# پایتون برای همه

**نویسنده : دکتر چارلز سورنس**

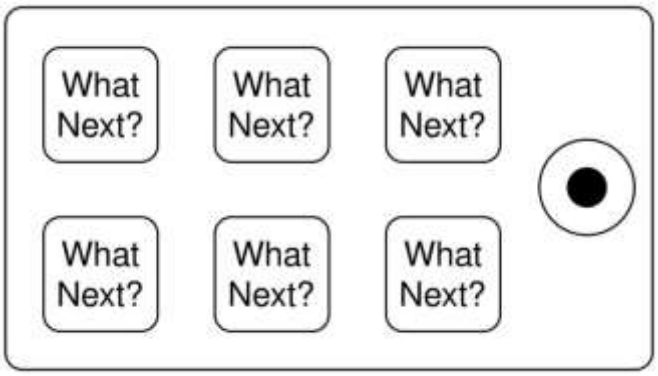
**ترجمه : نوشین ذکاوت**

# فصل اول

## چرا باید برنامه‌نویسی بیاموزیم؟

نوشتن برنامه‌ها (یا برنامه‌نویسی) فعالیتی بسیار خلاقانه و رضایت‌بخش است. دلایل متعددی برای نوشتن برنامه وجود دارد؛ از کسب درآمد گرفته تا حل یک مسئله‌ی دشوار در تحلیل داده، تفریح و سرگرمی، یا حتی کمک به دیگران برای حل مسائلشان. این کتاب بر این فرض استوار است که همه باید برنامه‌نویسی را بیاموزند، و زمانی که شما این مهارت را فرا گرفتید، خودتان تصمیم خواهید گرفت که چگونه از این مهارت استفاده کنید.

ما در زندگی روزمره، با انواع رایانه‌ها از لپ‌تاپ گرفته تا تلفن‌های همراه محاصره شده‌ایم. می‌توان این دستگاه‌ها را به‌عنوان «دستیاران شخصی» در نظر گرفت که می‌توانند بسیاری از کارها را برای ما انجام دهند. سخت‌افزار رایانه‌های امروزی طوری طراحی شده‌اند که مدام از ما بپرسند: «حالا باید چه کاری انجام دهم؟»




شکل شماره ۱: دستیار دیجیتال شخصی

برنامه‌نویسان با افزودن یک سیستم‌عامل و مجموعه‌ای از نرم‌افزارها به سخت‌افزار، دستگاهی را خلق می‌کنند که به‌عنوان «دستیار دیجیتال شخصی» شناخته می‌شود؛ ابزاری که بسیار مفید است و می‌تواند در انجام کارهای گوناگون به ما کمک کند.

رایانه‌های ما بسیار سریع هستند و حافظه‌های گسترده‌ای دارند و اگر ما زبان برقراری ارتباط با آن‌ها را می‌دانستیم، می‌توانستند کمک بزرگی به ما بکنند. اگر این زبان را فرا بگیریم، می‌توانیم از رایانه بخواهیم تا کارهایی را به‌جای ما انجام دهد؛ به‌ویژه کارهای تکراری و زمان‌بر. نکتهی جالب این است که کارهایی که رایانه‌ها در انجام آن‌ها مهارت دارند، معمولاً همان کارهایی هستند که برای ما انسان‌ها خسته‌کننده و طاقت‌فرسا به نظر می‌رسند.

برای مثال، در نسخه انگلیسی کتاب به سه پاراگراف ابتدایی این فصل نگاه کنید و بگویید پرتکرارترین واژه چیست و چند بار به‌کار رفته است. در حالی‌که شما می‌توانید واژه‌ها را در چند ثانیه بخوانید و درک کنید، شمردن تعداد آن‌ها تقریباً کاری آزاردهنده است؛ چرا که ذهن انسان برای حل چنین مسئله‌ای طراحی نشده است. در مقابل، برای رایانه قضیه برعکس است: خواندن و درک متن از روی کاغذ برای رایانه بسیار دشوار است، اما شمردن واژه‌ها و اعلام تعداد دفعات تکرار آن‌ها، کاری بسیار ساده به حساب می‌آید:

```
python words.py
Enter file: words.txt
to 16
```

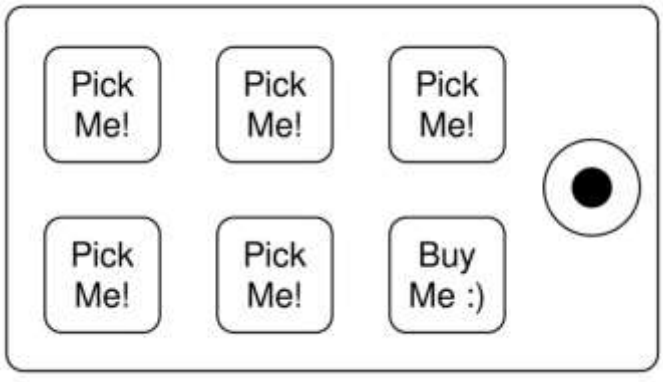
«دستیار شخصی تحلیل اطلاعات» ما به سرعت به ما اطلاع داد که واژه‌ی «to» شانزده بار در سه پاراگراف ابتدایی نسخه انگلیسی این فصل تکرار شده است.

این واقعیت که رایانه‌ها در کارهایی مهارت دارند که برای انسان‌ها دشوار است، دلیلی محکم است که باید توانایی صحبت‌کردن به «زبان رایانه» را در خود پرورش دهیم. هنگامی که این زبان جدید را بیاموزید، می‌توانید کارهای تکراری و زمان‌بر را به شریک خود (رایانه) بسپارید و زمان بیشتری برای پرداختن به اموری که تنها شما در آن‌ها توانمند هستید، اختصاص دهید. در این همکاری، شما خلاقیت، شهود، و نوآوری را وارد میدان می‌کنید.

## ۱,۱ خلاقیت و انگیزه

هرچند این کتاب برای برنامه‌نویسان حرفه‌ای نگاشته نشده است، اما باید بدانید که برنامه‌نویسی حرفه‌ای می‌تواند شغلی بسیار پربازده، هم از نظر مالی و هم از لحاظ شخصی باشد. ساخت برنامه‌هایی مفید، زیبا و هوشمندانه برای استفاده‌ی دیگران، فعالیتی بسیار خلاقانه است. رایانه یا دستیار دیجیتال شخصی (PDA) شما معمولاً شامل برنامه‌های متعددی است که توسط گروه‌های مختلفی از برنامه‌نویسان طراحی شده‌اند، و هر یک از آن‌ها در تلاش‌اند تا توجه و علاقه‌ی شما را جلب کنند. این برنامه‌نویسان نهایت تلاش خود را می‌کنند تا نیازهای شما را برآورده سازند و تجربه‌ی کاربری دل‌پذیری برایتان فراهم آورند. در برخی موارد، انتخاب شما برای استفاده از یک نرم‌افزار خاص، به‌طور مستقیم منجر به منفعت مالی برای برنامه‌نویسان آن می‌شود.

اگر برنامه‌ها را به‌عنوان خروجی‌های خلاقانه‌ی گروهی از برنامه‌نویسان در نظر بگیریم، شاید تصویر زیر توصیف دقیق‌تری از یک PDA(دستیار دیجیتال شخصی) باشد:

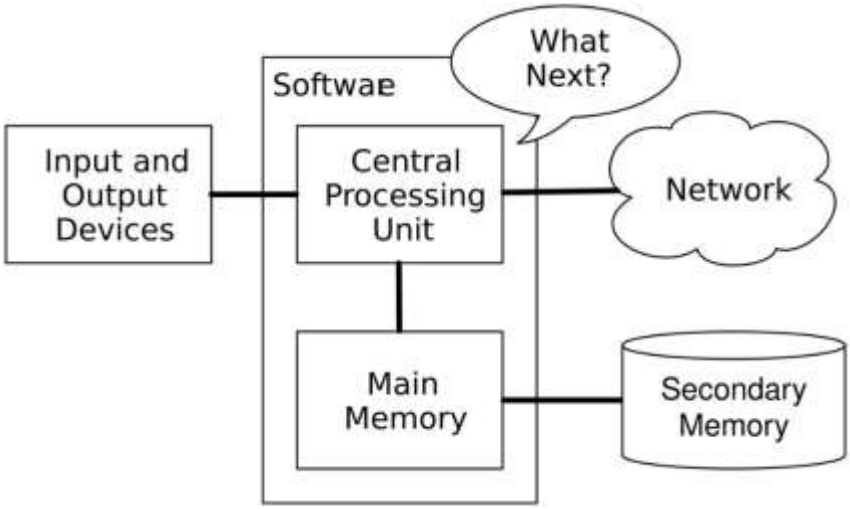



شکل شماره ۲: برنامه نویسان با شما گفتگو می کنند

در حال حاضر، انگیزه‌ی اصلی ما نه کسب درآمد است و نه جلب رضایت کاربران نهایی ؛ بلکه هدف ما این است که در مدیریت داده‌ها و اطلاعاتی که در طول زندگی با آن‌ها روبه‌رو می‌شویم، کارآمدتر عمل کنیم. در ابتدای مسیر، شما هم برنامه‌نویس هستید و هم کاربر نهایی برنامه‌هایی که خودتان طراحی کرده اید. اما با افزایش مهارت‌هایتان در برنامه‌نویسی و تبدیل‌شدن این فرایند به فعالیتی خلاقانه، ممکن است افکارتان به سوی ساخت برنامه‌هایی برای دیگران نیز معطوف شود.

## ۱.۲معماری سخت‌افزار رایانه

پیش از آن‌که زبان گفت‌وگو با رایانه برای توسعه‌ی نرم‌افزار را بیاموزیم، لازم است اندکی درباره‌ی ساختار درونی رایانه‌ها بدانیم. اگر رایانه یا تلفن همراه خود را باز کرده و به درون آن نگاهی بیندازید، با اجزای زیر مواجه خواهید شد:




شکل شماره ۳: معماری سخت افزار

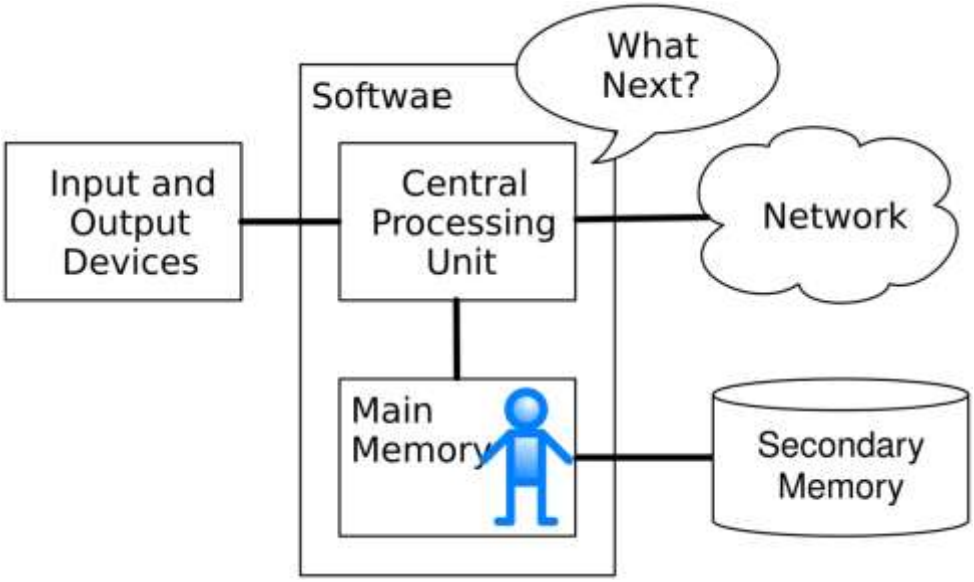
تعریف‌های سطح بالای این اجزا به شرح زیر است:

- **واحد پردازش مرکزی (CPU)** بخشی از رایانه است که به شکل وسواس‌گونه‌ای درگیر پرسیدن «قدم بعدی چیست؟» است. اگر رایانه‌ی شما دارای فرکانس ۳,۰ گیگاهرتز باشد، به این معناست که CPU سه میلیارد بار در ثانیه این سؤال را مطرح می‌کند: «گام بعدی چیست؟» بنابراین باید یاد بگیرید با سرعت بالا با CPU ارتباط برقرار کنید.
- **حافظه‌ی اصلی** برای ذخیره‌ی اطلاعاتی استفاده می‌شود که CPU به‌سرعت به آن‌ها نیاز دارد. سرعت حافظه‌ی اصلی تقریباً با CPU برابری می‌کند، اما داده‌های ذخیره‌شده در آن با خاموش شدن رایانه از بین می‌روند.
- **حافظه‌ی ثانویه** نیز برای ذخیره‌سازی اطلاعات به‌کار می‌رود، اما سرعت آن بسیار کمتر از حافظه‌ی اصلی است. مزیت حافظه‌ی ثانویه در این است که اطلاعات را حتی در نبود برق حفظ می‌کند. نمونه‌هایی از حافظه‌ی ثانویه شامل دیسک درایورها یا حافظه‌های فلش (که در USB و پخش‌کننده‌های موسیقی همراه یافت می‌شوند) است.

- **دستگاه‌های ورودی و خروجی** شامل صفحه‌نمایش، صفحه‌کلید، ماوس، میکروفون، بلندگو، تاچ‌پد و دیگر ابزارهایی هستند که ما از طریق آن‌ها با رایانه تعامل داریم.
- امروزه اکثر رایانه‌ها دارای **اتصال شبکه** نیز هستند تا اطلاعات را از طریق اینترنت یا شبکه‌های محلی دریافت کنند. می‌توان شبکه را به‌عنوان محل ذخیره‌سازی بسیار کند و گاه ناپایدار اطلاعات در نظر گرفت. بنابراین، شبکه نیز نوعی حافظه‌ی ثانویه، اما با سرعت پایین و قابلیت اطمینان محدود، به حساب می‌آید.

هرچند جزئیات نحوه‌ی کار این اجزا بیشتر در حوزه‌ی سازندگان سخت‌افزار قرار می‌گیرد، اما داشتن دانش اولیه و آگاهی در مورد واژگان مرتبط، برای درک بهتر مفاهیم در هنگام برنامه‌نویسی ضروری است.

به‌عنوان یک برنامه‌نویس، وظیفه‌ی شما استفاده از این منابع و ایجاد هماهنگی میان آن‌ها برای حل مسئله و یا تحلیل داده‌های به‌دست‌آمده است. بیشتر اوقات، شما در حال «گفت‌وگو» با CPU خواهید بود و به آن دستور می‌دهید که گام بعدی چیست. گاهی نیز لازم است از CPU بخواهید از حافظه‌ی اصلی، حافظه‌ی ثانویه، شبکه یا ابزارهای ورودی و خروجی استفاده کند.




شکل شماره ۴: شما کجا هستید؟

شما باید کسی باشید که به سوال پرتکرار CPU «گام بعدی چیست؟» پاسخ می‌دهد. اما به‌سختی می‌توان تصور کرد که شما به اندازه‌ی ۵ میلی‌متر کوچک شوید و وارد رایانه شوید تا بتوانید سه میلیارد بار در ثانیه دستور صادر کنید! پس در عوض، باید دستورالعمل‌های خود را از پیش بنویسید. ما این دستورالعمل‌های ذخیره‌شده را «برنامه» می‌نامیم، و عمل نوشتن این دستورها و اصلاح آن‌ها را «برنامه‌نویسی».

## ۱.۳ درک برنامه‌نویسی

در ادامه‌ی این کتاب، تلاش خواهیم کرد تا شما را به فردی ماهر در هنر برنامه‌نویسی تبدیل کنیم. در پایان، شما یک برنامه‌نویس خواهید بود – شاید نه به‌عنوان یک برنامه‌نویس حرفه‌ای، اما دست‌کم دارای مهارتی خواهید بود که بتوانید به مسئله‌ای در حوزه‌ی تحلیل داده یا اطلاعات نگاه کرده و برنامه‌ای برای حل آن مسئله ارائه دهید.

در واقع، برای برنامه‌نویس شدن به دو مهارت نیاز دارید:

- **نخست، باید زبان برنامه‌نویسی (پایتون) را بیاموزید** .باید با واژگان و اصول نگارش این زبان آشنا شوید. لازم است بتوانید کلمات را به درستی بنویسید و بدانید چگونه «جملات» درست و قابل فهمی در این زبان بسازید.
- **دوم، باید بتوانید «داستانی را روایت کنید»** هنگام نوشتن یک داستان، کلمات و جملات را ترکیب می‌کنید تا ایده‌ای را به خواننده منتقل کنید. در این مسیر، مهارت و هنر روایت اهمیت زیادی دارد. مهارتی که با تمرین و دریافت بازخورد رشد می‌کند. در برنامه‌نویسی، «برنامه» نقش داستان را ایفا می‌کند و «مسئله‌ای که می‌خواهید حل کنید»، همان ایده‌ای است که باید منتقل شود.

زمانی که یک زبان برنامه‌نویسی مانند پایتون را یاد بگیرید، آموختن زبان‌های دیگر مانند جاوااسکریپت یا C++ برایتان بسیار ساده‌تر خواهد بود. هر زبان برنامه‌نویسی کلمات و اصول نگارش خاص خود را دارد، اما مهارت‌های حل مسئله در میان همه‌ی زبان‌ها مشترک‌اند.

