آموزش پایتون

نویسنده

مرتضی مالکی

تابستان ۱۴۰۱

فهرست مطالب

فهرست جداول

فهرست تصاویر

فصل ۱

مبانی پایه

۱ـ۱ کامپیوتر و تعامل اجزای آن

چنانچه بخواهیم در یک زبان برنامه‌نویسی مهارت پیدا کنیم و برنامه‌های محکم و دقیقی از خود به جای بگذاریم، می‌بایست در مورد کامپیوتر، اجزای تشکیل دهنده‌ی آن و همچنین ارتباط آن اجزا با یکدیگر آگاهی داشته باشیم. کامپیوتر مانند خودرویی است که ما سوار می‌شویم و ما به عنوان راننده شاید نیازی به دانستن تک‌تک اجزای آن خودرو نداریم، اما برای اینکه بتوانیم با مهارت و به درستی این خودرو را برانیم ملزم به داشتن اطلاعاتی در مورد آن هستیم. حال برای اینکه مطلب را بهتر متوجه شویم شاید بهتر است با یک مثال، اجزای کامپیوتر و ارتباط آن‌ها به هم را مرور کنیم. چنانچه به یک نانوایی نگاه بیندازیم متوجه وجود اجزای اصلی زیر در آن می‌شویم:

۱. مخزن آرد

۲. ظرف خمیر

۳. تنور

۴. نانوا

البته اجزای دیگری نیز وجود دارد که به ما به اجزای اصلی بسنده کردیم. حال چنانچه به ارتباط این اجزا نگاه کنیم می‌توانیم رویه کار آن را به کامپیوتر تناظر دهیم. مثلا مخزن آرد می‌تواند حکم هارد کامپیوتر

ما را داشته باشد، وقتی نانوایی بسته می‌شود تنها قسمتی که فعال می‌ماند مخزن آرد است، همین رویه برای سیستم کامپیوتری ما صادق است و برای همین است فایل‌هایمان را پس از خاموش و روشن کردن سیستم از دست نمی‌دهیم.

توجه: در هارد کامپیوتر پردازش اطلاعات رخ نمی‌دهد و تنها مکانی برای ذخیره‌ی اطلاعات است.

حال چنانچه ظرف خمیر را بررسی کنیم می‌بینیم این قسمت نیز مانند رم (RAM) سیستم ما می‌باشد، بطوریکه هر موقع سیستم را روشن کنیم داده‌هایی که می‌خواهیم پردازش شوند در آن قرار می‌گیرند، همانطور که می‌دانید نانواها نمی‌توانند بیشتر از ظرف خمیری که دارند در هر وعده پخت داشته باشند که همین اتفاق برای کامپیوترهای ما رخ می‌دهد، یعنی ما نیز بیشتر از رم کامپیوتر نمی‌توانیم برنامه باز داشته باشیم و چنانچه این کار را انجام دهیم سیستم ما متوقف می‌شود.

بدانیم: حتما متوقف شدن سیستم یا اصطلاحا هنگ کردن سیستم را دیده‌اید، یکی از دلایل آن همین است.

اما تنور نانوایی، مهم‌ترین بخش نانوایی است که معادل واحد پردازش (CPU) کامپیوتر ما می‌باشد. تنور قوی معادل پردازش قوی است و سرعت اجرای برنامه‌هایی است که ما برای اجرا قرار داده‌ایم.

اما نانوا معادل چه چیزی در کامپیوتر است؟

تا الان متوجه شدیم که در رم و هارد پردازشی رخ نمی دهد و داده‌ها می‌بایست از هارد به رم منتقل شوند و سپس در صف قرار گیرند تا توسط **واحد پردازش**، پردازش شوند. وظیفه‌ی نقل و انتقال داده‌ها بین این بخش‌ها بر عهده‌ی BUS می‌باشد (که در نانوایی همان نانوا است).

حال که این بخش‌ها را بررسی کردیم احتمالا اهمیت هر کدام از بخش‌ها و البته تعامل آن‌ها با یکدیگر را بهتر متوجه شویم.

چنانچه یک کامپیوتر رم بالا و cpu ضعیفی داشته باشد، با چه مشکلی روبرو خواهد شد؟ به احتمال‌های دیگر نیز فکر کنید.

۱ـ۲ ارتباط زبان برنامه نویسی پایتون به کامپیوتر

۱ـ۳ مفسر پایتون

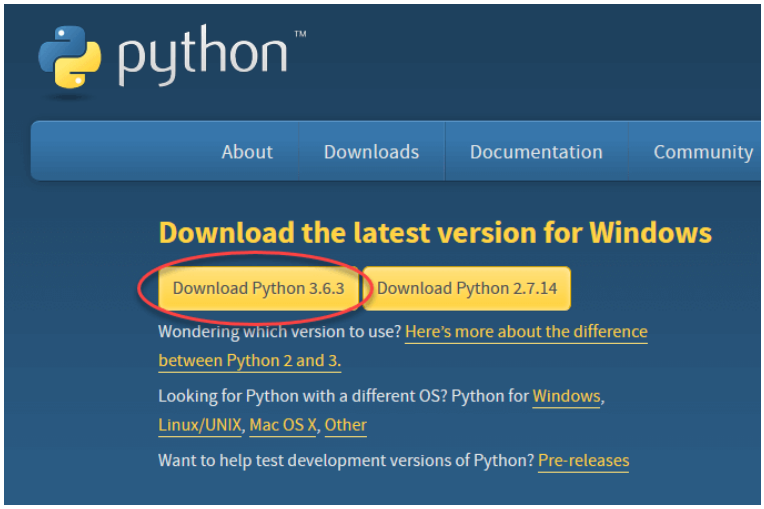
پایتون دارای یک محیط توسعه و یادگیری یکپارچه است که به IDLE یا حتی IDE کوتاه شناخته‌می‌شود. اینها دسته‌ای از برنامه‌ها هستند که به شما کمک می کنند کد را کارآمدتر بنویسید. در حالی که IDE های زیادی برای انتخاب وجود دارد، IDLE Python بسیار ساده است، که آن را به ابزاری عالی برای یک برنامه نویس مبتدی تبدیل می کند.

توجه: ویرایشگر استاندارد پایتون تنها موردی نیست که وجود دارد و علاوه بر آن ویرایشگرهای بسیار غنی‌تری نیز وجود دارد که برای شروع ما از این ویرایشگر استفاده خواهیم کرد و پس از آن به دیگر ویرایشگرها خواهیم پرداخت.

برای نصب این ویرایشگر می‌توانیم مراحل زیر را طی کنیم:

قدم اول

برای دانلود ویرایشگر به سایت رسمی پایتون با لینک www.python.org/downloads مراجعه کنید و نسخه‌ی مورد نظر را برای سیستم عامل خود انتخاب کنید.



توجه: پایتون دو نسخه‌ی با تغییرات اساسی دارد

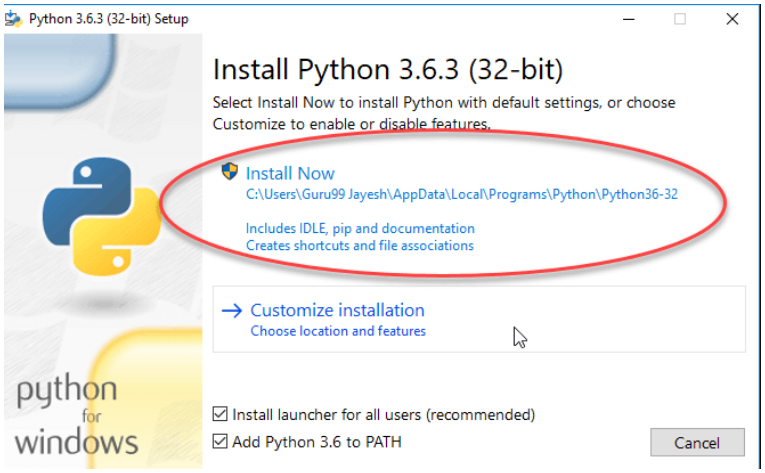
پایتون ۲: در سال ۲۰۲۰ از دور خارج شد.

پایتون ۳: نسخه‌ی کنونی

توجه: ممکن است برای شما نسخه‌ی ۱۰.۳ باشد و این برای کار ما تفاوت زیادی نمی‌کند.

قدم دوم

پس از دانلود و باز کردن فایل نصبی با صفحه‌ی زیر مواجه خواهیم شد



توجه: حتما تیک Add python ۶.۳ to PATH را بزنید تا در آینده برای نصب پکیج‌هابا مشکل مواجه نشویم.

قدم سوم

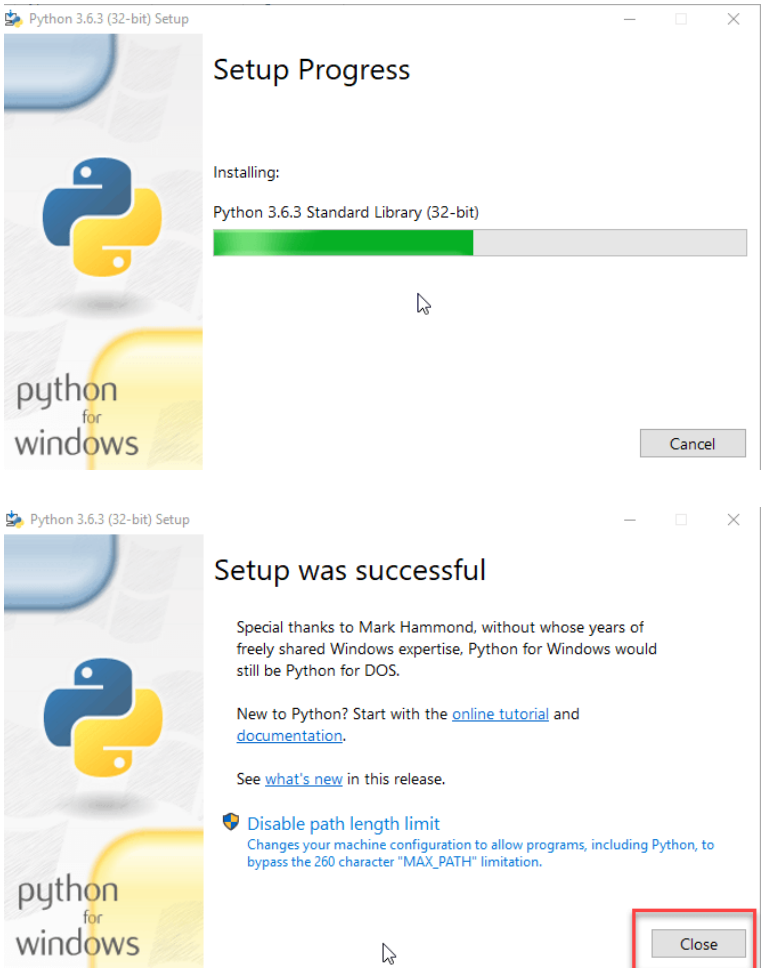
پس از کلیک بر now install پنجره‌ی زیر را خواهید دید:

قدم چهارم

پس از اتمام نصب با صفحه‌ی زیر مواجه خواهید شد که می‌توانید close را بزنید

حال همه چیز برای ما مهیا شد تا بتوانیم کدزنی را شروع کنیم.

در ابتدا بخش‌های مختلف IDLE پایتون را با هم بررسی می‌کنیم و پس از آن به مقدمات پایتون خواهیم



پرداخت. مفسر پایتون از دو بخش اصلی تشکیل می‌شود:

۱. مفسر تعاملی [1]

۲. ویرایشگر فایل

۱ـ۳ـ۱ مفسر تعاملی

بهترین مکان برای آزمایش کد پایتون در مفسر تعاملی است که به آن shell یا console نیز گفته می‌شود. shell یک حلقه اصلی Read-Eval-Print (REPL) است. عملکرد shell به این صورت است که

[1] Interactive Interpreter

یک دستور پایتون را می خواند، نتیجه آن عبارت را ارزیابی می کند و سپس نتیجه را روی صفحه چاپ می کند. سپس، برای خواندن عبارت بعدی، دوباره حلقه می‌زند.

shell پایتون یک بخش بسیار عالی برای بررسی کدهای کوچک است.

۱ـ۳ـ۲ ویرایشگر فایل

هر برنامه‌نویسی باید بتواند فایل‌های متنی را ویرایش و ذخیره کند. برنامه های پایتون فایل‌هایی با پسوند py هستند که حاوی خطوط کد پایتون هستند. IDLE پایتون به شما امکان ایجاد و ویرایش این فایل‌ها را به راحتی می دهد.

بدانیم: IDLE پایتون همچنین چندین ویژگی مفید را ارائه می‌دهد که در های‌IDE حرفه‌ای مشاهده خواهید کرد، مانند برجسته کردن سینتکس اولیه، تکمیل کد و تورفتگی خودکار. های‌IDE حرفه‌ای نرم افزارهای قوی‌تری هستند و یادگیری آن‌ها زمان‌برتراست. اگر به تازگی سفر برنامه نویسی پایتون خود را آغاز کرده‌اید، IDLE Python یک جایگزین عالی است!

بدانیم: فایل‌های اجرایی در پایتون پایتون به طور عام دو نوع فایل اجرایی دارد

۱. فایل با پسوند *.py* (اسکریپت پایتون)

۲. فایل با پسوند *.ipynb* (نوتبوک ژوپیتر)

ما در ابتدا کلاس‌مان را با اسکریپت پایتون شروع می‌کنیم و پس از آن به پسوند دیگر خواهیم پرداخت.

چنانچه از قسمت *start* ویندوز $IDLE$ را جستجو کنید می‌توانید $PythonIDLE$ را بیابید و با کلیک بر آن شروع کنید. با شروع آن با صفحه‌ای مانند صفحه‌ی زیر مواجه خواهید شد:

```
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 03:13:28)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>>
```

Ln: 4 Col: 4

که در قسمت بالا نسخه‌ی پایتونی که نصب کرده‌ایم را می‌بینیم و در جایی که <<< قرار گرفته است می‌توانیم فرمان پایتون را وارد کنیم و با فشردن کلید *Enter* آن دستور را اجرا کنیم و نتیجه را ببینیم.

۱ـ۳ـ۳ برنامه Hello World !

معمولا اولین کدی که برنامه‌نویسان در شروع کار به اجرا می‌گذارند *HelloWorld*! است، برای این کار می‌توانید از دستور پایتون استفاده کنید و متن خود را بین "" قرار دهید.

به طور مثال:

```
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 03:13:28)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>> print("Hello, from IDLE!")
Hello, from IDLE!
>>>
```

Ln: 6 Col: 4

توجه: تابع print یک تابع از پیش ساخته شده در پایتون است که در قسمت‌های بعد بیشتر با آن آشنا خواهیم شد.

۱ـ۴ اسکریپت در پایتون

در محاسبات، کلمه اسکریپت برای اشاره به فایلی استفاده می‌شود که حاوی توالی منطقی از دستوراتی است که می‌بایست به طور منظم پردازش شوند.

اسکریپت‌ها همواره توسط نوعی مفسر [۲] ، پردازش می شوند که وظیفه اجرای هر دستور را به صورت متوالی بر عهده دارد.

یک فایل متنی ساده حاوی کد پایتون که قرار است مستقیماً توسط کاربر اجرا شود معمولاً **اسکریپت** نامیده می‌شود.

چنانچه فایل متنی ساده که حاوی کد پایتون است را برای وارد کردن و استفاده در یک فایل پایتون دیگر استفاده کنیم به آن **ماژول** می‌گویند.

[2] Interpreter

بنابراین، تفاوت اصلی بین یک ماژول و یک اسکریپت در این است که ماژول‌ها برای وارد شدن هستند، در حالی که اسکریپت‌ها برای اجرای مستقیم ساخته می شوند.

در هر صورت، نکته مهم این است که بدانیم چگونه کد پایتونی را که می‌نویسیم در ماژول‌ها و اسکریپت‌های خود اجرا کنیم.

۱ـ۴ـ۱ اجرای اسکریپت در IDLE

۱ـ۴ـ۲ تفاوت اسکریپت و shell

۱ـ۴ـ۳ توضیح یا کامنت

۱ـ۵ انواع داده در پایتون

اکنون که می‌دانیم چگونه با مفسر پایتون تعامل داشته باشیم و کد پایتون را اجرا کنیم وقت آن رسیده که زبان پایتون را بررسی کنیم. ابتدا به انواع داده‌های اساسی که در پایتون وجود دارند خواهیم پرداخت.

۱ـ۵ـ۱ اعداد صحیح (integers)

در پایتون ۳، عملاً هیچ محدودیتی برای اندازه‌ی یک مقدار صحیح وجود ندارد. البته، مانند همه چیزها به مقدار حافظه‌ای که سیستم شما دارد محدود می‌شود، اما بصورت کلی، یک عدد صحیح می تواند تا هر اندازه‌ای که شما نیاز دارید باشد:

```
>>>print(123123123123123123123123123123123123123123123123 + 1)
123123123123123123123123123123123123123123123124
```

در پایتون عددی را که هیچگونه پیشوندی ندارد را یک عدد طبیعی در مبنای ده در نظر می‌گیرد. مثلا پایتون به طور خودکار عدد 10 را عددی صحیح و در مبنای 10 در نظر می‌گیرد.

بدانیم: بطور مثال چنانچه قبل از عددمان از $0b$ استفاده کنیم عدد باینری خواهیم داشت.

عدد صحیح یا $integer$ را می‌توان براحتی و فقط با نوشتن یک عدد صحیح بدون ممیز در پایتون

تعریف کرد. نوع این عدد در پایتون با int مشخص می‌شود. به طور مثال چنانچه از تابع از پیش تعریف شده‌ی $type()$ استفاده کنیم و از پایتون بخواهیم نوع این مقدار را به ما بدهد به ما می‌گوید که عددمان int است.

```
>>> type(10)
 <class int>
```

۱ـ۵ـ۲ اعداد اعشاری

ما در ریاضیات با اعداد اعشاری [۳] شده‌ایم، عددی مانند 2.4 اعشاری می باشد. در پایتون نوع این اعداد با $float$ شناخته می‌شود. این اعداد نام نوع خود را از عدد نقطه شناور گرفته‌اند که بحث مفصلی دارد و در آینده سعی می‌شود به آن پرداخته شود. بطورکلی چیزی که الان نیاز است در مورد این اعداد بدانیم این است که هنگام استفاده یا معرفی اعداد با ممیز، پایتون متوجه خواهد شد و آن اعداد را به عنوان اعداد اعشاری خواهد شناخت.

توجه: در زبان پایتون ممیز اعداد با . نمایش داده می شود و نباید با / که در زبان فارسی استفاده می‌شود اشتباه شود.

مانند عدد صحیح که پیش از این به آن پرداختیم چنانچه نوع یک عدد اعشاری را با استفاده از تابع $type()$ بپرسیم، به ما $float$ را خواهد داد.

```
>>>type(10)
 <class float>
```

[3] Floating-Point Numbers

توجه: در پایتون می‌توان از زبان علمی نیز استفاده کرد، به این صورت که به اندازه‌ی اعشاری که نیاز داریم **توان** آن را بنویسیم. می‌توان این کار را با افزودن e یا E در آخر عدد نوشت.

```
>>>5e-324
5e-324
```

در عدد بالا در واقع عدد 5×10^{-324} را نوشته‌ایم.

۱ـ۵ـ۳ رشته

رشته، نویسه یا *string* یکی از انواع داده در پایتون است که به صورت یک نوشته می‌باشد. در زبان پایتون این نوع داده نوشته‌ای است که بین دو علامت نقل قول "" قرار می‌گیرد. منظور از نوشته تنها حروف نیست و می‌توان از اعداد و علایم ریاضی نیز استفاده کرد، در واقع حتی می‌توان از emoji نیز استفاده کرد.

بدانیم: emoji شکل‌های گرافیکی می‌باشند که توسط کدهایی به عنوان *unicode* در پایتون تعبیه شده‌اند.

برای چاپ یک رشته می‌توان به صورت زیر عمل کرد:

```
>>> print("Hello World")
Hello World
```

همانطور که می‌بینید برای چاپ عبارت *Hello World* ما آن عبارت را بین دو علامت نقل قول به صورت "*Hello World*" نوشتیم، با این کار پایتون آن عبارت را به عنوان *string* شناخت و توانست آن را چاپ کند. چنانچه بخواهیم ببینیم نوع عبارت *string* چیست می‌توانیم از تابع *type* استفاده کنیم که نتیجه‌ی آن *str* خواهد بود.

```
>>>type("10")
 <class str>
```

همانطور که می‌بینید تابع *type* عدد ۱۰ را که در بین علامت نقل قول است را *str* در نظر گرفت.

۱ـ۵ـ۴ نوع Boolean

یکی از انواع موجود در پایتون $Boolean$ است که در کل دو حالت $True$ یا $False$ را به خود می‌گیرد.

```
>>>type(True)
<class 'bool'>
>>>type(False)
<class 'bool'>
```

همانطور که در آینده خواهید دید، عبارات در پایتون اغلب در زمینه بولی ارزیابی می شوند، به این معنی که آنها برای نشان دادن درستی یا نادرستی تفسیر می‌شوند. مقداری که در بافت بولی درست است، گاهی اوقات گفته می‌شود که «درست» است، و مقداری که در بافت بولی نادرست است، «نادرست» است. در آینده با این نوع داده در پایتون بیشتر آشنا خواهید شد.

۱ـ۶ ورودی و خروجی (input/print)

برای اینکه یک برنامه مفید باشد، معمولا نیاز به دریافت داده‌های ورودی از کاربر و نمایش داده‌های نتیجه به کاربر دارد و بطورکلی با دنیای خارج ارتباط برقرار کند. در این بخش، با ورودی و خروجی پایتون آشنا خواهیم شد. برای دریافت ورودی راههای متفاوتی وجود دارد، این ورودی می‌تواند مستقیم، از طریق صفحه کلید یا از منابع خارجی مانند فایل‌ها یا پایگاه‌های داده وارد شود. خروجی را نیز می‌توان مستقیما در کنسول یا IDE ،از طریق رابط گرافیکی کاربر (GUI) روی صفحه نمایش یا اینکه در یک منبع خارجی نمایش داد.

۱ـ۶ـ۱ خواندن ورودی از صفحه کلید

برنامه‌ها اغلب نیاز به دریافت داده‌ها از کاربر، معمولا از طریق ورودی از صفحه کلید دارند.یکی از راه‌های انجام این کار در پایتون استفاده از تابع $input()$ می‌باشد:

```
>>>user_input = input()
salam
>>>user_input
```

```
'salam'
```

مثال بالا در *shell* اجرا شده است و خط >>> بیانگر کدی است که نوشته‌ایم و خط بدون >>> نشان‌دهنده‌ی نتیجه‌ی کد قبلی است که پس از فشردن کلید *Enter* نمایش داده می‌شود. چنانچه در آرگومان اختیاری تابع *input* نوشته‌ای قرار دهیم، هنگام اجرای کد پیام شما برای کاربر نمایش داده خواهد شد.

```
>>> user_input = input(" :adadi vared konid")
adadi vared konid: 2
user_input<<<
'2'
```

تابع *input* همواره رشته برمی‌گرداند، چنانچه بخواهیم نوع دیگری (مانند *int* ، *float*) بگیریم، می‌بایست *string* را به نوع مورد نظرمان تبدیل نماییم که به این عمل *casting* می‌گوییم.

```
>>> number = input("Enter a number: ")
Enter a number: 50
>>>print(number + 100)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
 TypeError: must be str, not int

>>>number = int(input("Enter a number: "))
Enter a number: 50
>>>print(number + 100)
150
```

در مثال بالا در ابتدا از کاربر عددی خواستیم و کاربر 50 را وارد کرد. اما وقتی خواستیم آن عدد را با 100 جمع کنیم با خطای *TypeError* مواجه شدیم که به ما می گوید نمی‌توان *str* را با *int* جمع نمود. برای رهایی از این خطا در کد بعدی *input* را در

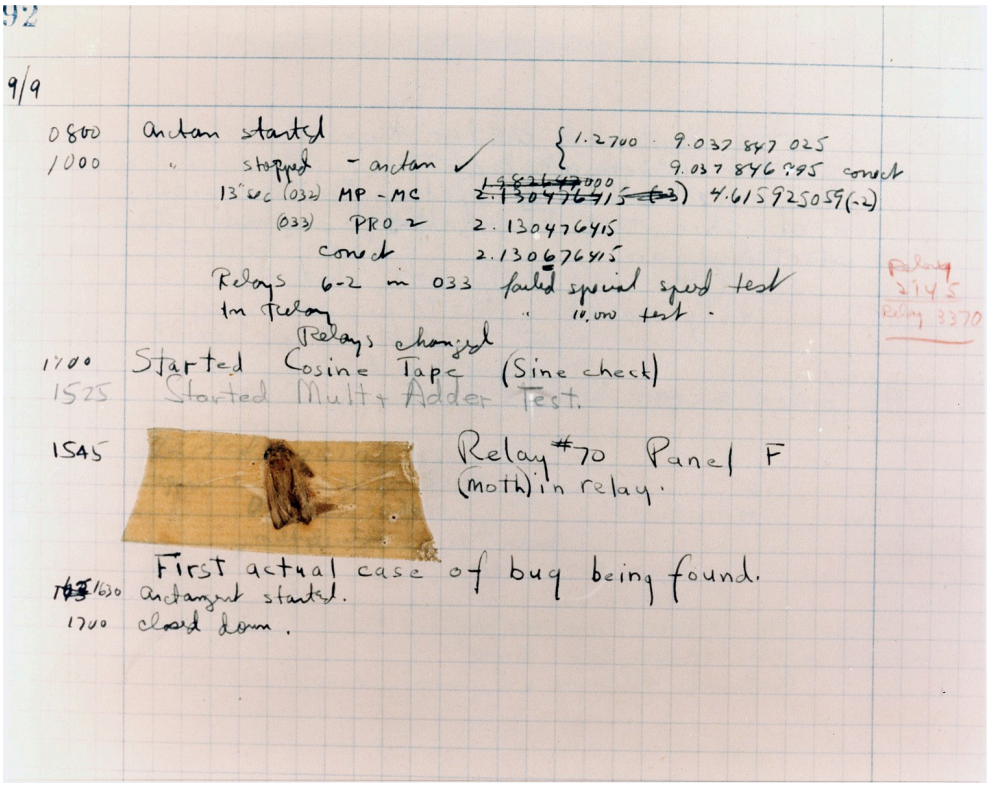
int قرار دادیم تا بتوانیم *string* را تبدیل به *int* نماییم. همانطور که می‌بینید با انجام این کار توانستیم با یک عدد جمع کنیم.

توجه: هنگامی که در کد ما خطایی رخ می‌دهد، پایتون به ما *Traceback*می‌دهد که در واقع مسیری است که در آن خطا رخ داده است. ما با دنبال کردن این مسیر از بالا به پایین می‌توانیم به منشا خطا برسیم، همچنین در آخر خطا همانطور که مشاهده می‌کنید نوع خطا نیز آورده شده است.

با مرور زمان و با دیدن خطاهای متفاوت، مهارت بیشتری در تشخیص مشکل در کد پیدا خواهید کرد.

بدانیم: به عیب‌یابی و یافتن مشکل کد در دنیای برنامه‌نویسی دیباگ (*debug*) می‌گویند که به معنی رفع خطا یا خارج کردن حشره می‌باشد!

کلمه‌ی *debug* به معنی رفع باگ (*bug*) است، که این کلمه خود به معنی حشره است و اولین بار هنگامی که حشره‌ای سبب مشکل پیدا کردن یکی از کامپیوترهای اولیه می‌شود وارد فرهنگ لغت برنامه نویسان می‌شود.



شکل ۱ـ۱: گزارشی از حشره‌ای که سبب ایراد یافتن کامپیوتر در سال ۱۹۴۵ می‌شود.

توجه:

تفاوت نسخه‌های پایتون:

اگر با کد $Python2.x$ کار می‌کنید، ممکن است با تفاوت جزئی در توابع ورودی بین نسخه‌های ۲ و ۳ پایتون مواجه شوید. $raw_input()$ در پایتون ۲ ورودی را از صفحه کلید می خواند و آن را برمی گرداند. $raw_input()$ در $Python2$ دقیقاً مانند $input()$ در $Python3$ رفتار می کند، همانطور که در بالا توضیح داده شد. اما پایتون ۲ تابعی به نام $input$ نیز دارد. در پایتون ۲، $input$ ورودی را از صفحه کلید می خواند، آن را به عنوان یک عبارت پایتون تجزیه و ارزیابی می کند و مقدار حاصل را برمی گرداند.

با تابع $input$ ما می‌توانیم از کاربر ورودی بگیریم، اما چنانچه بخواهیم عبارت یا مقداری را به کاربر نشان دهیم می‌توانیم از تابع $print$ استفاده کنیم.

۱ـ۶ـ۲ چاپ در صفحه shell با print

برای نمایش اشیاء روی صفحه‌ی $shell$، آنها را به صورت لیستی از آرگومان‌های جدا شده با کاما برای نمایش در $print$ قرار می‌دهیم. به این صورت:

```
print(<obj>, ..., <obj>)
```

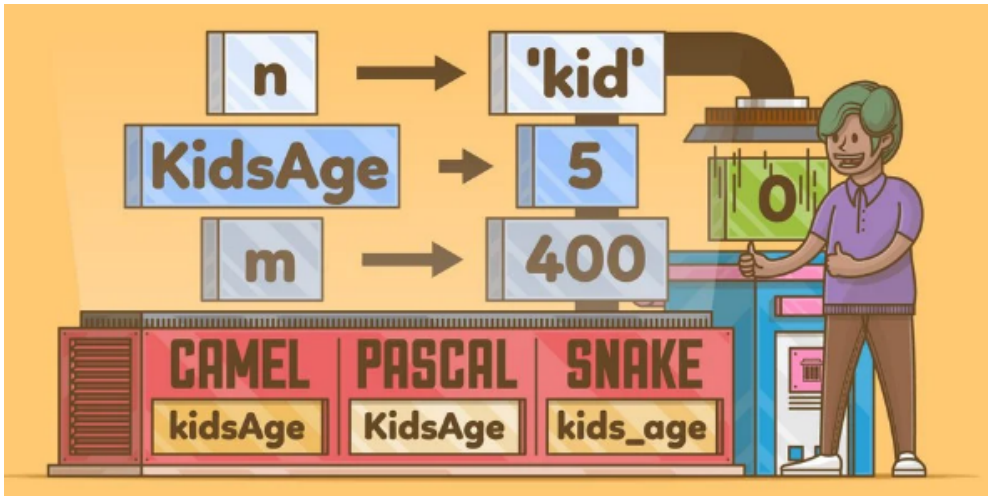
در کد بالا منظور از $< obj >$ چیزهایی است که می‌خواهیم به کاربر نشان دهیم.

توجه: وقتی از واژه‌ی آرگومان در تابع استفاده می‌کنیم منظورمان متغییرهایی است که یک تابع می‌پذیرد. مثلا در $print(x, y)$ آرگومان‌های تابع پرینت همان x ، y می‌باشند.

```
>>>"first_name = "ali
>>>"last_name = "mohammadi
>>>print("first name: ",first_name,"last name: ", last_name)
first name: ali last name: mohammadi
```

در کد قبل دیدیم که با قرار دادن $string$ و متغیرها در تابع $print$ می‌توانیم آن‌ها را در $shell$ ببینیم.

۱ـ۷ متغیرها



در بخش پیشین با انواع داده‌ها و روش نمایش آن‌ها در $shell$ آشنا شدیم، در این بخش می‌خواهیم با متغیرها و در واقع کاربرد داده‌های موجود در زبان پایتون صحبت کنیم.

ممکن است بپرسید که متغیر چیست و چه نیازی به آن داریم؟ می‌توان این سوال را اینگونه پاسخ داد که وقتی برنامه‌ی ما پیچیده می‌شود دیگر نمی‌توانیم از انواع داده‌ی اساسی استفاده کنیم و محاسبات ما به جای عدد و نوع اساسی به متغیرهایی نیاز خواهد داشت.

یک متغیر را به عنوان یک نام متصل به یک شی خاص در نظر بگیرید. در پایتون، برخلاف بسیاری از زبان‌های برنامه‌نویسی دیگر، متغیرها نیازی به تعریف یا تعریف از قبل ندارند. برای ایجاد یک متغیر، فقط یک مقدار به آن اختصاص می دهید و سپس شروع به استفاده از آن می کنید. تخصیص با یک علامت تساوی = انجام می شود:

```
>>>n = 300
```

کد بالا را می‌توان اینگونه تفسیر کرد: عدد ۳۰۰ را در n جایگذاری کردیم. از این پس می‌توانیم به جای عدد صحیح ۳۰۰ از n استفاده کنیم.

```
>>>print(n)
300
```

همانطور که می‌بینید هنگام چاپ n، عدد ۳۰۰ نمایش داده شد و این یعنی هر موقع از n استفاده کنیم

مقدار آن استفاده خواهد شد. چنانچه در $shell$ هستیم می‌توانیم n را بدون $print$ بنویسیم و دوباره مقدار آن نمایش داده شود.

```
>>>n
300
```

چنانچه پس از مقداردهی [۴] کردن عددی دوباره بیاییم با عدد یا نوع دیگری جایگذاری کنیم، عدد قبلی از بین می‌رود و عدد یا داده‌ی جدید جایگزین خواهد شد.

```
>>>n=200
>>>n
200
n = 750
>>>n
750
```

در پایتون همچنین می‌توان چند متغیر را به صورت زنجیره‌ای مقداردهی کرد.

```
>>>a = b = c = 300
>>>print(a, b, c)
300 300 300
```

در کد بالا سه متغیر a, b, c را با عدد ۳۰۰ مقداردهی کردیم و در نهایت هر سه را توسط تابع $print$ نمایش دادیم.

۱ـ۷ـ۱ نوع متغیرها

در بسیاری از زبان‌ها برنامه‌نویسی ما می‌بایست قبل از مقداردهی متغیر نوع آن را تعریف کنیم، اما در پایتون نیازی به این کار نیست. علاوه بر این نوع متغیرها در پایتون می‌تواند تغییر کند، مثلا در ابتدا مقداری که به متغیر می‌دهیم عدد صحیح باشد و در ادامه به آن عدد اعشاری یا حتی رشته بدهیم.

توجه: نوع رشته همان $string$ است که در پایتون به طور مخفف str نوشته می‌شود.

