



به سادگی پایتون

پویا اقبالی

[pooya.eghbamin@yandex.com](mailto:pooya.eghbamin@yandex.com)

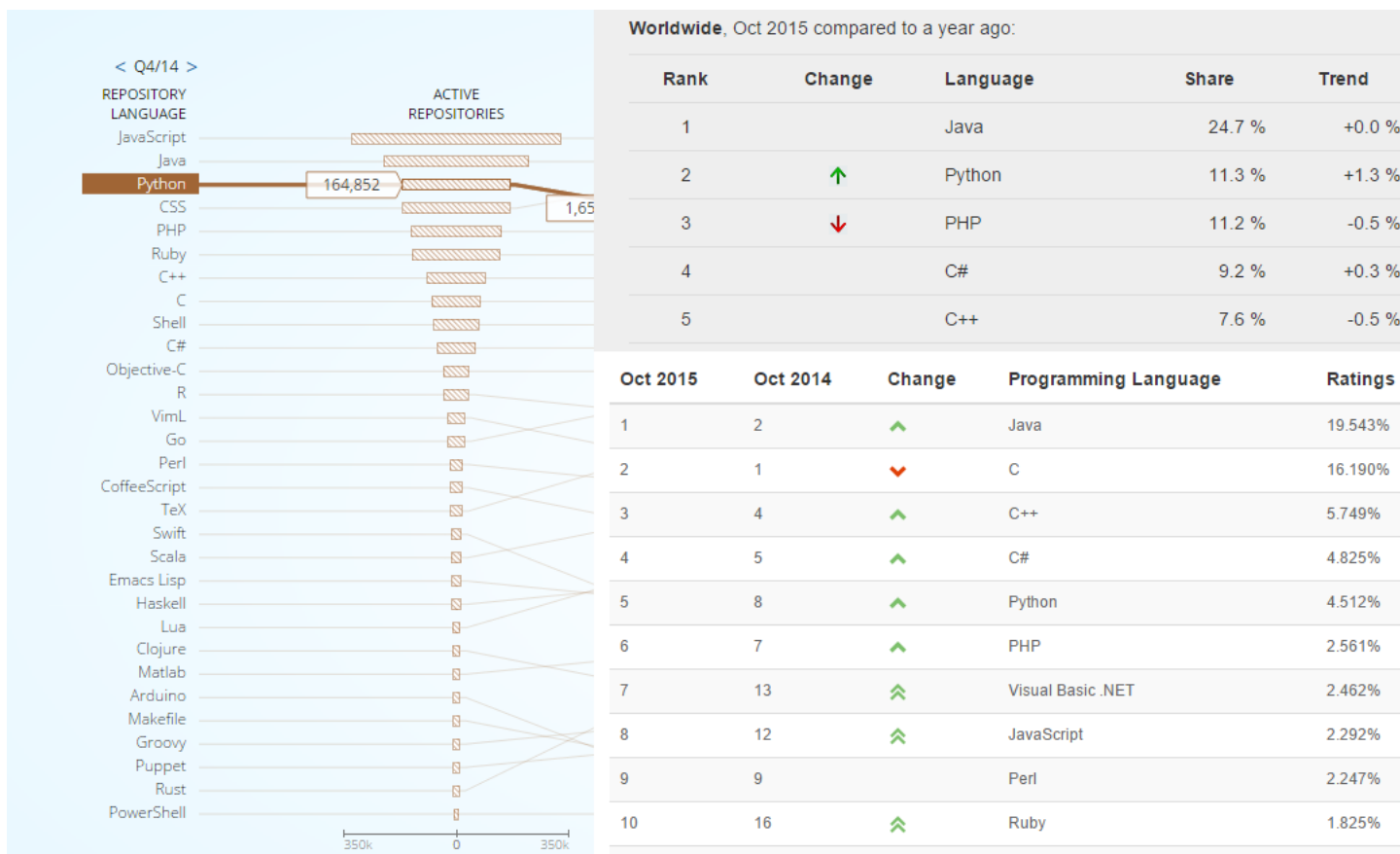
آموزش پایتون به زبان ساده



این کتاب برای آموزش پایتون 3 به فارسی زبانان نوشته شده است. مخاطب کتاب، کسانی هستند که تا به حال برنامه نویسی نکرده اند، در این کار مبتدی اند، و یا اینکه برنامه نویس هستند و می خواهند به تازگی پایتون را نیز یاد بگیرند. در نگارش این کتاب سعی شده است تا از به کار بردن اصطلاحات پیچیده و تخصصی برنامه نویسی و علوم رایانه ای خودداری شود، اگر در جایی از کتاب با اصطلاح و یا مفهومی روبرو شدید که معنی آن را نمی دانستید، مهم نیست، به آن توجه نکنید و از آن بگذرید، هیچکدام از ما زمانی حتی بلد نبودیم چطور غذا بخوریم، یاد گرفتید، مفاهیمی که متوجه نشدید را هم بعدا یاد خواهید گرفت. نکته آخر اینکه، نترسید. خرابکاری نمی کنید اگر همه مثالها را بنویسید و اجرا کنید یا چیزهای مختلف را تایپ کنید و نتیجه شان را ببینید، حتی اگر خرابکاری کنید، اشکالی ندارد، از خرابکاری نترسید، بدون خرابکاری هیچ چیز یاد نخواهید گرفت، چه برنامه نویسی باشد چه هر چیز دیگر. هر جا مشکلی داشتید، گوگل دوست شماست، می توانید سوالهای برنامه نویسی خودتان را جستجو کنید.

## پایتون؟

پایتون سومین زبان محبوب GitHub، دومین زبان محبوب بر اساس جستجوهای انجام شده در گوگل و پنجمین زبان محبوب در رتبه بندی TIOBE است. (اکتبر 2015)



پایتون در شرکت‌های بزرگی مانند گوگل، یاهو، ناسا و CERN و بسیاری دیگر استفاده می شود و تقریباً می توان در هر زمینه ای از جمله ساخت وبسایت (نمونه ها: dropbox, netflix, discuss, pinterest, reddit, youtube, quora, instagram)، ساخت بازی (نمونه ها: Battlefield 2, Star Trek Bridge Commander, Civilization, Eve Online)، گرافیک (نمونه ها: در شرکت والت دیزنی، Industrial Light & Magic، Blender 3D)، هوش مصنوعی، محاسبات علمی و مهندسی و بسیاری دیگر از آن استفاده کرد.

پایتون سال 1991 توسط Guido Van Rossum هلندی طراحی و اختراع شد. فلسفه ساخت پایتون ایجاد زبانی با خوانایی بالا، کوتاه و پر

بازده است.

## چرا پایتون؟

استفاده از پایتون به قدری آسان است که حتی بچه های دبستانی میتوانند آن را یاد بگیرند و شروع به برنامه نویسی کنند. وقتی پایتون را نصب می کنید، باطری ها درون محفظه اند! لازم نیست کار دیگری بکنید، بسیاری از ابزارهایی که یک برنامه نویس ممکن است به آنها نیاز داشته باشد از قبل در پایتون جاسازی شده اند.

کد پایتون روان و خواندن آن آسان است، این باعث می شود که هزینه های تولید و نگهداری نرم افزار به شدت پایین بیاید و در کمترین مدت بالاترین بازده ممکن را ایجاد کرد. پایتون دارای کتابخانه های بسیار زیاد است که شما را از نوشتن و ایجاد ابزارهای خودتان در هر زمینه ای بی نیاز می کند، می توانید کتابخانه های موجود را به برنامه خود اضافه کنید و به صورت رایگان از آنها استفاده کنید، به جای اینکه وقت بگذارید و یکی مثل آن را خودتان بنویسد، پایتون همه ابزارها را در اختیار شما قرار می دهد.

پایتون یک زبان سطح بالاست. نگران چیزهای سطح پایینتر مثل کنترل حافظه یا نوع متغیرها نباشید، همه اینها به طور خودکار انجام می شود.

## پایتون برای چه کسی مناسب نیست؟

هیچ زبانی کامل نیست. نمی توان زبانهای برنامه نویسی را با هم مقایسه کرد، هر زبان ویژگی هایی دارد که آن را برای انجام کاری مناسب می کند و ویژگی هایی که باعث می شود آن زبان برای انجام کاری مناسب نباشد. پایتون برای چه چیز مناسب نیست؟

پایتون برای انجام کارهایی که قدرت پردازشی بالا و سرعت زیاد نیاز دارند مناسب نیست. بهتر است این قسمت برنامه خود را با C بنویسد. برای مثال، برای انجام دادن محاسبات علمی و ریاضی، ما مستقیماً از پایتون استفاده نمی کنیم، بلکه از کتابخانه هایی مانند NumPy و SciPy استفاده میکنیم که به زبان C نوشته شده اند و به ما اجازه می دهند تا با سرعت بالا محاسبات عددی خود را در پایتون انجام دهیم.

با پایتون نمی توان یک سیستم عامل نوشت. محدودیت های زبان انتخابی خود را بشناسید. پایتون برای انجام کارهایی، که شکست آن، جان کسی را به خطر می اندازد، مناسب نیست، حتی C یا ++C و Java هم برای چنین کاری مناسب نیستند. پایتون یک زبان سطح بالاست. نمی توان نوع متغیرها را تعیین کرد یا حافظه را مدیریت کرد، پایتون برای کارهایی که نیاز به اینها دارند مناسب نیست.

## ذن پایتون

- زیبا بهتر از زشت است.
- صریح و روشن بهتر از پنهانکاری است.
- ساده بهتر از پیچیده است
- پیچیده بهتر از بغرنج است
- تخت بهتر از تو در تو است
- پراکنده بهتر از متراکم است
- خوانایی امتیاز است
- موارد استثنا به اندازه ای استثنایی نیستند که قوانین را بشکنند
- با اینکه عملی بودن، خلوص را شکست می دهد
- از خطاها نباید ساکت گذشت

- مگر اینکه به صراحت ساکت شده باشند
- اگر درباره چیزی شک دارید، وسوسه حدس زدن درباره آن را سرکوب کنید
- باید فقط یک راه --و ترجیحا فقط یک راه-- برای انجام یک چیز وجود داشته باشد
- حتی اگر آن راه در ابتدا واضح به نظر نرسد، مگر آنکه یک "هلندی" باشید
- الان بهتر از هیچوقت است
- البته بیشتر اوقات هیچوقت بهتر از \*همین\* الان است
- اگر توضیح یک پیاده سازی سخت است، مطمئنا ایده بدی است
- اگر توضیح یک پیاده سازی آسان است، ممکن است ایده خوبی باشد
- فضای نام یک ایده ی خیلی خیلی فوق العاده است! بیشتر از آن استفاده کنید

## چطور شروع کنیم؟

مرحله اول، نصب پایتون است که می توانید با مراجعه به وبسایت اصلی پایتون به آدرس [python.org](https://python.org) آن را دانلود کنید یا درباره نصب آن روی سیستم عاملی که استفاده می کنید راهنمایی بگیرید. اگر از لینوکس یا یونیکس استفاده می کنید نیازی نیست که به شما یاد بدهم چطور پایتون را نصب کنید، فقط یادتان باشد که آخرین به روزرسانی از نسخه سوم پایتون را نصب کنید.

برای سیستم عامل ویندوز، یا از وبسایت پایتون، مطابق با پردازنده خود بسته نصبی را دانلود کنید، و یا اگر حجم دانلودی برای شما مهم نیست از وبسایت <https://www.continuum.io/downloads> بسته نصبی Anaconda را دریافت کنید.

برای شروع نیازی به Text Editor ندارید و می توانید از IDLE که همراه با پایتون نصب میشود استفاده کنید، که البته بعدها به محیطهای دیگری نیاز پیدا خواهید کرد. روی سیستم عامل ویندوز من از Atom و Notepad++ و در سیستم عامل های BSD و Linux از Vi و Emacs استفاده می کنم، اما در این آموزش از IDLE استفاده خواهیم کرد.