شما واژگان و ساختار جملات زبان پایتون را نسبتاً سریع خواهید آموخت. اما توانایی نوشتن برنامه‌ای منسجم برای حل یک مسئله‌ی جدید نیاز به زمان بیشتری دارد. ما آموزش برنامه‌نویسی را همانند آموزش نوشتن پیش می‌بریم: ابتدا برنامه‌ها را می‌خوانیم و تحلیل می‌کنیم، سپس برنامه‌های ساده می‌نویسیم، و به‌تدریج برنامه‌های پیچیده‌تری خلق می‌کنیم. در نقطه‌ای از مسیر، شما به «سبک برنامه‌نویسی» خودتان می‌رسید؛ الگوها را تشخیص می‌دهید و به‌طور طبیعی درمی‌یابید که چگونه از یک مسئله به یک برنامه‌ی کاربردی برسید و وقتی به آن مرحله برسید، برنامه‌نویسی به فعالیتی بسیار لذت‌بخش و خلاقانه تبدیل خواهد شد.

در ادامه، از کلمات و ساختار برنامه‌های پایتون آغاز خواهیم کرد. صبور باشید، چرا که مثال‌های ساده‌ی ابتدایی شما را به یاد روزهایی خواهند انداخت که تازه خواندن را آغاز کرده بودید.

## ۱.۴ کلمات و جملات

برخلاف زبان انسان، واژگان پایتون بسیار محدود هستند. ما به این "واژگان" کلمات رزروشده یا **کلیدواژه‌ها (keywords)** می‌گوییم. این‌ها کلماتی هستند که برای پایتون معنی بسیار خاصی دارند. زمانی که پایتون این کلمات را در یک برنامه ببیند، از آن‌ها فقط یک معنی مشخص دریافت می کند.

شما در حین برنامه نویسی، می‌توانید کلمات دلخواه خود را به برنامه اضافه کنید که به آن‌ها **متغیر (variable)** می‌گوییم. شما در انتخاب نام برای متغیرها آزادی زیادی دارید، اما نمی‌توانید از کلمات رزروشده‌ی پایتون به عنوان نام متغیر استفاده کنید.

وقتی یک سگ را آموزش می‌دهیم، از کلماتی خاص مثل "بنشین"، "بمان" و "بیار" استفاده می‌کنیم. وقتی با سگ صحبت می‌کنید و هیچ‌کدام از این کلمات را به‌کار نمی‌برید، سگ فقط با چشمان گیج به شما نگاه می‌کند، تا زمانی که یکی از کلمات آشنا را بشنود. مثلاً اگر بگویید: "ای کاش مردم برای بهبود سلامتی بیشتر پیاده روی داشته باشند"، آن‌چه که سگ احتمالاً می‌شنود چیزی شبیه "فلان فلان فلان فلان فلان پیاده‌روی فلان فلان فلان "است. چون "پیاده‌روی" یک کلمه‌ی رزروشده در زبان سگ است. خیلی‌ها معتقدند که زبان بین انسان‌ها و گربه‌ها اصلاً کلمه‌ی رزروشده‌ای ندارد.

کلمات رزروشده که با استفاده از آنها با پایتون صحبت می کنیم شامل موارد زیر هستند:

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

همین! و برخلاف یک سگ، پایتون کاملاً آموزش‌دیده است. وقتی بگویید"try" ، پایتون آن را دقیقاً اجرا می‌کند و هیچ وقت اشتباه نمی کند.

ما این کلمات رزروشده (کلیدواژه‌ها) و نحوهی استفاده از آن‌ها را به موقع خواهیم آموخت، اما در حال حاضر تمرکز ما بر معادل "speak" در زبان پایتون است (مشابه فرمان "صحبت کن" از انسان به سگ). نکتهی جالب دربارهی دستور "صحبت کن" در پای تون این است که می‌توانیم به‌راحتی به آن بگوییم چه چیزی را بگوید؛ فقط کافی است پیام مورد نظر خود را درون علامت نقل‌قول (" ") قرار دهیم.

```
print('Hello world!')
```

و ما نخستین برنامه خود را به زبان پایتون نوشته‌ایم. جملهی ما با تابع print آغاز می‌شود و پس از آن، یک رشته ی متنی دلخواه قرار دارد که درون علامت نقل‌قول تکی قرار گرفته است.رشته‌های متنی در دستورات print درون علامت نقل‌قول نوشته می‌شوند.علامت نقل‌قول تکی (') و نقل‌قول دوتایی (") عملکرد یکسانی دارند؛ بیشتر افراد از نقل‌قول تکی استفاده می‌کنند، مگر در مواردی که درون رشته، از نقل‌قول تکی (که همچنین به‌عنوان آپاستروف نیز شناخته می‌شود) استفاده شده باشد.

## ۱.۵ گفتگو با پایتون

اکنون که یک واژه و یک جمله ی ساده را به زبان پایتون آموخته‌ایم، لازم است بدانیم چگونه می‌توان با پایتون وارد گفت‌وگو شد تا مهارت‌های زبانی جدید خود را آزمایش کنیم. پیش از آن‌که بتوانید با پایتون گفت‌وگو کنید، ابتدا باید نرم‌افزار پایتون را روی رایانه ی خود نصب کرده و روش اجرای آن را بیاموزید. از آن‌جا که این موضوع شامل جزئیات زیادی است و فراتر از حوصله ی این فصل است، پیشنهاد می‌کنم به وب‌سایت www.py4e.com مراجعه کنید. در آن‌جا راهنمای گام‌به‌گام و ویدئوهای آموزشی متعددی برای نصب و اجرای پایتون در سیستم‌عامل‌های مکینتاش و ویندوز ارائه شده است.

در ادامه، شما وارد محیط «Terminal» برای مکینتاش یا «CMD» برای ویندوز خواهید شد و با نوشتن عبارت python، مفسر پایتون به صورت تعاملی اجرا خواهد شد و نمایی مشابه تصویر زیر را نمایش خواهد داد:

```
Python 3.11.6 (main, Nov  2 2023, 04:39:43)
   [Clang 14.0.3 (clang-1403.0.22.14.1)] on darwin
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

علامت <<< نشانگر این است که مفسر پایتون از شما می‌پرسد: «برای مرحله بعد چه کاری می خواهی انجام بدهم؟» پایتون آماده است تا با شما وارد گفت‌وگو شود.تنها کاری که شما باید انجام دهید این است که بیاموزید چگونه به زبان پایتون صحبت کنید.

بیایید فرض کنیم که حتی ساده‌ترین واژگان یا جملات زبان پایتون را نیز نمی‌دانید.در این حالت، ممکن است بخواهید از جمله‌ای استاندارد استفاده کنید؛ همان جمله‌ای که فضانوردان هنگام فرود بر سیاره‌ای دوردست و تلاش برای برقراری ارتباط با ساکنان آن سیاره به‌کار می‌برند:

```
>>> I come in peace, please take me to your leader
File "<stdin>", line 1
    I come in peace take me tou your leader
      ^^^^
SyntaxError: invalid syntax
>>>
```

اوضاع چندان خوب پیش نمی‌رود.اگر سریع به فکر چاره نباشید، ساکنان این سیاره احتمالاً با نیزه‌هایشان به شما حمله خواهند کرد، شما را به سیخی خواهند کشید، روی آتش کباب می‌کنند و برای شام نوش جان خواهند کرد!

خوشبختانه، نسخه‌ای از این کتاب را در سفر با خود آورده‌اید. به سرعت به این صفحه رجوع می‌کنید و دوباره تلاش می‌کنید:

```
>>> print('Hello world!')
Hello world!
```

This is looking much better, so you try to communicate some more:

```
>>> print('You must be the legendary god that comes from the sky')
You must be the legendary god that comes from the sky
>>> print('We have been waiting for you for a long time')
We have been waiting for you for a long time
>>> print('Our legend says you will be very tasty with mustard')
Our legend says you will be very tasty with mustard
>>> print 'We will have a feast tonight unless you say
  File "<stdin>", line 1
    print 'We will have a feast tonight unless you say
          ^
SyntaxError: unterminated string literal (detected at line 1)
>>>
```

گفت‌وگو داشت به‌خوبی پیش می‌رفت، اما شما مرتکب یک اشتباه خیلی کوچک در زبان پایتون شدید و پایتون دوباره نیزه را بیرون کشید!

در این مرحله باید به این نکته نیز پی ببرید که اگرچه پایتون زبانی بسیار پیچیده، قدرتمند و دقیق از نظر نگارشی (syntax) است، اما هوشمند نیست.در حقیقت، شما صرفاً در حال گفت‌وگو با خودتان هستید — البته با رعایت دقیق قواعد نگارشی این زبان.

هنگامی که شما از برنامه‌ای استفاده می‌کنید که فرد دیگری آن را نوشته است، گفت‌وگو در واقع بین شما و آن برنامه‌نویس انجام می‌شود، با این تفاوت که پایتون نقش میانجی را ایفا می‌کند.پایتون ابزاری است که به سازندگان برنامه‌ها این امکان را می‌دهد که نشان دهند این مکالمه چطور باید باشد و به چه سمتی باید پیش برود.

با مطالعه چند فصل دیگر، شما نیز یکی از همان برنامه‌نویسان خواهید بود که از پایتون برای صحبت با کاربران برنامه ی خود استفاده می‌کنند.

پیش از آن‌که نخستین گفت‌وگوی خود با مفسر پایتون را به پایان ببرید، بهتر است روش صحیح خداحافظی با ساکنان سیاره ی پایتون را نیز بدانید:

```
>>> good-bye
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'good' is not defined
>>> if you don't mind, I need to leave
  File "<stdin>", line 1
    if you don't mind, I need to leave
             ^
SyntaxError: unterminated string literal (detected at line 1)
>>> quit()
```

اگر دقت کنید متوجه خواهید شد که پیام خطا در دو تلاش نخستِ نادرست، با یکدیگر متفاوت است. دلیل تفاوت در پیام دوم این است که `if` یک واژه ی رزروشده (کلیدواژه) در زبان پایتون است. پایتون با دیدن این واژه ی خاص تصور کرد که شما قصد دارید چیزی بیان کنید، اما شما ساختار جمله را به‌درستی رعایت نکرده‌اید.

روش صحیح خداحافظی با پایتون این است که پس از علامت <<<، دستور `quit()` را وارد کنید. احتمالاً مدت زیادی طول می‌کشید تا خودتان به این دستور پی ببرید، پس داشتن این کتاب برای شما بدون شک مفید خواهد شد.

## ۱.۶ اصطلاحات: مفسر و کامپایلر

پایتون یک **زبان برنامه‌نویسی سطح بالا** است. این به این معناست که طراحی آن به‌گونه‌ای است که خواندن و نوشتن آن برای انسان‌ها نسبتاً ساده باشد. از دیگر زبان‌های برنامه‌نویسی سطح بالا می‌توان به **جاوا، C++، پی‌اچ‌پی، روبی، بیسیک، پرل، و جاوااسکریپت** اشاره کرد.

با این حال، رایانه‌ها این زبان‌ها را به‌صورت مستقیم درک نمی‌کنند .**واحد پردازش مرکزی(CPU)** ، که به‌منزله مغز رایانه عمل می‌کند، تنها زبان خاصی را به نام **زبان ماشین** درک می کند.

**زبان ماشین** بسیار ساده اما خسته‌کننده است، زیرا تنها با **صفر و یک** (کد باینری) نمایش داده می‌شود. نوشتن برنامه به این زبان برای انسان‌ها دشوار است. به همین دلیل از زبان‌های سطح بالا مانند پایتون استفاده می‌شود تا برنامه‌نویسی آسان‌تر شود.

اما برای آنکه رایانه بتواند این زبان‌ها را درک کند، نیاز است کدهای نوشته‌شده به زبان ماشین ترجمه شوند. در اینجا مفاهیمی مانند **مفسر (Interpreter)** و **کامپایلر (Compiler)** وارد عمل می‌شوند؛ این ابزارها وظیفه تبدیل کدهای سطح بالا به زبانی قابل فهم برای CPU را بر عهده دارند.

```
00101000111010010010101000000001111
11100110000011101010010101101101
...
```

زبان ماشین در نگاه اول ساده به نظر می‌رسد، چرا که تنها از **صفر و یک** تشکیل شده است. اما در واقع، **ساختار دستوری (syntax)** آن بسیار پیچیده‌تر و دشوارتر از زبان‌هایی مانند پایتون است. به همین دلیل، تنها تعداد اندکی از برنامه‌نویسان به‌طور مستقیم با زبان ماشین کار می‌کنند.

در عوض، ما **مترجم‌هایی (Translators)** می‌سازیم که به برنامه‌نویسان اجازه می‌دهند تا به زبان‌های سطح بالا مانند **پایتون یا جاوا اسکریپت** برنامه‌نویسی کنند. این مترجم‌ها، کدهای نوشته‌شده را به زبان ماشین تبدیل می‌کنند تا پردازنده (CPU) بتواند آن‌ها را اجرا کند.

از آنجا که **زبان ماشین** به‌طور مستقیم با **سخت‌افزار رایانه** مرتبط است، این زبان قابلیت **جابجایی (Portability)** بین سخت‌افزارهای مختلف را ندارد. به عبارت دیگر، کدی که برای یک نوع سخت‌افزار نوشته شده، معمولاً روی سخت‌افزاری دیگر قابل اجرا نیست.در مقابل، برنامه‌هایی که با **زبان‌های سطح بالا** نوشته می‌شوند، این امکان را دارند که بین رایانه‌های مختلف منتقل شوند. این انتقال یا با استفاده از **مفسر مناسب در رایانه جدید** انجام می‌پذیرد، یا با **کامپایل مجدد برنامه** برای تولید نسخه‌ای از کد ماشین متناسب با سخت‌افزار جدید.

مترجم‌های زبان‌های برنامه‌نویسی به‌طور کلی به دو دسته اصلی تقسیم می‌شوند:

۱ .**مفسرها(Interpreters)**
۲ .**کامپایلرها(Compilers)**

**مفسر (Interpreter)** برنامه را همان‌گونه که توسط برنامه‌نویس نوشته شده است، می‌خواند، آن را **تجزیه (Parse)** می‌کند و **دستورها را به‌صورت هم‌زمان (On the fly)** تفسیر و اجرا می‌کند. پایتون نمونه‌ای از زبان های مفسری است. هنگامی که ما پایتون را به‌صورت **تعاملی (Interactive)** اجرا می‌کنیم، یک خط کد (مانند یک جمله) وارد کنیم و پایتون بلافاصله آن را پردازش کرده و برای دریافت خط بعدی آماده می‌شود.

برخی از خطوط برنامه در پایتون اعلام می‌کنند که ما می‌خواهیم **مقداری را برای استفاده در آینده ذخیره کنیم** .برای این منظور، لازم است **نامی برای آن مقدار انتخاب کنیم** تا بتوانیم بعدها با استفاده از آن نام، مجدداً به مقدار ذخیره‌شده دسترسی پیدا کنیم.ما از واژه‌ی **متغیر (Variable)** برای اشاره به **برچسب‌هایی** استفاده می‌کنیم که به داده‌های ذخیره‌شده نسبت داده می‌شوند. متغیرها در واقع نمادهایی هستند که به ما کمک می‌کنند تا مقادیر را نگه‌داری و در مواقع لازم بازیابی کنیم.

```
>>> x = 6
>>> print(x)
6
>>> y = x * 7
>>> print(y)
42
>>>
```

در این مثال، از پایتون می‌خواهیم که مقدار **شش** را به خاطر بسپارد و برای این منظور، **برچسب x** را به آن اختصاص می‌دهیم تا بتوانیم در آینده آن مقدار را بازیابی کنیم. سپس با استفاده از دستور print بررسی می‌کنیم که آیا پایتون واقعاً مقدار مورد نظر را به خاطر سپرده است یا نه. در ادامه، از پایتون می‌خواهیم مقدار موجود در **x را بازیابی کرده و در عدد هفت ضرب کند** .نتیجه این عملیات را در متغیر جدیدی به نام **y** قرار می‌دهیم. در پایان نیز از پایتون می‌خواهیم مقدار فعلی ذخیره‌شده در **y** را چاپ کند.

اگرچه این دستورات را **به‌صورت خط به خط** در پایتون وارد می‌کنیم، اما پایتون آن‌ها را به صورت **توالی منظمی از دستورات** در نظر می‌گیرد؛ به‌گونه‌ای که دستورات بعدی می‌توانند از داده‌هایی که در دستورات قبلی ایجاد شده‌اند، استفاده کنند. در واقع، ما در حال نوشتن **اولین پاراگراف ساده‌ی خود** هستیم؛ پاراگرافی شامل چهار جمله که با ترتیبی **منطقی و معنادار** کنار هم قرار گرفته‌اند.

**ماهیت یک مفسر (Interpreter)** این است که بتواند به صورت **تعاملی و خط به‌خط** با کاربر وارد گفتگو شود، همان‌گونه که در مثال‌های قبلی مشاهده کردید. در مقابل، **کامپایلر (Compiler)** نیاز دارد که **کل برنامه به‌صورت یکجا در قالب یک فایل** به آن داده شود. سپس کامپایلر فرآیندی را آغاز می‌کند که طی آن، **کد سطح بالا** را به **زبان ماشین** ترجمه می‌کند. پس از اتمام این فرآیند، کامپایلر **نسخه‌ی ترجمه‌شده به زبان ماشین را در یک فایل جداگانه ذخیره می‌کند** تا در زمان مناسب اجرا شود.

اگر از **سیستم‌عامل ویندوز** استفاده می‌کنید، معمولاً فایل‌هایی که حاوی **برنامه‌های قابل اجرا به زبان ماشین** هستند، دارای پسوندهایی مانند **".exe"** یا **".dll"** می‌باشند. این پسوندها به ترتیب مخفف واژه‌های **"Executable" و "Dynamic Link Library"** هستند. در سیستم‌عامل‌های **لینوکس** و **مکینتاش(macOS)** ، فایل‌های اجرایی معمولاً **پسوند مشخصی ندارند** که آن‌ها را به‌طور یکتا به‌عنوان فایل اجرایی معرفی کند.

اگر بخواهید یک فایل اجرایی را با استفاده از یک **ویرایشگر متنی (Text Editor)** باز کنید، محتوای آن به شکل کاملاً به‌هم‌ریخته و **غیرقابل‌خواندن** نمایش داده خواهد شد؛ زیرا زبان ماشین برای انسان قابل فهم نیست و تنها توسط پردازنده (CPU) تفسیر می‌شود.