[4] assign

۱ـ۷ـ۲ ارجاع به اشیا در پایتون

وقتی یک متغیر را مقداردهی می‌کنیم چه اتفاقی می افتد؟ این یک سوال مهم در پایتون است، زیرا پاسخ آن تا حدودی با آنچه در بسیاری از زبان‌های برنامه‌نویسی دیگر پیدا می کنید متفاوت است. پایتون یک زبان بسیار **شی گرا** است. در واقع، تقریباً هر چیزی که در پایتون می‌بینید یک شی است که از یک نوع یا کلاس خاص است(در طول کلاس بیشتر به این نکته می‌پردازیم). این کد را در نظر بگیرید:

```
>>>print(300)
300
```

در این کد اتفاقی که می‌افتد به این صورت است که مفسر پایتون

۱. یک شی [۵] عدد صحیح [۶] می‌سازد

۲. به آن مقدار 300 را می‌دهد

۳. در صفحه نمایش می‌دهد

چنانچه این مراحل را دنبال کنیم می‌توانیم کارهایی که مفسر پایتون انجام می‌دهد را بهتر درک کنیم. برای مثال وقتی که عددی را می‌نویسیم، مفسر پایتون همواره یک نوعی را برای آن در نظر می‌گیرد. مثلا برای عدد 300 چنانچه نوع آن را بررسی کنیم می‌بینیم یک کلاس int برای آن تخصیص داده شده است.

```
>>>type(300)
<class 'int'>
```

حال متغیری که ما برای این اعداد و داده‌ها در نظر می‌گیریم در واقع نامی برای شی‌ای هستند که قبلا تعریف کرده‌ایم، یعنی وقتی می‌گوییم $n = 300$ اتفاقی که می‌افتد این است که یک شی‌ای در پایتون با نوع int ساخته می‌شود و برای این شی اسم n را گذاشته‌ایم که هر موقع بخواهیم به این شی اشاره کنیم می‌توانیم از این اسم استفاده کنیم.

کد زیر به ما نشان می‌دهد که متغیر n واقعا به آن شی اشاره دارد:

```
>>>print(n)
```

[5] object
[6] integer

```
300
>>>type(n)
<class 'int'>
```

که از اینجا به این نتیجه می‌رسیم که هنگام بررسی نوع n می‌بینیم نتیجه $classint$ است که همان شی است. حالا کد زیر را در نظر بگیرید:

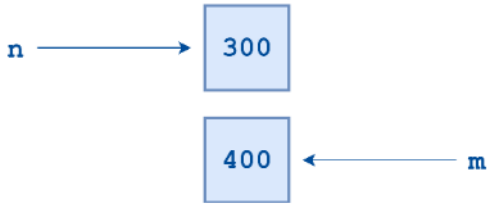
```
>>>m=n
```

اتفاقی که اینجا می‌افتد جالب است، پایتون شی جدیدی نمی‌سازد، بلکه یک شی درست می‌کند و دو اسم برای آن شی تعریف می‌کند، که این دو اسم هر دو به آن شی اشاره می‌کنند. حالا فرض کنید ما یکی از

متغیرها را عوض کنیم، آن وقت چه اتفاقی می‌افتد؟

```
>>>m=400
```

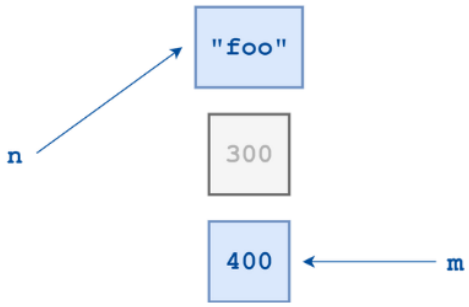
با این کار پایتون شی جدیدی برای 400 درست می‌کند و m را به آن ارجاع می‌دهد. حالا فرض کنید



که عبارت زیر اجرا شود:

```
>>>"n = "foo
```

حال پایتون شی جدید با مقدار "foo" می‌سازد و n را به آن ارجاع می‌دهد.



همانطور که می‌بینید الان یک مقدار بدون ارجاع داریم که راهی برای ارجاع به آن نداریم.

۱ـ۷ـ۳ مشخصه‌ی شی

در زبان برنامه‌نویسی پایتون برای هر شی‌ای که ایجاد می‌شود، یک عددی تعریف می‌شود که به صورت یکتا آن را تعریف می‌کند عدد مشخصه‌ی شی [۷] می‌باشد.

برای این که این عدد را ببینیم، می‌توانیم از تابع از پیش تعریف شده‌ی $id()$ استفاده کنیم. برای این که مقدار هر متغیر را بدست بیاوریم کافیست متغیری که داریم را به صورت زیر در تابع $id()$ استفاده کنیم.

```
>>>n = 300
>>>m = n
>>>id(n)
60127840
>>>id(m)
60127840

>>>m = 400
>>>id(m)
60127872
```

[7] Object Identity

در عبارت‌های بالا در ابتدا n را مقداردهی کردیم و پس از آن m را با n مقداردهی کردیم، بنابراین m و n هر دو به یک شی اشاره خواهند کرد و با نمایش id هر کدام می‌توانیم این را تایید کنیم(هر دو یک عدد برای id دارند چون به یک شی اشاره دارند).

۱ـ۷ـ۴ نام متغیرها

نمونه‌هایی که تاکنون دیدیم از نام‌های کوتاه و مختصر متغیری مانند m و n استفاده شده بود. اما نام متغیرها می تواند پرمعناتر باشد. در واقع، معمولاً بهتر است که نام معنا داشته باشد، چون اینگونه راحت‌تر می‌توان هدف و کار متغیر را متوجه شد. بطور کلی، نام متغیرها در پایتون می‌تواند هر طولی داشته باشد و می تواند

۱. از حروف بزرگ و کوچک $(A-Za-z)$

۲. ارقام $(0-9)$

۳. کاراکتر زیر خط _ $(underline)$

تشکیل شود. یک محدودیتی که وجود دارد این است که اگرچه نام متغیر می‌تواند دارای اعداد باشد، اما **اولین** کاراکتر نام متغیر نمی‌تواند عدد باشد. برای مثال نام‌های زیر معتبر هستند:

```
>>>"name = "Bob
>>>Age = 54
>>>has_W2 = True
>>>print(name, Age, has_W2)
Bob 54 True
```

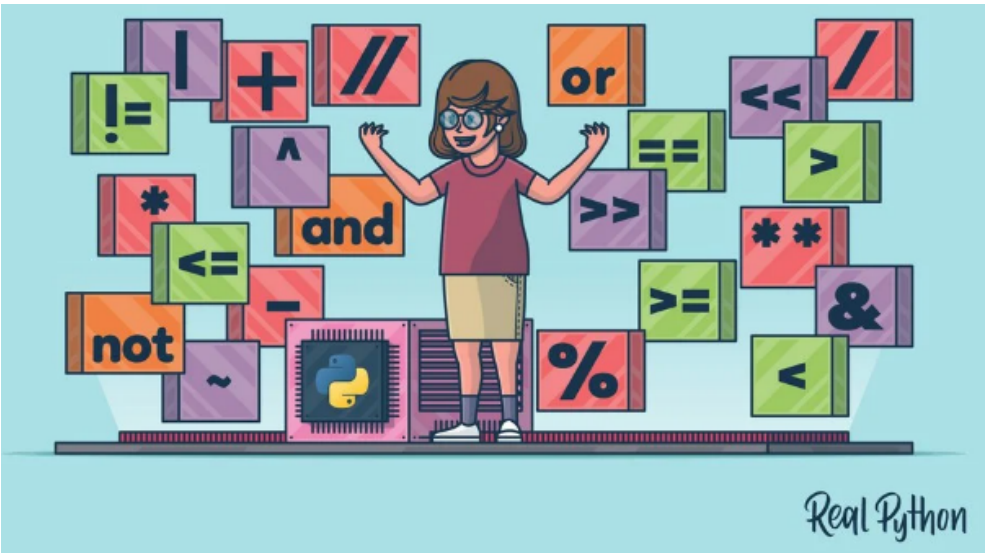
اما نام‌گذاری زیر درست نیست:

```
112w = 3
SyntaxError: invalid token
```

توجه: بزرگ و کوچک بودن حروف در نام متغیرها دارای اهمیت است و متغیر $E=1$با $e=1$ فرق دارد.

توجه: هنگامی دو متغیر را یکی با نام sky و دیگری را با نام SKY به طور همزمان ایجاد کنیم، از لحاظ نحوه نوشتار (سینتکس) پایتون هیچ ایرادی ندارد ولی بهتر است این کار را انجام ندهیم چون به عنوان برنامه نویس با مشکلات زیادی مواجه خواهیم شد.

۱ـ۸ عملگرها/عملوندها



حال که درک خوبی از متغیرها و انواع داده‌ها در پایتون پیدا کردیم، می‌توانیم به عملگرها اشاره کرد. در پایتون، عملگرها نمادهای خاصی هستند که نوعی محاسبات بر دو عملوند انجام می‌دهند. مقادیری که یک عملگر روی آنها عمل می کند عملوند نامیده می شود.

```
>>>a = 10
>>>b = 20
>>>a + b
30
```

در مثال بالا در ابتدا دو متغیر را مقداردهی کردیم و پس از آن آن دو متغیر را با استفاده از عملگر+ جمع کردیم که نتیجه جمع چاپ شد. می‌توانیم علاوه بر استفاده از یک عملگر از چند عملگر در یک خط کد استفاده کنیم:

```
>>>a = 10
>>>b = 20
>>>a + b – 5
25
```

۱ـ۸ـ۱ عملگرهای حسابی

از عملگرهای حسابی [۸] برای انجام محاسبات ریاضی مانند جمع، تفریق، ضرب و دیگر موارد استفاده می‌شود. در جدول زیر، عملگرهای حسابی رایج موجود در پایتون ارائه و عملکرد آن‌ها همراه با مثالی شرح داده شده است.

عملگر	شرح عملکرد	مثال
+	جمع کردن دو عمل‌وند یا عمل یگانی مثبت	y + x
–	تفریق عمل‌وند سمت راست از سمت چپی یا عمل یگانی منفی	y – x
*	ضرب دو عمل‌وند	y * x
/	تقسیم کردن عملوند سمت چپ بر سمت راستی	y / x
%	عملیات پیمانه‌ای (محاسبه باقی‌مانده تقسیم عملوند سمت چپ بر سمت راستی)	(باقی‌مانده x/y y % x)
//	خارج قسمت صحیح (این تقسیم، خارج قسمت صحیح را در خروجی ارائه می‌کند)	y // x
**	به توان y رساندن متغیر x	x**y (x به توان y شده)

جدول ۱ـ۱: جدول عملگرهای حسابی

مثال و نتیجه‌ی عمگرهای بالا را می‌توان در کد زیر مشاهده نمود:

```
>>>a = 4
>>>b = 3
>>>a + b
7
>>>a – b
1
>>>a * b
12
>>>a / b
1.3333333333333333
```

[8] Arithmetic Operators

```
>>>a % b
1
>>>a ** b
64
```

۱ـ۸ـ۲ عملگرهای مقایسه

در برنامه نویسی علاوه بر عملگرهای آشنای حسابی که در بخش پیش داشتیم نیاز است که متغیرها را با هم مقایسه کنیم، برای این کار می‌توان از عملگرهای مقایسه استفاده کرد. کاری که این عملگرها انجام می‌دهند این است که پاسخ مقایسه‌ی حاصل شده را برمی‌گردانند که معمولا $True$ یا $False$ است.

عملگر	شرح عملکرد	مثال
<	بزرگ‌تر است از (زمانی درست است که عمل‌وند سمت چپ بزرگ‌تر از سمت راستی باشد.)	>y x
>	کوچک‌تر است از (زمانی درست است که عمل‌وند سمت چپ کوچک‌تر از سمت راستی باشد.)	x<y
==	برابر است با (زمانی درست است که هر دو عمل‌وند برابر باشند.)	x==y
=!	نامساوی (زمانی درست است که هر دو عمل‌وند برابر نباشند.)	!=y x
=<	بزرگ‌تر یا مساوی (زمانی درست است که عمل‌وند سمت چپ، بزرگ‌تر یا مساوی سمت راستی باشد.)	x>=y
=>	کوچک‌تر یا مساوی (زمانی درست است که عمل‌وند سمت چپ، کوچک‌تر یا مساوی سمت راستی باشد.)	x<=y

جدول ۱ـ۲: جدول عملگرهای مقایسه

مثال‌های عملگرهای مقایسه:

```
>>>a = 10
>>>b = 20
>>>a == b
False
>>>a != b
True
>>>a <= b
True
>>>a >= b
False

>>>a = 30
```

```
>>>b = 30
>>>a == b
True
>>>a <= b
True
>>>a >= b
True
```

توجه: میان عملگر == و = تفاوت بنیادی وجود دارد که باید به آن توجه نمود.
عملگر = برای مقداردهی استفاده می‌شود و در عبارتی مانند $a = 2$ مقدار 2را در نام متغیر a قرار می‌دهد.
عملگر == برای مقایسه استفاده می‌شود و در عبارتی مانند $5 == 4$ بین دو عدد مقایسه انجام می‌دهد و اگر برابر بودند $True$ و اگر نابرابر بودند $False$ را برمی‌گرداند.

۱ـ۸ـ۳ عملگرهای منطقی

همانطور که در بخش پیش دیدیم، در بسیاری از اوقات نتیجه‌ی عملگرها $True$ یا $False$ است که در واقع $Boolean$ هستند. همچنین خیلی از اوقات نیاز داریم که ترکیبی از این هاBooleanها را داشته باشیم، برای انجام این ترکیب‌ها می‌توان از عملگرهای منطقی and، or و not استفاده کنیم.

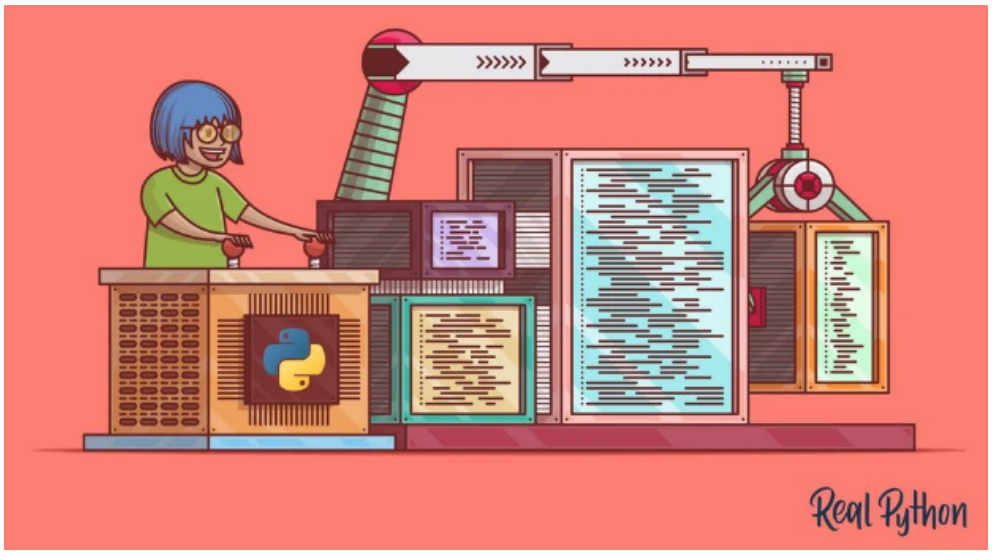
عملگر	**شرح عملکرد**	مثال
and	در صورتی که هر دو عمل‌وند درست (True) باشند، درست (True) است.	y and x
or	در صورتی درست (True) است که یکی از عمل‌وندها درست (True) باشد.	y or x
not	در صورتی درست (True) است که عمل‌وند غلط (False) باشد.	x not

جدول ۱ـ۳: جدول عملگرهای منطقی

```
>>>x = True
>>>y = False
>>>x and y
False
>>>x or y
True
```

```
>>>not x
False
```

۱ـ۹ برنامه‌نویسی مدولار



برنامه نویسی ماژولار یک تکنیک طراحی نرم افزاری است که بر اساس اصول کلی طراحی ماژولار(مدولار) است. طراحی ماژولار رویکردی است که حتی خیلی قبل از اولین کامپیوترها در مهندسی پدیدار شد. طراحی ماژولار به این معنی است که یک سیستم پیچیده به قطعات یا اجزای کوچکتر، یعنی ماژول‌ها تقسیم می شود. این اجزا را می توان به طور مستقل ایجاد و آزمایش کرد. در بسیاری از موارد، حتی می توان از آنها در سیستم‌های دیگر نیز استفاده کرد.

امروزه تقریباً هیچ محصولی وجود ندارد که بر ماژولارسازی متکی نباشد، مانند خودروها و تلفن های همراه. کامپیوترها متعلق به آن دسته از محصولاتی هستند که تا حد زیادی ماژولار شده‌اند. اگر می خواهید برنامه‌هایی را توسعه دهید که خوانا، قابل اعتماد و قابل نگهداری بدون تلاش زیاد باشند، می‌بایست از طراحی نرم افزار ماژولار استفاده کنید. به خصوص اگر برنامه شما اندازه خاصی داشته باشد. مفاهیم مختلفی برای طراحی نرم افزار به شکل ماژولار وجود دارد. برنامه نویسی ماژولار یک تکنیک طراحی نرم افزاری است که کد شما را به بخش‌های جداگانه تقسیم می‌کند. به این قطعات **ماژول** می گویند. تمرکز برای این جداسازی

باید داشتن ماژول‌هایی باشد که هیچ یا فقط وابستگی کمی به ماژول های دیگر ندارند. به عبارت دیگر: *به حداقل رساندن وابستگی‌ها هدف* است. هنگام ایجاد یک سیستم ماژولار، چندین ماژول به طور جداگانه و کم و بیش مستقل ساخته می شوند که برنامه اجرایی با کنار هم قرار دادن آنها ایجاد می‌شود.

۱ـ۹ـ۱ وارد کردن ماژول

تا اینجا ما توضیح ندادیم که ماژول پایتون چیست. به طور خلاصه: هر فایلی که پسوند فایل .py دارد و از کد پایتون مناسب تشکیل شده است، قابل مشاهده است یا یک ماژول است! برای تبدیل شدن چنین فایلی به یک ماژول، سینتکس خاصی لازم نیست. یک ماژول می تواند شامل اشیاء دلخواه باشد، برای مثال فایل‌ها، کلاس‌ها یا ویژگی‌ها [۹] را می‌توان در نظر گرفت. همه آن اشیا پس از وارد کردن قابل دسترسی هستند. راه‌های مختلفی برای وارد کردن یک ماژول وجود دارد. ما این را با ماژول ریاضی (**math**) نشان می دهیم:

```
import math
```

ماژول **math** ثابت‌ها و توابع ریاضی را ارائه می‌دهد، به عنوان مثال عدد پی (π) ($math.pi$)، تابع سینوس $math.sin()$ و تابع کسینوس .(‘()math.cos‘) هر ویژگی یا تابعی که در ماژول ‘math‘ وجود دارد را می‌توان با قرار دادن ‘math‘ در مقابل نام آن فراخوانی کرد:

```
math.pi

OUTPUT:  3.141592653589793

math.sin(math.pi/2)

OUTPUT:1.0

math.cos(math.pi/2)
OUTPUT:6.123233995736766e-17

math.cos(math.pi)
```

[9] attributes

```
OUTPUT: -1.0
```

می‌توان همزمان دو یا چند ماژول را وارد کرد:

```
import math, random
```

بدانیم: وارد کردن ماژول می‌تواند در هر قسمت از اسکریپت قرار بگیرد ولی بهتر است که در همان ابتدای فایل وارد کردن ماژول‌ها را قرار دهیم.

چنانچه تنها به بخش‌هایی از ماژول نیاز داریم می‌توانیم آن‌ها را فقط وارد کرد:

```
from math import sin, cos
```

گاهی اوقات نیاز داریم که همه‌ی بخش‌های یک ماژول را وارد کنیم که با این کار دیگر نیازی نیست نام ماژول را قبل از نام تابع مورد نظر بیاوریم. برای فراخوانی همه‌ی بخش‌های یک ماژول می‌توانیم از ستاره * استفاده کنیم، به صورت زیر:

```
from math import *
sin(3.01) + tan(cos(2.1)) + e
```

توجه: معمولا این کار توصیه نمی‌شود، چون ممکن است ماژول‌های متفاوت توابع با نام‌های یکسان داشته باشیم و این سبب نامعلوم شدن مرجع تابعی شود که وارد کردیم.

```
* from numpy import
* from math import
print(sin(3))
```

در مثال قبل هر دو ماژول $numpy$ و $math$ تابع sin را در خود دارند و با این کار ما در واقع sin ماژول $math$ را بکار برده‌ایم چون در آخر وارد شده است.

همچنین معمولا جامعه‌ی برنامه‌نویسان به طور قراردادی برای بعضی ماژول‌ها به صورت استاندارد از یک مخفف خاص استفاده می‌کنند.

```
import numpy as np
```

این کار یک قانون نانوشته است.

۱ـ۱۰ توابع از پیش تعریف شده

زبان برنامه‌نویسی پایتون توابع از پیش تعریف‌شده‌ی [۱۰] متعددی در خود دارد، تابع از پیش تعریف شده را می‌توان تابعی نامید که پایتون آن را می‌شناسد و نیازی نیست از جای دیگری وارد کنیم. نمونه‌هایی از این توابع را پیش از این در بخش‌های قبل دیدیم. توابع زیر از توابع پرکاربرد در پایتون هستند:

تابع	شرح عملکرد
input()	گرفتن ورودی از کاربر
print()	چاپ روی صفحه‌ی shell
len()	یافتن اندازه‌ی دنباله‌ای از مقادیر
int()	تابع عدد صحیح
bool()	تابع عدد Boolean
float()	تابع عدد اعشاری (شناور)
str()	تابع رشته (string)
max()	یافتن بزرگترین مقدار از دنباله‌ی اعداد
min()	یافتن کم‌ترین مقدار از دنباله‌ی اعداد
lis()	تابع لیست مقادیر (برای تعریف و cast کردن)
dict()	تابع دیکشنری (برای تعریف و cast کردن)
touple()	تابع چندتایی (تاپل) (برای تعریف و cast کردن)
set()	تابع مجموعه (برای تعریف و cast کردن)
sum()	یافتن مجموع دنباله‌ای از مقادیر
type()	نمایش نوع مقدار شی (یا داده‌ها)

جدول ۱ـ۴: جدول توابع از پیش تعریف شده‌ی پرکاربرد پایتون

توجه نمایید که این توابع فقط بخشی از توابع از پیش تعریف شده‌ی پایتون می‌باشند.

[10] built-in funcitons‘

۱ـ۱۱ تمرین‌های کلاسی

برنامه‌ای بنویسید که عددی از کاربر بگیرد، و بگوید که که آن عدد زوج است یا خیر؟

```
number = int(input("adadi vard namaeid!"))

zoj = (number%2 == 0)
print(zoj)
```

در کد بالا کاری که کردیم این بود که در ابتدا از کاربر عددی گرفتیم و پس از آن با % باقیمانده‌ی آن عدد بر ۲ را بررسی نمودیم، از آنجاییکه می‌دانیم چنانچه باقیمانده‌ی عددی بر ۲ صفر شود زوج خواهد بود پس همین کار را انجام دادیم و در نهایت صفر بودن باقیمانده را با عملوند == بررسی نمودیم، بنابراین چنانچه باقیمانده صفر باشد نتیجه‌ی ما $True$ خواهد بود و چنانچه غیر از آن باشد باقیمانده $False$ خواهد بود.

برنامه‌ای بنویسید که معادله‌ی درجه‌ی ۲ را حل کند.

```
import math

a = float(input("a:"))
b = float(input("b:"))
c = float(input("c:"))


delta = b**2-4*a*c
x1 = (-b +math.sqrt(delta))/2*a
x1 = (-b -math.sqrt(delta))/2*a
print(x1)
print(x2)
```

۱ـ۱۲ ساختار داده‌های متوالی

در زبان‌های برنامه‌نویسی معمولا نوعی از ساختار داده وجود دارد که در آن داده‌ها در مجموعه‌هایی کنار هم قرار می‌گیرند. این نوع ساختارهای متوالی عموما نوع داده بنیادی معرفی نمی‌کنند و در واقع بر اساس انواع داده‌هایی که تا حالا فراگرفتیم بنا می‌شوند.

هنگامی که معرفی ساختارهای جدید در برنامه‌نویسی شروع می‌شود معمولا سوالی که به ذهن دانشجویان می‌رسد این است که واقعا چه نیازی به وجود این ساختارها وجود دارد؟ البته که این یک سوال کاملا منطقی است و شاید هم در نگاه اول بتوان گفت که با یادگیری داده‌های بنیادی زبان عملا ما نیاز به ساختاری نداریم و یا حتی د صورت نیاز می‌توانیم آن‌ها را بسازیم! [۱۱] اما از آنجاییکه در برنامه‌نویسی معمولا توصیه می شود که خودمان را تکرار نکنیم، پس بهتر است ساختارهای مفیدی که در یک زبان به صورت استاندارد معرفی شده است را به خوبی فرا بگیریم و از آن‌ها استفاده کنیم.

بدانیم: پس از یاد گرفتن استفاده از ساختارهای زبان برنامه‌نویسی و یادگیری استفاده از آن‌ها، حتما باید الگوریتم و شیوه‌ای که آن ساختار در زبان برنامه‌نویسی ایجاد شده است دقت کنیم. چون از نظر ظاهر شاید به نظر یکسان باشند اما نکات ریزی وجود دارد که به بهینه شدن کد شما کمک شایانی خواهد کرد.

از ساختار داده‌هایی که در پایتون وجود داردد می‌توان به ساختارهای زیر اشاره کرد:

۱. لیست [۱۲]

۲. دیکشنری [۱۳]

۳. چندتایی [۱۴]

۴. مجموعه [۱۵]

از میان ساختارهای فوق **لیست** و **دیکشنری** بیشترین استفاده را دارند و از جهت قابلیت‌ها غنی‌تر و انعطاف پذیرتر می‌باشند. در این بخش به ساختارهای گفته شده می‌پردازیم.

[۱۱] که این امر ممکن است و ساختارهای جدیدی را در آینده از صفر خواهیم ساخت!

[12] List
[13] Dictionary
[14] Tuple
[15] Set

۱ـ۱۲ـ۱ لیست

یک لیست مجموعه‌ای از اشیاء دلخواه است که تا حدودی شبیه به یک آرایه در بسیاری از زبان‌های برنامه‌نویسی دیگر است، اما در پایتون لیست از انعطاف بیشتری برخوردار است. در پایتون می‌توان لیست را با قرار دادن دنباله‌ای از اشیاء جدا شده با کاما در براکت ([]) تعریف کرد، همانطور که در زیر نشان داده شده است:

```
>>> a = ['foo', 'bar', 'baz', 'qux']

>>> print(a)
['foo', 'bar', 'baz', 'qux']
>>> a
['foo', 'bar', 'baz', 'qux']
```

بدانیم: در دنیای برنامه‌نویسی کامپیوتر، "foo" و "bar" معمولا به عنوان مثالی از نام فایل‌ها، کاربران، برنامه‌ها، کلاس‌ها، هاست‌ها و غیره استفاده می‌شوند و معنایی ندارند.

ویژگی‌های مهم لیست‌های پایتون به شرح زیر است:

۱. لیست ها **ترتیب** دارند.

۲. لیست‌ها می‌توانند شامل هر شی دلخواهی باشند.

۳. می‌توان به مولفه‌های لیست با **اندیس** دسترسی پیدا کرد.

۴. می‌توان لیست‌ها را باعمق دلخواه به صورت **تودرتو** نوشت.

۵. لیست‌ها قابل تغییر هستند.

۶. لیست‌ها پویا [۱۶] هستند.

در ادامه هر یک از این ویژگی‌ها را بطور مفصل توضیح خواهیم داد.

[16] Dynamic

ترتیب لیست‌ها

یک لیست صرفا یک مجموعه‌ی از اشیاء نیست، بلکه مجموعه‌ای از اشیایی است که ترتیب دارند. این ترتیت هنگام تعریف لیست و مولفه‌های آن ایجاد می‌شود. هنگامی که ما یک مولفه را در ابتدا تعریف می‌کنیم در تمام طول عمر لیست جایگاهش همان اول خواهد بود و این یکی از ویژگی‌های ذاتی لیست است که ترتیب مولفه‌ها را حفظ می‌کند. (بعدا خواهید دید که ساختاری مانند دیکشنری ویژگی مرتب بودن را در خود ندارد.)

چنانچه ترتیب لیستی را بهم بزنیم در واقع یک لیست دیگری را ایجاد کرده‌ایم. می‌توان این نکته را در مثال زیر دید:

```
>>> a = ['foo', 'bar', 'baz', 'qux']
>>> b = ['baz', 'qux', 'bar', 'foo']
>>> a == b
False
>>> a is b
False
>>> [1, 2, 3, 4] == [4, 1, 3, 2]
False
```

لیست‌ها می‌توانند شامل اشیاء دلخواهی باشند

لیست‌ها می‌توانند شامل هر شی‌ای باشند، بطور مثال یک لیست می‌تواند شامل داده‌های با نوع یکسان باشد:

```
>>> a = [2, 4, 6, 8]
>>> a
[2, 4, 6, 8]
```

ولی لزومی بر یکسان بودن نوع داده‌ها نیست و یک لیست می‌تواند همزمان چندین نوع داده را داشته باشد:

```
>>> a = [21.42, "foobar", 3, 4, "bark", False, 3.14159]
>>> a
[21.42, "foobar", 3, 4, "bark", False, 3.14159]
```

لیست‌ها حتی می‌توانند شامل اشیاء پیچیده مانند توابع، کلاس‌ها و ماژول‌ها باشند که در بخش‌های بعدی با آنها آشنا خواهید شد:

```
>>> int
<class 'int'>
>>> len
<built-in function len>
>>> def foo():
...     pass
...
>>> foo
<function foo at 0x035B9030>
>>> import math
>>> math
<module 'math' (built-in)>

>>> a = [int, len, foo, math]
>>> a
[<class 'int'>, <built-in function len>, <function foo at 0x02CA2618>,
<module 'math' (built-in)>]
```

همچنین یک لیست می‌تواند هر تعداد شیء داشته باشد، از صفر تا هر تعداد که حافظه کامپیوتر شما اجازه می‌دهد.

```
>>> a = []
>>> a
[]

>>> a = [ 'foo' ]
>>> a
['foo']

>>> a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
```

```
    20,
... 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39,
    40,
... 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59,
    60,
... 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79,
    80,
... 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99,
    100]
>>> a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39,
40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58,
59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77,
78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96,
97, 98, 99, 100]
```

اشیای داخل نیست لزومی ندارد که یکتا و بدون تکرار باشند، بطور مثال:

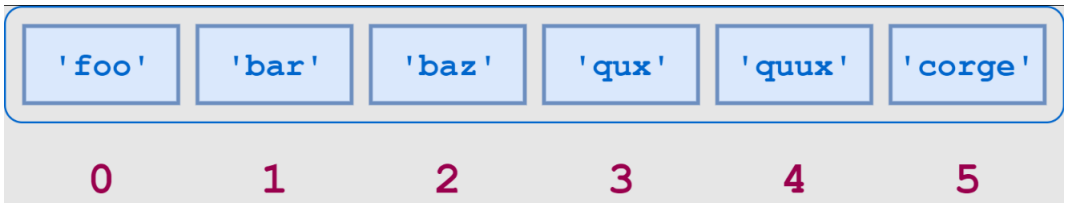
```
>>> a = ['bark', 'meow', 'woof', 'bark', 'cheep', 'bark']
>>> a
['bark', 'meow', 'woof', 'bark', 'cheep', 'bark']
```

می‌توان به مولفه‌های یک لیست با اندیس آن‌ها دسترسی پیدا کرد، همانطور که پیش از این در $string$ این را دیدیم، مولفه‌های لیست نیز هریک اندیس دارند که این اندیس از **صفر** شروع می‌شود.
لیست زیر را در نظر بگیرید:

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
```

اندیس‌های هر یک از مولفه‌های این لیست به صورت زیر می‌باشند:

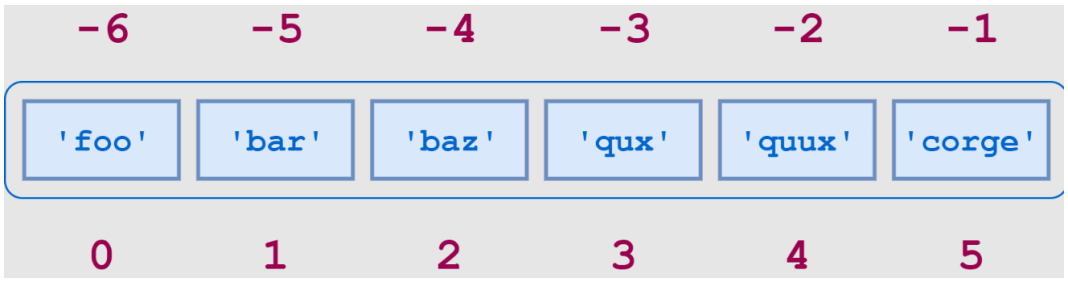
برای دسترسی به هر کدام از مولفه‌های لیست می‌توان اندیس هر مولفه را بین دو براکت به صورت $a[0]$ قرار داد، در اینجا a نام لیست و 0 اندیس مولفه‌ی مورد نظر ما می‌باشد.
در کد زیر می‌توان نحوه‌ی دسترسی به چند مولفه از لیستی که بالاتر تعریف کردیم را مشاهده کنیم:




```
>>> a[0]
'foo'
>>> a[2]
'baz'
>>> a[5]
'corge'
```

می‌توان گفت تقریبا هر آنچه در مورد اندیس‌های $string$ گفته شد برای اندیس‌های لیست نیز صادق است.

برای مثال می‌توانیم به مولفه‌های یک لیست از آخر آن لیست دسترسی پیدا کنیم، این کار را می‌توان با استفاده از اندیس منفی انجام داد. به این صورت که اندیس‌ها از سمت **راست** لیست با -1 شروع می‌شود.