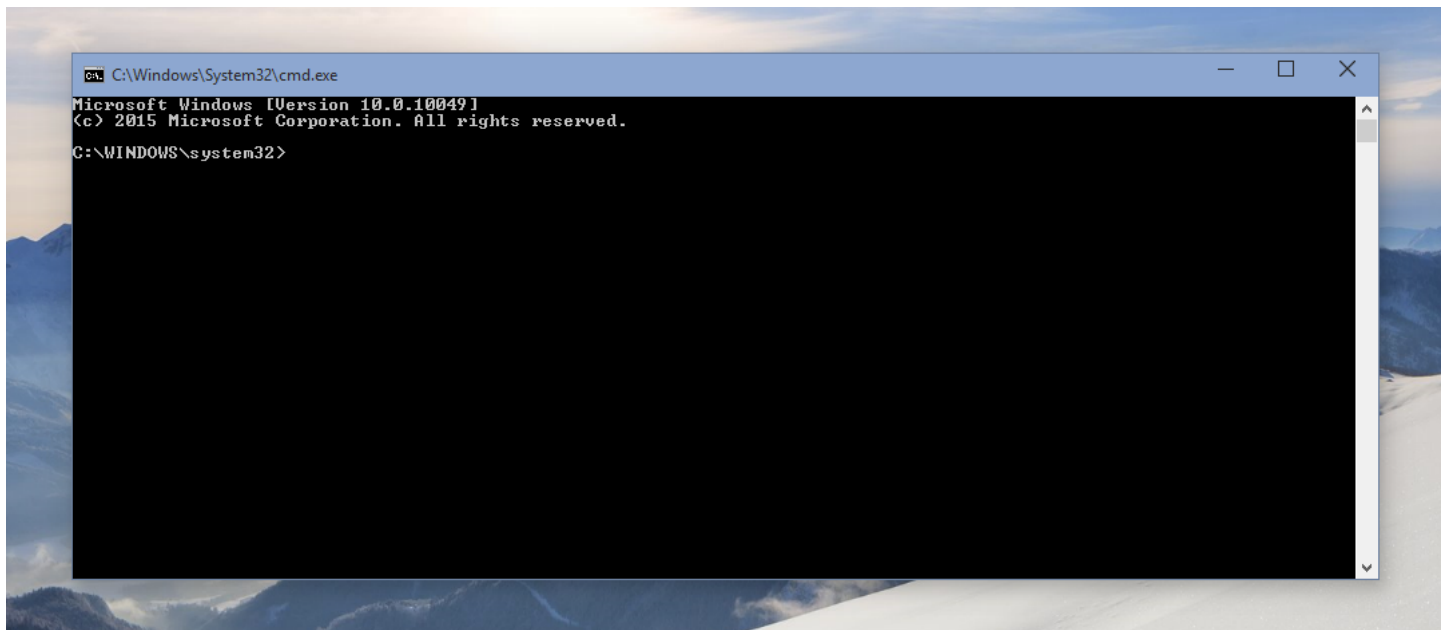
محیطهایی برای استفاده های مهندسی، محاسباتی و علمی پایتون نیز مانند IPython، Spyder و غیره وجود دارند که می توان از آنها استفاده کرد.

## طبق سنت

سنت برنامه نویسی این است که اول از نوشتن برنامه "Hello World" یا "سلام دنیا" شروع کنیم. ما هم با همین مثال شروع خواهیم کرد، ولی قبل از شروع، باید آماده شوید. اگر می خواهید برنامه نویس شوید، بهتر است دست از کلیک کردن بردارید و بیشتر بنویسید. بهتر است که با محیط cmd یا terminal آشنا باشید. اگر از سیستم عامل لینوکس یا یونیکس استفاده می کنید، احتمالا آشنا هستید، پس یک ترمینال باز کنید، یک پوشه درست کنید و یک فایل خالی به نام `hello.py` درست کنید.

ولی اگر از محیط ویندوز استفاده می کنید، فرض می کنم تازه کارید و مراحل انجام کار را توضیح خواهم داد. باور کنید که یاد گرفتن این موضوع، شما را برنامه نویس بهتری خواهد کرد. برای حالا، همان `cmd` ویندوز برایتان کافی است اما وقتی به اندازه کافی یاد گرفتید شاید بخواهید از `git bash` یا `babun` یا `cmdr` استفاده کنید.

به هر حال نحوه انجام کار به این صورت است، ابتدا کلید ویندوز (استارت منو) را روی کیبورد فشار دهید و بنویسید `cmd` و کلید اینتر را فشار بدهید. چنین چیزی را خواهید دید:



که البته متناسب با نسخه سیستم عامل شما ممکن است کمی با این تصویر متفاوت باشد، که تفاوتی نمی کند. بنویسید:

```
cd %HOMEPATH%\Desktop
```

و کلید اینتر را فشار دهید. (از اینجا به بعد دیگر نمی گویم کی باید کلید اینتر را فشار دهید)

این دستور ما را به پوشه دسکتاپ می برد. حالا می خواهیم در دسکتاپ یک پوشه برای آموزش پایتون درست کنیم و به آن پوشه برویم:

```
mkdir python && cd python
```

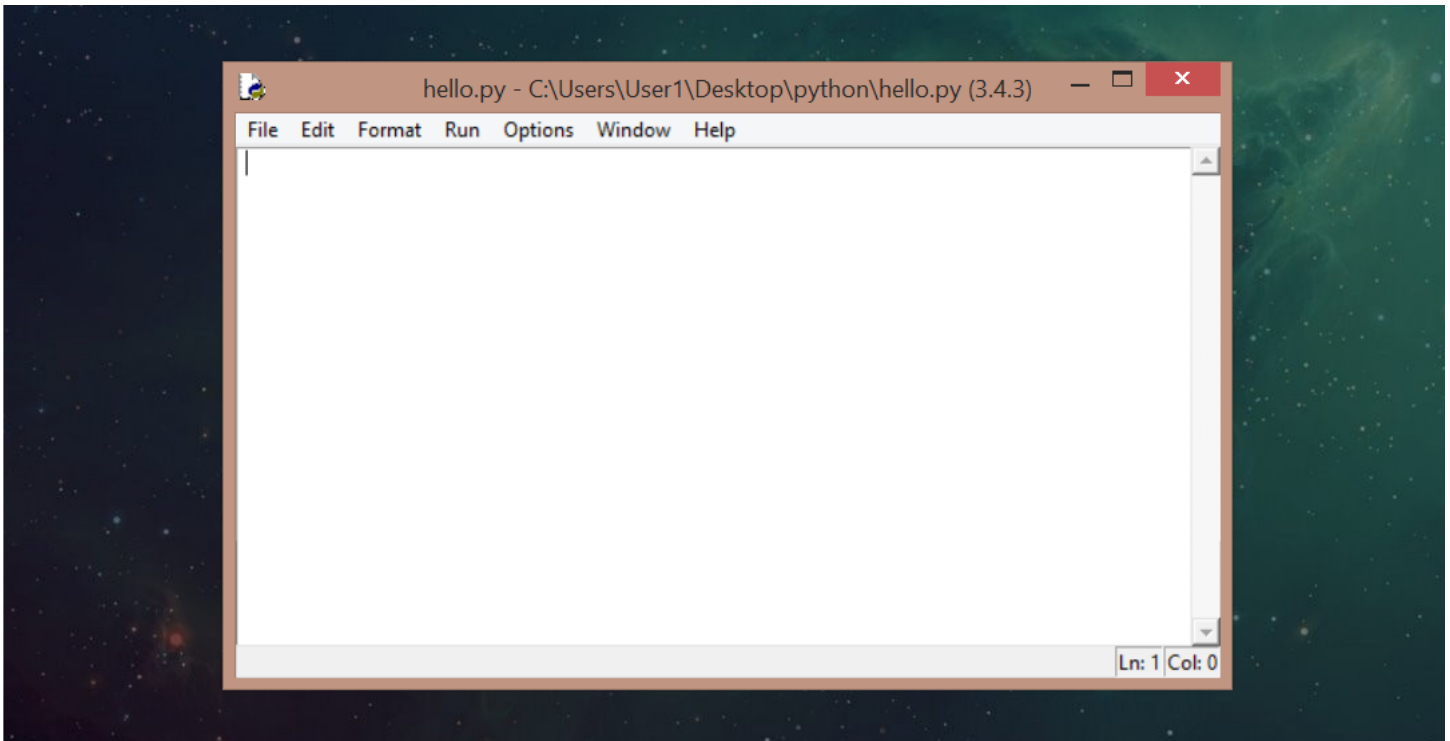
مرحله بعد ایجاد یک فایل خالی است که کد برنامه خود را درون آن قرار بدهیم:

```
type NUL > hello.py
```

این دستور یک فایل خالی به نام hello.py در پوشه python روی دسکتاپ ایجاد می کند که حالا می توانیم کد برنامه را در آن بنویسیم.  
چطور؟ بنویسید:

```
python -m idlelib hello.py
```

چنین چیزی را خواهید دید:



که البته، ممکن است با توجه به توزیع یا نسخه پایتونی که نصب کرده اید با این تصویر متفاوت باشد. حالا اولین برنامه پایتون خودتان را درون این پنجره تایپ کنید:

```
print('سلام دنیا')
```

کلیدهای Ctrl و S را روی کیبورد همزمان فشار دهید تا تغییرات ذخیره شوند، حالا کلید F5 را روی کیبوردتان فشار دهید تا معجزه اتفاق بیفتد! تبریک می گویم، شما اولین برنامه پایتون خود را نوشتید. بعدا به این دستور بر می گردیم و آن را توضیح می دهیم، نگران نباشید.

محیط تعاملی پایتون

بیا باید تغییری در مثال قبلی خود ایجاد کنیم. فایل قبلی را با idle باز کنید:

```
python -m idlelib hello.py
```

و تمام محتویات آن را پاک کنید و به جای آن بنویسید:

```
'سلام دنیا'
```

حالا، همانطور که بهتان یاد دادم، فایل را ذخیره و اجرا کنید. چه اتفاقی می افتد؟ اگر به یاد داشته باشید دفعه قبل عبارت 'سلام دنیا' به شما نمایش داده شد، حالا چطور؟ این دفعه هیچ چیز به شما نمایش داده نشد! بیایید یک چیز دیگر را امتحان کنیم. پنجره های باز شده را ببندید و در پوشه python و در cmd بنویسید:

```
python -m idlelib
```

و بعد در پنجره باز شده تایپ کنید:

```
'سلام دنیا'
```

و اینتر را بزنید. حالا چطور؟ این بار عبارت 'سلام دنیا' به شما نمایش داده خواهد شد! اگر دستور print را همانطور که در بخش قبلی دیدید اینجا بنویسید، باز هم همین نتیجه را خواهید دید، اما چرا اینبار بدون print عبارت را نمایش داد اما از فایل نه؟ دلیلش این است که حالا ما در محیط تعاملی پایتون هستیم، و هر چیزی که بنویسیم مقدار آن به ما نمایش داده می شود، اما در محیط واقعی، در محیط کد، یا در یک فایل یا حلقه، برای نمایش دادن مقدار یک چیز باید از دستور print استفاده کنیم.

علت اینکه شما را با محیط تعاملی پایتون آشنا می کنم این است که دوست ندارم هر دفعه برای یاد گرفتن و نمایش چیزهای پیش پا افتاده از دستور print استفاده کنیم، برای مثال چرا باید نتیجه جمع دو عدد را با دستور print نمایش دهیم، وقتی می توانیم از محیط تعاملی استفاده کنیم؟ محیط تعاملی آخرین نتیجه را به ما نشان می دهد و برای این منظور، یعنی آموزش، مناسب است.

## اعداد و ریاضیات

بهتر است از ساده ترین چیزها شروع کنیم، چیزی که همه ما با آنها آشناییم، اعداد و عملیات ریاضی ساده. در واقع، این موضوع به قدری ساده است، که حتی نیازی به توضیح ندارد، بگذارید اول انواع اعداد را بشناسیم. یک عدد در زبان پایتون به این شکل نوشته می شود (سعی کنید اینها را در محیط تعاملی بنویسید و نتیجه آنها را ببینید!)

```
3
6
9
12
15
```

اعداد اعشاری در پایتون به این شکل نوشته می شوند:

```
0.57721
1.6180339887
2.718
3.14
4.6692
```

که البته، اعداد بین 0 و 1 را می شود اینطور هم نوشت:

```
.12345
.57721
```

اعداد منفی، چه اعشاری یا صحیح را با یک علامت - کنارشان نمایش می دهیم:

```
-1
-1.6180339887
-3
-3.14
-6
```

یک قابلیت جالب دیگر درباره اعداد پایتون وجود دارد که نوشتن اعداد با توان است، مثلا، 0.2 را می توان به صورت 2 ضرب در 10 به توان منفی یک نوشت:

```
2e-1
```

یا مثلا عدد 2 با 10 صفر جلوی آن را می توان اینطوری نوشت:



```
2e10
```

که به معنی 2 ضرب در 10 به توان 10 است. انواع اعداد دیگری مثل اعداد مختلط هم در پایتون وجود دارند که فعلا از آنها رد می شویم. حالا ساده ترین عمل ریاضی را انجام می دهیم، در محیط idle تعاملی بنویسید:

```
1 + 2
```

این دستور مقدار 3 را به شما نمایش می دهد، توجه کنید که اگر قرار بود برنامه را در یک فایل بنویسیم باید می نوشتیم

```
print(1 + 2)
```

تا مقدار 3 نمایش داده شود. برای تفریق دقیقا مثل ریاضیات از علامت - استفاده می کنیم:

```
3 - 2  
2 - 3  
-3 - 2
```

تاکید می کنم که همه این مثالها را اجرا کنید و نتیجه آنها را ببینید، البته بهتر است که قبل از اجرا نتیجه آنها را حدس بزنید. حالا، سراغ ضرب و تقسیم برویم:

```
2/3  
3/2  
3*2  
3*3.14
```

یک نوع تقسیم دیگر هم در پایتون وجود دارد و آن تقسیم صحیح است. خودتان امتحان کنید و با تقسیم معمولی مقایسه کنید:

چند عمل ریاضی دیگر هم در پایتون هست، که مهمترین آنها را بیان می کنم. اولی، علامت توان است:

```
2**3
3.14**3.14
```

و بعدی علامت باقی مانده تقسیم اولی بر دومی:

```
2%3
3%2
```

همینطور برای رند کردن یک عدد داریم:

```
round(3.14)
```

توجه کنید که عملیات ریاضی را می توان به هم زنجیر کرد، یعنی می شود نوشت:

```
round(3.14) + 2 * 3
```

این کد اول 2 را در 3 ضرب می کند و بعد با رند 3.14 جمع می کند، ولی اگر بخواهیم اول عدد رند شده با 2 جمع شده و بعد ضرب در 3 شود باید چه کار کنیم؟ ساده است، می توانیم از پرانتز برای دسته بندی اعداد و عملیات استفاده کنیم:

```
(round(3.14) + 2) * 3
```

نوع دیگری از اعداد در پایتون هستند که به علت اهمیت آنها، بهتر است با آنها هم آشنا شویم. این اعداد، اعداد باینری یا مبنای دو، اعداد

اوکتت یا مبنای 8 و اعداد هگزا یا مبنای 16 هستند که به ترتیب به این شکل نوشته می شوند و خاصیت اعداد قبلی را دارند:

```
0b10
0o10
0x10
```

فکر می کنم کار ما با اعداد تمام شده است.