```
^?ELF^A^A^A^@^@^@^@^@^@^@^@^@^B^@^C^@^A^@^@^@\xa0\x82
^D^H4^@^@^@\x90^]^@^@^@^@^@^@4^@ ^@^G^@(^@$^@!^@^F^@
^@^@4^@^@^@4\x80^D^H4\x80^D^H\xe0^@^@^@\xe0^@^@^@^E
^@^@^@^D^@^@^@^C^@^@^@^T^A^@^@^T\x81^D^H^T\x81^D^H^S
^@^@^@^S^@^@^@^D^@^@^@^A^@^@^@^A\^D^HQVhT\x83^D^H\xe8
....
```

خواندن و نوشتن به **زبان ماشین** کار راحتی نیست؛ به همین دلیل، وجود **مفسرها (Interpreters)** و **کامپایلرها (Compilers)** نعمت بزرگی بحساب می آید، چرا که به ما این امکان را می‌دهند تا بتوانیم به زبان‌های سطح بالایی مانند **پایتون** یا **C** برنامه‌نویسی کنیم.

اکنون که با مفاهیم کامپایلر و مفسر آشنا شده‌اید، ممکن است این پرسش برایتان پیش بیاید که **خودِ مفسر پایتون چگونه نوشته شده است؟ مفسر پایتون با چه زبانی نوشته شده است؟ آیا پایتون با یک زبان کامپایل‌شونده نوشته شده است؟ وقتی در ترمینال یا خط فرمان کلمه‌ی "python" را تایپ می‌کنیم، دقیقاً چه اتفاقی می‌افتد؟** این پرسش‌ها بسیار مهم و پایه‌ای هستند و ما را به درک عمیق‌تری از ساختار زبان پایتون و نحوه اجرای برنامه‌ها رهنمون می‌سازند.

**مفسر پایتون** با استفاده از یک **زبان سطح بالا به نام «C»** نوشته شده است. شما می‌توانید کد منبع مفسر پایتون را با مراجعه به وب‌سایت رسمی آن به نشانی **www.python.org** مشاهده کنید و به بخش مربوط به **کد منبع (Source Code)** دسترسی پیدا کنید. بنابراین، پایتون خود نیز در واقع یک **برنامه** است که به زبان C نوشته شده و سپس **به زبان ماشین کامپایل شده است** .زمانی که شما پایتون را روی رایانه خود نصب کردید (یا شرکت سازنده‌ی سیستم عامل آن را برایتان نصب کرده است)، در حقیقت یک نسخه‌ی **ترجمه‌شده و قابل اجرای** برنامه‌ی پایتون را روی سیستم خود کپی کرده‌اید.

در سیستم‌عامل **ویندوز**، فایل اجرایی مربوط به کد ماشین پایتون معمولاً در قالب فایلی با نامی مشابه زیر ذخیره می‌شود:

```
C:\Python35\python.exe
```

That is more than you really need to know to be a Python programmer, but sometimes it pays to answer those little nagging questions right at the beginning.

دانستن این مطالب، بیش از آن چیزی است که برای برنامه‌نویس پایتون بودن به آن نیاز دارید، اما گاهی اوقات در ابتدای مسیر یادگیری پاسخ دادن به همین پرسش‌های کوچک، بسیار ارزشمند خواهد بود.

## ۱.۷ نوشتن یک برنامه

تایپ کردن دستورات در مفسر پایتون، روش بسیار خوبی برای **آزمایش ویژگی‌ها و قابلیت‌های زبان پایتون** است؛ اما برای **حل مسائل پیچیده‌تر**، این روش توصیه نمی‌شود.

وقتی می‌خواهیم یک برنامه واقعی بنویسیم، از یک **ویرایشگر متنی (Text Editor)** استفاده می‌کنیم تا دستورات پایتون را درون یک **فایل** وارد کنیم. به این فایل‌ها، در اصطلاح، **اسکریپت (Script)** گفته می‌شود. بر اساس یک **قرارداد عمومی**، اسکریپت‌های پایتون معمولاً نام‌هایی دارند که **به `.py` ختم می‌شوند.**

برای اجرای یک اسکریپت، باید نام فایل را به مفسر پایتون اعلام کنید. می توانید در پنجره‌ی دستورات(Command Window) ، این کار را با نوشتن دستور زیر انجام دهید:

```
$ cat hello.py
print('Hello world!')
$ python hello.py
Hello world!
```

علامت **`$` نشانه‌ی اعلان سیستم‌عامل** (Operating System Prompt) است، و دستور `cat hello.py` محتوای فایل **`hello.py`** را نمایش می‌دهد. این فایل حاوی یک **برنامه‌ی یک‌خطی پایتون** است که یک رشته متنی را چاپ می‌کند.

در اینجا ما مفسر پایتون را فراخوانی می‌کنیم و به آن می‌گوییم که **کد منبع خود را از فایل `hello.py` بخواند**، به‌جای آن‌که مانند حالت تعاملی، خط به خط از ما دستور دریافت کند.

همچنین اگر توجه کرده باشید **نیازی به استفاده از دستور `quit()` در انتهای برنامه پایتون** موجود در فایل نیست. زیرا هنگامی که پایتون کد را از یک فایل می‌خواند، به‌طور خودکار در پایان فایل می‌فهمد که اجرای برنامه باید به پایان برسد.

## ۱.۸برنامه چیست؟

در ساده‌ترین و بنیادی‌ترین تعریف، یک **برنامه، دنباله‌ای از دستورات پایتون** است که به صورت هدفمند با دقت برای انجام کاری نوشته شده‌اند. حتی اسکریپت ساده‌ی ما یعنی **`hello.py`** نیز **یک برنامه محسوب می‌شود.** گرچه این برنامه تنها **یک خط** دارد و کاربرد خاصی ندارد، اما **یک برنامه‌ی پایتون به‌شمار می‌آید.**

درک مفهوم «برنامه» ممکن است زمانی آسان‌تر شود که **به یک مسئله فکر کنیم**، که آیا برنامه‌ای می‌تواند برای حل آن نوشته شود، و سپس **نمونه‌ای از یک برنامه را ببینیم** که واقعاً آن مسئله را حل می‌کند.

فرض کنیم که شما در حال انجام **پژوهشی** بر روی **پست‌های فیس‌بوک** هستید و علاقه‌مندید بدانید **کدام واژه در میان مجموعه‌ای از پست‌ها بیش از همه تکرار شده است.** می‌توانید همه پست‌ها را چاپ کنید و شخصاً تک‌تک واژه‌ها را بررسی کرده و واژه‌ی پرتکرار را بیابید. اما این کار نه تنها **زمان‌بر** است، بلکه **احتمال خطا در آن بسیار بالا** خواهد بود. در عوض، راه

هوشمندانه این است که **یک برنامه‌ی پایتون بنویسید** تا این وظیفه را **سریع و دقیق** انجام دهد و شما نیز بتوانید آخر هفته‌تان را با کارهای دلخواه و تفریحی سپری کنید!

برای مثال، به متن ساده‌ی زیر در مورد یک دلقک و یک ماشین توجه کنید. به این متن نگاه کنید و سعی کنید واژه‌ای را که بیش از بقیه تکرار شده و تعداد تکرار آن را مشخص نمایید:

```
the clown ran after the car and the car ran into the tent
and the tent fell down on the clown and the car
```

حالا تصور کنید که قرار است این کار را نه فقط روی یک متن کوتاه، بلکه بر روی **میلیون‌ها خط متن** انجام دهید.صادقانه بگوییم، در چنین شرایطی **یادگیری زبان پایتون و نوشتن یک برنامه برای شمارش واژه‌ها، بسیار سریع‌تر و کارآمدتر از آن است که بخواهید واژه‌ها را به‌صورت دستی بررسی و شمارش کنید.**

من **یک برنامه‌ی ساده نوشته‌ام** که می‌تواند **پرتکرارترین واژه** را در یک فایل متنی پیدا کند. این برنامه را **نوشته‌ام، آزمایش کرده‌ام**، و اکنون آن را در اختیار شما قرار می‌دهم تا بتوانید **در وقت خود صرفه‌جویی** کنید.

```
name = input('Enter file: ')
handle = open(name, 'r')
counts = dict()

for line in handle:
    words = line.split()
    for word in words:
        counts[word] = counts.get(word, 0) + 1

bigcount = None
bigword = None
for word, count in list(counts.items()):
    if bigcount is None or count > bigcount:
        bigword = word
        bigcount = count

print(bigword, bigcount)

# Code: https://www.py4e.com/code3/words.py
```

شما حتی نیازی ندارید که زبان برنامه‌نویسی پایتون را بلد باشید تا بتوانید از این برنامه استفاده کنید . برای درک کامل تکنیک‌های شگفت‌انگیز پایتون که در ساخت این برنامه به‌کار رفته‌اند، باید تا فصل دهم این کتاب پیش بروید .شما به‌عنوان **کاربر نهایی** فقط کافی‌ست از برنامه استفاده کنید و از هوشمندی آن و زمانی که برایتان ذخیره کرده، شگفت‌زده شوید .کافی است کد برنامه را در فایلی به نام `words.py` وارد کرده و آن را اجرا کنید، یا اینکه مستقیماً کد منبع را از این آدرس دانلود و اجرا کنی http://www.py4e.com/code3/

این یک نمونه‌ی خوب است از اینکه چگونه **پایتون و زبان برنامه‌نویسی پایتون به عنوان واسطه‌ای** بین شما (کاربر نهایی) و من (برنامه‌نویس) عمل می‌کنند .پایتون ابزاری است که ما با استفاده از آن می‌توانیم دنباله‌ای از دستورالعمل‌های مفید (یعنی برنامه‌ها) را **به زبانی مشترک** رد و بدل کنیم—زبانی که هر کسی می‌تواند با نصب پایتون روی رایانه‌اش از آن استفاده کند. در واقع، **هیچ‌کدام از ما مستقیماً با پایتون حرف نمی‌زنیم**—بلکه از طریق پایتون، داریم با هم ارتباط برقرار می‌کنیم.

## ۱.۹ اجزای سازنده‌ی برنامه‌ها

در فصل‌های بعدی، با **واژگان، ساختار جمله، ساختار پاراگراف و حتی ساختار داستانی** در زبان پایتون بیشتر آشنا خواهیم شد. ما یاد می‌گیریم که پایتون چه توانایی‌های قدرتمندی دارد و چگونه می‌توانیم این قابلیت‌ها را **با هم ترکیب کنیم** تا برنامه هایی مفید بسازیم.

در برنامه‌نویسی، **الگوهای مفهومی سطح پایینی** وجود دارند که برای ساختن برنامه‌ها از آن‌ها استفاده می‌کنیم .این ساختارها مخصوص پایتون نیستند بلکه در **همه‌ی زبان‌های برنامه‌نویسی**، از زبان ماشین گرفته تا زبان‌های سطح بالا، وجود دارند و پایه‌ی ساخت برنامه‌ها را تشکیل می‌دهند.

**ورودی (Input)** برای دریافت داده از «دنیای بیرون»، این داده‌ها ممکن است از یک فایل خوانده شوند یا حتی از نوعی حسگر مانند میکروفون یا موقعیت‌یاب **(GPS)** دریافت گردند. در برنامه‌های ابتدایی ما، ورودی از طریق تایپ داده‌ها توسط کاربر روی صفحه‌کلید تأمین خواهد شد.

**خروجی (Output)** نمایش نتایج برنامه بر روی صفحه‌نمایش، یا ذخیره‌سازی آن‌ها در یک فایل، و یا نوشتن آن‌ها برای دستگاهی مانند بلندگو به‌منظور پخش موسیقی یا گفتار.

**اجرای ترتیبی (Sequential Execution)** اجرای دستورات یکی پس از دیگری به همان ترتیبی که در اسکریپت با آن‌ها برخورد می‌شود.

**اجرای شرطی (Conditional Execution)** بررسی شرایط خاص و سپس اجرای یک توالی از دستورات یا صرف نظر کردن از آن‌ها.

**اجرای تکرار شونده (Repeated Execution)** اجرای مجموعه‌ای از دستورات به صورت مکرر، معمولاً همراه با تغییراتی در هر تکرار.

**استفاده‌ی مجدد (Reuse)** یکبار نوشتن مجموعه‌ای از دستورالعمل‌ها و سپس اختصاص دادن یک نام به آن مجموعه، سپس استفاده‌ی مجدد از آن مجموعه در بخش‌های مختلف برنامه در صورت نیاز.

به‌نظر می‌رسد که این موضوع بیش از حد ساده باشد تا واقعی به‌نظر برسد و البته، هرگز آن قدرها هم ساده نیست. این مانند آن است که بگوییم راه رفتن صرفاً «گذاشتن یک پا جلوی پای دیگر» است. «هنر» برنامه‌نویسی در آن است که این عناصر پایه را بارها و بارها **با هم ترکیب و درهم تنیده کنیم** تا در نهایت چیزی پدید آید که برای کاربرانش مفید باشد.

برنامه‌ی شمارش واژه که در بالا آورده شد، به صورت مستقیم از تمامی این الگوها استفاده می‌کند (به‌جز یکی).

## ۱.۱۰ چه چیزی ممکن است اشتباه پیش برود؟

همان‌طور که در نخستین تعاملات‌مان با پایتون دیدیم، هنگام نوشتن کد در پایتون باید **با دقت بسیار بالا** ارتباط برقرار کنیم. حتی کوچک‌ترین انحراف یا اشتباه باعث می‌شود که پایتون از ادامه‌ی بررسی برنامه‌تان منصرف شود.

برنامه‌نویسان مبتدی اغلب این واقعیت را که پایتون هیچ فضایی برای خطا باقی نمی‌گذارد، به عنوان مدرکی می‌بینند مبنی بر اینکه پایتون زبان بدجنس، نفرت‌انگیز و بی‌رحمی است. انگار پایتون همه را دوست دارد، اما آن‌ها را **شخصاً می‌شناسد** و ازشان **کینه به دل گرفته است.** به همین دلیل، پایتون حتی برنامه‌هایی را که به‌نظر ما کاملاً درست نوشته شده‌اند نیز رد می‌کند و آن‌ها را «نامناسب» می‌نامد(فقط برای این‌که ما را عذاب بدهد).

```
>>> primt 'Hello world!'
File "<stdin>", line 1
    primt 'Hello world!'
          ^^^^^^^^^^^^^^
SyntaxError: invalid syntax

>>> primt ('Hello world')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'primt' is not defined. Did you mean: 'print'?