```
>>> a[-1]
'corge'
>>> a[-2]
'quux'
>>> a[-5]
'bar'
```

همچنین می‌توان قسمت‌هایی از یک لیست را انتخاب کرد، برای این کار می‌بایست از $a[m, n]$ استفاده کرد، در این عبارت در واقع بخشی از لیست را انتخاب کردیم که در آن اندیس m شروع و اندیس n انتهای

آن بخش است(شامل مولفه‌ی با اندیس m نخواهد شد)، از این پس انتخاب بخشی از لیست را برش [۱۷] می‌نامیم.

توجه: هنگامی که با استفاده از $a[m,n]$ بخشی از لیست را انتخاب می‌کنیم، باید توجه کنیم که مولفه‌ی با اندیس n جزو انتخاب ما نخواهد بود و به گونه‌ای بازه‌ی باز است.

برای روشن‌تر شدن برش لیست مثال زیر را در نظر بگیرید:

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> a[2:5]
['baz', 'qux', 'quux']
```

همچنین می‌توان از اندیس‌های منفی و مثبت نیز استفاده کرد:

```
>>> a[-5:-2]
['bar', 'baz', 'qux']
>>> a[1:4]
['bar', 'baz', 'qux']
>>> a[-5:-2] == a[1:4]
True
```

در برش لیست:

۱. $a[:n]$.حذف اندیس **اول** به معنی شروع از اول لیست است

۲. $a[m:]$ حذف اندیس **آخر** به معنی انتخاب تا آخر لیست است.

برای مثال:

```
>>> print(a[:4], a[0:4])
['foo', 'bar', 'baz', 'qux'] ['foo', 'bar', 'baz', 'qux']
>>> print(a[2:], a[2:len(a)])
['baz', 'qux', 'quux', 'corge'] ['baz', 'qux', 'quux', 'corge']

```

[17] Slicing

```
>>> a[:4] + a[4:]
['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
>>> a[:4] + a[4:] == a
True
```

همچنین در برش لیست‌ها علاوه برا اندیس شروع و پایان می‌توان برای انتخاب مولفه‌ها **گام** [18] تعریف کرد، که سینتکس آن به صورت $a[start : stop : step]$ می‌باشد.. می‌توان گام را منفی یا مثبت انتخاب کرد.

```
>>> a[0:6:2]
['foo', 'baz', 'quux']
>>> a[1:6:2]
['bar', 'qux', 'corge']
>>> a[6:0:-2]
['corge', 'qux', 'bar']
```

در خط 1 کد بالا، ما شروع را اندیس 0 پایان را اندیس 6 و گام را 2 در نظر گرفتیم. با این کار در واقع از 0 تا 6 را در گام‌های دوتایی انتخاب کرده‌ایم. در خط 2 تنها شرو را تغییر دادیم و در خط سوم ما از اندیس ششم شروع کردیم و پایان را اندیس 0 در نظر گرفتیم، و با معرفی -2 به عنوان گام در واقع اندیس‌های $6, 4, 2$ را انتخاب کرده‌ایم.

حال می‌توان با استفاده از سینتکس برشی که تا کنون گفتیم کار دیگری نیز انجام داد، مثلا می‌توان مولفه‌های یک لیست را برعکس کرد:

```
>>> a[::-1]
['corge', 'quux', 'qux', 'baz', 'bar', 'foo']
```

در کد بالا با خالی گذاشتن اندیس شروع و پایان در واقع از اول تا آخر لیست را انتخاب کرده‌ایم و گام را -1 قرار داده‌ایم، با این کار ما به پایتون گفتیم که از آخر لیست به ما کل لیست را بدهد، که این همان لیست برعکس [19] است.

[18] stride
[19] Reversed

توجه: سینتکس [:] به نظر برای لیست و *string* یکسان است، اما تفاوتی بنیادی وجود دارد. چنانچه از [:] برای *string* استفاده کنیم در واقع ارجاعی به همان شی را به ما می‌دهد، در صورتی که استفاده از [:] برای لیست به ما کپی متفاوتی از شی را به ما می‌دهد.

بطور مثال برای *string* کد زیر را در نظر بگیرید:

```
>>> s = 'foobar'
>>> s[:]
'foobar'
>>> s[:] is s
True
```

و همان مثال را برای لیست ببینید:

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
>>> a[:]
['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
>>> a[:] is a
False
```

همانطور که دیدیم موقعی که لیست داریم و از [:] استفاده می‌کنیم، دیگر خود شی را نداریم و نسخه‌ی دیگری از آن داریم که این نکته‌ی مهمی است که بعدها باید به آن نیاز خواهیم داشت.

پیش از این در مورد توابع و عملگرهای از پیش تعریف شده در پایتون صحبت کردیم، می‌توانیم کاربرد آن‌ها را در لیست ببینیم.

برای مثال کاربرد عملگرهای *in* و *not in* را در مثال زیر می‌بینید:

```
>>> a
['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> 'qux' in a
True
>>> 'thud' not in a
True
```

همچنین تاثیر عملگرهای $+$ و $*$ بر لیست نیز جالب است:

```
>>> a
['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> a + ['grault', 'garply']
['foo', 'bar', 'baz', 'qux', 'quux', 'corge', 'grault', 'garply']
>>> a * 2
['foo', 'bar', 'baz', 'qux', 'quux', 'corge', 'foo', 'bar', 'baz',
'qux', 'quux', 'corge']
```

همانطور که میبینید، همانند $string$ ضرب و جمع در لیست‌ها به معنی ریاضیاتی نمی‌باشند و در جمع دو لیست به هم پیوند داده می‌شوند و در ضرب یک لیست تکرار می‌شود.

می‌توان کاربرد $len()$، $max()$ و $min()$ را نیز دید:

```
>>> a
['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> len(a)
6
>>> min(a)
'bar'
>>> max(a)
'qux'
```

توجه: توابع $min()$ و $max()$ تنها برای اعداد نمی‌باشند، برای همین در مثال بالا که لیستی از $string$ داشتیم تابع $min()$ بر اساس ترتیب حروف الفبا کوچکترین رشته را برگرداند. بطور مثال در رشته‌ی bar، اولین حرف از نظر ترتیب b و دومین حرف در ترتیب a و حال بین bar و baz حرف r کوچکتر از z می‌باشد.

بدانیم: اینکه لیست و $string$ شبیه هم هستند اتفاقی نیست و در واقعا این دو حالت‌های خاصی از نوعی به نام تکرارگرها [a] می‌باشند که در بخش‌های دیگر به طور مفصل در موردشان صحبت خواهیم کرد.

[a] iterable

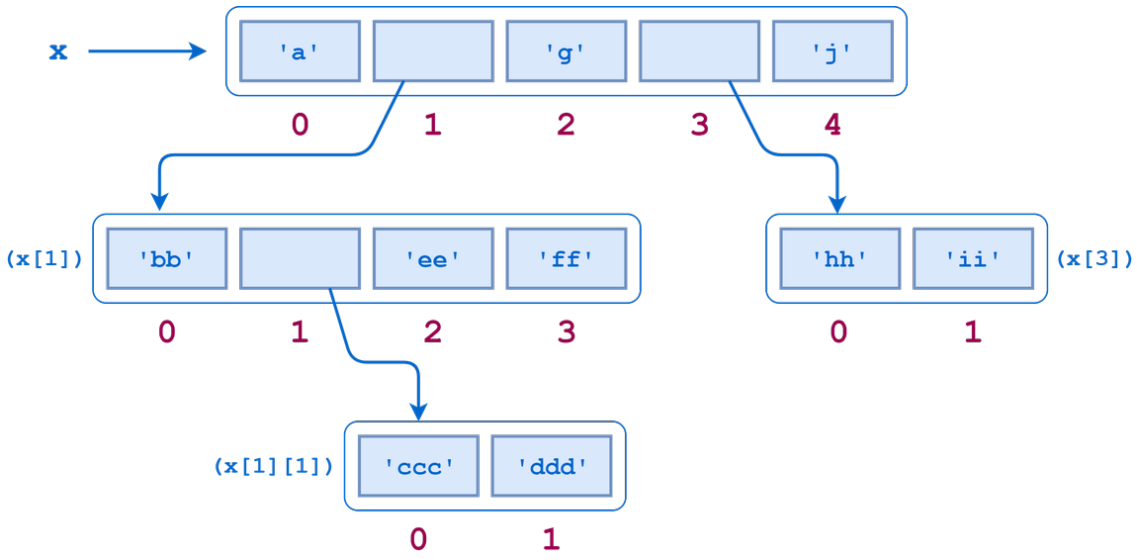
لیست‌ها می‌توانند تودرتو باشند

پیش از این گفتیم که لیست‌ها می‌توانند شامل هر شی‌ای باشند که این شامل خود لیست نیز می‌باشد. یعنی ما می‌توانیم در یک لیست، لیست دیگری معرفی کنیم.

لیست زیر را در نظر بگیرید:

```
>>> x = ['a', ['bb', ['ccc', 'ddd'], 'ee', 'ff'], 'g', ['hh', 'ii'], 'j']
>>> x
['a', ['bb', ['ccc', 'ddd'], 'ee', 'ff'], 'g', ['hh', 'ii'], 'j']
```

ساختار لیست x را می‌توان به صورت زیر رسم کرد:



$x[0]$، $x[2]$ و $x[4]$ رشته‌هایی هستند که هر کدام یک کاراکتر طول دارند:

```
>>> print(x[0], x[2], x[4])
a g j
```

اما $x[1]$ و $x[3]$ زیر لیست [20] هستند:

```
>>> x[1]
['bb', ['ccc', 'ddd'], 'ee', 'ff']

>>> x[3]
['hh', 'ii']
```

برای دسترسی به مولفه‌های موجود در یک زیرلیست، می‌توانیم اندیس دیگری اضافه کنیم:

```
>>> x[1]
['bb', ['ccc', 'ddd'], 'ee', 'ff']

>>> x[1][0]
'bb'
>>> x[1][1]
['ccc', 'ddd']
>>> x[1][2]
'ee'
>>> x[1][3]
'ff'

>>> x[3]
['hh', 'ii']
>>> print(x[3][0], x[3][1])
hh ii
```

اما $x[1][1]$ خود زیرلیست است، بنابراین با اضافه کردن یک اندیس دیگر می‌توانیم به عناصر آن دسترسی پیدا کنیم:

```
>>> x[1][1]
['ccc', 'ddd']
>>> print(x[1][1][0], x[1][1][1])
```

[20] Sublist

```
ccc ddd
```

می‌توان این کار را تا موقعی که حافظه‌ی کامپیوتر اجازه دهد ادامه دهیم و لیست‌های با عمق بیشتر بسازیم، منتها دسترسی به آن لیست‌ها و زیرلیست به همین شکل خواهد بود.

توجه: توابعی و عملگرهایی که بر لیست‌های تودرتو اعمال می‌کنیم تنها بر لایه‌ای که تعیین کرده‌ایم کار خواهند کرد و به صورت خودکار لیست عمق‌های دیگر را در نظر نمی‌گیرند، بطور مثال:

```
>>> x
['a', ['bb', ['ccc', 'ddd'], 'ee', 'ff'], 'g', ['hh', 'ii'], 'j']
>>> len(x)
5
```

در مثال بالا تابع $len()$ اندازه‌ی لیست را 5 گفت و کاری به دیگر زیرلیست‌های موجود در لیست نداشت.

لیست‌ها قابل تغییر هستند

تاکنون همه‌ی انواع داده‌ای که با آن‌ها مواجه شدیم تغییرناپذیر [۲۱] بودند، برای مثال، اعداد صحیح یا اعشاری، داده‌های ابتدایی هستند که نمی‌توان آنها را بیشتر از این شکست و کوچکتر کرد. این انواع تغییر ناپذیر هستند، به این معنی که پس از اختصاص دادن نمی‌توان آن‌ها را تغییر داد. البته فکر کردن به تغییر مقدار یک عدد صحیح چندان منطقی نیست. اگر یک عدد صحیح متفاوت می خواهید، می‌توانید عدد دیگری را اختصاص می‌دهید.

از جهت دیگر، نوع رشته از نوع **ترکیبی** است. رشته‌ها به قسمت‌های کوچکتر **قابل تقلیل** هستند (یعنی می‌توان آن‌ها با بخش‌های کوچکتر شکست). اما مولفه‌ها (یا کاراکترهای) رشته را نیز نمی‌توان تغییر داد. در این صورت رشته نیز تغییرناپذیر است.

این در حالتی است که لیست اولین نوع داده‌ی قابل تغییری است که با آن مواجه شدیم. پس از ایجاد یک لیست، مولفه‌های آن را می‌توان به دلخواه اضافه، حذف و جابجا کرد. پایتون طیف گسترده‌ای از راه‌ها را برای اصلاح لیست‌ها ارائه می‌دهد.

[21] Immutable

تغییر یکی از مولفه‌های لیست

یک مولفه از لیست را می‌توان با انتخاب توسط اندیس و مقداردهی آن تغییر داد:

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
>>> a
['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> a[2] = 10
>>> a[-1] = 20
>>> a
['foo', 'bar', 10, 'qux', 'quux', 20]
```

توجه: ما این کار را نمی‌توانستیم با رشته انجام دهیم و در صورت انجام این کار خطا می‌گرفتیم.

```
>>> s = 'foobarbaz'
>>> s[2] = 'x'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

همچنین یک مولفه از لیست را می‌توان با استفاده از کلیدواژه del حذف کرد.

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> del a[3]
>>> a
['foo', 'bar', 'baz', 'quux', 'corge']
```

در کد بالا وقتی $dela[3]$ را استفاده کردیم مولفه‌ی با اندیس 3 حذف شد و دیگر در لیست ما حضور ندارد.

تغییر همزمان چند مولفه‌ی لیست

چنانچه بخواهیم چندین مولفه‌ی یک لیست را به طور همزمان تغییر بدهیم، پایتون به ما این اجاز را می‌دهد که در البتدا اندیس‌هایی که می‌خواهیم تغییر دهیم را انتخاب کنیم (توسط برش لیست) و متغیرهای جدیدی ککه می‌خواهیم مقداردهی کنیم را در سمت راست قرار دهیم.
بنابراین به طور کلی باید مولفه‌ها را انتخاب کنیم و در یک سمت از تکرارگرها (مانند خود لیست) استفاده کرد.

```
a[m:n] = <iterable>
```

برای الان تکرارگرها را همان لیست فرض کنید، مقداردهی بالا در واقع بخشی که از لیست انتخاب کردیم را با تکراگر جایگزین می‌کند.

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> a[1:4]
['bar', 'baz', 'qux']
>>> a[1:4] = [1.1, 2.2, 3.3, 4.4, 5.5]
>>> a
['foo', 1.1, 2.2, 3.3, 4.4, 5.5, 'quux', 'corge']
>>> a[1:6]
[1.1, 2.2, 3.3, 4.4, 5.5]
>>> a[1:6] = ['Bark!']
>>> a
['foo', 'Bark!', 'quux', 'corge']
```

اگر خوب دقت کنید می‌بینید که اندازه‌ی برش لیست که انتخاب کردیم با تکرارگری که در سمت راست مساوی قرار دادیم برابر **نیست**، اما باز خطایی نگرفتیم، در واقع پایتون آن بخشی که برش دادیم را با تکرارگر جایگزین می‌کند و در صورت نیاز لیست را بزرگتر و یا کوچکتر می‌کند و نیازی به برابر اندازه‌های برش و تکرارگر **نیست**.
ما حتی می‌توانیم یک مولفه از لیست را با مجموعه‌ای از مولفه‌ها جایگزین کنیم.

```
>>> a = [1, 2, 3]
```

```
>>> a[1:2] = [2.1, 2.2, 2.3]
>>> a
[1, 2.1, 2.2, 2.3, 3]
```

باید توجه کرد که این حالت با موقعی که یک لیست را جایگزین می‌کنیم تفاوت دارد.

```
>>> a = [1, 2, 3]
>>> a[1] = [2.1, 2.2, 2.3]
>>> a
[1, [2.1, 2.2, 2.3], 3]
```

ما همچنین می‌توانیم بدون حذف مولفه‌ای، مولفه‌های متفاوتی را به لیست اضافه کنیم، برای این کار یک برش با اندازه‌ی صفر در جایی که می‌خواهیم انتخاب کنیم.

```
>>> a = [1, 2, 7, 8]
>>> a[2:2] = [3, 4, 5, 6]
>>> a
[1, 2, 3, 4, 5, 6, 7, 8]
```

برای حذف مولفه‌های مورد نظرمان از لیست می‌توانیم برشی از آن‌ها را انتخاب کنیم و با یک لیست خالی مقداردهی کنیم و یا اینکه آن برش را با استفاده از کلیدواژه del حذف نماییم، برای مثال:

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
>>> a[1:5] = []
>>> a
['foo', 'corge']

>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
>>> del a[1:5]
>>> a
['foo', 'corge']
```

افزودن مولفه به آخر یا اول لیست

همانطور که پیش از این دیدیم، ما می‌توانیم با استفاده از عملگر + دو لیست را به هم بچسبانیم [۲۲] ، حال چنانچه بخواهیم لیست دوم را به اول یا آخر لیست دوم اضافه کنیم کافی است هنگام استفاده از + ترتیب قرار گیری لیست‌ها را رعایت کنیم.

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> a += ['grault', 'garply']
>>> a
['foo', 'bar', 'baz', 'qux', 'quux', 'corge', 'grault', 'garply']

>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> a = [10, 20] + a
>>> a
[10, 20, 'foo', 'bar', 'baz', 'qux', 'quux', 'corge']
```

یادآوری: عبارت $a+=b$ در واقع معادل $a=a+b$ می‌باشد.

[22] concatenation

توجه: برای چسباندن شی‌ای به لیست حتما باید شی دوم نیز لیست باشد، ما نمی‌توانیم مثلا یک عدد به آن بچسبانیم.

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
>>> a += 20
Traceback (most recent call last):
  File "<pyshell#58>", line 1, in <module>
    a += 20
TypeError: 'int' object is not iterable

>>> a += [20]
>>> a
['foo', 'bar', 'baz', 'qux', 'quux', 'corge', 20]
```

همانطور که می‌بینید وقتی بخواهیم عددی را به لیست بچسبانیم با خطا مواجه $TypeError$ می‌شویم.

متدهایی که لیست را تغییر می‌دهند

حال به متدهایی که لیست را تغییر می‌دهند خواهیم پرداخت.

توجه: متدهای مورد استفاده در رشته‌ها را در بخش‌های قبل دیدیم، اما متدهای رشته‌ها با متدهای لیست تفاوت بنیادی دارند. متدهای رشته‌ها خود رشته‌ها را تغییر نمی‌دادند و فقط نسخه‌ی تغییر یافته‌ای از آن را برمی‌گردانند چون رشته‌های تغییرناپذیر هستند:

```
>>> s = 'foobar'
>>> t = s.upper()
>>> print(s, t)
foobar FOOBAR
```

اما متدهای لیست اینگونه نیستند و خود لیست را تغییر می‌دهند.

حال به متدهای لیست می‌پردازیم:

- **a.append(<obj>)**

یک شی را به انتهای لیست اضافه می‌کند.

```
>>> a = ['a', 'b']
>>> a.append(123)
>>> a
['a', 'b', 123]
```

هشدار: متدهای لیست، خود لیست را تغییر می‌دهند و چیزی را برنمی‌گردانند.

```
>>> a = ['a', 'b']
>>> x = a.append(123)
>>> print(x)
None
>>> a
['a', 'b', 123]
```

هنگامی که ما دو لیست را به هم می‌چسباندیم، نتیجه‌ی چسباندن دو لیست یک لیست بود.

```
>>> a = ['a', 'b']
>>> a + [1, 2, 3]
['a', 'b', 1, 2, 3]
```

اما در متد $append()$. ما باید مقدارها را یکی یکی اضافه کنیم و چنانچه یک لیست را اضافه کنیم، کل لیست به عنوان یه مولفه اضافه می‌شود.

```
>>> a = ['a', 'b']
>>> a.append([1, 2, 3])
>>> a
['a', 'b', [1, 2, 3]]
```

- **a.extend(<iterable>)**

این متد یک لیست را می‌گیرد و مولفه‌های آن لیست را به لیست‌مان اضافه می‌کند.

```
>>> a = ['a', 'b']
```

```
>>> a.extend([1, 2, 3])
>>> a
['a', 'b', 1, 2, 3]
```

در واقع $extend()$. مانند عملگر $+$ عمل می‌کند، یا به طور دقیق‌تر مانند $a+=b$ می‌باشد.

```
>>> a = ['a', 'b']
>>> a += [1, 2, 3]
>>> a
['a', 'b', 1, 2, 3]
```

- $a.insert(< index >, < obj >)$

این متد یک شی به اندیس مورد نظرمان اضافه می‌کند.

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
>>> a.insert(3, 3.14159)
>>> a[3]
3.14159
>>> a
['foo', 'bar', 'baz', 3.14159, 'qux', 'quux', 'corge']
```

- $a.remove(< obj >)$

این متد شی مورد نظرمان را از لیست حذف می‌کند، چنانچه آن شی در لیست نباشد خطایی به ما نشان خواهد داد.

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
>>> a.remove('baz')
>>> a
['foo', 'bar', 'qux', 'quux', 'corge']

>>> a.remove('Bark!')
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
```

```
    a.remove('Bark!')
ValueError: list.remove(x): x not in list
```

- $a.pop(index = -1)$

این متد شی با توجه به اندیس آن حذف می‌کند. اندیس -1 در تعریف این متد به این معناست که اگر هیچ اندیسی به آن ندهیم، آخرین اندیس را حذف خواهد کرد. بنابراین $a.pop()$ آخرین مولفه‌ی لیست را حذف خواهد کرد.

برای مثال چنانچه اندیسی ندهیم:

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> a.pop()
'corge'
>>> a
['foo', 'bar', 'baz', 'qux', 'quux']

>>> a.pop()
'quux'
>>> a
['foo', 'bar', 'baz', 'qux']
```

اما اگر به این متد اندیسی بدهیم، مولفه‌ی با آن اندیس را حذف خواهد کرد و مقدار حذف شده را برای نمایش برمی‌گرداند.

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> a.pop(1)
'bar'
>>> a
['foo', 'baz', 'qux', 'quux', 'corge']

>>> a.pop(-3)
```

```
'qux'
>>> a
['foo', 'baz', 'quux', 'corge']
```

لیست‌ها پویا هستند

تا الان پویایی لیست را در مثال‌های قبل دیدیم. هنگامی که می‌گوییم لیست‌ها پویا هستند به این معنا می‌باشد که در صورت لزوم بزرگتر می‌شوند:

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> a[2:2] = [1, 2, 3]
>>> a += [3.14159]
>>> a
['foo', 'bar', 1, 2, 3, 'baz', 'qux', 'quux', 'corge', 3.14159]
```

و گاهی اوقات کوچک می‌شوند:

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
>>> a[2:3] = []
>>> del a[0]
>>> a
['bar', 'qux', 'quux', 'corge']
```

تا اینجا با مهم‌ترین ویژگی‌های لیست و متدهای آن آشنا شدیم، در بخش‌های بعد با دیگر ساختارهای موجود در پایتون آشنا خواهیم شد.

۱ـ۱۲ـ۲ چندتایی

پایتون ساختار دیگری را نیز ارائه می‌کند که مجموعه‌ای مرتب از اشیاء است که به آن چندتایی یا تاپل [۲۳] می‌گویند.

[23] Tuple

تعریف و استفاده از تاپل

تاپل ها از همه جهات با لیست‌ها یکسان هستند، به جز ویژگی های زیر:

- تاپل‌ها با قرار دادن عناصر داخل پرانتز () به جای [] تعریف می شوند.
- تاپل‌ها تغییرناپذیرند

مثال زیر تعریف، اندیس‌دهی و برش تاپل‌ها را نشان می‌دهد:

```
>>> t = ('foo', 'bar', 'baz', 'qux', 'quux', 'corge')
>>> t
('foo', 'bar', 'baz', 'qux', 'quux', 'corge')

>>> t[0]
'foo'
>>> t[-1]
'corge'
>>> t[1::2]
('bar', 'qux', 'corge')
```

همچنین می‌توان مانند گذشته یک تاپل را با اندیس‌دهی برعکس کرد.

```
>>> t[::-1]
('corge', 'quux', 'qux', 'baz', 'bar', 'foo')
```

هرآنچه که تاکنون در مورد لیست‌ها یاد گرفتیم را می‌توان در تاپل‌ها به کار برد. اما تاپل‌ها برخلاف لیست‌ها تغییرناپذیرند و نمی‌توان مقدار آن‌ها را تغییر داد:

```
>>> t = ('foo', 'bar', 'baz', 'qux', 'quux', 'corge')
>>> t[2] = 'Bark!'
Traceback (most recent call last):
  File "<pyshell#65>", line 1, in <module>
    t[2] = 'Bark!'
TypeError: 'tuple' object does not support item assignment
```

حال ممکن است سوال کنید وقتی لیست می‌تواند همه‌ی کارها را انجام دهد چه نیازی به وجود تاپل می‌باشد؟

- اجرای برنامه هنگام استفاده از تاپل سریعتر از لیست است. (البته چنانچه لیست یا تاپل کوچک باشد تفاوت سرعت قابل توجه نخواهد بود.)
- گاهی اوقات نمی‌خواهیم داده‌ها اصلاح شوند. اگر قرار است مقادیر موجود در مجموعه در طول عمر برنامه ثابت بمانند، استفاده از تاپل به جای لیست از تغییرات تصادفی محافظت خواهد کرد.
- نوع دیگری از ساختار داده در پایتون به نام دیکشنری وجود دارد که به زودی با آن آشنا خواهیم شد، برای ایجاد این ساختار نیاز به استفاده از دوتایی‌های تغییرناپذیر است که تاپل‌ها در آن‌ها انتخاب خوبی می‌باشند، این در حالی است که نمی‌توان از لیست برای این منظور استفاده کرد.

وقتی چندین متغیر را در *shell* کنار هم قرار می‌دهیم، پایتون آن مقادیر را در پرانتز نشان می‌دهد چون آن‌ها را بصورت تاپل در نظر می‌گیرد.

```
>>> a = 'foo'
>>> b = 42
>>> a, 3.14159, b
('foo', 3.14159, 42)
```

یک ویژگی در مورد تعریف تاپل وجود دارد که باید از آن آگاه باشید. هیچ ابهامی در تعریف یک تاپل خالی وجود ندارد، و نه حتی یک تاپل با دو یا چند مولفه، چون پایتون می داند که شما در حال تعریف یک تاپل هستید:

```
>>> t = (1, 2)
>>> type(t)
<class 'tuple'>
>>> t = (1, 2, 3, 4, 5)
>>> type(t)
<class 'tuple'>
```

```
>>> a = 'foo'
```

```
>>> b = 42
>>> a, 3.14159, b
('foo', 3.14159, 42)
```

اما اگر بخواهیم تاپلی با یک مولفه را تعریف کنیم چه می‌شود؟

```
>>> t = (2)
>>> type(t)
<class 'int'>
```

عدد صحیح!! از آنجایی که از پرانتز برای تعریف اولویت عملگر در عبارات نیز استفاده می شود، پایتون عبارت (2) را صرفاً به عنوان عدد صحیح ۲ ارزیابی می کند و یک شی int ایجاد می‌کند. برای اینکه به پایتون بگویید که واقعاً می‌خواهید یک تاپل تک مولفه‌ای تعریف کنید، می‌بایست یک کاما , در انتهای مولفه‌ها درست قبل از پرانتز پایانی قرار دهید:

```
>>> t = (2,)
>>> type(t)
<class 'tuple'>
>>> t[0]
2
>>> t[-1]
2
```

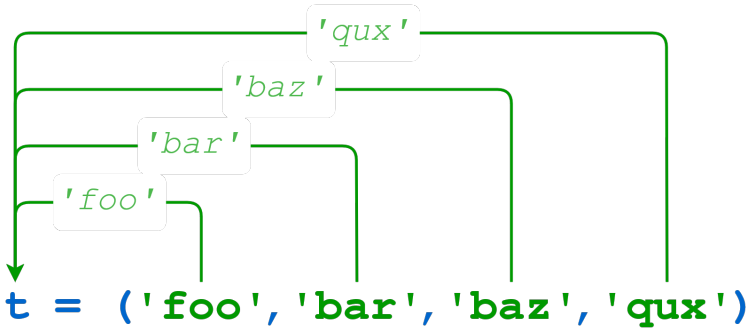
۱ـ۱۲ـ۲ـ۱ مقداردهی در تاپل

همانطور که پیش از این دیدیم یک تاپل می‌تواند شامل چندین مقدار باشد:

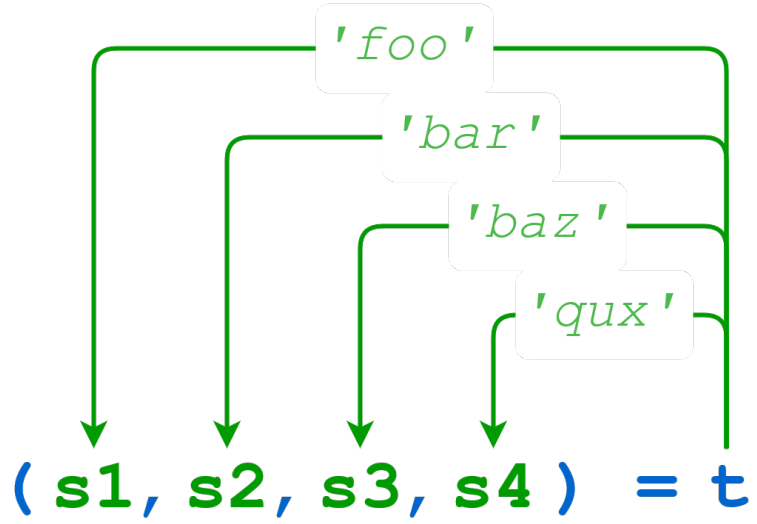
```
>>> t = ('foo', 'bar', 'baz', 'qux')
```

وقتی این اتفاق می‌افتد گویی چندین مقدار در یک شی بسته‌بندی شده‌اند.

```
>>> t
('foo', 'bar', 'baz', 'qux')
>>> t[0]
'foo'
```



اگر آن شی «بسته‌بندی شده» متعاقباً به یک تاپل جدید اختصاص داده شود، آیتم‌های جداگانه در اشیای تاپل «بازگشایی» می‌شوند:




```
>>> (s1, s2, s3, s4) = t
>>> s1
'foo'
>>> s2
'bar'
>>> s3
```

```
'baz'
>>> s4
'qux'
```

هنگام باز کردن بسته‌بندی تاپل، تعداد متغیرهای سمت چپ باید با تعداد مقادیر موجود در تاپل مطابقت داشته باشد:

```
>>> (s1, s2, s3) = t
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    (s1, s2, s3) = t
ValueError: too many values to unpack (expected 3)

>>> (s1, s2, s3, s4, s5) = t
Traceback (most recent call last):
  File "<pyshell#17>", line 1, in <module>
    (s1, s2, s3, s4, s5) = t
ValueError: not enough values to unpack (expected 5, got 4)
```

بسته بندی و باز کردن بسته بندی را می توان در یک عبارت ترکیب کرد تا یک تخصیص ترکیبی ایجاد شود:

```
>>> (s1, s2, s3, s4) = ('foo', 'bar', 'baz', 'qux')
>>> s1
'foo'
>>> s2
'bar'
>>> s3
'baz'
>>> s4
'qux'
```

مجدد یادآوری می‌کنیم که تعداد عناصر در سمت چپ با سمت راست باید یکسان باشد.

```
>>> (s1, s2, s3, s4, s5) = ('foo', 'bar', 'baz', 'qux')
Traceback (most recent call last):
  File "<pyshell#63>", line 1, in <module>
    (s1, s2, s3, s4, s5) = ('foo', 'bar', 'baz', 'qux')
ValueError: not enough values to unpack (expected 5, got 4)
```

برای کدهای ساده معمولا پایتون به ما اجازه می‌دهد که بدون استفاده از پرانتز تاپل را تعریف نماییم.

```
>>> t = 1, 2, 3
>>> t
(1, 2, 3)

>>> x1, x2, x3 = t
>>> x1, x2, x3
(1, 2, 3)

>>> x1, x2, x3 = 4, 5, 6
>>> x1, x2, x3
(4, 5, 6)

>>> t = 2,
>>> t
(2,)
```

در برنامه‌نویسی خیلی اوقات دو متغیر داریم که می‌خواهیم مقادیرشان را با هم جابجا کنیم، برای این کار در زبان‌های برنامه نویسی دیگر معمولا از یک متغیر موقت استفاده می‌شود تا این کار انجام شود:

```
>>> a = 'foo'
>>> b = 'bar'
>>> a, b
('foo', 'bar')

>>># We need to define a temp variable to accomplish the swap.
```

```
>>> temp = a
>>> a = b
>>> b = temp

>>> a, b
('bar', 'foo')
```

این کار در پایتون با استفاده از تاپل بسیار راحت‌تر انجام می‌شود:

```
>>> a = 'foo'
>>> b = 'bar'
>>> a, b
('foo', 'bar')

>>># Magic time!
>>> a, b = b, a

>>> a, b
('bar', 'foo')
```

اگر در مورد برنامه نویسی و تغییر مقدار متغیرها اطلاع دارید، حتما درک خواهید کرد که در پایتون این کار به زیبایی انجام شده است.

۱ـ۱۲ـ۳ دیکشنری

پایتون نوع دیگری از داده‌های متوالی به نام دیکشنری را ارائه می‌کند که مانند لیست مجموعه‌ای از اشیاء است. اما تفاوت‌های بنیادین بین این دو وجود دارد.

شباهت‌های لیست و دیکشنری:

- هر دو تغییر پذیر هستند.
- هر دو پویا هستند و می‌توانند در صورت لزوم بزرگ یا کوچک شوند.

- هر دو را می‌توان تودرتو کرد، یک لیست می‌تواند حاوی لیست دیگری باشد. یک دیکشنری می‌تواند دارای دیکشنری دیگری باشد. یک دیکشنری همچنین می‌تواند حاوی لیست باشد و بالعکس.

تفاوت‌های لیست و دیکشنری:

- نحوه‌ی دسترسی به عناصر در دیکشنری از طریق کلید آن‌ها است..
- نحوه‌ی دسترسی به عناصر لیست از طریق اندیس آن‌ها است.