## متغیرها و کامنتها

تا الان، یک سری اعداد را نوشتیم و برای خودمان مسخره بازی در آوردیم، ولی این اصلا به درد نمی خورد! منظورم این است که، خوب، مثلا دو تا عدد را با هم جمع کردیم و نتیجه اش را دیدیم، خوب که چی؟ اگر می توانستیم نتیجه این جمع را یک جایی نگه داریم و بعدا دوباره آنها را ببینیم و به آنها دسترسی داشته باشیم، بهتر بود. متغیرها به همین منظور وجود دارند.

متغیر چیست؟ متغیر به زبان ساده نتیجه اجرای یک دستور، یا خود دستور است که به آن یک اسم داده ایم. متغیر یک اسم است که به محلی در حافظه، که حاوی مقداری یا چیزی که برای ما دارای اهمیت است اشاره می کند. یعنی چه؟ در idle امتحان کنید، بنویسید:

```
x = 2
```

حالا بنویسید:

```
x
```

مقدار x نمایش داده خواهد شد. در این مثال x یک متغیر است. می توانیم یک توضیح به این کد اضافه کنیم، کامنتها در زبان پایتون با # نمایش داده می شوند. در یک خط هر چیزی که بعد از یک # نوشته شود یک کامنت محسوب شده و اجرا نمی شود:

```
# این یک کامنت است
x = 2 # این هم یک کامنت است
```

سعی کنید از کامنتها استفاده کنید تا اگر کس دیگری کد شما را می خواند بفهمد که چه چیز نوشته اید و چه کاری می خواستید انجام دهید، ولی زیاده روی نکنید، انقدری توضیح بدهید که بشود فهمید قصد چه کاری را داشته اید، مردم از طرز کار ذهن شما و پیچ و خم های

مغزتان خبر ندارند و کله شان متفاوت با شما کار می کند، منطقشان همیشه با شما یکی نیست و همیشه با دیدن کد شما منظورتان را نمی فهمند.

```
این متغیر 4 تا بیشتر از مقدار ایکس است و این یک کامنت احمقانه است # y = x + 4
```

یکی دیگر از فواید کامنت زمانی هست که شما بعد از مدت زیادی به کد خودتان برمیگردید و فراموش می کنید که منظورتان از کدی که نوشته اید چه بوده است. مطمئن باشید این اتفاق برای شما می افتد، چون اولاً شما همیشه در حال پیشرفت هستید و حالا دیگر کد برای یک منظور را مثل چند وقت قبل نمی نویسید، پس کد قبلی به نظرتان احمقانه و عجیب و غریب می رسد، دیگری اینکه در یک پروژه چند هزار خطی، به یاد داشتن همه خطوط، غیر ممکن نه، دشوار است.

نکات دیگری که باید رعایت کنید، تا برنامه های بهتری بنویسد، ذن پایتون و فلسفه تمیز نویسی پایتون است، تمیز بنویسد و سعی کنید هر خط کد بیشتر از 80 کاراکتر نشود.

در مورد متغیرها این موضوع را به یاد داشته باشید که، متغیری که مقدار آن 3.14 باشد هیچ فرقی با خود 3.14 ندارد و می توان مثل همان عدد (یا هر چیز دیگری که متغیر به آن اشاره می کند) با آن رفتار کرد، مثلاً:

```
pi = 3.14
round(pi) + 2
pi + .86
2pi = pi + pi
```

امتحان کنید و ببینید چه نامهایی برای یک متغیر غیر مجاز هستند.

رشته ها، یا همان، نوشته ها!

حتماً اولین برنامه ای که نوشتیم را به یاد دارید که عبارت "سلام دنیا" را نمایش دادیم. در آن مثال، "سلام دنیا" یک رشته بود. رشته ها را میشود به یکی از دو صورت زیر نمایش داد:

```
"hello world"
'hello world'
```

یا برای رشته های چند خطی:

```
"""Line 1
Line 2
Line 3"""

'''Line 1
Line 2
Line 3'''
```

حالا سعی کنید یک رشته بنویسید که حاوی کاراکتر ' باشد. (به هر کدام از حروف یک رشته، یک کاراکتر می گوئیم)

رشته ها را می توان به هم چسباند:

```
"hello" "world"
```

و می توان با هم جمع کرد:

```
"hello"+"world"
x = "hello"
x + "world"
```

حتی می شود یک عدد را در یک رشته ضرب کرد:

```
3 * "a"
"a" * 3
```

رشته ها یک خاصیت جالب دارند که آن دسترسی به حروف تشکیل دهنده ی آنها با دانستن مکان آنها در یک رشته است. بیااید امتحان کنیم. در دنیای کامپیوتر، ما شمارش را از صفر شروع می کنیم، پس برای گرفتن اولین حرف یا کاراکتر یک رشته باید کاراکتر شماره 0 را بگیریم، که به این صورت انجام می شود:

```
"hello"[0]
x = "hello"
x[0]
```

و همینطور به ترتیب برای دسترسی به کاراکترهای دیگر:

```
x[1] # کاراکتر دوم
x[2] # کاراکتر سوم
x[3] # کاراکتر چهارم
#... و به همین ترتیب تا آخرین حرف
```

به هر کدام از این اعداد اصطلاحاً یک index گفته می شود که به معنی شاخص یا نمایه است و در برنامه نویسی معمولاً با حرف i نمایش داده می شود:

```
x = "hello"
i = 0
x[i]
```

حالا امتحان کنید اگر شاخص یا ایندکس بزرگتر از تعداد حروف یک رشته را تقاضا کنید چه اتفاقی می افتد؟ سعی کنید با اعداد باینری، اوکتت و هگزا هم امتحان کنید.

در زبان پایتون، علاوه بر ایندکس مثبت، می شود از ایندکس منفی هم استفاده کرد، مثلاً ایندکس -1 حرف آخر یک رشته را نمایش می دهد، خودتان امتحان کنید:

```
x = "hello"
x[-1]
x[-2]
x[-3]
x[-4]
x[-5]
x[-0] # چه اتفاقی می افتد؟
```

حالا اگر بخواهیم حروف اول تا سوم (یعنی حرف اول و دوم) یک رشته را در یک متغیر بگذاریم، باید چه کار کنیم؟ خوب یک راهش این است که:

```
x = "hello"
y = x[0] + x[1]
```

ولی راه بهتری هم وجود دارد. در پایتون می شود رشته ها را برش زد (slice)، مثلاً، برای گرفتن 2 حرف اول یک رشته می شود اینطور عمل

کرد:

```
x = "hello"  
x[:2]
```

یا برای گرفتن همه حروف غیر از دو حرف اول:

```
x = "hello"  
x[2:]
```

جالب اینکه اینجا هم می شود از ایندکس منفی استفاده کرد، مثلاً برای گرفتن دو حرف آخر می شود نوشت:

```
x = "hello"  
x[-2:]
```

حالا اگر بخواهیم مثلاً حرف دوم و سوم را بگیریم باید چه کار کنیم؟ ساده است:

```
x = "hello"  
x[1:3]
```

از خودتان بپرسید و مرور کنید که چرا از اعداد 1 و 3 استفاده کردیم؟ یادتان باشد که شمارش را از 0 شروع می کنیم، پس معنی این عبارت این است که حرف دوم یا ایندکس شماره 1 تا حرف چهارم یا ایندکس شماره 3 را از متغیر x برش بزن، و یادتان باشد وقتی می گوییم تا حرف چهارم، یعنی حرف چهارم را نمی خواهیم!

اینجا هم می شود از ایندکس منفی استفاده کرد، مثلاً برای برش زدن و گرفتن همه حروف به جز حرف اول و آخر می نویسیم:

```
x[1:-1]
```

مبحث ایندکسها و شاخصها یکی از مهمترین مباحث برنامه نویسی است، باید یاد بگیرید که از صفر شمارش کنید. آنقدر با رشته ها و

ایندکسهای مختلف این کار را تمرین کنید تا بدون نیاز به فکر کردن قادر به انجام آن باشید و تا این کار را یاد نگرفتید از این بخش کتاب جلوتر نروید.

## لیستها

لیست، یعنی چه؟ مثلاً، یک لیست خرید را در نظر بگیرید که نام چیزایی که نیاز دارید را در آن یادداشت کرده اید. یا یک لیست از اعداد، مثلاً نمرات دانش آموزان، یا یک لیست از اعداد و کلمات، یا هر چیز دیگر، در پایتون می توان از این لیستها ساخت. شکل تعریف یک لیست در پایتون بسیار ساده است. یک لیست خالی در پایتون به این شکل نوشته می شود:

```
[]  
x = []
```

یک لیست که یک عدد را در خود داشته باشد، به شکل زیر نوشته می شود:

```
[1]  
x = [1]
```

لیستی که شامل چند عدد باشد به اینصورت نوشته می شود:

```
[1, 2, 3]  
x = [1, 2, 3]
```

و می شود هر چیزی را در این لیست قرار داد:

```
[1, 0x10, 2e10, 3.4, "hello", -1]  
x = [1, 0x10, 2e10, 3.4, "hello", -1]  
y = [x, 5, 3, "python"]
```

لیستها، مانند رشته ها، قابلیت ایندکس را دارند و می شود مثل رشته ها به اعضای آنها دسترسی داشت:



```
x = [1, 0x10, 2e10, 3.4, "hello", -1]
y = [x, 5, 3, "python"]
x[0]
x[-1]
x[1:-1]
y[0]
y[-1]
y[0][1]
y[0][1:-1]
```

همینطور اینکه، مانند رشته ها، لیست ها را هم می توان با هم جمع کرد یا اینکه یک عدد را در آنها ضرب کرد:

```
x = [1, 0x10, 2e10, 3.4, "hello", -1]
y = [x, 5, 3, "python"]
x + y
2 * y
```

یکی از چیزهایی که در لیستها با رشته ها متفاوت است، این است که می شود اعضای لیست را تغییر داد، اما این کار در مورد رشته ها امکان ناپذیر است:

```
x = [1, 2, 3]
x[0] = 4
x
y = "hello"
y[0] = "y" # Error
```

همینطور اینکه می شود یک اسلایس یا یک برش از یک لیست را تغییر داد:

```
x = [1, 2, 3]
x[0:2] = [2, 3, 4]
x
```

همینطور می شود یک اسلایس از یک لیست را به این شکل حذف کرد:

```
x = [1, 2, 3, 4, 5]
x[0:2] = []
x
```

برای اضافه کردن چیزی به یک لیست می توان اینطور عمل کرد:

```
x = [1, 2, 3]
x.append(4)
x = x + [5]
```

دستور append یکی به آخر یک لیست اضافه می کند. کلمه append یعنی الحاق کردن، اضافه کردن یا آویختن. حالا اگر بخواهیم از آخر یک لیست چیزی را حذف کنیم:

```
x = [1, 2, 3]
x.pop()
```

دستور pop یکی از آخر لیست حذف کرده و آن را بر می گرداند. یعنی چه؟ یعنی می شود مقدار آن را در یک متغیر قرار داد یا از آن استفاده کرد:

```
x = [1, 2, 3]
y = x.pop()
y
x.pop() + 2
```

کلمه pop یعنی ترکاندن یا پراندن، مثلا، وقتی یک بادکنک می ترکد و خالی می شود، در زبان انگلیسی به این ترکیدن pop می گویند. از دستور pop می شود برای حذف یک ایندکس خاص هم استفاده کرد، مثلا برای حذف دومین عضو یک لیست می نویسیم:

```
x = [1, 2, 3]
x.pop(1)
```

در زبان پایتون، دستوری وجود دارد به نام len که مخفف کلمه length به معنی طول است. از این دستور می شود برای به دست آوردن طول یک رشته یا یک لیست استفاده کرد:

```
len([1, 2, 3])
len("hello")
```

```
x = len([1, 2, 3])
x * 5 + len("hello")
y = [1, 2, 3, "hello", [2, 3]]
len(y)
len(y[-1])
```

## اولین قدم برای برنامه نویسی

اولین قدم برای برنامه نویسی؟! این همه کار انجام دادیم! تازه اولین قدم؟! بله، تازه اولین قدم برنامه نویسی. کارهایی که انجام دادیم، به هیچ دردی نمی خورند. هیچ منطقی ندارند. که چی بشود چهار تا عدد و رشته را در یک لیست بگذاریم و آنها را دوباره پس بگیریم و نمایش بدهیم؟! منظورم این است که خوب این به تنهایی چه کاربرد عملی دارد؟ هیچ. تا به حال برنامه های ما بدون منطق بودند، اما حالا می خواهیم با دستورات کنترلی پایتون آشنا شویم و به برنامه هایی که می نویسیم منطق اضافه کنیم.