>>> I hate you Python!
File "<stdin>", line 1
    I hate you Python!
      ^^^^
SyntaxError: invalid syntax
>>> if you come out of there, I would teach you a lesson
File "<stdin>", line 1
    if you come out of there, I would teach you a lesson
           ^^^^
SyntaxError: invalid syntax
>>>
```

بحث کردن با پایتون فایده‌ی چندانی ندارد. پایتون صرفاً یک **ابزار** است.هیچ احساسی ندارد و **همیشه آماده و مشتاق است** تا هر زمان که به آن نیاز داشته باشید، در خدمت شما باشد. پیغام‌های خطای آن ممکن است خشن به نظر برسند، اما در واقع

**درخواست کمک پایتون** هستند. پایتون آنچه را که شما تایپ کرده‌اید بررسی کرده، و تنها به این نتیجه رسیده که **نمی‌تواند آن را درک کند.**

پایتون بیشتر شبیه به یک **سگ وفادار** است با محبتی بی‌قید و شرط نسبت به شما، با مجموعه‌ای محدود از واژگان کلیدی که آن‌ها را درک می‌کند، و نگاهی مهربان بر چهره‌اش (که با نماد <<< نمایان می‌شود)، در حالی که منتظر است **چیزی بگویید که بتواند بفهمد.** وقتی پایتون می‌گوید: SyntaxError: invalid syntax در واقع با تکان دادن دمش می‌گوید: انگار چیزی گفتی، ولی راستش رو بخوای نفهمیدم منظورت چی بود اما لطفاً به حرف زدن با من ادامه بده.(<<<)

با پیچیده‌تر شدن برنامه‌هایتان، با سه نوع کلی از خطا روبه‌رو خواهید شد**:**

**خطاهای نحوی**(Syntax Errors) : این‌ها نخستین خطاهایی هستند که با آن‌ها مواجه می‌شوید و در عین حال آسان‌ترین برای رفع هستند**.** خطای نحوی به این معناست که شما یکی از «قوانین دستور زبان» (گرامر) زبان پایتون را نقض کرده‌اید**.**

پایتون تلاش می‌کند تا دقیقاً خط و کاراکتری را که در آن احساس سردرگمی کرده است به شما نشان دهد. اما نکته‌ی نسبتاً پیچیده در مورد خطاهای نحوی این است که گاهی اوقات، اشتباهی که باید اصلاح شود، **در خطی پیش از آن چیزی است که پایتون به شما نشان می‌دهد.** پس خط و کاراکتری که پایتون در پیام خطا مشخص می‌کند، **فقط نقطه‌ی شروعی برای بررسی شماست**، نه لزوماً محل دقیق خطا.

**خطاهای منطقی(Logic Errors)** : خطای منطقی زمانی رخ می‌دهد که برنامه‌ی شما از نظر نحوی کاملاً درست باشد، اما **اشتباهی در ترتیب دستورات** یا **نحوه‌ی ارتباط آن‌ها با یکدیگر** وجود داشته باشد.

یک مثال خوب برای درک این نوع خطا این است:

« از بطری آب خود یک جرعه بنوش، آن را در کوله‌پشتی‌ات بگذار، به کتابخانه برو، و سپس درِ بطری را ببند.»

همان‌طور که می‌بینید، این دستورها از نظر گرامری مشکلی ندارند،اما **ترتیب آن‌ها منطقی نیست** .بستن درِ بطری بعد از گذاشتن آن در کیف یا بعد از رسیدن به کتابخانه، کاری نادرست یا ناکارآمد است. در برنامه‌نویسی نیز همین‌طور است، ممکن است کد شما بدون هیچ خطای نحوی اجرا شود،اما نتیجه‌ی نهایی درست نباشد، چون منطق پشت دستورات ناقص یا اشتباه بوده است.

**خطاهای معنایی(Semantic Errors)** : خطای معنایی زمانی رخ می‌دهد که **توصیف شما از مراحل اجرای برنامه از نظر نحوی کاملاً صحیح** بوده و **ترتیب آن‌ها هم درست است**، اما با این حال، **برنامه نتیجه‌ای متفاوت از آنچه انتظار داشته‌اید تولید می‌کند.** در واقع، برنامه از نظر پایتون هیچ مشکلی ندارد و بدون خطا اجرا می‌شود،اما کاری را انجام می‌دهد که **شما نمی‌خواستید.**

یک مثال ساده:
فرض کنید به دوستی مسیر رفتن به یک رستوران را می‌دهید و می‌گویید:

« ... وقتی به تقاطع کنار پمپ بنزین رسیدی، به چپ بپیچ و یک مایل برو. رستوران ساختمانی قرمز رنگ است که در سمت چپ تو قرار دارد.»

دوست‌تان خیلی دیر می‌رسد و با شما تماس می‌گیرد و می‌گوید:

« من الان توی یک مزرعه‌ام، پشت یک طویله دارم راه می‌رم، هیچ اثری از رستوران نیست!»

شما می‌پرسید:

« مطمئنی سر پمپ بنزین به چپ پیچیدی؟»

او پاسخ می‌دهد:

« دقیقاً طبق راهنمایی شما عمل کردم، حتی نوشته‌ام رو هم دارم. گفته بودید به چپ بپیچم و یک مایل برم. »

و شما پاسخ می‌دهید:

« متأسفم... گرچه راهنمایی من از نظر ساختار کاملاً درست بود، اما متأسفانه یک **خطای معنایی کوچک اما نامشخص** داشت.»

در نهایت، در هر سه نوع خطا (چه نحوی، چه منطقی، و چه معنایی) پایتون صرفاً با تمام توان خود تلاش می‌کند **دقیقاً همان کاری را انجام دهد که شما از آن خواسته‌اید.**

## ۱.۱۱ اشکال‌زدایی(Debugging)

زمانی که پایتون خطایی را نشان می‌دهد یا حتی وقتی خروجی برنامه با آنچه شما انتظار داشتید متفاوت است، **جست‌وجو برای یافتن علت خطا آغاز می‌شود. اشکال‌زدایی** یعنی فرایند پیدا کردن دلیل بروز خطا در کد شما.

هنگامی که در حال اشکال‌زدایی یک برنامه هستید ( به ویژه اگر با یک اشکال سخت و پیچیده سروکار داری) چهار روش کلی وجود دارد که می‌توانید آن‌ها را امتحان کنید.چهار روش کلیدی برای اشکال‌زدایی:

**خواندن(Reading)** : کد خود را با دقت بررسی کنید، آن را برای خودتان با صدای بلند بخوانید، و مطمئن شوید که دقیقاً همان چیزی را می‌گوید که شما **قصد داشتید بنویسید.**

**اجرا کردن(Running)** : تغییرات ایجاد کنید و نسخه‌های مختلفی از برنامه را آزمایش کنید. اغلب اگر **در جای مناسب، چیز درست را چاپ یا نمایش دهید**، مشکل خیلی زود آشکار می‌شود. اما گاهی لازم است وقت بگذارید و یک سری «داربست موقتی» بسازید (مثلاً با چاپ مقادیر میانی) تا دید واضح‌تری از مشکل پیدا کنید.

**تأمل کردن(Ruminating)** : کمی وقت بگذارید و فکر کنید! با چه نوع خطایی مواجه‌اید؟ نحوی؟ زمان اجرا؟ معنایی؟ از پیام خطا یا خروجی برنامه **چه اطلاعاتی می‌توان گرفت**؟ چه نوع خطایی ممکن است باعث چنین رفتاری شود؟ آخرین تغییری که قبل از ظاهر شدن مشکل ایجاد کرده اید چه بوده است؟

**عقب‌نشینی(Retreating)** : گاهی بهترین کار این است که **یک گام عقب بروید**، تغییرات اخیر را یکی‌یکی لغو کنید تا به نسخه‌ای از برنامه برسید که **هم درست کار می‌کرد و هم شما آن را به‌خوبی می‌فهمیدید**. سپس از آن نقطه می‌توانید دوباره شروع به ساختن کنید.

برنامه‌نویسان مبتدی گاهی در یکی از این فعالیت‌ها گیر می‌افتند و باقی موارد را فراموش می‌کنند. یافتن یک اشکال سخت نیازمند ترکیبی از خواندن، اجرا کردن، تأمل کردن، و گاهی اوقات عقب‌نشینی است. اگر در یکی از این فعالیت‌ها به بن‌بست رسیدید، سعی کنید سراغ روش‌های دیگر بروید. هر فعالیت ممکن است به شیوه خاص خودش به شکست بینجامد.

برای مثال، خواندن کدتان ممکن است در صورتی که مشکل ناشی از یک اشتباه تایپی باشد، به شما کمک کند، اما اگر مشکل از یک سوءتفاهم مفهومی ناشی شده باشد، مفید نخواهد بود. اگر شما درک درستی از کاری که برنامه‌تان انجام می‌دهد نداشته باشید، حتی اگر ۱۰۰ بار کد را بخوانید، باز هم ممکن است نتوانید خطا را پیدا کنید، زیرا خطا در ذهن شماست.

اجرای آزمایش‌ها می‌تواند مفید باشد، به ویژه اگر آزمایش‌های کوچک و ساده‌ای انجام دهید. اما اگر بدون تفکر یا خواندن کدتان آزمایش‌ها را اجرا کنید، ممکن است وارد الگویی به نام «برنامه‌نویسی به‌صورت پیاده‌روی تصادفی» شوید. این فرآیند شامل اعمال تغییرات تصادفی است تا زمانی که برنامه درست عمل کند. واضح است که برنامه‌نویسی به‌صورت پیاده‌روی تصادفی ممکن است زمان زیادی ببرد .

باید زمانی را به تفکر اختصاص دهید. اشکال‌زدایی مانند یک علم تجربی است. شما باید حداقل یک فرضیه در مورد اینکه مشکل چیست داشته باشید. اگر دو یا چند احتمال وجود دارد، سعی کنید آزمایشی پیدا کنید که یکی از آن‌ها را رد کند. استراحت کردن در فرآیند تفکر کمک می‌کند. صحبت کردن نیز همینطور. اگر مشکل را برای کسی توضیح دهید (یا حتی به خودتان)، گاهی اوقات قبل از اینکه سؤال را تمام کنید، پاسخ را پیدا خواهید کرد.

اما حتی بهترین تکنیک‌های اشکال‌زدایی هم اگر تعداد خطاها زیاد باشد یا کدی که در حال تلاش برای اصلاح آن هستید خیلی بزرگ و پیچیده باشد، شکست خواهند خورد. گاهی بهترین گزینه این است که عقب‌نشینی کنید و برنامه را ساده کنید تا به چیزی برسید که کار کند و آن را درک کنید. برنامه‌نویسان مبتدی اغلب از عقب‌نشینی خودداری می‌کنند زیرا نمی‌توانند خطی از کد را حذف کنند (حتی اگر اشتباه باشد). اگر این کار به شما کمک می‌کند، قبل از اینکه برنامه را ساده کنید، آن را در یک فایل دیگر کپی کنید. سپس می‌توانید بخش‌ها را یکی یکی دوباره در برنامه قرار دهید.

## ۱.۱۲ سفر یادگیری

در حالی که با کتاب پیش می‌روید، نگران نباشید اگر فکر می کنید مفاهیم در ابتدا به خوبی با هم هماهنگ به نظر نرسند. وقتی که شما در حال یادگیری صحبت کردن بودید، هیچ مشکلی نبود که در چند سال اول فقط صداهای بامزه و خرخر کنید. و این کاملاً

طبیعی بود که شش ماه طول کشید تا از واژگان ساده به جملات ساده برسید و پنج یا شش سال بیشتر طول کشید تا از جملات به پاراگراف‌ها برسید و چند سال دیگر طول کشید تا بتوانید یک داستان کوتاه جذاب و کامل به طور مستقل بنویسید.

ما می‌خواهیم که شما Python را بسیار سریع‌تر یاد بگیرید، بنابراین تمام مفاهیم را به صورت همزمان در چند فصل آینده آموزش می‌دهیم. اما این شبیه یادگیری یک زبان جدید است که به زمان نیاز دارد تا آن را جذب و درک کنید تا زمانی که به طور طبیعی احساس شود. این امر منجر به کمی سردرگمی می‌شود، زیرا ما به طور مکرر به موضوعات مختلف می‌پردازیم تا به شما کمک کنیم تا تصویر کلی را ببینید در حالی که قطعات کوچکی که این تصویر بزرگ را می‌سازند را تعریف می‌کنیم. در حالی که کتاب به صورت خطی نوشته شده است و اگر در حال گذراندن دوره‌ای باشید، روند آن خطی خواهد بود، اما نگران نباشید اگر نحوه برخورد شما با مطالب غیرخطی باشد. به جلو و عقب نگاه کنید و با ذهنی باز مطالعه کنید. با مرور مطالب پیشرفته‌تر بدون اینکه جزئیات را کاملاً درک کنید، می‌توانید درک بهتری از "چرا؟" برنامه‌نویسی پیدا کنید. با مرور مطالب قبلی و حتی انجام مجدد تمرینات اولیه، متوجه خواهید شد که در واقع مطالب زیادی را یاد گرفته‌اید، حتی اگر مطالبی که اکنون در حال مطالعه آن هستید کمی دشوار به نظر برسد.

معمولاً وقتی که در حال یادگیری اولین زبان برنامه‌نویسی خود هستید، چند لحظه‌ی شگفت‌انگیز "آها!" وجود دارد که می‌توانید لحظه ی ضربه زدن به سنگ با چکش و قلم تراش را نگاه کنید، و دور شوید، و ببینید که در واقع یک مجسمه‌ی زیبا می‌سازید.

اگر چیزی به‌ویژه سخت به نظر می‌رسد، معمولاً هیچ فایده‌ای ندارد که تمام شب را بیدار بمانید و به آن نگاه کنید. یک استراحت کنید، چرتی بزنید، چیزی بخورید، مشکلتان را برای کسی (یا شاید سگتان) توضیح دهید، و سپس با چشم‌های تازه به آن بازگردید. به شما اطمینان می‌دهم که وقتی مفاهیم برنامه‌نویسی این کتاب را یاد بگیرید، به عقب نگاه خواهید کرد و خواهید دید که همه چیز واقعاً آسان و زیبا بوده و فقط کمی زمان برد تا آن را جذب کنید.

## ۱.۱۳ واژه نامه

**خطا:(bug)** یک اشتباه در برنامه.

**واحد پردازش مرکزی:(central processing unit)** قلب هر کامپیوتری است. این واحد نرم‌افزارهایی که می‌نویسیم را اجرا می‌کند؛ همچنین به آن "CPU" یا "پردازنده" نیز گفته می‌شود.

**کامپایل:(compile)** ترجمه یک برنامه نوشته‌شده در یک زبان سطح بالا به زبان سطح پایین به طور یکجا، به منظور آماده‌سازی برای اجرای بعدی.

**زبان سطح بالا:(high-level language)** یک زبان برنامه‌نویسی مانند Python که به گونه‌ای طراحی شده است که برای انسان‌ها خوانا و قابل نوشتن باشد.

**حالت تعاملی:(interactive mode)** یک روش برای استفاده از مفسر Python با تایپ دستورات و عبارات در خط فرمان.

**تفسیر: (interpret)** اجرای یک برنامه در یک زبان سطح بالا با ترجمه خط به خط آن.

**زبان سطح پایین: (low-level language)** یک زبان برنامه‌نویسی که به گونه‌ای طراحی شده است که اجرای آن برای کامپیوترها آسان باشد؛ همچنین به آن "کد ماشین" یا "زبان اسمبلی" گفته می‌شود.

**کد ماشین: (machine code)** پایین‌ترین زبان نرم‌افزار است که مستقیماً توسط واحد پردازش مرکزی (CPU) اجرا می‌شود.

**حافظه اصلی: (main memory)** برنامه‌ها و داده‌ها را ذخیره می‌کند. حافظه اصلی اطلاعات خود را زمانی که برق قطع می‌شود از دست می‌دهد.

**تجزیه: (parse)** بررسی یک برنامه و تجزیه و تحلیل ساختار نحوی آن.

**قابلیت حمل: (portability)** ویژگی یک برنامه که به آن اجازه می‌دهد روی بیش از یک نوع کامپیوتر اجرا شود.

**تابع: print** دستوری که باعث می‌شود مفسر Python یک مقدار را روی صفحه نمایش نشان دهد.

**حل مسأله: (problem solving)** فرآیند بیان یک مشکل، یافتن یک راه‌حل، و بیان آن راه‌حل.

**برنامه: (program)** مجموعه‌ای از دستورات که محاسباتی را مشخص می‌کند.

**خط فرمان: (prompt)** زمانی که یک برنامه پیامی را نمایش می‌دهد و از کاربر می‌خواهد تا ورودی‌ای را وارد کند.

**حافظه ثانویه: (secondary memory)** برنامه‌ها و داده‌ها را ذخیره می‌کند و اطلاعات خود را حتی زمانی که برق قطع می‌شود نگه می‌دارد. معمولاً کندتر از حافظه اصلی است. از جمله حافظه‌های ثانویه می‌توان به درایوهای دیسک و حافظه فلش در USBها اشاره کرد.

**معانی: (semantics)** معنای یک برنامه.

**خطای معنایی: (semantic error)** یک اشتباه در برنامه که باعث می‌شود برنامه کاری غیر از آنچه که برنامه‌نویس قصد داشته انجام دهد، انجام دهد.

**کد منبع: (source code)** یک برنامه نوشته‌شده در یک زبان سطح بالا.

## ۱.۱۴ تمرین ها

**تمرین ۱: عملکرد حافظه ثانویه در یک کامپیوتر چیست؟**

الف) انجام تمامی محاسبات و منطق برنامه

ب) بازیابی صفحات وب از اینترنت
ج) ذخیره‌سازی اطلاعات برای مدت طولانی، حتی پس از قطع برق
د) دریافت ورودی از کاربر

**تمرین ۲: یک برنامه چیست؟**

**تمرین ۳: تفاوت بین یک کامپایلر و یک مفسر چیست؟**

**تمرین ۴: کدام یک از موارد زیر شامل "کد ماشین" است؟**

الف) مفسرPython ب) صفحه‌کلید ج) فایل کد منبعPython د) سند پردازش کلمات

**تمرین ۵: مشکلِ کُدِ زیر چیست؟**

```
>>> primt 'Hello world!'
File "<stdin>", line 1
  primt 'Hello world!'
                ^
SyntaxError: invalid syntax
>>>
```

**تمرین ۶ :پس از اجرای خط زیر در پایتون، متغیری مانند «x» در کدام بخش رایانه ذخیره می‌شود؟**

x = ۱۲۳

الف) واحد پردازش مرکزی(CPU) ب) حافظه اصلی پ) حافظه ثانویه ت) دستگاه‌های ورودی ث) دستگاه‌های خروجی

**تمرین ۷: این برنامه چه چیزی را چاپ خواهد کرد:**

```
x = 43
x = x - 1
print(x)
```

الف) ۴۳ ب) ۴۲ پ) x + ۱ ت) خطا، زیرا x = x + ۱ از نظر ریاضی ممکن نیست

**تمرین ۸: با استفاده از یک نمونه از توانایی‌های انسانی، هر یک از موارد زیر را توضیح دهید:**

۱) واحد پردازش مرکزی(CPU) ،
۲) حافظه اصلی
۳) حافظه ثانویه
۴) دستگاه ورودی
۵) دستگاه خروجی.

برای مثال: «معادل انسانیِ واحد پردازش مرکزی چیست؟»

## فصل دوم

# مقدارها، عبارات و دستورات

### ۲,۱ مقدارها و نوع‌ها

یک مقدار یکی از اساسی‌ترین چیزهایی است که یک برنامه با آن کار می‌کند، مانند یک حرف یا یک عدد. مقادیری که تاکنون دیده‌ایم عبارتند از ۱، ۲ و"Hello, World!". این مقادیر به انواع مختلفی تعلق دارند: ۲ یک عدد صحیح است و "Hello, World!"یک رشته است، چون شامل یک "رشته" از حروف است. شما (و مفسر) می‌توانید رشته‌ها را تشخیص دهید چون در داخل علامت‌های نقل قول قرار دارند.

دستورات چاپ همچنین برای اعداد صحیح نیز کار می‌کنند. ما از دستور پایتون برای شروع مفسر استفاده می‌کنیم.

