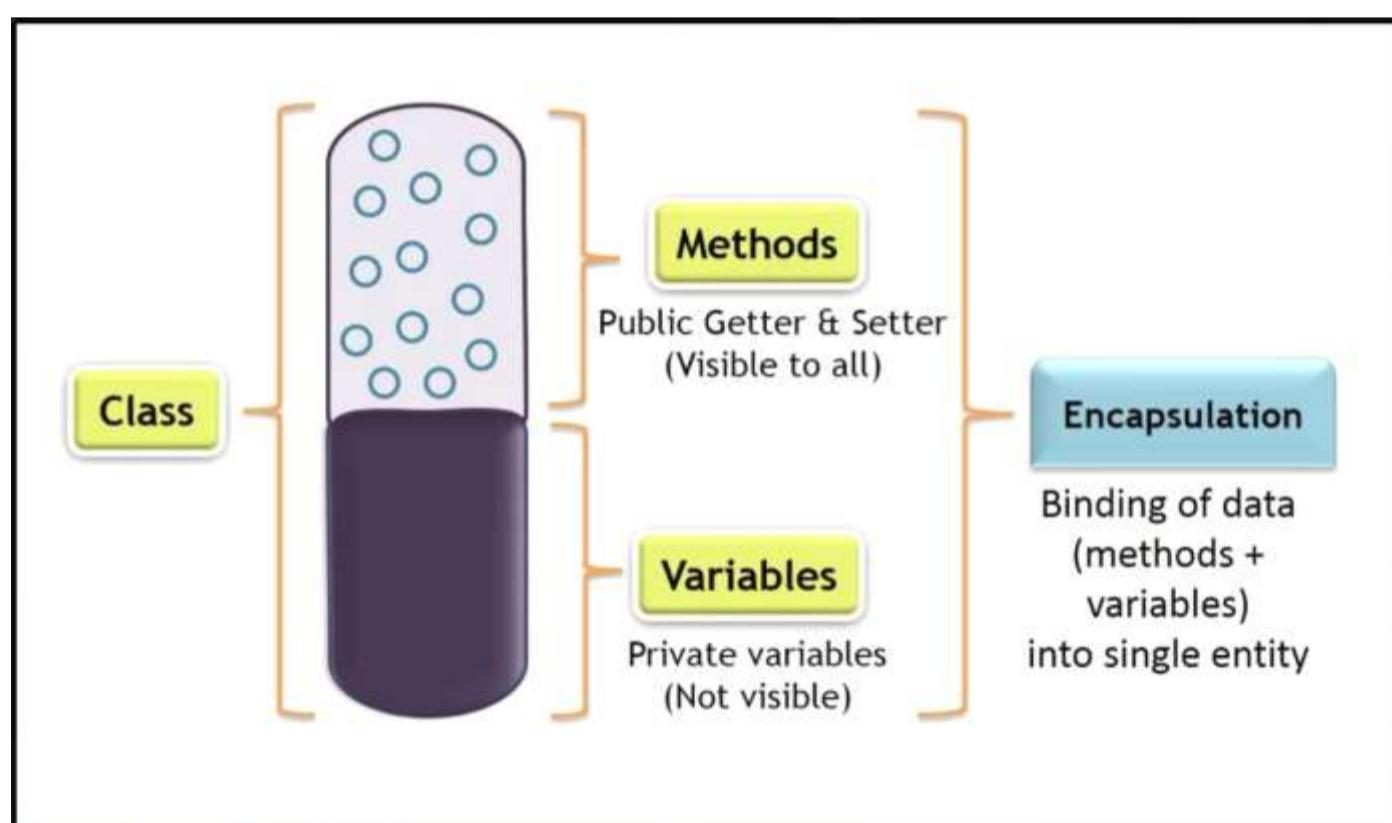
* اشیا در پایتون یا تغییر پذیرند یا غیرقابل تغییرند.

درس ۳: مفهوم کپسوله سازی، پنهان سازی داده و رابط

* کپسوله سازی (encapsulation) به معنی دسته بندی متدها و اtribووت‌هایی که آن متدها با آن کار می‌کنند.



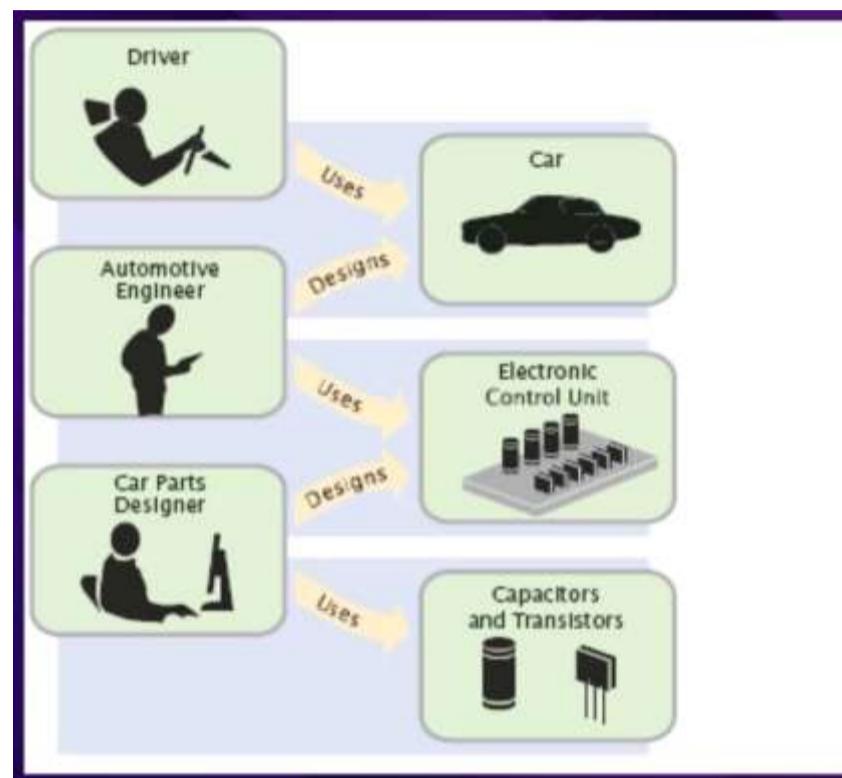
- * در هنگام ایجاد کلاس (class) از کپسوله سازی استفاده می‌کنیم.
- * به فرایندی که در آن داده‌ها را مخفی می‌کنیم و به کاربر اجازه دسترسی به آن‌ها را نمی‌دهیم، پنهان سازی داده (data hiding) می‌گویند.



- * برای تعامل کاربر با اشیا کاربر نیاز نیست که دستورات منجر به ایجاد آن شی را بداند و برای استفاده از آن فقط نیاز به رابط (interface) دارد.

درس ۴: مفهوم انتزاع

- * انتزاع به معنی توجه به کلیات، بدون توجه به جزئیات است.
- * انتزاع یک مفهوم است و وجود ندارد، فقط یک الگو (**concrete**) برای نمونه واقعی (**abstract**) است، حیوان یک مفهوم انتزاعی است و گربه یک نمونه از آن است.
- * انتزاع دارای سطوح مختلفی می باشد.

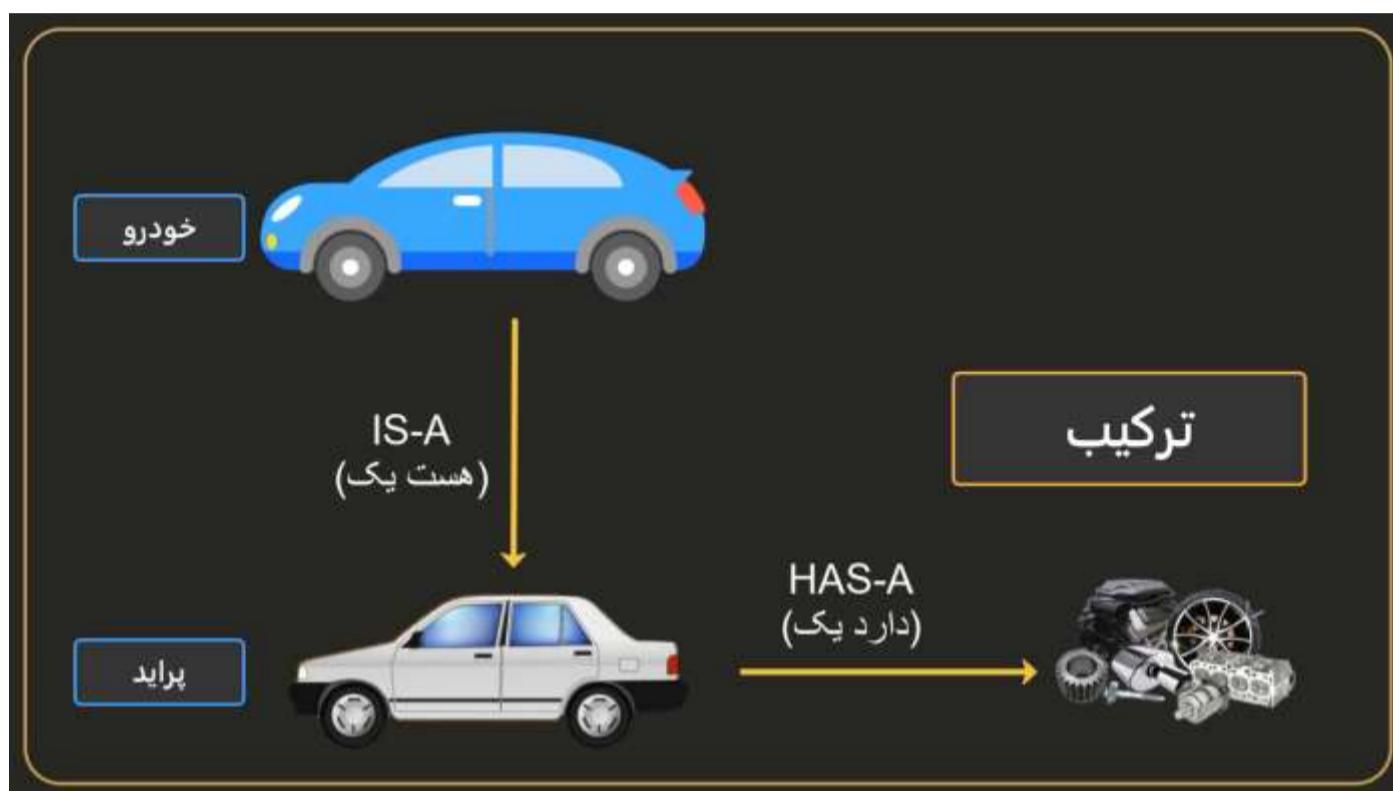


درس ۵: مفهوم رابطه، ترکیب و تجمع

* برای پیاده‌سازی انتزاع از وراثت و ترکیب استفاده می‌کنیم.

* ترکیب (composition) یعنی جمع‌آوری کردن چند شی باهم، برای ایجاد یک شی جدید. برای زمانی استفاده می‌شود که یک شی بخشی از شی دیگر باشد.

* پراید یک نمونه از خودرو است و پراید از آن ارث بری می‌کند اما موتور یک نمونه از پراید نیست بلکه پراید یک موتور دارد، به رابطه‌ی بین پراید و موتور ترکیب می‌گوییم.



* روابط بین اشیا شامل رابطه (association)، تجمع (aggregation) و ترکیب (composition) است.

* در رابطه (association) ارتباط بین اشیا خیلی کم است و وابستگی کمی باهم دارند، در این نوع رابطه معمولاً شی اول جزئی از شی دوم نیست.

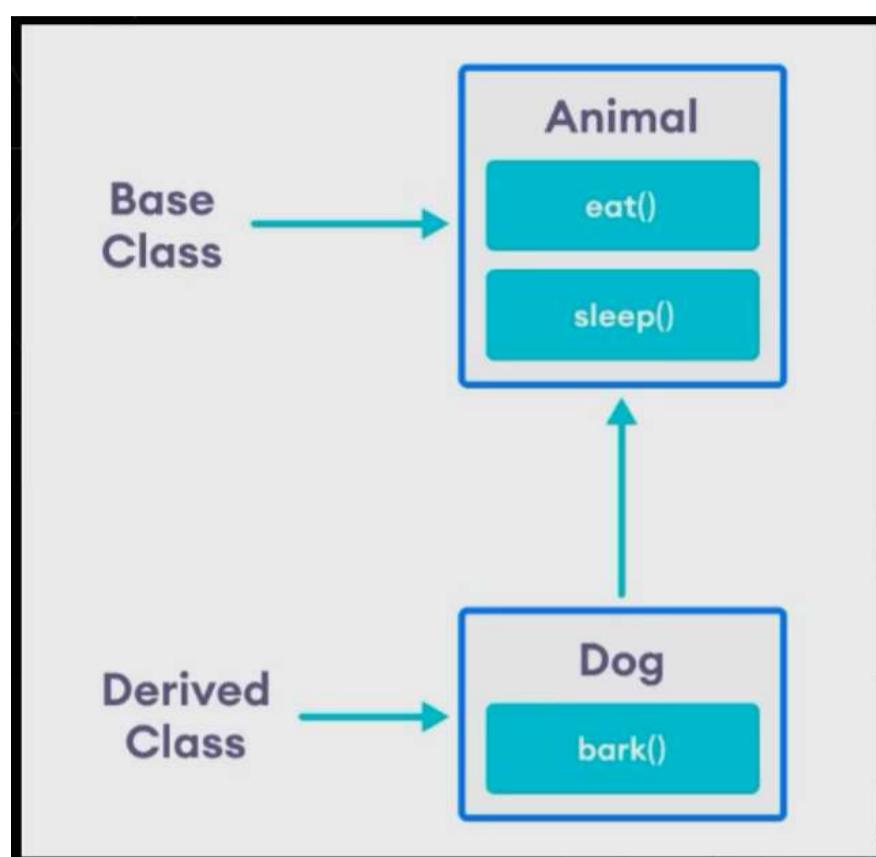
* تجمع (aggregation) یک نوع خاص رابطه (association) است که رابطه بین اشیا بیشتر است و رابطه جز به کل است و تعلق وجود دارد و اشیا کوچک‌تر متعلق به شی بزرگ‌تر هستند و وابستگی بین اشیا کم است و هر شی مستقل از دیگری است.

* ترکیب (composition) یک نوع خاص تجمع (aggregation) است که رابطه بین اشیا بیشتر است و به شدت به هم وابسته‌اند و اشیا مستقل از هم وجود ندارند.

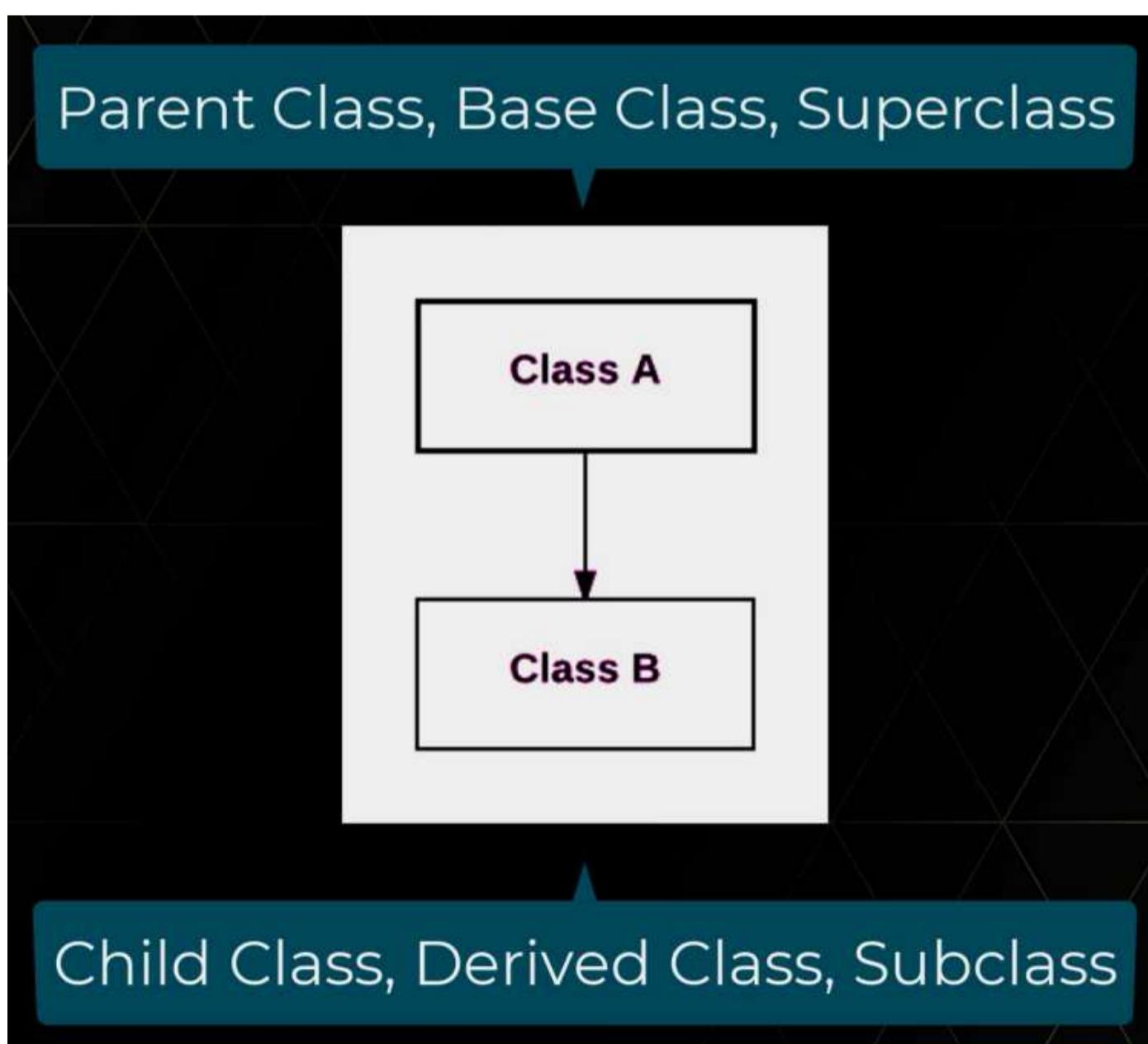
* در پایتون بیشتر از ترکیب (composition) استفاده می‌کنیم.

درس ۶: مفهوم وراثت و انواع ارث بری

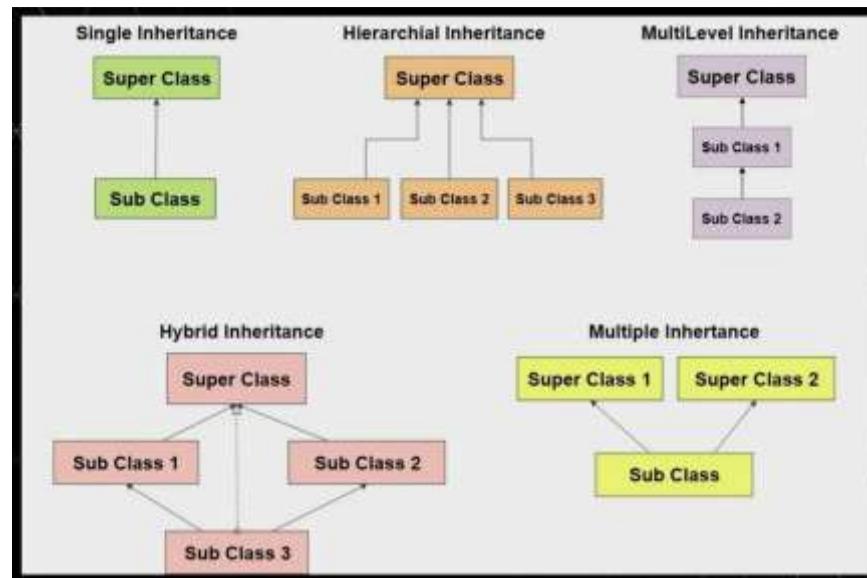
- * وراثت (inheritance) مهم‌ترین و پر استفاده‌ترین مفهوم در برنامه نویسی شی گرا است.
- * ارث بری یک کلاس از یک کلاس دیگر به این معنی است که آن کلاس همهی خصوصیت‌ها و رفتارهای کلاس دیگر را داشته باشد و علاوه بر آن نیز ممکن است خودش نیز ویژگی‌ها و خصوصیاتی داشته باشد.



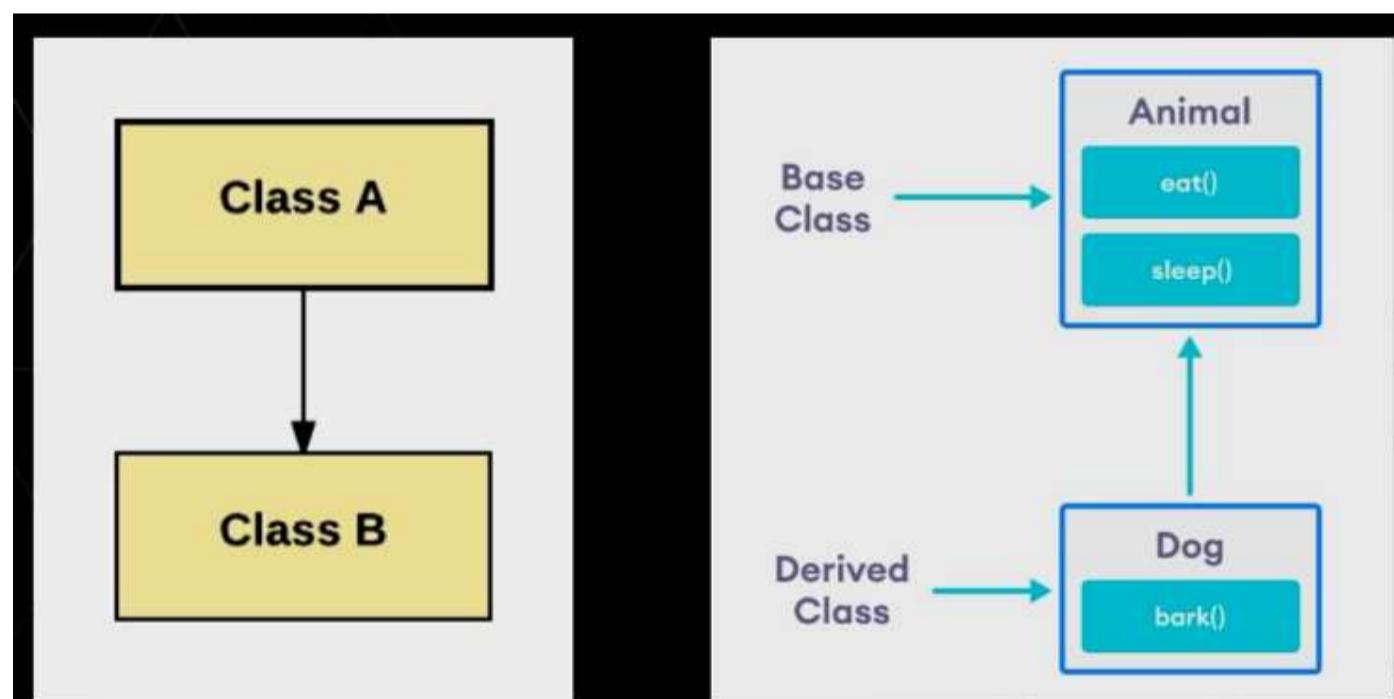
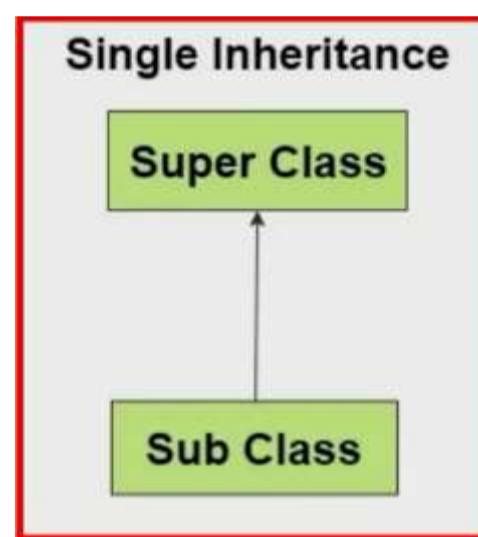
- * وراثت (inheritance) در برنامه نویسی قابلیت استفاده مجدد از کد را برای ما فراهم می‌کند.
- * همه کلاس‌ها در پایتون در نهایت از کلاس object ارث بری می‌کنند.



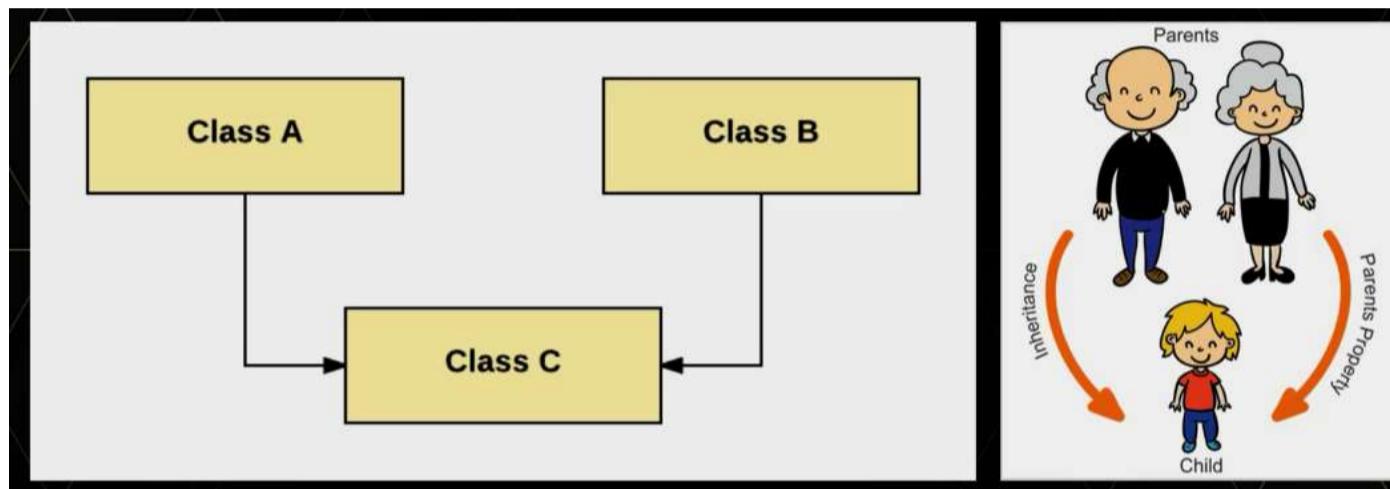
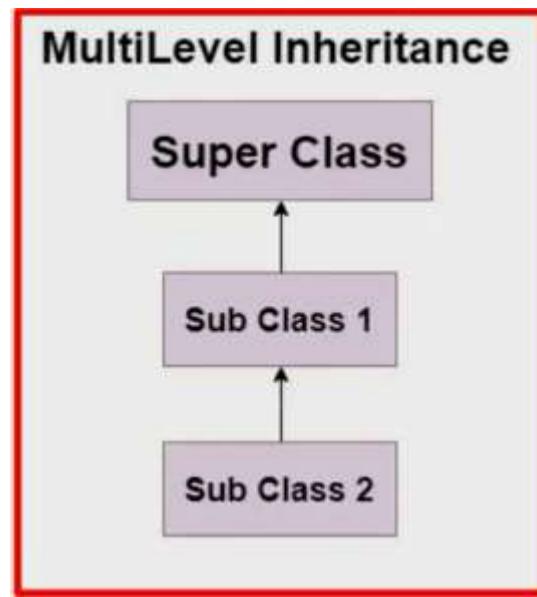
* وراثت (inheritance) ممکن است به شکل‌های مختلف پیاده سازی شود.



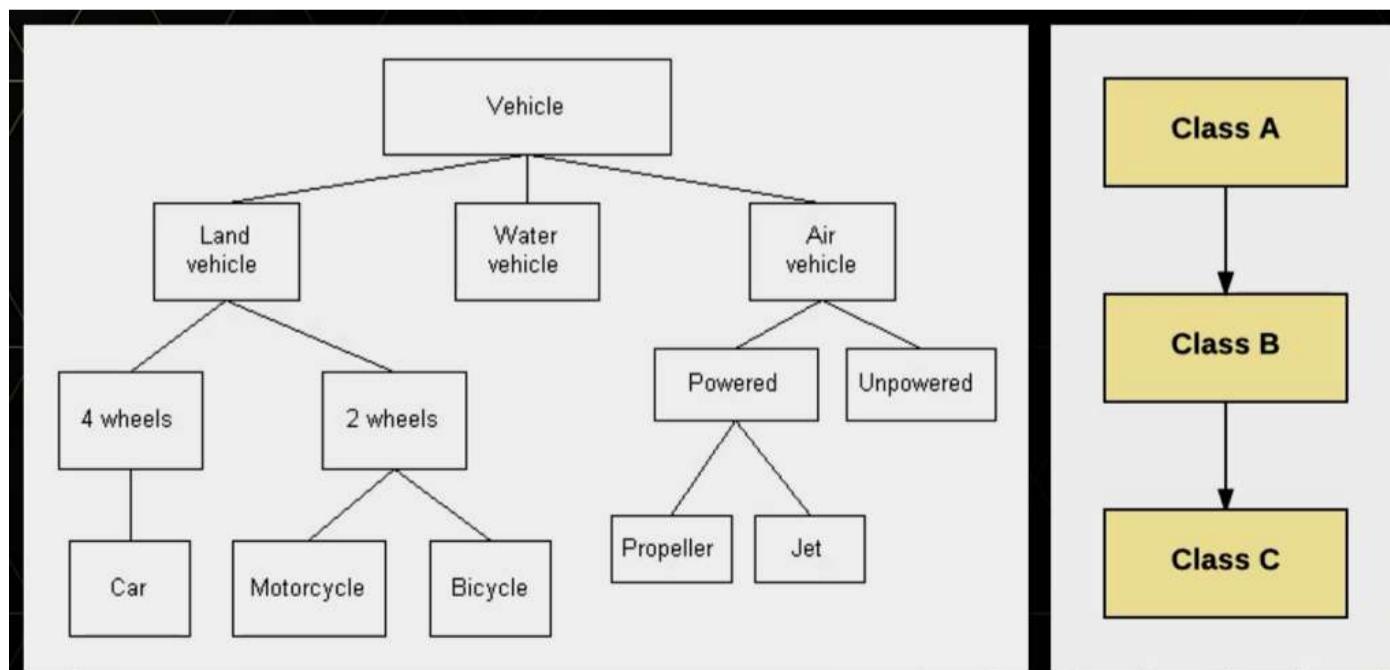
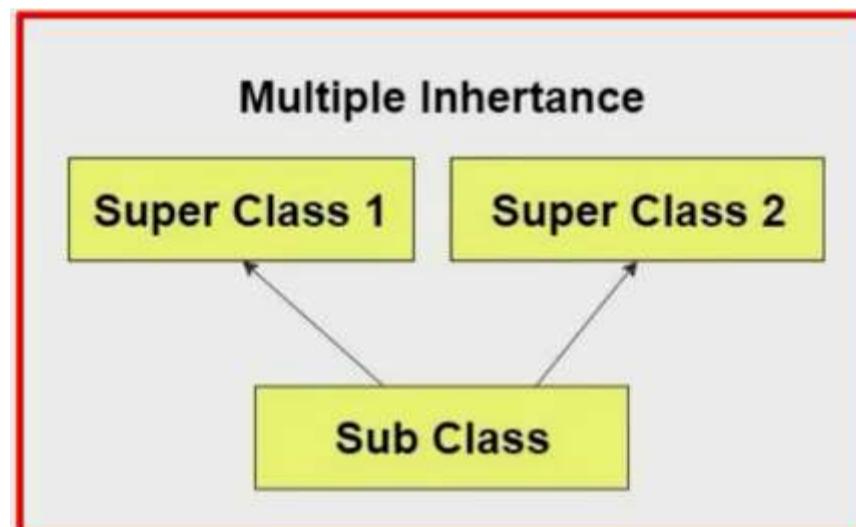
* وراثت یگانه (single inheritance) به معنی ارث بری یک کلاس مشتق شده از یک کلاس پایه است.