اولین دستوری که می خواهیم با آن آشنا شویم، while است، که معنی آن می شود "تا وقتی که". در واقع این یک دستور نیست، ولی شما فعلا به آن بگویید دستور! برای کار با while باید با چند علامت دیگر آشنا شوید. به زبان فارسی، وقتی می خواهیم از جمله ای شامل "تا وقتی که" استفاده کنیم، شرطی را تعیین می کنیم که ممکن است درست یا غلط باشد، تا وقتی که این شرط درست باشد کاری که می خواهیم انجام می شود. حالا، در زبان پایتون این درستی و یا غلط بودن یک چیز را چطور نمایش می دهیم؟ ساده است. کلمات انگلیسی True و False در زبان انگلیسی به ترتیب به معنی درست و اشتباه هستند، در زبان پایتون هم عینا از همین کلمات برای نمایش درستی یا نادرستی یک چیز استفاده می کنیم.

```
True
False
```

همینطور اینکه، می شود از کلیدواژه های and به معنی "و" و or به معنی "یا" و not به معنی "نفی" برای ترکیب مقادیر بولی استفاده کرد، برای مثال چیزی که صحیح نباشد، نادرست است و چیزی که نا درست نباشد، درست است:

```
not True
not False
```

ترکیب درست و نادرست، و همینطور ترکیب نادرست و درست، نادرست است، اما ترکیب دو مقدار درست، صحیح است:

```
True and False
False and True
False and False
True and True
```

و اما در نهایت، درست یا نادرست، نادرست یا درست، صحیح یا صحیح، درست هستند، اما نادرست یا نادرست نه:

```
True or False
False or True
False or False
True or True
```

همه ما با علامتهای ریاضی کوچکتر و بزرگتر آشنا هستیم! بیایید امتحان کنیم:

```
2 < 5
3 > 4

2 < 5 and 3 > 4
2 < 5 and not 3 > 4
2 < 5 and not 3 > 4 and False
2 < 5 or 3 > 4 and True

x = 2 < 5
x and 3 > 2
```

اگر اینها را امتحان کنید میبینید که اولی مقدار True و دومی مقدار False را برگرداند. فعلا با اینکه نمایش دادن با برگرداندن چه فرقی می کند ذهن خودتان را مشغول نکنید. علائم دیگری نیز برای مقایسه وجود دارند، مثلا علامت کوچکتر یا مساوی و بزرگتر یا مساوی:

```
2 <= 5
3 >= 3
```

برای اینکه بفهمیم دو مقدار با هم برابرند یا نه به این صورت عمل می کنیم:

```
2 == 3
2 == 2
```

و برای اینکه عکس این موضوع را بررسی کنیم:

```
2 != 3
2 != 2
```

به عنوان یک تمرین اینها را با لیست ها و رشته ها و همینطور ایندکسها و اسلایسها و طول رشته ها و اعداد امتحان کنید! خوب، حالا نوبت آن است که محیط تعاملی پایتون را کنار بگذاریم و وارد دنیای واقعی برنامه نویسی بشویم، پس دوباره cmd را باز کنید و به پوشه پایتون رو دسکتاپ بروید و یک فایل جدید به اسم "controlflow.py" درست کنید و آن را با idle باز کنید! سخت شد، نه؟ دوباره توضیح نمی دهم که چطور باید این کار را انجام بدهید، یک بار قبلا گفته شده که می توانید به آن مراجعه کنید.

حالا، می خواهیم برنامه ای را بنویسیم، که عددی را در یک متغیر می گذارد و تا وقتی که مقدار متغیر از 10 کوچکتر است، یکی به آن اضافه می کند و مقدار آن را نمایش می دهد. پس بیایید اول متغیر خودمان را مشخص کنیم:

```
x = 0
```

حالا شرط خود را می نویسیم:

```
while x < 10:
```

یعنی، تا وقتی که x از 10 کوچکتر است. حالا، با یک Tab یا 4 تا space (فضای خالی) یک فاصله در خط بعد از while ایجاد کنید تا کد شما به این شکل در بیاید:

```
x = 0
while x < 10:
    # چهار تا space
```

در زبان پایتون هر گروه کد با یک فاصله مشخص می شود، که بعدا بیشتر توضیح خواهیم داد. کد خود را به شکل زیر تغییر بدهید:

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

کلید f5 را بفشارید تا اجرا شود. چه اتفاقی می افتد؟ در کدی که نوشتید، قسمت:

```
x = x + 1
```

یکی به x اضافه می کند و قسمت

```
print(x)
```

مقدار ایکس را نمایش می دهد. به عنوان تمرین برنامه ای را بنویسید که یک متغیر را برابر یک لیست قرار می دهد و تا وقتی که اولین عضو این لیست از 25 کوچکتر است 5 تا به آن اضافه می کند و مقدار آن را نمایش می دهد.

خوب، حالا نوبت آن است که بگویم برگرداندن با نمایش دادن چه فرقی می کند؟ تا الان با محیط تعاملی پایتون کار کردید و در این محیط تفاوت نمایش دادن با برگرداندن برای شما مشخص نبود، چرا که همه مقدارهای برگردانده شده، نمایش داده می شدند! کد خود را به این شکل تغییر دهید و اجرا کنید:

```
x = 0
while x < 10:
    x = x + 1
    x
```

خواهید دید که اینبار دیگر مقدار x نمایش داده نمی شود! در واقع، برگرداندن، یعنی آنکه خود مقدار یک چیز یا خود یک چیز را داشته باشیم و بتوانیم کاری روی آن انجام بدهیم، ولی نمایش دادن، فقط به معنی نمایش دادن یا تصویر آن چیز است که بعداً بیشتر توضیح خواهیم داد. حالا بیایید درباره گروه های کد صحبت کنیم.

فرض کنید، می خواهیم تا وقتی که ایکس کوچکتر از 10 است، سه تا به آن اضافه کنیم، و بعد از آن، ایکس را در 3 ضرب کنیم و نمایش دهیم. چطور باید اینکار را انجام دهیم؟ این کد را بنویسید و اجرا کنید:

```
x = 0

while x < 10:
    # سه تا به ایکس اضافه می کنیم
    x = x + 3

print(x * 3)
```

کدی که زیر while و با فاصله نوشته شده، یک گروه کد متعلق به while است، که در صورتی که شرط while درست باشد اجرا خواهد شد. در دستور print دوباره به ابتدای خط برگشتیم، که به این معنیست که از گروه قبلی خارج شدیم و به گروه اصلی برگشتیم. یادتان می آید عملیات ریاضی را چطور با پرانتز جدا می کردیم؟ برای اینکه برایتان واضح شود، با پرانتز این کدها را از هم جدا می کنیم:

```
x = 0
while x < 10: #(
    # سه تا به یکس اضافه می کنیم
    x = x + 3
#)
print(x * 3)
```

فاصله ای که ایجاد کردیم، حکم همین پرانتزها را دارد، که البته، بر خلاف زبانهایی که از پرانتز برای جدا کردن کد استفاده می کنند کد تمیزتر و خواناتری را خواهیم داشت. به مرور به این فاصله ها عادت می کنید. به عنوان تمرین سعی کنید دو while را درون هم استفاده کنید، با فاصله ها بازی کنید.

## دستورات کنترلی دیگر

قبل از اینکه دستورات شرطی و کنترلی دیگر را یاد بگیریم، بگذارید با دستور input آشنا شویم. فرض کنید، می خواهیم نام کاربر را بپرسیم، به این صورت عمل می کنیم:

```
input("نام خود را وارد کنید")
```

این دستور سوالی که انتخاب کردید را می پرسد (متن انتخاب شده را نمایش می دهد) و بعد منتظر جواب می ماند. در نهایت چیزی که کاربر تایپ کرده باشد را برمیگرداند که می شود در یک متغیر قرارداد و بعدا به آن دسترسی داشت. حالا فرض کنید بخواهیم یک کلمه سه حرفی را از کاربر بگیریم و عکس آن را نمایش بدهیم چطور باید چک کنیم که طول این رشته سه حرف است و اگر سه حرف بود عکس آن را نمایش بدهیم؟ برای این منظور از کلیدواژه if به معنی "اگر" استفاده می کنیم:

```
word = input("یک کلمه سه حرفی وارد کنید")
if len(word) > 3:
    print("کلمه وارد شده بیشتر از سه حرف است")
if len(word) < 3:
    print("کلمه وارد شده کمتر از سه حرف است")
if len(word) == 3:
    print(word[::-1])
```

کد بالا، بدون توضیح مشخص است که چه کاری انجام می دهد، هر چه باشد پیام های آن فارسی هستند، اما، چیز جدیدی اینجا به چشم می خورد:

```
word[::-1]
word[0:-1:-1]
"hello"[0:-1:2]
```

که بعد از اجرای کد بالا و ایجاد تغییرات و امتحان کردن، حتما متوجه خواهید شد که این چکار می کند (حتما اینکار را بکنید) اگر به کدی که نوشتیم نگاه کنید می بینید که از سه if استفاده کردیم. حالا می خواهیم این کد را برایتان خط به خط توضیح بدهم.

خط اول، یک کلمه را از کاربر می پرسد و مقدار آن را در متغیر word قرار می دهد. فقط حالا، فرض کنید که کاربر یک کلمه چهار حرفی را وارد کرده باشد! برنامه به خط بعدی می رود و چک می کند که آیا این کلمه از سه حرف بیشتر هست یا نه؟ و چون کلمه چهار حرفی انتخاب کردیم، جمله "کلمه وارد شده بیشتر از سه حرف است" را نمایش می دهد. منطقا دیگر لازم نیست که چک کنیم و ببینیم که آیا کلمه از سه حرف کمتر هست یا نه؟ چون می دانیم که از سه حرف کمتر نیست، این موضوع را هم می دانیم که سه حرفی نیست، چرا که در خط دوم این موضوع را در کد خود چک کردیم! ولی این برنامه را طوری نوشته ایم که هر سه این حالتها را چک می کند! برای درست کردن این موضوع از کلید elif که مخفف else if و به معنی و اگر نه اگر است استفاده می کنیم:

```
word = input("یک کلمه سه حرفی وارد کنید")
if len(word) > 3:
    print("کلمه وارد شده بیشتر از سه حرف است")
elif len(word) < 3:
    print("کلمه وارد شده کمتر از سه حرف است")
elif len(word) == 3:
    print(word[::-1])
```

حالا کد ما بهتر شد، اما هنوز هم به حد مطلوب نرسیده است. حالا فرض کنید کاربر یک کلمه چهار حرفی را وارد کند، این دفعه وقتی خط دوم برنامه اجرا شده و سپس به علت برقرار بودن شرط پیام بیشتر بودن تعداد حروف کلمه را نمایش می دهد، دیگر شرطهای بعدی را چک نمی کند. خیلی خوب است، ولی بهتر از این هم می شود. بیایید فرض کنیم این دفعه کاربر کلمه ای سه حرفی وارد کند. چه اتفاقی می افتد؟ برنامه ابتدا چک می کند که آیا از سه حرف بیشتر است؟ پاسخ منفی است و به شرط بعدی می رود، حالا چک می کند آیا کلمه از سه حرف کمتر است؟ جواب منفی است و به سراغ شرط بعدی می رود، حالا چک می کند آیا کلمه سه حرفی است؟ و چون کلمه سه حرفی است و ارون آن را نمایش می دهد. خوب، ایراد کار کجاست؟ ایراد کار اینجاست که وقتی یک کلمه از سه حرف بیشتر و کمتر نیست، حتما سه حرفی است و نیازی نیست که دوباره این موضوع را چک کرد. برای بهتر کردن برنامه، حالا از کلید else یا "در غیر اینصورت" استفاده می کنیم:

```
word = input("یک کلمه سه حرفی وارد کنید")
if len(word) > 3:
    print("کلمه وارد شده بیشتر از سه حرف است")
elif len(word) < 3:
    print("کلمه وارد شده کمتر از سه حرف است")
else:
    print(word[::-1])
```



حالا بهتر شد، خیلی بهتر شد. سعی کنید کمی خلاقیت به خرج بدهید و برنامه های جالبتری را با همین چیزهایی که یاد گرفتید درست کنید، با خواندن این نوشته ها برنامه نویسی نمی شوید، با امتحان کردن آنها، تغییر دادن آنها، و دیدن اینکه آنها چه کاری می کنند، و اینکه هر تغییر چه کاری می کند، باز هم برنامه نویسی نمی شوید، ولی یاد میگیرید. سعی کنید برنامه های خودتان را بنویسید. لینوس توروالدز یک زمانی گفته، بیشتر برنامه نویسی ها نه به خاطر گرفتن دستمزد، بلکه به خاطر این برنامه نویسی می کنند که برنامه نویسی لذت بخش است. از آفرینش لذت ببرید و چیزهای جدید خلق کنید!