```
python
>>> print(4)
4
```

اگر مطمئن نیستید که یک مقدار چه نوعی دارد، مفسر می‌تواند به شما بگوید.

```
>>> type('Hello, World!')
<class 'str'>
>>> type(17)
<class 'int'>
```

بدیهی است که رشته‌ها به نوع `str` و اعداد صحیح به نوع `int` تعلق دارند. اعداد با نقطه اعشاری به نوعی به نام `float` تعلق دارند، زیرا این اعداد در قالبی به نام ممیز شناور نمایش داده می‌شوند .

```
>>> type(3.2)
<class 'float'
```

نظرتان در مورد مقادیری مانند "۱۷" و "۳,۲" که به نظر می‌رسند اعداد هستند، اما در نقل قول قرار دارند چیست؟ بیایید از مفسر پایتون بپرسیم

```
>>> type('17')
<class 'str'>
>>> type('3.2')
<class 'str'>
```

از نظر پایتون آنها رشته هستند.

وقتی یک عدد صحیح بزرگ می‌نویسید، ممکن است وسوسه شوید که از ویرگول برای جدا کردن گروه‌های سه رقمی استفاده کنید، مانند ۱،۰۰۰،۰۰۰. این فرمت برای پایتون قابل درک است اما چیزی که پایتون از آن برداشت می کند کاملا متفاوت از قصد ما است.

```
>>> print(1,000,000)
1 0 0
```

چیزی که پایتون نمایش می دهد کاملا متفاوت از چیزی است که ما انتظار داشتیم! پایتون ۱،۰۰۰،۰۰۰ را به عنوان یک دنباله از اعداد جدا شده با ویرگول تفسیر می‌کند و آن ها را با گذاشتن فاصله بین هر کدام چاپ می‌کند.

این اولین نمونه‌ای است که از یک خطای معنایی دیده‌ایم: کد بدون نمایش پیام خطا اجرا می‌شود، اما کاری که "باید" انجام دهد را انجام نمی‌دهد.

## ۲,۲ متغیرها

یکی از قدرتمندترین ویژگی‌های زبان‌های برنامه‌نویسی توانایی کار با متغیرها است. یک متغیر نامی است که به یک مقدار اشاره می‌کند.
یک "دستور انتساب" یک متغیر را ایجاد کرده و یا مقادیری به آن اختصاص می دهد:

```
>>> message = 'And now for something completely different'
>>> n = 17
>>> pi = 3.1415926535897931
```

در این مثال سه انتساب انجام می‌شود. اولین دستور یک رشته را به متغیر جدیدی به نام `message` اختصاص می‌دهد؛ دومین دستور عدد صحیح ۱۷ را به `n` اختصاص می‌دهد؛ سومین دستور مقدار (تقریبی) $\pi$ را به `pi` اختصاص می‌دهد. برای نمایش مقدا ر یک متغیر، می‌توانید از دستور `print` استفاده کنید:

```
>>> print(n)
17
>>> print(pi)
3.141592653589793
```

نوع یک متغیر نوع مقداری است که به آن اشاره می‌کند.

```
>>> type(message)
<class 'str'>
>>> type(n)
<class 'int'>
>>> type(pi)
<class 'float'>
```

## ۲,۳ انتخاب نام متغیر و کلمات کلیدی

برنامه‌نویسان معمولاً برای متغیرهای خود نام‌هایی انتخاب می‌کنند که معنادار بوده و توضیح دهند متغیر برای چه کاری استفاده می‌شود.
نام متغیر می‌تواند به طور دلخواه طولانی باشند. آنها می‌توانند شامل حروف و اعداد باشند، اما نمی‌توانند با عدد شروع شوند. استفاده از حروف بزرگ قانونی است، اما بهتر است نام متغیرها با حروف کوچک شروع شوند. علامت زیرخط ( _ ) می‌تواند در یک نام ظاهر

شود. معمولاً از آن در نام‌های چندکلمه‌ای مانند `my_name` یا `airspeed_of_unladen_swallow` استفاده می‌شود. نام متغیر می‌تواند با زیرخط شروع شود، اما معمولاً از این کار اجتناب می‌کنیم مگر اینکه در حال نوشتن کد کتابخانه‌ای برای استفاده دیگران باشیم.

اگر نامی غیرقانونی به یک متغیر بدهید، خطای نحوی دریافت خواهید کرد:

```
>>> 76trombones = 'big parade'
SyntaxError: invalid syntax
>>> more@ = 1000000
SyntaxError: invalid syntax
>>> class = 'Advanced Theoretical Zymurgy'
SyntaxError: invalid syntax
```

`76trombones` غیرقانونی است زیرا با عدد شروع می‌شود `more@` .غیرقانونی است زیرا شامل یک کاراکتر غیرمجاز (@) است. اما مشکل با `class` چیست؟ معلوم می‌شود که `class` یکی از کلمات کلیدی پایتون است. مفسر از کلمات کلیدی برای شناسایی ساختار برنامه استفاده می‌کند و نمی‌توان از آنها به عنوان نام متغیر استفاده کرد.

پایتون ۳۵ کلمه کلیدی رزرو شده دارد:

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

ممکن است بخواهید این لیست را همیشه در دسترس داشته باشید. اگر مفسر در مورد یکی از نام‌های متغیر شما شکایت کند و شما ندانید چرا، بررسی کنید که آیا در این لیست هست یا خیر.

## ۲,۴ دستورات

یک دستور واحدی از کد است که مفسر پایتون می‌تواند آن را اجرا کند. ما دو نوع دستور دیده‌ایم: دستور `print` که برای چاپ یک عبارت است و دستور انتساب که برای نسبت دادن یک مقدار به یک متغیر است.

وقتی شما یک دستور را در حالت تعاملی تایپ می‌کنید، مفسر آن را اجرا کرده و نتیجه را نمایش می‌دهد، البته اگر نتیجه‌ای وجود داشته باشد. یک اسکریپت معمولاً شامل یک دنباله از دستورات است. اگر بیش از یک دستور وجود داشته باشد، نتایج به ترتیب اجرا شدن دستورات ظاهر می‌شوند.

به عنوان مثال، اسکریپت

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

خروجی زیر را تولید می‌کند:

```
1
2
```

نکته : دستور انتساب خروجی قابل نمایش تولید نمی‌کند.

## ۲,۵ عملگرها و عملوندها

عملگرها نمادهای ویژه‌ای هستند که محاسباتی مانند جمع و ضرب را انجام می‌دهند. مقادیری که عملگر بر آن‌ها اعمال می‌شود، عملوند نامیده می‌شوند. عملگرهای +، -، *، /، و ** به ترتیب عملیات جمع، تفریق، ضرب، تقسیم، و توان را انجام می‌دهند، همان‌طور که در مثال‌های زیر مشاهده می‌شود:

```
20+32
hour-1
hour*60+minute
minute/60
5**2
(5+9)*(15-7)
```

بین نسخه ی ۲ و ۳ پایتون، تغییری در عملگر تقسیم ایجاد شده است. در پایتون ۳، نتیجه این تقسیم به صورت یک عدد اعشاری (floating point) خواهد بود:

```
>>> minute = 59
>>> minute/60
0.9833333333333333
```

در حالی که در پایتون ۲، عملگر تقسیم، دو عدد صحیح را تقسیم می‌کرد و نتیجه را به صورت یک عدد صحیح با حذف قسمت اعشاری ارائه می‌داد:

```
>>> minute = 59
>>> minute/60
0
```

برای به‌دست آوردن همان نتیجه در پایتون ۳، برای تقسیم عدد صحیح از (عملگر //) استفاده کنید.

```
>>> minute = 59
>>> minute//60
0
```

در پایتون ۳، نتیجه تقسیم بیشتر شبیه به چیزی است که از حساب کتاب یا ماشین حساب انتظار دارید.

## ۲,۶ عبارات (Expressions)

یک عبارت ترکیبی از مقادیر، متغیرها و عملگرها است.

یک مقدار می تواند به تنهایی نیز یک عبارت محسوب شود، همچنین یک متغیر هم به‌تنهایی می‌تواند یک عبارت باشد. بنابراین، موارد زیر همگی عبارات معتبری هستند (به‌شرط آن‌که متغیر x قبلاً مقداردهی شده باشد)

```
17
x
x + 17
```

اگر یک عبارت را در حالت تعاملی (interactive mode) تایپ کنید، مفسر (interpreter) آن را ارزیابی کرده و نتیجه را نمایش می‌دهد:

```
>>> 1 + 1
2
```

اما در یک اسکریپت، یک عبارت به‌تنهایی کاری انجام نمی‌دهد! این موضوع یکی از موارد رایج سردرگمی برای مبتدیان است.

**تمرین ۱ :**دستورات زیر را در مفسر پایتون تایپ کنید تا ببینید چه کاری انجام می‌دهند:

```
5
x = 5
x + 1
```

### ۲,۷ ترتیب انجام عملیات

وقتی بیش از یک عملگر در یک عبارت ظاهر می‌شود، ترتیب ارزیابی آن به قوانین تقدم بستگی دارد. برای عملگرهای ریاضی، پایتون از قراردادهای ریاضی پیروی می‌کند. واژه‌ی اختصاری **PEMDAS** راه مفیدی برای به‌خاطر سپردن این قوانین است:

– **P •پرانتزها (Parentheses)** بالاترین اولویت را دارند و می‌توان از آن‌ها برای وادار کردن یک عبارت به ارزیابی به ترتیبی خاص استفاده کرد. از آنجا که عبارات داخل پرانتز اول ارزیابی می‌شوند، مقدار `2 * (3 - 1)` برابر با ۴ خواهد بود، و `(1 + 1) ** (5 - 2)` برابر با ۸ است. حتی اگر نتیجه تغییری نکند، می‌توان از پرانتز برای خواناتر کردن یک عبارت استفاده کرد، مثل `(minute * 100) / 60`

- **E •توان (Exponentiation)** دارای اولویت بعدی است. بنابراین، عبارت `2**1 + 1` برابر با ۳ است، نه ۴، و `3 * 1**3` برابر با ۳ است، نه ۲۷.

- **MD •ضرب و تقسیم (Multiplication and Division)** دارای اولویتی برابر هستند، که از جمع و تفریق بالاتر است. جمع و تفریق نیز اولویتی برابر دارند. بنابراین، `1 - 3 * 2` برابر با ۵ است، نه ۴، و `2 / 4 + 6` برابر با ۸ است، نه ۵.

- **AS •عملگرهایی با اولویت برابر از چپ به راست ارزیابی می‌شوند** .بنابراین عبارت `1 - 3 - 5` برابر با ۱ است، نه ۳، چون ابتدا `3 - 5` محاسبه می‌شود و سپس ۱ از آن کم می‌شود.

وقتی شک دارید، همیشه از پرانتز در عبارات خود استفاده کنید تا مطمئن شوید که محاسبات دقیقاً به ترتیبی که مد نظر دارید انجام شوند.

## ۲,۸ عملگر باقی مانده (Modulus Operator)

عملگر باقی‌مانده روی اعداد صحیح عمل می‌کند و باقیمانده‌ی تقسیم عملوند اول بر عملوند دوم را برمی‌گرداند. در پایتون، این عملگر با علامت درصد (`%`) نشان داده می‌شود. نحو (syntax) آن مانند دیگر عملگرهاست:

```
>>> quotient = 7 // 3
>>> print(quotient)
2
>>> remainder = 7 % 3
>>> print(remainder)
1
```

پس ۷ تقسیم بر ۳ برابر با ۲ است و ۱ باقی‌مانده دارد. عملگر باقی‌مانده (Modulus) در واقع بسیار مفید است. به عنوان مثال، می‌توانید بررسی کنید که آیا یک عدد بر عدد دیگری بخش‌پذیر است یا خیر: اگر `x % y` برابر صفر باشد، به این معناست که `x` بر `y` بخش‌پذیر است.

همچنین، می‌توانید آخرین رقم یا ارقام یک عدد را استخراج کنید. به عنوان مثال، `x % 10` آخرین رقم عدد `x` را در سیستم ده دهی می‌دهد. به همین ترتیب، `x % 100` دو رقم آخر عدد `x` را باز می‌گرداند.

.

## ۲,۹ عملیات روی رشته ها

عملگر `+` با رشته‌ها نیز کار می‌کند، اما این جمع با آنچه در معنای ریاضی وجود دارد تفاوت دارد. این عملگر الحاق (concatenation) را انجام می‌دهد، به این معنی که رشته‌ها را به هم متصل می‌کند و آنها را پشت سر هم می‌چسباند. برای مثال:

```
>>> first = 10
>>> second = 15
>>> print(first+second)
25
>>> first = '100'
```

```
>>> second = '150'
>>> print(first + second)
100150
```

عملگر * نیز با رشته‌ها کار می‌کند و محتوای یک رشته را با یک عدد صحیح ضرب می‌کند. خروجی نمایش رشته به تعداد عدد صحیح وارد شده می باشد.  برای مثال:

```
>>> first = 'Test '
>>> second = 3
>>> print(first * second)
Test Test Test
```

## ۲,۱۰ درخواست ورودی از کاربر

در بعضی مواقع، ممکن است بخواهیم مقدار یک متغیر را از کاربر از طریق کیبورد دریافت کنیم. پایتون یک تابع داخلی به نام input فراهم می‌کند که ورودی را از کیبورد می گیرد. وقتی این تابع فراخوانی می‌شود، برنامه متوقف می‌شود و منتظر می‌ماند تا کاربر چیزی تایپ کند. وقتی کاربر دکمه Return یا Enter را فشار دهد، برنامه ادامه پیدا می‌کند و ورودی که کاربر تایپ کرده است را به عنوان یک رشته برمی‌گرداند.

```
>>> inp = input()
Some silly stuff
>>> print(inp)
Some silly stuff
```

قبل از دریافت ورودی از کاربر، بهتر است که یک پیام راهنما چاپ کنید تا به کاربر بگویید که چه چیزی وارد کند. می‌توانید یک رشته به تابع input پاس دهید تا قبل از توقف برای دریافت ورودی، به کاربر نمایش داده شود:

```
>>> name = input('What is your name?\n')
What is your name?
Chuck
>>> print(name)
Chuck
```

دنباله \n در انتهای پیام راهنما نمایانگر کاراکتر خط جدید است که باعث ایجاد یک شکست خط می‌شود. به همین دلیل ورودی کاربر در زیر پیام راهنما ظاهر می‌شود.

اگر انتظار دارید که کاربر یک عدد صحیح وارد کند، می‌توانید سعی کنید مقدار برگشتی را با استفاده از تابع int() به یک عدد صحیح تبدیل کنید:

```
>>> prompt = 'What...is the airspeed velocity of an unladen swallow?\n'
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
17
>>> int(speed)
17
```

```
>>> int(speed) + 5
22
```

اما اگر کاربر چیزی غیر از یک رشته از اعداد وارد کند، خطا دریافت خواهید کرد:

```
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
What do you mean, an African or a European swallow?
>>> int(speed)
ValueError: invalid literal for int() with
base 10: 'What do you mean, an African or a European swallow?'
```

در ادامه خواهیم دید که چگونه می‌توانیم این نوع خطاها را مدیریت کنیم.

## ۲,۱۱ نظرات

با بزرگ‌تر و پیچیده‌تر شدن برنامه‌ها، خواندن آن‌ها نیز سخت‌تر می‌شود و اغلب سخت است که نگاهی به یک قطعه کد انداخته و بفهمیم که چه کاری انجام می‌دهد یا چرا این‌طور نوشته شده است .به همین دلیل، ایده خوبی است که یادداشت‌هایی به برنامه‌های خود اضافه کنید تا به زبان طبیعی توضیح دهید که برنامه چه کاری انجام می‌دهد. این یادداشت‌ها به نام "نظرات" شناخته می‌شوند و در پایتون با نماد # شروع می‌شوند:

```
# compute the percentage of the hour that has elapsed
percentage = (minute * 100) / 60
```

در این حالت، نظر در یک خط جداگانه قرار می‌گیرد. همچنین می‌توانید نظرات را در انتهای یک خط قرار دهید:

```
percentage = (minute * 100) / 60     # percentage of an hour
```

تمامی متن از # تا انتهای خط نادیده گرفته می‌شود و هیچ تأثیری بر روی برنامه ندارد. نظرات زمانی مفید هستند که ویژگی‌های غیر واضح کد را مستند کنند. منطقی است که فرض کنیم خواننده می‌تواند متوجه شود که کد چه کاری انجام می‌دهد؛ اما بسیار مفیدتر است که توضیح دهید چرا این کار انجام می‌شود. نظر اضافه شده در مثال زیر با کد نوشته شده تکراری است و بنابراین بی فایده است:

```
v = 5      # assign 5 to v
```

اما این نظر شامل اطلاعات مفیدی است که در کد وجود ندارد:

```
v = 5      # velocity in meters/second.
```

انتخاب نام متغیر خوب می‌تواند نیاز به کامنت‌ها را کاهش دهد، اما نام‌های طولانی ممکن است عبارات را پیچیده و آنها را سخت‌خوان کنند، بنابراین بهتر است تعادل وجود داشته باشد.

## ۲,۱۲ انتخاب نام متغیرهای به یاد ماندنی

تا زمانی که قوانین ساده نام‌گذاری متغیرها را دنبال کنید و از کلمات رزرو شده اجتناب کنید، در انتخاب نام متغیرهای خود آزادی زیادی دارید. در ابتدا، این انتخاب می‌تواند هم هنگام خواندن یک برنامه و هم هنگام نوشتن برنامه‌ها گیج‌کننده باشد. به عنوان مثال، سه برنامه زیر از نظر عملکرد مشابه هستند، اما هنگام خواندن و تلاش برای درک آن‌ها تفاوت‌های زیادی دارند.