تعریف دیکشنری

دیکشنری‌ها پیاده‌سازی پایتون از ساختار داده‌ای هستند که عموماً به عنوان آرایه انجمنی [۲۴] شناخته می شود. یک دیکشنری از مجموعه‌ای از جفت‌های کلیدـ مقدار تشکیل شده است. هر جفت کلیدـ مقدار کلید را به مقدار مرتبط خود نگاشت می‌کند.
می‌توان با قرار دادن فهرستی از جفت‌های کلیدـ مقدار جدا شده با کاما در کروشه یک دیکشنری تعریف کرد. یک کولون : هر کلید را از مقدار مربوط به آن جدا می کند:

```
d = {<key>: <value>, <key>: <value>, . . . <key>: <value>}
```

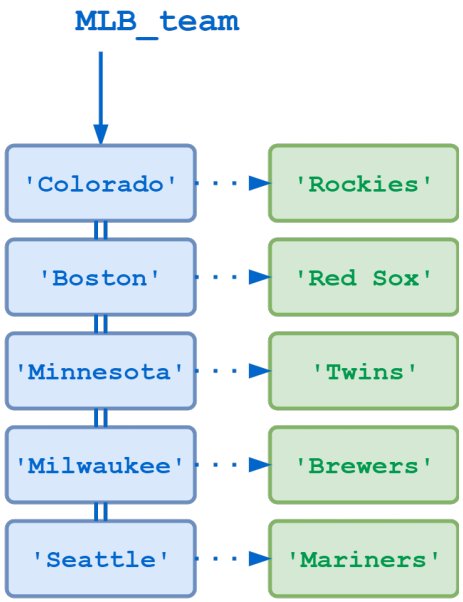
البته در پایتون می‌توان تعریف دیشکنری را به شکل زیر نیز نوشت که شکلی منظم‌تر از حالت قبل است.

```
d = {
    <key>: <value>,
    <key>: <value>,
      .
      .
      .
    <key>: <value>
}
```

در مثال زیر یک دیکشنری را تعریف کردیم که یک موقعیت مکانی را به نام تیم بیسبال لیگ برتر مربوطه خود نگاشت می‌کند:

[24] Associative array

```
>>> MLB_team = {
...       'Colorado' : 'Rockies',
...       'Boston'   : 'Red Sox',
...       'Minnesota': 'Twins',
...       'Milwaukee': 'Brewers',
...       'Seattle'  : 'Mariners'
... }
```



همچنین می توان با تابع از پیش تعریف شده‌ی $dict()$ یک دیکشنری ساخت. آرگومان $dict()$ باید دنباله‌ای از جفت‌های کلیدـمقدار باشد. لیستی از تاپل ها برای این کار خوب است:

```
d = dict([
    (<key>, <value>),
    (<key>, <value),
       .
       .
       .
    (<key>, <value>)
])
```

حال دیکشنری سابق را می‌توان به صورت زیر نیز تعریف کرد:

```
>>> MLB_team = dict([
...         ('Colorado', 'Rockies'),
...         ('Boston', 'Red Sox'),
...         ('Minnesota', 'Twins'),
...         ('Milwaukee', 'Brewers'),
...         ('Seattle', 'Mariners')
... ])
```

هنگامی که دیکشنری را تعریف شد، می‌توان آن را به صورت زیر نمایش داد:

```
>>> type(MLB_team)
<class 'dict'>

>>> MLB_team
{'Colorado': 'Rockies', 'Boston': 'Red Sox', 'Minnesota': 'Twins',
'Milwaukee': 'Brewers', 'Seattle': 'Mariners'}
```

دسترسی به مقادیر دیکشنری

برای دسترسی به مقادیر یک دیکشنری نمی‌توان از اندیس استفاده کرد. برای دسترسی به مقادیر موجود در دیکشنری می‌بایست از طریق کلید هر کدام از مقدارها به آن دسترسی پیدا کرد. این کار را می‌توان با گذاشتن کلید مقدار مورد نظر در [] انجام داد.

```
>>> MLB_team['Minnesota']
'Twins'
>>> MLB_team['Colorado']
'Rockies'
```

حال چنانچه از دیکشنری کلیدی را بخواهیم که در آن وجود ندارد، پایتون به ما خطا خواهد داد:

```
>>> MLB_team['Toronto']
Traceback (most recent call last):
  File "<pyshell#19>", line 1, in <module>
```

```
    MLB_team['Toronto']
KeyError: 'Toronto'
```

بدانیم: هنگامی که مانند مثال قبل پایتون خطایی به ما نشان می‌دهد در واقع این خطا نیست یک استثنا است که ممکن است در منابع دیگر ببینید برای اینگونه خطاها از اصطلاح *Exception* یا استثنا استفاده شده است.

حال برای اینکه به دیکشنری مقدار جدیدی اضافه کنیم، کافیست برای یک کلید مقدار آن را مقداردهی کنیم، مانند کاری که برای لیست انجام دادیم با این تفاوت که اینجا به جای اندیس از کلید استفاده می‌کنیم.

```
>>> MLB_team['Kansas City'] = 'Royals'
>>> MLB_team
{'Colorado': 'Rockies', 'Boston': 'Red Sox', 'Minnesota': 'Twins',
'Milwaukee': 'Brewers', 'Seattle': 'Mariners', 'Kansas City': 'Royals'}
```

برای حذف یک مولفه می‌توان از *del* استفاده کرد:

```
>>> del MLB_team['Seattle']
>>> MLB_team
{'Colorado': 'Rockies', 'Boston': 'Red Sox', 'Minnesota': 'Twins',
'Milwaukee': 'Brewers', 'Kansas City': 'Royals'}
```

باید توجه کرد که لزومی ندارد که کلیدهای دیکشنری حتما رشته باشند و می‌توان از اعداد صحیح نیز استفاده کرد:

```
>>> d = {0: 'a', 1: 'b', 2: 'c', 3: 'd'}
>>> d
{0: 'a', 1: 'b', 2: 'c', 3: 'd'}

>>> d[0]
'a'
>>> d[2]
'c'
```

توجه: کلیدهای دیکشنری که در مثال قبل داشتیم عدد صحیح بودند و در استفاده از کلیدهای عدد صحیح و اعشاری هیچ محدودیتی وجود ندارد، منتها باید توجه کنیم در مثال قبل اعداد صحیح کلید دیکشنری هستند و نباید با اندیس لیست اشتباه شود، مثلا اگر ترتیب کلیدـمقدارها را تغییر دهیم، کد ما تغییری نخواهد کرد.

```
>>> d = {3: 'd', 2: 'c', 1: 'b', 0: 'a'}
>>> d
{3: 'd', 2: 'c', 1: 'b', 0: 'a'}

>>> d[0]
'a'
>>> d[2]
'c'
```

همانطور که در کد بالا مشاهده می‌کنید با تغییر ترتیب همچنان مانند قبل می توانیم به مقادیردیکشنری با استفاده از کلیدهای آن ها دسترسی پیدا کنیم.

ساخت دیکشنری به صورت افزایشی

تعریف یک دیکشنری با استفاده از و لیستی از جفت‌های کلیدـمقدار موقعی مناسب است که از قبل همه کلیدها و مقادیر را داشته باشیم. اما اگر مولفه‌های یک دیکشنری را نداشته باشیم و می‌خواهیم یکی‌یکی آن‌ها را اضافه کنیم می‌بایست کار دیگری انجام دهیم. برای این کار می‌توان با ایجاد یک دیکشنری خالی شروع کرد سپس کلیدها و مقادیر جدید را یکی‌یکی اضافه کرد:

```
>>> person = {}
>>> type(person)
<class 'dict'>

>>> person['fname'] = 'Joe'
>>> person['lname'] = 'Fonebone'
>>> person['age'] = 51
```

```
>>> person['spouse'] = 'Edna'
>>> person['children'] = ['Ralph', 'Betty', 'Joey']
>>> person['pets'] = {'dog': 'Fido', 'cat': 'Sox'}
```

وقتی به این صوت یک دیکشنری را تعریف کردیم می‌توان مانند قبل به مولفه‌های آن دسترسی پیدا کرد:

```
>>> person
{'fname': 'Joe', 'lname': 'Fonebone', 'age': 51, 'spouse': 'Edna',
'children': ['Ralph', 'Betty', 'Joey'], 'pets': {'dog': 'Fido', 'cat': 'Sox'}}

>>> person['fname']
'Joe'
>>> person['age']
51
>>> person['children']
['Ralph', 'Betty', 'Joey']
```

محدودیت‌های کلیدهای دیکشنری

همانطور که پیش از این دیدیم عموما هر نوع داده‌ای را می‌توان به عنوان کلید برای دیکشنری استفاده کرد، بطور مثال اعداد صحیح و اعشاری و نوع بولی و رشته را می‌توان به کار برد:

```
>>> foo = {42: 'aaa', 2.78: 'bbb', True: 'ccc'}
>>> foo
{42: 'aaa', 2.78: 'bbb', True: 'ccc'}
```

ما حتی می‌توانیم توابع و نوع داده‌ها را به عنوان کلید استفاده کنیم:

```
>>> d = {int: 1, float: 2, bool: 3}
>>> d
{<class 'int'>: 1, <class 'float'>: 2, <class 'bool'>: 3}
>>> d[float]
2

```

```
>>> d = {bin: 1, hex: 2, oct: 3}
>>> d[oct]
3
```

اما بااینحال محدودیت‌هایی در تعریف کلیدهای دیکشنری وجود دارد.

- یک کلید تنها یک بار می‌تواند در دیکشنری ظاهر شود و اگر کلید تکراری تعریف کنیم مقدار آن جایگزین مقدار قبلی خواهد بود.
- کلید دیکشنری می‌بایست تغییرناپذیر باشد، برای همین **تاپل** را می‌توان به عنوان کلید تعریف کرد، اما **لیست** را نمی‌توان کلید در نظر گرفت.

بطور مثال برای تاپل:

```
>>> d = {(1, 1): 'a', (1, 2): 'b', (2, 1): 'c', (2, 2): 'd'}
>>> d[(1,1)]
'a'
>>> d[(2,1)]
'c'
```

بطور مثال برای لیست:

```
>>> d = {[1, 1]: 'a', [1, 2]: 'b', [2, 1]: 'c', [2, 2]: 'd'}
Traceback (most recent call last):
  File "<pyshell#20>", line 1, in <module>
    d = {[1, 1]: 'a', [1, 2]: 'b', [2, 1]: 'c', [2, 2]: 'd'}
TypeError: unhashable type: 'list'
```

همانطور که می‌بینید برای لیست با خطا مواجه شدیم.

محدودیت‌های مقادیر دیکشنری

برخلاف کلیدهای دیکشنری، هیچ محدودیتی برای تعریف مقادیر دیکشنری وجود ندارد. به معنای واقعی کلمه هیچ! یک مقدار دیکشنری می‌تواند هر شیای باشد که پایتون پشتیبانی می‌کند، از جمله نوع‌های تغییرپذیر مانند لیست‌ها و دیکشنری‌ها و یا اشیاء تعریف شده توسط کاربر، که در آینده بیشتر با آن‌ها آشنا

خواهیم شد.

همچنین هیچ محدودیتی برای تکرار مقدار در دیکشنری وجود ندارد:

```
>>> d = {0: 'a', 1: 'a', 2: 'a', 3: 'a'}
>>> d
{0: 'a', 1: 'a', 2: 'a', 3: 'a'}
>>> d[0] == d[1] == d[2]
True
```

عملگرها و توابع از پیش تعریف شده در دیکشنری

تاکنون با عملگرهایی که بر روی لیست و تاپل و رشته اعمال می شوند آشنا شدیم، بسیای از این عملگرها بر روی دیکشنری نیز کار می‌کنند.

برای مثال توابع in، $notin$ با توجه به موجود بودن کلید در دیکشنری به ما $True$ یا $False$ می‌دهد.

```
>>> MLB_team = {
...         'Colorado' : 'Rockies',
...         'Boston'   : 'Red Sox',
...         'Minnesota': 'Twins',
...         'Milwaukee': 'Brewers',
...         'Seattle'  : 'Mariners'
... }

>>> 'Milwaukee' in MLB_team
True
>>> 'Toronto' in MLB_team
False
>>> 'Toronto' not in MLB_team
True
```

تابع دیگری که معمولا استفاده می‌شود $len()$ می‌باشد که تعداد جفت کلیدـمقدار یک دیشکنری را به ما می‌دهد.

```
>>> MLB_team = {
...      'Colorado' : 'Rockies',
...      'Boston'   : 'Red Sox',
...      'Minnesota': 'Twins',
...      'Milwaukee': 'Brewers',
...      'Seattle'  : 'Mariners'
... }
>>> len(MLB_team)
5
```

متدهای از پیش تعریف شده‌ی دیکشنری

همانطور که در **لیست** و **رشته** متدهای از پیش تعریف شده‌ای داشتیم باری دیکشنری نیز متدهایی وجود دارد. اکثر متدهای موجود برای لیست با متدهای دیکشنری هم نام هستند.

- **d.clear()**

دیکشنری را خالی می‌کند.

```
>>> d = {'a': 10, 'b': 20, 'c': 30}
>>> d
{'a': 10, 'b': 20, 'c': 30}

>>> d.clear()
>>> d
{}
```

- $d.get(< key > [, < default >])$

مقدار یک کلید در دیکشنری را در صورت موجود بودن برمی‌گرداند.

پیش از این دیدیم که اگر کلیدی در دیکشنری نباشد و ما آن را صدا کنیم با خطا مواجه می شویم، اما با استفاده از متد $get()$. د صورت عدم وجود کلید $None$ برمی‌گرداند و خطا نمی‌دهد.

```
>>> d = {'a': 10, 'b': 20, 'c': 30}
```

```
>>> print(d.get('b'))
20
>>> print(d.get('z'))
None
```

حال چنانچه کلید در دیکشنری وجود نداشت و به جای $default$ مقداری را قرار داده باشیم به جای $None$ همان مقدار را برمی‌گرداند.

```
>>> print(d.get('z', -1))
-1
```

- $d.items()$

این متد لیستی از زوج‌های کلیدـمقدار را به صورت تاپل برمی‌گرداند:

```
>>> d = {'a': 10, 'b': 20, 'c': 30}
>>> d
{'a': 10, 'b': 20, 'c': 30}

>>> list(d.items())
[('a', 10), ('b', 20), ('c', 30)]
>>> list(d.items())[1][0]
'b'
>>> list(d.items())[1][1]
20
```

- $d.d.keys()$

این متد لیستی از کلیدهای دیکشنری را برمی‌گرداند:

```
>>> d = {'a': 10, 'b': 20, 'c': 30}
>>> d
{'a': 10, 'b': 20, 'c': 30}

```

```
>>> list(d.keys())
['a', 'b', 'c']
```

- $d.values()$

این متد لیستی از مقادیر دیکشنری را برمی‌گرداند:

```
>>> d = {'a': 10, 'b': 20, 'c': 30}
>>> d
{'a': 10, 'b': 20, 'c': 30}

>>> list(d.values())
[10, 20, 30]
```

- $d.pop(< key > [, < default >])$

می‌توان با این متد کلید موجود در دیکشنری را حذف نمود، در صورت وجود آن کلید مقدار آن را برمی‌گرداند:

```
>>> d = {'a': 10, 'b': 20, 'c': 30}

>>> d.pop('b')
20
>>> d
{'a': 10, 'c': 30}
```

- $d.popitem()$

این متد آخرین جفت کلیدـمقداری که به دیکشنری اضافه شده است را حذف می‌کند آن جفت را به صورت تاپل برمی‌گرداند.

```
>>> d = {'a': 10, 'b': 20, 'c': 30}

>>> d.popitem()
('c', 30)
```

```
>>> d
{'a': 10, 'b': 20}

>>> d.popitem()
('b', 20)
>>> d
{'a': 10}
```

اگر دیکشنری خالی باشد، متد $d.popitem()$ خطا خواهد داد:

```
>>> d = {}
>>> d.popitem()
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    d.popitem()
KeyError: 'popitem(): dictionary is empty'
```

- $d.update(< obj >)$

این متد شی دیگری را وارد دیکشنری می‌کند، این شی جدید می‌توان دیکشنری دیگر و یا جفت کلیدـمقدار به صورت تاپل باشد.

```
>>> d1 = {'a': 10, 'b': 20, 'c': 30}
>>> d2 = {'b': 200, 'd': 400}

>>> d1.update(d2)
>>> d1
{'a': 10, 'b': 200, 'c': 30, 'd': 400}
```

```
>>> d1 = {'a': 10, 'b': 20, 'c': 30}
>>> d1.update([('b', 200), ('d', 400)])
>>> d1
{'a': 10, 'b': 200, 'c': 30, 'd': 400}
```

۱ـ۱۳ ساختارهای شرطی

حال که به این بخش از آموزش رسیدیم با مطالب متفاوتی آشنا شدیم و دیدیم که عبارت‌های پایتون خط به خط اجرا می‌شوند، اما جهان ما معمولا به این صورت نیست و پیچیده‌تر است. در بسیاری از اوقات نیاز داریم که کارهایی را تکرار کنیم یا تحت شرایطی عبارتی ایجاد شود.

این همان جایی است که ساختار کنترلی ظاهر می‌شود، ساختار کنترلی بر اجرای بخش‌های برنامه‌ی ما نظارت دارد. در برنامه‌نویسی پایتون برای بکارگیری ساختار کنترلی از عبارت if استفاده می‌شود. عبارت if با توجه به عبارتی که به آن می‌دهیم اجازه‌ی اجرای دستورات را به برنامه می‌دهد.

در این بخش ابتدا، یک نمای کلی سریع از دستور **if** در ساده‌ترین شکل آن را کار خواهیم کرد.

پس از آن با بررسی دستور if خواهیم دید که برای ساختار کنترلی به گروه‌بندی یا بلوک‌بندی از عبارت‌ها نیازمند است. در نهایت، همه را به هم متصل می‌کنیم و یاد می‌گیریم که چگونه کد تصمیم‌گیری پیچیده بنویسیم. حال به ساختار عبارت شرطی نگاه می‌کنیم. ساده‌ترین حالت ساختارهای شرطی در پایتون به صورت زیر است:

```
1 if <expr>:
2     <statement>
```

در کدی که بالا می‌بینیم،

- عبارت $< expr >$ به صورت عبارت بولی ارزیابی می‌شود (به صورت $True$ یا $False$ که در بخش عملگرهای مقایسه دیدیم)
- عبارت $< statement >$ کد پایتون معمولی است که پیش از این نیز می‌نوشیم، با این تفاوت که در اینجا عبارتی که قرار است اجرا شود حتما با تورفتگی‌ای نسبت به if قرار می‌گیرد.

اگر عبارت $< expr >$ درست باشد (یعنی $True$)، عبارت پس از آن اجرا خواهد شد و اگر $False$ باشد، عبارت پس از آن اجرا نخواهد شد.

توجه: در عبارت بالا قرار دادن : ضروری است، در زبان‌های برنامه نویسی دیگر معمولا عبارت زیرنظر if توسط بسته می‌شود.

کد زیر مثال‌های متعدی از کاربرد if را نمایش می‌دهد:

```
>>> x = 0
>>> y = 5

>>> if x < y:                          # Truthy
...     print('yes')
...
yes
>>> if y < x:                          # Falsy
...     print('yes')
...

>>> if x:                              # Falsy
...     print('yes')
...
>>> if y:                              # Truthy
...     print('yes')
...
yes

>>> if x or y:                         # Truthy
...     print('yes')
...
yes
>>> if x and y:                        # Falsy
...     print('yes')
...

>>> if 'aul' in 'grault':              # Truthy
...     print('yes')
...
```

```
yes
>>> if 'quux' in ['foo', 'bar', 'baz']:  # Falsy
...     print('yes')
...
```

در عبارت‌های کد بالا جاهایی که شرط داخل if درست باشد نتیجه چاپ می‌شود و جایی که شرط داخل if نادرست باشد کد زیر if اجرا نمی‌شود و هیچ خروجی نخواهیم داشت.

۱ـ۱۳ـ۱ گروه‌بندی عبارات: بلوک‌بندی و تورفتگی

تا الان همه چیز خوب بود، اما فرض کنید که می‌خواهیم در هنگام درستی یک شرط بیش از یک عبارت را ارزیابی کنیم، چگونه باید این کار را انجام دهیم؟
برای ارزیابی بیش از یک عبارت زبان‌های برنامه‌نویسی معمولا یک بلوک که گروهی از عبارت‌ها می‌باشد. اما با اینکه همه‌ی زبان‌های برنامه‌نویسی این مفهوم را در خود دارند ولی طریقه‌ی استفاده یا سینتکس آن‌ها معمولا متفاوت است.

در پایتون همه چیز حول تورفتگی می‌چرخد !

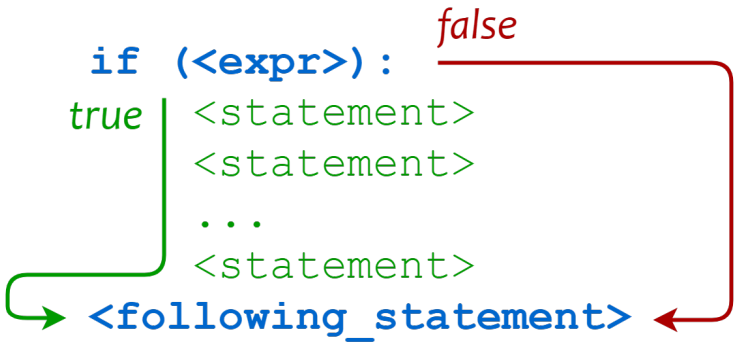
پایتون از قراردادی که به عنوان قانون آف‌ـساید شناخته می‌شود، پیروی می‌کند. این اصطلاح توسط دانشمند کامپیوتر بریتانیایی پیتر جی. لندین ابداع شده است [۲۵] . زبان‌هایی که به قانون آفساید پایبند هستند، بلوک‌ها را با تورفتگی تعریف می‌کنند. پایتون یکی از مجموعه نسبتاً کوچکی از زبان‌های قاعده آف‌ساید است.
برای همین است که در پایتون فاصله بسیار مهم است و این به دلیل اهمیت تورفتگی است. بنابراین از تورفتگی برای تعریف دستورات یا بلوک‌های ترکیبی استفاده می‌شود. در یک برنامه پایتون، عبارات پیوسته‌ای که در یک سطح فرورفته هستند، بخشی از یک بلوک در نظر گرفته می‌شوند.
بنابراین، یک دستور ترکیبی if در پایتون به شکل زیر خواهد بود:

```
if <expr>:
    <statement>
    <statement>
    ...
```

[۲۵] این اصطلاح از قانون آفساید در فوتبال گرفته شده است.

```
    <statement>
<following_statement>
```

در اینجا، تمام عباراتی که تورفتگی منطبقی دارند (خطوط ۲ تا ۵) بخشی از همان بلوک در نظر گرفته می‌شوند. اگر $< expr >$ درست باشد، کل بلوک اجرا می‌شود و اگر $< expr >$ نادرست باشد از آن می‌گذرد. در هر دو حالت، اجرای کد با $< following\_statement >$ (خط ۶) ادامه می‌یابد.



توجه: همانطور که می‌بینید در کد بالا هیچ نشانه‌ای برای پایان بلوک ندایم و همین که تورفتگی تمام می‌شود بلاک نیز تمام می‌شود.

مثال زیر را در نظر بگیرید:

```
if 'foo' in ['bar', 'baz', 'qux']:
    print('Expression was true')
    print('Executing statement in suite')
    print('...')
    print('Done.')
print('After conditional')
```

همانطور که می‌بینید به دلیل اینکه foo در لیست قرار ندارد و ما پرسیده بودیم که آیا foo در لیست وجود دارد یا نه و این عبارت به ما $Fasle$ را بر می‌گرداند. برای همین است که هیچ کدام از عبارت‌های درون شرط if اجرا نمی‌شوند.

همچنین می‌توان بلوک‌های ساختار شرطی پیچیده‌تری ایجاد نمود. مثلا با بکارگیری عبارت‌های شرطی

تودرتو (استفاده از if درون if دیگر) که if دوم خود زیرمجموعه‌ی if دوم باشد.

مثال زیر را ببینید:

```
# Does line execute?                    Yes    No
#                                       ---    --
if 'foo' in ['foo', 'bar', 'baz']:      #  x
    print('Outer condition is true')    #  x

    if 10 > 20:                         #  x
        print('Inner condition 1')      #        x

    print('Between inner conditions')   #  x

    if 10 < 20:                         #  x
        print('Inner condition 2')      #  x

    print('End of outer condition')     #  x
print('After outer condition')          #  x
```

توجه کنید در سمت راست مثال بالا $True$ و $False$ بودن هر کدام از شرطها آورده است.

نتیجه‌ی مثال بالا به صورت زیر است:

```
Outer condition is true
Between inner conditions
Inner condition 2
End of outer condition
After outer condition
```

۱ـ۱۳ـ۲ elif و else

اکنون که می‌دانیم چگونه از دستور if برای اجرای ساختار شرطی برای بلوکی از چندین دستور استفاده کنیم، وقت آن است که ببینیم چه کارهای دیگری می‌توان انجام داد.

گاهی اوقات، ما می‌خواهیم یک شرط را ارزیابی نماییم و اگر شرط درست بود یک مسیر را انتخاب کنیم، اما اگر درست نبود، مسیر جایگزین را مشخص کنیم. این کار را می‌توان با $else$ انجام داد:

```
if <expr>:
    <statement(s)>
else:
    <statement(s)>
```

اگر $< expr >$ درست باشد، مجموعه‌ی اول اجرا می‌شود و دومی **حذف** می‌شود. اگر $< expr >$ نادرست باشد، مجموعه‌ی اول **حذف** می‌شود و مجموعه‌ی دوم اجرا می شود. در هر صورت، اجرا پس از مجموعه دوم از سر گرفته می‌شود. همانطور که در بالا توضیح داده شد، هر دو مجموعه با تورفتگی تعریف می شوند.

در این مثال، x کوچکتر از ۵۰ است، بنابراین مجموعه‌ی اول (خطوط ۴ تا ۵) اجرا می شود و مجموعه‌ی دوم (خطوط ۷ تا ۸) حذف می‌شوند:

```
>>> x = 20

>>> if x < 50:
...     print('(first suite)')
...     print('x is small')
... else:
...     print('(second suite)')
...     print('x is large')
...
(first suite)
x is small
```

اما در مثال بعد، x بزرگتر از ۵۰ است، بنابراین مجموعه اول عبور داده می‌شود، و مجموعه دوم اجرا می شود:

```
>>> x = 120
>>>
```

```
>>> if x < 50:
...     print('(first suite)')
...     print('x is small')
... else:
...     print('(second suite)')
...     print('x is large')
...
(second suite)
x is large
```

همچنین دستوری برای اجرای انشعاب بر اساس چندین گزینه وجود دارد. برای این کار، از یک یا چند عبارت $elif$ (مخفف $elseif$) استفاده می‌کنیم. پایتون هر $< expr >$ را به نوبه خود ارزیابی می‌کند و مجموعه‌ی مربوط به اولین مورد درست را اجرا می‌کند. اگر هیچ یک از عبارات درست نباشد، و یک عبارت $else$ مشخص شده باشد، عبارت $else$ اجرا خواهد شد:

```
if <expr>:
    <statement(s)>
elif <expr>:
    <statement(s)>
elif <expr>:
    <statement(s)>
    ...
else:
    <statement(s)>
```

به طور کلی می‌توان به تعداد دلخواه $elif$ ایجاد کرد. اما بند $else$ اختیاری است و در صورت وجود، تنها مجاز به نوشتن یک مورد از آن هستیم که حتما باید در آخر مشخص شود:

```
>>> name = 'Joe'
>>> if name == 'Fred':
...     print('Hello Fred')
... elif name == 'Xander':
...     print('Hello Xander')
```

```
... elif name == 'Joe':
...     print('Hello Joe')
... elif name == 'Arnold':
...     print('Hello Arnold')
... else:
...     print("I don't know who you are!")
...
Hello Joe
```

۱ـ۱۳ـ۳ عبارت pass در پایتون

گاهی اوقات در برنامه‌نویسی نیاز داریم که قطعه‌ای از کدی را بنویسیم که الان فقط طرح کلی آن را در ذهن داریم و عبارتی را مدنظر نداریم، مثلا می‌دانیم که باید در یک جای مشخص یک عبارت شرطی قرار گیرد اما هیچ محتوایی برای آن در نظر نداریم، هب این کدها خرده کد می‌گوییم.
فرض کنید که می‌خواهیم عبارت شرطی if را ایجاد کنیم اما می‌خواهیم فعلا خالی باشد. چنانچه در پایتون این کار را انجام بدهیم با خطای تورفتگی مواجه خواهیم شد:

```
if True:

print('foo')
```

پیام خطا به صورت زیر خواهد بود:

```
C:\Users\john\Documents\Python\doc>python foo.py
  File "foo.py", line 3
    print('foo')
        ^
IndentationError: expected an indented block
```

دستور $pass$ این مشکل را حل می‌کند و به هیچ وجه رفتار برنامه را تغییر نمی‌دهد. این عبارت به عنوان یک مکان نگهدار برای راضی نگه داشتن مفسر در هر موقعیتی که یک عبارت از نظر نحوی مورد نیاز است استفاده می‌شود.

```
if True:
    pass

print('foo')
```

اکنون برنامه‌ی ما بدون خطا اجرا خواهد شد.

۱ـ۱۴ حلقه‌ها در پایتون

تکرار [۲۶] به این معنی است که یک بلوک کد را بارها و بارها اجرا کنیم. ساختار برنامه‌نویسی که تکرار را پیاده‌سازی می کند حلقه نامیده می شود.

در برنامه نویسی دو نوع تکرار وجود دارد، تکرار **نامعین** و تکرار **معین**:

- در تکرار نامعین، تعداد دفعاتی که حلقه اجرا می‌شود، از قبل به صراحت مشخص نشده است. در عوض، بلوک تعیین شده به طور مکرر اجرا می شود تا زمانی که برخی از شرایط برآورده شود.
- در تکرار معین، تعداد دفعاتی که بلوک حلقه اجرا می‌شود، در زمان شروع حلقه به صراحت مشخص می‌شود.

۱ـ۱۴ـ۱ حلقه‌ی while

بیایید ببینیم که چگونه از دستور $while$ پایتون برای ساخت حلقه‌ها استفاده می‌شود. فرمت یک حلقه $while$ در مثال زیر نشان داده شده است:

```
while <expr>:
    <statement(s)>
```

$< statement(s) >$ نشان دهنده بلوکی است که باید به طور مکرر اجرا شود که اغلب به عنوان بدنه‌ی حلقه از آن یاد می‌شود. همانطور که مشاهده می‌کنید بدنه‌ی حلقه با تورفتگی نشان داده می شود، درست مانند ساختاری است که در عبارت if دیدیم.

[26] iteration

توجه: همانطور که پیش از این گفته شد، تمام ساختارهای کنترلی از تورفتگی استفاده می‌کنند.

معمولا بدنه‌ی حلقه‌ی $while$ شامل یک یا چند متغیر است که قبل از شروع حلقه مقداردهی اولیه می‌شوند و سپس در جایی در بدنه حلقه اصلاح می شوند.

هنگامی که با یک حلقه $while$ مواجه می‌شویم، ابتدا عبارت $< expr >$ از نظر بولی ارزیابی می شود($True$ یا $False$ بودن)، اگر درست باشد، بدنه‌ی حلقه اجرا خواهد شد. سپس $< expr >$ دوباره بررسی می‌شود و اگر هنوز درست باشد، بدنه دوباره اجرا می‌شود. این کار تا زمانی ادامه می‌یابد که عبارت $< expr >$ مقدار $False$ شود، در این مرحله حلقه‌ی $while$ اصطلاحا تمام می‌شود و اجرای برنامه به بعد از تورفتگی می‌رود.

حلقه‌ی زیر را در نظر بگیرید:

```
>>> n = 5
>>> while n > 0:
...     n = n - 1
...     print(n)
...
4
3
2
1
0
```

آنچه در این مثال اتفاق می‌افتد به شرح زیر است:

n در ابتدا 5 است. عبارت درون شرط دستور while در خط ۲ عبارت n > ۰ است که درست است، بنابراین بدنه حلقه اجرا می‌شود. در داخل بدنه‌ی حلقه در خط ۳، یک عدد از n کم می‌شود و مقدار آن از ۵ به ۴ کاهش می یابد و سپس چاپ می‌شود. وقتی بدنه‌ی حلقه به پایان رسید، اجرای برنامه به بالای حلقه در خط ۲ برمی‌گردد و عبارت دوباره ارزیابی می‌شود که هنوز هم درست است، بنابراین بدنه دوباره اجرا می شود و ۳ چاپ می شود. این عمل تا زمانی ادامه می یابد که n به ۰ تبدیل شود. در آن نقطه، زمانی که عبارت ارزیابی می‌شود، نادرست است، و حلقه تمام می‌شود. اجرا در اولین عبارت پس از بدنه حلقه از سر

گرفته می شود، اما در این مثال چیزی وجود ندارد.

توجه: عبارت کنترل کننده‌ی حلقه $while$ ابتدا قبل از هر اتفاق دیگری ارزیابی می‌شود. اگر در همان ابتدا اشتباه باشد، بدنه‌ی حلقه هرگز اجرا نخواهد شد:

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

در مثال بالا، وقتی با حلقه مواجه می‌شویم، n برابر 0 است. عبارت کنترلی $n > 0$ از قبل نادرست است، بنابراین بدنه حلقه هرگز اجرا نمی‌شود.

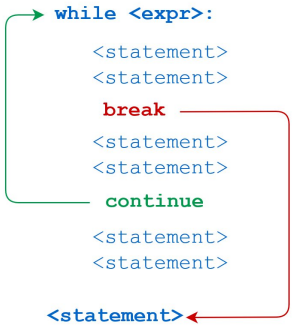
عبارت‌های break و continue در حلقه‌ی while

در همه‌ی مثالهایی که تاکنون دیدیم، کل بدنه‌ی حلقه while در هر تکرار اجرا می شود. پایتون دو کلمه کلیدی را ارائه می‌دهد که تکرار حلقه را زودتر از موعد متوقف می کنند:

دستور **break** در پایتون بلافاصله یک حلقه را به طور کامل خاتمه می‌دهد. اجرای برنامه به اولین عبارت پس از بدنه‌ی حلقه ادامه می‌یابد.

دستور **continue** اجرای عبارت‌های حلقه را متوقف می‌کند و اجرا را به بالای حلقه می‌برد تا عبارت کنترل‌کننده دوباره ارزیابی شود تا مشخص شود که آیا حلقه دوباره اجرا شود یا خاتمه یابد.

تمایز بین شکستن و ادامه دادن در نمودار زیر نشان داده شده است:



کد زیر استفاده از عبارت *break* را نشان می‌دهد:

```
n = 5
while n > 0:
    n = n - 1
    if n == 2:
        break
    print(n)
print('Loop ended.')
```

چنانچه اسکریپت بالا را اجرا کنیم خروجی زیر را نشان خواهد داد:

```
4
3
Loop ended.
```

در کد بالا در ابتدا n برابر 5 بود و هنگامی که وارد حلقه‌ی *while* می‌شود پس از هر تکرار یک عدد از آن کم می‌شود. اما اگر به کد توجه کنید می‌بینید که درون *while* یک شرط وجود دارد که اگر عدد برابر 2 شد درست خواهد شد و در نتیجه حلقه را می شکند و خارج می‌شود. برای همین است که فقط اعداد 4 و 3 را در نتیجه می‌بینیم.