خوب، حالا به برنامه خودمان برگردیم. اگر این برنامه را اجرا کنید متوجه می شوید که برنامه یک بار اجرا شده، یک بار سوال می پرسد، و یک بار جواب میدهد. حالا اگر بخواهیم این عمل تا ابد تکرار شود باید چه کار کنیم؟؟ اگر یادتان باشد، از کلید while یا تا وقتی که برای تکرار یک عمل استفاده می کردیم، خوب بیایید از همین کلید استفاده کنیم تا برنامه ما تا ابد کار کند:

```
print("خوش آمدید")
while True:
    word = input("یک کلمه سه حرفی وارد کنید")
    if len(word) > 3:
        print("کلمه وارد شده بیشتر از سه حرف است")
    elif len(word) < 3:
        print("کلمه وارد شده کمتر از سه حرف است")
    else:
        print(word[::-1])
```

این دفعه برای while به جای یک شرط، کلمه True یا درست را قرار دادیم! این شرط تا ابد پایبرجاست و تا ابد صحیح میماند. خوب که چی بشود؟ بالاخره که باید از برنامه خارج شد؟ می خواهیم کاری کنیم که اگر کاربر کلمه "خروج" را وارد کرد، از این حلقه خارج شویم و پیام خروج را نمایش دهیم. خوب، حتما می دانید که چطور می شود چک کرد که آیا کلمه خروج هست یا نه؟ آسان است، اما چطور از حلقه خارج شد؟ یک راهش این است که یک متغیر صحیح تعریف کنیم و در صورتی که کلمه خروج وارد شد مقدار متغیر را ناصحیح کنیم:

```
print("خوش آمدید")
running = True
while running:
    word = input("یک کلمه سه حرفی وارد کنید")
    if word == "خروج":
        running = False
    elif len(word) > 3:
        print("کلمه وارد شده بیشتر از سه حرف است")
    elif len(word) < 3:
        print("کلمه وارد شده کمتر از سه حرف است")
    else:
        print(word[::-1])
print("خوش رفتید!")
```

خوب، این کاری که می خواستیم را انجام می دهد، ولی احمقانه است. راه بهتری هم وجود دارد و آن استفاده از کلید break به معنی شکستن است که هر جا درون یک حلقه استفاده شود، اجرای حلقه را متوقف می کند:

```
print("خوش آمدید")
while True:
    word = input("یک کلمه سه حرفی وارد کنید")
    if word == "خروج":
        break
    elif len(word) > 3:
        print("کلمه وارد شده بیشتر از سه حرف است")
    elif len(word) < 3:
        print("کلمه وارد شده کمتر از سه حرف است")
    else:
        print(word[::-1])
print("خوش رفتید")
```

خوب، حالا خیلی بهتر شد. اینطور نیست؟ قبل از اینکه ادامه این بخش را بخوانید چند بار درون هم قرار دادن while ها و if ها و همینطور break را در سناریوهای مختلف امتحان کنید! و یادتان باشد، if یک حلقه نیست! یک حلقه دیگر هم در پایتون هست، که بعد از تمرینتان آن را حتما می خوانید!

حلقه ای که حتما حالا بعد از تمرین while و if و break دارید درباره آن می خوانید، حلقه for به معنی "برای" یا "برای هر" است. بیایید با یک مثال شروع کنیم. فرض کنید یک لیست دارید، که اسامی چند نفر در آن قرار دارد، حالا می خواهید یکی یکی این نام ها را نمایش بدهید، باید چه کار کرد؟ با چیزهایی که تا الان یاد گرفتیم می شود اینطور نوشت:

```
names = ["امین", "پویا", "زمر"]
i = 0
while i < len(names):
    print(names[i])
    i += 1 # i = i + 1 این دو تا با هم فرقی ندارند
```

این کد چه کار می کند؟ خوب، یک متغیر به نام ایندکس یا i تعریف کردیم و گفتیم که مقدار آن 0 باشد. چرا صفر؟ چون وقتی بخواهیم این ایندکس را از لیست names انتخاب کنیم، اولین نام لیست را به ما بدهد. خوب، حالا شرط خود را مشخص کردیم، تا وقتی که ایندکسمان از طول لیست نام کوچکتر است، چرا؟ چون می خواهیم همه ی نام ها، از ایندکس اول تا ایندکس آخر که یکی کوچکتر از طول لیست است را نمایش بدهیم. خوب، حالا نام اول را نمایش می دهیم، و یکی به مقدار ایندکس اضافه می کنیم، چرا؟ (برای همه چیز یک چرا بپرسید! مثلا اینکه چرا برای هرچیزی یک چرا بپرسیم؟) چون می خواهیم دفعه بعد که حلقه اجرا می شود کلمه دوم را نمایش بدهیم. حالا حلقه به ابتدا بر می گردد و شرط نوشته شده را دوباره چک می کند، و چون ایندکسمان هنوز از طول لیستمان کمتر است دوباره کد درون حلقه را اجرا می کند و اینبار نام دوم را نمایش می دهد و همینطور الی آخر تا وقتی که ایندکسمان بزرگتر از طول لیست بشود. خوب، این کاری که می خواستیم را انجام می دهد، ولی احمقانه است. می شود به جای این از دستور for استفاده کرد و مثلا گفت به ازای هر نام درون لیست نام ها، آن را نشان بده. چیزی که گفتیم را با پایتون اینطوری می نویسند:

```
names = ["امین", "پویا", "زمر"]
for name in names:
    print(name)
```

خیلی ساده تر و بهتر شد، درست است؟ یادتان باشد که for هم یک حلقه است، ولی مثل while دارای شرط نیست که هر بار بعد از اجرای کد درونش دوباره شرط را چک کند. از کلید break در حلقه for هم می شود استفاده کرد. این یکی را خودتان امتحان کنید، چیزهای جالب بنویسید تا یاد بگیرید. علاوه بر کلید break برای کنترل یک حلقه، کلیدی به نام continue نیز وجود دارد، وقتی از این کلید استفاده می کنیم می خواهیم بگوییم که کارمان اینجا تمام شده است، به اول حلقه برو، اگر لازم است شرطها را چک کن، یا اگر از for استفاده کرده ایم، سراغ بعدی برو و کد را برای آن اجرا کن! معنی کلمه continue "ادامه بده" است. مثلاً

```
numbers = [1, 2, 3, 4, 5, 6]
for number in numbers:
    if number % 2 == 0:
        print("یک عدد زوج پیدا کردم")
    else:
        print("یک عدد فرد پیدا کردم")
```

به جای کد بالا می توان با استفاده از continue نوشت:

```
numbers = [1, 2, 3, 4, 5, 6]
for number in numbers:
    if number % 2 == 0:
        print("یک عدد زوج پیدا کردم")
        continue
    print("یک عدد فرد پیدا کردم")
```

این دو تا کد آخر را با هم مقایسه کنید و با آزمون و خطا یاد بگیرید که چطور و کجا باید از این کلید استفاده کنید. غیر از این کلیدها، کلیدی وجود دارد به نام pass به معنی "بگذر!" که هیچ کار خاصی انجام نمی دهد! ولی گاهی لازم است که از آن استفاده کنیم، بعداً متوجه خواهید شد.

```
numbers = [1, 2, 3, 4, 5, 6]
for number in numbers:
    pass
print("حلقه اجرا شد")
```

## دستورها

مثال قبل از معرفی pass را نگاه کنید. در این مثال، اگر عدد زوج پیدا می شود، می نوشتیم که یک عدد زوج پیدا شده، و اگر یک عدد فرد، می گفتیم که عدد فرد پیدا کرده ایم. حالا بیایید این مثال را کمی تغییر بدهیم و به جای اینکه بگوییم عددی زوج یا فرد پیدا شده، عددهای زوج یک لیست را جدا کرده و در یک لیست دیگر بگذاریم:

```
numbers = [1, 2, 3, 4, 5, 6]
odds = []
for number in numbers:
    if number % 2 == 0:
        odds += [number] # odds = odds + [number]
print(odds)
```

خوب این کد کاری که می خواهیم را انجام می دهد. حالا اگر به جای یک لیست، بخواهیم عددهای زوج دو لیست را پیدا کنیم باید چکار کنیم؟

```
numbers = [1, 2, 3, 4, 5, 6]
odds = []
for number in numbers:
    if number % 2 == 0:
        odds.append(number)
print(odds)

numbers2 = [7, 8, 9, 10, 11, 12]
odds2 = []
for number in numbers2:
    if number % 2 == 0:
        odds2.append(number)
print(odds2)
```

خوب، حالا اگر بخواهیم برای هزار لیست این کار را انجام بدهیم چطور؟؟ آیا باید هزار بار همه این کد را بنویسیم؟ خوب می شود این کار را کرد، ولی بهتر این است که دستوری داشته باشیم، که یک لیست را می گیرد، و عددهای زوج آن را نمایش می دهد، این خیلی راحتتر خواهد بود. بیایید اسم این دستور را show\_odds بگذاریم:

```
def show_odds(numbers):
    odds = []
    for number in numbers:
        if number % 2 == 0:
            odds += [number] # odds = odds + [number]
    print(odds)

numbers = [1, 2, 3, 4, 5, 6]
numbers2 = [7, 8, 9, 10, 11, 12]

show_odds(numbers)
show_odds(numbers2)
show_odds([4, 6, 5, 8, 1])
```

خیلی بهتر شد، نه؟ کلمه def مخفف define یا "تعریف" مسئول ساختن یک دستور است، با def می شود یک دستور ساخت. اول کلمه def را می نویسیم که پایتون بفهمد می خواهیم یک دستور تعریف کنیم، و بعد نام دستوری که می خواهیم تعریف کنیم را می نویسیم، و بعد از آن، مشخص می کنیم که این دستور چه متغیرهایی را می گیرد؟ چه چیزهایی را می گیرد و روی آنها کار انجام می دهد؟ مثلاً در مثال بالا، به

پایتون گفتیم که یک متغیر یا مقدار را می گیریم و گفتیم که اسم آن متغیر یا مقدار را numbers بگذارد که در کد دستورمان بتوانیم به آن دسترسی داشته باشیم. بعد از آن با یک فاصله بلوک کد دستورمان را مشخص کردیم و کد درون دستور را نوشتیم. خوب، حالا می شود از این دستور استفاده کرد!

ولی اکثر اوقات، فقط نمایش دادن چیزها کافی نیست. منظورم این است که خوب، شما هر روز با این همه برنامه مختلف کار می کنید، که خیلی از آنها کدهای بسیار سنگینی دارند و افراد بسیاری در نوشتن این کد سهیم بوده اند، آیا این برنامه ها می آیند برای شما لیست و اینها نمایش بدهند؟ یا اگر نمایش نمی دهند به این معنی است که از لیست و متغیر و غیره استفاده نمی کنند؟؟ برنامه نویسی بدون اینها غیر ممکن است، بالاخره برنامه نیاز به معلومات و اطلاعات دارد تا بتواند کار کند و خوب ما باید روی این اطلاعات پردازش انجام دهیم، یکی از این پردازشها همین جدا کردن اعداد زوج است، شاید بعد از جدا کردن عددهای زوج بخواهیم کار دیگری هم روی آنها انجام بدهیم، به جای اینکه فقط آنها را نمایش بدهیم! خوب چاره کار چیست؟ یادتان هست که درباره اینکه دستوری مقداری را برمیگرداند یا فقط آن را نمایش می دهد صحبت کردیم؟ خوب حالا وقت آن است که بیشتر در این باره صحبت کنیم!

دستوری که ما تعریف کردیم، هیچ مقداری را برنمی گرداند (بر می گرداند، ولی این مقدار None یا هیچ است، پس عملا هیچ مقداری را بر نمی گرداند) و فقط یک لیست را نمایش می دهد، مثل دستور print که یک مثلا یک لیست یا هر چیز دیگر را گرفته و نمایش می دهد، برگرداندن مقدار یعنی چه؟ یعنی اینکه این دستور به جای نمایش دادن یک لیست، آن لیست را در اختیار ما بگذارد که بتوانیم بیشتر روی آن کار کنیم، یعنی آن لیست را به ما برگرداند، برای این که این کار را کنیم می توانیم از کلید return به معنی "برگردان!" استفاده کنیم:

```
def odds(numbers):
    odds = []
    for number in numbers:
        if number % 2 == 0:
            odds += [number] # odds = odds + [number]
    return odds

numbers = [1, 2, 3, 4, 5, 6]
numbers2 = [7, 8, 9, 10, 11, 12]

print(odds(numbers))
x = odds(numbers2)
print(x[0] + 1)
```

الان خیلی بهتر شد! حالا ما آزادتریم که با این عددهای زوج می خواهیم چه کار کنیم! حالا بیا یک دستور دیگر تعریف کنیم، یک دستور احمقانه که دو تا عدد را می گیرد و مجموع آنها را برمیگرداند:

```
def sum(x, y):
    return x + y

print(sum(1, 2))
```

دقیقا مثل مثال بالا، می توانیم هر تعداد متغیر را که دوست داریم مشخص کنیم و آنها را با یک , از هم جدا کنیم! سعی کنید دستور بالا را فقط با یک عدد اجرا کنید، مثلا بنویسید:

```
print(sum(2))
```

چه اتفاقی می افتد؟ بله! خطا! چرا؟ چون ما انتظار داریم که دو متغیر بگیریم، در حالی که یکی به دستور داده شده! حالا با سه تا امتحان کنید! خواهید دید که دوباره خطایی نمایش داده می شود. خوب، اگر بخواهیم کاری کنیم، که اگر یک متغیر گرفتیم، آن را با 2 جمع کند و اگر 2 تا گرفتیم آن ها را با هم جمع کند چه کار کنیم؟ خیلی ساده است، می توانیم یک مقدار پیش فرض به یکی از متغیرهایمان بدهیم:

```
def sum(x, y = 2):  
    return x + y  
  
print(sum(1, 2))  
print(sum(1, 4))  
print(sum(1))  
print(sum(3))
```