```
a = 35.0
b = 12.50
c = a * b
print(c)
hours = 35.0
rate = 12.50
pay = hours * rate
print(pay)
x1q3z9ahd = 35.0
x1q3z9afd = 12.50
x1q3p9afd = x1q3z9ahd * x1q3z9afd
print(x1q3p9afd)
```

مفسر پایتون هر این سه برنامه ها را دقیقا یکسان می‌بیند، اما انسان‌ها این برنامه‌ها را بسیار متفاوت می‌بینند و درک می‌کنند. انسان‌ها سریع‌تر هدف برنامه دوم را درک خواهند کرد زیرا برنامه‌نویس نام متغیرهایی انتخاب کرده که بازتاب‌دهنده‌ی هدف او درباره داده‌هایی است که در هر متغیر ذخیره می‌شود.

ما به این نام متغیرهایی که با دقت انتخاب شده‌اند، «نام‌های متغیرهای به یاد ماندنی» می‌گوییم. کلمه‌ی "mnemonic" به معنی «به یاد ماندنی» است. ما نام‌های متغیر به یاد ماندنی را برای کمک به یادآوری این که چرا ابتدا آن متغیر را ایجاد کرده‌ایم، انتخاب می‌کنیم.

در حالی که همه اینها عالی به نظر می‌رسند و ایده بسیار خوبی است که از نام‌های متغیر به یاد ماندنی استفاده کنیم، این نام‌ها می‌توانند درک و تجزیه کد را برای برنامه‌نویسان مبتدی مختل کنند. این به این دلیل است که برنامه‌نویسان مبتدی هنوز کلمات رزرو شده را حفظ نکرده‌اند (که تعداد آن‌ها تنها ۳۵ کلمه است) و گاهی اوقات متغیرهایی که نام‌های بیش از حد توصیفی دارند، شروع به شبیه شدن به بخشی از زبان می‌کنند.

نگاهی سریع به کد زیر در پایتون بیندازید که از یک حلقه برای عبور از داده‌ها استفاده می‌کند. ما به زودی حلقه‌ها را پوشش خواهیم داد، اما برای الآن سعی کنید فقط معنای این کد را حدس بزنید:

```
for word in words:
    print(word)
```

در اینجا چه اتفاقی می‌افتد؟ کدام یک از توکن‌ها (مثل for ، word، in و غیره) کلمات رزرو شده هستند و کدام‌ها فقط نام‌های متغیر هستند؟ آیا پایتون در سطح بنیادین مفهوم کلمات را درک می‌کند؟ برنامه‌نویسان مبتدی در تفکیک قسمت‌هایی از کد که باید دقیقاً مشابه این مثال باشند و قسمت‌هایی از کد که صرفاً انتخاب‌های برنامه‌نویس هستند، دچار مشکل می‌شوند.

کد زیر معادل کد قبلی است:

```
for slice in pizza:
    print(slice)
```

برای برنامه‌نویس مبتدی، آسان‌تر است که به این کد نگاه کند و بداند کدام قسمت‌ها کلمات رزرو شده‌ای هستند که توسط پایتون تعریف شده‌اند و کدام قسمت‌ها صرفاً نام‌های متغیری هستند که توسط برنامه‌نویس انتخاب شده‌اند. واضح است که پایتون هیچ درک بنیادینی از مفاهیمی چون پیتزا و برش‌ها ندارد و یا این که یک پیتزا از مجموعه‌ای از یک یا چند برش تشکیل می‌شود.

اما اگر برنامه ما واقعاً درباره خواندن داده‌ها و جستجوی کلمات در داده‌ها باشد، انتخاب نام‌های متغیر غیرمناسب مانند pizza و slice ، نام‌های متغیر غیرمفهومی هستند. انتخاب این نام‌ها به‌عنوان نام‌های متغیر، ما را از معنی واقعی برنامه منحرف می‌کند.

پس از گذشت مدت زمان کوتاهی، شما بیشتر کلمات رزرو شده رایج را یاد خواهید گرفت و شروع خواهید کرد به تشخیص سریع‌تر این کلمات رزرو شده : قسمت‌هایی از کد که توسط پایتون تعریف شده‌اند (مثل for ، in، print، و :) به صورت برجسته نشان داده می‌شوند و متغیرهای انتخابی برنامه‌نویس (مثل word و words ) برجسته نیستند. بسیاری از ویرایشگرهای متنی از نحو پایتون آگاه هستند و کلمات رزرو شده را به صورت متفاوت و رنگی نشان می‌دهند تا به شما کمک کنند که متغیرها و کلمات رزرو شده را از هم تفکیک کنید. پس از مدتی، شما شروع به خواندن کد پایتون خواهید کرد و به راحتی می‌توانید تشخیص دهید که کدام قسمت‌ها متغیر هستند و کدام کلمات رزرو شده‌اند.

## ۲,۱۳ اشکال زدایی

تا کنون، با خطای دستوری SyntaxError که احتمالاً بیشتر مرتکب خواهید شد، نام متغیرهای غیرمجاز مانند class و yield که کلمات کلیدی هستند، یا odd~job و US$ که شامل کاراکترهای غیرمجاز هستند آشنا شده اید.

اگر فاصله‌ای در نام یک متغیر بگذارید، پایتون فکر می‌کند که این دو عملوند هستند بدون عملگر و در نهایت باعث صادر شدن خطا می شود :

```
>>> bad name = 5
SyntaxError: invalid syntax
```

برای خطاهای دستوری، پیام‌های خطا کمکی زیادی نمی‌کنند. رایج‌ترین پیام‌ها SyntaxError: invalid syntax است که خیلی اطلاعاتی نمی‌دهد.

خطای زمان اجرا ( Run Time )که احتمالاً بیشتر با آن روبه‌رو خواهید شد، خطای "استفاده قبل از تعریف" است؛ یعنی سعی می‌کنید از یک متغیر قبل از این که مقداری به آن اختصاص دهید استفاده کنید. این حتی می‌تواند زمانی اتفاق بیفتد که نام متغیر را اشتباه بنویسید:

```
>>> principal = 327.68
>>> interest = principle * rate
NameError: name 'principle' is not defined
```

نام متغیرها حساس به حروف بزرگ و کوچک هستند، بنابراین LaTeX با latex متفاوت است و این خطایی است که ممکن است تازه کاران را درگیر کند.

یکی دیگر از خطاها "خطای معنایی" است، رایج‌ترین علت خطای معنایی احتمالاً ترتیب عملیات است. برای مثال، برای محاسبه ۱/۲π، ممکن است وسوسه شوید که بنویسید:

```
>>> 1.0 / 2.0 * pi
```

اما تقسیم ابتدا انجام می‌شود، بنابراین شما $\pi/2$ را خواهید گرفت، که همان نتیجه دلخواه شما نیست! پایتون هیچ راهی ندارد که بداند شما چه می‌خواستید بنویسید، بنابراین در این حالت شما خطای خاصی دریافت نمی‌کنید؛ فقط پاسخ اشتباهی دریافت می‌کنید.

## ۲,۱۴ واژه نامه

**اختصاص:** عبارتی که مقداری را به یک متغیر نسبت می‌دهد.

**پیوستن(Concatenate)** : اتصال دو عامل به یکدیگر از انتها.

**نظر(Comment)** : اطلاعاتی در یک برنامه که برای سایر برنامه‌نویسان یا هر کسی که کد منبع را می‌خواند نوشته شده است و هیچ تاثیری بر اجرای برنامه ندارد.

**ارزیابی(Evaluate)** : سادگی یک عبارت با انجام عملیات‌ها به منظور به دست آوردن یک مقدار واحد.

**عبارت(Expression)** : ترکیبی از متغیرها، عملگرها و مقادیر که نمایانگر یک مقدار نتیجه واحد است.

**نقطه شناور(Floating point)** : نوعی که نمایانگر اعداد با بخش کسری است.

**عدد صحیح(Integer)** : نوعی که نمایانگر اعداد صحیح است.

**کلمه کلیدی(Keyword)** : کلمه‌ای رزرو شده که توسط کامپایلر برای تجزیه برنامه استفاده می‌شود؛ شما نمی‌توانید از کلمات کلیدی مانندif ، def و while به عنوان نام متغیر استفاده کنید.

**یادآور(Mnemonic)** : یک ابزار حافظه. ما معمولاً به متغیرها نام‌های یادآوری می‌دهیم تا به ما کمک کند که به خاطر بیاوریم چه چیزی در آن متغیر ذخیره شده است.

**عملگر مدولوس(Modulus operator)** : یک عملگر که با علامت درصد (٪) نشان داده می‌شود و روی اعداد صحیح عمل می‌کند و باقیمانده زمانی که یک عدد بر عدد دیگر تقسیم می‌شود را می‌دهد.

**عامل(Operand)** : یکی از مقادیری که عملگر روی آن عمل می‌کند.

**عملگر(Operator)** : یک نماد ویژه که نمایانگر یک محاسبه ساده مانند جمع، ضرب یا پیوستن رشته‌ها است.

**قوانین اولویت(Rules of precedence)** : مجموعه قوانینی که ترتیب ارزیابی عبارات شامل چندین عملگر و عامل را تنظیم می‌کند.

**بیانیه(Statement)** : بخشی از کد که نمایانگر یک دستور یا عمل است. بیانیه‌هایی که تاکنون دیده‌ایم عبارتند از اختصاص و

بیانیه چاپ عبارت.

**رشته(String)** : نوعی که نمایانگر دنباله‌ای از کاراکترها است.

**نوع(Type)** : دسته‌بندی مقادیر. انواعی که تاکنون دیده‌ایم عبارتند از اعداد صحیح (نوعint )، اعداد شناور (نوعfloat )، و رشته‌ها (نوعstr ).

**مقدار(Value)** : یکی از واحدهای اصلی داده، مانند یک عدد یا رشته، که یک برنامه آن را دستکاری می‌کند.

**متغیر(Variable)** : نامی که به یک مقدار اشاره دارد.

## ۲,۱۵ تمرین ها

**تمرین ۲:** برنامه‌ای بنویسید که از ورودی برای درخواست نام کاربر استفاده کند و سپس او را خوش‌آمد بگوید.

نام خود را وارد کنیدChuck : سلامChuck

**تمرین ۳:** برنامه‌ای بنویسید که از کاربر ساعات و نرخ ساعت را بپرسد تا حقوق ناخالص را محاسبه کند.

ساعات را وارد کنید: ۳۵

نرخ را وارد کنید: ۲,۷۵

حقوق: ۹۶,۲۵

ما فعلاً نگران این نیستیم که حقوق دقیقاً دو رقم اعشاری داشته باشد. اگر دوست دارید، می‌توانید با استفاده از تابع داخلی roundدر پایتون نتیجه حقوق را به درستی گرد کنید تا به دو رقم اعشاری برسد.

**تمرین ۴:** فرض کنید که دستورات اختصاصی زیر را اجرا می‌کنیم:

عرض = ۱۷ ارتفاع = ۱۲,۰

برای هرکدام از عبارات زیر، مقدار عبارت و نوع آن را بنویسید (نوع مقدار عبارت)

1. width//2
2. width/2.0
3. height/3
4. 1 + 2 * 5

از مفسر پایتون برای بررسی جواب‌های خود استفاده کنید.

**تمرین ۵:** برنامه‌ای بنویسید که از کاربر دمای سلسیوس را درخواست کند، دما را به فارنهایت تبدیل کند و دمای تبدیل‌شده را چاپ کند.

# فصل سوم

# اجرای شرطی

## ۳,۱ عبارات بولی

عبارت بولی (Boolean Expression) عبارتی است که مقدار آن یا درست (True) یا نادرست (False) است. نمونه‌های زیر از عملگر == استفاده می‌کنند. این عملگر دو عملوند را با یکدیگر مقایسه کرده و در صورتی که برابر باشند مقدار True و در غیر این صورت مقدار False تولید می‌کند:

```
>>> 5 == 5
True
>>> 5 == 6
False
```

Trueو False مقادیر ویژه‌ای هستند که به کلاس bool تعلق دارند؛ این مقادیر رشته (string) نیستند

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

عملگر == یکی از عملگرهای مقایسه‌ای است؛ سایر عملگرهای مقایسه‌ای عبارتند از:

```
x != y               # x is not equal to y
x > y                # x is greater than y
x < y                # x is less than y
x >= y               # x is greater than or equal to y
x <= y               # x is less than or equal to y
x is y               # x is the same as y
x is not y           # x is not the same as y
```

اگرچه احتمالاً این عملیات برای شما آشنا هستند، اما نمادهای پایتون با نمادهای ریاضی مربوط به همین عملیات متفاوت هستند. یک اشتباه رایج این است که به جای استفاده از دو علامت مساوی (==)، از یک علامت مساوی (=) استفاده شود. به خاطر داشته باشید که = یک عملگر انتساب است و == یک عملگر مقایسه می باشد. چیزی به نام =< یا => در پایتون وجود ندارد .

## ۳,۲ عملگرهای منطقی

سه عملگر منطقی وجود دارد or، and و not معنای این عملگرها مشابه معنای آن‌ها در زبان انگلیسی است. برای مثال عبارت زیر

```
x > 0 and x < 10
```

فقط زمانی صحیح (True) است که مقدار x بزرگ‌تر از ۰ و کوچک‌تر از ۱۰ باشد. یا عبارت زیر

```
n%2 == 0 or n%3 == 0
```

در صورتی صحیح است که یکی از دو شرط برقرار باشد؛ یعنی اگر عدد بر ۲ یا ۳ بخش‌پذیر باشد.

در نهایت عملگر not عبارت بولی را معکوس می کند. بنابراین not(x>y) صحیح است اگر x>y نادرست باشد . مثال:

```
>>> x = 1
>>> y = 2
>>> x > y
False
>>> not (x > y)
True
```

از نظر فنی، عملوندهای عملگرهای منطقی باید عبارات بولی باشند، اما پایتون چندان سخت‌گیر نیست. هر عددی که صفر نباشد به عنوان «صحیح» (True) تفسیر می‌شود.

```
>>> 17 and True
True
```

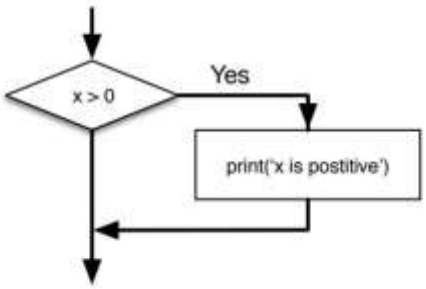
این انعطاف‌پذیری می‌تواند در برخی موقعیت‌ها مفید باشد، اما نکات ظریفی دارد که ممکن است گیج‌کننده باشد. بهتر است تا زمانی که کاملاً مطمئن نشده‌اید چه کاری انجام می‌دهید، از آن استفاده نکنید.

## ۳,۳ اجرای شرطی

برای نوشتن برنامه‌های کاربردی، تقریباً همیشه به بررسی شرایط و تغییر رفتار برنامه بر اساس آن نیاز داریم. دستورات شرطی این امکان را برای ما فراهم می‌کنند. ساده‌ترین شکل آن، دستور if است:

```
if x > 0 :
    print('x is positive')
```

عبارت بولی که بعد از دستور if می‌آید، «شرط» نامیده می‌شود. ما دستور if را با یک کاراکتر دونقطه (:) به پایان می‌بریم و خط یا خطوطی که پس از دستور if قرار می‌گیرند، با تورفتگی (indentation) نوشته می‌شوند.



اگر شرط منطقی درست باشد، دستور تورفته اجرا می‌شود. اگر شرط منطقی نادرست باشد، دستور تورفته نادیده گرفته می‌شود. دستورات if ساختاری مشابه با تعریف توابع یا حلقه‌های for دارند. دستور شامل یک خط عنوان است که با کاراکتر دونقطه (:) به پایان می‌رسد و سپس یک بلوک تورفته قرار می‌گیرد. دستورات این‌چنینی را دستورات ترکیبی می‌نامند زیرا بیش از یک خط را شامل می‌شوند.