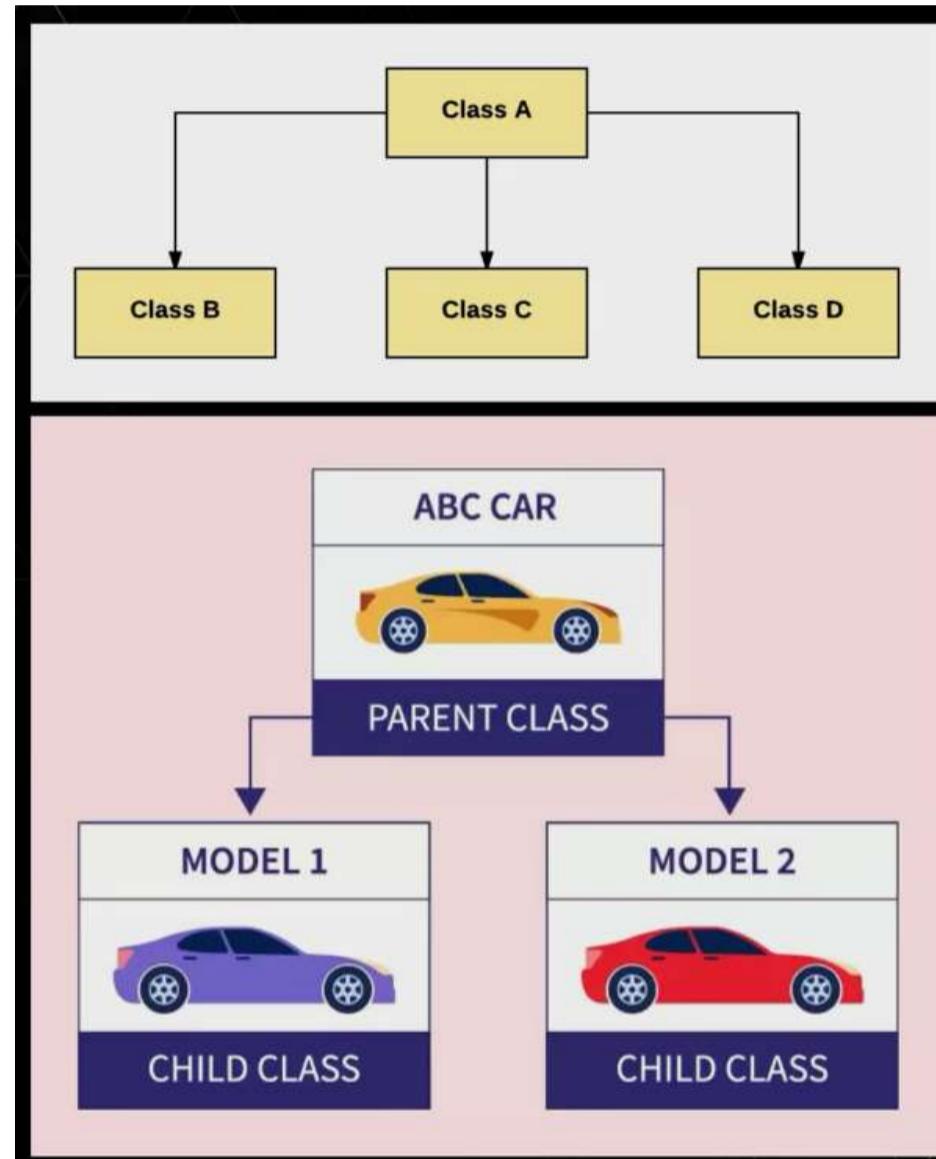
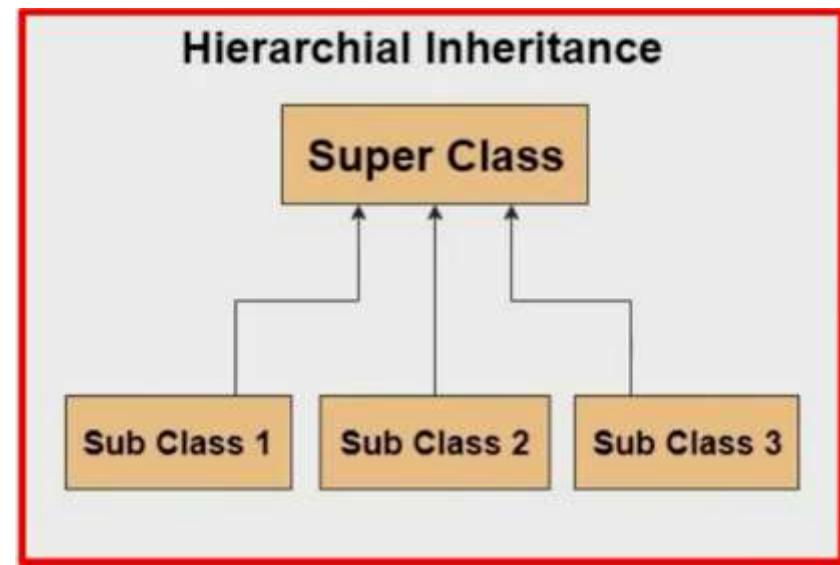
* وراثت چندگانه (multiple inheritance) به معنی ارث بری یک کلاس مشتق شده از چند کلاس پایه است.



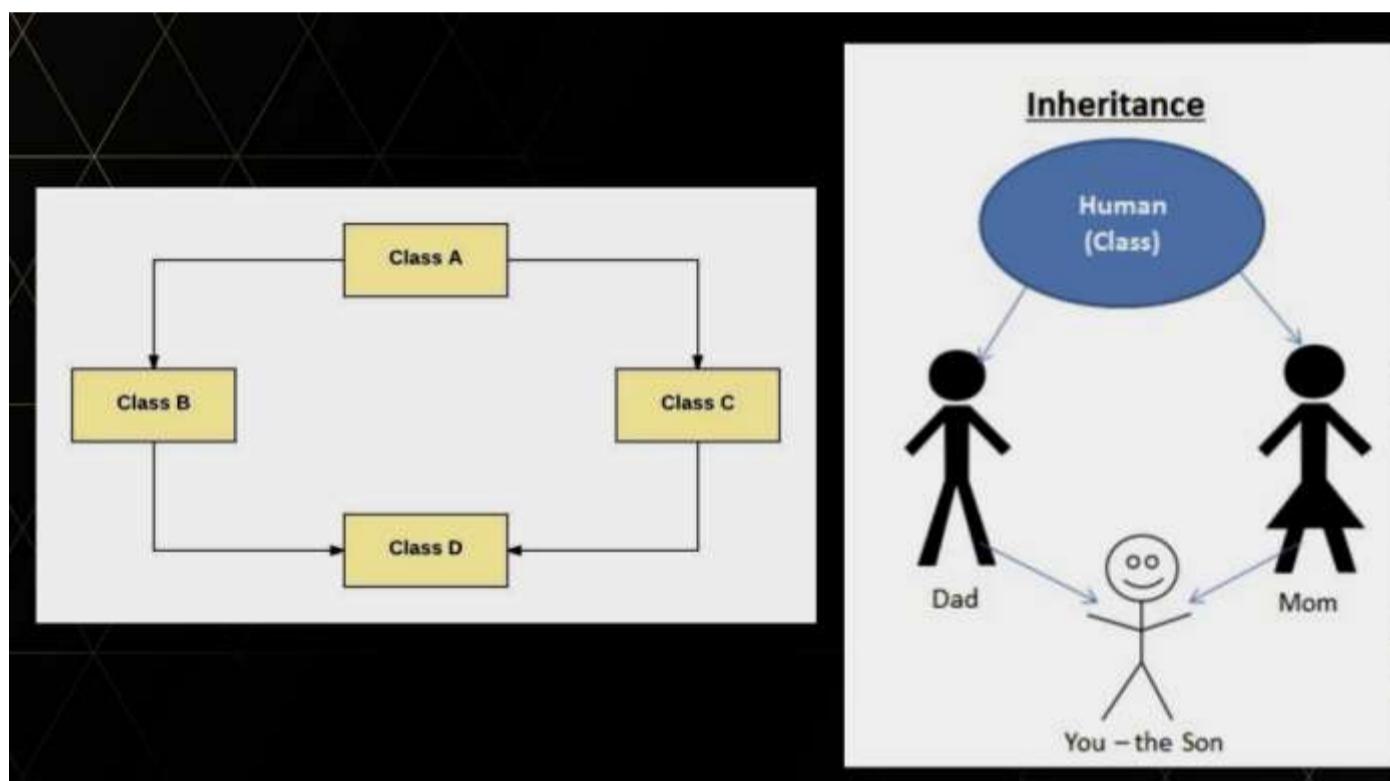
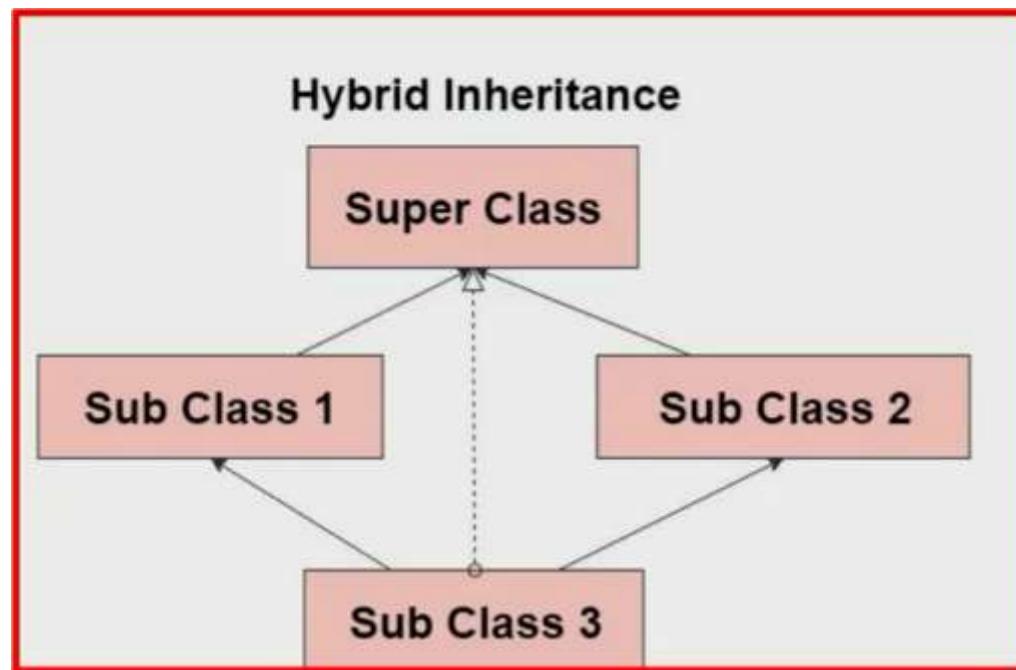
* وراثت چند سطحی (multilevel inheritance) یعنی اینکه یک کلاس از یک کلاس دیگر مشتق می‌شود که آن هم از یک کلاس دیگر مشتق شده است.



* وراثت سلسله مراتبی (**hierarchial inheritance**) به معنی ارث بری چند کلاس مختلف از یک کلاس پایه است.

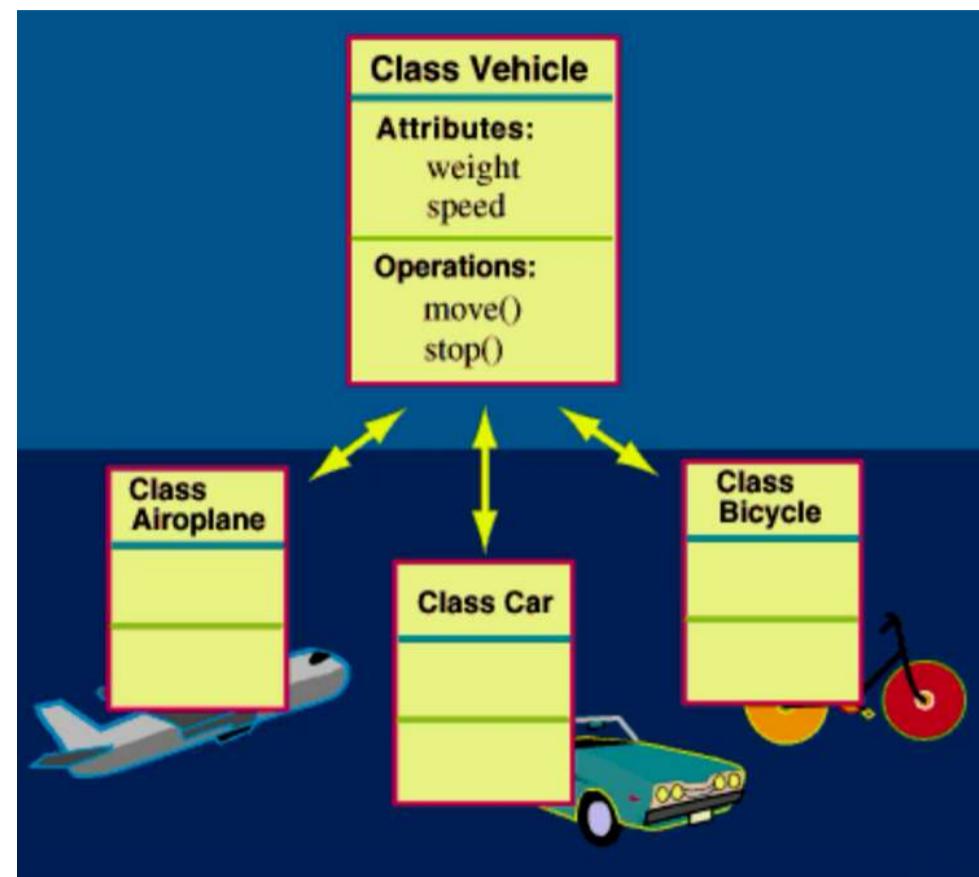


* وراثت ترکیبی (**hybrid inheritance**) که ممکن است از چند نوع مختلف وراثت ترکیب شده باشد.

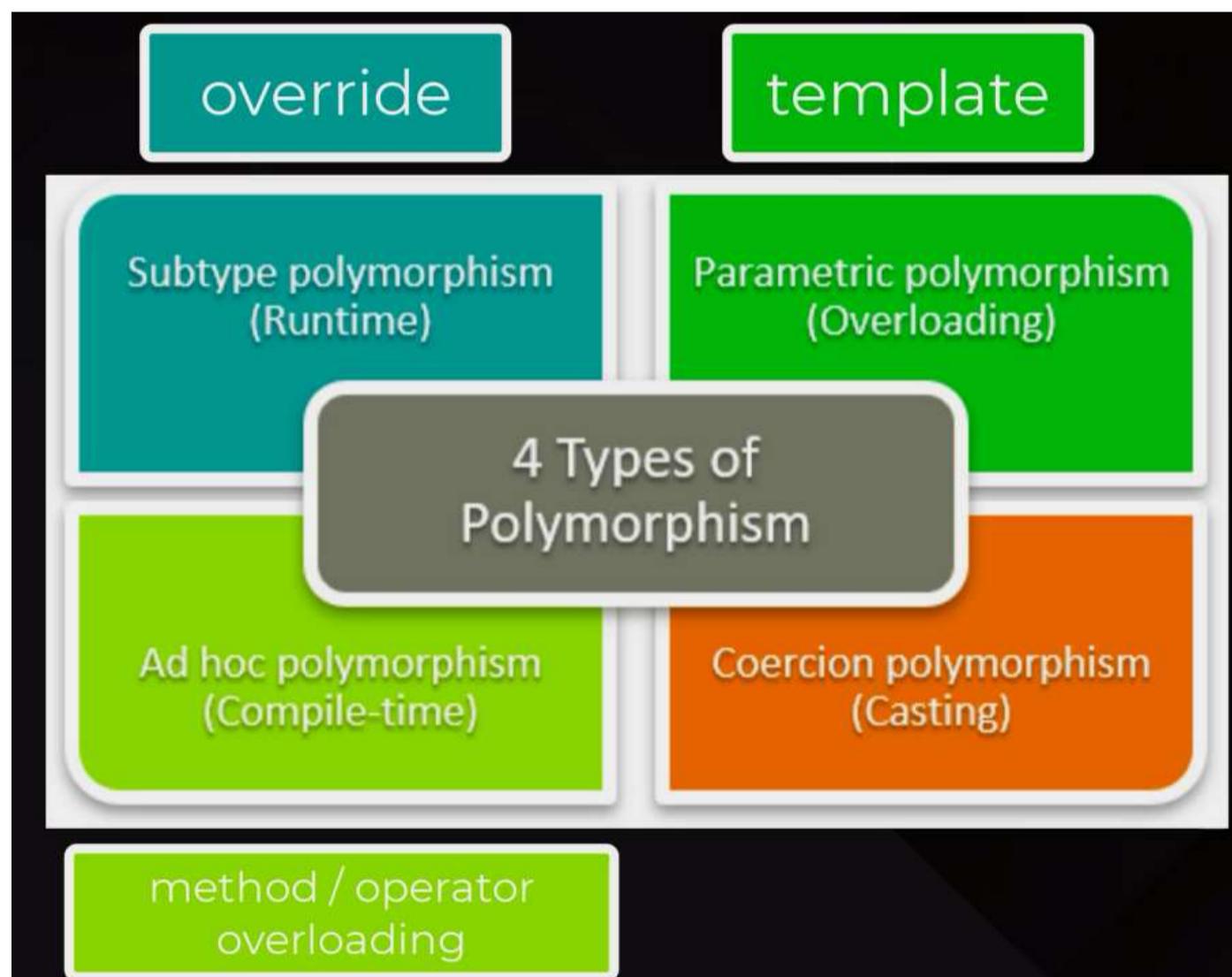


درس ۷: مفهوم پلی مورفیسم (چندریختی)

* چندریختی (polymorphism) به معنی این که کلاس های ارث بری شده یک رفتار را برای خودشان شخصی سازی (override) می کنند.



* انواع چندریختی (polymorphism) :



* در تایپ دهی اردکی (duck typing) خصوصیات و توابع شی معنای آن را مشخص می کنند و نه نوع و ... آن، در این تایپ دهی تمرکز بر روی رفتار شی است.

* پایتون از تایپ دهی اردکی (duck typing) استفاده می کند و در آن نوع شی مهم نیست بلکه خصوصیات و رفتار آن مهم است.

درس ۸: اشیا در پایتون

- * همه چیز در پایتون یک شی است.
- * هر شی حداقل باید نمونه‌ای از یک کلاس باشد.
- * یک متغیر، یک ارجاع به شی است.
- * وقتی یک مقدار را به یک متغیر اختصاص می‌دهیم، در حال ایجاد نمونه‌ای از کلاس‌های **built-in** پایتون هستیم.

```
x = 56
print(type(x)) # -> <class 'int'>
s = 'hello'
print(type(s)) # -> <class 'str'>
```

* در مثال بالا **X** یک شی از کلاس **int** و **s** یک شی از کلاس **str** پایتون هستند.

* برای نمایش اسم کلاس یک شی می‌توانیم از **__class__** نیز استفاده کنیم.

```
x = 56
print(x.__class__) # -> <class 'int'>
s = 'hello'
print(s.__class__) # -> <class 'str'>
```

* در هنگام جمع و سایر عملگرها در واقع در حال صدا زدن یک متده از آن کلاس هستیم.

```
x = 4
y = 6
print(x + y) # > 10
print(x.__add__(y)) # > 10
print(x * y) # > 24
print(x.__mul__(y)) # > 24
```

* کلاس (**class**) در پایتون نیز یک شی است و نمونه‌ای از کلاس **type** است.

```
class A:
    pass

print(A.__class__) # -> <class 'type'>
```

* دستور کلاس (**class**) کمک می‌کند که یک نوع (**type**) جدید ایجاد کنیم.

* همه‌ی کلاس‌هایی که در پایتون ایجاد می‌شوند از کلاس **object** ارث بری می‌کنند.

* با استفاده از **__base__** می‌توانیم مشخص کنیم که یک کلاس از چه کلاس‌هایی ارث بری می‌کند.

```
class A:
    pass

print(A.__bases__) # -> (<class 'object'>,)
print(type.__bases__) # -> (<class 'object'>,)
```

* در ساختار جدید پایتون، مفهوم کلاس برابر مفهوم **type** در نظر گرفته شده است.

* متغیر فقط یک اسم است که به شی می‌دهیم و خصوصیات مقدار ربطی به متغیر ندارد.

```
x = 4
y = 4
print(id(x)) # -> 140711494280072
print(id(y)) # -> 140711494280072
print(id(4)) # -> 140711494280072
```

درس ۹: ساخت کلاس و اضافه کردن `method` و `attribute` به آن

- * برای تعریف کلاس از کلمه کلیدی `class` استفاده می‌کنیم.
- * اسم کلاس بهتر است `PascalCase` باشد، یعنی حرف اول هر کلمه‌ای حرف بزرگ باشد و بین کلمات اندراسکور (`_`) قرار ندهیم.
- * در نوشتن کلاس، قوانین تورفتگی باید رعایت شود.

```
class PascalCase:  
    pass
```

- * برای ایجاد یک شی از کلاس، اسم کلاس را می‌نویسیم و در جلویش `پرانتزها ()` را باز می‌کنیم.

```
class Point:  
    pass
```

```
p1 = Point()  
p2 = Point()
```

- * اگر پرانتز نگذاریم در واقع شی ایجاد نمی‌کنیم بلکه به همان کلاس یک اسم دیگر می‌دهیم که باز هم می‌توانیم از آن شی ایجاد کنیم.

```
class Point:  
    pass
```

```
c = Point  
p1 = c()  
print(c) # -> <class '__main__.Point'>  
print(Point) # -> <class '__main__.Point'>  
print(id(c)) # -> 2411630400176  
print(id(Point)) # -> 2411630400176
```

- * اشیایی که از یک کلاس ایجاد می‌کنیم، یکی نیستند و در مکان‌های متفاوت از حافظه ذخیره می‌شوند.

```
class Point:  
    pass
```

```
p1 = Point()  
p2 = Point()  
  
print(id(p1)) # -> 1401766579088  
print(id(p2)) # -> 1401766579152  
print(p1 is p2) # -> False
```

* به اشیایی که ایجاد کردیم می‌توانیم با استفاده از نقطه(.) اtribut‌های مدنظرمان را اضافه کنیم، به این صورت که ابتدا اسم شی را می‌نویسیم و سپس نقطه می‌گذاریم و اسم دلخواه اtribut را می‌نویسیم و در نهایت مقدار می‌دهیم.

```
class Point:  
    pass  
  
p1 = Point()  
p2 = Point()  
  
# <object>.<attribute> = <value>  
p1.x = 5  
p1.y = 3  
print(p1.x) # -> 5  
print(p1.y) # -> 3
```

* برای اtributی که به شی می‌دهیم می‌توانیم یک راهنمایی بنویسیم که از چه نوعی است.

```
1 class Point:  
2     pass  
3  
4  
5 p1 = Point()  
6 p2 = Point()  
7  
8 p2.x: int = 'ldfj'  
9 Expected type 'int', got 'str' instead :
```

* به کلاسی که ایجاد کردیم می‌توانیم با استفاده از توابع رفتار اضافه کنیم، به این صورت که بعد از اسم تابع و داخل پرانتز self را می‌نویسیم.

* توابع اگر داخل کلاس تعریف شوند به آن‌ها متده (method) گفته می‌شود.

```
class Point:  
    def reset(self):  
        self.x = 0  
        self.y = 0  
  
p1 = Point()  
p2 = Point()  
p2.x = 4  
p2.y = 5  
print('x: ', p2.x)  
print('y: ', p2.y)  
p2.reset()  
print('-' * 5, 'after reset', '-' * 5)  
print('x: ', p2.x)  
print('y: ', p2.y)
```

```
x: 4  
y: 5  
----- after reset -----  
x: 0  
y: 0
```

- * تفاوت تابع و متده است که متده دارای **self** می باشد که به آن متغير نمونه (**instance variable**) گفته می شود.
- * با **self** به شی که ایجاد کردیم، توانایی مشاهده و تغییر دادن اtribووت هایش را می دهیم.
- * در واقع خود شی است و وقتی این متده را برای شی صدا می زنیم، آن شی را به متده می دهد.

```
class Point:
    def reset(self):
        print(self)
        self.x = 0
        self.y = 0
```

```
p1 = Point()

print(p1)
p1.reset()
```

```
<__main__.Point object at 0x00000206DEBB08D0>
<__main__.Point object at 0x00000206DEBB08D0>
```

- * اسم **self** اختیاری است و می توانیم هر اسمی بگذاریم اما بهتر است از **self** استفاده کنیم.

- * به روش زیر نیز می توانیم متده را صدا بزنیم.

```
class Point:
    def reset(self):
        self.x = 0
        self.y = 0
```

```
p1 = Point()
p2 = Point()
p2.x = 4
p2.y = 5
print('x: ', p2.x)
print('y: ', p2.y)
Point.reset(p2)
print('-' * 5, 'after reset', '-' * 5)
print('x: ', p2.x)
print('y: ', p2.y)
```

```
x: 4
y: 5
----- after reset -----
x: 0
y: 0
```

- * هر تابعی که داخل کلاس بنویسیم باید **self** را به عنوان ورودی به آن بدهیم.

```
class Point:
    def reset():
        x = 0
        y = 0
```

```
p2 = Point()
p2.x = 4
Point.reset(p2)
```

```
Traceback (most recent call last):
  File "F:\python\pythonProject\test.py", line 9, in <module>
    Point.reset(p2)
TypeError: Point.reset() takes 0 positional arguments but 1 was given
```

* بهتر است برای متدها از **یادداشت (annotations)** استفاده کنیم.