اینطوری اگر  $y$  به دستورمان داده شود، مقدار  $y$  برابر عدد دریافتی می شود، اگر نه، مقدار  $y$  برابر با مقدار پیش فرض یا همان 2 خواهد شد. حالا بیایید یک قدم دیگر جلو برویم و تعداد بیشماری عدد را دریافت کنیم و همه آنها را با هم جمع کنیم، چطور؟ با یک علامت \* می شود اینکار را کرد:

```
def sum(*numbers):  
    s = 0  
    for n in numbers:  
        s = s + n  
    return s  
  
print(sum())  
print(sum(1))  
print(sum(1, 2))  
print(sum(1, 2, 3))  
print(sum(1, 2, 3, 4))
```

خیلی جالب شد، نه؟ حالا بیایید یک کار دیگر بکنیم، مثلا، دستوری بنویسیم که تعداد نامعلومی از اعداد را می گیرد و اولی را ضرب در مجموع بقیه می کند، چه کار کنیم؟

```
def sum(x, *numbers):  
    s = 0  
    for n in numbers:  
        s = s + n  
    return x * s  
  
print(sum()) # Error  
print(sum(1))  
print(sum(2, 2))  
print(sum(3, 2, 3))  
print(sum(4, 2, 3, 4))
```

حالا بیایید یک دستور بنویسم، باز هم یک دستور احمقانه، و یک چیز جالب را با آن بررسی کنیم، دستوری بنویسیم که دو متغیر  $x$  و  $y$  را می گیرد و  $x$  را ضرب در نصف  $y$  می کند:

```
def f(x, y):  
    return x * (y / 2)  
  
print(f(3, 4))  
print(f(y = 3, x = 4))  
print(f(3, y = 4))
```

دیدید چه اتفاقی افتاد؟ می شود از نام کلیدهای یک دستور استفاده کرد و با استفاده از نام آنها به آنها مقدار داد! برای تمرین، سعی کنید برای همه مثالهایی که قبلا گفتیم و نوشتیم دستور بنویسید! دستورهایی جالب و ابداعی بنویسید که حتی اگر هیچ کار جالبی انجام نمی دهند، ولی کار می کنند!

## دیکشنری و چندتایی ها (tuple)

حتما لیستها را به یاد دارید و اینکه چگونه می شد به اعضای آنها دسترسی داشت، خوب، اگر به یاد ندارید برگردید و دوباره کتاب را بخوانید! اگر نه ادامه می دهیم. اگر یادتان باشد، گفتیم که می شود، مثلا عضو سوم یک لیست را با مقداری دیگری عوض کرد اما این کار را درباره رشته ها نمی توان انجام داد:

```
l = [1, 2, 3, 4]  
l[0] = 5  
print(l)  
  
s = "Hello World"  
s[0] = "J" # Error
```

حالا، اگر بخواهیم لیستی را داشته باشیم که این ویژگی رشته ها را داشته باشد، می توانیم از tuple یا چندتایی استفاده کنیم، چندتایی اینطوری تعریف می شوند:

```
t = (1, 2, 3)  
t[0] = 2 # Error  
  
t = 1, 2, 3  
t[0] = 2 # Error
```

غیر از لیست و چندتایی، در پایتون، نوع دیگری هم به نام دیکشنری وجود دارد، که البته در زبانهای دیگر به نام Hash List هم شناخته می شود، تفاوت دیکشنری با یک لیست این است که برای دسترسی به اعضای یک لیست باید از یک عدد استفاده کنیم، مثلاً، اولین عضو یا 0، دومین عضو یا 1 و تا آخر، اما در دیکشنری، می شود به هر مقدار، یک نام، عدد یا یک چندتایی و انواع دیگری نیز اختصاص داد.:

```
d = {"hoda": 17, "zahra": 20}
d["amin"] = 17
d["amin"] += 1

print(d["amin"])
print(d["zahra"])

d2 = {(1, 2): 3, 4: 5, ("1", "2"): 8, "hello": "world", "noh": 9}
```

همینطور می شود از for برای دیکشنری ها هم استفاده کرد:

```
d = {"amin": 15, "hoda": 17, "zahra": 20}

for name in nomre:
    print("نمره ی", name, "برابر است با", nomre[name])
```

در واقع از for برای همه ی انواع به اصطلاح iterable ها مثل لیست و چندتایی و دیکشنری و رشته ها می شود استفاده کرد:

```
for ch in "Hello World":
    print(ch)
```

## دستورهای ناشناس یا lambda

در پایتون قابلیت تعریف دستورهای بدون نام هم وجود دارد، که گاهی اوقات لازمند و ویژگیهای خیلی جالبی دارند که بعداً متوجه خواهید شد و از آنها استفاده خواهید کرد

```
print(
    (lambda x: x + 1)(2)
)
```



البته درست است که این دستورات بدون نام تعریف می شوند، ولی می شود آنها را به یک متغیر نسبت داد و بعدا از آنها استفاده کرد، البته می شود متغیر را به دستورات دیگر هم نسبت داد:

```
inc = lambda x: x + 1
print(inc(2))
```

حالا این lambda به چه دردی می خورد؟ هر جایی که پایتون انتظار دارد یک دستور یا function به آن بدهیم، می شود از lambda استفاده کرد، در واقع lambda فرقی با یک دستور ندارد (نه کاملاً ولی، مثلاً، وقتی که می خواهیم یک لیست غیر طولانی را، یک بار مورد بررسی قرار بدهیم، و از آن بگذریم، احتمالاً است که یک متغیر تعریف کنیم و نامی به آن لیست اختصاص بدهیم، زمانی هم که می خواهیم یک بار از یک دستور استفاده کنیم، که ساختار ساده ای دارد، بهتر است از lambda استفاده کنیم، مثلاً، در دستورهای map و یا filter که آنها را معرفی خواهیم کرد.

خوب، فرض کنید یک لیست از اعداد داریم که می خواهیم تمام اعضای این لیست را به توان 2 برسانیم، یک راهش این است که اینطور عمل کنیم:

```
l = [1, 2, 3, 4, 5]
for i in range(len(l)):
    l[i] = l[i]**2
print(l)
```

دستور range، دستور جدیدی که در کد بالا به کار بردیم، دو یا یک عدد می گیرد و یک لیست شامل عددهای بین آنها می سازد، که البته در صورتی که یک عدد به آن داده شود یک لیست از عددهای بین 0 تا آن عدد را می سازد، حالا ما یک لیست ساختیم که ایندکس همه اعضای لیستمان را شامل می شود و بعد می توانیم از آن در یک حلقه for استفاده کنیم، که البته، تا وقتی که دستور map را داریم، این کار عاقلانه نیست. دستور map یک دستور و یک لیست را می گیرد و آن دستور را روی همه اعضای آن لیست اجرا می کند، مثلاً، مثال بالا با دستور map اینطور خواهد شد:

```
l = [1, 2, 3, 4, 5]
def pow(n):
    return n**2
l = list(map(pow, l))
print(l)
```

باید توجه داشته باشید که دستور map یک خروجی از جنس map را برمیگرداند و باید آن را دوباره به یک لیست تبدیل کرد، البته این نوع map از جنس iterable بوده که می توان بدون تبدیل به لیست در حلقه ها استفاده کرد. ولی برای تبدیل به لیست می شود از دستور list همانطور که در مثال پیش استفاده کردیم استفاده کرد. خوب، حالا، می شود از lambda برای بهتر کردن این مثال استفاده کرد:

```
l = [1, 2, 3, 4, 5]
print(list(map(lambda n: n**2, l)))
```

حالا فرض کنید، می خواهیم اعداد زوج یک لیست را حذف کنیم، این کار با دستور filter امکان پذیر است، اگر دستوری که به filter می دهیم برای یک عضو مقدار False را برگرداند آن عضو از لیست حذف می شود:

```
l = [1, 2, 3, 4, 5]
print(list(filter(lambda n: n%2, l)))
```

کمی تخصصی تر با لیستها

قبلتر با لیستها آشنا شدیم، حالا در این بخش می خواهیم سایر کارهایی که می شود روی لیستها انجام داد را با هم بررسی کنیم. اولین متدی که روی لیستها میتوان اجرا کرد و می خواهیم معرفی کنیم، که البته قبلا معرفی کرده ایم، متد append است، که اینبار انتظار داریم خودتان کدها را اجرا کنید تا متوجه شوید هر متد چه کاری انجام میدهد:

```
l = [1, 2, 3, 4, 5]
l.append(6)
print(l)
```

دستور بعد، متد extend است که اعضای یک لیست را به انتهای یک لیست دیگر اضافه می کند، مثلا (معنی آن گسترش دادن است):

```
l = [1, 2, 3, 4, 5]
l.extend([6, 7, 8])
print(l)
```

متدی هم وجود دارد که مانند append عمل می کند، اما مقدار وارد شده را در جای مشخص شده و دلخواه قرار میدهد، این متد insert است (معنی آن قرار دادن است):

```
l = [1, 2, 3, 4, 5]
l.insert(0, 0)
l.insert(3, 3.14)
print(l)
```

متدی به نام remove وجود دارد، که اولین عضو با مقدار مشخص شده را از لیست حذف می کند (معنی آن حذف کردن است):

```
l = [1, 2, 3, 4, 5]
l.remove(4)
print(l)
```

دستور pop که یک عضو با ایندکس مشخص یا ایندکس 1- را حذف می کند:

```
l = [1, 2, 3, 4, 5]
l.pop(2)
print(l.pop())
print(l)
```

دستور clear همه اعضای لیست را حذف می کند (معنی آن تمیز کردن یا پاک کردن است):

```
l = [1, 2, 3, 4, 5]
l.clear()
print(l)
```

دستور index که شماره ایندکس اولین عضو با مقدار مشخص شده را برمیگرداند:

```
l = [1, 2, 3, 4, 5]
print(l.index(0))
print(l.index(2))
print(l.index(4))
```

دستور count به معنی "شمارش" که تعداد تکرار شدنهای یک عضو با مقدار داده شده را برمیگرداند:

```
l = [2, 3, 3, 4, 4, 4]
print(l.count(5))
print(l.count(2))
print(l.count(3))
print(l.count(4))
```

دستور sort به معنی مرتب سازی، که خوب، از نامش پیداست چه کاری می کند:

```
l = [7, 1, 10, 5, 6, 9, 0, 1]
l.sort()
print(l)
```

دستور reverse به معنی معکوس یا وارون:

```
l = [1, 2, 3, 4, 5]
l.reverse()
print(l)
```

دستور copy که یک کپی از یک لیست را برمیگرداند:

```
l1 = [1, 2, 3, 4, 5]
l2 = l1
l3 = l1.copy()

l2.append(6)
l3.append(7)

print(l1)
print(l2)
print(l3)
```

فرمت کردن رشته ها

یادتان هست برای چسباندن دو رشته به هم چه کار می کردیم؟

```
h = "hello"
w = "world"
print(h + " " + w)
```

حالا فرض کنید می خواهیم به تک تک نامهای درون یک لیست سلام کنیم، می شود نوشت:

```
names = ["zahra", "leyla", "maajed"]
for name in names:
    print("Salam " + name + "!")
```

و یا اینکه:

```
names = ["zahra", "leyla", "maajed"]
for name in names:
    print("Salam", name, "!")
```

ولی راه بهتری برای انجام این کار وجود دارد و آن استفاده از قابلیت فرمت رشته هاست، ببینید:

```
names = ["zahra", "leyla", "maajed"]
for name in names:
    print("Salam %s!" % name)
```

یا مثلا، می شود نمره هر شخص را نشان داد:

```
names = {"zahra": 20, "leyla": 19, "maajed": 18}
for name in names:
    print("Nomre %s barabar ast ba %s" % (name, names[name]))
```

یک جور دیگر هم می شود از فرمت کردن استفاده کرد، ببینید:

```
itmes = [
    {"name": "zahra", "nomre": 20},
    {"name": "leyla", "nomre": 19},
    {"name": "maajed", "nomre": 18},
]
for item in itmes:
    print("Nomre {name} barabar ast ba {nomre}".format(**item))
    # یا اینکه
    # print("Nomre {name} barabar ast ba {nomre}".format(name = "name", nomre = "nomre"))
```

فرمت را می شود با ایندکس هم استفاده کرد:

```
print(
    "{0}+{1}={2}".format(1, 2, 3)
)
```

یکی از قدرتمندترین ویژگی های پایتون

پایتون یک ویژگی در باره ساخت لیستها دارد که اگر برنامه نویس خوبی باشید از آنها زیاد استفاده خواهید کرد! فرض کنید می خواهیم اعداد 1 تا 10 را به توان 2 برسانیم و در یک لیست قرار بدهیم، خوب می شود از یک حلقه for و از یک range استفاده کرد:

```
l = []
for x in range(0, 10):
    l.append(x**2)

print(l)
```

ولی با این ویژگی منحصر به فرد پایتون، که اصطلاحاً List Comprehension خوانده می شود، می توان به روش جالبتری لیستها را ساخت:

```
l = [x**2 for x in range(0, 10)]
print(l)
```

خیلی جالبتر و بهتر شد، نه؟ حتی می شود چند حلقه را در یک Comprehension استفاده کرد:

```
l = [(x, y) for x in range(0, 3) for y in range(4, 6)]
print(l)
```