حال کدی مانند کد قبل را برای عبارت *continue* می‌نویسیم:

```
n = 5
while n > 0:
    n = n - 1
    if n == 2:
        continue
    print(n)
print('Loop ended.')
```

که خروجی آن به صورت زیر می‌باشد:

```
4
3
1
```

```
0
Loop ended.
```

این بار، زمانی که n برابر 2 باشد، عبارت *continue* باعث خاتمه آن تکرار می‌شود. بنابراین، ۲ چاپ نمی‌شود. اجرا به بالای حلقه برمی گردد، شرط مجدداً ارزیابی می شود و همچنان درست است. حلقه از سر گرفته می‌شود و زمانی که n مانند قبل 0 شود به پایان می‌رسد.

حلقه‌ی تودرتو while

به طور کلی، ساختارهای کنترل پایتون را می توان درون یکدیگر قرار داد. برای مثال، دستورات شرطی if/elif/else را می توان تودرتو کرد:

```
if age < 18:
    if gender == 'M':
        print('son')
    else:
        print('daughter')
elif age >= 18 and age < 65:
    if gender == 'M':
        print('father')
    else:
        print('mother')
else:
    if gender == 'M':
        print('grandfather')
    else:
        print('grandmother')
```

به طور مشابه، یک حلقه *while* را می‌توان در حلقه *while* دیگری قرار داد، همانطور که در اینجا نشان داده شده است:

```
>>> a = ['foo', 'bar']
>>> while len(a):
```

```
...     print(a.pop(0))
...     b = ['baz', 'qux']
...     while len(b):
...         print('>', b.pop(0))
...
foo
> baz
> qux
bar
> baz
> qux
```

توجه: عبارت break یا continue که در حلقه‌های تودرتو مشاهده می‌کنید فقط بر اولین حلقه‌ی قبل از خود اعمال می شود و به حلقه‌های دیگر کاری ندارد.

```
while <expr1>:
    statement
    statement

    while <expr2>:
        statement
        statement
        break  # Applies to while <expr2>: loop

    break  # Applies to while <expr1>: loop
```

علاوه بر این، حلقه‌های *while* را می‌توان در داخل دستورات if/elif/else قرار داد و بالعکس:

```
if <expr>:
    statement
    while <expr>:
        statement
```

```
        statement
else:
    while <expr>:
        statement
        statement
    statement
```

```
if <expr>:
    statement
    while <expr>:
        statement
        statement
else:
    while <expr>:
        statement
        statement
    statement
```

در واقع، تمام ساختارهای کنترل پایتون را می توان تا هر اندازه که نیاز دارید با یکدیگر ترکیب کنید که البته این کار طبیعی است، تصور کنید اگر محدودیت‌های غیرمنتظره‌ای مانند «حلقه $while$ را نمی‌توان در یک عبارت if قرار داد» یا «حلقه‌ها را می‌توان حداکثر در چهار عمق درون یکدیگر قرار داد، چقدر خسته‌کننده خواهد بود». به خاطر سپردن همه آنها برای شما بسیار مشکل خواهد بود.
محدودیت‌های عددی یا منطقی به ظاهر دلخواه، نشانه‌ای از طراحی ضعیف زبان برنامه در نظر گرفته می‌شود. خوشبختانه، پایتون محدودیت‌های خیلی کمی دارد.

۱ـ۱۴ـ۲ حلقه‌ی for

در زبان‌های برنامه‌نویسی علاوه بر حلقه‌ی نامعین $while$ حلقه‌هایی وجود دارند که معین می‌باشند، منظور از معین و غیر معین بودن در حلقه‌ها تعداد دورهایی است که حول عبارت‌ها انجام می‌دهیم، بطور مثال حلقه‌ی $while$ تا زمانی که شرط درست است ادامه خواهد داشت. اما نوع دیگری از حلقه‌ها وجود دارد که مقدار

دورهایی که باید حول عبارت‌ها انجام شود معین می‌باشد. د زبان برنامه نویسی پایتون برای حلقه‌های معین فقط یک نوع داریم که البته شاید در نگاه اول به نظر محدودیت بیاید، اما به شما اطمینان می‌دهیم که این حلقه به گونه‌ای نوشته شده است که شما نیازی به نوع دیگری نخواهید داشت.

حلقه‌ی for پایتون به شکل زیر است:

```
for <var> in <iterable>:
    <statement(s)>
```

- <iterable> مجموعه‌ای از اشیاء است، برای مثال، یک لیست یا یک تاپل را می‌توان استفاده کرد.
- <statement(s)> بدنه‌ی حلقه می‌باشد و در واقع عبارت‌هایی است که می‌خواهیم حول آن‌ها بچرخیم.
- <var> متغیری است که می خواهیم در هر تکرار از حلقه می‌خواهیم آن را با یک عنصر از لیست مقداردهی کنیم.

```
>>> a = ['foo', 'bar', 'baz']
>>> for i in a:
...     print(i)
...
foo
bar
baz
```

در این مثال، <iterable> لیست a است و <var> متغیر i است. هر بار که که حلقه حول لیست a می‌چرخد متغیر i یک مقدار از لیست را می‌گیرد، بنابراین تابع $print()$ مقادیر foo ، bar و baz را به ترتیب نمایش می‌دهد. حلقه‌ی for یک روش پایتونی است تا حول مقادیر یک چندتایی مانند لیست بچرخیم و از مقادیر آن استفاده کنیم.

حلقه‌ی for حول مقادیر تکرارپذیر تکرار می‌شود که این به این معناست که در حلقه‌ی for ما یک شی تکرارپذیر مانند لیست، دیکشنری و تاپل داریم که شامل مجموعه‌ای از عناصر است. حال حلقه‌ی for حول

این مقادیر می‌چرخد و در هر تکرار مقدار یکک عنصر را به متغیری که تعریف کرده‌ایم می‌دهد. حال مثال زیر را که تکرار حول دیکشنری است را ببینید:

```
>>> d = {'foo': 1, 'bar': 2, 'baz': 3}
>>> for k in d:
...     print(k)
...
foo
bar
baz
```

همانطور که در مثال بالا دیده می‌شود، حلقه‌ی for تنها حول کلیدهای دیکشنری چرخید و آن‌ها را فقط توانستیم چاپ کنیم. اما اگر بخواهیم حول مقادیر دیکشنری بچرخیم می‌توا روش دیگری را دنبال کرد. روش اول به صورت زیر است:

```
>>> for k in d:
...     print(d[k])
...
1
2
3
```

در کد بالا مانند قبل حول کلیدهای دیکشنری چرخیدیم و کلید به صورت $d[key]$ در دیکشنری قرار دادیم تا مقدار متناظر آن را بیابیم. اما روش راحت‌تری برای چرخیدن حول مقدارهای یک دیکشنری نیز وجود دارد و آن استفاده از متد $values()$ برای دیکشنری است که در کد زیر آورده شده است.

```
>>> for v in d.values():
...     print(v)
...
1
2
3
```

همچنین می‌توان بطور همزمان حول کلیدها و مقادیر یک دیکشنری چرخید که این کار را می‌توان با کاربرد متد $items()$ انجام داد.

```
>>> d = {'foo': 1, 'bar': 2, 'baz': 3}

>>> d.items()
dict_items([('foo', 1), ('bar', 2), ('baz', 3)])
```

توجه: حلقه‌ی for حول مجموعه‌ای از اشیا می‌چرخد، بنابراین اگر مجموعه‌ای از لیست‌ها مانند $[[1,2,3],[72,77,4]]$ یا مجموعه‌ای از دوتایی‌هامانند $[(”foo”,”bar”,3),(72,77,4)]$ داشته باشیم حلقه‌ی for حول آن‌ها می‌چرخد و محتویات آن را خواهد داد.

```
>>> l = [[1,2,3],[72,77,4]]
>>> for b in l:
...     print(b)
...
[1,2,3]
[72,77,4
```

۱ـ۱۴ـ۳ تابع range()

تابع $range()$ دنباله‌ای از اعداد را برمی‌گرداند که به طور پیش‌فرض از0 شروع می‌شوند و به صورت پیش‌فرض با 1 افزایش می‌یابند، و قبل از یک عدد مشخص متوقف می‌شوند.
سینتکس تابع $range()$ به صورت زیر است:

```
range(start, stop, step)
```

همانطور که مشاهده می‌کنید تابع $range()$ سه آرگومان می‌گیرد که شرح آن‌ها به صورت زیر است:

- start **(اختیاری)**: یک عدد صحیح است که مشخص کننده‌ی نقطه‌ی شروع دنباله است و مقدار پیش‌فرض آن صفر است.

- stop **(ضروری)**: یک عدد صحیح است که مشخص می‌کند دنباله‌ی ما در کدام مقدار باید متوقف شود (خود عدد در دنباله)
- step **(اختیاری)**: یک عدد صحیح که افزایش گام را مشخص می‌کند، پیش فرض 1 است.

می‌توان از تابع $range()$ برای تولید اعداد استفاده کرد که در حلقه‌ی for بکار گرفته می‌شود:

```
x = range(3, 20, 2)
for n in x:
  print(n)
```

که نتیجه‌ی آن به صورت زیر است:

```
3
5
7
9
11
13
15
17
19
```

همانطور که مشاهده می‌کنید، اعدادی که تابع $range()$ داده است از عدد ۳ تا ۱۹ می‌باشند.

حال می‌توان همان مثال قبل را به جای مقداردهی $range()$ در متغیر x بطور مستقیم در for استفاده کرد:

```
for n in range(3, 20, 2):
  print(n)
```

این مثال همان نتیجه‌ی کد قبل را خواهد داد.

۱ـ۱۵ توابع در پایتون

در بسیاری از اوقات هنگام برنامه‌نویسی نیاز داریم قسمت‌هایی از کد را چندین بار استفاده کنیم. برای انجام این کار در اکثر زبان‌های برنامه‌نویسی از تابع استفاده می شود.

یک تابع بلوکی از کد سازمان یافته با قابلیت استفاده مجدد است که برای انجام یک عمل واحد استفاده می‌شود. توابع به کدها استقلال بیشتر می‌دهند و اصطلاحا کد ما مدولارتر می‌کنند. ما پیش از این با توابع از پیش تعریف شده‌ی متعددی در پایتون آشنا شدیم، تابعی مانند $print()$ یکی از آن توابع است. حال در این بخش می‌خواهیم یاد بگیریم چگونه تابع خود را ایجاد نماییم.
ما می‌توانیم توابعی برای انجام کار موردنظرمان تعریف کنیم. برای ایجاد یک تابع قواعد ساده‌ای وجود دارد:

- بلوک‌های تابع با کلمه کلیدی def و سپس نام تابع و پرانتز () شروع می‌شوند.

```
def my_function():
  print("Hello from a function")
```

- هر پارامتر یا آرگومان ورودی که می‌خواهیم تعریف کنیم را می‌بایست در بین این پرانتزها قرار دهیم.
- بلوک کدهایی که در تابع نوشته می‌شوند با : شروع می‌شوند و درون تورفتگی نوشته می‌شوند (مانند if، for و...).
- برای استفاده از تابع می‌بایست آن را فراخوانی کرد که این کار با نوشتن نام آن تابع همراه پرانتز انجام می‌شود.

```
def my_function():
  print("Hello from a function")

my_function()
```

در کد بالا $my\_function$ نام تابع است که می‌توان هر نامی را برای آن انتخاب کرد.

۱ـ۱۵ـ۱ آرگومان‌های تابع

اطلاعات و متغیرها را می توان به عنوان آرگومان به توابع ارسال کرد. می‌توان تابع را قسمتی از یک اداره در نظر گرفت که وظیفه‌ی معینی دارد، حال وقتی به آن اداره می‌رویم برای کار مورد نظرمان باید به بخش یا باجه‌ی مورد نظر برویم که در اینجا به معنی فراخوانی تابع است. حال ممکن است آن باجه برای ما کاری

انجام بدهد که نیاز به اطلاعاتی ندارد، اما گاهی اوقات نیاز است اطلاعاتی به باجه بدهیم تا کار مورد نظرمان انجام شود که این همان آرگومان‌های یک تابع است.

در تابع‌ها آرگومان‌ها بعد از نام تابع در داخل پرانتز مشخص می شوند. می توان هر تعداد آرگومان اضافه کرد، فقط آنها را باید با کاما جدا کرد. مثال زیر تابعی با یک آرگومان ($name$) دارد. هنگامی که تابع فراخوانی می‌شود، یک متغیر نام را به همراه آن می‌فرستیم که در داخل تابع برای چاپ نام و خوشامدگویی استفاده می‌شود:

```
def my_function(name):
  print(name + ", Welcome!")

my_function("Mohsen")
my_function("Zahra")
```

که خروجی آن به صورت زیر خواهد بود:

```
Mohsen, Welcome!
Zahra, Welcome!
```

همانطور که می‌بینید می‌توان از این تابع برای نام‌های دیگر نیز استفاده کرد، بنابراین ما یک اسکریپتی داشتیم که قابلیت استفاده‌ی مجدد داشت و برای اینکه نیاز به تکرار آن نداشته باشیم، آن را به طور جدا به عنوان یک تابع نوشتیم تا هر زمان که بخواهیم آن را با متغیر جدیدی فراخوانی کنیم. این کار در آینده برای کدهای پیچیده‌تر بسیار ضروری‌تر خواهد بود.

توجه: واژه‌های پارامتر و آرگومان معمولا با معناهای یکسانی استفاده می‌شود، در اینجا منظور ما اطلاعاتی است که می‌خواهیم به یک تابع بدهیم.

۱ـ۱۵ـ۲ تعداد آرگومان‌ها

به طور پیش فرض، یک تابع باید با تعداد آرگومان‌های صحیح فراخوانی شود. به این معنی که اگر تابع شما انتظار ۲ آرگومان داشته باشد، باید تابع را با ۲ آرگومان فراخوانی کنید، نه بیشتر و نه کمتر.

بطور مثال تابع زیر با دو آرگومان تعریف شده است و با دو آرگومان فرخوانی شد که درست است.

```
def my_function(fname, lname):
  print(fname + " " + lname)

my_function("Emil", "Refsnes")
```

اما در کد زیر تابع را با دو آرگومان تعریف کردیم اما هنگام فراخوانی آن فقط یک متغیر استفاده کردیم که خطا می‌دهد.

```
def my_function(fname, lname):
  print(fname + " " + lname)

my_function("Emil")
```

که خطای آن $TypeError$ است و خروجی آن به صورت زیر خواهد بود:

```
Traceback (most recent call last):
  File "./prog.py", line 4, in <module>
TypeError: my_function() missing 1 required positional argument: 'lname'
```

۱ـ۱۵ـ۳ برگرداندن مقادیر با return

چنانچه مثال اداره را در نظر بگیریم، در بسیاری اوقات نیاز داریم که پس از دادن اطلاعات به باجه‌ی مورد نظر نتیجه‌ی کار را نیز بگیریم، در تابع نیز این مفهوم وجود دارد که با کلیدواژه‌ی $return$ انجام می‌شود. برای برگرداندن مقدار مورد نظرمان می‌توان از $return$ به صورت زیر استفاده کد:

```
def my_function(x):
  return 5 * x

print(my_function(3))
print(my_function(5))
print(my_function(9))
```

در کد بالا سه دفعه تابع را فراخوانی کرده‌ایم و در هر دفعه مقداری به آن داده‌ایم. تابع ما مانند کارخانه‌ای عمل خواهد کرد که اعدادی می دهیم را در ۵ ضرب کند و نتیجه را برمی‌گرداند، برای همین است که می‌توانیم

نتیجه‌ی آن را می‌توانیم چاپ کنیم.

توجه: تابعی که $return$ ندارد در واقع چیزی برنمی‌گرداند و اگر مانند کد قبل آن را چاپ کنیم $None$ را خواهد داد.

```
def my_function(x):
  y = 5*x

print(my_function(3))
```

در کد بالا با اینکه مانند ظاهرا کد سابق است ولی چون چیزی را برنمی‌گرداند پس وقتی چاپ می‌کنیم $None$ نشان خواهد داد.

۱ـ۱۵ـ۴ عبارت pass

ما نمی‌توانیم یک تابع خالی تعریف کنیم، اما اگر به دلایلی بخواهیم تابعی را بدون محتوا تعریف کنیم با خطا مواجه خواهیم شد. برای اینکه خطا را رفع کنیم می‌توانیم از عبارت $pass$ استفاده کنیم، عبارت $pass$ هیچ کاری انجام نمی‌دهد و فقط برای رفع خطای سینتکس در پایتون نوشته می‌شود.
مثال استفاده از عبارت $pass$ به صورت زیر می‌باشد:

```
def myfunction():
  pass
```

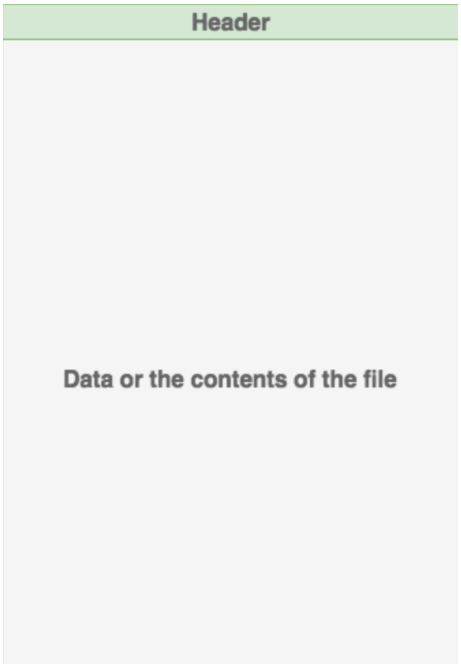
۱ـ۱۶ فایل‌ها در پایتون

یکی از رایج‌ترین کارهایی که می‌توان با پایتون انجام داد خواندن و نوشتن فایل‌ها است. منظور از خواندن و نوشتن فایل می‌تواند نوشتن در یک فایل متنی ساده، خواندن یک گزارش سرور پیچیده، یا حتی تجزیه و تحلیل داده‌های بایتی، همه این شرایط نیاز به خواندن یا نوشتن یک فایل دارند.

۱ـ۱۶ـ۱ فایل چیست؟

پیش از اینکه به نحوه کار با فایل‌ها در پایتون بپردازیم، بهتر است که بدانیم یک فایل دقیقا چیست و چگونه سیستم عامل‌های مدرن برخی از جنبه های آن را مدیریت می کنند.
بطور کلی یک فایل مجموعه‌ای از بایت‌های پیوسته است که برای ذخیره داده‌ها استفاده می‌شود. این داده ها در یک فرمت خاص سازماندهی شده‌اند و می‌توانند هر چیزی به سادگی یک فایل متنی یا به پیچیدگی یک برنامه اجرایی باشند. در نهایت این فایل‌های بایتی برای پردازش آسان‌تر توسط کامپیوتر به ۱ و ۰ ترجمه می‌شوند.
فایل ها در اکثر سیستم های فایل مدرن از سه بخش اصلی تشکیل شده اند:

- اطلاعاتی:Header درباره‌ی محتوای فایل دارد (مانند نام فایل، اندازه و نوع آن).
- محتویات:Data فایل همانطور که توسط سازنده یا ویرایشگر ایجاد شده است.
- کاراکتر:(EOF) خاصی که پایان فایل را نشان می دهد. file of End



اینکه این داده چه چیزی را نشان می‌دهد بستگی به مشخصات فرمت مورد استفاده دارد، که معمولاً با یک پسوند نشان داده می‌شود. به عنوان مثال، فایلی که پسوند آن gif است به احتمال زیاد با مشخصات فرمت

تبادل گرافیکی (برای تصاویر متحرک) مطابقت دارد. صدها، اگر نه هزاران، پسوند فایل وجود دارد.

۱ـ۱۶ـ۲ آدرس فایل

برای دسترسی به یک فایل در سیستم عامل، یک مسیر فایل مورد نیاز است. مسیر فایل رشته‌ای است که مکان یک فایل را نشان می‌دهد. این رشته به سه بخش عمده تقسیم می شود:

- **مسیر پوشه:** محل پوشه فایل در سیستم است که در آن پوشه‌های بعدی با / (یونیکس) یا \ (ویندوز) جدا می‌شوند.
- **نام فایل:** نام واقعی فایل
- **پسوند:** انتهای مسیر فایل از قبل با یک نقطه (.) برای نشان دادن نوع فایل استفاده می شود.

فرض کنید فایلی داریم که ساختار قرارگیری آن به صوت زیر است:

```
/
|
├── path/
|   |
|   ├── to/
|   |   └── cats.gif
|   |
|   └── dog_breeds.txt
|
└── animals.csv
```

فرض کنید می‌خواستید به فایل $cats.gif$ دسترسی داشته باشید و مکان فعلی شما در همان پوشه مسیر قرار دارد. برای دسترسی به فایل باید از پوشه $path$ و سپس پوشه to عبور کنید و در نهایت به فایل $cats.gif$ برسید. مسیر پوشه مسیر/به/ است. نام فایل $cats$ است. پسوند فایل $.gif$ است. بنابراین مسیر کامل $path/to/cats.gif$ است.

اکنون فرض کنید که مکان فعلی یا فهرست کاری فعلی شما (cwd) در پوشه to ساختار پوشه نمونه ما است.

به جای ارجاع به $cats.gif$ با مسیر کامل $path/to/cats.gif$، فایل را می توان به سادگی با نام فایل و پسوند $cats.gif$ ارجاع داد. حال فرض کنید که فایل ما در مسیر فعلی کاری (cwd) قرار دارد. این بار به جای آدرس‌دهی کامل می‌توانیم تنها از نام فایل استفاده کرد.

توجه: منظور از مسیر فعلی فایل آدرسی است که در آن فایل پایتون ما اجرا شده است.

```
/
|
├── path/
|   |
|   ├── to/  ← Your current working directory (cwd) is here
|   |   └── cats.gif  ← Accessing this file
|   |
|   └── dog_breeds.txt
|
└── animals.csv
```

اما چگونه می‌توانیم به $dog\_breeds.txt$ دسترسی پیدا کنیم؟ آیا می‌توان بدون داشتن آدرس کامل فایل به آن دسترسی داشت؟ برای این کار می‌توان از کاراکترهای ویژه‌ی .. استفاده کرد. این کاراکتر برای انتقال یک دایرکتوری به بالا استفاده می‌شود، به این معنا که $../dog\_breeds.txt$ فایل $dog\_breeds.txt$ را از دایرکتوری to می‌خواند. می‌توان از کاراکتر .. به صورت زنجیروار برای دسترسی به فایل‌های متفاوت

```
/
|
├── path/  ← Referencing this parent folder
|   |
|   ├── to/  ← Current working directory (cwd)
|   |   └── cats.gif
|   |
|   └── dog_breeds.txt  ← Accessing this file
|
└── animals.csv
```

استفاده کرد و یا به آدرس‌های بالاتری مانند $../../animals.csv$ دست یافت.

۱ـ۱۷ تمام کننده‌ی خط

یکی از مشکلاتی که اغلب هنگام کار با داده‌های فایل با آن مواجه می‌شویم، نمایش کاراکتر خط جدید یا **انتهای خط** است. کاراکتر پایان خط ریشه در دوران کد مورس دارد، در آن زمان که پیام‌ها به صورت متوالی به صورت کد ارسال می‌شد برای پایان هر خط از یک کاراکتر ویژه‌ای که پایان خط را نشان می‌دهد

استفاده می شود.

بعدها، این مورد توسط سازمان بین المللی استاندارد (ISO) و انجمن استاندارد آمریکا (ASA) برای چاپگرهای تلفنی استفاده شد. استاندارد ASA بیان می کند که انتهای خطوط باید از دنباله‌ی کاراکترهای $CarriageReturn$ (CR یا $\r$) و برای کاراکتر خط بعد (LF یا $\n$) ($CR+LF$ یا $\r\n$) استفاده کنند. با این حال استاندارد ISO برای کاراکترهای CR+LF یا فقط کاراکتر LF مجاز است. بنابراین یک فایل در ویندوز به شکل زیر می‌باشد: همانطور که مشاهده می‌کنید هر خط با $\r\n$ نمایش داده می شود که

```
Pug\r\n
Jack Russell Terrier\r\n
English Springer Spaniel\r\n
German Shepherd\r\n
Staffordshire Bull Terrier\r\n
Cavalier King Charles Spaniel\r\n
Golden Retriever\r\n
West Highland White Terrier\r\n
Boxer\r\n
Border Terrier\r\n
```

خواندن فایل را مقداری با مشکل مواجه خواهد کرد.

۱ـ۱۷ـ۱ بازکردن و بستن فایل در پایتون

وقتی می خواهیم با یک فایل کار کنیم، اولین کاری که باید انجام دهیم این است که آن را باز کنیم. این کار با فراخوانی تابع داخلی $open()$ انجام می‌شود. $open()$ یک آرگومان مورد نیاز دارد که مسیر فایل است. تابع $open()$ یک چیز را بازمی‌گرداند و آن شی فایل است:

```
1 file = open('dog_breeds.txt')
```

کد بالا فایل را می‌خواند و شی فایل را در متغیر $file$ قرار خواهد داد.

پس از باز کردن فایل، نکته بعدی که باید یاد بگیریم نحوه بستن آن است.

هشدار: همیشه باید مطمئن شوید که یک فایل باز شده به درستی بسته شده است.

مهم است که به یاد داشته باشید که این مسئولیت شماست که فایل را ببندید. در بیشتر موارد، پس از

پایان یک برنامه یا اسکریپت، یک فایل در نهایت بسته می‌شود. با این حال، هیچ تضمینی وجود ندارد که دقیقا چه زمانی این اتفاق بیفتد. این می‌تواند منجر به رفتار ناخواسته از جمله **نشت منابع** شود.
هنگامی که در حال دستکاری یک فایل هستید، از دو راه می‌توانید برای اطمینان از بسته شدن صحیح فایل استفاده کنید، حتی زمانی که با خطا مواجه می شوید. اولین راه برای بستن یک فایل استفاده از متد $close()$ است:

```
reader = open('dog_breeds.txt')
    # Further file processing goes here
reader.close()
```

در کد بالا در ابتدا فایل را خواندیم و در نهایت آن را بستیم، در میان خواندن و بستن می‌توانیم کارهایی که می‌خواهیم را بر روی فایل انجام دهیم.
راه دیگری که معمولا توصیه می‌شود آن را انتخاب کنیم استفاده از عبارت $with$ می‌باشد:

```
with open('dog_breeds.txt') as reader:
    # Further file processing goes here
```

دستور with پس از خروج از بلوک with به طور خودکار فایل را می بندد که این کار را حتی در مواردی که خطا رخ می‌دهد انجام خواهد داد و این امنیت آن را بیشتر خواهد کرد. به این منظور به شدت توصیه می‌کنیم تا حد امکان از عبارت with استفاده کنید، زیرا کدی پاک‌تر خواهیم داشت و مدیریت هر گونه خطای غیرمنتظره را آسان‌تر خواهد کرد.
تابع فایل آرگومان دیگری نیز می‌گیرد که مود [۲۷] خواندن فایل است. به طور مثال چنانچه مود خواند فایل را $'r'$ قرار دهیم، تابع $open()$ فایل را خواهد خواند و در صورت عدم وجود آن به ما خطا خواهد داد.

توجه: مودهای خواندن فایل در تابع $open()$ در واقع حالت‌های خواندن فایل را تعیین می‌کنند، بطور مثال اگر فایلی از قبل وجود داشته باشد می‌توانیم برای خواندن آن از مود r استفاده کرد ولی باید حواسمان باشد که در صورت عدم وجود فایل نمی‌توانیم از مود r استفاده کنیم و با خطا مواجه خواهیم شد.

مود خواندن فایل ها را به صورت زیر وارد می‌کنیم:

```
file = open("file_name.txt", "r")
```

[27] Mode

مود	معنی
'r'	مود خواندن فایل، در صورت عدم وجود فایل خطا خواهد داد (حالت پیش‌فرض)
'w'	فایل را با مود نوشتن ایجاد خواهد کرد، در صورت وجود آن ابتدا فایل را خالی خواهد کرد بعد می‌نویسد.
'wb' or 'rb'	فایل در مود باینری را ایجاد می‌کند.

جدول ۱ـ۵: جدول مودهای تابع خواندن فایل

```
2
3 # The work that we want to do on file
4
5 file.close()
```

برای حالت با عبارت *with* به صورت زیر می‌شود:

```
1 with open("file_name.txt", "r") as file:
2   # The work that we want to do on file
```

سایر مودهای حالت‌ها به‌طور کامل به صورت آنلاین مستند شده‌اند، اما رایج‌ترین آنها موارد زیر هستند:

۱ـ۱۷ـ۲ فایل نوع متنی

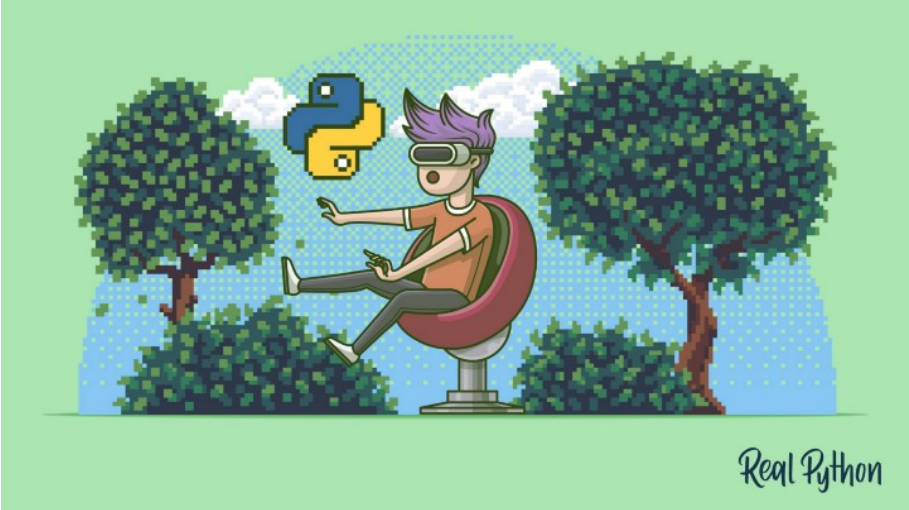
فایل متنی رایج‌ترین فایلی است که با آن مواجه می شویم. در اینجا چند نمونه از نحوه باز کردن این فایل‌ها را آوردیم:

```
1 open('abc.txt')
2 open('abc.txt', 'r')
3 open('abc.txt', 'w')
```

۱ـ۱۸ محیط مجازی پایتون

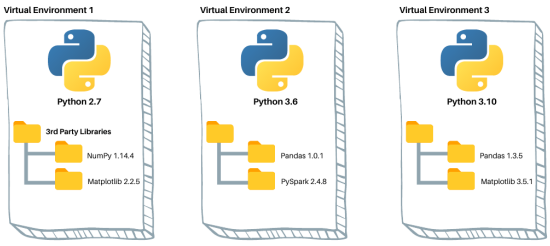
یک محیط مجازی پایتون از دو جزء اصلی تشکیل می‌شود: مفسر پایتون که محیط مجازی روی آن اجرا می‌شود و یک پوشه‌ی حاوی کتابخانه‌های شخص ثالث نصب شده در محیط مجازی. محیط‌های مجازی از سایر محیط‌های مجازی جدا شده‌اند، به این معنی که هرگونه تغییر در وابستگی‌های [۲۸] نصب‌شده در یک

[28] Dependency



محیط مجازی بر وابستگی‌های سایر محیط‌های مجازی یا کتابخانه‌های نصب شده در سراسر سیستم تاثیری نمی‌گذارد. بنابراین، ما می توانیم چندین محیط مجازی با نسخه‌های مختلف پایتون، به علاوه کتابخانه‌های مختلف یا همان کتابخانه‌ها در نسخه‌های مختلف ایجاد نماییم.

توجه: منظور ما از واژه‌ی وابستگی‌ها پکیج‌ها و نرم افزارهایی است که یک کتابخانه یا پکیج برای کار کردن به آن‌ها نیاز دارد.



شکل بالا نشان می‌دهد که وقتی چندین محیط مجازی پایتون ایجاد می‌کنیم، چه چیزی در سیستم خود داریم. همانطور که تصویر بالا نشان می‌دهد، یک محیط مجازی یک پوشه‌ی درختی حاوی یک نسخه خاص از پایتون، کتابخانه های شخص ثالث و اسکریپت‌های دیگر است. بنابراین، هیچ محدودیتی در تعداد محیط های مجازی در یک سیستم وجود ندارد زیرا آنها فقط پوشه‌هایی هستند که برخی فایل‌ها را شامل می شوند.

۱ـ۱۸ـ۱ اهمیت محیط مجازی

اهمیت محیط‌های مجازی پایتون زمانی آشکار می‌شود که ما پروژه‌های مختلف پایتون را روی یک ماشین داشته باشیم که به نسخه‌های مختلف پکیج‌های مشابه بستگی دارد. به عنوان مثال، تصور کنید روی دو پروژه کار می‌کنید که از پکیج $pygame$ استفاده می‌کنند که در هریک از پروژه‌ها از $pygame$ با ورژن متفاوتی استفاده می‌شود. این امر منجر به مشکلات سازگاری می‌شود زیرا پایتون نمی‌تواند به طور همزمان چندین نسخه از یک پکیج در خود داشته باشد. مورد دیگری که اهمیت استفاده از محیط‌های مجازی پایتون را افزایش می‌دهد، زمانی است که روی سرورهایی کار می‌کنید که در آن‌ها اجازه ندارید ورژن پکیجی را تغییر دهید.