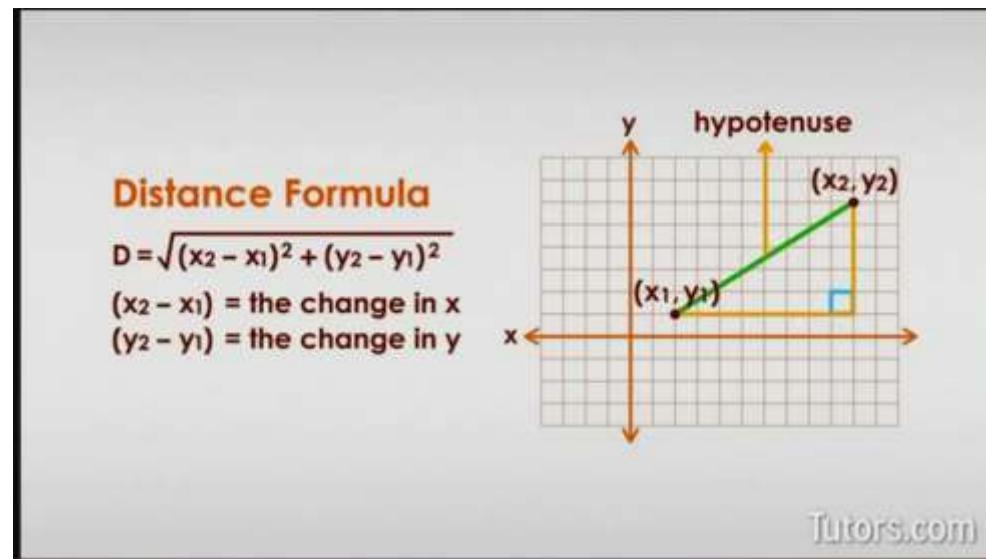
```
class Point:  
    def move(self, x: float, y: float) -> None:  
        self.x = x  
        self.y = y  
  
    def reset(self):  
        self.move(0, 0)  
  
p1 = Point()  
p2 = Point()  
p2.x = 4  
p2.y = 5  
print('x: ', p2.x)  
print('y: ', p2.y)  
p2.reset()  
print('--- * 5, 'after reset', '--- * 5)  
print('x: ', p2.x)  
print('y: ', p2.y)  
p2.move(8, 9)  
print('--- * 5, 'after move', '--- * 5)  
print('x: ', p2.x)  
print('y: ', p2.y)
```

```
x: 4  
y: 5  
---- after reset ----  
x: 0  
y: 0  
---- after move ----  
x: 8  
y: 9
```

* با متدهای **math.hypot** از مازول **math** می توان وتر یک مثلث را به دست آورد.

```
from math import hypot  
  
print(hypot(4, 3)) # -> 5.0
```

* برای به دست آوردن فاصله‌ی دو نقطه نیز می‌توان از **hypot** استفاده کرد.



```
from math import hypot

class Point:
    def move(self, x: float, y: float) -> None:
        self.x = x
        self.y = y

    def reset(self):
        self.move(0, 0)

    def distance(self, other: 'Point'):
        return hypot((self.x - other.x), (self.y - other.y))

p1 = Point()
p2 = Point()

p1.reset()
print('x:', p1.x, 'y:', p1.y)
p2.move(3, 4)
print('x:', p2.x, 'y:', p2.y)
print('distance:', p1.distance(p2))
```

```
x: 0 y: 0
x: 3 y: 4
distance: 5.0
```

درس ۱۰: مقدار دهی اولیه شی

- * برای مقداردهی اولیه در پایتون از `__new__` و `__init__` با هم استفاده می‌کنیم.
- * `__new__` وظیفه‌ی ایجاد کردن شی را برعهده دارد، همه‌ی اشیا این دو متده را به صورت پیش فرض دارند.
- * `__init__` یک پارامترش همان شی است که قرار است مقداردهی اولیه کند و پارامترهای دیگر مقادیری هستند که صفات را مقداردهی می‌کنیم.
- * می‌توانیم کاربر را مجبور کنیم، که خودش مقدار اولیه را وارد کند.

```
class Point:  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y
```

```
p1 = Point(4, 3)  
print(p1.x) # -> 4  
print(p1.y) # -> 3
```

- * می‌توانیم مقدار پیش‌فرض تعیین کنیم که کاربر نیاز نباشد مقدار اولیه بدهد.

```
class Point:  
    def __init__(self, x: float = 0, y: float = 0) -> None:  
        self.x = x  
        self.y = y
```

```
p1 = Point(4, 3)  
p2 = Point()  
print(p1.x) # -> 4  
print(p1.y) # -> 3  
print(p2.x) # -> 0  
print(p2.y) # -> 0
```

* مقدار پیشفرض را با استفاده از متدهای دیگر نیز می‌توانیم تعیین کنیم.

```
class Point:

    def __init__(self, x: float = 0, y: float = 0) -> None:
        self.move(x, y)

    def move(self, x: float, y: float) -> None:
        self.x = x
        self.y = y

p1 = Point(4, 3)
p2 = Point()
print(p1.x) # -> 4
print(p1.y) # -> 3
print(p2.x) # -> 0
print(p2.y) # -> 0
```

نمی‌تواند خروجی داشته باشد.

```
class Point:

    def __init__(self, x: float = 0, y: float = 0) -> None:
        self.move(x, y)
        return 5

    def move(self, x: float, y: float) -> None:
        self.x = x
        self.y = y

p1 = Point(4, 3)
```

```
Traceback (most recent call last):
  File "F:\python\pythonProject\test.py", line 12, in <module>
    p1 = Point(4, 3)
           ^
TypeError: __init__() should return None, not 'int'
```

درس ۱۱: مستند سازی کلاس و ابزار mypy و doctest

- * در داک استرینگ کلاس به طور مختصر و کوتاه هدف کلاس نوشته می شود و مشخص می شود که وظیفه‌ی هر متده است.
- * در داک استرینگ معمولاً مثال هم نوشته می شود، بهتر است مثال را همان‌گونه که در ترمینال می‌نویسیم، بنویسیم.

```
from math import hypot

class Point:
    """
    Represents a point on a two-dimensional coordinate axis.
    # >>> p1 = point()
    # >>> p2 = point(3, 4)
    # >>> p1.reset()
    # >>> p1.distance(p2)
    5.0
    """

    def __init__(self, x: float = 0, y: float = 0) -> None:
        self.move(x, y)

    def move(self, x: float, y: float) -> None:
        self.x = x
        self.y = y

    def reset(self):
        self.move(0, 0)

    def distance(self, other: 'Point'):
        return hypot((self.x - other.x), (self.y - other.y))
```

- * برای متدهای کلاس نیز بهتر است داک استرینگ نوشته شود.

```
class Point:

    def __init__(self, x: float = 0, y: float = 0) -> None:
        """
        Initializes the x and y coordinates of the new point.
        :param x: x-coordinates
        :param y: y-coordinates
        """
        self.move(x, y)
```

* برای مشاهده داک استرینگ می‌توانیم از `__doc__` استفاده کنیم.

```
class Point:  
    """  
    Represents a point on a two-dimensional coordinate axis.  
    # >>> p1 = point()  
    # >>> p2 = point(3, 4)  
    # >>> p1.reset()  
    # >>> p1.distance(p2)  
    5.0  
    """  
  
print(Point.__doc__)
```

```
Represents a point on a two-dimensional coordinate axis.  
# >>> p1 = point()  
# >>> p2 = point(3, 4)  
# >>> p1.reset()  
# >>> p1.distance(p2)  
5.0
```

* برای مشاهدهی داک استرینگ بهتر است از `help` استفاده کنیم، زیرا `help` علاوه بر داکیومنت کلاس، داک استرینگ متدهای آن را نیز نمایش می‌دهد.

```
class Point:  
    """  
    Represents a point on a two-dimensional coordinate axis.  
    # >>> p1 = point()  
    # >>> p2 = point(3, 4)  
    # >>> p1.reset()  
    # >>> p1.distance(p2)  
    5.0  
    """  
  
    def __init__(self, x: float = 0, y: float = 0) -> None:  
        """  
        Initializes the x and y coordinates of the new point.  
        :param x: x-coordinates  
        :param y: y-coordinates  
        """  
        self.move(x, y)  
  
print(help(Point))
```

```
Help on class Point in module __main__:  
  
class Point(builtins.object)  
| Point(x: float = 0, y: float = 0) -> None  
|  
| Represents a point on a two-dimensional coordinate axis.  
| # >>> p1 = point()  
| # >>> p2 = point(3, 4)  
| # >>> p1.reset()  
| # >>> p1.distance(p2)  
| 5.0  
|  
| Methods defined here:  
|  
|     __init__(self, x: float = 0, y: float = 0) -> None  
|         Initializes the x and y coordinates of the new point.  
|         :param x: x-coordinates  
|         :param y: y-coordinates  
|  
|-----  
| Data descriptors defined here:  
|  
|     __dict__  
|         dictionary for instance variables (if defined)  
|  
|     __weakref__  
|         list of weak references to the object (if defined)  
  
None
```

* برای تست مثالی که در داک استرینگ استفاده می‌کنیم، می‌توانیم از ماژول **testmod** و متدهای **doctest** استفاده کنیم، اگر مثال درست باشد چیزی نشان نمی‌دهد، اما اگر مثال اشتباه باشد، اشتباه و پاسخ صحیح را نشان می‌دهد.

```
import doctest

class Point:
    """
    Represents a point on a two-dimensional coordinate axis.
    # >>> p1 = point()
    # >>> p2 = point(3, 4)
    # >>> p1.reset()
    # >>> p1.distance(p2)
    5.0
    """
doctest.testmod()
```

```
exec(compile(example.source, filename, "s"))
File "<doctest __main__.Point[3]>", line 1,
      p1.distance(p2)
      ^
NameError: name 'p1' is not defined
*****
1 items had failures:
    4 of    4 in __main__.Point
***Test Failed*** 4 failures.
```

* برای بررسی یادداشت‌ها بهتر است ماژول **mypy** را نصب کنیم و در ترمینال اجرا کنیم.

```
Terminal Local + ∨
(venv) PS F:\python\pythonProject> mypy test.py
Success: no issues found in 1 source file
```

* اگر مطابق نوعی که در یادداشت مشخص کردیم، ورودی را ارسال نکنیم، **mypy** تشخیص می‌دهد.

```
from math import hypot

class Point:

    def __init__(self, x: float = 0, y: float = 0) -> None:
        self.move(x, y)

    def move(self, x: float, y: float) -> None:
        self.x = x
        self.y = y

    def reset(self):
        self.move(0, 0)

    def distance(self, other: 'Point'):
        return hypot((self.x - other.x), (self.y - other.y))

p1 = Point(4, '3')
```

```
Terminal Local + ∨
(venv) PS F:\python\pythonProject> mypy test.py
test.py:36: error: Argument 2 to "Point" has incompatible type "str"; expected "float"  [arg-type]
Found 1 error in 1 file (checked 1 source file)
```

درس ۱۲: ماثول‌ها و کلاس‌ها (بخش اول)

- * بهتر است دستورات ماثول را با ***** وارد نکنیم، به دلیل ریشه یابی آن دستور و تداخل اسم با دستورات دیگر.
- * اگر یک ماثول در یک پکیج تودرتو باشد و بخواهیم دستورات آن را وارد کنیم، نوشتن مسیر آن ماثول ممکن است طولانی شود.

A screenshot of a PyCharm interface. On the left, the project tree shows a 'pythonProject' folder containing '.myapp_cache', 'parent' (which has 'child1' and 'child2' subfolders), 'venv', and 'wa_re_time'. In the center, there are two tabs: 'test.py' and 'm1.py'. The 'm1.py' tab is active, displaying the following code:

```
MONTH = 12
class Animal:
    pass
```

```
from parent.child1.m1 import MONTH, Animal
print(MONTH) # -> 12
print(Animal) # -> <class 'parent.child1.m1.Animal'>
```

- * می‌توانیم این ماثول را داخل فایل **init** پکیج پدر وارد کنیم و با وارد کردن این پکیج، دستوراتی که در **init** نوشته‌ایم نیز به صورت خودکار وارد می‌شوند.

A screenshot of a PyCharm interface. The 'test.py' tab is active. The code in '_init_.py' is:

```
from .child1.m1 import MONTH, Animal
```

```
from parent import MONTH, Animal
print(MONTH) # -> 12
print(Animal) # -> <class 'parent.child1.m1.Animal'>
```

درس ۱۳: مازول‌ها و کلاس‌ها (بخش دوم)

* بهتر است کلاس‌ها را با توجه به موضوعات در مازول‌های جدا از هم قرار دهیم.

* برای بعضی از کلاس‌ها **فقط یک شی** نیاز داریم، که بهتر است آن را داخل مازول آن کلاس بسازیم و اگر این شی را که ساختیم به دو متغیر اختصاص بدھیم در واقع هر دو متغیر یکی هستند.

```
test.py point.py x
1 from math import hypot
2
3
4 > class Point:...
46
47
48 point = Point()
49
```

```
from point import point
p1 = point
p2 = point
print(p1)
print(p2)
```

```
<point.Point object at 0x00000227E9B80910>
<point.Point object at 0x00000227E9B80910>
```

* گاهی ممکن است ساخت شی مدتی طول بکشد، برای حل این مشکل می‌توانیم آن شی را در مازول، داخل تابع بگذاریم تا با وارد کردن مازول ساخته نشود و هر وقت خواستیم آن را بسازیم.

```
point = None
```

```
def get_point():
    global point
    if not point:
        point = Point()
    return point
```

```
from point import get_point
p1 = get_point()
```

* اگر برای شی که ایجاد کردیم یادداشت بنویسیم که یک شی از کلاس است، **mypy** خطا می‌دهد چون مقدارش **None** می‌باشد و اگر نوعش را **None** بنویسیم، در هنگام ایجاد شی خطا می‌دهد چون در آن حالت یک شی از کلاس است.

```
point: Point = None
```

```
Terminal Local x
(venv) PS F:\python\pythonProject> mypy test.py
point.py:48: error: Incompatible types in assignment (expression has type "None", variable has type "Point") [assignment]
Found 1 error in 1 file (checked 1 source file)
```

* برای حل این مشکل می‌توانیم از **typing** وارد می‌کنیم، در این حالت برای نوشتگر نو، ابتدا **Optional** را می‌نویسیم و سپس داخل براکت نوع را می‌نویسیم.

```
test.py
point.py x

1 from math import hypot
2 from typing import Optional
3
4
5 > class Point:...
47
48
49 point: Optional[Point] = None
```

```
Terminal Local + v
(venv) PS F:\python\pythonProject> mypy test.py
Success: no issues found in 1 source file
```

* با متدهای **__name__** می‌توانیم اسم ماژول را مشخص کنیم، اگر اسم فایلی که در آن هستیم را بخواهیم نشان دهد، نتیجه **__main__** می‌باشد و اگر ماژول دیگری که وارد کردیم را بخواهیم نشان دهد، نتیجه اسم آن ماژول است.

```
import point

print(__name__) # -> __main__
print(point.__name__) # -> point
```

* ممکن است در ماژول دستوراتی باشند که فقط وقتی می‌خواهیم ماژول را اجرا کنیم، اجرا شوند، اما وقتی ماژول را در یک فایل دیگر وارد کنیم این دستورات نیز اجرا می‌شوند.

```
test.py
point.py x

5 > class Point:...
47
48
49 po1 = Point()
50 po1.move(3, 6)
51 print(po1.x, po1.y)
```

```
Kun test x
test.py x point.py
1 import point
2
3 6
```

* برای حل این مشکل می‌توانیم از شرط‌ها استفاده کنیم، که اگر اسم ماژول **__main__** باشد دستورات را اجرا کند، زمانی که ماژول را **import** کنیم، دستوراتش اجرا نمی‌شوند.

```
test.py
point.py x

59 def main():
60     po1 = Point()
61     po1.move(3, 6)
62     print(po1.x, po1.y)
63
64
65 > if __name__ == '__main__':
66     main()
```

```
Run test x
test.py x point.py
1 import point
2
```

درس ۱۴: سطح دسترسی و کاربردهای underscore (بخش اول)

- * در پایتون، داده‌ها در داخل کلاس، **خصوصی** (**Private**) و یا **محفظت شده** (**protected**) نیستند و همیشه یک راه برای دسترسی به آن‌ها پیدا می‌شود.
- * اگر می‌خواهیم داده‌ها در دسترس نباشند، باید در خارج از کلاس از آن‌ها استفاده نکنیم.

درس ۱۵: سطح دسترسی و کاربردهای underscore (بخش دوم)

- * اگر از متغیر استفاده ای نمی‌شود بهتر است از **اندراسکور** (**_**) استفاده کنیم. (می‌توانیم از مقداری که به اندراسکور دادیم استفاده کنیم، اما طبق قرار داد اگر اسم متغیری اندراسکور باشد، به این معنی است که آن متغیر هیچ استفاده ای ندارد.)

```
for _ in range(10):
    print('hi')
name, _ = ('ali', 12)
```

- * در ترمینال و هنگام استفاده از حالت تعاملی، آخرین مقدار در **اندراسکور** ذخیره می‌شود.

```
Terminal Local × +
>>> 'hi'
'hi'
>>> 5
5
>>> x = _
>>> x
5
```

- * اگر بخواهیم مشخص کنیم که متغیر یا متند **محفظت شده** (**protected**) است از اسم متغیر و یا متند استفاده می‌کنیم. (متغیر کاملاً در دسترس است اما طبق قرارداد، این متغیر فقط در داخل کلاس و کلاس‌هایی که از آن ارث بری کرده‌اند، در دسترس باشد.)

```
test.py × point.py
1 class User:
2     def __init__(self, name='', phone=''):
3         self._user_id = id(self)
4         self.name = name
5         self.phone = phone
6
7
8 ali = User('ali', '0912')
9 print('id:', ali._user_id)
10 print('name:', ali.name)
11 print('phone number:', ali.phone)

Run test ×
F:\python\pythonProject\venv\Scripts\python.exe F:/python/pythonProject/test.py
id: 2573863946256
name: ali
phone number: 0912
```

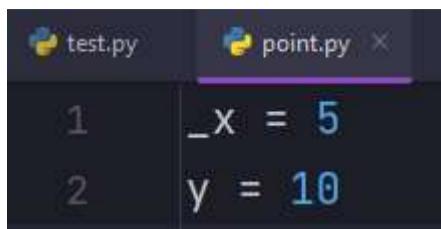
Access to a protected member _user_id of a class

Add property for the field Alt+Shift+Enter More actions... Alt+Enter

Instance attribute _user_id of test.User

_user_id: int = id(self)

* اگر اسم متغیر داخل یک ماژول با **اندراسکور** شروع شود، هنگامی که با ***** همهی متغیرهای آن ماژول را وارد می‌کنیم، آن متغیر در دسترس نمی‌باشد.



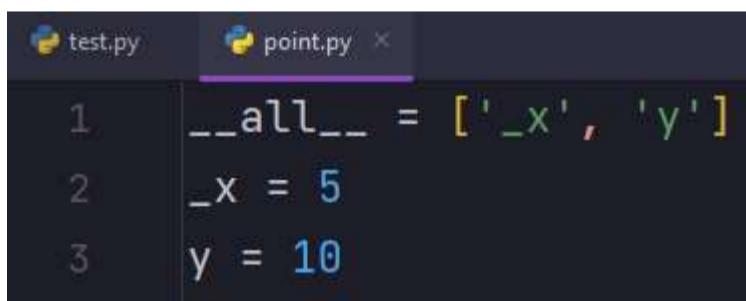
```
test.py point.py
1 _x = 5
2 y = 10
```

```
from point import *

print(y)
print(_x)
```

```
10
Traceback (most recent call last):
  File "F:\python\pythonProject\test.py", line 4, in <module>
    print(_x)
           ^
NameError: name '_x' is not defined
```

* برای حل این مشکل می‌توان داخل ماژول، اسم متغیر را داخل **__all__** قرار داد.



```
test.py point.py
1 __all__ = ['_x', 'y']
2 _x = 5
3 y = 10
```

```
from point import *

print(y) # -> 10
print(_x) # -> 5
```

* در پایتون نمی‌توانیم از کلمات کلیدی برای اسم متغیر استفاده کنیم، اگر بخواهیم از این اسمی برای متغیر استفاده کنیم باید آن را تغییر دهیم، برای این کار بهتر است بعد از اسم **اندراسکور** بگذاریم.

```
min_ = 0
print(min_) # -> 0
print(min) # -> <built-in function min>
```

درس ۱۶: سطح دسترسی و کاربردهای underscore (بخش سوم)

- * اگر بخواهیم مشخص کنیم که متغیر یا متند خصوصی (private) است از دو اندراسکور (__) قبل از اسم متغیر و یا متند استفاده می‌کنیم. (متغیر کاملا در دسترس است اما طبق قرارداد، این متغیر فقط باید در داخل کلاس، در دسترس باشد.)
- * متغیر خصوصی (private) در داخل کلاس با همین اسم در دسترس است.

```
class User:  
    def __init__(self, name='', phone=''):   
        self.__user_id = id(self)  
        self.name = name  
        self.phone = phone  
        self.x = '+' + str(self.__user_id)
```

```
ali = User('ali', '0912')  
print(ali.x) # -> +2049374179792
```

- * متغیر خصوصی (private) در خارج از کلاس با این اسم در دسترس نیست.

```
class User:  
    def __init__(self, name='', phone=''):   
        self.__user_id = id(self)  
        self.name = name  
        self.phone = phone  
        self.x = '+' + str(self.__user_id)
```

```
ali = User('ali', '0912')  
print('id:', ali.__user_id)
```

```
Traceback (most recent call last):  
  File "F:\python\pythonProject\test.py", line 10, in <module>  
    print('id:', ali.__user_id)  
    ^^^^^^^^^^^^^^  
AttributeError: 'User' object has no attribute '__user_id'
```

- * علت این است که هنگامی که از داندر اندراسکور (__) استفاده می‌کنیم، پایتون اسم ایتن متغیر را در خارج از کلاس تغییر می‌دهد تا از آن استفاده نکنیم، به این صورت که قبل از اسم متغیر از اندراسکور و اسم کلاس استفاده می‌کند.

The screenshot shows the PyCharm IDE interface. On the left, there are two tabs: 'test.py' and 'point.py'. The 'test.py' tab contains the following Python code:

```
1 class User:  
2     def __init__(self, name='', phone=''):   
3         self.__user_id = id(self)  
4         self.name = name  
5         self.phone = phone  
6         self.x = '+' + str(self.__user_id)  
7  
8  
9 ali = User('ali', '0912')  
10 print('id:', ali.__user_id)
```

On the right, the 'Run' tool window is open, showing the output of the run command:

```
Run test  
F:\python\pythonProject  
id: 1741153174352
```

* علت دیگر تغییر اسم متغیر خصوصی (**private**) در پایتون این است که اگر در یک کلاس ارث بری شده از اسم آن متغیر استفاده کنیم، دچار تداخل نشویم.

```
class User:  
    def __init__(self, name='', phone=''):   
        self.__user_id = id(self)  
        self.name = name  
        self.phone = phone  
  
class Account(User):  
    def __init__(self):  
        super().__init__()  
        self.__user_id = '1234'  
  
ali = User('ali', '0912')  
ali_account = Account()  
print(dir(ali_account)) # -> ['_Account__user_id', '_User__user_id', ... ]
```

* اگر قبل و بعد از اسم متدهای خاص (**special**) استفاده شود، آن متدهای خاص (**special**) پایتون میباشد و بهتر است ما از این حالت برای نامگذاری متغیرها استفاده نکنیم، زیرا این حالت برای شناسایی متدهای خاص پایتون میباشد.

* برای کنترل مقدار یک خصوصیت میتوانیم از **getter** و **setter** استفاده کنیم، اسم این دو دلخواه است اما معمولاً از **get** و **set** استفاده میکنیم، از **getter** برای تغییر و از **setter** برای برگرداندن استفاده میکنیم.

```
class User:  
    def __init__(self, name='', phone=''):   
        self.user_id = id(self)  
        self.name = name  
        self.__phone = phone  
  
    def set_phone(self, phone):  
        if len(phone) == 11 and phone.isdigit():  
            self.__phone = phone  
  
    def get_phone(self):  
        return self.__phone  
  
ali = User()  
ali.set_phone('01234567891')  
number = ali.get_phone()  
print(number) # -> 01234567891
```

درس ۱۷: متدهای `str` و `repr`

* متدهای `str` و `repr` رشته را همان‌گونه که هست نشان می‌دهد.

```
print(str('ali')) # -> ali
print(repr('ali')) # -> 'ali'
```

* متدهای `str` و `repr` را صدا می‌زنند و متدهای `__str__` و `__repr__` را صدا می‌زنند.

* متدهای `str` و `repr` بیشتر برای دیباگ کردن و توسعه استفاده می‌شود و بیشتر به درد برنامه نویس می‌خورد، اما `str` بیشتر برای کاربر است.

```
import datetime

today = datetime.datetime.now()
print(str(today))
print(repr(today))
```

```
2023-12-20 22:26:57.427589
datetime.datetime(2023, 12, 20, 22, 26, 57, 427589)
```

* چیزی که متدهای `str` و `repr` معمولاً ارائه می‌دهد، شکلی است که دوباره می‌توانیم شی را بسازیم.

```
import datetime

print(datetime.datetime(2023, 12, 20, 22, 26, 57, 427589))
```

```
2023-12-20 22:26:57.427589
```

* پیشفرضی که برای `print` وجود دارد، `str` است.

```
print('reza') # -> reza
```

* پیشفرضی که برای `print` در ترمینال وجود دارد، `repr` است.

```
>>> 'reza'
'reza'
```

* وقتی برای یک شی از کلاس `str` را صدا می‌زنیم، پیشفرض `str` کلاس `__repr__` است.

```
class Person:
    def __init__(self, first_name, last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age
```

```
person = Person('ali', 'ahmadi', 24)
print(str(person))
print(repr(person))
```

```
<__main__.Person object at 0x000001B573F75090>
<__main__.Person object at 0x000001B573F75090>
```

* می‌توانیم داخل کلاس `__str__` را تعریف کنیم، آن‌گاه اگر از `str` استفاده کنیم، چیزی را که ما تعریف کردیم برمی‌گرداند.

```
class Person:  
    def __init__(self, first_name, last_name, age):  
        self.first_name = first_name  
        self.last_name = last_name  
        self.age = age  
  
    def __str__(self):  
        return f'{self.first_name} {self.last_name}'
```

```
person = Person('ali', 'ahmadi', 24)  
print(str(person))
```

ali ahmadi

* می‌توانیم داخل کلاس `__repr__` را تعریف کنیم، آن‌گاه اگر از `repr` استفاده کنیم، چیزی را که ما تعریف کردیم برمی‌گرداند، بهتر است `repr` را به گونه‌ای تعریف کنیم که نتیجه‌اش را اگر اجرا کنیم، بتوانیم یک شی جدید بسازیم.

```
class Person:  
    def __init__(self, first_name, last_name, age):  
        self.first_name = first_name  
        self.last_name = last_name  
        self.age = age  
  
    def __str__(self):  
        return f'{self.first_name} {self.last_name}'  
  
    def __repr__(self):  
        return f"{self.__class__.__name__}" \  
               f"('{self.first_name}', '{self.last_name}'" \  
               f", {self.age})"
```

```
person = Person('ali', 'ahmadi', 24)  
print(repr(person))
```

Person('ali', 'ahmadi', 24)

درس ۱۸: حل مثال: کلاس حساب بانکی

* به جای `pass` می‌توانیم از ... استفاده کنیم.

```
class BankAccount:  
    ...
```

* اگر یک متغیر تعریف کنیم و از `self` برای مقداردهی استفاده نکنیم، آن متغیر بین همهٔ شیوه‌ایی که از آن کلاس ساخته می‌شود، مشترک است.

```
class BankAccount:  
    account_numbers = set()
```

مثال: کلاس بانکی

```
from random import randrange  
from typing import Set  
  
class BankAccount:  
    """  
    Bank account management.  
    """  
    all_account_numbers: Set[int] = set()  
  
    def __init__(self, first_name: str, last_name: str) -> None:  
        self.account_number: int = 0  
        while True:  
            if (an := randrange(10000, 100000)) not in BankAccount.all_account_numbers:  
                BankAccount.all_account_numbers.add(an)  
                self.account_number = an  
                break  
        self.first_name = first_name  
        self.last_name = last_name  
        self.balance: float = 0  
  
    def display(self) -> None:  
        """  
        Show account balance.  
        :return:  
        """  
        print(40 * '-')  
        print(f"Hi, {self.first_name}. \nYour current balance is: {self.balance} ")  
        print(40 * '-')  
  
    def deposit(self) -> None:  
        """  
        Increase account balance.  
        :return:  
        """  
        amount = float(input('Please enter amount to deposit: '))  
        self.balance += amount
```

```
self.display()

def withdraw(self) -> None:
    """
    withdraw from bank account.
    :return:
    """

    amount = float(input('Please enter amount to withdraw: '))
    if amount > self.balance:
        print('Insufficient balance!')
    else:
        self.balance -= amount
    self.display()

def main():
    account1 = BankAccount('Ali', 'Ahmadi')
    print(40 * '*')
    print(f'account number: {account1.account_number}')
    print(40 * '*')
    account1.display()

    while True:
        choice = int(input('Enter\n1 to see your balance,\n2 to deposit,\n'
                           '3 to withdraw,\n4 to exit.\n\t\t your choice: '))
        if choice == 1:
            account1.display()
        elif choice == 2:
            account1.deposit()
        elif choice == 3:
            account1.withdraw()
        elif choice == 4:
            break
        else:
            print('please enter a valid number!')

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

* می توانیم شماره حساب را بهینه تر بنویسیم.

```
class BankAccount:  
    """  
    Bank account management.  
    """  
  
    all_account_numbers: list[int] = []  
    last_account_number = 999  
  
    def __init__(self, first_name: str, last_name: str) -> None:  
        BankAccount.last_account_number += 1  
        an = BankAccount.last_account_number  
        self.account_number: int = an  
        BankAccount.all_account_numbers.append(an)  
        self.first_name = first_name  
        self.last_name = last_name  
        self.balance: float = 0
```

درس ۱۹: انواع attribute در کلاس

* دو نوع اtribut کلاس، اtribut نمونه (class attribute) و اtribut کلاس (instance attribute) هستند.

* به اtribut‌هایی که مختص یک شی هستند و ممکن است در اشیا دیگر وجود نداشته باشند و اگر هم باشند دارای مقدار متفاوتی باشند، اtribut نمونه (instance attribute) گفته می‌شود. در مثال کلاس حساب بانکی اtribut‌هایی که با Self مقداردهی شده اند، اtribut نمونه هستند. (از مثال کلاس حساب بانکی استفاده شده است)

```
a = BankAccount('Ali', 'Rezai')
a.x = 5
b = BankAccount('Mohammad', 'Hassani')
print(a.first_name)
print(a.x)
print(b.first_name)
print(b.x)
```

```
Ali
5
Mohammad
Traceback (most recent call last):
  File "F:\python\pythonProject\test.py", line 85, in <module>
    print(b.x)
      ^
AttributeError: 'BankAccount' object has no attribute 'x'
```

* به اtribut‌هایی که بین کلاس و همه اشیا مشترک اند و با self مقداردهی نشده اند، اtribut کلاس (class attribute) گفته می‌شود. در مثال کلاس حساب بانکی اtribut‌هایی که با Self مقداردهی نشده اند، اtribut کلاس هستند.

```
a = BankAccount('Ali', 'Rezai')
b = BankAccount('Mohammad', 'Hassani')
print(BankAccount.all_account_numbers) # -> [1000, 1001]
print(a.all_account_numbers) # -> [1000, 1001]
print(b.all_account_numbers) # -> [1000, 1001]
```

* اگر از یک پایتون نسخه قدیمی استفاده می‌کنیم و بخواهیم از ویژگی‌های پایتون نسخه جدید استفاده کنیم باید آن ویژگی را از ماژول future وارد کنیم، ماژول future باید در خط اول وارد شود.

مثال: در پایتون نسخه ۳.۸ نمی‌توانیم از یادداشت استفاده کنیم و حتما باید آن را از typing وارد کنیم، که با وارد کردن این مشکل برطرف خواهد شد.

```
from __future__ import annotations

class User:
    all_users: list['User'] = []
```

* اگر در هنگام فرمت دهی رشته بعد از داده از `r!` استفاده کنیم، آن را به صورت `repr` نمایش می‌دهد.

```
class User:  
    all_users: list['User'] = []  
  
    def __init__(self, user_name: str, email: str, password: str) -> None:  
        self.user_name = user_name  
        self.email = email  
        self.password = password  
        User.all_users.append(self)  
  
    def __repr__(self):  
        return f"{self.__class__.__name__}({self.user_name!r}, " \\\n            f" {self.email!r}, {self.password!r})"  
  
user1 = User('rezadolati01', 'reza@gmail.com', '12345')  
print(repr(user1))
```

```
User('rezadolati01', 'reza@gmail.com', '12345')
```

* می‌توانیم از `pprint` به جای `print` استفاده کنیم، `pprint` نتیجه را به صورت مرتب نمایش می‌دهد، برای استفاده از `pprint` باید آن را از مژول `pprint` وارد کنیم.

```
from pprint import pprint  
  
class User: ...  
  
user1 = User('rezadolati01', 'reza@gmail.com', '12345')  
user2 = User('aliadsf3', 'ali@gmail.com', '9876')  
user3 = User('ahmad3eref', 'ahmad@gmail.com', '11111')  
pprint(User.all_users)
```

```
[User('rezadolati01', 'reza@gmail.com', '12345'),  
 User('aliadsf3', 'ali@gmail.com', '9876'),  
 User('ahmad3eref', 'ahmad@gmail.com', '11111')]
```

درس ۲۰: تکالیف مبحث ایجاد کلاس

۱. یک کلاس پایتون به نام `Rectangle` ایجاد کنید تا مستطیل‌ها را نشان دهد. کلاس باید دارای اtribut‌های عرض و ارتفاع و متدهایی برای محاسبه مساحت و محیط مستطیل باشد.
۲. یک کلاس پایتون به نام `Car` برای نمایش اشیاء ماشین ایجاد کنید. کلاس باید دارای اtribut‌های برنده، مدل و سال و متدهایی برای دریافت اطلاعات خودرو باشد.
۳. یک کلاس پایتون به نام `Student` ایجاد کنید تا اشیاء دانشآموز را نشان دهد. کلاس باید دارای اtribut‌های نام، سن و نمرات و متدهایی برای اضافه کردن نمرات و محاسبه میانگین نمره باشد.
۴. یک کلاس پایتون به نام `Book` برای نمایش کتاب‌ها ایجاد کنید. کلاس باید دارای اtribut‌های عنوان، نویسنده و سال و متدهایی برای نمایش اطلاعات کتاب باشد.
۵. یک کلاس پایتون به نام `Employee` ایجاد کنید تا کارمندان را نمایندگی کند. کلاس باید دارای اtribut‌های نام، حقوق و متدهایی برای افزایش حقوق و نمایش اطلاعات کارکنان باشد.
۶. یک کلاس به نام `Email` برای نمایش پیام‌های ایمیل ایجاد کنید. کلاس باید دارای اtribut‌های فرستنده، گیرنده‌گان، موضوع و بدهه و متدهایی برای ارسال ایمیل باشد.

درس ۲۱: وراثت یگانه

- * ارث بری باعث می‌شود که از تکرار کدها جلوگیری کنیم.
- * کلاس‌ها می‌توانند از یک کلاس کلی‌تر ارث ببرند و این کلاس‌ها همه‌ی ویژگی‌ها و اتریبوت‌های کلاس کلی‌تر را دارند و می‌توانند آن‌ها را توسعه دهند.

* همه‌ی کلاس‌ها از کلاس **object** ارث بری می‌کنند.

* برای این‌که مشخص کنیم یک کلاس از چه کلاسی ارث بری کرده است می‌توانیم از متدهای **base** استفاده کنیم.