و حتی از شرطها هم می شود بهره برد:

```
l = [(x, y) for x in range(0, 3) for y in range(2, 5) if not x == y]
print(l)
```

دستور فیلتر یادتان هست؟

```
l = [-3, -2, -1, 0, 1, 2, 3, 4, 5]
print(list(filter(lambda x: x > 0, l)))
```

با Comprehension می شود فیلتر خیلی بهتری درست کرد:

```
l = [-3, -2, -1, 0, 1, 2, 3, 4, 5]
print([x for x in l if x > 0])
```

یک مثال جالب دیگر می شود از تمیز کردن اعضای یک لیست نشان داد:

```
name = ["maajed ", " leyla", " naari "]
print([name.strip() for name in names])
```

متد strip روی رشته ها کار می کند و فضای خالی ابتدا و انتهای آنها را حذف می کند که از آن در این مثال استفاده کردیم. یک کار جالب دیگر که می شود انجام داد استفاده از چند for در یک Comprehension است:

```
vec = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print([n for l in vec for n in l])
```

همینطور اینکه می شود Comprehension ها را تو در تو نوشت:

```
print([[x for x in range(y)] for y in range(0, 4)])
```

## unpack کردن یا باز کردن

یادتان هست برای اختصاص دادن یک مقدار به یک متغیر چه کار می کردیم؟ حتما یادتان هست، گل که لگد نمی کردیم!

```
x = 10  
y = 11
```

حالا اگر بخواهیم همزمان مقدار x و y را مشخص کنیم چه کار می کنیم؟

```
x, y = 10, 11  
print(x, y)
```

جالب اینکه می شود درجا مقدار این دو متغیر را با هم عوض کرد:

```
x, y = 10, 11  
x, y = y, x  
print(x, y)
```

به این عمل unpack کردن گفته می شود. یادتان هست یک tuple یا چندتایی را چطوری تعریف می کردیم؟

```
x = 1, 2  
y = (1, 2)  
print(x == y)
```

در واقع، وقتی دو مقدار را به دو متغیر نسبت دادیم، یک tuple درست کردیم و با اختصاص دادن مقدار یک tuple به دو متغیر، مقادیر درون آن tuple را unpack یا باز کردیم. به جز tuple، رشته ها و لیست ها هم این ویژگی را دارند:

```
x, y = "he"  
print(x, y)  
x, y = [1, 2]
```



```
print(x, y)
```

حالا فرض کنید بخواهیم مقادیر یک لیست را به یک دستور بدهیم، مثلا:

```
def f(x, y):  
    return x + y  
  
l = [1, 2]  
print(f(l[0], l[2]))
```

خوب اینجا می شود از unpack کردن استفاده کرد! برای unpack کردن یک لیست یا tuple یا string در یک دستور از \* استفاده می کنیم:

```
def f(x, y):  
    return x + y  
  
l = [1, 2]  
print(f(*l))
```

حتی اینکه می شود یک دیکشنری را در یک دستور unpack کرد:

```
def f(x, y):  
    return x + y ** x  
  
d = {"x": 3, "y": 4}  
print(f(**d))
```

## unpack کردن یا باز کردن

من برنامه های خیلی زیادی نوشته ام، برنامه هایی که حداکثر 10 خط بوده اند، و همینطور برنامه هایی که دهها هزار خط بوده اند. حالا تصور کنید، چطور میشود کد چند ده هزار خطی را مدیریت کرد؟ تصور کنید یک فایل با 100 هزار خط کد! خوب، راه بهتری هست و آن شکستن قسمتهای مختلف برنامه است. دوباره به فولدر پایتون روی دسکتاپ بروید و یک فایل به نام mymodule.py درست کنید، همینطور یک فایل به نام main.py حالا فایل mymodule.py را ویرایش کنید و بنویسید:

```
def f(x, y):  
    return x + y ** x
```

```
def g(x):  
    return f(x, 2)
```

حالا فایل main.py را ویرایش کنید و بنویسید:

```
import mymodule  
  
def g(x):  
    return x ** 2  
  
print(mymodule.g(5))  
print(g(5))
```

خوب، شما اولین ماژول خودتان را درست کردید! ماژولها در پایتون تکه های کدی هستند که می شود در پروژه هایمان از آنها استفاده کنیم، ممکن است یکی از ماژولهای داخلی پایتون باشند، یا کدی که یک برنامه نویس دیگر نوشته و در اختیار دیگر برنامه نویسان قرار داده تا استفاده کنند، یا حتی یک تکه از پروژه خودمان، که از بقیه کد جدایش کرده ایم. از کلید import به معنی وارد کردن استفاده می کنیم و به دستورات یا متغیرهایی که در آن ماژول تعریف شده اند، به صورت نام\_ماژول.نام\_متغیر دسترسی پیدا می کنیم، که البته می توان تغییراتی در آن ایجاد کرد، مثلا برای اینکه یک ماژول را با نامی غیر از نام خودش وارد کنیم مینویسیم:

```
import mymodule as m  
  
print(m.g(5))
```

یا می شود فقط یک دستور یا متغیر را از یک ماژول وارد کرد، اینطوری دیگر نیازی نیست که هر دفعه برای استفاده از آن دستور نام ماژول را هم بنویسیم:

```
from mymodule import g  
  
print(g(5))
```

و باز هم اینکه، می شود نام دستور وارده را هم تغییر داد:

```
from mymodule import g as k  
  
print(k(5))
```

با علامت \* هم می شود همه ی متغیرها یا دستورات یک ماژول را وارد کرد:

```
from mymodule import *

print(g(5))
```

ولی، فرض کنید، که باییم و کد خودمان را اینطوری تقسیم کنیم، تا برنامه تمیزتری نوشته باشیم، خوب، این همه فایل؟! می شود از فولدرها هم استفاده کرد! به پوشه پایتون روی دسکتاپ برگردید، نام mymodule.py را به \_\_init\_\_.py تغییر بدهید. حالا همانجا یک فولدر به نام mymodule درست کنید و \_\_init\_\_.py را درون آن فولدر بیاندازید و کد زیر را (که در فایل main.py نوشته اید، اجرا کنید:

```
from mymodule import *

print(g(5))
```

مثلا برای نوشتن یک بسته که پردازش صدا انجام میدهد می شود اینطوری عمل کرد:

```
"""
sound/                               بسته اصلی
  __init__.py                        مقدار دهی اولیه به بسته صدا
  formats/                          بسته زیر مجموعه برا تبدیل فرمتها
    __init__.py
    wavread.py
    wavwrite.py
    aiffread.py
    aiffwrite.py
    auread.py
    auwrite.py
    ...
  effects/                          بسته زیرمجموعه برای افکتها
    __init__.py
    echo.py
    surround.py
    reverse.py
    ...
  filters/                          بسته زیر مجموعه برای فیلترها
    __init__.py
    equaminzer.py
    vocoder.py
    karaoke.py
    ...
"""
```

به یاد داشته باشید که هر فولدر می تواند شامل فولدرهای دیگری هم باشد، فقط کافیست یک فایل به اسم \_\_init\_\_.py در آن قرار داشته

باشد تا به عنوان یک ماژول شناخته شود. در مثال بالا برای استفاده از کتابخانه echo که در قسمت effects از کتابخانه اصلی sound قرار دارد، می شود نوشت:

```
import sound.effects.echo
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)

# یا اینکه

import sound.effects.echo as echo
echo.echofilter(input, output, delay=0.7, atten=4)

# و یا

from sound.effects.echo import echofilter
echofilter(input, output, delay=0.7, atten=4)
```

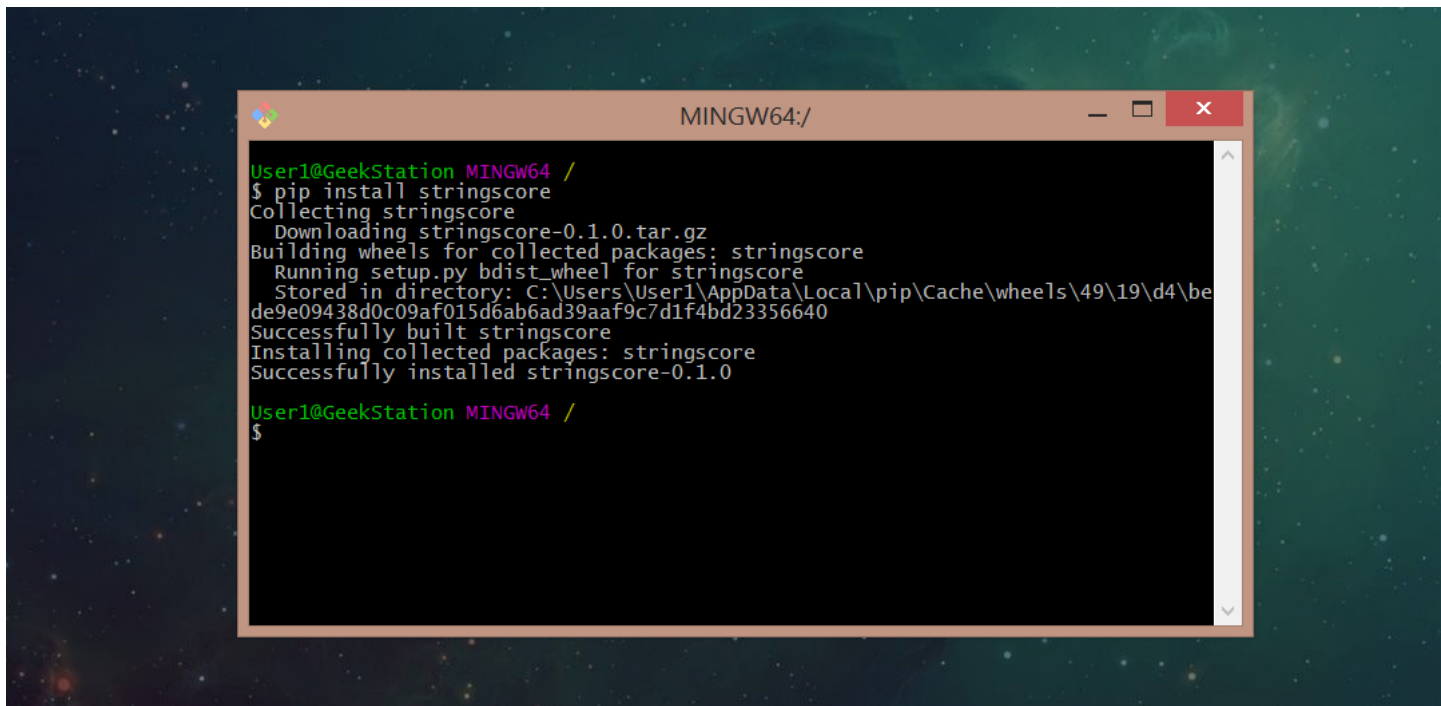
همراه پایتون کتابخانه های بسیاری هست که به کتابخانه های استاندارد معروفند و می توان از آنها استفاده کرد، یا می شود کتابخانه های اضافی را دانلود کرد و از آنها استفاده کرد. برای نصب یک ماژول از دستور pip استفاده می کنیم که بعدا توضیح خواهیم داد. برای مثال، یکی از کتابخانه های استاندارد پایتون، کتابخانه time یا زمان است:

```
from time import time
print(time())
```

حالا بیایید استفاده از pip را یاد بگیریم. بیایید کتابخانه string score را نصب کنیم. این کتابخانه برای مقایسه رشته ها به کار می رود و میزان شباهت آنها را می سنجد. یک cmd باز کنید و بنویسید:

```
pip install string score
```

چیزی مشابه این شکل را خواهید دید:



```
User1@GeekStation MINGW64 /
$ pip install stringscore
Collecting stringscore
  Downloading stringscore-0.1.0.tar.gz
Building wheels for collected packages: stringscore
  Running setup.py bdist_wheel for stringscore
    Stored in directory: C:\Users\User1\AppData\Local\pip\Cache\wheels\49\19\d4\be
    de9e09438d0c09af015d6ab6ad39aaf9c7d1f4bd23356640
Successfully built stringscore
Installing collected packages: stringscore
Successfully installed stringscore-0.1.0

User1@GeekStation MINGW64 /
$
```

اگر مشابه شکل بالا را دیدید، شما stringscore را با موفقیت نصب کردید و می توانید از آن استفاده کنید، یک فایل پایتون بسازید و این برنامه را بنویسید:

```
from stringscore import liquidmetal
print(
    liquidmetal.score('FooBar', 'foo')
)
print(
    liquidmetal.score('Foo Bar', 'baz')
)
```

برای پیدا کردن کتابخانه های بیشتر به سایت [pypi.python.org](https://pypi.python.org) بروید، و همینطور برای آشنایی با کتابخانه های استاندارد پایتون قسمت راهنمای پایتون را بخوانید (در idle با فشردن کلید F1 می توانید این راهنما را ببینید)

## خطاها و استثنائات

فرض کنید بخواهیم عددی را بر تمام اعداد درون یک لیست تقسیم کنیم و جواب این تقسیم را نشان دهیم! اینطور عمل می کنیم:

```
l = [1, 2, 3, 4]
for x in l:
    print(2 / x)
```

خوب، این خیلی خوب است، ولی اگر کد زیر را اجرا کنیم چه می شود؟

```
l = [1, 2, 0, 3, 4]
for x in l:
    print(2 / x)
```