```
if x > y:
    print(x)
    print(y)
```

هیچ محدودیتی برای تعداد دستورات موجود در بدنه وجود ندارد، اما باید حداقل یک دستور وجود داشته باشد. گاهی اوقات، مفید است که بدنه‌ای بدون دستور داشته باشید (معمولاً به عنوان جایگزینی برای کدی که هنوز ننوشته‌اید). در این حالت، می‌توانید از دستور pass استفاده کنید تا از بررسی مفسر پایتون عبور کنید، که هیچ‌کاری انجام نمی‌دهد .

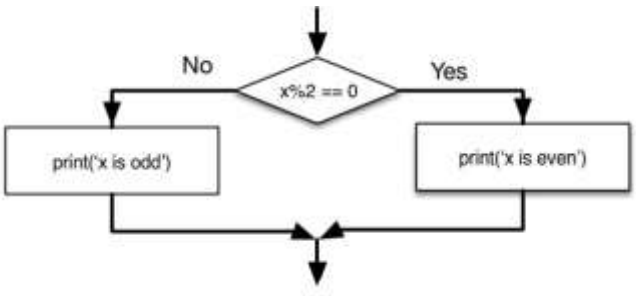
```
if x < 0 :
    pass   # need to handle negative values, do nothing for now.
```

گر یک دستور if را در مفسر پایتون وارد کنید، پرامپت از سه علامت بیشتر (>>>) به سه نقطه (...) تغییر خواهد کرد تا نشان دهد که شما در وسط یک بلوک دستورات هستید، همان‌طور که در زیر نشان داده شده است:

```
>>> x = 3
>>> if x < 10:
...     print('Small')
...
Small
>>>
```

هنگام استفاده از مفسر پایتون، بهتر است که در انتهای بلوک دستورات یک خط خالی بگذارید، در غیر این صورت پایتون خطا خواهد داد:

```
>>> x = 3
>>> if x < 10:
...     print('Small')
... print('Done')
  File "<stdin>", line 3
    print('Done')
        ^
SyntaxError: invalid syntax
```



وجود یک خط خالی در انتهای یک بلوک از دستورات هنگام نوشتن و اجرای اسکریپت ضروری نیست، اما ممکن است خوانایی کد شما را بهبود بخشد.

## ۳,۴ اجرای جایگزین

فرم دوم دستورif ، اجرای جایگزین است که در آن دو امکان وجود دارد و شرط تعیین می‌کند که کدام یک اجرا شود. نحو نوشتن آن به این شکل است:

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

اگر باقی‌مانده تقسیم x بر ۲ صفر باشد، آنگاه می‌دانیم که x عددی زوج است و برنامه پیامی در این زمینه نمایش می‌دهد. اگر شرط نادرست باشد، مجموعه دوم دستورات اجرا می‌شود.

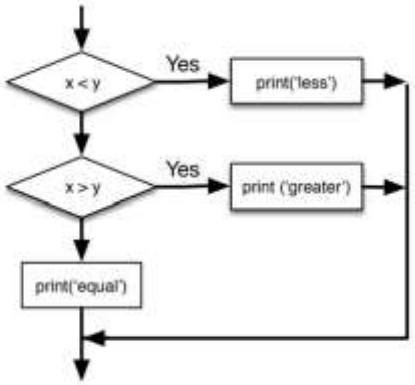
منطقIf-Then-Else از آنجا که شرط باید یا درست باشد یا نادرست، دقیقاً یکی از دو جایگزین اجرا خواهد شد. این جایگزین‌ها به نام شاخه‌ها (branches) شناخته می‌شوند، زیرا آن‌ها شاخه‌هایی در جریان اجرای برنامه هستند.

## ۳,۵ شرایط زنجیره ای

گاهی اوقات بیش از دو گزینه وجود دارد و به بیش از دو شاخه نیاز داریم. یکی از روش‌ها برای بیان چنین محاسباتی، استفاده از شرایط زنجیره‌ای است:

```
if x < y:
    print('x is less than y')
elif x > y:
    print('x is greater than y')
else:
    print('x and y are equal')
```

`elif` مخفف عبارت "else if" است.که در آن باز هم، تنها یکی از شاخه‌ها اجرا خواهد شد.




منطقIf-Then-ElseIf هیچ محدودیتی برای تعداد دستورات `elif` وجود ندارد. اگر یک `else` وجود داشته باشد، باید در انتها قرار گیرد، اما لزومی ندارد که حتماً وجود داشته باشد.

```
if choice == 'a':
    print('Bad guess')
elif choice == 'b':
    print('Good guess')
elif choice == 'c':
    print('Close, but not correct')
```

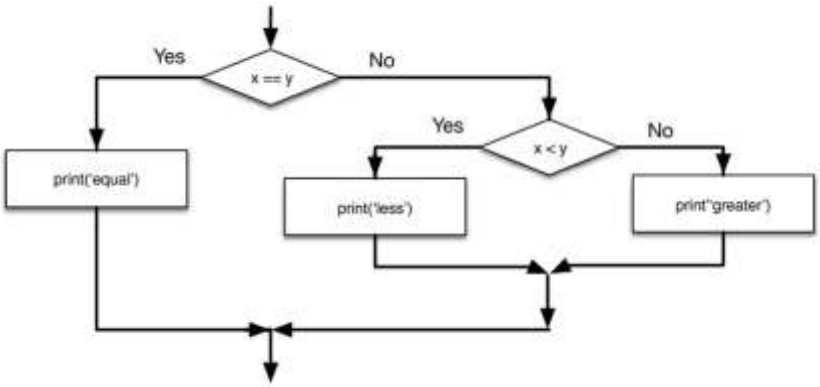
هر شرط به ترتیب بررسی می‌شود. اگر اولین شرط نادرست باشد، شرط بعدی بررسی می‌شود و به همین ترتیب ادامه می‌یابد. اگر یکی از شرایط درست باشد، شاخه مربوطه اجرا می‌شود و اجرای دستور پایان می‌یابد. حتی اگر بیش از یک شرط درست باشد، تنها اولین شاخه درست اجرا می‌شود.

### ۳,۶ شرط های تو در تو

شرط‌ها می‌توانند داخل هم یا به عبارتی تو در تو (nested) قرار بگیرند. به این صورت که می‌توانیم مثال سه شاخه‌ای را به شکل زیر بنویسیم:

```
if x == y:
    print('x and y are equal')
else:
    if x < y:
        print('x is less than y')
    else:
        print('x is greater than y')
```

شرط بیرونی دو شاخه دارد. شاخه اول شامل یک دستور ساده است. شاخه دوم شامل یک دستور if دیگر است که خود دو شاخه دارد. این دو شاخه هر کدام شامل دستورات ساده هستند، هرچند که می‌توانستند شامل دستورات شرطی هم باشند.




اگرچه با تورفتگی دستورات ساختارشان آشکار است، اما خیلی سریع خواندن آن‌ها دشوار می‌شود. به طور کلی، بهتر است زمانی که می‌توانید از این نوع دستورات اجتناب کنید .عملگرهای منطقی اغلب راهی برای ساده‌سازی عبارات شرطی تو در تو فراهم می‌کنند. به عنوان مثال، می‌توانیم کد زیر را با استفاده از یک عبارت شرطی بازنویسی کنیم:

```
if 0 < x:
    if x < 10:
        print('x is a positive single-digit number.')
```

دستور `print` تنها زمانی اجرا می‌شود که هر دو شرط درست باشند. می‌توانیم همان اثر را با استفاده از عملگر `and` هم بدست آوریم:

```
if 0 < x and x < 10:
    print('x is a positive single-digit number.')
```

## ۳,۷ گرفتن استثناها با استفاده از try و except

قبلاً یک بخش کد را دیدیم که در آن از توابع `input` و `int` برای خواندن و تجزیه یک عدد صحیح وارد شده توسط کاربر استفاده کردیم. همچنین دیدیم که این کار چقدر می‌تواند خطرناک باشد:

```
>>> prompt = "What is the air velocity of an unladen swallow?\n"
>>> speed = input(prompt)
What is the air velocity of an unladen swallow?
What do you mean, an African or a European swallow?
>>> int(speed)
ValueError: invalid literal for int() with
base 10: 'What do you mean, an African or a European swallow?'
>>>
```

زمانی که این دستورات را در مفسر پایتون اجرا می‌کنیم، مفسر از ما خواسته جدیدی نمایش می‌دهد، به این فکر می‌کنیم که "اوه"، و به دستور بعدی خود ادامه می‌دهیم .اما اگر این کد را در یک اسکریپت پایتون قرار دهید و این خطا رخ دهد، اسکریپت شما بلافاصله متوقف می‌شود و پیامی با نام traceback نمایش می‌دهد. سپس دستورات بعدی اجرا نمی‌شوند.

در اینجا یک برنامه نمونه برای تبدیل دمای فارنهایت به دمای سلسیوس آورده شده است:

```
inp = input('Enter Fahrenheit Temperature: ')
fahr = float(inp)
cel = (fahr - 32.0) * 5.0 / 9.0
print(cel)

# Code: https://www.py4e.com/code3/fahren.py
```

اگر این کد را اجرا کنیم و ورودی نامعتبری بدهیم، با یک پیام خطای ناخوشایند مواجه خواهیم شد که برنامه را به سادگی متوقف می‌کند:

```
python fahren.py
Enter Fahrenheit Temperature:72
22.22222222222222
python fahren.py
Enter Fahrenheit Temperature:fred
Traceback (most recent call last):
  File "fahren.py", line 2, in <module>
    fahr = float(inp)
ValueError: could not convert string to float: 'fred'
```

یک ساختار اجرایی شرطی در پایتون برای مدیریت این نوع خطاهای پیش‌بینی‌شده و یا پیش‌بینی نشده وجود دارد که به آن "try / except" گفته می‌شود. هدف از استفاده از try و except این است که بدانید ممکن است برخی از دستورات به مشکلی برخورد کنند و بخواهید دستورات اضافی را برای زمانی که خطا رخ می‌دهد، اضافه کنید. این دستورات اضافی (بلاکexcept ) در صورتی که خطایی وجود نداشته باشد نادیده گرفته می‌شوند.

می‌توانید ویژگی try و except در پایتون را به عنوان یک "بیمه‌نامه" برای یک سری دستورات در نظر بگیرید. در صورتی که مشکلی در حین اجرای دستورات پیش بیاید، این بخش از کد به شما کمک می‌کند تا با آن مشکل برخورد کنید.

ما می‌توانیم مبدل دمای خود را به این صورت بازنویسی کنیم:

```
inp = input('Enter Fahrenheit Temperature:')
try:
    fahr = float(inp)
    cel = (fahr - 32.0) * 5.0 / 9.0
    print(cel)
except:
    print('Please enter a number')

# Code: https://www.py4e.com/code3/fahren2.py
```

پایتون ابتدا شروع به اجرای دستورات در بلوک try می‌کند. اگر همه چیز به خوبی پیش برود، بلوک except را نادیده می‌گیرد و ادامه می‌دهد. اگر در بلوک try استثنایی رخ دهد، پایتون از بلوک try خارج شده و دستورات موجود در بلوک except را اجرا می‌کند.

```
python fahren2.py
Enter Fahrenheit Temperature:72
22.22222222222222
python fahren2.py
Enter Fahrenheit Temperature:fred
Please enter a number
```

مدیریت استثنا با استفاده از دستور try به عنوان "گرفتن استثنا" شناخته می‌شود. در این مثال، بخش except یک پیام خطا چاپ می‌کند. به طور کلی، گرفتن استثنا به شما این فرصت را می‌دهد که مشکل را اصلاح کنید، دوباره تلاش کنید، یا حداقل برنامه را به صورت مرتب و بدون مشکل خاتمه دهید.

## ۳,۸ ارزیابی کوتاه‌مدت (Short-circuit evaluation) در عبارات منطقی

زمانی که پایتون یک عبارت منطقی مانند `x >= 2 and (x/y) > 2` را پردازش می‌کند، عبارت را از چپ به راست ارزیابی می‌کند. به دلیل تعریف عملگر `and`، اگر `x` کمتر از ۲ باشد، عبارت `x >= 2` نادرست (False) است و بنابراین کل عبارت نادرست خواهد بود، فارغ از اینکه `(x/y) > 2` درست یا نادرست باشد.

وقتی پایتون تشخیص می‌دهد که ادامه ارزیابی یک عبارت منطقی فایده‌ای ندارد، ارزیابی را متوقف می‌کند و محاسبات باقی‌مانده در عبارت منطقی را انجام نمی‌دهد. زمانی که ارزیابی یک عبارت منطقی متوقف می‌شود چون مقدار نهایی از قبل مشخص شده است، این رفتار به نام «کوتاه‌کردن ارزیابی» (short-circuiting) شناخته می‌شود.

اگرچه این ممکن است به نظر نکته کوچکی بیاید، اما این رفتار کوتاه‌مدت منجر به تکنیک هوشمندانه‌ای به نام «الگوی نگهبان» (guardian pattern) می‌شود. به کد زیر در مفسر پایتون توجه کنید:

```
>>> x = 6
>>> y = 2
>>> x >= 2 and (x/y) > 2
True
>>> x = 1
>>> y = 0
>>> x >= 2 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and (x/y) > 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
```

محاسبه سوم به دلیل این که پایتون در حال ارزیابی عبارت `(x/y)` بود و مقدار `y` برابر با صفر بود، با خطای زمان اجرا مواجه شد. اما در محاسبات اول و دوم این مشکل پیش نیامد، زیرا در محاسبه اول مقدار `y` غیر صفر بود و در محاسبه دوم بخش اول

عبارت‌های `x >= 2` به مقدار `False` ارزیابی شد، بنابراین عبارت `(x/y)` هرگز اجرا نشد به دلیل قانون کوتاه‌کردن ارزیابی و هیچ خطایی پیش نیامد.

ما می‌توانیم عبارت منطقی را طوری تنظیم کنیم که به طور استراتژیک یک ارزیابی محافظتی را درست قبل از ارزیابی‌ای که ممکن است باعث خطا شود، قرار دهیم به شکل زیر:

```
>>> x = 1
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x >= 2 and (x/y) > 2 and y != 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
```

در اولین عبارت منطقی، `x >= 2` برابر با False است، بنابراین ارزیابی در قسمت `and` متوقف می‌شود. در عبارت منطقی دوم، `x >= 2` برابر با True است، اما `y != 0` برابر با False است، بنابراین هرگز به `(x/y)` نمی‌رسیم.

در عبارت منطقی سوم، `y != 0` بعد از محاسبه `(x/y)` قرار دارد، بنابراین عبارت با خطا مواجه می‌شود.

در عبارت دوم، می‌گوییم که `y != 0` به عنوان یک محافظ عمل می‌کند تا اطمینان حاصل کنیم که تنها زمانی که `y` غیر صفر باشد، محاسبه `(x/y)` انجام می‌شود.

## ۳,۹ اشکال‌زدایی

ردیابی خطا (traceback) که پایتون هنگام وقوع یک خطا نمایش می‌دهد، شامل اطلاعات زیادی است، اما ممکن است گاهی گیج‌کننده باشد. بخش‌های مفید معمولاً شامل موارد زیر هستند: نوع خطایی که رخ داده است و محل وقوع آن خطا.

خطاهای نحوی معمولاً راحت‌تر شناسایی می‌شوند، اما چند نکته وجود دارد که ممکن است مشکل‌ساز شوند. خطاهای فضای خالی (whitespace)می‌توانند پیچیده باشند، زیرا فضاها و تب‌ها (tabs) غیرقابل مشاهده هستند و ما معمولاً به آن‌ها توجهی نمی‌کنیم.