```
class User:  
    ...  
  
print(User.__base__) # -> <class 'object'>
```

* برای ارث بری یک کلاس، کافی است جلوی اسم کلاس پرانتز باز کنیم و اسم کلاسی که می‌خواهیم از آن ارث بری کند را بنویسیم.

```
class User:  
    all_users: list['User'] = []  
  
    def __init__(self, user_name: str, email: str, password: str) -> None:  
        self.user_name = user_name  
        self.email = email  
        self.password = password  
        User.all_users.append(self)  
  
    def __repr__(self):  
        return f'{self.__class__.__name__}({self.user_name!r}, '\br/>               f'{self.email!r}, {self.password!r})'  
  
    def __str__(self):  
        return f'{self.user_name}'
```

```
class Seller(User):  
    pass
```

```
print(Seller.__base__) # -> <class '__main__.User'>
```

* در این مثال کلاس **Seller** از کلاس **User** مشتق شده است (ارث بری کرده است و یا در حال توسعه کلاس **User** می‌باشد).

* کلاس ارث بری شده دارای همهی متدها و اتریبیوت‌های کلاس پدر خودش می‌باشد.

```
class User:  
    all_users: list['User'] = []  
  
    def __init__(self, user_name: str, email: str, password: str) -> None:  
        self.user_name = user_name  
        self.email = email  
        self.password = password  
        User.all_users.append(self)  
  
    def __repr__(self):  
        return f'{self.__class__.__name__}({self.user_name!r}, '\  
               f'{self.email!r}, {self.password!r})'  
  
    def __str__(self):  
        return f'{self.user_name}'  
  
class Seller(User):  
    pass  
  
s = Seller('Ali34', 'ali@gmail.com', '1234')  
print(s.user_name) # -> Ali34
```

* می‌توانیم اتریبیوت‌ها و متدهای دیگر برای کلاس ارث بری شده بنویسیم.

```
class User: ...  
  
class Seller(User):  
    def order(self, order: 'Order') -> None:  
        print(f'{self.user_name}, From your products, {order!r} was sold!')  
  
s = Seller('Ali34', 'ali@gmail.com', '1234')  
s.order('bag')
```

```
Ali34, From your products, 'bag' was sold!
```

درس ۲۲: ارث بری از کلاس‌های داخلی

* می‌توانیم کلاس‌هایی بنویسیم که از کلاس‌های `built_in` پایتون ارث بری کنند.

```
class UserList(list):  
    pass
```

```
li = UserList()  
x = list()  
print(x) # -> []  
print(li) # -> []  
li.append(3)  
print(li) # -> [3]
```

* می‌توانیم به یک کلاس `built_in` پایتون متدهایی که می‌خواهیم را اضافه کنیم.

```
from pprint import pprint
```

```
class UserList(list):  
    def search(self, user_name: str) -> list['User']:  
        matching_users: list['User'] = []  
        for user in self:  
            if user_name in user.user_name:  
                matching_users.append(user)  
        return matching_users
```

```
class User:  
    all_users: list['User'] = UserList()  
  
    def __init__(self, user_name: str, email: str, password: str) -> None:  
        self.user_name = user_name  
        self.email = email  
        self.password = password  
        User.all_users.append(self)  
  
    def __repr__(self):  
        return f'{self.__class__.__name__}({self.user_name!r}, '\  
               f'{self.email!r}, {self.password!r})'  
  
    def __str__(self):  
        return f'{self.user_name}'  
  
class Seller(User):  
    def order(self, order: 'Order') -> None:  
        print(f'{self.user_name}, From your products, {order!r} was sold!')
```

```
user1 = User('reza', 'e', '1')
```



```
user2 = User('ali', 'a', '4')
user3 = User('ali01', 'b', '6')
user4 = User('rezadl09', 'u', '5')
print('all users:')
pprint(User.all_users)
print(40 * '-', '\n')
print('reza users:')
pprint(User.all_users.search('reza'))
print(40 * '-', '\n')
print('ali users:')
pprint(User.all_users.search('ali'))
print(40 * '-', '\n')
print('mohammad users:')
pprint(User.all_users.search('mohammad'))
```

```
all users:
[User('reza', 'e', '1'),
 User('ali', 'a', '4'),
 User('ali01', 'b', '6'),
 User('rezadl09', 'u', '5')]
-----
reza users:
[User('reza', 'e', '1'), User('rezadl09', 'u', '5')]
-----
ali users:
[User('ali', 'a', '4'), User('ali01', 'b', '6')]
-----
mohammad users:
[]
```

درس ۲۳: تابع overriding و super متد

* به تغییر یا جایگزینی (بازنویسی) متدهای یک سوپرکلاس با متدهای جدید که هم اسم آن در زیرکلاس (کلاس ارث بری شده) است، گفته می‌شود..

مثال: در مثال زیر متد `init` سوپرکلاس با متد `init` کلاس `Buyer` جایگزین شده است.

```
class UserList(list):
    def search(self, user_name: str) -> list['User']:
        matching_users: list['User'] = []
        for user in self:
            if user_name in user.user_name:
                matching_users.append(user)
        return matching_users

class User:
    all_users: list['User'] = UserList()

    def __init__(self, user_name: str, email: str, password: str) -> None:
        self.user_name = user_name
        self.email = email
        self.password = password
        User.all_users.append(self)

    def __repr__(self):
        return f'{self.__class__.__name__}({self.user_name!r}, '\
               f'{self.email!r}, {self.password!r})'

    def __str__(self):
        return f'{self.user_name}'

class Seller(User):
    def order(self, order: 'Order') -> None:
        print(f'{self.user_name}, From your products, {order!r} was sold!')

class Buyer(User):
    def __init__(self, phone: str) -> None:
        self.phone = phone

buyer_account = Buyer('038549034')
print(buyer_account.user_name)

print(buyer_account.user_name)
^^^^^^^^^^^^^^^^^^^^^^^^^
AttributeError: 'Buyer' object has no attribute 'user_name'
```

* برای این که متدهای دلخواهمان را اضافه کنیم، می‌توانیم دقیقاً همان متدهای سوپرکلاس را در ساب کلاس (کلاس ارث بری شده) بنویسیم و ویژگی‌های جدید به آن اضافه کنیم. (استفاده از این روش بهینه نیست، چون کدها تکرار می‌شوند و هدف از ارث بری جلوگیری از تکرار است.)

```
class Buyer(User):
    def __init__(self, user_name, email, password, phone):
        self.phone = phone
        self.user_name = user_name
        self.email = email
        self.password = password
        User.all_users.append(self)

buyer_account = Buyer('ali', "ali@gmail.com", '123', '0912')
print(buyer_account.user_name) # -> ali
```

* برای این که متدهای دلخواهمان را اضافه کنیم، می‌توانیم از تابع `super` استفاده کنیم، `super` باعث دسترسی به سوپرکلاس می‌شود، `super` یک شی موقت است.

```
class Buyer(User):
    def __init__(self, user_name, email, password, phone):
        super().__init__(user_name, email, password)
        self.phone = phone

buyer_account = Buyer('ali', "ali@gmail.com", '123', '0912')
print(buyer_account.user_name) # -> ali
print(buyer_account.phone) # -> 0912
```

* ساب کلاس از `repr` سوپرکلاس استفاده می‌کند.

```
class User:
    all_users: list['User'] = UserList()

    def __init__(self, user_name: str, email: str, password: str) -> None: ...

    def __repr__(self):
        return f'{self.__class__.__name__}({self.user_name!r}, '\
               f'{self.email!r}, {self.password!r})'

class Buyer(User): ...

buyer_account = Buyer('ali', "ali@gmail.com", '123', '0912')
print(buyer_account.all_users) # -> [Buyer('ali', 'ali@gmail.com', '123')]
```

* برای حل این مشکل می‌توانیم متدها `__repr__` را برای ساب کلاس `User` `override` کنیم.

```
class Buyer(User):
    def __init__(self, user_name, email, password, phone):
        super().__init__(user_name, email, password)
        self.phone = phone

    def __repr__(self):
        return f'{self.__class__.__name__}({self.user_name!r}, '\
               f'{self.email!r}, {self.password!r}, {self.phone!r})'
```

```
buyer_account = Buyer('ali', "ali@gmail.com", '123', '0912')
print(buyer_account.all_users) # -> [Buyer('ali', 'ali@gmail.com', '123', '0912')]
```

* شی که از ساب کلاس ایجاد می‌کنیم، قطعاً علاوه بر نوع ساب کلاس از نوع سوپر کلاس نیز می‌باشد.

```
class User: ...
```

```
class Buyer(User): ...
```

```
buyer_account = Buyer('ali', "ali@gmail.com", '123', '0912')
print(isinstance(buyer_account, Buyer)) # -> True
print(isinstance(buyer_account, User)) # -> True
```

* شی که از سوپر کلاس می‌سازیم از نوع ساب کلاس نیست.

```
class User: ...
```

```
class Buyer(User): ...
```

```
u = User('ali', "ali@gmail.com", '123')
print(isinstance(u, Buyer)) # -> False
print(isinstance(u, User)) # -> True
```

```

class First():
    def __init__(self):
        print('first')

class Second(First):
    def __init__(self):
        super().__init__()
        print('second')

class Third(Second):
    def __init__(self):
        super().__init__()
        print('third')

class Fourth(Third):
    def __init__(self):
        super().__init__()
        print('fourth')

x = Fourth()
print('-' * 20)
print(isinstance(x, Third))
print(isinstance(x, Second))
print(isinstance(x, First))

```

first
second
third
fourth

True
True
True

* می‌توانیم یک متده از کلاس **Built-in** پایتون را نیز **override** کنیم. (این کار بسیار حساس است و نیاز به شناخت دقیق از کلاس و متدهای آن دارد).

مثال: در این مثال قابلیت `append` لیست را دچار تغییر می‌کنیم که فقط نوع کلاس `User` را به لیست اضافه کند و در غیر این صورت خطای دهد.

```
class UserList(list):
    def search(self, user_name: str) -> list['User']:
        matching_users: list['User'] = []
        for user in self:
            if user_name in user.user_name:
                matching_users.append(user)
        return matching_users

    def append(self, obj) -> None:
        if not isinstance(obj, User):
            raise TypeError('this list only accept User')
        return super().append(obj)

class User: ...

class Buyer(User): ...

class Seller(User): ...

b = Buyer('ali', "ali@gmail.com", '123', '0912')
s = Seller('reza', "reza@gmail.com", '454')
li = UserList()
li.append(b)
li.append(s)
print(li) # ->[Buyer('ali', 'ali@gmail.com', '123', '0912'), Seller('reza', 'reza@gmail.com', '454')]

li.append(4)
```

```
File "F:\python\pythonProject\test.py", line 25, in <module>
    li.append(4) #-> TypeError: this list only accept User
    ^^^^^^
```

درس ۲۴: وراثت چندگانه

* یک کلاس می‌تواند از چند کلاس به صورت همزمان ارث ببری کند. (بهتر است از ارث بری چندگانه استفاده نکنیم مگر در مواردی که راه دیگری نبود).

```
class SuperClass1:  
    def print1(self):  
        print('SuperClass1')  
        print(self)  
  
class SuperClass2:  
    def print2(self):  
        print('SuperClass2')  
        print(self)  
  
class SubClass(SuperClass1, SuperClass2):  
    pass
```

```
obj = SubClass()  
obj.print1()  
obj.print2()
```

```
SuperClass1  
<__main__.SubClass object at 0x000001F3583256D0>  
SuperClass2  
<__main__.SubClass object at 0x000001F3583256D0>
```

* اگر در سوپرکلاس‌ها متدهایی با نام مشترک باشد، در سابکلاس متدها را که صدابزنیم، متدهایی را اجرا می‌کند که به عنوان اولین کلاس به سابکلاس داده ایم.

```
class SuperClass1:  
    def print(self):  
        print('SuperClass1')  
        print(self)  
  
class SuperClass2:  
    def print(self):  
        print('SuperClass2')  
        print(self)  
  
class SubClass(SuperClass1, SuperClass2):  
    pass  
  
obj = SubClass()  
obj.print()
```

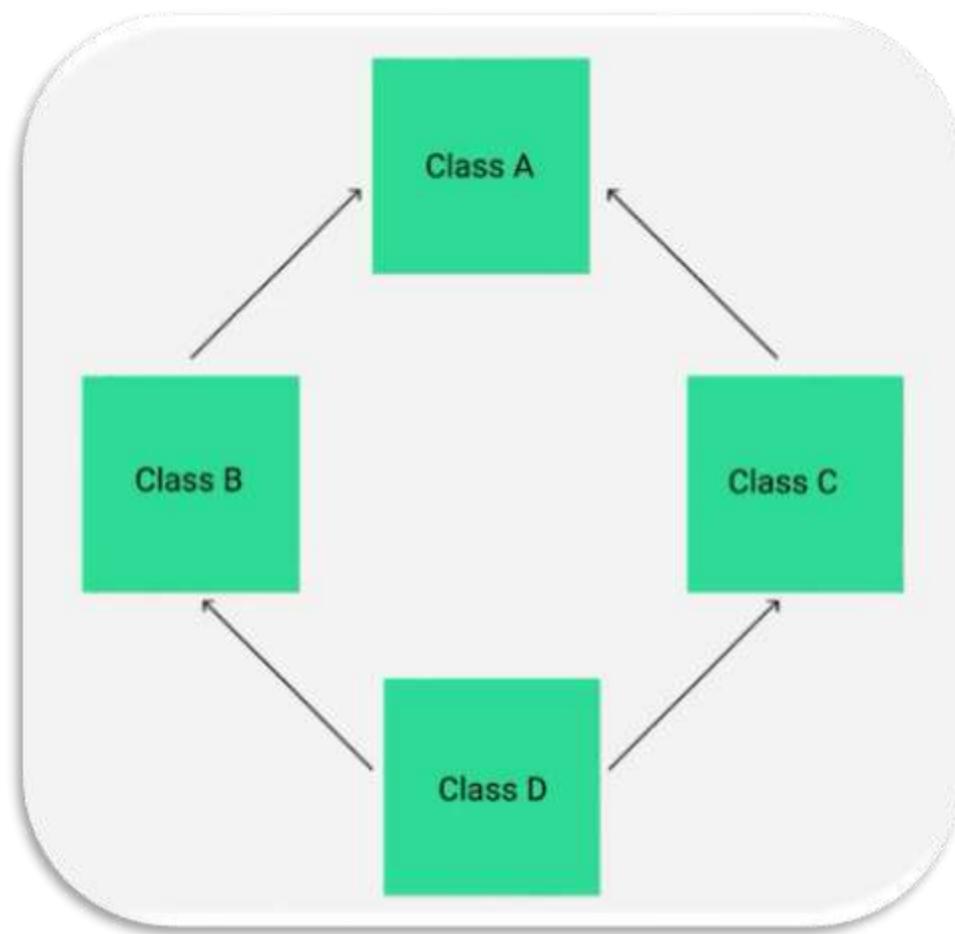
```
SuperClass1  
<__main__.SubClass object at 0x000001BCDBE25610>
```

* برای این که همهی متدهای با نام یکسان سوپرکلاس در سابکلاس اجرا شوند، می‌توانیم در آن متند سابکلاس، متند سوپرکلاس را صدابزنیم و مقدار دهی کنیم.

```
class SuperClass1:  
    def __init__(self, p1):  
        self.p1 = p1  
  
class SuperClass2:  
    def __init__(self, p2):  
        self.p2 = p2  
  
class SubClass(SuperClass1, SuperClass2):  
    def __init__(self, p1, p2, p3):  
        SuperClass1.__init__(self, p1)  
        SuperClass2.__init__(self, p2)  
        self.p3 = p3  
  
obj = SubClass(1, 2, 3)  
print(obj.p1) # -> 1  
print(obj.p2) # -> 2  
print(obj.p3) # -> 3
```

درس ۲۵: MRO و مشکل الماس در وراثت چندگانه

* هنگامی که یک کلاس از دو کلاس دیگر ارث بری کند و آن کلاس‌ها نیز از یک کلاس دیگر ارث بری کنند، به آن وراثت الماس می‌گوییم.



مثال: در این مثال متدهای `call_me` کلاس `BaseClass` دوبار صدا زده شده است چون `SubClass` از هردو کلاس به طور همزمان ارث می‌برد و آن‌ها نیز هر کدام از `BaseClass` ارث بری می‌کنند، که به این مشکل، مشکل الماس می‌گویند.

```
class BaseClass:
    num_base_calls = 0

    def call_me(self):
        print('Calling method on BaseClass!')
        self.num_base_calls += 1


class LeftSubClass(BaseClass):
    num_left_calls = 0

    def call_me(self):
        BaseClass.call_me(self)
        print('Calling method on LeftSubClass!')
        self.num_left_calls += 1


class RightSubClass(BaseClass):
    num_right_calls = 0

    def call_me(self):
        BaseClass.call_me(self)
        print('Calling method on RightSubClass!')
        self.num_right_calls += 1
```

```

class SubClass(LeftSubClass, RightSubClass):
    num_sub_calls = 0

    def call_me(self):
        LeftSubClass.call_me(self)
        RightSubClass.call_me(self)
        print('Calling method on SubClass!')
        self.num_sub_calls += 1

s = SubClass()
s.call_me()
print(s.num_sub_calls, s.num_left_calls, s.num_right_calls, s.num_base_calls)

```

```

Calling method on BaseClass!
Calling method on LeftSubClass!
Calling method on BaseClass!
Calling method on RightSubClass!
Calling method on SubClass!
1 1 1 2

```

- * برای حل این مشکل می‌توانیم از الگوریتم MRO استفاده کنیم.
- * این الگوریتم ممکن است ترتیب‌ها را رعایت نکند.
- * تابع super از الگوریتم MRO استفاده می‌کند.

```

class BaseClass:
    num_base_calls = 0

    def call_me(self):
        print('Calling method on BaseClass!')
        self.num_base_calls += 1

class LeftSubClass(BaseClass):
    num_left_calls = 0

    def call_me(self):
        super().call_me()
        print('Calling method on LeftSubClass!')
        self.num_left_calls += 1

class RightSubClass(BaseClass):
    num_right_calls = 0

    def call_me(self):
        super().call_me()
        print('Calling method on RightSubClass!')

```

```
self.num_right_calls += 1  
  
↓  
  
class SubClass(LeftSubClass, RightSubClass):  
    num_sub_calls = 0  
  
    def call_me(self):  
        super().call_me()  
        print('Calling method on SubClass!')  
        self.num_sub_calls += 1  
  
s = SubClass()  
s.call_me()  
print(s.num_sub_calls, s.num_left_calls, s.num_right_calls, s.num_base_calls)
```

```
Calling method on BaseClass!  
Calling method on RightSubClass!  
Calling method on LeftSubClass!  
Calling method on SubClass!  
1 1 1 1
```

* با متد `__mro__` می‌توانیم ترتیب را مشاهده کنیم.

```
pprint(SubClass.__mro__)
```

```
(<class '__main__.SubClass'>,  
<class '__main__.LeftSubClass'>,  
<class '__main__.RightSubClass'>,  
<class '__main__.BaseClass'>,  
<class 'object'>)
```

درس ۲۶: مقداردهی اولیه در وراثت چندگانه

* در وراثت الماس هنگامی که با **Super** مقداردهی اولیه کنیم، ممکن است با خطأ مواجه شویم، چون ساب کلاس از دو کلاس ارث بری می‌کند.

مثال: در این مثال برای **RightSubClass** با **Super** **LeftSubClass** مقدار فرستادیم اما برای **RightSubClass** نکردیم.

```
class BaseClass:  
    num_base_calls = 0  
  
    def __init__(self, a, b):  
        self.a = a  
        self.b = b  
  
    def call_me(self):  
        print('Calling method on BaseClass!')  
        self.num_base_calls += 1  
  
  
class LeftSubClass(BaseClass):  
    num_left_calls = 0  
  
    def __init__(self, c):  
        super().__init__()  
        self.c = c  
  
    def call_me(self):  
        super().call_me()  
        print('Calling method on LeftSubClass!')  
        self.num_left_calls += 1  
  
  
class RightSubClass(BaseClass):  
    num_right_calls = 0  
  
    def __init__(self, d, e, f):  
        self.d = d  
        self.e = e  
        self.f = f  
  
    def call_me(self):  
        super().call_me()  
        print('Calling method on RightSubClass!')  
        self.num_right_calls += 1  
  
  
class SubClass(LeftSubClass, RightSubClass):  
    num_sub_calls = 0  
  
    def __init__(self, a, b, c, d, e, f, g):  
        super().__init__(c)  
        self.g = g
```



```
def call_me(self):
    super().call_me()
    print('Calling method on SubClass!')
    self.num_sub_calls += 1
```

```
s = SubClass(1, 2, 3, 4, 5, 6, 7)
s.call_me()
```

```
super().__init__()
TypeError: RightSubClass.__init__() missing 3 required positional arguments: 'd', 'e', and 'f'
```

* و اگر مقادیر را برای `LeftSubClass` ارسال کنیم، خطای دهنده که به `RightSubClass` ارسال کردہ ایم.

```
class SubClass(LeftSubClass, RightSubClass):
    num_sub_calls = 0

    def __init__(self, a, b, c, d, e, f, g):
        super().__init__(c, d, e, f)
        self.g = g

    def call_me(self):
        super().call_me()
        print('Calling method on SubClass!')
        self.num_sub_calls += 1
```

```
s = SubClass(1, 2, 3, 4, 5, 6, 7)
s.call_me()
```

```
TypeError: LeftSubClass.__init__() takes 2 positional arguments but 5 were given
```

* برای حل این مشکل می‌توانیم از ستاره (*) قبیل از مقداردهی استفاده کنیم.

* در این روش باید حتماً ترتیب فراخوانی را بدانیم.

```
from pprint import pprint

class BaseClass:
    num_base_calls = 0

    def __init__(self, a, b, *args):
        self.a = a
        self.b = b

    def call_me(self):
        print('Calling method on BaseClass!')
```

```
    self.num_base_calls += 1  
  
↓  
  
class LeftSubClass(BaseClass):  
    num_left_calls = 0  
  
    def __init__(self, c, *args):  
        super().__init__(*args)  
        self.c = c  
  
    def call_me(self):  
        super().call_me()  
        print('Calling method on LeftSubClass!')  
        self.num_left_calls += 1  
  
  
class RightSubClass(BaseClass):  
    num_right_calls = 0  
  
    def __init__(self, d, e, f, *args):  
        super().__init__(*args)  
        self.d = d  
        self.e = e  
        self.f = f  
  
    def call_me(self):  
        super().call_me()  
        print('Calling method on RightSubClass!')  
        self.num_right_calls += 1  
  
  
class SubClass(LeftSubClass, RightSubClass):  
    num_sub_calls = 0  
  
    def __init__(self, g, *args):  
        super().__init__(*args)  
        self.g = g  
  
    def call_me(self):  
        super().call_me()  
        print('Calling method on SubClass!')  
        self.num_sub_calls += 1  
  
  
s = SubClass(1, 2, 3, 4, 5, 6, 7)  
pprint([s.a, s.b, s.c, s.d, s.e, s.f, s.g])
```

```
[6, 7, 2, 3, 4, 5, 1]
```

- * در این مثال با توجه به ترتیب ارسال ابتدا مقدار 1 به g از SubClass ارسال شده است و سپس مقدار 2 به c از LeftSubClass و مقدار 3، 4 و 5 به e، d و f از BaseClass و درنهایت مقدار 6 و 7 به RightSubClass ارسال شده است.
- * ترتیب فراخوانی را می‌توان با دستور `__mro__` مشخص کرد.

```
pprint(SubClass.__mro__)
```

```
(<class '__main__.SubClass'>,
 <class '__main__.LeftSubClass'>,
 <class '__main__.RightSubClass'>,
 <class '__main__.BaseClass'>,
 <class 'object'>)
```

- * اگر بخواهیم بدون دانستن ترتیب مقداردهی کنیم به جای ستاره (*) می‌توانیم، دو ستاره (*) بگذاریم و به صورت کلید و مقدار، مقدار را ارسال کنیم.

```
from pprint import pprint

class BaseClass:
    num_base_calls = 0

    def __init__(self, a, b, **kwargs):
        self.a = a
        self.b = b

    def call_me(self):
        print('Calling method on BaseClass!')
        self.num_base_calls += 1


class LeftSubClass(BaseClass):
    num_left_calls = 0

    def __init__(self, c, **kwargs):
        super().__init__(**kwargs)
        self.c = c

    def call_me(self):
        super().call_me()
        print('Calling method on LeftSubClass!')
        self.num_left_calls += 1


class RightSubClass(BaseClass):
    num_right_calls = 0

    def __init__(self, d, e, f, **kwargs):
        super().__init__(**kwargs)
        self.d = d
        self.e = e
```



```

self.f = f

def call_me(self):
    super().call_me()
    print('Calling method on RightSubClass!')
    self.num_right_calls += 1

class SubClass(LeftSubClass, RightSubClass):
    num_sub_calls = 0

    def __init__(self, g, **kwargs):
        super().__init__(**kwargs)
        self.g = g

    def call_me(self):
        super().call_me()
        print('Calling method on SubClass!')
        self.num_sub_calls += 1

```

```
s = SubClass(a=1, b=2, c=3, d=4, e=5, f=6, g=7)
pprint([s.a, s.b, s.c, s.d, s.e, s.f, s.g])
```

```
[1, 2, 3, 4, 5, 6, 7]
```

:مثال

```

from pprint import pprint

class UserList(list):
    def search(self, user_name: str) -> list['User']:
        matching_users: list['User'] = []
        for user in self:
            if user_name in user.user_name:
                matching_users.append(user)
        return matching_users

    def append(self, obj) -> None:
        if not isinstance(obj, User):
            raise TypeError('this list only accept User')
        return super().append(obj)

class User:
    all_users: list['User'] = UserList()

    def __init__(self, user_name: str, email: str, password: str, **kwargs) ->
None:
        self.user_name = user_name
        self.email = email
        self.password = password
        User.all_users.append(self)

```

```
def __repr__(self):
    return f'{self.__class__.__name__}({{self.user_name!r}},' \
           f' {self.email!r}, {self.password!r})'

def __str__(self):
    return f'{self.user_name}'

class Seller(User):
    def __init__(self, shaba, **kwargs):
        super().__init__(**kwargs)
        self.shaba = shaba

    def order(self, order: 'Order') -> None:
        print(f'{self.user_name}, From your products, {order!r} was sold!')

class Buyer(User):
    def __init__(self, phone: str, **kwargs) -> None:
        super().__init__(**kwargs)
        self.phone = phone

    def __repr__(self):
        return f'{self.__class__.__name__}({{self.user_name!r}},' \
               f' {self.email!r}, {self.password!r}, {self.phone!r})'

class SellerAndBuyer(Seller, Buyer):
    def __init__(self, score, **kwargs):
        super().__init__(**kwargs)
        self.score = score

def main():
    b = Buyer(user_name='ali', email="ali@gmail.com",
              password='123', phone='0912')
    s = Seller(user_name='reza', email="reza@gmail.com",
               password='454', shaba='NJ0139503')
    sb = SellerAndBuyer(user_name='amir', email='amir@gmail.com',
                         password='5tyy', shaba='UT5647', score='345',
                         phone='0911')

    sb.order('Hello')

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

```
amir, From your products, 'Hello' was sold!
```

درس ۲۷: تکالیف مبحث ارث بری

۱. سه کلاس ایجاد کنید: حیوانات، پستانداران و سگ.

الف - کلاس **Animal** باید دارای ویژگی‌های زیر باشد:

نام (string)

گونه (string)

ب - کلاس **Mammal** باید از کلاس **Animal** ارث بری کند و یک ویژگی اضافی داشته باشد:

تعداد پاها (integer)

ج - کلاس **Dog** باید از کلاس **Mammal** ارث بری کند و دارای ویژگی‌های زیر باشد:

نژاد (string)

د - همه نژادها باید متدهی به نام `(make_sound)` داشته باشند که صدایی را که حیوان تولید می‌کند چاپ می‌کند:

Animal: "Generic animal sound."

Mammal : "Generic mammal sound."

Dog : "woof!"

فراموش نکنید که از **super** برای مقداردهی کلاس پدر و **overriding** برای بازنویسی متدهای کلاس فرزند استفاده کنید.

درس ۲۸: چندریختی (polymorphism)

* مفهوم پلی مورفیسم یعنی چندشکلی بودن، یک اسم دارای رفتار و شکل‌های مختلف است، به عنوان مثال عملگر جمع نسبت به نوع داده‌ای که می‌گیرد و رفتار متفاوتی نشان می‌دهد.

```
print('r' + 'd') # -> rd
print(2 + 3) # -> 5
print(3 + 2j + 4j) # -> (3+6j)
```

:مثال

```
print(len([1, 2, 3, 4])) # -> 4
print(len('reza dolati')) # -> 11
print(len({'a': 4, 'b': 5})) # -> 2
```

* می‌توانیم از پلی مورفیسم در کلاس‌ها استفاده کنیم.

```
class Cat:
    def __init__(self, name='', color=''):
        self.name = name
        self.color = color

    def info(self):
        print(f'Hello I am a cat. My name is {self.name}, I am {self.color}')

    def make_sound(self):
        print('Meow...')

class Cow:
    def __init__(self, name='', color=''):
        self.name = name
        self.color = color

    def info(self):
        print(f'Hello I am a cow. My name is {self.name}, I am {self.color}')

    def make_sound(self):
        print('Moo...')

cat = Cat('Jacki', 'black')
cow = Cow('Gavi', 'brown')

for animal in [cat, cow]:
    animal.make_sound()
    animal.info()
```

```
Meow...
Hello I am a cat. My name is Jacki, I am black
Moo...
Hello I am a cow. My name is Gavi, I am brown
```

* می توانیم پلی مورفیسم کلاس ها را با استفاده از توابع پیاده سازی کنیم.

```
def func(obj):
    obj.make_sound()
    obj.info()
```

```
cat = Cat('Jacki', 'black')
cow = Cow('Gavi', 'brown')

func(cat)
func(cow)
```

```
Meow...
Hello I am a cat. My name is Jacki, I am black
Moo...
Hello I am a cow. My name is Gavi, I am brown
```

* رایج ترین حالت استفاده از پلی مورفیسم، در ارث بری است.

```
class Animal:
    def __init__(self, name='___', color='___'):
        self.name = name
        self.color = color

    def info(self):
        print(f'Hello I am an animal. My name is {self.name}, I am {self.color}')

    def make_sound(self):
        print('I can make sound!')


class Cat(Animal):
    def __init__(**kwargs):
        super().__init__(**kwargs)

    def info(self):
        print(f'Hello I am a cat. My name is {self.name}, I am {self.color}')

    def make_sound(self):
        print('Meow...')


class Cow(Animal):
    def __init__(**kwargs):
        super().__init__(**kwargs)

    def info(self):
        print(f'Hello I am a cow. My name is {self.name}, I am {self.color}')

    def make_sound(self):
        print('Moo...')


def func(obj):
    obj.make_sound()
```

```
obj.info()
```

```
animal = Animal()
cat = Cat(name='Jacki', color='black')
cow = Cow(name='Gavi', color='brown')

func(animal)
print('_' * 40)
func(cat)
print('_' * 40)
func(cow)
```

```
I can make sound!
Hello I am an animal. My name is ___, I am ___
-----
Meow...
Hello I am a cat. My name is Jacki, I am black
-----
Moo...
Hello I am a cow. My name is Gavi, I am brown
```

درس ۲۹: تکالیف مبحث چندریختی

۱. با استفاده از مفهوم پلی مورفیسم (چندریختی) در شی گرایی کلاس و زیرکلاس، وسیله نقلیه را به این شکل پیاده سازی کنید:

- (a) یک کلاس پدر با نام `Vehicle`
- (b) کلاس‌های فرزند به نام `Car`, `Bicycle`, `Airplane` ارث بری می‌کنند.
- (c) کلاس‌های فرزند به نام `Car`, `Benz`, `Lamborghini` ارث بری می‌کنند.
- (d) برای تمام کلاس‌های بالا رفتار (متدهای خودتان مانند حرکت کردن و ...) را اضافه کنید.
- (e) برای تمام کلاس‌های بالا ویژگی (اتربیوت) دلخواه خودتان مانند رنگ، وزن، سرعت و ... را اضافه کنید.