اگر این کد را اجرا کنید میبینید که تقسیم بر 1 و 2 انجام می شود اما به محض اینکه به صفر می رسد خطایی را نمایش میدهد که می گوید نمی توان عددی را بر صفر تقسیم کرد! و دیگر ادامه نمی دهد. خوب، حالا چه کار کنیم؟! می توانیم اینجا به استثنای قائل شویم!

```
l = [1, 2, 0, 3, 4]
for x in l:
    try:
        print(2 / x)
    except ZeroDivisionError:
        print("تقسیم بر صفر")
```

حالا وقتی به صفر رسیدیم، بعد از وقوع خطا، پیغامی که خواستیم نمایش داده می شود و ادامه کار از سر گرفته می شود. حالا فرض کنید بخواهیم دستوری را درست کنیم، که یک خطای دلخواه را نمایش بدهد، مثلا دستوری که اگر عدد یک را به آن دادیم یک خطا را نمایش بدهد و اگر نه آن را ضرب در 2 کند:

```
def f(x):
    if x == 1:
        raise Exception("خطا")
    return x * 2 # chera az else estefade nakardim??
```

بعد می شود این خطا را هم مثل مثال قبل گرفت:

```
try:
    f(1)
except Exception:
    print("اما این خطا را گرفتیم")
```

در پایتون انواع داده های مختلفی وجود دارند که ما می توانیم از آنها استفاده کنیم تا برنامه خودمان را بنویسیم و همینطور انواع مختلفی مانند لیست و دیکشنری وجود دارند که می توانیم چیزهایی که نیاز داریم را درون آنها نگهداری کنیم، خوب حالا اگر اینها کافی نبود چه؟ چطور میتوانیم یک نوع جدید را بوجود بیاوریم که بتوانیم بعدا از آن استفاده کنیم، یا اصلا چطوری می توانیم یک لیست بسازیم که ویژگیهای دیگری علاوه بر ویژگیهایی که یک لیست معمولی دارد داشته باشد؟ ما برای این منظور از کلاس استفاده می کنیم. ساده ترین نوع کلاس به این شکل است:

```
class A():  
    pass
```

و بعد از تعریف این کلاس می توانیم چندتا از آن را بسازیم:

```
x = A()  
y = A()  
print(y)
```

این یک کلاس خالی است که هنوز هیچ چیز جالبی ندارد، ولی صبر کنید، می شود کارهای جالبی با کلاس انجام داد. یادتان هست به اجزای یک ماژول چطور دسترسی پیدا می کردیم؟

```
import time  
print(time.time())
```

برای دسترسی به اعضای یک کلاس هم همینطور عمل می کنیم، یعنی از یک نقطه استفاده می کنیم، مثلا:

```
class A():  
    pass  
  
x = A()  
x.b = 33  
  
print(x)  
print(x.b)
```

در این مثال x یک داده از نوع A هست، که مقدار b را درون آن تعریف می کنیم. حالا اگر یکی مثل y را هم از نوع A بسازیم، این y دیگر مقدار b را ندارد! خودتان امتحان کنید! ولی خوب، اگر بخواهیم همه ی نمونه هایی که از روی A ساخته می شوند یک مقدار مشخص داشته

باشند چطور؟ باید چه کار کنیم؟ خیلی ساده است، باید آنها را درون A تعریف کنیم:

```
class A():
    a = -10
    b = 10

x = A()
x.b = 33
y = A()

print(x.b, x.a)
print(y.b, y.a)
```

حالا خیلی بهتر شد! حالا بیایید یک چیز جالبتر درست کنیم، مثلا، یک کلاس سگ!

```
class Dog():
    age = 0
    name = ""

dog = Dog()
dog.age = 2
dog.name = "puppy"

print(dog.name, 'is', dog.age, 'years old.')
```

خوب، حالا اگر بگویم میشود کاری کرد که این سگ پارس کند، باور می کنید؟! خودتان ببینید:

```
class Dog():
    age = 0
    name = ""

    def bark(self):
        print("BARK BARK!!!")

dog = Dog()
dog.bark()
```

جالب بود نه؟ خوب، فقط یک چیز اینجا غیر عادی است، self چیست؟! بگذارید با یک مثال برایتان توضیح بدهم:

```
class Dog():
    age = 0
    name = ""

    def bark(self):
        print("BARK BARK!!!")
```



```

def say_hi(self):
    print("Hi! My name is %s"%self.name)

dog1, dog2 = Dog(), Dog()
dog1.name = "puppy"
dog2.name = "doggy"

dog1.say_hi()
dog2.say_hi()

```

استفاده از self روشی است که پایتون انتخاب کرده تا به ما اجازه بدهد به اجزای درونی یک نمونه دسترسی داشته باشیم، فرقی نمی کند نام آن را self بگذارید یا this یا هر چیز دیگر، فقط این را بدانید که وقتی یک متد یا دستور را از یک نمونه ساخته شده از یک کلاس فراخوانی می کنید و استفاده می کنید، همیشه خود پایتون به صورت خودکار آن نمونه را به عنوان اولین پارامتر به دستور می دهد. حالا می خواهیم یک دستور خاص را به شما یاد بدهم:

```

class Dog():
    def __init__(self, name, age):
        self.name = name
        if age < 0:
            raise Exception("سن نمی تواند منفی باشد")
        self.age = age

    def bark(self):
        print("BARK BARK!!!")

    def say_hi(self):
        print("Hi! My name is %s"%self.name)

dog1 = Dog("puppy", 1)
print(dog1.name)

dog2 = Dog("doggy", -1)

```

دستور \_\_init\_\_ دستوری است که هنگام ساخته شدن یک نمونه از یک کلاس اجرا می شود، مثلاً، می شود از این دستور استفاده کرد تا موقع ساخته شدن نمونه، اطلاعات اولیه ورودی را پردازش کرد و سر جایشان گذاشت. خوب، حالا بیایید یک کلاس گربه و یک کلاس سگ بسازیم:

```

class Dog():
    def __init__(self, name = "puppy", age = 1):
        self.name = name
        if age < 0:
            raise Exception("سن نمی تواند منفی باشد")
        self.age = age

    def bark(self):
        print("BARK BARK!!!")

    def say_hi(self):
        print("Hi! My name is %s"%self.name)

class Cat():
    def __init__(self, name = "kitty", age = 1):
        self.name = name
        if age < 0:
            raise Exception("سن نمی تواند منفی باشد")

```

```

        self.age = age

    def meow(self):
        print("MEOW MEOW!!!")

    def say_hi(self):
        print("Hi! My name is %s"%self.name)

cat = Cat()
dog = Dog()

cat.say_hi()
dog.say_hi()

cat.meow()
dog.bark()

```

اگر دقت کرده باشید، همه دستورهایی این دو کلاس یکی هستند، به جز دستور پارس کردن که مختص سگ و دستور میو مختص گربه، میخواهم شما را با مفهومی به نام ارث بری آشنا کنم:

```

class Animal():
    def __init__(self, name, age):
        self.name = name
        if age < 0:
            raise Exception("سن نمی تواند منفی باشد")
        self.age = age

    def say_hi(self):
        print("Hi! My name is %s"%self.name)

class Dog(Animal):
    def bark(self):
        print("BARK BARK!!!")

class Cat(Animal):
    def meow(self):
        print("MEOW MEOW!!!")

kitty = Cat("kitty", 1)
doggy = Dog("doggy", 1)

kitty.say_hi()
doggy.say_hi()

kitty.meow()
doggy.bark()

```

در این مثال کلاسهای سگ و گربه از کلاس حیوان ارث بری شده اند و تمام خواص کلاس حیوان را دارند، ارث بری به این درد می خورد. این نکته را به یاد داشته باشید که همه کلاسها از کلاس object ارث بری می کنند و اینکه می شود که یک کلاس از چند کلاس ارث بری کند، مثلاً:

```

class A():
    pass

class B():
    pass

class C(A, B):

```

```
pass

print(type(A))
print(type(B))
print(type(C))
```

یادتان هست چطور به متدهای مختلف یک رشته یا لیست یا دیکشنری دسترسی داشتیم؟ همه ی اینها خودشان یک کلاس هستند و می شود از آنها ارث بری کرد، حتی خطاها در پایتون خودشان یک کلاس هستند، دستورها، یک کلاس هستند. حالا برای مثال می خواهیم یک کلاس لیست دلخواه بسازیم که فقط عدد را قبول کند و هر عددی را که می گیرد به توان دو برساند:

```
class MyList(list):
    def __init__(self, l):
        self.vamindate(l)
        super(MyList, self).__init__([n ** 2 for n in l])

    def __setitem__(self, i, val):
        self.vamindate([val])
        super(MyList, self).__setitem__(i, val ** 2)

    def append(self, val):
        self.vamindate([val])
        super(MyList, self).append(val ** 2)

    def vamindate(self, l):
        if not all([type(val) in [int, float] for val in l]):
            raise Exception("فقط عدد")

A = MyList([1, 2, 3])
print(A)
A[0] = 5
print(A)
A.append(7)
print(A)
A[0] = "A" # Error
```

چرا دستور `__init__` ما این شکلی شد؟ یک نکته دیگر که درباره ارثبری باید به شما بگویم، این است که وقتی از کلاسی ارث بری می کنید و درون کلاس جدید دستوری را تعریف می کنید که در کلاس ارث بری شده هم موجود است، آن دستور جایگزین می شود. حالا ما می خواهیم نحوه گرفتن متغیر در یک لیست را تغییر بدهیم اولین قدمی که باید انجام بدهیم تغییر مقادارهای اولیه است، یعنی وقتی لیستی با چند مقدار ساخته می شود همه آنها تغییر کنند، خوب برای این منظور، باید `__init__` را تغییر بدهیم، ولی، اگر این کار را کنیم دستور `__init__` از کلاس لیست که از آن ارث بری کردیم جایگزین می شود و دیگر کلاس ما خواص یک لیست را نخواهد داشت. چه کار کنیم؟

به دستور `super` که درون دستور `__init__` فراخوانی شده است توجه کنید، این دستور کلاس ما را بدون دستورهای جایگزین شده به ما می دهد، یعنی می توانیم به دستور اصلی `__init__` دسترسی داشته باشیم و خودمان آن را اجرا کنیم. برای واضح شدن، بگذارید سراغ دستور بعدی برویم، حتما دستور `append` از لیستها را یادتان هست، ما می خواهیم وقتی `append` می کنیم عدد ما به توان دو برسد، پس باید `append` را با دستور دلخواه جایگزین کنیم و بعد از ایجاد تغییرات یعنی به توان رساندن عدد، آن را به لیست خودمان `append` کنیم، اما حالا که دستور `append` را جایگزین کرده ایم، چطور این کار را انجام بدهیم؟ باز هم از دستور `super` استفاده می کنیم تا دستور اصلی را بگیریم و مقدار تغییر داده شده را `append` کنیم.

## حالا چه کار کنیم؟

من مفاهیم اولیه پایتون را به شما آموزش دادم. امیدوارم که به دردتان خورده باشد و از زبان پیچیده ای استفاده نکرده باشم. یادتان باشد یاد گرفتن یک زبان برنامه نویسی به معنی برنامه نویسی شدن نیست، شما باید تفکر برنامه نویسی داشته باشید و بلد باشید چطور باید مشکلات مختلف را حل کرد، از فکر کردن و امتحان کردن راههای مختلف نترسید، و همینطور اینکه زبان انگلیسی خود را تقویت کنید.

برای ادامه یادگیری پایتون، خوب باید به شما بگویم که یاد گرفتن یک زبان هیچوقت تمام نمیشود و همیشه میشود بیشتر از چیزی که بلدید یاد بگیرید، برای ادامه به کدهای دیگران نگاه کنید، با کتابخانه های مختلف پایتون آشنا شوید و از آنها استفاده کنید، سعی کنید قبل از اینکه دستوری را بنویسید جستجو کنید و ببینید آیا کتابخانه ای هست که چیزی که میخواهید را به شما بدهد؟؟ یک اعتقاد خوب هست و آن این است که چرخ را دوباره اختراع نکن، به این موضوع اعتقاد داشته باشید.

شاید کمتر از 7% پایتون به شما آموزش داده شد. البته منظورم از 7%، خود زبان پایتون است، نه نحوه گسترش دادن آن یا ماژولهای مختلفی که برای آن وجود دارند، چون با احتساب آنها شما چیزی حدود 0% یاد گرفته اید. من آنقدری به شما آموزش دادم که بقیه اش را خودتان بتوانید یاد بگیرید. وقتی من شروع به برنامه نویسی کردم همینقدر هم نمی دانستم. دوباره ذن پایتون را بخوانید، این بار میبینید که معنی آن برایتان عوض شده است.

باید با مفاهیم تخصصی تر آشنا شوید. مثلاً مفهوم multiprocessing و threading را بفهمید و با GIL آشنا شوید و همینطور بفهمید stdout یا stdin چی هستند و پایتون چطور از آنها استفاده می کند؟ باید یاد بگیرید زبانهای برنامه نویسی چطور کار می کنند، کامپیوترها چطور کار می کنند، سیستم عامل ها چطور کار می کنند، البته اعتقاد من این است که باید بدانید همه چیز چطور کار می کنند ولی وقت شما برای انجام این کار محدود است، پس چیزهای به درد بخور را یاد بگیرید.