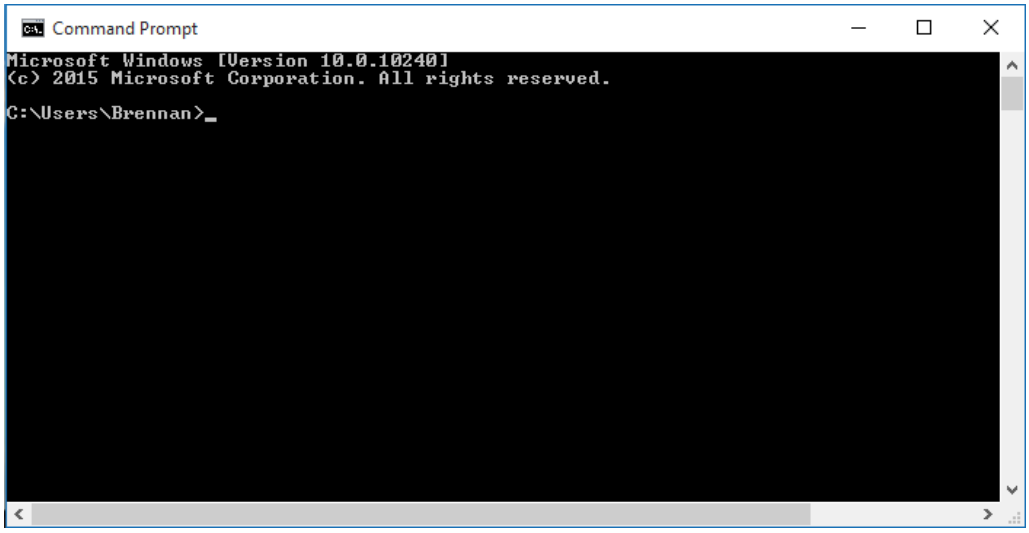
محیط‌های مجازی پایتون محفظه‌های مجزایی را ایجاد می‌کنند تا وابستگی‌های مورد نیاز پروژه‌های مختلف را جدا نگه دارند تا با پروژه‌های دیگر یا پکیج‌های سراسری سیستم تداخل نداشته باشند. اساساً راه اندازی محیط‌های مجازی بهترین راه برای جداسازی پروژه‌های مختلف پایتون است، به خصوص اگر این پروژه‌ها وابستگی‌های متفاوت و متناقضی داشته باشند. برای همین توصیه می‌کنیم که برای هر پروژه‌ی جدیدی که کار می‌کنید، همیشه یک محیط مجازی مجزا اندازی کنید و تمام وابستگی‌های مورد نیاز را در داخل آن نصب کنید و هرگز پکیج‌ها را به صورت سراسری نصب نکنید.

۱ـ۱۸ـ۲ روش ساخت و استفاده از محیط مجازی

تا اینجا یاد گرفتیم که محیط مجازی چیست و چرا به آن نیاز داریم. در این بخش نحوه ایجاد، فعال سازی و (به طور کلی) کار با محیط‌های مجازی را یاد خواهیم گرفت.

برای ایجاد محیط مجازی می‌بایست پوشه‌ای ایجاد کنیم و محیط مجازی را درون آن قرار دهیم. می‌توان پوشه را در هر مکانی از سیستم عامل ایجاد کرد. پس از آن می‌بایست با استفاده از **ترمینال** (در لینوکس) یا خط فرمان (در ویندوز) محیط مجازی را در پوشه‌ی مورد نظرمان ایجاد کنیم.

توجه: برنامه خط فرمان (به انگلیسی Command Prompt) که به نام (cmd) هم شناخته می‌شود رابط کاربری و دستوری متنی سیستم عامل‌ها مانند ویندوز به حساب می‌آید.خط فرمان در مقایسه با رابط کاربری گرافیکی محیطی بی‌روح دارد. رابط کاربری متنی در مقابل رابط کاربری گرافیکی (Graphic User Interface / GUI) قرار می‌گیرد. در واقع در رابط کاربری گرافیکی از اشیای سیستم عامل استفاده می‌کنیم تا کارهای رایانه‌ای‌مان را انجام دهیم و در رابط کاربری متنی از دستورنویسی استفاده می‌کنیم. برای مثال (در ویندوز ما روی My Computer دابل کلیک می‌کنیم و وارد C Drive می‌شویم و فایل File.txt را حذف می‌کنیم، از رابط کاربری گرافیکی سیستم عامل ویندوز برای حذف یک فایل در درایو C استفاده کرده‌ایم اما همین کار را می‌توان با رابط کاربری متنی انجام داد، که باید در Prompt Command دستور $DEL \quad C:\backslash File.txt$ را وارد کنیم. در محیط خط فرمان معمولا صفحه‌های نمایش دارای ۲۵ سطر و ۸۰ ستون هستند و در هر ستون یک کاراکتر تایپ می‌شود.



شکل ۱ـ۲: نمونه‌ای از محیط خط فرمان در ویندوز

برای ایجاد محیط مجازی در پایتون می‌توانیم از دو ابزار موجود استفاده کنیم:

- virtualenv: از نسخه‌های قدیمی پایتون پشتیبانی می‌کند و باید با استفاده از دستور *pip* نصب شود.
- venv: فقط با پایتون 3.3 یا بالاتر استفاده می‌شود و در کتابخانه استاندارد پایتون گنجانده شده است و نیازی به نصب ندارد.

روش اول: virtualenv

اگر سعی کردید virtualenv را اجرا کنید و متوجه شدید که وجود ندارد، می توانید آن را با استفاده از pip نصب کنید:

```
pip install virtualenv
```

که طبق معمول می‌بایست فرمان بالا را در خط فرمان CMD بنویسیم که پس از اجرا با اتصال به مخزن پکیج‌های پایتون برای ما نصب می‌کند. پس از نصب virtualenv احتمالا در آدرسی که پایتون قرار دارد یک فایل $virtualenv.exe$ ایجاد خواهد شد. حال می‌توان با استفاده از خط فرمان به آدرس پوشه‌ای که ایجاد کردیم برویم:

```
cd my_project_path
```

پس از اینکه به آدرس پوشه رفتیم می‌توانیم با اجرای کد زیر

```
virtualenv my_venv
```

در پوشه‌ای که درون آن هستیم محیط مجازی پایتون ایجاد کنیم.

فعال‌سازی محیط مجازی

ایجاد کردن محیط مجازی فقط یک بار انجام می‌شود و برای استفاده از آن محیط می‌بایست آن را فعال کرد. برای استفاده از محیط مجازی و نصب پکیج‌های مورد نظرمان در آن می‌بایست آن را فعال کنیم. برای فعال سازی مانند قبل از خط فرمان استفاده می‌کنیم و به آدرس محیط مجازی می‌رویم. اما این بار از دستور زیر استفاده می‌کنیم:

```
source .\my_venv\bin\activate
```

در واقع کاری که انجام می‌دهیم این است که یک کد به نام $activate$ را از درون پوشه‌ی $script$ واقع در پوشه‌ی محیط مجازی را فراخوانی می‌کنیم.

روش دوم: venv

راه دیگر ایجاد کردن محیط مجازی استفاده از کتابخانه‌ی $venv$ می‌باشد که از پیش در پایتون وجود دارد و نیاز به نصب ندارد. برای این کار می‌بایست در خط فرمان کد زیر را اجرا کنیم:

```
python -m venv my_env
```

در فرمان بالا در واقع از پایتون خواسته‌ایم ماژول $venv$ را فراخوانی کند که این کار را با قرار دادن $-m$ انجام دادیم. بنابراین $-m$ نشان دهنده‌ی فراخوانی یک ماژول است که در اینجا ماژول ما $venv$ بود. حال که $venv$ را فراخواندیم از آن می‌خواهیم که محیط مجازی با نام my_env ایجاد کند. برای فعال‌سازی محیط ایجاد شده می‌توانیم مانند روش قبل از کد زیر استفاده کرد:

```
source .\my_venv\bin\activate
```

در این بخش ما بصورت کلی با مفهوم محیط مجازی در پایتون آشنا شدیم و در بخش‌های آینده از آن بیشتر استفاده خواهیم کرد.

۱ـ۱۹ امتحان و استثنا در پایتون

خطا در پایتون می تواند دو نوع باشد، خطاهای نحوی و استثناها. خطاها مشکلاتی در یک برنامه هستند که به دلیل آن برنامه متوقف می‌شود. از سوی دیگر، استثنائات زمانی مطرح می‌شوند که رویدادهایی درون برنامه رخ می‌دهند که جریان عادی برنامه را تغییر می‌دهند.

۱ـ۱۹ـ۱ خطاهای سینتکسی

خطاهای سینتکسی یا نحوی که به عنوان خطاهای تجزیه نیز شناخته می‌شوند، شاید رایج‌ترین نوع خطاهایی باشند که در حین یادگیری پایتون با آن‌ها مواجه می‌شوید:

```
>>> while True print('Hello world')
  File "<stdin>", line 1
    while True print('Hello world')
                   ^
SyntaxError: invalid syntax
```

تجزیه کننده خط خطا را تکرار می کند و یک **پیکان** کوچک را نشان می‌دهد که به اولین نقطه در خطی که خطا در آن شناسایی شده است اشاره می‌کند. این خطا توسط نشانه‌ی قبل از فلش ایجاد می‌شود. در این مثال، خطا در تابع $print()$ شناسایی می‌شود، زیرا یک دونقطه (':') قبل از آن وجود نداشت. نام

فایل و شماره خط چاپ شده است، بنابراین شما می‌توانید با این اطلاعات جایی که خطا رخ داده است را تشخیص دهید.

۱ـ۱۹ـ۲ استثناها

حتی اگر یک عبارت یا گزاره از نظر نحوی صحیح باشد، ممکن است هنگام تلاش برای اجرای آن خطا رخ دهد. خطاهایی که در حین اجرا شناسایی می شوند **استثنا** نامیده می‌شوند و می‌بایست تعامل با آن‌ها را فرا گرفت. به زودی یاد خواهیم گرفت که چگونه آنها را در برنامه‌های پایتون مدیریت کنیم. با این حال، اکثر استثناها توسط برنامه‌ها مدیریت نمی‌شوند و به پیام‌های خطایی منجر می‌شوند که در اینجا نشان داده شده است:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

خط آخر پیام نشان می‌دهد که چه اتفاقی افتاده است. استثناها انواع مختلفی دارند و برای همین است که نوع استثنا نیز جزو پیام خطا نمایش داده می‌شود. در این مثال استثناهای نوع ZeroDivisionError، NameError و TypeError را می‌بینیم.

برخی از خطاهای استثنای رایج عبارتند از:

- IOError: اگر فایل نتواند باز شود.
- KeyboardInterrupt: وقتی برنامه‌را با فشردن یک کلید غیر ضروری می‌بنیدم.

- ValueError: هنگامی که یک تابع از پیش تعریف شده آرگومان نادرستی بگیرد.
- EOFError: هنگامی که بدون خواندن داده‌ای به خط آخر می‌رسیم.
- ImportError: اگر نتوانیم ماژولی را پیدا کنیم.

۱ـ۱۹ـ۳ Try و Except در پایتون

برای مدیریت اینگونه استثناها از دستور Try و Except در کد پایتون استفاده می‌شود. بلوک try برای بررسی برخی از کدها برای وجود خطا استفاده می‌شود، یعنی کد داخل بلوک try زمانی اجرا می‌شود که هیچ خطایی در برنامه وجود نداشته باشد. در حالی که هر زمان که برنامه در بلوک try با خطا مواجه شود، کد داخل بلوک except اجرا خواهد شد. سینتکس کد try و except صورت زیر است:

```
try:
    # Some Code
except:
    # Executed if error in the
    # try block
```

بلوک try چگونه کار می‌کند؟ ابتدا عبارت try اجرا می شود (یعنی کد بین عبارت try و except). اگر هیچ استثنایی وجود نداشته باشد، فقط عبارت try اجرا می شود، با این تفاوت که بند تمام شده است. اگر هر استثنایی رخ دهد، از عبارت try حذف می شود و عبارت استثنا اجرا می شود. اگر استثنایی رخ دهد، اما عبارت استثنا در کد آن را مدیریت نمی کند، به دستورات try خارجی منتقل می شود. اگر استثنا بدون کنترل باقی بماند، اجرا متوقف می شود. دستور try می تواند بیش از یک عبارت به جز بند داشته باشد

۱ـ۲۰ شی‌گرایی

تا الان در مورد اشیا در پایتون صحبت کردیم، اینکه بعضی توابع شی برمی‌گردانند و یا اینکه انواع داده‌ای که ما داشتیم در پایتون نماینده‌ی شی بودند و دیدیم که عموما متدهایی دارند که می‌توانیم از آن‌ها استفاده

کنیم. اما اینکه این اشیا چه چیزی هستند و یا چگونه ما خودمان در پایتون یک شی با ویژگی‌های متفاوت بسازیم. بحث این بخش درباره اشیا در پایتون و روش سایجاد آن‌ها و اهمیت آن هاست.
از نظر مفهومی، اشیا مانند اجزای یک سیستم هستند. یک برنامه را به عنوان یک خط مونتاژ کارخانه در نظر بگیرید. در هر مرحله از خط مونتاژ، یک جزء سیستم مقداری مواد را پردازش می کند و در نهایت مواد خام را به محصول نهایی تبدیل می‌کند.

۱ـ۲۰ـ۱ شی‌گرایی در پایتون

برنامه نویسی شی‌گرا [۲۹] یک الگوی برنامه نویسی است که ابزاری را برای ساختاربندی برنامه‌ها فراهم می‌کند تا ویژگی‌ها و رفتارها در اشیاء منفرد جمع شوند.
به عنوان مثال، یک شی می تواند فردی را با ویژگی هایی مانند نام، سن و آدرس و رفتارهایی مانند راه رفتن، صحبت کردن، تنفس و دویدن نشان دهد. یا می تواند نشان دهنده یک ایمیل با ویژگی‌هایی مانند لیست گیرندگان، موضوع و بدنه و رفتارهایی مانند افزودن پیوست‌ها و ارسال باشد.
به عبارت دیگر، برنامه‌نویسی شی‌گرا رویکردی برای مدل‌سازی چیزهای واقعی، مانند ماشین‌ها و همچنین روابط بین چیزها، مانند شرکت‌ها و کارمندان، دانش‌آموزان و معلمان و غیره است. شی‌گرایی مسائل دنیای واقعی را به عنوان اشیایی در برنامه‌نویسی مدل می‌کند و با برخورداری از داده‌ها و عملکردهای آن اشیا به آن‌ها روح می‌دهد.

۱ـ۲۰ـ۲ کلاس در پایتون

ساختارهای داده اولیه ـ مانند اعداد، رشته ها و لیست‌ها ـ برای نمایش اطلاعات ساده مانند قیمت یک کتاب، نام یک شعر یا رنگ‌های مورد علاقه شما طراحی شده‌اند. اگر بخواهید چیزی پیچیده‌تر را نشان دهیم چه کاری باید انجام دهیم؟ به عنوان مثال، فرض کنید می‌خواهیم کارکنان یک شرکت را دنبال کنیم. ما باید برخی از اطلاعات اولیه در مورد هر کارمند مانند نام، سن، موقعیت شغلی و سال شروع به کار را ذخیره کنیم.

[29] Object-oriented programming

```
zahra = ["Zahra Rezaei", 34, "Engineer", 2265]
keyvan = ["Keyvan Alavi", 35, "Teacher", 2254]
mohsen = ["Mohsen Jafari", "IT", 2266]
```

یک سری مسائل و مشکلات در این رویکرد وجود دارد.

۱. مدیریت فایل‌ها و داده‌های بزرگتر را دشوار می‌کند به طور مثال اگر به $zahra[0]$ چندین خط دورتر از جایی که لیست $zahra$ اعلام شده است ارجاع دهیم، آیا یادمان می‌ماند که عنصر با اندیس 0 نام کارمند است؟

۲. ممکن است کارمندان دارای تعداد یکسانی از عناصر در لیست نباشند که این خود مشکل ساز است. مثلا در لیست $mohsen$ سن وجود ندارد و اگر $mohsen[1]$ را بخواهیم به جای سن به ما شغل آن را می‌دهد.

۱ـ۲۰ـ۳ کلاس و نمونه

از کلاس‌ها برای ایجاد ساختارهای داده تعریف شده توسط کاربر استفاده می‌شود. کلاس‌ها توابعی به نام متدها را تعریف می‌کنند که رفتارها و اعمالی را که یک شی ایجاد شده از کلاس می تواند با داده‌های خود انجام دهد را مشخص می‌کند. در این بخش یک کلاس Dog ایجاد خواهیم کرد و خواص و ویژگی‌های سگ را در آن وارد خواهیم کرد.

یک کلاس نقشه یا طرحی برای تعریف یک شی می‌باشد که هیچ داده‌ای ندارد. بطور مثال برای کلاس Dog ما مشخص می‌کنیم که نام و سن برای تعریف سگ ضروری است، اما نام یا سن سگ خاصی را تعریف نمی‌کنیم.

همانطور که گفتیم کلاس یک طرح اولیه است اما از جهت دیگر یک نمونه یک شی است که از یک کلاس ساخته شده و حاوی داده‌های واقعی است. یک نمونه از کلاس Dog دیگر یک طرح اولیه نیست. این یک سگ واقعی با نام است، مانند مایلز، که چهار ساله است.

به عبارت دیگر، یک کلاس مانند یک فرم یا پرسشنامه است. یک نمونه مانند فرمی است که با اطلاعات پر شده است. درست مانند بسیاری از افراد که می‌توانند یک فرم را با اطلاعات منحصر به فرد خود پر کنند،

نمونه‌های زیادی نیز می‌توانند از یک کلاس ایجاد شوند.

۱ـ۲۰ـ۴ تعریف کلاس

تمام تعاریف کلاس با کلمه کلیدی کلاس شروع می‌شود و به دنبال آن نام کلاس و دونقطه قرار می‌گیرد. هر کدی که در تورفتگی کلاس تعریف شود، بخشی از بدنه کلاس محسوب می‌شود.

در اینجا نمونه‌ای از کلاس سگ آورده شده است:

```
1 class Dog:
2     pass
```

بدنه کلاس Dog از یک عبارت واحد تشکیل شده است: کلمه کلیدی .pass برای الان این کلمه کلیدی را قرار دادیم تا کد را بدون خطای پایتون اجرا کنیم، در آینده بدنه‌ی کلاس را کامل‌تر خواهیم کرد.

توجه: نام کلاس‌های پایتون به صورت قراردادی با حروف بزرگ نوشته می‌شود. به عنوان مثال، یک کلاس برای یک نژاد سگ خاص مانند جک راسل ترر به صورت JackRussellinTerrieri نوشته می‌شود. اما همانطور که بعدا خواهیم دید برای متدهای کلاس‌ها که مانند تابع هستند مانند قبل قرارداد است که با حروف کوچک بنویسیم.

کلاس Dog در حال حاضر هیجان انگیز نیست، بنابراین بیایید با تعریف برخی از ویژگی‌هایی که همه اشیاء Dog باید داشته باشند، آن را بهتر کنیم. ما می‌توانیم از میان برخی ویژگی‌ها، از جمله نام، سن، رنگ پوشش و نژاد انتخاب چند ویژگی را انتخاب کنیم. برای ساده نگه داشتن موارد، فقط از نام و سن استفاده می‌کنیم.

همه اشیاء Dog باید دارای ویژگی‌های تعریف شده در متدی به نام $init\_\_()$. باشند. هر بار که یک شی Dog جدید ایجاد می شود، $init\_\_()$. وضعیت اولیه شی را با تخصیص مقادیر ویژگی‌های شی تعیین می‌کند. یعنی $init\_\_()$. هر نمونه جدید کلاس را مقداردهی اولیه می‌کند.

ما می توانیم به $init\_\_()$. هر تعداد پارامتر بدهیم، اما اولین پارامتر همیشه متغیری به نام self خواهد بود. هنگامی که یک نمونه کلاس جدید ایجاد می‌شود، نمونه به طور خودکار به پارامتر self در $init\_\_()$. ارسال می‌شود تا بتوان ویژگی‌های جدیدی را روی شی تعریف کرد.

بیایید کلاس Dog را با متد $__init__()$. به روز کنیم که ویژگی‌های $name$. و age. را ایجاد می‌کند:

```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

توجه: به فاصله‌ی متد $__init__()$. دقت کنید، این متد درون تورفتگی کلاس Dog قرار دارد و چهار فاصله دارد و از جهت دیگر بدنه متد $__init__()$. نیز تورفتگی دارد. این تورفتگی‌ها بسیار مهم است و به پایتون می‌گوید که متد $__init__()$. متعلق به کلاس Dog است.

در بدنه‌ی $__init__()$.، دو عبارت وجود دارند که متغیر self در ابتدای آن‌ها وجود دارد:

۱. self.name : یک ویژگی به نام $name$ ایجاد می‌کند و مقدار $name$ را به آن اختصاص می‌دهد

۲. self.age : یک ویژگی به نام age ایجاد می‌کند و مقدار پارامتر age را به آن اختصاص می‌دهد.

ویژگی‌های ایجاد شده در $__init__()$. ویژگی‌های نمونه [۳۰] نامیده می‌شوند. مقدار ویژگی‌های نمونه مخصوص نمونه خاصی از کلاس است (هر سگی یک نام و سنی دارد). اما مقادیر نام و سن به نمونه‌ای که تعریف کردیم دارد، هر سگی نام خود و سن خود را دارد.

از جهت دیگر، ویژگی‌های کلاس، ویژگی‌هایی هستند که برای همه نمونه‌های کلاس، مقدار یکسانی دارند. ما می‌توانید با اختصاص مقداری به نام متغیر خارج از $__init__()$. یک ویژگی کلاس تعریف کنیم.

به عنوان مثال، کلاس Dog زیر دارای یک ویژگی کلاس به نام species با مقدار "$Canis familiaris$" است:

```
class Dog:
    # Class attribute
    species = "Canis familiaris"

    def __init__(self, name, age):
        self.name = name
        self.age = age
```

[30] instance attributes

ویژگی‌های کلاس مستقیماً در زیر خط اول نام کلاس تعریف می‌شوند و چهار فاصله تورفتگی دارند. همیشه باید یک مقدار اولیه به آنها اختصاص داده شود. هنگامی که نمونه‌ای از کلاس ایجاد می‌شود، ویژگی‌های کلاس به طور خودکار ایجاد شده و به با مقادیر اولیه مقداردهی می‌شوند.

توجه: از ویژگی‌های کلاس برای تعریف ویژگی‌هایی استفاده کنید که باید برای هر نمونه کلاس مقدار یکسانی داشته باشند و از ویژگی‌های نمونه برای ویژگی‌هایی که از یک نمونه به نمونه دیگر متفاوت است استفاده کنید.

اکنون که کلاس سگ داریم، بیایید چند سگ ایجاد کنیم!

۱ـ۲۰ـ۵ نمونه‌سازی یک شی در پایتون

برای ایجاد چند نمونه از کلاس Dog مراحل زیر را انجام دهیم تا فرآیند آن را فراگیریم. در ابتدا با نوشتن کد زیر یک کلاس Dof ایجاد کنید:

```
>>> class Dog:
...     pass
```

در کد بالا یک کلاس Dog جدید بدون ویژگی یا متد ایجاد کردیم.

ایجاد یک شی جدید از یک کلاس را نمونه‌سازی یک شی می‌نامند. می‌توان با تایپ کردن نام کلاس و سپس باز کردن و بستن پرانتزها، یک شی جدید Dog را نمونه‌سازی کرد:

```
>>> Dog()
<__main__.Dog object at 0x106702d30>
```

وقتی این کار را انجام می‌دهیم در واقع یک شی Dog جدید در ۰x۱۰۶۷۰۲d۳۰ ایجاد کردیم. این رشته به ظاهر بی‌معنی و شاید خنده‌دار از حروف و اعداد یک آدرس حافظه است که نشان می‌دهد شی Dog کجا در حافظه رایانه شما ذخیره می‌شود. توجه داشته باشید که آدرسی که روی صفحه نمایش خود می‌بینید متفاوت خواهد بود.

حالا یک شی Dog دوم را ایجاد می‌کنیم:

```
>>> Dog()
<__main__.Dog object at 0x0004ccc90>
```

نمونه جدید Dog در آدرس حافظه دیگری قرار دارد. این به این دلیل است که این یک نمونه کاملاً جدید ایجاد کرده‌ایم و کاملاً مستقل از اولین شی ایDog است که پیش از این درست کردیم. برای اینکه از متفاوت بودن دو شی مطمئن شویم می‌توانیم مقایسه‌ی زیر را انجام دهیم:

```
>>> a = Dog()
>>> b = Dog()
>>> a == b
False
```

در این کد دو شی Dog جدید ایجاد کردیم و آنها را به متغیرهای a و b اختصاص دادیم. وقتی a و b را با استفاده از عملگر == مقایسه می‌کنیم، نتیجه False است. حتی اگر a و b هر دو نمونه‌هایی از کلاس Dog باشند دو شی مجزا را در حافظه نشان می‌دهند.

۱ـ۲۰ـ۶ کلاس و ویژگی‌های نمونه

اکنون یک کلاس Dog جدید با یک ویژگی کلاس به نام *species*. و دو ویژگی نمونه به نام *name*. و *age*. ایجاد می‌کنیم.

```
>>> class Dog:
...     species = "Canis familiaris"
...     def __init__(self, name, age):
...         self.name = name
...         self.age = age
```

توجه: حواستان به تفاوت بین ویژگی‌های کلاس و ویژگی‌های نمونه باشد، ویژگی‌های کلاس در زیر تعریف کلاس آورده می‌شوند و ویژگی‌های نمونه در زیر *__init__*() . آورده می‌شوند.

همانطور که در کد بالا مشاهده می‌کنید متد *__init__*() . پس از عبارت *self* دو متغیر دیگر را به عنوان آرگومان دریافت می‌کند، بنابراین برای ایجاد نمونه می بایست این دو آرگومان را بدهیم. اگر هنگام نمونه‌سازی این کار را انجام ندهیم، پایتون یک TypeError را ایجاد به ما خواهد داد:

```
>>> Dog()
Traceback (most recent call last):
```

```
  File "<pyshell#6>", line 1, in <module>
    Dog()
TypeError: __init__() missing 2 required positional arguments: 'name' and 'age'
```

برای دادن آرگومان‌ها به پارامترهای name و age، مقادیری را در پرانتز بعد از نام کلاس می‌دهیم که این کار بسیار شبیه استفاده از توابع در بخش‌های قبل است:

```
>>> buddy = Dog("Buddy", 9)
>>> miles = Dog("Miles", 4)
```

کد بالا دو نمونه از کلاس Dog را ایجاد خواهد کرد(یکی برای یک سگ ۹ ساله به نام بادی و دیگری برای یک سگ ۴ ساله به نام مایلز).

حال یک سوال مهم پیش می‌آید، متد $__init__()$. کلاس Dog دارای سه پارامتر است، پس چرا در مثال فقط دو آرگومان به آن ارسال می شود؟

هنگامی که یک شی Dog را نمونه‌سازی می‌کنیم، پایتون یک نمونه جدید ایجاد می‌کند و آن را به پارامتر اول $__init__()$. می‌دهد. بنابراین یکی از پارامترهای متد $__init__()$. همواره خود نمونه است، اینگونه است که وقتی متغیری را تعریف می‌کنیم مخصوص خود نمونه خواهد بود.

از آنجاییکه درک این قسمت ممکن است دشوار باشد بیایید بیشتر در مورد آن صحبت کنیم. اتفاقی که هنگام ایجاد نمونه رخ می دهد این است که وقتی نمونه را ایجاد می‌کنیم به متد $__init__()$. خود نمونه را می‌دهیم تا هنگام ایجاد متغییری در آن معین کنیم که آن متغیر در آن نمونه ایجاد شود، بنابراین $self$ همان نمونه‌ی ایجاد شده است و متغیری مانند $self.age$ در واقع به پایتون می‌گوید من می‌خواهم یک متغیر در نمونه‌ی مورد نظرم ایجاد کنم. بعدا خواهید دید که تمام متدهایی که در کلاس ایجاد می‌کنیم پارامتر $self$ را به عنوان پارامتر اول دارند و این یعنی همه‌ی متدها به وجود نمونه‌ی ما آگاه هستند.

حال پس از ایجاد نمونه‌های Dog ، می‌توانیم با استفاده از علامت **نقطه** به ویژگی‌های نمونه آن‌ها دسترسی پیدا کنیم:

```
>>> buddy.name
'Buddy'
>>> buddy.age
9

```

```
>>> miles.name
'Miles'
>>> miles.age
4
```

به همین ترتیب می‌توانیم به ویژگی‌های کلاس دسترسی پیدا کنیم:

```
>>> buddy.species
'Canis familiaris'
```

یکی از بزرگترین مزایای استفاده از کلاس‌ها برای سازماندهی داده‌ها این است که نمونه‌ها دارای ویژگی‌های مورد انتظار ما هستند. همه نمونه‌های Dog دارای ویژگی‌های *species*.، *name*. و *age*. هستند، بنابراین می‌توان با اطمینان خاطر از این ویژگی‌ها استفاده کرد و دانست که آن‌ها همیشه مقداری را برمی‌گردانند.

اگرچه وجود ویژگی‌ها تضمین شده است، اما مقادیر آن‌ها را می توان به صورت پویا تغییر داد:

```
>>> buddy.age = 10
>>> buddy.age
10

>>> miles.species = "Felis silvestris"
>>> miles.species
'Felis silvestris'
```

در کد بالا، ما ویژگی *age*. شی *buddy* را به ۱۰ تغییر دادیم. پس از آن ویژگی *species*. شی *miles* را به "*Felissilvestris*" تغییر دادیم (گونه‌ای گربه). با اینکه این کار مایلز را به یک سگ بسیار عجیب تبدیل می‌کند، اما برای پایتون معتبر است!

نکته‌ی کلیدی در اینجا این است که اشیاء به طور پیش فرض قابل تغییر هستند. یک شی قابل تغییر است اگر بتوان آن را به صورت پویا تغییر داد. به عنوان مثال، لیست‌ها و دیکشنری‌ها قابل تغییر هستند، اما رشته‌ها و تاپل‌ها تغییرناپذیر هستند.

۱ـ۲۰ـ۷ متدهای نمونه

متدهای نمونه، توابعی هستند که در داخل یک کلاس تعریف می‌شوند و فقط می‌توان آنها را از نمونه‌ی کلاس فراخوانی کرد و مثل $init\_\_()\_\_$. پارامتر اول آنها همواره باید self باشد.
حالا بیایید چند متد برای کلاسمان ایجاد کنیم:

فصل ۲

پروژه‌های پایتون

در فصل قبل تلاش کردیم که سینتکس پایه‌ی زبان برنامه‌نویسی پایتون را پوشش دهیم که البته بسیاری از بخش‌هایی که گفته شد عمدا غیرکامل گذاشته شد. ما بر این باور هستیم که دانشجویان می‌بایست آرام آرام در برنامه‌نویسی عمق پیدا کنند و جستجو و یادگیری مداوم را تشویق می‌کنیم. اما هدف از توضیح سریع مطالب پایه‌ای در بخش قبل این بود که سریع‌تر وارد بخش پروژه‌ها شویم که در این فصل به آن خواهیم پرداخت.

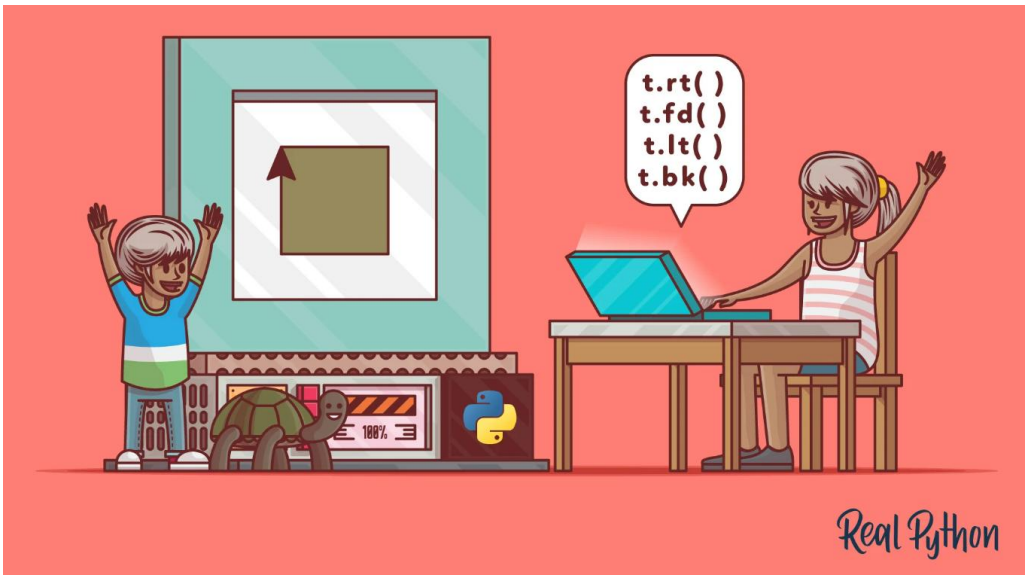
در این فصل که از پروژه‌های گوناگونی تشکیل می‌شود، سعی بر این است که کار کردن با کتابخانه‌های متعددی را فراگیریم و از مفاهیم پایه‌ای که تاکنون گفته شده است به درستی و به طور ابتکاری استفاده کنیم.

بخش اول این فصل به کار کردن با کتابخانه‌ی $Turtle$ اختصاص داده شده است.

۲ـ۱ کتابخانه‌ی Turtle

تا الان با مفاهیم اساسی پایتون آشنا شدیم و وقت آن رسیده است که از این مفاهیم به شکل سرگرم کننده‌ای استفاده کنیم.

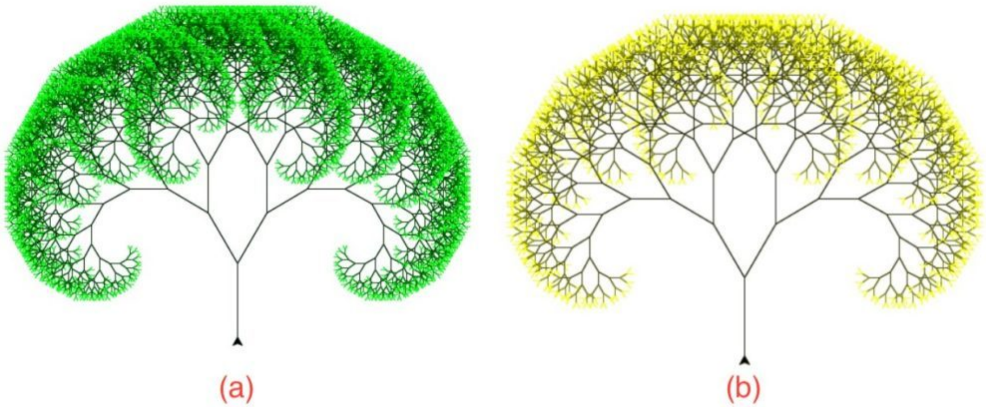
کتابخانه‌ی $turtle$ یک کتابخانه‌ی از پیش نصب‌شده در پایتون می‌باشد که به کاربر امکان رسم تصاویر و اشکال خود در یک بوم مجازی را می‌دهد. قلمی که روی بوم نقاشی برای طراحی از آن استفاده می‌شود لاک‌پشت یا $turtle$نام دارد. بومی که کتابخانه‌ی $turtle$ در اختیارمان می‌گذارد همانند نرم‌افزار $paint$



می‌باشد، اما در اینجا شما امکان خلق شکل‌های پیچیده و زیبایی توسط کدهای پایتون دارید.

کتابخانه‌ی پایتون معمولا انتخاب بسیار خوبی برای افرادی است که قدم‌های اول خود را در شروع برنامه‌نویسی برمی‌دارند، اما این کتابخانه تنها مخصوص افراد مبتدی نیست و می‌توان از آن به عنوان ابزاری آموزشی در تدریس یا خلق برنامه‌های زیبایی استفاده کرد.

پیش از فراگفتن اصول اولیه‌ی کار کردن با کتابخانه‌ی $turtle$ بهتر است مثالی از این کتابخانه را مشاهده کنیم:



۲ـ۱ـ۱ شروع کار با *turtle*

کتابخانه‌ی *turtle* یک کتابخانه‌ی از پیش نصب شده در پایتون است بنابراین نیازی به نصب آن نخواهیم داشت و تنها کاری که باید انجام دهیم وارد کردن آن توسط عبارت *import* است. برای وارد کردن آن می‌بایست در ابتدای اسکریپت پایتون خود به صورت زیر عمل کرد:

```
>>> import turtle
```

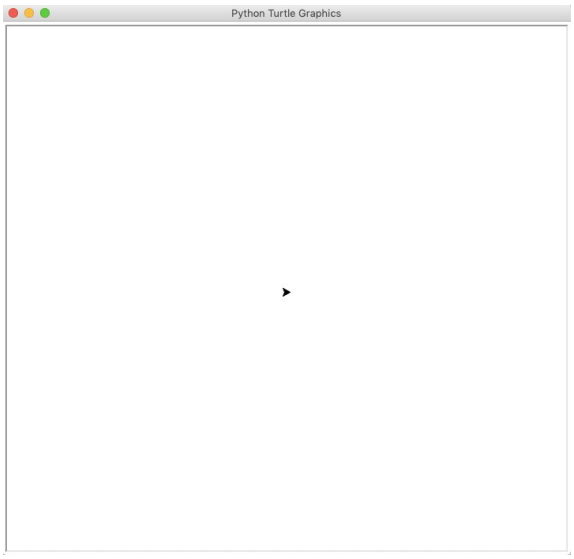
اکنون که *turtle* را در محیط پایتون خود دارید، می‌توانید برنامه‌نویسی را با آن شروع کنید. *turtle* یک کتابخانه‌ی گرافیکی است، به این معنی که برای اجرای هر دستور طراحی باید یک پنجره جداگانه (به نام صفحه نمایش) ایجاد کنیم. می‌توان این صفحه را به یک متغیر نسبت داد تا از آن در طول برنامه استفاده کنیم.

توجه: هنگامی که صفحه‌ی نمایش را به یک متغیر نسبت می‌دهیم می‌توانیم در مراحل بعد در کدمان اشیاء مورد نظرمان را وارد کنیم و اصطلاحا صفحه‌ی نمایش را بروزرسانی کنیم.

برای باز کردن صفحه *turtle*، صفحه‌ی نمایش آن را به یک متغیر نسبت می‌دهیم:

```
>>> import turtle
```

که می‌بایست صفحه‌ای مانند صفحه‌ی نمایش زیر ظاهر شود: این پنجره صفحه نمایش نامیده می‌شود.



جایی است که می‌توانید خروجی کد خود را مشاهده کنید. مثلث کوچک سیاه و سفید در وسط صفحه‌ی

نمایش لاک‌پشت [1] نامیده می‌شود.

توجه: توجه: یادتان باشد که هنگام انتخاب نام برای یک متغیر باید نامی را انتخاب کنید که برای کسی که به برنامه شما نگاه می‌کند به راحتی قابل درک باشد. با این حال، شما همچنین باید نامی را انتخاب کنید که استفاده از آن برای شما راحت باشد، مخصوصاً به این دلیل که اغلب در طول نوشتن آن برنامه چندین بار از آن استفاده خواهید کرد!

سپس برای کنترل لاک‌پشت، متغیر t را مقداردهی اولیه می‌کنیم تا از آن در سرتاسر برنامه برای اشاره به لاک پشت استفاده کنیم:

```
>>> t = turtle.Turtle()
```

درست مانند متغیری که به صفحه نسبت دادیم برای لاک‌پشت نیز می‌توانستیم هر نامی را انتخاب کنیم، اما حالا که نام t را برگزیدیم در ادامه نیز می‌بایست از همین نام استفاده کنیم.
اکنون صفحه نمایش و لاک پشت خود را ایجاد کردیم. صفحه نمایش به عنوان یک بوم عمل می کند، در حالی که لاک‌پشت مانند یک خودکار است. می‌توان لاک‌پشت را به گونه‌ای برنامه‌ریزی کنید که در اطراف صفحه حرکت کند. لاک‌پشت دارای ویژگی‌های قابل تغییری مانند اندازه، رنگ و سرعت است و همواره به یک جهت خاص اشاره می‌کند و در آن جهت حرکت می‌کند مگر اینکه خلاف آن را تعیین کنیم.

۲ـ۱ـ۲ برنامه‌نویسی با $turtle$

اولین چیزی که در مورد برنامه‌نویسی با کتابخانه $turtle$ پایتون یاد خواهیم گرفت این است که چگونه لاک پشت را در جهتی که می‌خواهید حرکت دهید. در مرحله بعد، یاد می‌گیریم که چگونه لاک‌پشت و محیط آن را سفارشی کنیم. در نهایت، چند دستور اضافی را یاد خواهیم گرفت که با آنها می‌توانیم کارهای جالبی را انجام دهیم.

۲ـ۱ـ۲ـ۱ حرکت دادن لاک‌پشت

لاک پشت در چهار جهت می‌تواند حرکت کند:

- رو به جلو: Forward

[1] Turtle

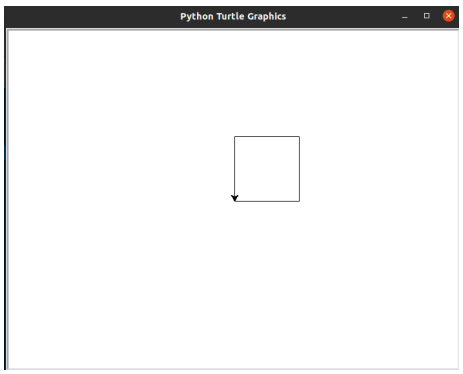
- رو به عقب: Backward
- چپ: Left
- راست: Right

می‌توان لاک‌پشت را با فرمان‌های $forward()$. و $backward()$. به عقب و جلو برد و همچنین با استفاده از فرمان‌های $left()$. و $right()$. می‌توان جهت انتقال لاک‌پشت را تغییر داد.

توجه: تغییر جهت لاک‌پشت هیچ گونه حرکتی به لاک‌پشت نمی‌دهد و فقط جهت انتقال است. مثلا برای ایجاد یک مربع می‌بایست در ابتدا به اندازه‌ی ضلع مربع به جلو حرکت کزد و پس از آن به اندازه‌ی ۹۰ درجه جهت را تغییر داد و مجدد به اندازه‌ی ضلع مربع حرکت کرد و این کار را تا ایجاد مربع ادامه داد.

```
>>> import turtle
>>> t = turtle.Turtle()
>>> t.forward(100)
>>> t.left(90)
>>> t.forward(100)
>>> t.left(90)
>>> t.forward(100)
>>> t.left(90)
>>> t.forward(100)
```

کد بالا یک مربع را به صورت زیر رسم خواهد کرد:



همچنین برای اختصار می‌توان از عبارت‌های کوتاه‌شده‌ی زیر استفاده کرد

۲ـ۲ کتابخانه‌ی pygame

۲ـ۳ کتابخانه‌ی پایکیوت

وقتی صحبت از برنامه‌نویسی می‌شود اولین چیزی که به ذهن می‌رسد برنامه‌های موبایل و وب هستند که بازار توسعه نرم‌افزار را تسخیر کرده‌اند، با این وجود هنوز تقاضا برای برنامه‌های رابط کاربری گرافیکی سنتی (GUI) دسکتاپ وجود دارد. اگر علاقه‌مند به ساخت این نوع برنامه‌ها در پایتون هستید، کتابخانه‌های مختلفی برای انتخاب پیدا خواهید کرد. از برنامه و کتابخانه‌هایی که در این حوزه وجود دارند می‌توان به Tkinter، wxPython، PyQt PySide اشاره کرد.

۲ـ۳ـ۱ آشنایی با پایکیوت

پایکیوت ($PyQt$) در واقع اتصالی است به کیوت (Qt) که مجموعه‌ای از کتابخانه‌ها و ابزار توسعه رابطه‌های گرافیکی نوشته شده توسط زبان برنامه نویسی $C++$ می‌باشند.
آخرین نسخه های PyQt که توسط RiverBank Ltd Computing توسعه یافته است عبارتند از:

- PyQt۵: نسخه ای که فقط با $Qt5.x$ ساخته شده است
- PyQt۶: نسخه ای که فقط با $Qt6.x$ ساخته شده است

در این بخش از PyQt۶ استفاده خواهیم کرد زیرا این نسخه آینده‌ی کتابخانه خواهد بود. از این به بعد، هرجا پایکیوت را یاد کردیم منظورمان PyQt۶ می‌باشد.
پایکیوت ۶ مبتنی بر Qt v۶ است. بنابراین، کلاس‌ها و ابزارهایی را برای ایجاد رابط کاربری گرافیکی، مدیریت ،XML ارتباطات شبکه، عبارات منظم، پایگاه‌های داده SQL خواهد داشت که در Qt وجود دارد. PyQt۶ اتصالی برای بسیاری از کلاس‌های Qt را در مجموعه‌ای از ماژول‌های پایتون پیاده‌سازی می‌کند که در یک بسته‌ی سطح بالای پایتون به نام PyQt۶ سازماندهی شده‌اند. برای کار کردن با PyQt۶، به پایتون 3.6.1 یا بالاتر نیاز داریم.

PyQt۶ با ویندوز، یونیکس، لینوکس، macOS، iOS و اندروید سازگار است. اگر به دنبال یک چارچوب رابط کاربری گرافیکی برای توسعه برنامه‌های چند پلتفرمی که در هر پلتفرم ظاهر یکسانی داشته باشند هستیم، این یک ویژگی جذابی است که در پایکیوت ۶ وجود دارد.

۲ـ۳ـ۲ نصب پایکیوت

چندین روش برای نصب PyQt در سیستم وجود دارد. گزینه‌ی پیشنهادی استفاده از $wheel$های باینری است. $wheel$ روش استاندارد برای نصب بسته‌های پایتون از نمایه‌ی بسته‌های پایتون (PyPI) می‌باشند.

توجه: باید در نظر داشته باشید که $wheel$های PyQt۶ تنها برای $Python3.6.1$ به بعد در دسترس هستند. همه این چرخ ها شامل کپی هایی از کتابخانه های Qt مربوطه هستند، بنابراین نیازی به نصب جداگانه آنها نخواهید داشت.

نصب در محیط مجازی

بیشتر اوقات، باید یک محیط مجازی پایتون ایجاد کنیم تا PyQt۶ را به صورت مجزا نصب کنیم. برای ایجاد یک محیط مجازی و نصب PyQt۶ در آن همانطور که در بخش محیط مجازی دیدیم، می‌بایست فرمان‌های زیر را در خط فرمان خود اجرا کنیم:

```
PS> python -m venv venv
PS> venv\Scripts\activate
(venv) PS> python -m pip install pyqt6
```

در اینجا ابتدا یک محیط مجازی با استفاده از ماژول venv از کتابخانه استاندارد ایجاد کردیم. سپس آن را فعال نمودیم و در نهایت PyQt۶ را با استفاده از pip در آن نصب کردیم. توجه داشته باشید که برای اینکه دستور install به درستی کار کند باید پایتون 3.6.1 یا بالاتر داشته باشید.

نصب به طور سراسری

معمولا توصیه می‌شود که پایکیوت را در محیط مجازی نصب کنیم، اما اگر به هر دلیلی نیاز داشته باشیم آن را به طور سراسری در سیستم نصب کنیم می‌توانیم این کار را به صورت زیر انجام داد:

```
python -m pip install pyqt6
```

۲ـ۳ـ۳ ساخت اولین برنامه در پایکیوت

حال که پایکیوت را نصب کرده‌ایم و نسخه‌ی فعالی از آن در سیستم داریم، می‌توانیم اولین برنامه رابط کاربری گرافیکی خود در پایکیوت را ایجاد کنیم. طبق معمول اول اپلیکیشن $HelloWorld$! خواهد بود. برای ایجاد برنامه مراحلی وجود دارد که باید دنبال کنیم:

۱. QApplication و تمام ویجت های مورد نیاز را از $PyQt6.QtWidgets$ وارد کنید.

۲. یک مورد از QApplication ایجاد کنید.

۳. رابط کاربری گرافیکی برنامه خود را ایجاد کنید.

۴. رابط کاربری گرافیکی برنامه خود را نشان دهید.

۵. حلقه رویداد یا حلقه اصلی برنامه خود را اجرا کنید.

توجه: در علوم کامپیوتر، حلقه رویداد یک ساختار برنامه‌نویسی یا الگوی طراحی است که منتظر رویدادها یا پیام‌های یک برنامه می‌شود. حلقه رویداد با درخواست یک **ارائه‌دهنده رویداد** داخلی یا خارجی (که معمولاً درخواست را تا رسیدن یک رویداد مسدود می‌کند) کار می‌کند، سپس کنترل‌کننده رویداد مربوطه را فراخوانی می‌کند (رویداد را ارسال می‌کند). حلقه رویداد گاهی اوقات به عنوان توزیع کننده پیام، حلقه پیام، پمپ پیام یا حلقه اجرا نیز شناخته می شود. بنابراین بطور کلی حلقه‌ی رویداد الگویی در برنامه‌نویسی که منتظر می‌ماند پیام یا رویدادی از خارج یا داخل برنامه اجرا شود تا بلوکی از برنامه را به اجرا بگذارد. این الگو را پیش از این به طور ضمنی در بازی snake دیدیم که یک حلقه‌ی رویداد داشتیم که منتظر کلیدهای ما بود تا جهت مار را تعیین کند.

برای شروع یک فایل پایتون به نام $hello.py$ به صورت زیر ایجاد نمایید:

```
# hello.py

"""Simple Hello, World example with PyQt6."""
```

```
4
5 import sys
6
7 # 1. Import QApplication and all the required widgets
8 from PyQt6.QtWidgets import QApplication, QLabel, QWidget
```

ابتدا، $system$ را وارد می‌کنیم، که به ما این امکان را می‌دهد تا از طریق تابع $exit()$ وضعیت خاتمه و خروج برنامه را مدیریت کنیم. سپس $QApplication$ ، $QLabel$ و $QWidget$ را از$QtWidgets$ وارد می‌کنیم که بخشی از پکیج $PyQt6$ است. با این کار، مرحله اول را تمام کرده‌ایم.
برای تکمیل مرحله‌ی دوم، فقط باید یک نمونه از QApplication ایجاد کنیم. این کار را همانطور که یک نمونه از هر کلاس [۲] پایتون ایجاد می کنیم انجام می‌دهیم:

```
1 # hello.py
2 # ...
3
4 # 2. Create an instance of QApplication
5 app = QApplication([])
```

در کد بالا، نمونه QApplication را ایجاد نمودیم.

هشدار: پیش از ایجاد هر شی در رابط گرفیکی حتما باید نمونه برنامه‌ی خود را ایجاد کنیم، این کار را در بخش ایجاد بازی با $pygame$ نیز شاهد بودیم که حتما باید یک صفحه‌ی $screen$ می‌کردیم و اشیا را درون آن وارد کنیم.

بطورکلی برنامه‌هایی که ایجاد می‌کنیم می‌توانند با خط فرمان نیز تعامل داشته باشند و آرگومان‌هایی که می‌پذیرند را کلاس QApplication برای ما مدیریت می‌کند. اما برنامه‌ی ساده‌ی ما ارگومان‌هایی نمی‌گیرد و برای همین دلیل است که یک لیست خالی استفاده کردیم.

[۲] ایجاد نمونه از اپلیکیشن به ساختار شی‌گرایی برمی‌گردد که فعلا تا بخشی از ایجاد برنامه با پایکیوت به مفاهیم شی گرایی و کلاس نیاز نخواهیم داشت اما از جایی به بعد برای ساخت برنامه‌ی خوب به آن نیاز خواهیم داشت.

توجه: اغلب متوجه می شوید که توسعه دهندگان sys.argv را به سازنده‌ی QApplication ارسال می‌کنند. این شی حاوی لیستی از آرگومان‌های خط فرمان است که به اسکریپت پایتون ارسال می شود. اگر برنامه شما نیاز به پذیرش آرگومان‌های خط فرمان دارد، باید از sys.argv برای مدیریت آنها استفاده کنید. در غیر این صورت، می توانید مانند مثال بالا از یک لیست خالی استفاده کنید.

مرحله‌ی سوم شامل ایجاد رابط کاربری گرافیکی برنامه است. در این مثال، رابط کاربری گرافیکی شما بر اساس کلاس QWidget است که کلاس پایه تمام اشیای رابط کاربری در پایکیوت است.

```
# hello.py
# ...

# 3. Create your application's GUI
window = QWidget()
window.setWindowTitle("PyQt App")
window.setGeometry(100, 100, 280, 80)
helloMsg = QLabel("<h1>Hello, World!</h1>", parent=window)
helloMsg.move(60, 15)
```

در این کد، پنجره نمونه‌ای از QWidget است که تمام ویژگی‌هایی را که برای ایجاد پنجره یا فرم برنامه به آن نیاز دارید، ارائه می‌کند. همانطور که از نام آن پیداست، $setWindowTitle()$. عنوان پنجره را در برنامه ما تنظیم می‌کند. در این مثال، پنجره برنامه App PyQt را به عنوان عنوان آن نشان می‌دهد. برای تعیین اندازه پنجره و موقعیت صفحه می‌توان از $setGeometry()$. استفاده کرد. دو آرگومان اول مختصات صفحه x و y هستند که در آن پنجره قرار خواهد گرفت. آرگومان های سوم و چهارم عرض و ارتفاع پنجره هستند.

هر برنامه رابط کاربری گرافیکی به ویجت‌ها یا اجزای گرافیکی نیاز دارد که رابط کاربری گرافیکی برنامه را بسازند. در این مثال، ما از یک ویجت $QLabel$ برای نشان دادن پیام $Hello, World!$ در پنجره برنامه استفاده کردیم که نتیجه‌ی آن را در $helloMsg$ قرار دادیم.

اشیا QLabel می توانند متن با فرمت HTML را نمایش دهند، بنابراین می‌توانید از عنصر HTML $<$ " $h1>Hello, World!</h1>$ " برای نمایش متن مورد نظر به عنوان هدر h۱ استفاده کنیم. در

نهایت، از $move()$. برای قرار دادن helloMsg در مختصات (۶۰، ۱۵) در پنجره برنامه استفاده می‌کنیم. حال مرحله‌ی سوم را تمام کردیم، بنابراین می‌توانیم دو مرحله آخر را انجام دهیم و برنامه GUI PyQt خود را برای اجرا آماده‌سازی کنیم:

```
# hello.py
# ...

# 4. Show your application's GUI
window.show()

# 5. Run your application's event loop
sys.exit(app.exec())
```

در این کد متد $show()$. پنجره را فراخوانی کردیم. فراخوانی $show()$ در واقع یک **رویداد رنگ** را زمان‌بندی می‌کند، که درخواستی برای رنگ آمیزی ویجت‌هایی است که یک رابط کاربری گرافیکی را می‌سازند. این رویداد سپس به صف رویداد برنامه اضافه می‌شود. در بخش بعدی درباره حلقه رویداد PyQt بیشتر خواهید آموخت.

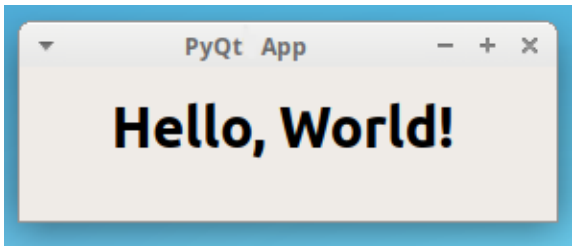
توجه: تصور کنید که یک پنجره‌ی کلی دارید و هر لحظه شی‌ای از آن کم و زیاد و یا در آن جابجا می‌شود، برای اینکه نمایش درستی از برنامه و پنجره ی آن داشته باشیم نیاز داریم که در هر لحظه که تغییری رخ می‌دهد آن‌ها را نمایش دهیم که این همان رویداد رنگ‌آمیزی است که در مورد آن صحبت می‌کنیم.

در نهایت، حلقه رویداد برنامه را با فراخوانی $exec()$. شروع می‌کنیم. فراخوانی $exec()$. در یک فراخوانی $sys.exit()$ پیچیده می‌شود که به ما اجازه می‌دهد هر موقع برنامه پایان یابد به زیبایی از پایتون خارج شویم و منابع حافظه را هنگام پایان برنامه آزاد کنیم.

حال به مرحله‌ی اجرای برنامه رسیدیم، برای این کار می‌توانیم برنامه را از طریق Run کردن (در Python IDLE) اجرا کنیم و یا اینکه در همان جایی که کدمان هست خط فرمان را آدرس‌دهی می‌کنیم و فرمان زیر را اجرا کنیم:

```
python hello.py
```

با اجرای این فرمان برنامه‌ای به شکل زیر مشاهده خواهید کرد: برنامه شما یک پنجره بر اساس QWid-



get نشان می دهد. پنجره سلام جهان را نشان می دهد! برای نشان دادن این پیام از ویجت $QLabel$ استفاده کردیم. و با آن، شما اولین برنامه دسکتاپ رابط کاربری گرافیکی خود را با استفاده از PyQt و Python نوشتید! این باحال نیست؟

استایل برنامه‌نویسی

هنگامی که از یک چارچوب برنامه‌نویسی استفاده می‌کنیم می‌بایست از استایل‌های موجود استفاده کنیم. استایل برنامه‌نویسی قانون نیستند و وقتی از آن‌ها پیروی نکنیم با خطا مواجه نخواهیم شد. استایل‌ها خود را به صورت قرارداد نشان می‌دهند و ما با پیروی از آن‌ها در واقع گویی به سنت‌ها و قواعد اجتماعی یک جامعه (در اینجا برنامه‌نویسان آن زبان) احترام گذاشته‌ایم و البته کد خود را برای دیگران خواناتر کرده‌ایم.

قواعد و قوانین سبک و استایل برنامه‌نویسی پایتون به PEP ۸ معروف هستند و مجموعه‌ای از قواعد هستند که معمولا برنامه‌نویسان باید از آن پیروی کنند تا کدهای منسجم‌تری ارائه دهند. بطور مثال یکی از این قواعد این است که اسم توابع باید با حروف کوچک نوشته شود و بخش‌های متفاوت اسم‌ها را باید با _ جدا کرد. از جهت دیگر در زبان‌های برنامه نویسی دیگر مانند زبان $C++$ برای نام‌گذاری از نوع کوهان شتر استفاده می‌شود که برای جدا کردن بخش‌های متفاوت در یک نام از حرف بزرگ استفاده می‌شود.

اگر کد برنامه رابط کاربری گرافیکی بخش قبل را بررسی کنید، متوجه خواهید شد که PyQt API از سبک کدنویسی و قراردادهای نامگذاری PEP ۸ پیروی نمی‌کند. PyQt بر اساس Qt ساخته شده است که در $C++$ نوشته شده است و از سبک نامگذاری case camel برای توابع، متدها و متغیرها استفاده می‌کند. وقتی شروع به نوشتن یک پروژه PyQt می‌کنید، باید تصمیم بگیرید که از کدام سبک نام‌گذاری استفاده کنید.

در این رابطه $PEP\ \ 8$ بیان می‌کند که:

ماژول‌ها و بسته‌های جدید (از جمله چارچوب‌های شخص ثالث) باید با این استانداردها نوشته شوند، اما

در جایی که کتابخانه‌ی موجود سبک متفاوتی دارد، سازگاری داخلی ترجیح داده می‌شود.
از جهت دیگر Zen پایتون به ما می‌گوید:
... کاربردی بودن بر خلوص ارجحیت دارد.
بنابراین در این بخش ما از نوع نوشتن کوهان شتر استفاده خواهیم کرد.

۲ـ۳ـ۴ اجزای اصلی پایکیوت

اگر بخواهیم به طور ماهرانه از این کتابخانه برای توسعه برنامه‌های رابط کاربری گرافیکی خود استفاده کنیم، باید بر اجزای اصلی پایکیوت تسلط داشته باشیم. برخی از این اجزا عبارتند از:

- Widgets : ویجت‌ها
- managers Layout : مدیران چیدمان
- Dialogs
- windows Main
- Applications
- loops Event
- slots and Signals

این عناصر بلوک‌های سازنده‌ی هر برنامه رابط کاربری گرافیکی PyQt هستند. اکثر آنها به عنوان کلاس های پایتون نشان داده می‌شوند که در ماژول PyQt۶.QtWidgets وجود دارند. در بخش‌های بعد در مورد آنها بیشتر خواهیم آموخت.

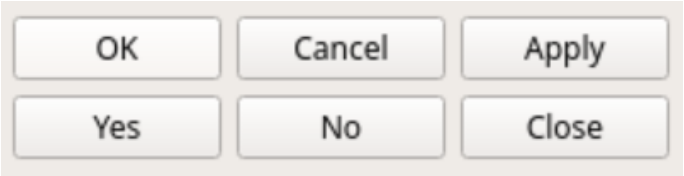
۲ـ۳ـ۵ ویجت‌ها

ویجت‌ها اجزای گرافیکی مستطیلی هستند که می توانیم برای ایجاد رابط کاربری گرافیکی در پنجره‌ی برنامه خود قرار دهیم. ویجت‌ها چندین ویژگی و متد دارند که به ما امکان تغییر ظاهر و رفتار را می‌دهند. همچنین

می‌توان ویجت‌ها را در پنجره اصلی نمایش داد.
ویجت‌ها همچنین کلیک‌های ماوس، فشردن کلیدها و رویدادهای دیگر را از کاربر، سیستم پنجره و سایر منابع شناسایی می‌کنند. هر بار که یک ویجت رویدادی را دریافت می کند، سیگنالی برای اعلام تغییر وضعیت خود منتشر می کند. PyQt مجموعه‌ای غنی و مدرن از ویجت‌ها دارد. هر یک از این ویجت‌ها هدف متفاوتی دارند.
برخی از رایج‌ترین و مفیدترین ویجت‌های PyQt عبارتند از:

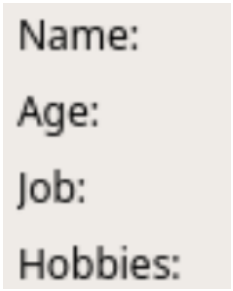
- Buttons: دکمه
- Labels: برچسب
- edits Line : ویرایش خطی
- boxes: Combo جعبه‌ی ترکیبی
- buttons: Radio دکمه رادیویی

اول به دکمه می‌پردازیم. شما می‌توانید یک دکمه با استفاده از QPushButton [۳] ایجاد کنید. معمول‌ترین دکمه‌هایی که احتمالا پیش از این با آن‌ها مواجه شده‌اید دکمه‌های Ok، Cancel، Apply، Yes No و Close هستند که معمولا به شکل زیر هستند: دکمه هایی مانند این شاید رایج ترین ویجت های مورد

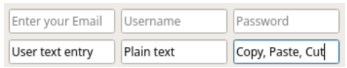


استفاده در هر رابط کاربری گرافیکی باشند. وقتی شخصی روی آنها کلیک می‌کند، برنامه شما به رایانه دستور می‌دهد تا اقداماتی را انجام دهد.
ویجت بعدی برچسب است که می‌توانیم با QLabel ایجاد کنیم. برچسب‌ها به ما اجازه می‌دهند اطلاعات مفیدی را به صورت متن یا تصویر نمایش دهیم: از برچسب‌هایی مانند این برای توضیح نحوه استفاده از رابط کاربری گرافیکی برنامه خود استفاده خواهیم کرد. ما می توانیم ظاهر یک برچسب را به روش‌های

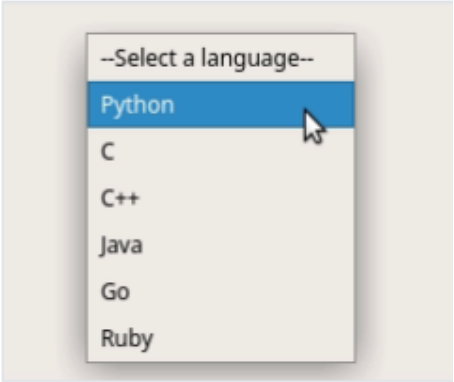
[۳] کلاسی که یک دکمه دستوری کلاسیک را ارائه می دهد



مختلفی تغییر دهیم. همانطور که قبلا دیدیم، یک برچسب حتی می‌تواند متن با فرمت HTML را بپذیرد.
یکی دیگر از ویجت‌های رایج **ویرایش خط** است که به عنوان جعبه‌ی ورودی نیز شناخته می شود. این ویجت به ما امکان می‌دهد یک خط متن را وارد کنید. می توان ویرایش‌های خط را با کلاس QLineEdit ایجاد کرد. ویرایش خط برای زمانهایی مفید است که نیاز داریم ورودی کاربر را به صورت متن ساده دریافت کنیم. ویرایش‌های خطی مانند این به طور خودکار عملیات ویرایش اولیه مانند کپی، چسباندن، لغو، انجام



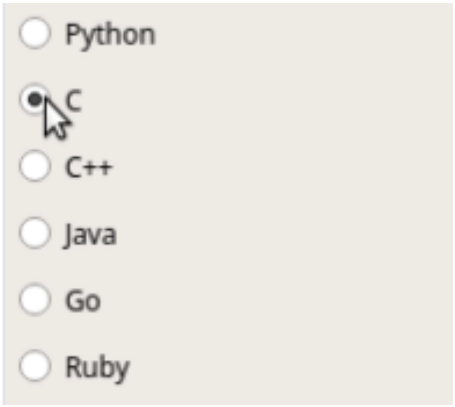
مجدد، کشیدن و رها کردن و غیره را ارائه می‌کنند. همانطور که در شکل بالا می‌توان دید، علاوه بر اشیایی که ویرایش خطی می‌توان دریافت کند می‌توان به کاربر توضیحاتی از شی مورد نظر داد که آن را به شکل خاکستری کم‌رنگ می‌بینیم.
جعبه‌های ترکیبی یکی دیگر از انواع اساسی ویجت در برنامه‌های رابط کاربری گرافیکی هستند. ما می‌توانیم آنها را با استفاده از QComboBox ایجاد کنیم. یک جعبه ترکیبی لیست کشویی از گزینه‌ها را به کاربر ارائه می دهد به گونه‌ای که فضای صفحه را اشغال نکند. این جعبه ترکیبی فقط خواندنی (اصطلاحا read-only) است، به این معنی که کاربر می‌تواند یکی از چندین گزینه را انتخاب کند اما نمی‌تواند گزینه‌های



خود را اضافه کند. جعبه‌های ترکیبی نیز می‌توانند قابل ویرایش باشند و به کاربران امکان می‌دهند گزینه‌های جدیدی را در لحظه اضافه کنند. جعبه های ترکیبی همچنین می توانند حاوی ،pixmaps رشته‌ها یا هر دو باشند.

بدانیم: یک pixmap یک تصویر گرافیکی را به صورت یک آرایه مستطیلی از مقادیر پیکسل رنگی ذخیره و نمایش می دهد. (اصطلاح "pixmap" مخفف "pixel map" است.)
فرمت تصویری بیتمپ (Bitmap) نمایشی از پیکسل‌های با تک رنگ یا تک مقداری در هر پیکسل است.

آخرین ویجتی که در مورد آن خواهیم آموخت دکمه رادیویی است که می توان با استفاده از QRadioButton آن را ایجاد کرد. یک شی QRadioButton یک دکمه گزینه‌ای است که می‌توان برای روشن کردن آن روی آن کلیک کرد. دکمه‌های رادیویی زمانی مفید هستند که نیاز داریم کاربر یکی از گزینه‌های متعدد را انتخاب نماید. همه‌ی گزینه‌های موجود در یک دکمه رادیویی به طور همزمان روی صفحه قابل مشاهده هستند: در گروهی از دکمه‌های رادیویی فقط یک دکمه را می توان در یک زمان مشخص بررسی



کرد. اگر کاربر دکمه رادیویی دیگری را انتخاب نماید، دکمه‌ی انتخاب شده قبلی به طور خودکار خاموش خواهد شد.

پایکیوت مجموعه بزرگی از ویجت‌ها دارد و ما فقط نمونه‌ی کوچکی را دیدیم. با این حال، این برای نشان دادن قدرت و انعطاف پایکیوت کافی است. در بخش بعدی، نحوه چیدمان ویجت‌های مختلف برای ساختن رابط کاربری گرافیکی مدرن و کاملا کاربردی برای برنامه‌های خود را یاد خواهیم گرفت.

۲ـ۳ـ۶ مدیریت چیدمان

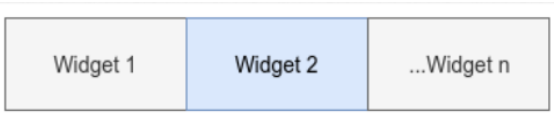
اکنون که در مورد ویجت‌ها و نحوه‌ی استفاده از آنها برای ساخت رابط کاربری اطلاعاتی داریم، باید بدانیم که چگونه مجموعه‌ای از ویجت‌ها را مرتب کنیم تا رابط کاربری گرافیکی ما منسجم و کاربردی شود. در پایکیوت چند تکنیک برای چیدمان ویجت‌ها در یک فرم یا پنجره وجود دارد. به عنوان مثال، می توان از متدهای $resize()$ و $move()$ برای دادن اندازه و موقعیت مطلق به ویجت‌ها استفاده نمود. با این حال، این تکنیک می‌تواند معایبی داشته باشد چون در این صورت مجبور خواهیم بود:

- برای تعیین اندازه و موقعیت صحیح هر ویجت، محاسبات دستی زیادی را انجام دهیم.
- برای رویدادهای تغییر اندازه پنجره، محاسبات اضافی انجام دهیم.
- وقتی طرح پنجره به هر نحوی تغییر کند، می‌بایست اکثر محاسبات خود را از نو انجام دهیم.

روش دیگر شامل استفاده از $resizeEvent()$ برای محاسبه اندازه و موقعیت ویجت به صورت پویا است. در این مورد، مشکلاتی مشابه روش قبلی را خواهیم داشت.
مؤثرترین و توصیه‌شده‌ترین تکنیک استفاده از مدیریت چیدمان پایکیوت است. این تکنیک بهره‌وری را افزایش می دهد، خطر خطاها را کاهش می‌دهند و قابلیت نگهداری کد را بهبود می‌بخشند.
مدیریت چیدمان کلاس‌هایی هستند که به ما اجازه می‌دهند ویجت‌های خود را در پنجره یا فرم‌های متفاوت قرار دهیم. آنها به طور خودکار با تغییر اندازه رویدادها و تغییرات رابط کاربری گرافیکی سازگار می‌شوند و اندازه و موقعیت همه ویجت‌های فرزند خود را کنترل می‌کنند. بطورکلی می‌توان گفت که مدیریت چیدمان‌ها فرم‌هایی هستند که با اندازه و شکل پنجره تغییر می‌کنند و اندازه و شکل هر آنچه درون آن‌ها تعریف می‌شود را متناسب با آن تغییرات تغییر می‌دهند.
کلاس‌های مدیریت چیدمان موجود در پایکیوت به صورت زیر هستند:

۱. QHBoxLayout
۲. QVBoxLayout
۳. QGridLayout
۴. QFormLayout

اولین کلاس مدیریت چیدمان $QHBoxLayout$ است. این کلاس ویجت‌ها را به صورت افقی از چپ به راست مرتب می‌کند. بطور مثال ویجت‌های فرضی زیر را در نظر بگیرید:



در طرح افقی، ویجت‌ها یکی در کنار دیگری ظاهر می‌شوند و از سمت چپ شروع می‌شوند. کد زیر مثالی است از نحوه‌ی استفاده از QHBoxLayout برای مرتب کردن نمایش سه دکمه به صورت افقی را نشان می‌دهد:

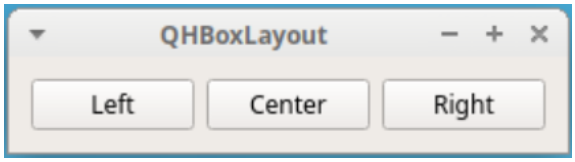
```
# h_layout.py