توجه داشته باشید که باید از `method overriding` برای متدهای هم‌نام استفاده کنید.

درس ۳۰: معرفی `Mixin` در وراثت چندگانه

* یک طرح و الگو در وراثت چندگانه است.

* یک کلاس است که فقط شامل متدهای دیگر متدهایی اضافه می‌کند و هدف آن توسعه کلاس‌های دیگر است.

* در `Mixin` رابطه از نوع `is-a` نیست.

* از کلاس `Mixin` بهتر است شی ایجاد نکنیم.

* از کلاس `Mixin` می‌توانیم برای دادن قابلیت‌های اختیاری به کلاس‌ها استفاده کنیم.

* بهتر است در اسم کلاس `Mixin` از `Mixin` استفاده کنیم تا مشخص شود که این کلاس `Mixin` است.

```
class WiFiMixin:  
    def connect_tp_wifi(self):  
        print('You are connected to Wi-Fi')
```

```
class Vehicle:  
    def move(self):  
        pass
```

```
class Car(Vehicle):  
    pass
```

```
class Airplane(Vehicle, WiFiMixin):  
    pass
```

```
class Motorcycle(Vehicle):  
    pass
```

```
airplane = Airplane()  
airplane.connect_tp_wifi()
```

You are connected to Wi-Fi

* در این مثال اگر قابلیت `Wi-Fi` را در سوپر کلاس می‌نوشیتم، کلاس‌هایی که این قابلیت را ندارند نیز این قابلیت را به ارث می‌برند که برای جلوگیری از این کار بهتر است از `Mixin` استفاده کنیم.

درس ۳۱: تکالیف مبحث mixin

۱. یک mixin به نام PrintInfoMixin ایجاد کنید که متدهای print_info را به یک کلاس اضافه می‌کند. متدهای print_info باید نام کلاس و ویژگی‌های آن را چاپ کند.

۲. یک میکسین به نام MathoperationsMixin ایجاد کنید که عملیات ریاضی پایه (جمع، تفریق، ضرب و تقسیم) را به یک کلاس اضافه می‌کند. کلاس مدنظر باید دارای اtribut‌های `x` و `y` باشد.

۳. یک میکسین به نام LoggerMixin ایجاد کنید که یک قابلیت لاگ را به یک کلاس اضافه می‌کند. هر زمان که متدهای از کلاس فراخوانی می‌شود، mixin باید نام متدها و آرگومان‌های آن را ثبت کند. (این تمرین نیاز به سرچ کرد و آشنایی با ماژول logging دارد)

درس ۳۲: نوع دهی اردکی (duck typing)

- * در نوع دهی اردکی، شی را براساس رفتار قضاوت می‌کنیم و نه چیزی که واقعاً هست.
- * پایتون از نوع دهی اردکی استفاده می‌کند، به عنوان مثال تابع `len` نوع ورودی مهم نیست و اگر آن شی متده است، به یک نحو رفتار می‌کند و قضاوت یکسانی دارد.

```
s = 'reza'  
t = (1, 2)  
print(len(s)) # -> 4  
print(len(t)) # -> 2
```

* اگر شی آن متده باشد و آن متده را صدا بزنیم با خطأ مواجه می‌شویم.

```
print(len(3))
```

```
print(len(3))  
^^^^^  
TypeError: object of type 'int' has no len()
```

- * برای این مشکل می‌توانیم از مفهوم **LBYL** (Look Before You Leap) استفاده کنیم، یعنی قبل از این‌که آن متده را برای شی صدا بزنیم، برسی کنیم که آیا آن متده دارد یا خیر، می‌توانیم از `hasattr` استفاده کنیم که این تابع بررسی می‌کند که آیا شی آن متده دارد یا خیر.

```
def check_len(obj):  
    if hasattr(obj, '__len__'):  
        print(len(obj))  
    else:  
        print('this object has no attribute len.')
```

```
check_len('reza') # -> 4  
check_len([1, 2, 4]) # -> 3  
check_len((1, 4)) # -> 2  
check_len(5) # -> this object has no attribute len.
```

- * به جای **LAFP** (it's easier to forgiveness than permission) بهتر است از **LBYL** استفاده کنیم، به این معنی که از استثنای برای مدیریت خطأ استفاده کنیم.

```
def check_len(obj):  
    try:  
        print(len(obj))  
    except:  
        print('this object has no attribute len.')
```

```
check_len('reza') # -> 4  
check_len([1, 2, 4]) # -> 3  
check_len((1, 4)) # -> 2  
check_len(5) # -> this object has no attribute len.
```

مثال:

```
class Duck:  
    def move(self):  
        print('I am swimming...')  
  
class Person:  
    def move(self):  
        print('I am walking...')  
  
class Plane:  
    def move(self):  
        print('I am flying...')  
  
def func(obj):  
    obj.move()  
  
d = Duck()  
h = Person()  
p = Plane()  
  
func(d)  
func(h)  
func(p)
```

```
I am swimming...  
I am walking...  
I am flying...
```

درس ۳۳: حل مثال: کلاس شکل‌های هندسی

* با استفاده `setattr` که ورودی آن یک کلید و مقدار است، می‌توانیم یک اتریبوت برای کلاس ایجاد کنیم.

```
class Shape:  
    def __init__(self, **kwargs):  
        for key, value in kwargs.items():  
            setattr(self, key, value)  
  
s = Shape(a=5, b=6)  
print(s.a) # -> 5  
print(s.b) # -> 6  
print(s.__dict__) # -> {'a': 5, 'b': 6}
```

مثال: کلاس شکل‌های هندسی

تصویر شکل	محیط	مساحت	نام شکل هندسی
	مجموع ۴ ضلع	یک ضلع × خودش	مربع
	مجموع ۴ ضلع	طول × عرض	مستطیل
	مجموع ۳ ضلع	(قاعده × ارتفاع) $\div 2$	مثلث
	مجموع ۴ ضلع	(قطر بزرگ × قطر کوچک) $\div 2$	لوزی
	مجموع ۴ ضلع	قاعده × ارتفاع	متوازی الاضلاع
	مجموع ۴ ضلع	(قاعده بزرگ + قاعده کوچک) \times ارتفاع $\div 2$	ذوزنقه
	πr^2	شعاع \times شعاع \times $\pi / 4$	دایره

```
class Shape:  
    def __init__(self, **kwargs):  
        for key, value in kwargs.items():  
            setattr(self, key, value)  
        self.area = 0  
        self.perimeter = 0  
  
    def calculate_area(self):  
        pass  
  
    def calculate_perimeter(self):  
        pass  
  
    def show(self):  
        info = ''  
        for key, value in self.__dict__.items():  
            if value > 0:  
                info += f'{key}: {value:.2f}\n'
```



```
print(info)

def __str__(self):
    return self.__class__.__name__

# length
class Square(Shape):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)

    def calculate_area(self):
        self.area = self.length ** 2

    def calculate_perimeter(self):
        self.perimeter = 4 * self.length

# length, width
class Rectangle(Shape):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)

    def calculate_area(self):
        self.area = self.length * self.width

    def calculate_perimeter(self):
        self.perimeter = 2 * (self.length + self.width)

# base, height, side1, side2
class Triangle(Shape):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)

    def calculate_area(self):
        self.area = (self.base * self.height) / 2

    def calculate_perimeter(self):
        self.perimeter = self.base + self.side1 + self.side2

# diameter1, diameter2, length
class Rhombus(Shape):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)

    def calculate_area(self):
        self.area = (self.diameter1 * self.diameter2) / 2

    def calculate_perimeter(self):
        self.perimeter = 4 * self.length
```



```
# length, height, width
class Parallelogram(Shape):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)

    def calculate_area(self):
        self.area = self.length * self.height

    def calculate_perimeter(self):
        self.perimeter = 2 * (self.length + self.width)

# height, base, top, side1, side2
class Trapezium(Shape):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)

    def calculate_area(self):
        self.area = 1 / 2 * self.height * (self.top + self.base)

    def calculate_perimeter(self):
        self.perimeter = self.top + self.base + self.side1 + self.side2

# radius
class Circle(Shape):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)

    def calculate_area(self):
        self.area = self.radius ** 2 * 3.14

    def calculate_perimeter(self):
        self.perimeter = 2 * self.radius * 3.14
```



درس ۳۴: اشیاء قابل فراخوانی (متده `call`)

* تابع یک شی است که قابلیت فراخوانی در آن وجود دارد.

```
def func(x=5):  
    print(x ** 2)  
  
func(8) # -> 64
```

* کلاس دارای قابلیت فراخوانی است.

```
class A:  
    def __init__(self, x):  
        self.x = x  
  
a = A(2)  
print(a.x) # -> 2
```

* با استفاده از تابع `callable` می‌توانیم مشخص کنیم که یک شی قابلیت فراخوانی دارد یا خیر.

```
def func(x=5):  
    print(x ** 2)  
  
class A:  
    def __init__(self, x):  
        self.x = x  
  
y = 4  
  
print(callable(func)) # -> True  
print(callable(A)) # -> True  
print(callable(y)) # -> False
```

* شی که از کلاس ایجاد می‌کنیم قابلیت فراخوانی ندارد.

```
class A:  
    def __init__(self, x):  
        self.x = x  
  
a = A(2)  
print(callable(a)) # -> False
```

* با استفاده از متدهای `__call__` می‌توانیم به اشیایی که از کلاس ایجاد می‌کنیم، قابلیت فراخوانی بدهیم.

```
class A:  
    def __init__(self, x):  
        print('init')  
        self.x = x  
  
    def __call__(self, z):  
        print('call')  
        self.z = z
```

```
obj = A(10)  
print(obj.x)  
obj(8)  
print(obj.z)
```

```
init  
10  
call  
8
```

مثال:

```
class Shape:  
    def __init__(self, **kwargs):  
        for key, value in kwargs.items():  
            setattr(self, key, value)  
        self.area = 0  
        self.perimeter = 0  
  
    def calculate_area(self):  
        pass  
  
    def calculate_perimeter(self):  
        pass  
  
    def show(self):  
        info = ''  
        for key, value in self.__dict__.items():  
            if value > 0:  
                info += f'{key}: {value:.2f}\n'  
        print(info)  
  
    def __str__(self):  
        return self.__class__.__name__
```

```
class Square(Shape):  
    def __init__(self, **kwargs):  
        super().__init__(**kwargs)  
  
    def calculate_area(self):  
        self.area = self.length ** 2  
  
    def calculate_perimeter(self):  
        self.perimeter = 4 * self.length
```



```
def __call__(self, length):  
    self.length = length
```



```
s = Square(length=4)  
s.calculate_area()  
s.calculate_perimeter()  
print(s)  
s.show()  
print(40 * '-')
```

```
s(8)  
s.calculate_area()  
s.calculate_perimeter()  
s.show()
```

```
Square  
length: 4.00  
area: 16.00  
perimeter: 16.00  
  
-----  
length: 8.00  
area: 64.00  
perimeter: 32.00
```

درس ۳۵: انواع متدها و حل مثال برای کلاس کامنت

- * **متدهای نمونه (Instance attribute)** رایج‌ترین مدت‌ها هستند و برای دستیابی به اtribیوت‌های نمونه (Instance method) از آن‌ها استفاده می‌شود، اtribیوت‌های نمونه برای هر شی به صورت اختصاصی وجود دارند و برای هر شی ممکن است متفاوت باشند.
- * **متدهای نمونه، self را به عنوان ورودی می‌گیرند.**
- * **atribیوت کلاس (class attribute)** بین همه‌ی اشیا مشترک است.

```
from datetime import datetime
```

```
class Comment:  
    # class attribute  
    website_name = 'sabzlearn.ir'  
  
    # instance method  
    def __init__(self, product, name, description, like, dislike):  
        self.product = product  
        self.name = name  
        self.description = description  
        self.date = datetime.now()  
        self.like = like  
        self.dislike = dislike
```

- * **متدهای کلاس (class method)** که به اtribیوت‌های کلاس (class attribute) دسترسی دارند و به جای self، کلاس را به عنوان ورودی می‌گیرند که طبق قرارداد اسم ورودی را cls می‌گذاریم.
- * برای این‌که یک متدهای کلاس متد شود باید از دکوراتور @classmethod را قبل از متد بنویسیم.

```
class Comment:  
    # class attribute  
    website_name = 'sabzlearn.ir'  
  
    @classmethod  
    def info(cls):  
        print(f'website name: {cls.website_name}')  
  
Comment.info() # -> website name: sabzlearn.ir
```

- * می‌توانیم از کلاس متدهای شی بسازیم.

```
from datetime import datetime  
  
class Product:  
    def __init__(self, product_name, price, off):  
        self.product_name = product_name  
        self.price = price  
        self.off = off
```



```
def __str__(self):
    return self.product_name
```

```
class Comment:
    # class attribute
    website_name = 'sabzlearn.ir'

    # instance method
    def __init__(self, product, name, description, like, dislike):
        self.product = product
        self.name = name
        self.description = description
        self.date = datetime.now()
        self.like = like
        self.dislike = dislike

    # instance method
    def show(self):
        print(f'product: {self.product}\n'
              f'name: {self.name}\n'
              f'description: {self.description}\n'
              f'date: {self.date}\n'
              f'like: {self.like} and dislike: {self.dislike}')

    @classmethod
    def info(cls):
        print(f'website name: {cls.website_name}')

    @classmethod
    def censorship(cls, product, name, description, like, dislike):
        print('the comment was censored')
        sn = description.replace('بیشур', '****')
        return cls(product, name, sn, like, dislike)

python_course = Product('python expert', 0, 0)

Comment.info()
c1 = Comment.censorship(python_course, 'reza', 'دوره بیشур بسیاری دوست!', 50, 10)

c1.show()
```

```
website name: sabzlearn.ir
the comment was censored
product: python expert
name: reza
description: دوره بیشур بسیاری دوست!
date: 2023-12-21 21:14:30.542024
like: 50 and dislike: 10
```

* متدهای استاتیک (static method) وابستگی کمتری به کلاس‌ها دارند، از آن‌ها استفاده می‌کنیم ولی هیچ کاری باشی و یا کلاس ندارند و می‌توانیم داخل و یا خارج از کلاس تعریف کنیم.

```
from datetime import datetime
from time import sleep

class Product: ...

class Comment: ...

python_course = Product('python expert', 0, 0)

# static method
def elapsed_time(time):
    sleep(3)
    print(datetime.now() - time)

Comment.info()
c2 = Comment.censorship(python_course, 'reza', '!', 50, 10)
c2.show()
elapsed_time(c2.date) # -> 0:00:03.000484
```

* برای تعریف متدهای استاتیک (static method) در داخل کلاس باید از دکوراتور @staticmethod قبل از متدهای استفاده کنیم.

```
from datetime import datetime
from time import sleep

class Product:
    def __init__(self, product_name, price, off):
        self.product_name = product_name
        self.price = price
        self.off = off

    def __str__(self):
        return self.product_name

class Comment:
    # class attribute
    website_name = 'sabzlearn.ir'

    # instance method
    def __init__(self, product, name, description, like, dislike):
        self.product = product
        self.name = name
        self.description = description
```



```
self.date = datetime.now()
self.like = like
self.dislike = dislike

# instance method
def show(self):
    print(f'product: {self.product}\n'
          f'name: {self.name}\n'
          f'description: {self.description}\n'
          f'date: {self.date}\n'
          f'like: {self.like} and dislike: {self.dislike}')

@classmethod
def info(cls):
    print(f'website name: {cls.website_name}')

@classmethod
def censorship(cls, product, name, description, like, dislike):
    print('the comment was censored')
    sn = description.replace('بیشور', '****')
    return cls(product, name, sn, like, dislike)

@staticmethod
# static method
def elapsed_time(time):
    sleep(3)
    print(datetime.now() - time)

python_course = Product('python expert', 0, 0)

Comment.info()
c2 = Comment.censorship(python_course, 'reza', '!', 50, 10)
c2.show()
c2.elapsed_time(c2.date) # -> 0:00:03.000500
```

* در این مثال نباید از **sleep** استفاده کنیم و در اینجا بخاطر درک عملکرد متده استفاده شده است.

درس ۳۶: getter و setter و property

- * از `set` برای کنترل مقدار یک خصوصیت و اعتبار سنجی استفاده می‌کنیم، اسم این دو دلخواه است اما معمولاً از `getter` و `setter` استفاده می‌کنیم، از `getter` برای تغییر و از `setter` برای برگرداندن استفاده می‌کنیم.
- * با `get` به جای دسترسی مستقیم به اtribیوت از این متدها استفاده می‌کنیم.

```
class Color:  
    def __init__(self, rgb, name):  
        self.rgb = rgb  
        self.name = name  
  
    def set_name(self, name):  
        self.name = name  
  
    def get_name(self):  
        return self.name  
  
    def set_rgb(self, rgb):  
        self.rgb = rgb  
  
    def get_rgb(self):  
        return self.rgb  
  
  
c = Color(0x6783F5, 'light blue')  
print(c.get_name())  
print(c.get_rgb())  
print(20 * '-')c.set_rgb(0x091B66)  
c.set_name('blue')  
print(c.get_name())  
print(c.get_rgb())
```

```
light blue  
6783989  
-----  
blue  
596838
```

* در این حالت کدها زیبایی خودشان را از دست می‌دهند و باید از متدهای **get** و **set** استفاده کنیم، برای حل این مشکل می‌توانیم از تابع استفاده کنیم که مقدار اول **getter** و مقدار دوم **setter** است و خود این تابع تشخیص می‌دهد چه زمانی از **property** چه زمانی از **setter** استفاده کند، این تابع باعث می‌شود که کدهایمان را همان‌طور که بدون **getter** و **setter** می‌نویسیم، بنویسیم و همچنین به راحتی کدهایمان را ارزیابی کنیم.

```
class Color:
    def __init__(self, rgb, name):
        self.rgb = rgb
        self.name = name

    def _set_name(self, name):
        if name:
            self._name = name
        else:
            raise ValueError(f'Invalid name {name!r}')

    def _get_name(self):
        return self._name

    def _set_rgb(self, rgb):
        self._rgb = rgb

    def _get_rgb(self):
        return self._rgb

    name = property(_get_name, _set_name)
    rgb = property(_get_rgb, _set_rgb)

c = Color(0x6783F5, 'light blue')
print(c.name)
print(c.rgb)
print(20 * '-')
c.rgb = 0x091B66
c.name = 'blue'
print(c.name)
print(c.rgb)
```

```
light blue
6783989
-----
blue
596838
```

درس ۳۷: آرگومان‌های دیگر property و دکوراتور

- * به آرگومان **property** می‌توانیم به عنوان ورودی **getter** و **setter** را ندهیم که در این صورت مقدار نمی‌تواند **get** یا **set** شود.
- * به آرگومان **property** ابتدا باید **getter** و سپس **setter** را ارسال کنیم، می‌توانیم بدون ترتیب ارسال کنیم به این صورت که نام اتریبوت را بنویسیم، برای **setter** از **fset** و برای **getter** از **fget** استفاده می‌کنیم.

```
class Color:  
    def __init__(self, rgb, name):  
        self.rgb = rgb  
        self.name = name  
  
    def _set_name(self, name):  
        if name:  
            self._name = name  
        else:  
            raise ValueError(f'Invalid name {name!r}')  
  
    def _get_name(self):  
        return self._name  
  
    name = property(fset=_set_name, fget=_get_name)
```

- * از آرگومان **fdel** (سومین آرگومان) در **property** زمانی استفاده می‌کنیم که بخواهیم یک اتریبوت را حذف کنیم.

```
def _del_name(self):  
    print('deleting...')  
    del self._name  
  
    name = property(fset=_set_name, fget=_get_name, fdel=_del_name)  
  
c = Color(0x6783F5, 'light blue')  
c.name = 5  
print(c.name)  
del c.name  
print(c.name)
```

```
5  
deleting...  
  
AttributeError: 'Color' object has no attribute '_name'. Did you mean: 'name'?
```

- * از آرگومان **doc** (چهارمین آرگومان) در **property** زمانی استفاده می‌کنیم که بخواهیم از داک استرینگ استفاده کنیم.

```
name = property(fset=_set_name, fget=_get_name, fdel=_del_name, doc='this is a doc')  
  
c = Color(0x6783F5, 'light blue')  
c.name = 5  
print(help(c))
```

```
name  
|   this is a doc
```

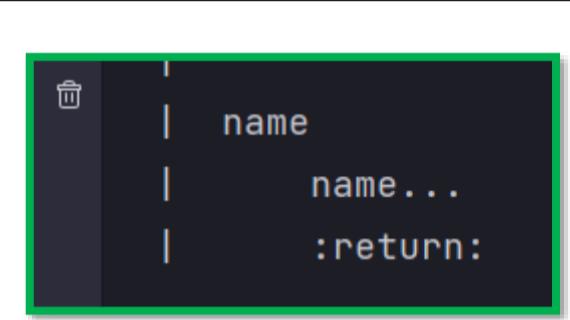
* اگر آرگومان `doc` را برای `property` ارسال نکنیم از داک استرینگ `getter` استفاده می‌کند.

```
def _get_name(self):
    """
    name...
    :return:
    """
    return self._name

def _del_name(self):
    print('deleting...')
    del self._name

name = property(_get_name, _set_name, _del_name)

c = Color(0x6783F5, 'light blue')
c.name = 5
print(help(c))
```



* تابع `property` را به صورت نیز می‌توانیم بنویسیم.

```
name = property()
name = name.setter(_set_name)
name = name.getter(_get_name)
name = name.deleter(_del_name)
```

* اگر دقت کنیم، توابع به عنوان آرگومان ارسال شده است که در واقع `property` یک دکوراتور است.

* می‌توانیم از دکوراتور `property` استفاده کنیم.

```
@property
def name(self):
    """
    name...
    :return:
    """
    return self._name

@name.setter
def name(self, name):
    if name:
        self._name = name
    else:
        raise ValueError(f'Invalid name {name!r}')

@name.deleter
def name(self):
    print('deleting...')
    del self._name
```

* در زمانی که به اعتبار سنجی نیاز داریم یا می‌خواهیم بر روی اtribیوت قبل از ذخیره شدن یک سری تغییرات اعمال شود از **property** استفاده می‌کنیم.

مثال: یک کلاس لیست جدید بنویسید که میانگین تمام عناصرش را نیز محاسبه کند.

```
class NewList(list):
    @property
    def ave(self):
        return sum(self) / len(self)

lst1 = NewList([1, 2, 3, 4])
print(lst1) # -> [1, 2, 3, 4]
print(lst1.ave) # -> 2.5
```

درس ۳۸: تکالیف مبحث **getter و setter و property**

۱. یک کلاس **Circle** با یک اتریبوت **radius** ایجاد کنید و یک **property** برای محاسبه مساحت بنویسید.
 ۲. کلاس **Circle** تمرین قبلی را گسترش دهید و یک متدهای **setter** برای **radius** اضافه کنید. متدهای **getter** باید اطمینان حاصل کند که شعاع همیشه مثبت است.
 ۳. یک کلاس **Temperature** ایجاد کنید که دما را بر حسب سانتی گراد نشان دهد. برای تبدیل دما به فارنهایت یک **property** بنویسید.
-

درس ۳۹: پیاده سازی رابطه composition و aggregation

- * روابط بین اشیا شامل رابطه (association)، تجمع (aggregation) و ترکیب (composition) است.
- * در رابطه (association) ارتباط بین اشیا خیلی کم است و وابستگی کمی باهم دارند، در این نوع رابطه معمولاً شی اول جزئی از شی دوم نیست.
- * تجمع (aggregation) یک نوع خاص رابطه (association) است که رابطه بین اشیا بیشتر است و رابطه جز به کل است و تعلق وجود دارد و اشیا کوچک تر متعلق به شی بزرگ‌تر هستند و وابستگی بین اشیا کم است و هر شی مستقل از دیگری است.
- * ترکیب (composition) یک نوع خاص تجمع (aggregation) است که رابطه بین اشیا بیشتر است و به شدت به هم وابسته اند و اشیا مستقل از هم وجود ندارند.
- * در پایتون بیشتر از ترکیب (composition) استفاده می‌کنیم.

مثال: یک کلاس برگه امتحانی که سوالات و جواب‌ها جزئی از آن هستند و این نوع رابطه از نوع ترکیب (composition) می‌باشد چون شی را داخل خود کلاس ایجاد کردیم و با نابودی شی اشیا دیگر جز آن شی نیز از بین می‌روند.

```
class Question:  
    def __init__(self, q: str, a: list):  
        self.q = q  
        self.a = a  
  
class ExamPaper:  
    def __init__(self):  
        self.question = Question('what is your name?', ['reza', 'ali'])  
  
    def __str__(self):  
        return f'{self.question.q}\n{self.question.a}'  
  
e = ExamPaper()  
print(e)
```

```
what is your name?  
['reza', 'ali']
```

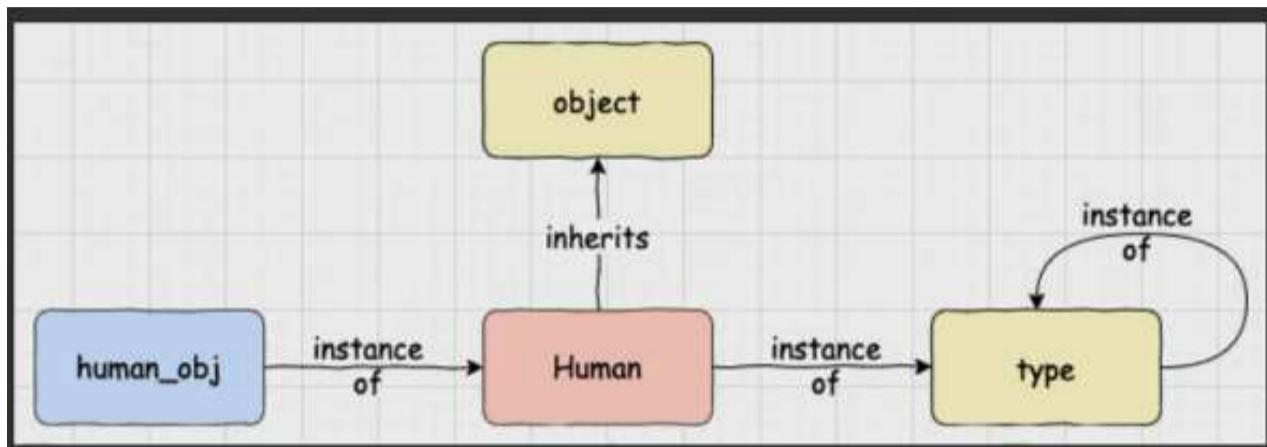
مثال: دانشجو و دانشگاه باهم رابطه تجمع (aggregation) دارند و شی دانشجو را مستقل از دانشگاه ایجاد می‌کنیم و به عنوان آرگومان به دانشگاه ارسال می‌کنیم و اگر شی دانشگاه نابود شود، شی دانشجو نابود نمی‌شود.

```
class Student:  
    def __init__(self, name, number):  
        self.name = name  
        self.number = number  
  
    def __str__():  
        return f'{self.name}: {self.number}'  
  
class University:  
    def __init__(self, students):  
        self.students = students  
  
st = [Student('naser', '12'), Student('reza', '98')]  
university = University(st)  
for s in university.students:  
    print(s)
```

```
naser: 12  
reza: 98
```

درس ۴۰: آشنایی با متاکلاس

- * همه چیز در پایتون یک شی (object) است، حتی خود کلاس‌ها نیز شی‌اند.
- * تمام کلاس‌هایی که می‌سازیم از کلاس object ارث بری می‌کنند.
- * تمام کلاس‌هایی که می‌سازیم یک نمونه از کلاس type هستند.
- * کلاس type نیز نمونه‌ای از خودش است.



```
class Human:  
    pass  
  
human_obj = Human()  
print(type(human_obj)) # -> <class '__main__.Human'>  
print(isinstance(human_obj, Human)) # -> True  
print(Human.__base__) # -> <class 'object'>  
print(type(Human)) # -> <class 'type'>  
print(isinstance(Human, type)) # -> True  
print(type(type)) # -> <class 'type'>
```

- * متاکلاس کلاسی است که اشیایی که از آن ایجاد می‌شوند کلاس هستند.
- * کلاس type یک متاکلاس است.
- * تمام کلاس‌های داخلی پایتون (list, str و ...) نیز یک نمونه از کلاس type هستند.

```
print(isinstance(str, type)) # -> True
```

- * در پایتون می‌توانیم متاکلاس ایجاد کنیم.
- * برای ساخت متاکلاس از کلاس type ارث بری می‌کنیم.
- * پس از ساخت متاکلاس برای ایجاد یک کلاس از آن متاکلاس به metaclass کلاس جدید مقدار مقدار متاکلاس می‌دهیم.

```
class HumanMeta(type):  
    pass  
  
class Human(metaclass=HumanMeta):  
    pass  
  
h = Human()  
print(type(h)) # -> <class '__main__.Human'>  
print(type(Human)) # -> <class '__main__.HumanMeta'>  
print(type(HumanMeta)) # -> <class 'type'>
```

درس ۴۱: ساخت کلاس انتزاعی

* کلاس انتزاعی (**abstract class**) کلاسی است که کلیات را در آن می‌نویسیم و جزئیات در کلاس‌هایی که از کلاس انتزاعی ارث بری می‌کنند، نوشته می‌شوند.

* در کلاس انتزاعی (**abstract class**) متدهای انتزاعی نوشته می‌شوند، یعنی فقط اسم متدها را می‌نویسیم و جزئیات در کلاس‌های ارث بری شده نوشته می‌شوند.

* در پایتون اگر بخواهیم کلاس انتزاعی بسازیم باید از ماثول **abc** استفاده کنیم.

* با ارث بری از کلاس **ABC** ماثول **abc** می‌توانیم کلاس انتزاعی بسازیم.

* برای ساخت متدهای انتزاعی باید از دکوراتور **abstractmethod** ماثول **abc** استفاده کنیم.

```
from abc import ABC, abstractmethod
```

```
class Vehicle(ABC):
    @abstractmethod
    def move(self):
        """this method should be implemented!"""
```

* به روش زیر و با استفاده از **Metaclass ABC** نیز می‌توانیم کلاس انتزاعی بسازیم.

```
from abc import ABCMeta, abstractmethod
```

```
class Vehicle(metaclass=ABCMeta):
    @abstractmethod
    def move(self):
        """this method should be implemented!"""

    @abstractmethod
    def repair(self):
        """this method should be implemented!"""
```

* کلاس‌های انتزاعی می‌توانند شامل متدهای عادی نیز باشند.

* از کلاس انتزاعی (**abstract class**) برخلاف کلاس‌های عادی (**concrete class**) نمی‌توانیم شی بسازیم.

```
from abc import ABC, abstractmethod

class Vehicle(ABC):
    @abstractmethod
    def move(self):
        """this method should be implemented"""

    @abstractmethod
    def repair(self):
        """this method should be implemented"""

    def class_name(self):
        print(self.__class__)

v = Vehicle()
```

```
Traceback (most recent call last):
  File "F:\python\pythonProject\test.py", line 17, in <module>
    v = Vehicle()
               ^
TypeError: Can't instantiate abstract class Vehicle with abstract methods move, repair
```

* متدهای انتزاعی می‌توانند داخل کلاس انتزاعی پیاده سازی شوند ولی بهتر است از این کار استفاده نکنیم.

* کلاسی که از کلاس انتزاعی ارث بری می‌کند تا زمانی که همه‌ی متدهای انتزاعی کلاس انتزاعی را پیاده سازی نکند کلاس انتزاعی محسوب می‌شود و نمی‌توانیم از آن شی بسازیم.

```
from abc import ABC, abstractmethod

class Vehicle(ABC):
    @abstractmethod
    def move(self):
        """this method should be implemented"""

    @abstractmethod
    def repair(self):
        """this method should be implemented"""

    def class_name(self):
        print(self.__class__)
```

```
class Car(Vehicle):
    pass

c = Car()
```

```
Traceback (most recent call last):
  File "F:\python\pythonProject\test.py", line 21, in <module>
    c = Car()
           ^
TypeError: Can't instantiate abstract class Car with abstract methods move, repair
```

* کلاس ارث بری شده باید همهی متدهای انتزاعی را پیاده سازی کند.

```
from abc import ABC, abstractmethod

class Vehicle(ABC):
    @abstractmethod
    def move(self):
        """this method should be implemented"""

    @abstractmethod
    def repair(self):
        """this method should be implemented"""

    def class_name(self):
        print(self.__class__)

class Car(Vehicle):
    def move(self):
        ...

c = Car()
```

```
Traceback (most recent call last):
  File "F:\python\pythonProject\test.py", line 22, in <module>
    c = Car()
               ^
TypeError: Can't instantiate abstract class Car with abstract method repair
```

* از کلاس ارث بری شده زمانی می‌توانیم شی ایجاد کنیم که همهی متدهای انتزاعی را پیاده سازی کند.

```
class Car(Vehicle):
    def move(self):
        ...

    def repair(self):
        ...

c = Car()
```

* می‌توانیم کلاس‌های انتزاعی بنویسیم که از کلاس انتزاعی ارث بری کنند و خودشان نیز دارای متدهای انتزاعی باشند.

```
from abc import ABC, abstractmethod

# abstract class
class Vehicle(ABC):
    @abstractmethod
    def move(self):
        """this method should be implemented!"""

    @abstractmethod
    def repair(self):
        """this method should be implemented!"""

    def class_name(self):
        print(self.__class__)

# abstract class
class LandVehicle(Vehicle):
    @abstractmethod
    def brake(self):
        """this method should be implemented!"""

class AirVehicle(Vehicle):
    @abstractmethod
    def eject(self):
        """this method should be implemented!"""

# concrete class
class Car(LandVehicle):

    def move(self):
        print('Driving...')

    def repair(self):
        print('Under repair...')

    def brake(self):
        print('Braking...')

# concrete class
class Airplane(AirVehicle):

    def move(self):
        print('flying...')

    def repair(self):
        print('Under repair...')
```



```
def eject(self):  
    print('Ejecting...')
```

```
c = Car()  
c.move()  
c.repair()  
c.brake()  
print(20 * '-')  
a = Airplane()  
a.move()  
a.repair()  
a.eject()
```

```
Driving...  
Under repair...  
Braking...  
-----  
flying...  
Under repair...  
Ejecting...
```

درس ۴۲: تکالیف مبحث کلاس انتزاعی

۱. یک کلاس انتزاعی Employee با متدهای انتزاعی calculate_salary ایجاد کنید. سپس، دو زیرکلاس SalariedEmployee و HourlyEmployee (محاسبه حقوق) را پیاده سازی کنید.

۲. یک کلاس انتزاعی Car با متدهای انتزاعی start و stop ایجاد کنید. سپس، دو زیرکلاس Sedan و SportsCar ایجاد کنید. برای هر زیرکلاس، متدهای start و stop را پیاده سازی کنید. علاوه بر این، به هر زیرکلاس یک اتربیوت برنده اضافه کنید تا نشان دهندهی برنده خودرو باشد.

۳. یک کلاس انتزاعی Person با متدهای انتزاعی greet (سلام) و introduce (معرفی) ایجاد کنید. سپس، دو زیرکلاس Teacher و Student ایجاد کنید. متدهای introduce و greet را برای هر زیرکلاس پیاده سازی کنید. علاوه بر این، ویژگی‌هایی مانند نام، سن و نقش (به عنوان مثال، "دانشآموز" یا "معلم") را به زیرکلاس اضافه کنید.

درس ۴۳: سربارگذاری (overloading) عملگرها

* یک عملگر با توجه به نوع شی ممکن است عملکرد متفاوتی داشته باشد که این کار را متدهای آن شی انجام می‌دهند که به این کار سربارگذاری عملگر (overloading) گفته می‌شود.

* ما می‌توانیم سربارگذاری عملگر را برای کلاس‌های خودمان پیاده سازی کنیم.

* متدهای مربوط به هر عملگر را می‌توانیم در [geekforgeeks](https://www.geekforgeeks.org/python-magic-methods/) ببینیم.

Operator	Magic Method
+	<code>__add__(self, other)</code>
-	<code>__sub__(self, other)</code>
*	<code>__mul__(self, other)</code>
/	<code>__truediv__(self, other)</code>
//	<code>__floordiv__(self, other)</code>
%	<code>__mod__(self, other)</code>
**	<code>__pow__(self, other)</code>
>>	<code>__rshift__(self, other)</code>
<<	<code>__lshift__(self, other)</code>
&	<code>__and__(self, other)</code>
	<code>__or__(self, other)</code>
^	<code>__xor__(self, other)</code>

Operator	Magic Method
<	<code>__lt__(self, other)</code>
>	<code>__gt__(self, other)</code>
<=	<code>__le__(self, other)</code>
>=	<code>__ge__(self, other)</code>
==	<code>__eq__(self, other)</code>
!=	<code>__ne__(self, other)</code>

Operator	Magic Method
-=	<code>__isub__(self, other)</code>
+=	<code>__iadd__(self, other)</code>
*=	<code>__imul__(self, other)</code>
/=	<code>__idiv__(self, other)</code>
//=	<code>__ifloordiv__(self, other)</code>
%=	<code>__imod__(self, other)</code>
**=	<code>__ipow__(self, other)</code>
>>=	<code>__irshift__(self, other)</code>
<<=	<code>__ilshift__(self, other)</code>
&=	<code>__iand__(self, other)</code>
=	<code>__ior__(self, other)</code>
^=	<code>__ixor__(self, other)</code>

مثال: در این مثال عملگر جمع را برای کلاس پیاده سازی می‌کنیم.

```
class Person:  
    def __init__(self, first_name, last_name, age, national_code):  
        self.first_name = first_name  
        self.last_name = last_name  
        self.age = age  
        self.national_code = national_code  
  
    def __add__(self, other):  
        return self.age + other.age  
  
person1 = Person('ali', 'mohammadi', 26, '1234')  
person2 = Person('reza', 'dolati', 27, '456')  
  
print(person1 + person2) # -> 53
```

مثال: در این مثال عملگر کوچکتر و بزرگتر را برای کلاس پیاده سازی می‌کنیم.

```
class Person:  
    def __init__(self, first_name, last_name, age, national_code):...  
  
    def __lt__(self, other):  
        return self.age < other.age  
  
    def __gt__(self, other):  
        return self.age > other.age  
  
person1 = Person('ali', 'mohammadi', 26, '1234')  
person2 = Person('reza', 'dolati', 27, '456')  
print(person1 < person2) # -> True  
print(person1 > person2) # -> False
```

مثال: در این مثال عملگر مساوی را برای کلاس پیاده سازی می‌کنیم.

```
class Person:  
    def __init__(self, first_name, last_name, age, national_code):...  
  
    def __eq__(self, other):  
        return self.national_code == other.national_code  
  
person1 = Person('ali', 'mohammadi', 26, '1234')  
person2 = Person('reza', 'dolati', 27, '456')  
person3 = Person('reza', 'd', 27, '456')  
print(person1 == person2) # -> False  
print(person2 == person3) # -> True
```

درس ۴۴: محدود کردن اtribووت‌ها با `__slots__`

* تمام اtribووت‌هایی که برای یک شی ایجاد می‌کنیم در یک دیکشنری ذخیره می‌شوند که با دستور `__dict__` می‌توانیم به آن‌ها دسترسی یابیم.

```
class MyClass:  
    def __init__(self, a, b):  
        self.a = a  
        self.b = b  
  
obj = MyClass(1, 4)  
obj.c = 6  
print(obj.__dict__) # -> {'a': 1, 'b': 4, 'c': 6}  
obj.__dict__['d'] = 4  
print(obj.__dict__) # -> {'a': 1, 'b': 4, 'c': 6, 'd': 4}
```

* برای محدود کردن اtribووت‌ها و جلوگیری از ایجاد اtribووت‌های اضافی می‌توانیم از `__slots__` در کلاس استفاده کنیم، این متد دسترسی به دیکشنری را نیز غیرفعال می‌کند.

```
class MyClass:  
    __slots__ = ('a', 'b')  
  
    def __init__(self, a, b):  
        self.a = a  
        self.b = b  
  
obj = MyClass(1, 4)  
obj.c = 6
```

```
Traceback (most recent call last):  
  File "F:\python\pythonProject\test.py", line 9, in <  
    obj.c = 6  
    ^^^^^^  
AttributeError: 'MyClass' object has no attribute 'c'
```

```
print(obj.__dict__)
```

```
print(obj.__dict__)  
^^^^^^^^^^^^^  
AttributeError: 'MyClass' object has no attribute '__dict__'. Did you mean: '__dir__'?
```

* از مزیت‌های `__slots__` می‌توان به صرفه جویی در حافظه و دسترسی سریع‌تر به اtribووت اشاره کرد.

* با `__slots__` مشخص می‌کنیم چه اtribیوت‌هایی مجاز است ساخته شود.

```
class MyClass:  
    __slots__ = ('a', 'b', 'c')  
  
    def __init__(self, a, b):  
        self.a = a  
        self.b = b  
  
obj = MyClass(1, 4)  
obj.c = 7  
print(obj.c) # -> 7
```

* اگر از یک کلاس از کلاسی ارث بری کند که در آن اtribیوت با `__slots__` محدود شده‌اند، در کلاس ارث بری شده می‌توانیم بدون مشکل اtribیوت‌های جدید معرفی کنیم نمی‌شود با `__dict__` به اtribیوت‌های سوپرکلاس دسترسی پیدا کرد.

```
class ParentClass:  
    __slots__ = ('a', 'b')  
  
    def __init__(self, a, b):  
        self.a = a  
        self.b = b  
  
class MyClass(ParentClass):  
  
    def __init__(self, a, b, c):  
        super().__init__(a, b)  
        self.c = c  
  
obj = MyClass(1, 4, 5)  
obj.d = 7  
print(obj.d) # -> 7  
print(obj.c) # -> 5  
print(obj.__dict__) # -> {'c': 5, 'd': 7}
```

درس ۴۵: ساخت `iterator` و `iterable` با کلاس‌ها

* کلاس‌هایی که ما ایجاد می‌کنیم `iterable` نیستند و نمی‌شود در اشیاء آن‌ها حلقه بزنیم.

```
class NewObj:  
    def __init__(self):  
        self.items = [1, 2, 3, 4]  
  
n = NewObj()  
for i in n:  
    print(i)
```

```
Traceback (most recent call last):  
  File "F:\python\pythonProject\test.py", line 7,  
    for i in n:  
TypeError: 'NewObj' object is not iterable
```

* با استفاده از متدهای `__iter__` در کلاس می‌توانیم شی‌هایی که از آن کلاس ایجاد می‌کنیم را `iterable` کنیم.

```
class NewObj:  
    def __init__(self):  
        self.items = [1, 2, 3, 4]  
  
    def __iter__(self):  
        for i in self.items:  
            yield i  
  
n = NewObj()  
for j in n:  
    print(j)
```

```
1  
2  
3  
4
```

* با استفاده از متدهای `__next__` در کلاس می‌توانیم شی‌هایی که از کلاس ایجاد می‌کنیم را `iterator` کنیم و دیگر نیازی نیست از استفاده کنیم `iter`.

```
class NewObj:  
    def __init__(self):  
        self.x = 0  
  
    def __next__(self):  
        self.x += 1  
        return self.x
```

```
n = NewObj()  
print(next(n))  
print(next(n))  
print(next(n))
```

```
1  
2  
3
```

* در این حالت نمی‌توانیم از `for` استفاده کنیم زیر شی `iterable` نیست.

* می‌توانیم هم‌زمان از `__next__` و `__iter__` استفاده کنیم که در این حالت حلقه `iter` را صدای زند و `next` هم را صدای زند.

```
class NewObj:  
    def __init__(self):  
        self.x = 5  
  
    def __iter__(self):  
        for i in range(4):  
            yield i  
  
    def __next__(self):  
        self.x += 2  
        return self.x
```

```
n = NewObj()  
for j in n:  
    print(j)  
  
print(20 * '-')  
print(next(n))  
print(next(n))  
print(next(n))
```

0
1
2
3

7
9
11

* اگر بخواهیم وقتی از `__next__` و `__iter__` هم‌زمان استفاده می‌کنیم، هماهنگ باشند و یک نتیجه را برگردانند بهتر است در `__next__(self)` خود شی (self) را برگردانیم که در این حالت در حلقه `for` سپس `__next__` را اجرا می‌کند.

```
class PowTwo:  
    def __init__(self, max_pow):  
        self.n = 0  
        self.max_pow = max_pow  
  
    def __iter__(self):  
        return self  
  
    def __next__(self):  
        if self.n <= self.max_pow:  
            result = self.n ** 2  
            self.n += 1  
            return result  
        else:  
            raise StopIteration  
  
n = PowTwo(5)  
for j in n:  
    print(j)
```

0
1
4
9
16
25

درس ۴۶: دکوراتور کلاس

- * با استفاده از کلاس نیز می‌توانیم دکوراتور بسازیم و همچنین دکوراتور را روی کلاس‌ها و متدها اعمال کنیم.
- * برای اعمال دکوراتور روی کلاس، به جای تابع کلاس را ارسال می‌کنیم که در این حالت دکوراتور روی شی‌هایی که از آن کلاس ایجاد می‌کنیم اعمال می‌شود.

```
from functools import wraps

def my_decorator_func(cls):
    @wraps(cls)
    def wrapper_func(*args, **kwargs):
        print('*' * 10)
        cls(*args, **kwargs)
        print('-' * 10)

    return wrapper_func

@my_decorator_func
class Test:
    pass

obj = Test()
```

```
*****  
-----
```

- * با استفاده از کلاس نیز می‌توانیم دکوراتور را پیاده سازی کنیم، که در این حالت کلاس دکوراتور حتما باید دو متده `__init__` و `__call__` در آن تعریف شده باشند.

- * در `__init__` تابع را به عنوان ورودی می‌گیریم و در `__call__` کارهایی را که می‌خواهیم روی آن تابع انجام بدهیم را می‌نویسیم.

```
class MyClassDecorator:
    def __init__(self, func):
        self.func = func

    def __call__(self):
        print('*' * 10)
        self.func()
        print('-' * 10)

@MyClassDecorator
def my_func():
    print('reza')

my_func()
```

```
*****  
reza  
-----
```

* اگر بخواهیم داک استرینگ و اسم تابع حفظ شود نمی‌توانیم `wraps` استفاده کنیم بلکه باید از `update_wrapper` مازول قبل از مقداردهی تابع در `__init__` `functools` استفاده کنیم.

```
from functools import update_wrapper

class MyClassDecorator:
    def __init__(self, func):
        update_wrapper(self, func)
        self.func = func

    def __call__(self):
        print('*' * 10)
        self.func()
        print('-' * 10)

@MyClassDecorator
def my_func():
    """This is my func"""
    print('reza')

print(my_func.__doc__) # -> This is my func
print(my_func.__name__) # -> my_func
```

* می‌توانیم دکوراتور کلاس را بر روی کلاس‌ها نیز اعمال کنیم فقط به جای ورودی تابع باید کلاس بدهیم.

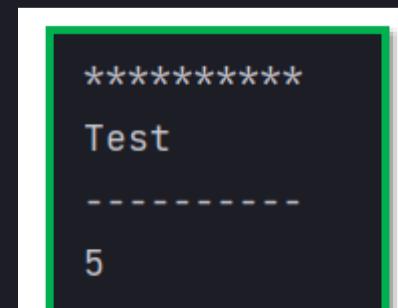
```
from functools import update_wrapper

class MyClassDecorator:
    def __init__(self, cls):
        update_wrapper(self, cls)
        self.cls = cls

    def __call__(self):
        print('*' * 10)
        print(self.cls.__name__)
        obj = self.cls()
        print('-' * 10)
        return obj

@MyClassDecorator
class Test:
    def __init__(self):
        self.a = 5

ob = Test()
print(ob.a)
```



درس ۴۷: توصیف گر (descriptor)

* اگر بخواهیم همهی اtribووت‌ها را با **property** ارزیابی کنیم، کار بسیار وقت گیر و همچنین تعداد کدها نیز بسیار زیاد می‌شود.

```
class Parent:  
    def __init__(self, child_name, father_name, mother_name):  
        self._child_name = child_name  
        self._father_name = father_name  
        self._mother_name = mother_name  
  
    @property  
    def child_name(self):  
        return self._child_name  
  
    @child_name.setter  
    def child_name(self, child_name):  
        if 0 < len(child_name) < 15:  
            self._child_name = child_name  
        else:  
            raise ValueError(f'Invalid name {child_name!r}')  
  
    @child_name.deleter  
    def child_name(self):  
        print('Deleting...')  
        del self._child_name  
  
    @property  
    def father_name(self):  
        return self._father_name  
  
    @father_name.setter  
    def father_name(self, father_name):  
        if 0 < len(father_name) < 15:  
            self._father_name = father_name  
        else:  
            raise ValueError(f'Invalid name {father_name!r}')  
  
    @father_name.deleter  
    def father_name(self):  
        print('Deleting...')  
        del self._father_name  
  
    @property  
    def mother_name(self):  
        return self._mother_name  
  
    @mother_name.setter  
    def mother_name(self, mother_name):  
        if 0 < len(mother_name) < 15:  
            self._mother_name = mother_name  
        else:  
            raise ValueError(f'Invalid name {mother_name!r}')
```

```
@mother_name.deleter
def mother_name(self):
    print('Deleting...')
    del self._mother_name
```



- * برای حل این مشکل می‌توانیم از **descriptor** استفاده کنیم.
- * کلاس‌های **descriptor** کلاس‌هایی هستند که یکی از متدهای **get**, **set** و **del** را **override** می‌کنند.
- * از **class attribute** فقط برای **descriptor** استفاده می‌شود.
- * اگر متدهای **__get__** و **__del__** را **override** کنیم و در یک کلاس دیگر شی ایجاد کنیم به جای آن شی چیزی را در زمان **print** شی نشان می‌دهد که در **__get__** نوشتم.

```
class NameField:
    def __init__(self, name=None):
        self.name = name

    def __get__(self, instance, owner):
        return 5

obj = NameField('reza')
print(obj) # -> <__main__.NameField object at 0x0000019FE30B02D0>

class Parent:
    name = NameField()

p = Parent()
print(p.name) # -> 5
```

- * آرگومان دوم (**instance**) متدهای **__get__** نشان می‌دهد که شی از چه کلاسی ایجاد شده است و آرگومان سوم (**owner**) متدهای **__get__** نشان می‌دهد که شی از چه نوعی است و اسم کلاس را نشان می‌دهد.

```
class NameField:
    def __init__(self, name=None):
        self.name = name

    def __get__(self, instance, owner):
        print(instance)
        print(owner)
        return 5

class Parent:
    name = NameField()

p = Parent()
print(p.name)
```

- * از متدهای زمانی استفاده می‌کنیم که بخواهیم یک متغیر را مقداردهی کنیم.
- * آرگومان دوم (instance) متدهای نشان می‌دهد که شی از چه کلاسی ایجاد شده است و آرگومان سوم (value) متدهای نشان دهندهٔ مقدار است.

```
class NameField:
    def __init__(self, name=None):
        self.name = name

    def __get__(self, instance, owner):
        return self.name

    def __set__(self, instance, value):
        if 0 < len(value) < 15:
            self.name = value
        else:
            raise ValueError(f'Invalid name {value!r}')

class Parent:
    name = NameField()

p = Parent()
p.name = 'mohammad'
print(p.name) # -> mohammad
```

- * شی name که از کلاس NameField در کلاس parent ایجاد کردیم، همانند یک اتریبوت عمل می‌کند و می‌توانیم مقدار بدهیم.

- * از این descriptor که ساختیم می‌توانیم به جای property استفاده کنیم و از طولانی شدن کدها جلوگیری کنیم.

```
class NameField:
    def __init__(self, name=None):
        self.name = name

    def __get__(self, instance, owner):
        return self.name

    def __set__(self, instance, value):
        if 0 < len(value) < 15:
            self.name = value
        else:
            raise ValueError(f'Invalid name {value!r}')

    def __delete__(self, instance):
        print('Deleting...')
        del self.name

class Parent:
    child_name = NameField()
    father_name = NameField()
```



```
mother_name = NameField()

def __init__(self, child_name, father_name, mother_name):
    self.child_name = child_name
    self.father_name = father_name
    self.mother_name = mother_name
```

```
p = Parent('reza', 'hassan', 'mahsa')
```

* در این حالت اگر بخواهیم مقادیر دیکشنری مقادیر را با مقدار خالی مواجه می‌شویم، چون این مقادیر در شی که از کلاس ساختیم ذخیره نمی‌شوند بلکه در شی از کلاس **descriptor** ذخیره می‌شوند.

```
print(p.__dict__) # -> {}
```

* برای حل این مشکل باید **descriptor** را تغییر بدھیم تا به جای ذخیره مقادیر در شی **descriptor** در شی که از کلاس ذخیره شوند و در هنگام ایجاد شی از **descriptor** در کلاس به عنوان مقدار باید اسم متغیر را بدھیم.

```
class NameField:
    def __init__(self, name=None):
        self.name = name

    def __get__(self, instance, owner):
        return instance.__dict__[self.name]

    def __set__(self, instance, value):
        if 0 < len(value) < 15:
            instance.__dict__[self.name] = value
        else:
            raise ValueError(f'Invalid name {value!r}')

    def __delete__(self, instance):
        print('Deleting...')
        del instance.__dict__[self.name]
```

```
class Parent:
    child_name = NameField('child_name')
    father_name = NameField('father_name')
    mother_name = NameField('mother_name')
```

```
def __init__(self, child_name, father_name, mother_name):
    self.child_name = child_name
    self.father_name = father_name
    self.mother_name = mother_name
```

```
p = Parent('reza', 'hassan', 'mahsa')
print(p.__dict__)
```

```
{'child_name': 'reza', 'father_name': 'hassan', 'mother_name': 'mahsa'}
```

* ارسال اسم متغیرها زیاد جالب نیست و به عنوان حل این مشکل می‌توانیم به جای `descriptor` در `__init__` از متدهای استفاده کنیم.

```
class NameField:
    def __set_name__(self, owner, name):
        self.name = name

    def __get__(self, instance, owner):
        return instance.__dict__[self.name]

    def __set__(self, instance, value):
        if 0 < len(value) < 15:
            instance.__dict__[self.name] = value
        else:
            raise ValueError(f'Invalid name {value!r}')

    def __delete__(self, instance):
        print('Deleting...')
        del instance.__dict__[self.name]

class Parent:
    child_name = NameField()
    father_name = NameField()
    mother_name = NameField()

    def __init__(self, child_name, father_name, mother_name):
        self.child_name = child_name
        self.father_name = father_name
        self.mother_name = mother_name

p = Parent('reza', 'hassan', 'mahsa')
print(p.__dict__)
```

```
{'child_name': 'reza', 'father_name': 'hassan', 'mother_name': 'mahsa'}
```

درس ۴۸: ساخت context manager با کلاس

* در پایتون اشیایی هستند به نام `context manager` که دو متده `enter` و `exit` داخلشان تعریف شده است که این اشیا قبل و بعد از دستورات یک سری کار انجام می‌دهند.

* همه‌ی `with` ها با دستور `with` فعال می‌شوند، به این صورت که `with` قبل از اجرای دستورات `enter` را صدا می‌زند و اجرا می‌کند و بعد از اتمام دستورات `exit` را صدا می‌زند و اجرا می‌کند.

```
class A:  
    def __enter__(self):  
        print('start!')  
  
    def __exit__(self, exe_type, exe_value, exe_tb):  
        print('end!')
```

```
with A():  
    print('I am Reza')
```

```
start!  
I am Reza  
end!
```

* اگر در شی `context manager` استثنای خطا رخ بدهد باز هم دستورات `__exit__` اجرا می‌شوند.

```
with A():  
    print('I am Reza')  
    print(10 / 0)
```

```
start!  
I am Reza  
end!  
ZeroDivisionError: division by zero
```

* نشان دهنده‌ی نوع خطاست.

* نشان دهنده‌ی مقدار خطاست.

* نشان دهنده‌ی مسیریابی خطاست.

```
class A:  
    def __enter__(self):  
        print('start!')  
  
    def __exit__(self, exe_type, exe_value, exe_tb):  
        print(exe_type)  
        print(exe_value)  
        print(exe_tb)  
        print('end!')  
  
with A():  
    print('I am Reza')  
    print(10 / 0)
```

```
<class 'ZeroDivisionError'>  
division by zero  
<traceback object at 0x000001B849600B40>  
end!  
Traceback (most recent call last):  
  File "F:\python\pythonProject\test.py", line  
      print(10 / 0)  
      ~~~^~~~  
ZeroDivisionError: division by zero
```

* اگر در `__exit__` مقدار `return True` کنیم، با خطا مواجه نمی‌شویم، اما دستورات پس از دستوری که منجر به خطا شده است، اجرا نمی‌شوند.

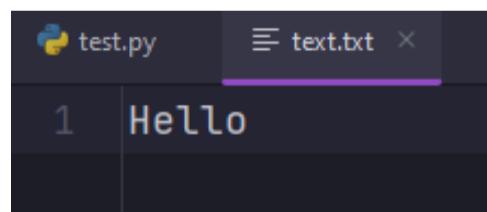
```
class A:  
    def __enter__(self):  
        print('start!')  
  
    def __exit__(self, exc_type, exc_value, exc_tb):  
        print('end!')  
        return True  
  
with A():  
    print('I am Reza')  
    print(10 / 0)  
    print('hello')  
    print('-----')  
    print(40 * 50)
```

start!
I am Reza
end!

* می‌توانیم `context manager` اختصاصی با کلاس بسازیم.

```
class FileManager:  
    def __init__(self, filename, mode):  
        self.filename = filename  
        self.mode = mode  
        self.file = None  
  
    def __enter__(self):  
        print('The file was opened!')  
        self.file = open(self.filename, self.mode)  
        return self.file  
  
    def __exit__(self, exc_type, exc_val, exc_tb):  
        print('The file was closed!')  
        self.file.close()  
  
with FileManager('text.txt', 'w') as f:  
    f.write('Hello')
```

The file was opened!
The file was closed!