"""Horizontal layout example."""

import sys

from PyQt6.QtWidgets import (
    QApplication,
    QHBoxLayout,
    QPushButton,
    QWidget,
)

app = QApplication([])
window = QWidget()
window.setWindowTitle("QHBoxLayout")

layout = QHBoxLayout()
layout.addWidget(QPushButton("Left"))
layout.addWidget(QPushButton("Center"))
layout.addWidget(QPushButton("Right"))
window.setLayout(layout)
```

```
23
24 window.show()
25 sys.exit(app.exec())
```

شرح مراحل ایجاد دکمه در کد بالا بصورت زیر است:

- خط ۱۸ یک شی QHBoxLayout به نام layout ایجاد می کند.
- خطوط ۱۹ تا ۲۱ با فراخوانی متد $addWidget()$. سه دکمه به چیدمان اضافه می‌کنند.
- خط ۲۲ طرح‌بندی‌ای که ایجاد کردیم و به آن دکمه‌ها را اضافه کردیم را به‌عنوان طرح‌بندی پنجره ما با $setLayout()$. تنظیم می‌کند.

هنگامی که کد بالا را در خط فرمان اجرا می‌کنیم، خروجی زیر را دریافت خواهیم کرد: شکل بالا سه دکمه



را به صورت افقی نشان می‌دهد. دکمه ها از چپ به راست به همان ترتیبی که در کد خود اضافه کرده‌ایم نشان داده می‌شوند.

کلاس بعدی مدیریت چیدمان QVBoxLayout است که ویجت‌ها را به صورت عمودی از بالا به پایین مرتب می‌کند، مانند شکل زیر:

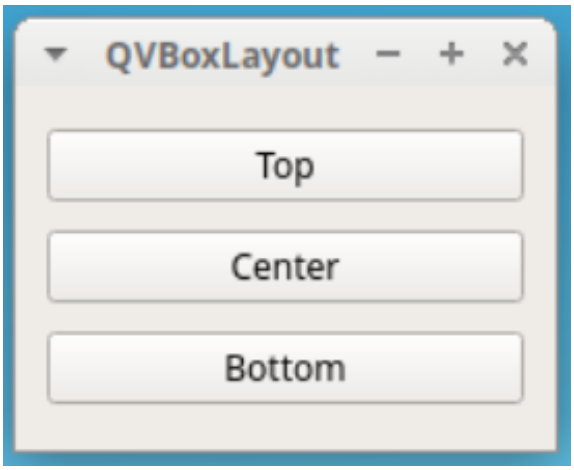
هر ویجت جدید در زیر ویجت قبلی ظاهر می‌شود. این چیدمان به ما امکان می‌دهد طرح‌بندی‌های عمودی بسازیم و ویجت‌های خود را از بالا به پایین در رابط کاربری گرافیکی سازماندهی کنیم.

در کد زیر نحوه ایجاد یک شی QVBoxLayout حاوی سه دکمه آمده است:

```
1  # v_layout.py
2
3  """Vertical layout example."""
4
5  import sys
6
7  from PyQt6.QtWidgets import (
8      QApplication,
9      QPushButton,
10     QVBoxLayout,
11     QWidget,
12 )
13
14 app = QApplication([])
15 window = QWidget()
16 window.setWindowTitle("QVBoxLayout")
17
18 layout = QVBoxLayout()
19 layout.addWidget(QPushButton("Top"))
20 layout.addWidget(QPushButton("Center"))
21 layout.addWidget(QPushButton("Bottom"))
22 window.setLayout(layout)
23
24 window.show()
25 sys.exit(app.exec())
```

در خط ۱۸، ما یک نمونه از QVBoxLayout به نام layout ایجاد نمودیم. در سه خط بعدی، سه دکمه به طرح اضافه می‌کنیم. در نهایت، از شی layout برای مرتب کردن ویجت در یک طرح عمودی از طریق متد $setLayout()$ در خط ۲۲ استفاده می‌کنیم.

هنگامی که این کد را اجرا می کنیم، پنجره‌ی زیر را نمایش خواهد داد: این شکل سه دکمه را به صورت عمودی، یکی زیر دیگری نشان می‌دهد. دکمه‌ها به همان ترتیبی که به کد خود اضافه کرده‌ایم، از بالا به



پایین ظاهر می‌شوند.

سومین کلاس مدیریت چیدمان QGridLayout می‌باشد. این کلاس ویجت‌ها را در شبکه‌ای از ردیف‌ها و ستون‌ها مرتب می‌کند. هر ویجت یک موقعیت نسبی در شبکه خواهد داشت. می‌توانیم موقعیت یک ویجت را با یک جفت مختصات مانند (ردیف، ستون) تعریف کنیم. هر مختصه باید یک عدد صحیح باشد. این جفت مختصات تعیین می‌کنند که یک ویجت معین کدام سلول را در شبکه اشغال کند.

طرح شبکه چیزی شبیه به شکل زیر خواهد بود: در کد زیر نحوه ایجاد آرایشی از چیدمان شبکه‌ای در رابط

Widget (0, 0)	Widget (0, 1)	...Widget (0, n)
Widget (1, 0)	Widget (1, 1)	...Widget (1, n)
...Widget (n, 0)	...Widget (n, 1)	...Widget (n, n)

کاربری گرافیکی آورده شده است:

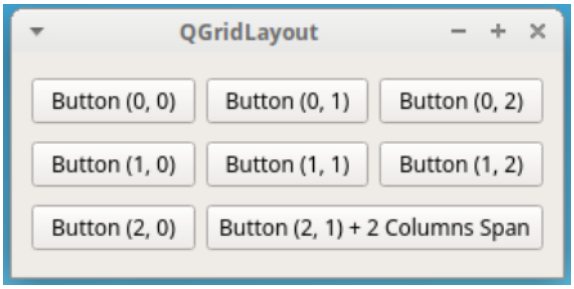
```
# g_layout.py

"""Grid layout example."""

import sys

```

```
from PyQt6.QtWidgets import (
    QApplication,
    QGridLayout,
    QPushButton,
    QWidget,
)

app = QApplication([])
window = QWidget()
window.setWindowTitle("QGridLayout")

layout = QGridLayout()
layout.addWidget(QPushButton("Button (0, 0)"), 0, 0)
layout.addWidget(QPushButton("Button (0, 1)"), 0, 1)
layout.addWidget(QPushButton("Button (0, 2)"), 0, 2)
layout.addWidget(QPushButton("Button (1, 0)"), 1, 0)
layout.addWidget(QPushButton("Button (1, 1)"), 1, 1)
layout.addWidget(QPushButton("Button (1, 2)"), 1, 2)
layout.addWidget(QPushButton("Button (2, 0)"), 2, 0)
layout.addWidget(
    QPushButton("Button (2, 1) + 2 Columns Span"), 2, 1, 1, 2
)

window.setLayout(layout)

window.show()
sys.exit(app.exec())
```

در این مثال، ما برنامه‌ای ایجاد کردیم که از یک شی QGridLayout برای سازماندهی ویجت‌های خود روی صفحه استفاده می‌کند. توجه داشته باشید که در این مورد، آرگومان‌های دوم و سومی که به $addWidget()$ ارسال کردیم، اعداد صحیحی هستند که موقعیت هر ویجت را در شبکه مشخص می‌کنند.

در خطوط ۲۶ تا ۲۸، ما دو آرگومان دیگر را به $addWidget()$ دادیم که این آرگومان‌ها $rowSpan$ و $columnSpan$ هستند و چهارمین و پنجمین آرگومان‌هایی هستند که به تابع ارسال می‌شوند. می‌توانیم از آن‌ها برای ایجاد ویجتی که بیش از یک سطر یا ستون را اشغال می‌کند، استفاده کنیم. همانطور که در کد بالا انجام دادیم.

اگر این کد را از خط فرمان اجرا کنیم، پنجره‌ای به شکل زیر خواهیم داشت: در این شکل می توانیم



ویجت‌های خود را که در شبکه‌ای از ردیف‌ها و ستون‌ها مرتب شده‌اند مشاهده نماییم. همانطور که در خطوط ۲۶ تا ۲۸ مشخص کردیم، آخرین ویجت دو ستون را اشغال می‌کند.

آخرین کلاس مدیریت چیدمان که با آن آشنا خواهیم شد QFormLayout است. این کلاس ویجت‌ها را در یک طرح دو ستونی مرتب می‌کند. ستون اول معمولاً پیام‌ها را در برچسب‌ها نمایش می‌دهد و ستون دوم به طور کلی شامل ویجت‌هایی مانند QLineEdit، QComboBox و QSpinBox و غیره است. اینها به کاربر اجازه می‌دهد تا داده‌های مربوط به اطلاعات ستون اول را وارد یا ویرایش کند.

شکل زیر عملکرد چیدمان فرم‌ها را نشان می‌دهد: ستون سمت چپ از برچسب‌ها و ستون سمت راست از

Label 1	Widget 1
Label 2	Widget 2
...Label n	...Widget n

ویجت‌های ورودی تشکیل شده است. اگر در حال توسعه یک برنامه پایگاه داده باشیم، این نوع چیدمان می‌تواند ابزار مفیدی باشد که بهره‌وری ما را هنگام ایجاد فرم‌های ورودی افزایش دهد.

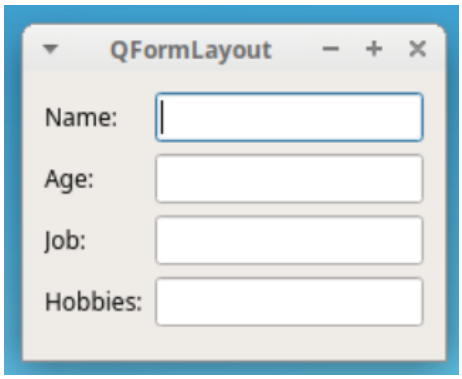
مثال زیر نحوه ایجاد برنامه‌ای را نشان می‌دهد که از یک شی QFormLayout برای مرتب کردن ویجت‌های خود استفاده می‌کند:

```
1  # f_layout.py
2
3  """Form layout example."""
4
5  import sys
6
7  from PyQt6.QtWidgets import (
8      QApplication,
9      QFormLayout,
10     QLineEdit,
11     QWidget,
12 )
13
14 app = QApplication([])
15 window = QWidget()
16 window.setWindowTitle("QFormLayout")
17
18 layout = QFormLayout()
19 layout.addRow("Name:", QLineEdit())
20 layout.addRow("Age:", QLineEdit())
21 layout.addRow("Job:", QLineEdit())
22 layout.addRow("Hobbies:", QLineEdit())
23 window.setLayout(layout)
24
25 window.show()
26
27 sys.exit(app.exec())
```

در کد بالا خطوط ۱۸ تا ۲۳ کار سخت را برای ما انجام می‌دهند.QFormLayout یک متد راحت به

نام $addRow()$. دارد. می توان از این متد برای اضافه کردن یک ردیف دو ویجتی به چیدمان استفاده کرد. اولین آرگومان برای addRow() باید یک برچسب یا یک رشته باشد و آرگومان دوم می‌تواند هر ویجتی باشد که به کاربر اجازه می‌دهد داده‌ها را وارد یا ویرایش کند. در این مثال خاص، ما از ویجت ویرایش خطی استفاده کرده‌ایم. اگر این کد را اجرا کنیم، پنجره‌ای به شکل زیر خواهیم داشت:

شکل بالا پنجره‌ای را نشان می‌دهد که از مدیریت چیدمان فرم استفاده می‌کند. ستون اول حاوی برچسب‌هایی



برای درخواست برخی اطلاعات از کاربر است و ستون دوم ویجت‌هایی را نشان می‌دهد که به کاربر اجازه می‌دهد اطلاعات مورد نیاز را وارد یا ویرایش کند.

Dialogs ۷ـ۳ـ۲

با پایکیوت می‌توان دو نوع برنامه دسکتاپ رابط کاربری گرافیکی ایجاد نمود. بسته به کلاسی که برای ایجاد فرم یا پنجره اصلی استفاده می‌کنیم، یکی از موارد زیر را خواهیم داشت:

۱. یک برنامه به سبک پنجره اصلی: پنجره اصلی برنامه از QMainWindow به ارث می‌رسد.

۲. یک برنامه به سبک گفتگو: پنجره اصلی برنامه از QDialog به ارث می‌رسد.

ابتدا با برنامه‌های با سبک **دیالوگ** شروع خواهیم کرد. در بخش بعد با برنامه‌های کاربردی به سبک **پنجره اصلی** آشنا خواهیم شد.

برای توسعه یک برنامه به سبک دیالوگ، باید یک کلاس رابط کاربری گرافیکی ایجاد نماییم که از QDialog به ارث می‌رسد، این کلاس پایه‌ی همه پنجره‌های گفتگو می‌باشد. پنجره دیالوگ یک پنجره مستقل است که می توان از آن به عنوان پنجره اصلی برنامه خود استفاده کنیم.

در اینجا مثالی از نحوه استفاده از QDialog برای توسعه یک برنامه کاربردی به سبک دیالوگ آورده شده است:

```
# dialog.py

"""Dialog-style application."""

import sys

from PyQt6.QtWidgets import (
    QApplication,
    QDialog,
    QDialogButtonBox,
    QFormLayout,
    QLineEdit,
    QVBoxLayout,
)

class Window(QDialog):
    def __init__(self):
        super().__init__(parent=None)
        self.setWindowTitle("QDialog")
        dialogLayout = QVBoxLayout()
        formLayout = QFormLayout()
        formLayout.addRow("Name:", QLineEdit())
        formLayout.addRow("Age:", QLineEdit())
        formLayout.addRow("Job:", QLineEdit())
        formLayout.addRow("Hobbies:", QLineEdit())
        dialogLayout.addLayout(formLayout)
        buttons = QDialogButtonBox()
        buttons.setStandardButtons(
            QDialogButtonBox.StandardButton.Cancel
```

```
            | QDialogButtonBox.StandardButton.Ok
        )

        dialogLayout.addWidget(buttons)

        self.setLayout(dialogLayout)

if __name__ == "__main__":
    app = QApplication([])
    window = Window()
    window.show()

    sys.exit(app.exec())
```

۲ـ۴ عبارت باقاعده

در بخش‌های قبل با رشته و روش‌های کار با آن‌ها آشنا شدیم، نحوه تعریف و دستکاری رشته‌ها را یاد گرفتیم. روش بررسی برابری دو رشته (==) و متدهای متفاوت آن را دیدیم. اما از آنجاییکه رشته یک داده‌ی بسیار مهم و کاربردی در دنیای واقعی است گاهی اوقات نیاز داریم از شیوه‌های قدرتمند دیگری برای ویرایش و جستجوی درون آن‌ها استفاده کنیم. برای اهمیت بسیار زیاد رشته‌ها در این بخش به عبارت باقاعده یا Expression Regular خواهیم پرداخت.

۲ـ۴ـ۱ استفاده از Regex در پایتون

تصور کنید یک رشته s دارید. حالا فرض کنید باید کد پایتونی بنویسید تا بفهمد آیا s حاوی زیررشته '۱۲۳' است یا خیر. حداقل دو راه برای انجام این کار وجود دارد. می توانید از عملگر in استفاده کنید:

```
>>> s = 'foo123bar'
>>> '123' in s
```

```
True
```

اگر علاوه بر دانستن اینکه «۱۲۳» در s وجود دارد، بخواهیم موقعیتش را در رشته بخواهیم، می‌توانیم از $find()$. یا $index()$. استفاده کنیم. هر یک از اینها موقعیت کاراکتر را در s که «۱۲۳» در آن قرار دارد برمی‌گرداند:

```
>>> s = 'foo123bar'
>>> s.find('123')
3
>>> s.index('123')
3
```

در این مثال‌ها، تطبیق با یک مقایسه ساده کاراکتر به کاراکتر انجام می‌شود. این کار در بسیاری از موارد انجام می شود. اما گاهی اوقات، مشکل پیچیده‌تر از این است.

به عنوان مثال، به جای جستجوی یک زیررشته ثابت مانند «۱۲۳»، فرض کنید بخواهیم تعیین کنیم که آیا یک رشته دارای سه کاراکتر عددی متوالی است، مانند رشته‌های «foo۱۲۳bar»، «foo۴۵۶bar»، «۲۳۴baz» و «qux۶۷۸». مقایسه دقیق کاراکترها در اینجا به کمکی نخواهد کرد. اینجاست که رجکس‌ها در پایتون به کمک ما می‌آیند.

۲ـ۴ـ۲ ماژول re

می‌توان رجکس در پایتون را در ماژولی به نام re یافت. ماژول re شامل توابع و روش‌های مفیدی است که به آن‌ها خواهیم پرداخت.

در ابتدا با تابع $search()$ کار خواهیم کرد.

$$re.search(< regex >, < string >)$$

تابع $search()$ یک الگو (regex) و یک رشته (string) را به عنوان آرگومان دریافت می‌کند و اولین الگویی را که پیدا می‌کند به عنوان تطابق برمی‌گرداند. بنابراین تابع $search()$ تطابق‌ها را به صورت شی برمی‌گرداند و اگر مطابقتی نیابد None به ما می‌دهد. البته تابع $search()$ آرگومان اختیاری نیز دریافت

می‌کند که در بخش‌های بعد به آن خواهیم پرداخت.

۲ـ۴ـ۳ روش وارد کرد ماژول re

از آنجا که $search()$ در ماژول re قرار دارد، قبل از استفاده از آن باید آن را وارد کنیم. یک راه برای انجام این کار این است که کل ماژول را وارد کنیم و سپس هنگام فراخوانی تابع از نام ماژول به عنوان پیشوند استفاده کنیم:

```
import re
re.search(...)
```

البته می‌توانستیم از روش زیر نیز تابع $search()$ را وارد کنیم:

```
import re
re.search(...)
```

۲ـ۴ـ۴ اولین مثال تطابق

الان که می‌دانیم چگونه به $re.search()$ دسترسی پیدا کنیم، می توانیم آن را امتحان کنید:

```
>>> s = 'foo123bar'
>>> import re
>>> re.search('123', s)
<_sre.SRE_Match object; span=(3, 6), match='123'>
```

در اینجا، الگوی جستجو ۱۲۳ و رشته‌ی مورد نظر s است. هنگامی که کد را اجرا می‌کنیم تابع $search()$ شی‌ای را برمی‌گرداند که نتیجه‌ی مطابقت را به ما بگوید، این شی اطلاعات مفیدی دارد که به مرور در مورد آن صحبت می‌کنیم. همانطور که دیده می‌شود، شی بازگشتی $None$ نیست و این یعنی یک مطابقت پیدا شده است و الگوی 123 در s وجود داشته است.

شی مطابقت را می‌توان شرط if استفاده کرد چون مانند True و False عمل می‌کند. بطور مثال کد زیر را در نظر بگیرید:

```
>>> if re.search('123', s):
```

```
...         print('Found a match.')
... else:
...         print('No match.')
...
Found a match.
```

مفسر شی مطابقت شده را به صورت $<_s re.SRE_Matchobject.span = (3,6), match =' 123' >$ نشان می‌دهد که حاوی اطلاعات مفیدی است. بطور مثال $span = (3,6)$ به ما می‌گوید که الگوی ما از اندیس ۳ تا اندیس ۶ را پوشانده است که این دقیقا مانند اندیس رشته‌ای است که برای برش‌های رشته استفاده می‌کردیم:

```
>>> s[3:6]
'123'
```

از جهت دیگر $match =' 123'$ نشان می‌دهد که کدام کاراکتر از <string> مطابقت دارد.

تا الان خوب بود اما در این مورد، الگوی رجکس فقط رشته‌ی ساده $'123'$ است. الگوی تطبیق در اینجا هنوز فقط مقایسه کاراکتر به کاراکتر است، تقریباً مشابه مثال‌های عملگر in و $find()$. که قبلا نشان داده شد. درست است که در اینجا شی تطبیق داده شده به ما می‌گوید که تطابقی وجود دارد اما این کار خیلی کار سختی نیست چون ما همان کاراکترها را جستجو کرده بودیم.

این فقط یک دست‌گرمی بود و قدرت واقعی رجکس‌ها را هنوز ندیدیم. قدرت واقعی تطبیق رجکس در پایتون زمانی دیده می‌شود که ما از کاراکترهای خاصی به نام متاکاراکترها استفاده کنیم. این کاراکترها برای رجکس معنا دارند و قابلیت جستجو را بسیار افزایش می‌دهند.

حال مثال قبل را در نظر بگیرید و ببینم چگونه می‌توان مقادیر سه رقمی را درون رشته بیابیم.

در رجکس برای بررسی مجموعه‌ای از کاراکترها می‌توان از [] استفاده کرد. بنابراین به جای تعیین یک کاراکتر می‌توانیم بگوییم که مثال ما شامل یکی از مجموعه کاراکترهایی است که تعیین می‌کنیم. برای مثال:

```
>>> s = 'foo123bar'
>>> re.search('[0-9][0-9][0-9]', s)
<_sre.SRE_Match object; span=(3, 6), match='123'>
```

مجموعه $[0-9]$ با هر کاراکتر عدد تک رقمی مطابقت دارد(هر کاراکتر بین «۰» و «۹»). از جهت

دیگر عبارت $[0-9][0-9][0-9]$ با هر دنباله‌ای از کاراکترهای سه رقمی مطابقت خواهد داشت. در این مثال، s با الگوی ما مطابقت دارد زیرا حاوی سه کاراکتر سه رقمی متوالی '123' است.
رشته‌های زیر نیز با الگوی ما مطابقت دارند:

```
>>> re.search('[0-9][0-9][0-9]', 'foo456bar')
<_sre.SRE_Match object; span=(3, 6), match='456'>

>>> re.search('[0-9][0-9][0-9]', '234baz')
<_sre.SRE_Match object; span=(0, 3), match='234'>

>>> re.search('[0-9][0-9][0-9]', 'qux678')
<_sre.SRE_Match object; span=(3, 6), match='678'>
```

از جهت دیگر، رشته‌ای که شامل سه رقم متوالی نباشد مطابقت نخواهد داشت:

```
>>> print(re.search('[0-9][0-9][0-9]', '12foo34'))
None
```

با رجکس‌ها در پایتون، می‌توان الگوهایی را در یک رشته شناسایی کرد که با عملگر in یا با متدهای رشته‌ای نمی‌توانستیم پیدا کنیم.
حالا بیایید به یک متاکاراکتر دیگر رجکس نگاهی بیندازیم. متاکاراکتر نقطه (.) با هر کاراکتری به جز یک خط جدید ($\backslash n$) مطابقت دارد.

```
>>> s = 'foo123bar'
>>> re.search('1.3', s)
<_sre.SRE_Match object; span=(3, 6), match='123'>

>>> s = 'foo13bar'
>>> print(re.search('1.3', s))
None
```

در مثال اول، رجکس 1.3 را با '123' مطابقت می‌دهد زیرا '1' و '3' وجود دارند و مطابقت دارند و . با '2' مطابقت دارد (چون همانطور که گفتیم . با هر کاراکتری مطابقت دارد). در اینجا، شما اساساً می‌پرسید: "آیا s حاوی "1" است، سپس هر کاراکتری (به جز یک خط جدید)، سپس "3"؟" پاسخ برای

$'foo123bar'$ بله است اما برای $'foo13bar'$ خیر.

مثال قبل تصویری از قدرت و توانایی رجکس را به نمایش می‌گذارد. در بخش بعد با متاکاراکترهای بیشتری آشنا خواهیم شد.

۲ـ۴ـ۵ متاکاراکترهای ماژول re

جدول زیر به طور خلاصه تمام متاکاراکترهایی که توسط ماژول re پشتیبانی می‌شوند را بطور خلاصه آورده است. برخی از کاراکترها هدف‌های متفاوتی دنبال می‌کنند: این ممکن است حجم زیادی از اطلاعات به

.	با هر کاراکتری به جز خط جدید مطابقت دارد
^	ابتدای یک رشته را مطابقت می‌دهد
$	انتهای یک رشته را مطابقت می‌دهد
*	صفر یا چند تکرار را مطابقت می‌دهد
+	یک یا چند تکرار را مطابقت می‌دهد
؟	ـ صفر یا یک تکرار را مطابقت می‌دهد ـ یک گروه با نام ایجاد می‌کند
{}	تعداد تکرار معینی را مطابقت می‌دهد
\	ـ کاراکترهای با معنای خاص را به حالت عادی می‌برد ـ یک کلاس کاراکتر را ایجاد می‌کند
[]	کلاسی از کاراکترها را معین می‌کند
|	تناوب را تعیین می‌کند
()	یک گروه ایجاد می‌کند
: =# !	یک گروه تخصصی ایجاد می‌کند
<>	یک گروه نام‌دار ایجاد می‌کند

نظر برسد، اما نترسید! بخش‌های بعدی به تفصیل به هر یک از این موارد می‌پردازیم.

رجکس هر کاراکتری را که در بالا ذکر نشده است به عنوان یک کاراکتر معمولی در نظر می گیرد که فقط با خودش مطابقت دارد. به عنوان مثال، در اولین مثال گفته شده این را مشاهده کردیم:

```
>>> s = 'foo123bar'
>>> re.search('123', s)
<_sre.SRE_Match object; span=(3, 6), match='123'>
```

در این مورد،123 از نظر فنی یک رجکس است، اما خیلی جالب نیست زیرا حاوی هیچ متاکاراکتری نیست. فقط با رشته "123" مطابقت دارد.

فصل ۳

تمرین‌ها

۳ـ۱ تمرین ورودی و خروجی پایتون

در پایتون، می‌توان از $input()$ برای گرفتن ورودی کاربر و $print()$ برای نمایش خروجی روی کنسول استفاده کرد. همچنین می‌توان از پایتون برای مدیریت فایل‌ها (خواندن، نوشتن، الحاق و حذف فایل‌ها) استفاده کرد.

هدف از این تمرین‌ها فراگیری توابع از پیش تعریف شده‌ی $print()$ و $input()$ پایتون برای گرفتن ورودی و نمایش خروجی است.

۳ـ۱ـ۱ گرفتن عدد از کاربر

برنامه‌ای بنویسید که دو عدد را از کاربر بپذیرد و ضرب آن دو را محاسبه کند و نتیجه را چاپ کند.

راهنمایی:

- برای گرفتن ورودی از تابع از پیش تعریف شده‌ی $input()$ استفاده کنید.
- ورودی کاربر را با تابع $int()$ به عدد صحیح تبدیل کنید.

۳ـ۱ـ۲ نمایش چند رشته با فرمت خاص

از تابع $print()$ برای نمایش کلمات داده شده در فرمت ذکر شده استفاده کنید. جداکننده‌ی ** بین هر رشته را نمایش دهید.

ورودی مورد نظر:

$print('My','Name','Is','Mohammad')$

خروجی مورد نظر:

$My**Name**Is**Mohammad$

راهنمایی: از پارامتر sep تابع $print()$ برای تعریف نماد جداکننده بین هر کلمه استفاده کنید.

۳ـ۱ـ۳ نمایش عدد float با ۲ رقم اعشار با استفاده از ()print

ورودی مورد نظر:

$num = 458.541315$

خروجی مورد نظر:

458.54

راهنمایی: از کد فرمت $\%2f$ در تابع ()print برای فرمت اعداد float به دو رقم اعشار استفاده نمایید.

۳ـ۱ـ۴ هر سه رشته را از یک ورودی بپذیرید

برنامه‌ای بنویسید که با یک فراخوانی تابع $input()$ سه نام را به عنوان ورودی از کاربر بگیرد و آن‌ها را به صورت جداگانه چاپ نماید.

```
Enter three Names: Zahra Ali Radin
Name1: Zahra
Name2: Ali
Name3: Radin
```

راهنمایی:

- از کاربر بخواهید سه نام را که با فاصله از هم جدا شده‌اند وارد کند
- رشته‌ی ورودی را با متد $split()$ جدا نمایید.

۳ـ۱ـ۵ با استفاده از متد format() متغیری را وارد رشته نمایید

برنامه‌ای بنویسید تا از متد $string.format()$ برای قالب‌بندی سه متغیر زیر مطابق با خروجی مورد انتظار استفاده کند.

```
totalMoney = 1000
quantity = 3
price = 450
```

خروجی مورد انتظار:

```
I have 1000 dollars so I can buy 3 football for 450.00 dollars.
```

۳ـ۲ تمرین‌های حلقه‌ی while

۳ـ۲ـ۱ چاپ ده رقم اول از اعداد صحیح با حلقه‌ی while

توسط حلقه‌ی while ده عدد اول از اعداد صحیح را چاپ نمایید.

نتیجه‌ی مورد انتظار

```
1
2
3
4
5
6
7
```

```
8
9
10
```

۳ـ۲ـ۲ جمع اعداد تا ۱۰۰ با حلقه‌ی while

برنامه‌ای بنویسید که با استفاده از حلقه‌ی while اعداذ را با هم جمع کند و این کار را تا عدد ۱۰۰ انجام دهد.

راهنمایی:

- منظور از جمع اعداد $1 + 2 + 3 + ..$ می‌باشد.
- بهتر است متغیری را با نگهداری مجموع اعداد ایجاد کنید.

۳ـ۲ـ۳ حدس عدد کاربر

فرض کنید کاربر عددی را در نظر دارد، برنامه‌ای بنویسید که با استفاده از حلقه‌ی while عدد مورد کاربر را حدس بزند. برنامه‌ی شما می‌بایست عدد را چاپ نماید و هر دفعه که به کاربر ععدی نشان می‌دهد از او بپرسد که عدد مورد نظر آن بزرگتر یا کوچکتر از عدد چاپ شده است یا نه و برا این اساس عدد را پیدا کند.

```
adadi bein 0-100 dar nazar begirid,

adad shoma 20 ast? bozorgtra ya kochektar?
bozorgtar

adad shoma 50 ast? bozorgtra ya kochektar?
kochektar

adad shoma 40 ast? bozorgtra ya kochektar?
bale

```

```
Horra!! adad peyda shod!!
```

۳ـ۲ـ۴ چاپ الگوی اعداد

الگوی اعداد زیر را با استفاده از حلقه‌ی while چاپ نمایید:

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

۳ـ۳ لیست

۳ـ۳ـ۱ جمع اعداد درون یک لیست

برنامه ای بنویسید که اعداد درون یک لیست را جمع کند و چاپ نماید.

۳ـ۳ـ۲ بیشترین مولفه‌ی لیست

برنامه‌ای بنویسید که بزرگترین عدد درون یک لیست را چاپ نماید.

۳ـ۳ـ۳ جایگزینی مولفه‌های لیست

برنامه‌ای بنویسید که جای مولفه‌ی آخر یک لیست را با مولفه‌ی اول تعویض کند.
بطور مثال: لیست $[1, 2, 5]$ به لیست $[5, 2, 1]$ تبدیل شود.

۳ـ۳ـ۴ حذف اعضای تکراری از لیست

برنامه‌ای بنویسید که اعضای تکراری یک لیست را حذف کند.

۳ـ۳ـ۵ یافتن اندیس مولفه‌ای در لیست

برنامه‌ای بنویسید که اندیس یک مولفه از لیست را پیدا کند.

۳ـ۳ـ۶ ترکیب مولفه‌های دو لیست

برنامه‌ای بنویسید که مولفه‌های دو لیست را با هم ترکیب کند.
بطور مثال ترکیب $[2, 34, 77]$ و $[7, 33, 1]$ بصورت $[2, 34, 77, 7, 33, 1]$ شود.

۳ـ۳ـ۷ یافتن تعداد مولفه‌های لیست

برنامه‌ای بنویسید که تعداد مولفه‌های یک لیست را پیدا کند.

۳ـ۳ـ۸ یافتن میانگین یک لیست

برنامه‌ای بنویسید که میانگین یک لیست را پیدا کند.

۳ـ۳ـ۹ مرتب کردن لیست

برنامه‌ای بنویسید که مولفه‌های یک لیست را بر اساس کوچک به بزرگ مرتب کند.