درس ۴۹: دیتا کلاس (data class)

- * برای ذخیره سازی اطربیوت‌ها می‌توانیم از دیتا کلاس (data class) استفاده کنیم.
- * برای ایجاد دیتا کلاس باید دکوراتور `dataclass` را از مژول `dataclasses` وارد کنیم.
- * در دیتا کلاس نیازی نیست `__init__` را بنویسیم.

```
from dataclasses import dataclass

@dataclass
class Person:
    name: str
    family: str
    age: int

p = Person(name='reza', family='dolati', age=27)
print(p)
```

```
Person(name='reza', family='dolati', age=27)
```

- * در دیتا کلاس نوشتن نوع اطربیوت (در دیتا کلاس به اطربیوت `field` گفته می‌شود) اجباری است.

```
from dataclasses import dataclass

@dataclass
class Person:
    name
    family: str
    age: int

p = Person(name='reza', family='dolati', age=27)
print(p)
```

```
Traceback (most recent call last):
  File "F:\python\pythonProject\test.py", line 5, in <module>
    class Person:
  File "F:\python\pythonProject\test.py", line 6, in Person
    name
NameError: name 'name' is not defined
```

* در دیتا کلاس نیاز نیست که مشخص کنیم هر مقداری به چه فیلدی اختصاص دارد و به ترتیب مقداردهی می‌کند.

```
from dataclasses import dataclass
```

```
@dataclass  
class Person:  
    name: str  
    family: str  
    age: int
```

```
p = Person('reza', 'dolati', 27)  
print(p)
```

```
Person(name='reza', family='dolati', age=27)
```

* در دیتا کلاس به جای `__init__` می‌توانیم از `__post_init__` استفاده کنیم.

```
from dataclasses import dataclass
```

```
@dataclass  
class Person:  
    name: str  
    family: str  
    age: int
```

```
def __post_init__(self):  
    if self.age < 0:  
        self.age = 0
```

```
p = Person('reza', 'dolati', -21)  
print(p.age) # -> 0
```

* در دیتا کلاس اگر بخواهیم یک فیلد فقط به `_post__init__` ارسال شود باید نوع آن را `IntVar[]` استفاده کنیم، نوع در برآخت نوشته می‌شود و `import IntVar` نیز باید شود.

```
from dataclasses import dataclass, InitVar

@dataclass
class Person:
    name: str
    family: str
    age: int
    gender: InitVar[str]

    def __post_init__(self, gender):
        if gender == 'male':
            self.name = f'*{self.name}*' 

p = Person('reza', 'dolati', -21, 'male')
print(p.name) # -> *reza*
```

* اگر بخواهیم فیلدهای دیتاکلاس قابلیت تغییر نداشته باشند مقدار پارامتر `frozen` را در دیتاکلاس `True` قرار می‌دهیم، که در حالت پیش‌فرض `False` است.

```
from dataclasses import dataclass

@dataclass(frozen=True)
class Person:
    name: str
    family: str
    age: int

p = Person('reza', 'dolati', 20)
p.name = 'ali'
```

```
dataclasses.FrozenInstanceError: cannot assign to field 'name'
```

* به فیلدها می‌توانیم مقدار پیش‌فرض بدهیم.

```
from dataclasses import dataclass

@dataclass(frozen=True)
class Person:
    name: str
    family: str
    age: int = 18
```

* دیتا کلاس ها می توانند از یک دیگر ارث بری کنند که اولویت در مقدار دهی با کلاس بالاتر است.

```
from dataclasses import dataclass, InitVar
from pprint import pprint

@dataclass
class Person1:
    name: str
    family: str
    age: int

@dataclass
class Person2(Person1):
    name2: str
    family2: str
    age2: int = 18

p = Person2('reza', 'dolati', 20, 'Ahmad', 'nemati', 23)
pprint(p)
```

```
Person2(name='reza',
         family='dolati',
         age=20,
         name2='Ahmad',
         family2='nemati',
         age2=23)
```

* می توانیم فیلد از نوع کلاس اتریبوت ایجاد کنیم، باید نوع آن را **ClassVar[]** تعیین کنیم و در برآکت نوعش را بنویسیم، **typing** وارد کنیم.

```
from dataclasses import dataclass
from typing import ClassVar

@dataclass
class Person:
    name: str
    family: str
    age: ClassVar[int] = 4

p1 = Person('reza', 'dolati')
p2 = Person('ali', 'ahmadi')
print(Person.age) # -> 4
print(p1.age) # -> 4
print(p2.age) # -> 4
```

درس ۵۰: تکالیف مبحث سربارگذاری عملگرها

۱. یک کلاس به نام **Point** ایجاد کنید تا یک نقطه دو بعدی با مختصات x و y را نشان دهد. اپراتور (\cdot) را سربارگذاری کنید تا بتواند دو شی **Point** را باهم اضافه کند و یک شی **Point** جدید با مجموع مختصات آن‌ها تولید کند.
۲. یک کلاس به نام **ComplexNumber** برای نمایش اعداد مختلط ایجاد کنید. عملگرهای $(+)$ و $(-)$ و $(*)$ را برای مدیریت جمع، تفریق و ضرب اعداد مختلط سربارگذاری کنید.
۳. یک کلاس به نام **Matrix** ایجاد کنید تا یک ماتریس 2×2 را نشان دهد. عملگرهای $(+)$ و $(-)$ را برای مدیریت جمع و تفریق ماتریس بارگذاری کنید. (درمورد جمع و تفریق ماتریس سرج کنید).

پایان فصل دهم

درس ۱: استثناء چیست؟

* خطاهای ساختاری (SyntaxError) قبل از اجرای برنامه رخ می‌دهند و مربوط به رعایت نکردن قواعد نوشتاری پایتون می‌باشند.

The screenshot shows a Python file named `test.py` in the left editor pane. The code contains two lines:

```
1 while True
2     print('hi')
```

A red arrow points to the closing brace of the first line, indicating a syntax error. The right pane shows the output of the `Run` command, which includes the file path `F:\python\pythonProject\ve`, the code being run, and the resulting `SyntaxError: expected ':'`.

* به خطاهایی که پس از اجرای برنامه رخ می‌دهند، استثنای **RuntimeError** می‌گویند.

مثال: برنامه زیر از نظر سینتکس کاملا درست است اما در صورتی که کاربر یک عدد را بر صفر تقسیم کند، با خطای `ZeroDivisionError` مواجه می‌شویم که یک استثنای است (در برنامه نویسی تقسیم عدد بر صفر ممکن نیست).

```
x = int(input('x: '))
y = int(input('y: '))
print('x / y =', x / y)
```

مثال: برنامه زیر از نظر سینتکس کاملا درست است اما در هنگام اجرا با خطای **NameError** مواجه می‌شویم زیرا متغیر تعریف نشده است.

```
print(x + 5)
```

مثال: برنامه زیر از نظر سینتکس کاملا درست است اما در هنگام اجرا با خطای **TypeError** مواجه می‌شویم زیرا جمع رشته با عدد ممکن نیست.

```
print('2' + 3)
```

```
Traceback (most recent call last):
  File "F:\python\pythonProject\test.py", line 1, in <module>
    print('2' + 3)
               ~~~~~^~~
TypeError: can only concatenate str (not "int") to str
```

مثال: برنامه زیر از نظر سینتکس کاملا درست است اما در صورتی که کاربر یک رشته غیر عددی را وارد کند، با خطای **ValueError** مواجه می‌شویم چون برنامه باید رشته را به عدد تبدیل کند.

```
x = int(input('x: '))
print(x + 5)
```

```
x: s
Traceback (most recent call last):
  File "F:\python\pythonProject\test.py", line 1, in <module>
    x = int(input('x: '))
               ^^^^^^^^^^^^^^^^^^
ValueError: invalid literal for int() with base 10: 's'
```

* اگر با استثنای مواجه شویم برنامه متوقف می‌شود اما می‌توانیم این استثنای را مدیریت کنیم که در صورت رخداد آن‌ها بازهم برنامه متوقف نشود و اجرا شود.

درس ۲: دستور try/except

* با دستور `try` و `except` می‌توانیم استثناهای را مدیریت کنیم، پس از `try` دستورات را می‌نویسیم و پس از `except` مشخص می‌کنیم که اگر با استثنای مواجه شد چگونه عمل کند.

```
try:  
    x = int(input('x: '))  
    print(x)  
except:  
    print('Error!')
```

```
x: 5  
y: 0  
Error!
```

* اگر در بدنی `try` استثنای رخ دهد، بقیه دستورات بدن را اجرا نمی‌کند و دستورات `except` را اجرا می‌کند و اگر در بدنی `try` خطای خطا داشته باشد، دستورات `except` اجرا نمی‌شوند.

```
try:  
    x = int(input('x: '))  
    y = int(input('y: '))  
    print('x / y =', x / y)  
    print('+'*15)  
except:  
    print('Error!')
```

```
x: 4  
y: 0  
ZeroDivisionError!  
y: 3  
x / y = 1.3333333333333333
```

* اگر بدانیم که چه استثنایی ممکن است رخ دهد، در `except` می‌توانیم نوع استثنای را بنویسیم به این صورت که اگر استثنای رخ داد چگونه عمل کند.

```
x = int(input('x: '))  
y = int(input('y: '))  
  
while True:  
    try:  
        print('x / y =', x / y)  
        break  
    except ZeroDivisionError:  
        print('ZeroDivisionError!')  
        y = int(input('y: '))
```

```
x: s  
Error!
```

* می‌توانیم به تعداد دلخواه **except** بنویسیم و چند استثنای را همزمان مدیریت کنیم.

```
try:  
    x = int(input('x: '))  
    y = int(input('y: '))  
    print('x / y =', x / y)  
  
except ZeroDivisionError:  
    print('ZeroDivisionError!')  
  
except ValueError:  
    print('ValueError!')  
  
except:  
    print('UnknownError!')  
  
    print('UnknownError!')  
  
print('End!')
```

```
x: 4  
y: 0  
ZeroDivisionError!  
End!
```

```
x: s  
ValueError!  
End!
```

* می‌توانیم برای چند استثنای یک **except** بنویسیم.

```
try:  
    x = int(input('x: '))  
    y = int(input('y: '))  
    print('x / y =', x / y)  
  
except (ZeroDivisionError, ValueError):  
    print('Error!')  
  
print('End!')
```

```
x: 4  
y: 0  
ZeroDivisionError!  
End!
```

```
x: s  
ValueError!  
End!
```

* می‌توانیم به استثناهای با دستور **as** یک اسم مستعار بدهیم و از آن اسم مستعار استفاده کنیم.

```
try:  
    x = int(input('x: '))  
    y = int(input('y: '))  
    print('x / y =', x / y)  
  
except (ZeroDivisionError, ValueError) as E:  
    print(E)  
  
print('End!')
```

```
x: 6  
y: 0  
division by zero  
End!
```

```
x: s  
invalid literal for int() with base 10: 's'  
End!
```

* اگر نوع استثنای را ندانیم می‌توانیم از **Exception** به جای این‌که چیزی ننویسیم استفاده کنیم و یک اسم مستعار به آن بدهیم و سپس اسم آن شی را نمایش بدهیم.

```
try:  
    x = int(input('x: '))  
    y = int(input('y: '))  
    print('x / y =', x / y)  
  
except Exception as Ex:  
    print(Ex.__class__.__name__)  
  
print('End!')
```

```
x: 4  
y: 0  
ZeroDivisionError  
End!
```

```
x: s  
ValueError  
End!
```

درس ۳: دستور try/except/else

* می‌توانیم پس از `except` دستور `else` را نیز بنویسیم که زمانی اجرا می‌شود که در بدنه‌ی `else` با خطا مواجه نشویم.

```
def div():
    try:
        x = int(input('x: '))
        y = int(input('y: '))
        n = int(input('z: '))
        s = x + y
        result = s / n

    except TypeError as te:
        print(te.__class__.__name__)

    except ValueError as ve:
        print(ve.__class__.__name__)

    except ZeroDivisionError as zde:
        print(zde.__class__.__name__)

    else:
        print(f'** {result} **')

div()
```

```
x: 4
y: 2
z: 0
ZeroDivisionError
```

```
x: 4
y: s
ValueError
```

```
x: 4
y: 3
z: 4
** 1.75 **
```

درس ۴: دستور try/except/else/finally

* استثناء اتفاق بیفتد یا خیر دستورات داخل **finally** در هر صورت و قطعاً اجرا می‌شوند.

```
def div():
    try:
        x = int(input('x: '))
        y = int(input('y: '))
        n = int(input('z: '))
        s = x + y
        result = s / n

    except TypeError as te:
        print(te.__class__.__name__)

    except ValueError as ve:
        print(ve.__class__.__name__)

    except ZeroDivisionError as zde:
        print(zde.__class__.__name__)

    else:
        print(f'** {result} **')

    finally:
        print('-({~!@#$%^&*})-')
```

```
div()
```

```
x: 2
y: 3
z: 0
ZeroDivisionError
-({~!@#$%^&*})-
```

```
x: e
ValueError
-({~!@#$%^&*})-
```

```
x: 3
y: 2
z: 1
** 5.0 **
-({~!@#$%^&*})-
```

* اگر در **try** در تابع از **return** استفاده کنیم دستورات **else** **finally** اجرا نمی‌شوند.

* اگر در `return` از `finally` استفاده کنیم، فقط مقدار `finally` را بر می‌گرداند.

```
def div():
    try:
        x = int(input('x: '))
        y = int(input('y: '))
        n = int(input('z: '))
        s = x + y
        result = s / n
        return result
    except TypeError as te:
        print(te.__class__.__name__)
    except ValueError as ve:
        print(ve.__class__.__name__)
    except ZeroDivisionError as zde:
        print(zde.__class__.__name__)
    else:
        print(f'** {result} **')
        return 2

    finally:
        print()
        print('-(~!@#$%^&*)-')
        return 10

print(div())
```

```
x: 3
y: 1
z: 2

-(~!@#$%^&*)-
10
```

درس ۵: مدیریت استثنا تو در تو

* می‌توانیم از **try** های تو در تو استفاده کنیم، اما بهتر است این‌گونه عمل نکنیم.

```
def div(x, y, n):
    try:
        s = x + y
        result = s / n
        print(result)

    except ZeroDivisionError as zde:
        print(zde.__class__.__name__)
        try:
            print(z)
        except NameError:
            print('***')

div(3, 2, 0)
```

```
ZeroDivisionError
***
```

درس ۶: دستور raise

* می‌توانیم با دستور `raise` یک استثنا تولید کنیم، به این صورت که بعد از `raise` اسم استثنایی را که می‌خواهیم رخ بدهد را بنویسیم.

```
def div(x, y):
    if y == 0:
        raise ZeroDivisionError

    d = x / y
    print(d)

div(3, 0)
```

```
Traceback (most recent call last):
  File "F:\python\pythonProject\test.py", line 9, in <module>
    div(3, 0)
  File "F:\python\pythonProject\test.py", line 3, in div
    raise ZeroDivisionError
ZeroDivisionError
```

* می‌توانیم در `raise` پس از مشخص کردن نوع استثنا یک متن نیز بنویسیم تا آن را نمایش دهد.

```
def div(x, y):
    if y == 0:
        raise ZeroDivisionError('*** division by zero ***')

    d = x / y
    print(d)

div(3, 0)
```

```
Traceback (most recent call last):
  File "F:\python\pythonProject\test.py", line 9, in <module>
    div(3, 0)
  File "F:\python\pythonProject\test.py", line 3, in div
    raise ZeroDivisionError('*** division by zero ***')
ZeroDivisionError: *** division by zero ***
```

* در می‌توانیم مرجع استثنا را مشخص کنیم.

```
def div(x, y):
    try:
        d = x / y
        print(d)

    except ZeroDivisionError as zde:
        raise RuntimeError('oops') from zde

div(3, 0)
```

```
Traceback (most recent call last):
  File "F:\python\pythonProject\test.py", line 3, in div
    d = x / y
    ~~^~~
ZeroDivisionError: division by zero

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "F:\python\pythonProject\test.py", line 10, in <module>
    div(3, 0)
  File "F:\python\pythonProject\test.py", line 7, in div
    raise RuntimeError('oops') from zde
RuntimeError: oops
```

```
BaseException
├── BaseExceptionGroup
├── GeneratorExit
├── KeyboardInterrupt
├── SystemExit
└── Exception
    ├── ArithmeticError
    │   ├── FloatingPointError
    │   ├── OverflowError
    │   └── ZeroDivisionError
    ├── AssertionError
    ├── AttributeError
    ├── BufferError
    ├── EOFError
    ├── ExceptionGroup [BaseExceptionGroup]
    ├── ImportError
    │   └── ModuleNotFoundError
    ├── LookupError
    │   ├── IndexError
    │   └── KeyError
    ├── MemoryError
    ├── NameError
    │   └── UnboundLocalError
    ├── OSError
    │   ├── BlockingIOError
    │   ├── ChildProcessError
    │   ├── ConnectionError
    │   │   ├── BrokenPipeError
    │   │   ├── ConnectionAbortedError
    │   │   ├── ConnectionRefusedError
    │   │   └── ConnectionResetError
    │   ├── FileExistsError
    │   ├── FileNotFoundError
    │   ├── InterruptedError
    │   ├── IsADirectoryError
    │   ├── NotADirectoryError
    │   ├── PermissionError
    │   ├── ProcessLookupError
    │   └── TimeoutError
    ├── ReferenceError
    ├── RuntimeError
    │   ├── NotImplementedError
    │   └── RecursionError
    ├── StopAsyncIteration
    ├── StopIteration
    ├── SyntaxError
    │   └── IndentationError
    │       └── TabError
    ├── SystemError
    ├── TypeError
    ├── ValueError
    │   └── UnicodeError
    │       ├── UnicodeDecodeError
    │       ├── UnicodeEncodeError
    │       └── UnicodeTranslateError
    └── Warning
        ├── BytesWarning
        ├── DeprecationWarning
        ├── EncodingWarning
        ├── FutureWarning
        ├── ImportWarning
        ├── PendingDeprecationWarning
        ├── ResourceWarning
        ├── RuntimeWarning
        ├── SyntaxWarning
        ├── UnicodeWarning
        └── UserWarning
```

* می توانیم کلاس های استثنا اختصاصی خودمان را بنویسیم.

```
class DigitError(Exception):
    pass

def func(s):
    if not s.isdigit():
        raise DigitError('The string does not contain numbers! ')
    numbers = []
    for i in s:
        numbers.append(int(i))
    print(numbers)

func('a')
```

```
Traceback (most recent call last):
  File "F:\python\pythonProject\test.py", line 14, in <module>
    func('a')
  File "F:\python\pythonProject\test.py", line 7, in func
    raise DigitError('The string does not contain numbers! ')
DigitError: The string does not contain numbers!
```

* می توانیم کلاس استثنایی را که ساختیم شخصی سازی کنیم.

```
class DigitError(Exception):
    def __init__(self, s, message='Error'):
        self.message = message
        self.s = s
        super().__init__(self.message)

def func(s):
    if not s.isdigit():
        raise DigitError(s, 'The string does not contain only numbers!')
    numbers = []
    for i in s:
        numbers.append(int(i))
    print(numbers)

func('76035a')
```

```
Traceback (most recent call last):
  File "F:\python\pythonProject\test.py", line 17, in <module>
    func('76035a')
  File "F:\python\pythonProject\test.py", line 10, in func
    raise DigitError(s, 'The string does not contain only numbers!')
DigitError: The string does not contain only numbers!
```

درس ۸: مازول warnings

* از استثناهای **warnings** برای دادن پیام هشدار استفاده می‌کنیم به این صورت که از مازول **warn** تابع **warnings** را وارد می‌کنیم و آرگومان اول **warn** پیام هشدارمان را می‌نویسیم و برای آرگومان دوم **warn** نوع استثنای (نوشتن نوع استثنای اجباری نیست) را می‌نویسیم و برنامه متوقف نمی‌شود و اجرا می‌شود.

```
from warnings import warn

def func(x, y):
    if not isinstance(x, int) and not isinstance(y, int):
        warn('x and y should be integer!', UserWarning)
    print(x + y)

func('a', 'b')
print('ok')
```

```
F:\python\pythonProject\test.py:6: UserWarning: x and y should be integer!
  warn('x and y should be integer!', UserWarning)
ab
ok
```

درس ۹: دستور assert

* از زمانی استفاده می‌شود که انتظار داریم یک شرط درست باشد و **assert** را قبل از شرط می‌نویسیم تا در صورتی که شرط درست نباشد با استثنای **AssertionError** مواجه می‌شویم و برنامه متوقف می‌شود.

* از **assert** بیشتر برای دیباگ و تست نویسی استفاده می‌کنیم.

```
def func(x, y):
    assert isinstance(x, int) and isinstance(y, int), 'x and y should be integer'
    print(x + y)

func('a', 'b')
print('ok')
```

```
Traceback (most recent call last):
  File "F:\python\pythonProject\test.py", line 6, in <module>
    func('a', 'b')
  File "F:\python\pythonProject\test.py", line 2, in func
    assert isinstance(x, int) and isinstance(y, int), 'x and y should be integer'
AssertionError: x and y should be integer
```

درس ۱۰: تکالیف مبحث مدیریت خطای

۱. تابعی بنویسید که لیستی از اعداد را بگیرد و مجموع همه عناصر را برگرداند. اگر با هر عنصر غیر عدد (به عنوان مثال، یک رشته) مواجه شد، `TypeError` را مدیریت کنید و از آن عنصر در هنگام جمع بندی صرف نظر کنید.
۲. تابعی بنویسید که یک مسیر فایل را به عنوان ورودی گرفته و محتوای فایل را بخواند. `FileNotFoundException` را مدیریت کنید و در صورت نبود فایل پیغام خطای نمایش دهید.
۳. تابعی بنویسید که یک دیکشنری و یک کلید را به عنوان ورودی می‌گیرد و مقدار مربوط به کلید را برمی‌گرداند. `KeyError` را مدیریت کنید و "Key not found" را برگردانید.
۴. تابعی بنویسید که یک رشته را بگیرد و عدد صحیح آن را برگرداند. اگر رشته را نمی‌توان به عدد تبدیل کرد، `ValueError` را مدیریت کنید و `None` را برگردانید.
۵. تابعی بنویسید که لیستی از اعداد صحیح را از کاربر بخواند و مجموع همه اعداد را برگرداند. هر ورودی نامعتبر (مقادیر غیر صحیح) را با پیام خطای مناسب مدیریت کنید.

پایان فصل یازدهم

فصل دوازدهم: مباحث تکمیلی

درس ۱: متدهای کاربردی مازول `math` (ریاضی)

* عدد پی

```
import math

print(math.pi) # -> 3.141592653589793
```

* عدد اویلر (ثابت نپر، پایه لگاریتم طبیعی)

```
import math

print(math.e) # -> 2.718281828459045
```

* ثابت تاو

```
import math

print(math.tau) # -> 6.283185307179586
```

* بینهایت و منفی بینهایت

```
import math

print(math.inf) # -> inf
print(-math.inf) # -> -inf
```

* با متدهای `isinf` و `isnan` میتوانیم بررسی کنیم که یک عدد بینهایت یا خیر است.

```
import math

x = math.inf
print(math.isinf(x)) # -> True
```

* ثابت `nan` که با متدهای `isinf` و `isnan` میتوانیم بررسی کنیم که `nan` است یا خیر.

```
import math

x = math.nan
print(x) # -> nan
print(math.isnan(x)) # -> True
```

* متدهایی که بزرگترین عدد صحیح را نشان می‌دهد که بزرگ‌تر یا مساوی یک عدد است.

```
import math

x = 4.53
print(math.ceil(x)) # -> 5
```

* متدهایی که کوچکترین عدد صحیح را نشان می‌دهد که کوچک‌تر یا مساوی یک عدد باشد.

```
import math

x = 4.53
print(math.floor(x)) # -> 4
```

* از متدهایی که مطلقاً قدر مطلق استفاده می‌کنیم، این متدهای خود پایتون با نام `abs` موجود است.

```
import math

x = -4.53
print(math.fabs(x)) # -> 4.53
print(abs(x)) # -> 4.53
```

* از متدهایی که فاکتوریل یک عدد استفاده می‌کنیم.

```
import math

x = 5
print(math.factorial(x)) # -> 120
```

* از متدهایی که لگاریتم استفاده می‌کنیم، ورودی اول عدد و ورودی دوم پایه آن می‌باشد، به عنوان مثال لگاریتم `8` بر پایه `2`.

```
import math

print(math.log(8, 2)) # -> 3.0
```

* از متدهایی که لگاریتم `10` یک عدد و از `log2` برای محاسبه لگاریتم `2` استفاده می‌کنیم.

```
import math

print(math.log10(100)) # -> 2.0
print(math.log2(1024)) # -> 10.0
```

* از متدهایی که ریشه دوم (جذر) یک عدد استفاده می‌کنیم.

```
import math

print(math.sqrt(81)) # -> 9.0
```

* از متدهایی برای محاسبه یک عدد به توان عدد دیگر استفاده می‌کنیم، این متدها در پایتون نیز داریم.

```
import math

print(math.pow(2, 3)) # -> 8.0
print(pow(2, 3)) # -> 8
```

* از متدهایی برای محاسبه سینوس، از متدهایی برای محاسبه کسینوس، از متدهایی برای محاسبه تانژانت بر حسب رادیان استفاده می‌کنیم.

```
import math

print(math.sin(90)) # -> 0.8939966636005579
print(math.cos(90)) # -> -0.4480736161291701
print(math.tan(90)) # -> -1.995200412208242
```

* از متدهایی برای تبدیل رادیان به درجه و از متدهایی برای تبدیل درجه به رادیان استفاده می‌کنیم.

```
import math

print(math.degrees(1.5707963267948966)) # -> 90.0
print(math.radians(90)) # -> 1.5707963267948966
```

* متدهایی که عدد می‌گیرد و بخش عدد و اعشاری آن را جدا و در یک تایپل قرار می‌دهد.

```
import math

print(math.modf(3.27)) # -> (0.27, 3.0)
```

* متدهایی که دو عدد می‌گیرد و مقدار ولی و علامت دومی را برمی‌دارد.

```
import math

print(math.copysign(8, -9)) # -> -8.0
```

* متدهایی که باقیمانده تقسیم را نمایش می‌دهد.

```
import math

print(math.fmod(9, 2)) # -> 1.0
```

* متدهایی که لیست یا تایپل را باهم جمع می‌کند، `SUM` در پایتون این کار را برعهده دارد.

```
import math

print(math.fsum((1, 5, 7, 3))) # -> 16.0
print(sum((1, 5, 7, 3))) # -> 16
```

* متدهای gcd بزرگ‌ترین مقسوم علیه مشترک دو عدد را محاسبه می‌کند.

```
import math  
  
print(math.gcd(6, 12)) # -> 6
```

* متدهای lcm کوچک‌ترین مضرب مشترک دو عدد را محاسبه می‌کند.

```
import math  
  
print(math.lcm(6, 12)) # -> 12
```

* متدهای hypot وتر یک مثلث را محاسبه می‌کند.

```
import math  
  
print(math.hypot(3, 4)) # -> 5.0
```

* متدهای isqrt جذر یک عدد را محاسبه می‌کند و آن را به نزدیک‌ترین عدد صحیح گرد می‌کند.

```
import math  
  
print(math.isqrt(23)) # -> 4
```

* متدهای prod اعداد یک لیست یا تاپل را در هم ضرب می‌کند، اگر آرگومان start این متدر را مقدار بدھیم حاصل را در آن ضرب می‌کند.

```
import math  
  
print(math.prod([1, 2, 3, 4, 5])) # -> 120  
print(math.prod([1, 2, 3, 4, 5], start=2)) # -> 240
```

* متدهای trunc قسمت اعشار یک عدد را جدا می‌کند و قسمت صحیح را نمایش می‌دهد.

```
import math  
  
print(math.trunc(17.32432)) # -> 17
```

* از متدهای ماژول cmath برای کار با اعداد مختلط استفاده می‌شود.

درس ۲: قابلیت نسخه جدید پایتون: عملگر پایپ برای Union

* از Union برای بروزرسانی محدودیت نوع استفاده کنیم که مثلاً یک داده از چند نوع باشد.

```
from typing import Union
x: Union[int, float, str] = 4
y: Union[int, float, str] = 6.7
z: Union[int, float, str] = 'ali'
```

A screenshot of a terminal window titled "Local". The command "mypy test.py" is run, resulting in the output "Success: no issues found in 1 source file".

```
Terminal Local + ~
(venv) PS F:\python\pythonProject> mypy test.py
Success: no issues found in 1 source file
```

* به جای Union و وارد کردن آن از مازول typing می‌توانیم از اپراتور پایپ (|) استفاده کنیم.

```
x: int | float | str = 4
y: int | float | str = 2.454
z: int | float | str = 'ali'
```

A screenshot of a terminal window titled "Local". The command "mypy test.py" is run, resulting in the output "Success: no issues found in 1 source file".

```
Terminal Local + ~
(venv) PS F:\python\pythonProject> mypy test.py
Success: no issues found in 1 source file
```

درس ۳: دستور match-case

```
x = int(input('x: '))
y = int(input('y: '))
op = input('op: ')
if op == '+':
    print(x + y)
elif op == '-':
    print(x - y)

elif op == '*':
    print(x * y)

elif op == '/':
    print(x / y)
else:
    print('Invalid operator')
```

```
x: 4
y: 3
op: +
7
```

```
x: 4
y: 7
op: *
28
```

```
x: 4
y: 8
op: @
Invalid operator
```

* به جای **if** برای مقایسه مقدار می‌توانیم از ساختار **match-case** استفاده کنیم در این ساختار به جای **else** از **_** استفاده می‌کنیم.

```
x = int(input('x: '))
y = int(input('y: '))
op = input('op: ')

match op:
    case '+':
        print(x + y)
    case '-':
        print(x - y)
    case '*':
        print(x * y)
    case '/':
        print(x / y)
    case _:
        print('Invalid operator')
```

```
x: 4
y: 3
op: +
7
```

```
x: 4
y: 7
op: *
28
```

```
x: 4
y: 8
op: @
Invalid operator
```

مثال: دو تاس بندازیم و با توجه به نتیجه یک رفتار پیاده سازی کنیم.

```
from random import randint

x = randint(1, 6)
y = randint(1, 6)
print('x: ', x)
print('y: ', y)
match (x, y):
    case (6, 6):
        print('win')
    case (1, 1):
        print('lose')
    case (6, y):
        print('repeat')
    case (x, 6):
        print('repeat')
    case (x, y):
        print('next')
```

* می‌توانیم از یا در **match-case** استفاده کنیم، به این صورت که اگر این حالت نبود حالت دیگر نیز بود دستور اجرا شود.

```
x = int(input('x: '))
y = int(input('y: '))

match (x, y):
    case (6, 6) | (1, 1):
        print('win')
    case (x, y):
        print('repeat')
```

The image shows three separate terminal windows, each with a green border. The first window contains the output: x: 6, y: 6, win. The second window contains: x: 1, y: 1, win. The third window contains: x: 3, y: 4, repeat.

* می‌توانیم از شرط‌ها در **match-case** استفاده کنیم.

```
x = int(input('x: '))
y = int(input('y: '))

match (x, y):
    case (x, y) if 1 <= x <= 6 and 1 <= y <= 6:
        print('repeat')
    case (x, y):
        print('Invalid value')
```

The image shows two separate terminal windows, each with a green border. The first window contains the output: x: 3, y: 6, repeat. The second window contains: x: 7, y: 3, Invalid value.

* علاوه بر تاپل از لیست و دیکشنری نیز می‌توانیم در **match-case** استفاده کنیم.

```
x = int(input('x: '))
y = int(input('y: '))

match {'x': x, 'y': y}:
    case {'x': 6, 'y': 6}:
        print('win')
    case {'x': x, 'y': y}:
        print('lose')
```

x: 6
y: 6
win

x: 5
y: 2
lose

فقط اهمیت داشته باشد می‌توانیم از **ستاره (*)** برای بقیه‌ی متغیرها استفاده کنیم، از دو **ستاره (**)** برای دیکشنری استفاده می‌کنیم.

```
x = int(input('x: '))
y = int(input('y: '))
z = int(input('z: '))

match (x, y, z):
    case (6, *others):
        print('win')
    case _:
        print('lose')
```

x: 6
y: 5
z: 2
win

x: 4
y: 6
z: 6
lose

* از اسم مستعار می‌توانیم در **match-case** استفاده کنیم.

```
x = int(input('x: '))
y = int(input('y: '))
z = int(input('z: '))

match {'x': x, 'y': y, 'z': z}:
    case {'x': 6, **others} as a:
        print('win', a)
    case _ as b:
        print('lose', b)
```

x: 6
y: 5
z: 3
win {'x': 6, 'y': 5, 'z': 3}

x: 5
y: 6
z: 1
lose {'x': 5, 'y': 6, 'z': 1}

* می‌توانیم نوع‌ها را نیز در **match-case** بررسی کنیم.

```
x = 4
y = 's'

match (x, y):
    case (int(), str()):
        print('yes')
    case (int(), int()):
        print('no')
```

yes

درس ۴: کار با حالت تعاملی

* در صورت نیاز به فیلم مراجعه شود و یا داکیومنت پایتون مطالعه شود.

[لينک داکیومنت پایتون](#)

درس ۵: متدهای کاربردی مازول `datetime` (زمان و تاریخ)

* با متدهای `now()` و `today()` می‌توانیم زمان فعلی را مشخص کنیم.

```
import datetime
now = datetime.datetime.now()
print(now) # -> 2023-07-21 12:27:15.785370
```

* با متدهای `date()` و `time()` می‌توانیم تاریخ امروز را مشخص کنیم.

```
import datetime
now = datetime.date.today()
print(now) # -> 2023-07-21
```

* می‌توانیم تاریخ و زمان را تعیین کنیم.

```
import datetime
d = datetime.datetime(2023, 6, 12, 12, 45, 23)
print(d) # -> 2023-06-12 12:45:23
```

* می‌توانیم تاریخ را تعیین کنیم.

```
import datetime
d = datetime.date(2023, 6, 12)
print(d) # -> 2023-06-12
```

* می‌توانیم زمان را تعیین کنیم.

```
import datetime

d = datetime.time(11, 47, 36)
print(d) # -> 11:47:36
```

* می‌توانیم با استفاده از `fromtimestamp()` زمان را از سال ۱۹۷۰ محاسبه کنیم، ورودی آن ثانیه است و مشخص می‌کند که با گذشت این مقدار ثانیه از مبدأ (۱۹۷۰) به چه زمانی رسیده ایم.

```
from datetime import date

d = date.fromtimestamp(1635689734)
print(d) # -> 2021-10-31
```

* با `today()` می‌توانیم مشخص کنیم که امسال، این ماه و یا تاریخ امروز را نشان دهد.

```
from datetime import date

d = date.today()
print(d.year) # -> 2023
print(d.month) # -> 7
print(d.day) # -> 21
```

* در **time** می‌توانیم مشخص کنیم که ساعت، دقیقه و یا ثانیه آن را نشان دهد.

```
from datetime import time

d = time(17, 34, 24)
print(d.hour) # -> 17
print(d.minute) # -> 34
print(d.second) # -> 24
```

* می‌توانیم اختلاف دو تاریخ را محاسبه کنیم.

```
from datetime import datetime

d1 = datetime(2023, 3, 2, 17, 43, 22)
d2 = datetime(2023, 7, 21, 12, 50, 3)
d3 = d2 - d1
print(3) # -> 10367 days, 19:06:41
print(type(d3)) # -> <class 'datetime.timedelta'>
```

* با **timedelta** می‌توانیم مشخص کنیم که مثلاً چند هفته باهم چند روز یا ساعت اختلاف دارند و با **total_seconds** می‌توانیم مشخص کنیم که این اختلاف چند ثانیه است.

```
from datetime import timedelta

td1 = timedelta(weeks=3, hours=7, minutes=45)
td2 = timedelta(weeks=6, days=6, hours=7, seconds=45)
td3 = td2 - td1
print(td3) # -> 26 days, 23:15:45
print(td3.total_seconds()) # -> 2330145.0
```

* با استفاده از متدهای **strftime** می‌توانیم زمان را با فرمات دلخواه نمایش دهیم، حرف **d** مربوط به روز، **m** مربوط به ماه، **Y** مربوط به سال، **H** مربوط به ساعت، **M** مربوط به دقیقه، **S** مربوط به ثانیه و ... (اطلاعات بیشتر در [دایکومنت پایتون: لینک](#) می‌باشد، باید قبل از سال، ساعت و ... بگذاریم و متدهای **strptime** تاریخ را می‌گیرد و تاریخ را به شکل عادی نمایش می‌دهد، ورودی اول تاریخ و ورودی دوم فرمات است.)

```
from datetime import datetime

now = datetime.now()
print(now.strftime('%Y/%b/%d-%H:%M:%S')) # -> 2023/Jul/21-13:17:11
print(now.strptime('2023/Jul/21-13:14:21', '%Y/%b/%d-%H:%M:%S')) # -> 2023-07-21 13:14:21
```

درس ع: متدهای کاربردی مازول os (سیستم عامل)

* مازول **os** کمک می‌کند تا با سیستم عامل تعامل داشته باشیم.

* برای اینکه مشخص کنیم در چه سیستم عاملی هستیم از **nt** استفاده می‌کنیم که به معنای ویندوز و سیستم عامل‌های دیگر نیز اسم آن‌ها نمایش داده می‌شود.

```
import os

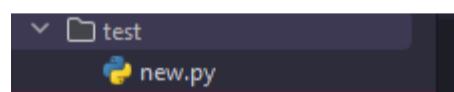
print(os.name) # -> nt
```

* با استفاده از متدهای **getcwd** می‌توانیم مشخص کنیم که در چه مسیری هستیم.

```
import os

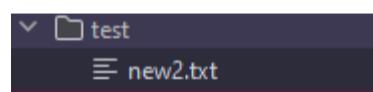
print(os.getcwd()) # -> F:\python\pythonProject
```

* با استفاده از متدهای **rename** می‌توانیم اسم یک فایل یا دایرکتوری را عوض کنیم که در ورود اول مسیر آن فایل و در ورودی دوم مسیر آن فایل با اسم جایگزین را مشخص می‌کنیم.



```
import os

os.rename('test/new.py', 'test/new2.txt')
```



* با استفاده از متدهای **access** می‌توانیم بررسی کنیم که به یک فایل دسترسی داریم یا خیر، به عنوان ورودی اول مسیر فایل را می‌دهیم و برای دسترسی داشتن از **os.F_OK**، برای قابل خواندن بودن از **os.R_OK**، برای قابل نوشتن بودن از **os.W_OK** و برای قابل اجرا بودن از **os.X_OK** به عنوان ورودی دوم استفاده می‌کنیم.

```
import os

print(os.access('test/new2.txt', os.F_OK)) # -> True
print(os.access('test/new2.txt', os.R_OK)) # -> True
print(os.access('test/new2.txt', os.W_OK)) # -> True
print(os.access('test/new2.txt', os.X_OK)) # -> True
```

* با استفاده از متدهای **listdir** می‌توانیم تمام فایل‌هایی که در دایرکتوری فعلی یا اینکه با مشخص کردن مسیر در دایرکتوری که می‌خواهیم را نمایش دهیم.

```
import os

print(os.listdir()) # -> ['.idea', '.mypy_cache', 'decorators', ...]
print(os.listdir('test')) # -> ['new2.txt']
```

* با استفاده از متده `mkdir` میتوانیم یک دایرکتوری جدید بسازیم.

```
import os  
  
os.mkdir('test/newdir')
```

* با استفاده از متده `makedirs` میتوانیم دایرکتوریهای تودرتو جدید بسازیم.

```
import os  
  
os.makedirs('test/newdir/anotherdir/lastdir')
```

* با استفاده از متده `chdir` میتوانیم مسیر دایرکتوری که در آن هستیم را تغییر بدهیم.

```
import os  
  
print(os.getcwd()) # -> F:\python\pythonProject  
os.chdir('test/newdir')  
print(os.getcwd()) # -> F:\python\pythonProject\test\newdir
```

* با استفاده از `environ` میتوانیم به متغیرهای محیطی دسترسی داشته باشیم.

```
import os  
  
print(os.environ) # -> environ({'ALLUSERSPROFILE': 'C:\\\\ProgramData', ... })
```

* با استفاده از متده `getlogin` میتوانیم مشخص کنیم چه کاربری وارد سیستم شده است.

```
import os  
  
print(os.getlogin()) # -> REZA
```

* با استفاده از متده `walk` میتوانیم محتویات یک دایرکتوری و زیر دایرکتوریها یش را مشخص کنیم.

```
import os  
  
for i in os.walk('test'):   
    print(i)
```

```
('test', ['newdir'], ['new2.txt'])  
(('test\\\\newdir', [], []))
```

* با استفاده از متده `remove` میتوانیم یک فایل را حذف کنیم.

```
import os  
  
os.remove('test/new2.txt')
```

* با استفاده از متدهای `rmdir` می‌توانیم یک دایرکتوری خالی را حذف کنیم.

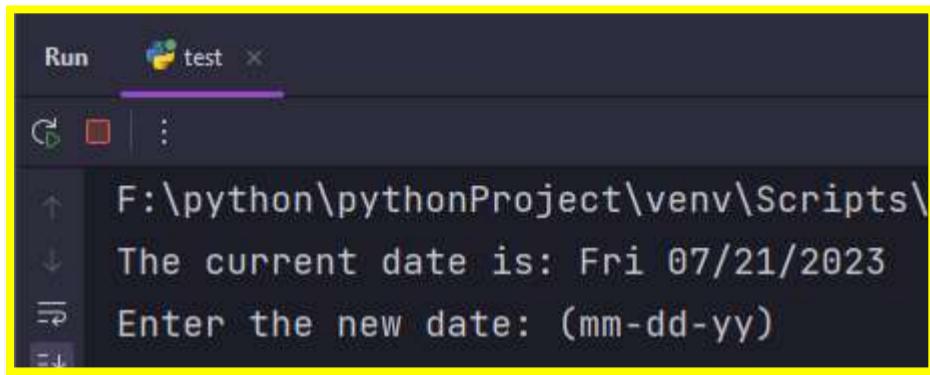
```
import os

os.rmdir('test/newdir/anotherdir/lastdir')
```

* با استفاده از متدهای `system` می‌توانیم دستورات `cmd` را اجرا کنیم.

```
import os

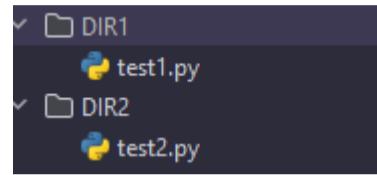
os.system('date')
```



درس ۷: متدهای کاربردی ماذول shutil (عملیات فایل)

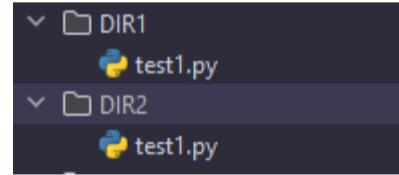
* با استفاده از متدهای `copyfile` می‌توانیم یک فایل را کپی کنیم، ورودی اول مسیر فایل و ورودی دوم مسیری که می‌خواهیم کپی شود، می‌توانیم با نام جدید کپی کنیم.

```
import shutil  
  
shutil.copyfile('DIR1/test1.py', 'DIR2/test2.py')
```



* اگر بخواهیم اسم فایل تغییر نکند می‌توانیم از متدهای `copy` این ماذول استفاده کنیم، این متدها متأذیت (اطلاعاتی در مورد زمان ایجاد، دسترسی و ...) را منتقل نمی‌کنند برای انتقال متأذیتی به جای `copy` می‌توانیم از `copy2` استفاده کنیم.

```
import shutil  
  
shutil.copy('DIR1/test1.py', 'DIR2')
```



* اگر بخواهیم همه فایلهای یک دایرکتوری کپی شود از متدهای `copytree` استفاده می‌کنیم، متدهای اول دایرکتوری مبدا و متدهای دوم دایرکتوری جدید که می‌خواهیم فایلها در آن کپی شود می‌باشد.

```
import shutil  
  
shutil.copytree('DIR1', 'DIR3')
```

* با استفاده از متدهای `rmtree` می‌توانیم یک دایرکتوری را با محتویاتش پاک کنیم.

```
import shutil  
  
shutil.rmtree('DIR3')
```

* با استفاده از متدهای `move` می‌توانیم یک فایل را از یک دایرکتوری به دایرکتوری دیگر منتقل کنیم، اگر فایل از قبل وجود داشته باشد آن را جایگزین می‌کند.

```
import shutil  
  
shutil.move('DIR1/test1.py', 'DIR2')
```

درس ۱: چرا همزمانی

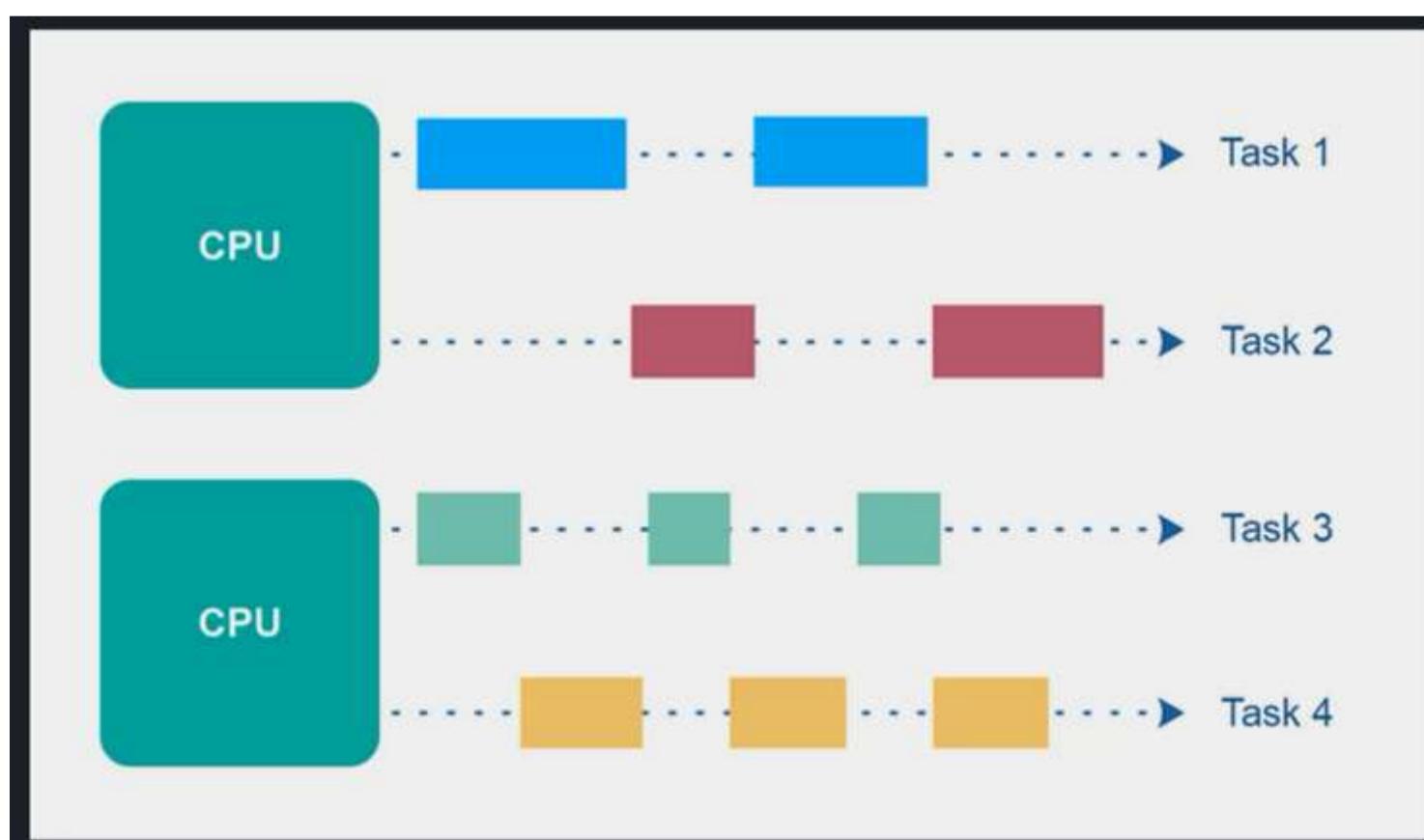
- * از **ترانزیستور** برای ذخیره ولتاژ الکتریکی که معادل یک بیت است استفاده می‌شود، اگر ولتاژ ترانزیستور سطح بالا باشد، روشن (۱) است و اگر ولتاژ سطح پایین باشد، خاموش (۰) است.
- * یک مجموعه‌ی خیلی بزرگ از ترانزیستورهاست و **وظیفه‌ی CPU** پردازش دستورهاست.
- * به رشته‌ای از دستورعمل‌ها گفته می‌شود که برای پردازنده ارسال می‌شوند.
- * ما باید بتوانیم برنامه‌هایی بنویسیم که همزمان اجرا شوند.

درس ۲: تفاوت موازی سازی و همزمانی

- * **همزمانی (Concurrency)** به معنی انجام چندین کار به صورت همزمان می‌باشد، به این صورت که چند کار را همزمان انجام می‌دهیم، از هر کار نوبتی یک بخش را انجام می‌دهیم تا کارها به پایان برسند، مانند یک آشپز که همزمان هم در حال خورد کردن است و هم در حال اضافه کردن ادویه، اما در واقع این کارها را نوبتی و بسیار سریع انجام می‌دهد و بین آن‌ها جابه‌جا می‌شود تا به پایان برسند.
- * **موازی سازی (parallelism)** به معنی انجام چندین کار به صورت همزمان به معنی واقعی کلمه می‌باشد.

درس ۳: مفهوم process, thread, coroutine

- * در **همزمانی** به یک جز یا موجودیتی که قادر به انجام یک کار و یا مجموعه‌ای از دستورات به صورت همزمان می‌باشد **واحد اجرا (execution unit)** گفته می‌شود.



* در **همزمانی** یک کار را به کارهای کوچک‌تر تقسیم می‌کنیم تا به صورت همزمان انجام شوند.

* در پایتون از واحدهای اجرایی **process**, **thread** و **coroutine** پشتیبانی می‌شود.

* **فرایند (process)** یک نمونه (**instance**) از برنامه‌ی کامپیوترا در حال اجرا که شامل کد برنامه و کارهایی است که قرار است انجام شوند، هر فرایند فضای حافظه و منابع سیستم خودش را دارد و مستقل از فرایندهای دیگر است، فرایندها می‌توانند در کنار هم‌دیگر و همزمان باهم اجرا شوند و امکان موازی سازی را نیز فراهم کنند.

* کوچک‌ترین واحد پردازشی که می‌توانیم در یک سیستم عامل اجرا کنیم، **thread** مجموعه‌ای از چند دستور عمل در یک برنامه است که می‌تواند به صورت مستقل اجرا شود، **thread** در واقع یک واحد اجرا داخل **process** می‌باشد، یک **process** می‌تواند **thread** چندین داشته باشد، **thread**ها فضای حافظه و منابع پردازشی یکسانی را به اشتراک می‌گذارند، فقط از **process** تشکیل نشده است و واحدهای دیگری نیز دارد، کنترل **thread** توسط سیستم عامل انجام می‌شود.

* **coroutine** داخل **thread** است و کنترل آن توسط خود برنامه انجام می‌شود، **coroutine** عملکردی است که می‌تواند خودش را به حالت تغییر در بیارد و بعدا خودش را از سر بگیرد، **coroutine** به ما این قابلیت را می‌دهد که چند کار کنترل را به یک دیگر واگذار کنند و همزمانی را ممکن سازند، فقط از **thread** تشکیل نشده است و واحدهای دیگری نیز دارد.

* در پایتون **process**, **thread** و **coroutine** مکانیسم‌هایی هستند که امکان همزمانی در پایتون را فراهم می‌کنند.

درس ۴: روش‌های پیاده سازی همزمانی در پایتون

* **multiprocessing** به معنی توانایی اجرای چند **process** (فرایند) به صورت همزمان می‌باشد و زمانی انجام می‌شود که **CPU** چند هسته‌ای است، در **multiprocessing** می‌توانیم موازی سازی واقعی را پیاده سازی کنیم، این روش با **GIL** محدود نمی‌شود، امکان اجرای موازی چند فرایند را فراهم می‌کند که هر کدام مفسر پایتون و فضای حافظه‌ی جدا دارند و برای این کار از مژول **multiprocessing** در پایتون استفاده می‌کنیم.

* **multithreading** به معنی اجرای همزمان چند **thread** با تغییر سریع کنترل **cpu** بین **thread**ها گفته می‌شود، برای از مژول **threading** در پایتون استفاده می‌کنیم، **multithreading** در پایتون در معرض یک محدودیت است به نام **GIL** است، **GIL** مکانیسمی است که تضمین می‌کند فقط یک **thread** اجازه دارد کنترل مفسر پایتون را نگه دارد، یعنی در هر لحظه فقط یک **thread** می‌تواند اجرا شود.

* **asynchronous programming** (برنامه نویسی ناهمگام یا غیر هم روند) یک الگوی برنامه نویسی است که به ما این امکان را می‌دهد که یک کد بنویسیم که بتواند چندین عملیات را همزمان انجام دهد، برای **asynchronous programming** از مژول **asyncio** در پایتون استفاده می‌کنیم، در این روش از **event loop**, **coroutine**ها و عملیات ورودی خروجی **non-blocking** برای رسیدن به همزمانی استفاده می‌شود، **non-blocking** در یک **thread** می‌تواند اجرای چند کار مختلف را مدیریت کند و برنامه نویسی همزمان را ممکن سازد.

* از **asynchronous programming** معمولاً زمانی استفاده می‌شود که یک سری وظایف ورودی و خروجی (I/O) داریم، مانند درخواست‌های شبکه، عملیات فایل، تعامل با پایگاه داده‌ها و ...، این روش کمک می‌کند که به طور کارآمد چند عملیات I/O را به طور همزمان مدیریت کنیم و استفاده از منابع بهینه‌تر انجام شود و پاسخگویی نیز سریع‌تر انجام شود، برای برنامه‌های **real-time** (رویداد محور) مانند سوکت‌های وب و ... استفاده می‌شود.

* از **multiprocessing** معمولاً برای کارهایی استفاده می‌شود که به **CPU** وابستگی دارند، برای کارهایی که می‌توانیم به صورت موازی آن‌ها را اجرا کنیم مانند محاسبات عددی فشرده، یادگیری ماشین، پردازش تصویر، رمز نگاری ویدیو و کارهای مستقلی که نیاز به حافظه مشترک یا اشتراک داده ندارند.

* از **multithreading** معمولاً زمانی استفاده می‌شود که ترکیبی از وظایفی که هم به ورودی و خروجی (I/O) مربوط‌اند و هم به قدرت پردازشی **CPU**، برای برنامه‌های رابط کاربری گرافیکی مناسب‌تر است.

درس ۵: ایجاد اولین برنامه multithreading

- * به thread اجرایی اولیه که با شروع برنامه پایتون ایجاد می‌شود، main-thread گفته می‌شود.
- * برنامه را اجرا می‌کند و وظیفه‌ی مدیریت thread های جدید را دارد.
- مثال: در برنامه زیر دستورات خط به خط و به ترتیب اجرا می‌شوند.

```
import time
print('Main Thread Started')

def my_func(n):
    print(f'function {n} started!')
    time.sleep(3)
    print(f'function {n} finished!')

my_func(1)
my_func(2)
my_func(3)

print('Main Thread Finished')
```

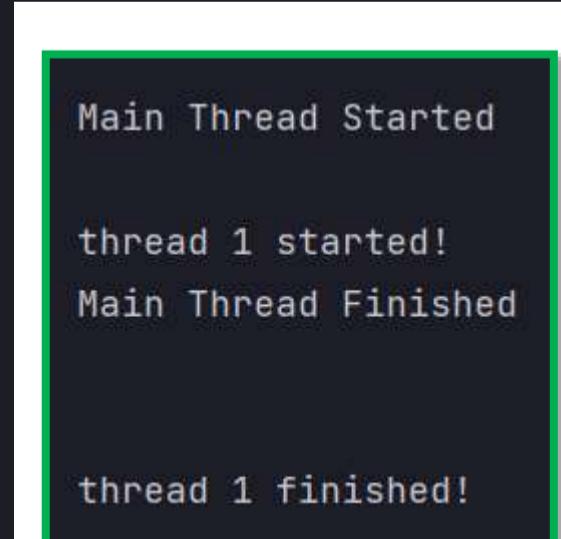
- * با استفاده از thread می‌توانیم این توابع را همزمان اجرا کنیم.
- * برای استفاده از threading باید ماژول thread را وارد کنیم.
- * برای ایجاد thread باید یک شی از کلاس Thread بسازیم و به آرگومان target آن تابعی را که می‌خواهیم اجرا کند را بدهیم و اگر تابع آرگومانی دارد باید در یک لیست به args ارسال شوند.
- * برای اجرای thread باید از متد start استفاده کنیم، بعد از start بقیه دستورات همزمان اجرا می‌شوند و منتظر نمی‌ماند تا دستورات thread به پایان برسند.

```
import time
import threading
print('\nMain Thread Started')

def th_func(n):
    print(f'\nthread {n} started!')
    time.sleep(3)
    print(f'\nthread {n} finished!')

thread = threading.Thread(target=th_func, args=[1])
thread.start()

print('\nMain Thread Finished')
```



* اگر بخواهیم `join` همزمان با `thread` اجرا نشود و منتظر بماند تا `main-thread` به پایان برسد می‌توانیم از متده استفاده کنیم.

```
import time
import threading
print('\nMain Thread Started')

def th_func(n):
    print(f'\nthread {n} started!')
    time.sleep(3)
    print(f'\nthread {n} finished!')

thread = threading.Thread(target=th_func, args=[1])
thread.start()
thread.join()

print('\nMain Thread Finished')
```

```
Main Thread Started
thread 1 started!
thread 1 finished!
Main Thread Finished
```

- * اگر بخواهیم چند **thread** همزمان با یک تابع هدف ایجاد کنیم می‌توانیم در یک لیست با استفاده از **for** آنها را ایجاد کنیم.
- * چون این **thread** ها همزمان اجرا می‌شوند، به پایان رسیدن آنها هیچ ترتیبی ندارد.

```
import time
import threading
print('\nMain Thread Started')

def th_func(n):
    print(f'\nthread {n} started!')
    time.sleep(3)
    print(f'\nthread {n} finished!')

threads = [threading.Thread(target=th_func, args=[i]) for i in range(3)]
for th in threads:
    th.start()

print('\nMain Thread Finished')
```

```
Main Thread Started

thread 0 started!

thread 1 started!

thread 2 started!
Main Thread Finished

thread 1 finished!

thread 2 finished!
thread 0 finished!
```

درس ۶: مشکل دسترسی به حافظه multithreading

* هیچ تضمینی نیست که **thread** ها در همه‌ی سیستم عامل‌ها و همه‌ی ورژن‌های پایتون درست اجرا شوند و احتمال دارد با مشکل مواجه شویم.

مثال: برنامه زیر در نسخه‌های مختلف پایتون نتیجه متفاوتی میدهد و لزوماً درست اجرا نمی‌شود.

```
from threading import Thread

thread_visits = 0

def visit_counter():
    global thread_visits
    for i in range(1_000_000):
        value = thread_visits
        thread_visits = value + 1

count = 2
threads = [Thread(target=visit_counter) for _ in range(count)]

for thread in threads:
    thread.start()

for thread in threads:
    thread.join()

print(f'{count}, {thread_visits}')
```

```
count=2, thread_visits=2000000
8:1  CRLF  UTF-8  4 spaces  Python 3.11 (pythonProject)
```

```
thread_count=2, thread_visits=1203264
python 3.8
```

* همانطور که می‌دانیم **thread** ها از حافظه مشترک استفاده می‌کنند و وقتی چند **thread** به طور همزمان بخواهند به آن دسترسی یابند، دچار این چنین مشکلاتی می‌شویم که به این مشکل **race condition** یا **race hazard** گفته می‌شود.

* برای حل این مشکل می‌توانیم از **Lock** استفاده کنیم، که در هنگام استفاده یک **thread** از حافظه، اجازه دسترسی سایر **thread**‌ها به حافظه را نمی‌دهد و حافظه را قفل می‌کند.

* عملکرد برنامه با **Lock** کاهش می‌یابد اما دقیق‌تر افزایش می‌یابد.

```
from threading import Thread, Lock

thread_visits = 0
thread_visits_lock = Lock()

def visit_counter():
    global thread_visits
    for i in range(1_000_000):
        with thread_visits_lock:
            thread_visits += 1

count = 2
threads = [Thread(target=visit_counter) for _ in range(count)]

for thread in threads:
    thread.start()

for thread in threads:
    thread.join()

print(f'{count=}, {thread_visits=}')
```

درس ۷: مثال کاربردی با multithreading

* استفاده از **threading** باعث می‌شود که برنامه ما خیلی سریع‌تر و بهینه‌تر اجرا شود.

مثال: یک برنامه نوشتیم که **url** چند سایت را می‌گیریم و **length** محتوای صفحه **html** آن را نمایش می‌دهیم

بدون استفاده از **threading**

```
import requests
import time

def fetch_url(url):
    response = requests.get(url)
    print(f'URL: {url}, length: {len(response.text)}')

start_time = time.time() # زمان شروع اجرا

urls = [
    'https://www.digikala.com/',
    'https://snapp.ir/',
    'https://www.aparat.com/',
    'https://divar.ir/',
    'https://shaparak.ir/',
    'https://namnak.com/',
    'https://www.digikala.com/',
    'https://snapp.ir/',
    'https://www.aparat.com/',
    'https://divar.ir/',
    'https://shaparak.ir/',
    'https://namnak.com/']

for url in urls:
    result = fetch_url(url)

end_time = time.time() # زمان پایان اجرا
execution_time = end_time - start_time # محاسبه مدت زمان اجرا به صورت دقیق
print(f'run time without thread: {execution_time} s')
```

```
URL: https://www.digikala.com/, length: 7844
URL: https://snapp.ir/, length: 444016
URL: https://www.aparat.com/, length: 31165
URL: https://divar.ir/, length: 265111
URL: https://shaparak.ir/, length: 70497
URL: https://namnak.com/, length: 717
run time without thread: 19.146974086761475 s
```

* بدون استفاده از **threading** اجرای این برنامه 19 ثانیه طول کشید.

```

import requests
import threading
import time

def fetch_url(url):
    response = requests.get(url)
    print(f'URL: {url}, length: {len(response.text)}')

start_time = time.time() # زمان شروع اجرا

urls = [
    'https://www.digikala.com/',
    'https://snapp.ir/',
    'https://www.aparat.com/',
    'https://divar.ir/',
    'https://shaparak.ir/',
    'https://namnak.com/',
    'https://www.digikala.com/',
    'https://snapp.ir/',
    'https://www.aparat.com/',
    'https://divar.ir/',
    'https://shaparak.ir/',
    'https://namnak.com/',

]

threads = []
for url in urls:
    thread = threading.Thread(target=fetch_url, args=(url,))
    threads.append(thread)
    thread.start()

end_time = time.time() # زمان پایان اجرا
execution_time = end_time - start_time # محاسبه مدت زمان اجرا به صورت دقیق
print(f'run time with thread: {execution_time} s')

```

```

run time with thread: 0.005993366241455078 s
URL: https://namnak.com/, length: 717
URL: https://www.digikala.com/, length: 7844
URL: https://namnak.com/, length: 717
URL: https://www.digikala.com/, length: 7844
URL: https://www.aparat.com/, length: 31165

```

* با استفاده از **threading** اجرای این برنامه کمتر از 1 ثانیه طول کشید و همانطور که مشاهده می‌کنیم استفاده از همزمانی در مواردی که نیاز است چندین تسک انجام شود، بهتر است و برنامه بهینه‌تر اجرا شود.