```
>>> x = 5
>>>  y = 6
  File "<stdin>", line 1
    y = 6
    ^
IndentationError: unexpected indent
```

در این مثال، مشکل این است که خط دوم با یک فضای اضافی نسبت به خط اول تورفته است. اما پیام خطا به متغیر y اشاره می‌کند که این می‌تواند گمراه‌کننده باشد. به طور کلی، پیام‌های خطا محل کشف مشکل را نشان می‌دهند، اما ممکن است مشکل واقعی در قسمت‌های قبلی کد باشد، گاهی اوقات حتی در خطی که قبلاً نوشته شده است. به طور کلی، پیام‌های خطا به شما محل کشف مشکل را می‌گویند، اما این اغلب جایی نیست که خطا واقعاً اتفاق افتاده است.

## ۳,۱۰ واژه نامه

**بدنه:(body)** دنباله‌ای از دستورات که درون یک دستور ترکیبی قرار دارند.

**عبارت بولی:(boolean expression)** عبارتی که مقدار آن یا True است یا.False

**شاخه:(branch)** یکی از دنباله‌های جایگزین دستورات در یک دستور شرطی.

**دستور شرطی زنجیره‌ای:(chained conditional)** دستور شرطی با مجموعه‌ای از شاخه‌های جایگزین.

**عملگر مقایسه:(comparison operator)** یکی از عملگرهایی که عمل مقایسه را روی دو عملوند انجام می‌دهد== :، = !، <، >، =<، و .=>

**دستور شرطی:(conditional statement)** دستوری که جریان اجرای برنامه را بسته به یک شرط کنترل می‌کند.

**شرط:(condition)** عبارت بولی در یک دستور شرطی که تعیین می‌کند کدام شاخه اجرا شود.

**دستور ترکیبی:(compound statement)** دستوری که از یک سربرگ و یک بدنه تشکیل شده است. سربرگ با علامت دو نقطه (:) پایان می‌یابد و بدنه به‌طور نسبی نسبت به سربرگ تورفته است.

**الگوی محافظ:(guardian pattern)** وقتی که یک عبارت منطقی با مقایسه‌های اضافی ساخته می‌شود تا از رفتار کوتاه‌مدت (short-circuit)استفاده کند.

**عملگر منطقی:(logical operator)** یکی از عملگرهایی که عبارات بولی را ترکیب می‌کندand :، or، و .not

**دستور شرطی تو در تو:(nested conditional)** دستوری شرطی که در یکی از شاخه‌های یک دستور شرطی دیگر قرار دارد.

**ردیابی خطا:(traceback)** فهرستی از توابعی که در حال اجرا هستند و زمانی که یک استثنا رخ می‌دهد، چاپ می‌شود.

**قطع مسیر :(short circuit)** وقتی که پایتون در حال ارزیابی یک عبارت منطقی است و ارزیابی را متوقف می‌کند زیرا پایتون می‌داند که مقدار نهایی عبارت بدون ارزیابی ادامه عبارت قابل تعیین است.

## ۳,۱۱ تمرین ها

**تمرین ۱:** محاسبه‌ی دستمزد خود را بازنویسی کنید تا برای ساعات کاری بیش از ۴۰ ساعت، به کارمند ۱,۵ برابرِ نرخ ساعتی پرداخت شود.

```
Enter Hours: 45
Enter Rate: 10
Pay: 475.0
```

**تمرین ۲** :برنامه محاسبه حقوق خود را بازنویسی کنید و در آن از $try$ و $except$ استفاده کنید تا برنامه در صورت دریافت ورودی غیرعددی، ورودی را به‌درستی مدیریت کرده و با چاپ یک پیام، از برنامه خارج شود. دو اجرای زیر نمونه‌ای از رفتار مورد انتظار برنامه را نشان می‌دهند:

```
Enter Hours: 20
Enter Rate: nine
Error, please enter numeric input
Enter Hours: forty
Error, please enter numeric input
```

**تمرین ۳**: یک برنامه بنویسید که برای نمره‌ای بین ۰,۰ و ۱,۰ درخواست کند. اگر نمره خارج از این بازه بود، یک پیام خطا چاپ کند. اگر نمره بین ۰,۰ و ۱,۰ بود، براساس جدول زیر یک درجه (grade) چاپ کند:

```
 Score   Grade
>= 0.9     A
>= 0.8     B
>= 0.7     C
>= 0.6     D
 < 0.6     F
Enter score: 0.95
A
Enter score: perfect
Bad score
Enter score: 10.0
Bad score
Enter score: 0.75
C
Enter score: 0.5
F
```

برنامه را چندین بار اجرا کنید، همان‌طور که در بالا نشان داده شده است، تا مقادیر مختلف ورودی را آزمایش کنید.

# فصل چهارم

# توابع

## ۴,۱ فراخوانی توابع

در زمینه برنامه‌نویسی، تابع دنباله‌ای نام‌گذاری شده از دستورات است که یک محاسبه را انجام می‌دهد. زمانی که یک تابع را تعریف می‌کنید، نام و دنباله دستورات آن را مشخص می‌کنید. بعداً می‌توانید این تابع را با نامش "فراخوانی" کنید. پیش‌تر یک نمونه از فراخوانی تابع را مشاهده کرده‌ایم:

```
>>> type(32)
<class 'int'>
```

نام تابع، type است. عبارتی که درون پرانتز قرار دارد، «آرگومان» تابع نامیده می‌شود. آرگومان، یک مقدار یا متغیری است که به عنوان ورودی به تابع داده می‌شود. نتیجه، در مورد تابع type ، نوع آرگومان است.

رایج است که بگوییم یک تابع «آرگومانی دریافت می‌کند» و «نتیجه‌ای بازمی‌گرداند». این نتیجه «مقدار بازگشتی» نامیده می‌شود.

## ۴,۲ توابع داخلی (Built-in functions)

پایتون تعدادی تابع داخلی مهم فراهم کرده است که می‌توانیم بدون نیاز به تعریف، از آنها استفاده کنیم. سازندگان پایتون مجموعه‌ای از توابع را برای حل مشکلات رایج نوشته‌اند و آن‌ها را برای استفاده ما درون پایتون قرار داده‌اند.

توابع max و min به ترتیب بزرگ‌ترین و کوچک‌ترین مقدار موجود در یک لیست را به ما می‌دهند:

```
>>> max('Hello world')
'w'
>>> min('Hello world')
' '
>>>
```

تابع max بزرگ‌ترین "کاراکتر" در یک رشته را به ما نشان می‌دهد (که در این مثال حرف "w" است) و تابع min کوچک‌ترین کاراکتر (که یک فاصله‌ی خالی یا "space" است) را مشخص می‌کند. یکی دیگر از توابع داخلی بسیار رایج، تابع len است که تعداد آیتم‌های موجود در آرگومان خود را به ما می‌گوید. اگر آرگومان len یک رشته باشد، تعداد کاراکترهای آن رشته را بازمی‌گرداند.

```
>>> len('Hello world')
11
>>>
```

این توابع فقط محدود به بررسی رشته‌ها (strings) نیستند. آن‌ها می‌توانند روی هر مجموعه‌ای از مقادیر عمل کنند، این موضوع را در فصل‌های بعدی خواهیم دید. باید نام توابع داخلی را مانند کلمات رزرو شده در نظر بگیرید (یعنی از استفاده از نام‌هایی مثل `max` برای متغیرهای خودداری کنید)

## ۴,۳ توابع تبدیل نوع

پایتون همچنین توابع داخلی‌ای ارائه می‌دهد که مقادیر را از یک نوع به نوع دیگر تبدیل می‌کنند. تابع `int` هر مقداری را می‌گیرد و اگر بتواند، آن را به عدد صحیح (integer) تبدیل می‌کند، در غیر این صورت خطا می‌دهد:

```
>>> int('32')
32
>>> int('Hello')
ValueError: invalid literal for int() with base 10: 'Hello'
```

تابع `int` می‌تواند مقادیر اعشاری (floating-point) را به عدد صحیح (integer) تبدیل کند، اما عدد را گرد (round) نمی‌کند؛ بلکه قسمت اعشاری را حذف می‌کند:

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

تابع `float` مقادیر صحیح (integer) و رشته‌ها (string) را به اعداد اعشاری (floating-point) تبدیل می‌کند:

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

در نهایت، تابع `str` آرگومان خود را به رشته (string) تبدیل می‌کند :

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

## ۴,۴ توابع ریاضی

پایتون یک ماژول ریاضی به نام `math` دارد که بیشتر توابع ریاضی آشنا را فراهم می‌کند. قبل از اینکه بتوانیم از این ماژول استفاده کنیم، باید آن را وارد (import) کنیم:

```
>>> import math
```

این دستور یک ماژول به نام `math` ایجاد می‌کند. اگر این ماژول را چاپ کنید، اطلاعاتی در مورد آن به شما نمایش داده می‌شود:

```
>>> print(math)
<module 'math' (built-in)>
```

ماژول شامل توابع و متغیرهایی است که در داخل آن تعریف شده‌اند. برای دسترسی به یکی از توابع، باید نام ماژول و نام تابع را با یک نقطه (که به آن نقطه‌گذاری نیز گفته می‌شود) از هم جدا کنید. این فرمت به عنوان "نقطه‌گذاری" شناخته می‌شود.

```
>>> ratio = signal_power / noise_power
>>> decibels = 10 * math.log10(ratio)

>>> radians = 0.7
>>> height = math.sin(radians)
```

مثال اول لگاریتم با پایه ۱۰ نسبت سیگنال به نویز را محاسبه می‌کند. ماژول math همچنین تابعی به نام log دارد که لگاریتم با پایه e را محاسبه می‌کند.

مثال دوم سینوس رادیان‌ها را پیدا می‌کند. نام متغیر نشان‌دهنده این است که تابع sin و دیگر توابع مثلثاتی (cos، tan و غیره) آرگومان‌هایی به صورت رادیان می‌گیرند. برای تبدیل از درجه به رادیان، باید عدد را بر ۳۶۰ تقسیم کرده و سپس در ۲ π ضرب کنید:

```
>>> degrees = 45
>>> radians = degrees / 360.0 * 2 * math.pi
>>> math.sin(radians)
0.7071067811865476
```

عبارت `math.pi` متغیر π را از ماژول math می‌گیرد. مقدار این متغیر تقریباً برابر با π است و دقت آن تا حدود ۱۵ رقم اعشار می‌باشد.

اگر با مثلثات آشنا باشید، می‌توانید نتیجه قبلی را با مقایسه آن با ریشه دوم دو تقسیم بر دو بررسی کنید:

```
>>> math.sqrt(2) / 2.0
0.7071067811865476
```

## ۴,۵ اعداد تصادفی

برنامه‌های کامپیوتری معمولاً همان خروجی‌ها را با همان ورودی‌ها تولید می‌کنند، بنابراین به آنها گفته می‌شود که الگوریتم‌های تعیین‌کننده (deterministic) هستند. تعیین‌کنندگی معمولاً یک ویژگی مثبت است، زیرا ما انتظار داریم که هر محاسبه همان نتیجه را هر بار بدهد. اما در برخی از کاربردها، ما می‌خواهیم که کامپیوتر غیرقابل پیش‌بینی باشد. بازی‌ها یک مثال واضح هستند، اما کاربردهای دیگری نیز وجود دارند.

ساخت یک برنامه به‌طور واقعی غیرتعیین‌کننده (nondeterministic) واقعاً کار آسانی نیست، اما روش‌هایی وجود دارد که می‌توانیم آن را حداقل به‌طور ظاهری غیرتعیین‌کننده کنیم. یکی از این روش‌ها استفاده از الگوریتم‌هایی است که اعداد شبه تصادفی (pseudorandom numbers)تولید می‌کنند. اعداد شبه تصادفی واقعاً تصادفی نیستند زیرا توسط یک محاسبه تعیین‌کننده تولید می‌شوند، اما با نگاه به اعداد، تقریباً غیرممکن است که آنها را از اعداد تصادفی واقعی تشخیص دهیم.

ماژول random توابعی فراهم می‌کند که اعداد شبه تصادفی تولید می‌کنند (که از این به بعد آنها را صرفاً "تصادفی" می‌نامم.

تابع random یک عدد اعشاری تصادفی بین ۰,۰ و ۱,۰ (شامل ۰,۰ ولی نه ۱,۰) باز می‌گرداند. هر بار که تابع random را فراخوانی کنید، عدد بعدی را در یک سری طولانی دریافت می‌کنید. برای مشاهده نمونه‌ای از این اعداد، این حلقه را اجرا کنید:

```
import random

for i in range(10):
    x = random.random()
    print(x)
```

این برنامه فهرستی از ۱۰ عدد تصادفی بین ۰,۰ و تا قبل از ۱,۰ تولید می‌کند.

```
0.11132867921152356
0.5950949227890241
0.04820265884996877
0.841003109276478
0.997914947094958
0.04842330803368111
0.7416295948208405
0.510535245390327
0.27447040171978143
0.028511805472785867
```

**تمرین ۱**: برنامه را در سیستم خود اجرا کنید و ببینید چه اعدادی دریافت می‌کنید. برنامه را چندین بار اجرا کنید و ببینید چه اعدادی دریافت می‌کنید.

تابع `random` تنها یکی از بسیاری از توابعی است که اعداد تصادفی را مدیریت می‌کند. تابع `randint` دو پارامتر `low` و `high` را می‌گیرد و یک عدد صحیح بین `low` و `high`( شامل هر دو عدد) برمی‌گرداند.

```
>>> random.randint(5, 10)
5
>>> random.randint(5, 10)
9
```

برای انتخاب یک عنصر به طور تصادفی از یک دنباله، می‌توانید از تابع choice استفاده کنید :

```
>>> t = [1, 2, 3]
>>> random.choice(t)
2
>>> random.choice(t)
3
```

ماژول random توابعی برای تولید مقادیر تصادفی از توزیع‌های پیوسته مختلف از جمله گاوسی، نمایی، گاما و چند مورد دیگر فراهم می‌کند.

## ۴,۶ افزودن توابع جدید

تا به حال، تنها از توابعی که همراه با پایتون هستند استفاده کرده‌ایم، اما امکان افزودن توابع جدید نیز وجود دارد. هنگام تعریف یک تابع نام و دنباله ی دستوراتی که هنگام فراخوانی تابع اجرا می شود را مشخص می کنیم. پس از تعریف تابع، می‌توانیم تابع را بارها و بارها در سراسر برنامه خود استفاده کنیم.

Here is an example:

```
def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print('I sleep all night and I work all day.')
```

کلمه کلیدی def نشان می‌دهد که این یک تعریف تابع (function definition) است. نام این تابع print_lyrics است. قوانین نام‌گذاری توابع همانند قوانین نام‌گذاری متغیرهاست: استفاده از حروف، اعداد و برخی علائم نگارشی مجاز است، اما اولین نویسه نمی‌تواند عدد باشد. همچنین نمی‌توان از کلمات کلیدی به عنوان نام تابع استفاده کرد، و بهتر است نام یک تابع و یک متغیر یکسان نباشد تا از بروز سردرگمی جلوگیری شود .

پرانتزهای خالی پس از نام تابع نشان می‌دهند که این تابع هیچ آرگومانی دریافت نمی‌کند. در ادامه، توابعی خواهیم ساخت که ورودی‌هایی به صورت آرگومان دریافت می‌کنند.

خط اول تعریف تابع را «سربرگ» (header) می‌نامند؛ باقی خطوط «بدنه» (body) تابع محسوب می‌شوند. سربرگ باید با یک دونقطه (:) به پایان برسد و بدنه نیز باید به صورت تورفتگی‌دار نوشته شود. طبق قرارداد، میزان تورفتگی معمولاً چهار فاصله (space) است. بدنه می‌تواند شامل هر تعداد دستور باشد.

اگر تابعی را در حالت تعاملی (interactive mode) تایپ کنید، مفسر برای نشان دادن اینکه تعریف تابع هنوز کامل نشده، از علامت سه‌نقطه (...) استفاده می‌کند.

```
>>> def print_lyrics():
...     print("I'm a lumberjack, and I'm okay.")
...     print('I sleep all night and I work all day.')
...
```

برای پایان دادن به تعریف تابع در حالت تعاملی(interactive mode) ، باید یک خط خالی وارد کنید (این کار در اسکریپت‌ها الزامی نیست).

تعریف یک تابع باعث می‌شود یک متغیر با همان نام تابع ساخته شود.

```
>>> print(print_lyrics)
<function print_lyrics at 0xb7e99e9c>
>>> print(type(print_lyrics))
<class 'function'>
```

مقدار `print_lyrics` یک شیء تابع (function object) است که نوع آن "function" می‌باشد. نحو (syntax) فراخوانی این تابع جدید مانند توابع داخلی (built-in functions) در پایتون است:

```
>>> print_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

زمانی که یک تابع را تعریف کردید، می‌توانید از آن درون یک تابع دیگر استفاده کنید. برای مثال، برای تکرار قسمت قبلی، می‌توانیم تابعی به نام `repeat_lyrics` بنویسیم:

```
def repeat_lyrics():
    print_lyrics()
    print_lyrics()
```

و سپس تابع `repeat_lyrics` را فراخوانی کنیم:

```
>>> repeat_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

اما در واقع آهنگ این‌گونه اجرا نمی‌شود.

## ۴,۷ تعریف ها و کاربردها

اگر بخش‌های مختلف کد از قسمت قبل را کنار هم بگذاریم، کل برنامه به صورت زیر خواهد بود:

```
def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print('I sleep all night and I work all day.')


def repeat_lyrics():
    print_lyrics()
    print_lyrics()

repeat_lyrics()

# Code: https://www.py4e.com/code3/lyrics.py
```

این برنامه دو تعریف تابع دارد `print_lyrics` و `repeat_lyrics`. تعریف توابع مانند سایر دستورات اجرا می‌شوند، اما اثر آن‌ها ایجاد شیء تابع است. دستورات درون تابع تا زمانی که تابع فراخوانی نشود اجرا نمی‌شوند، و تعریف تابع هیچ خروجی‌ای تولید نمی‌کند.

همان‌طور که ممکن است انتظار داشته باشید، باید یک تابع را قبل از اجرای آن ایجاد کرده باشید. به عبارت دیگر، تعریف تابع باید قبل از اولین باری که فراخوانی می‌شود، اجرا شده باشد.

**تمرین ۲ :**خط آخر این برنامه را به ابتدای برنامه منتقل کنید، به طوری که فراخوانی تابع قبل از تعریف‌ها قرار گیرد. برنامه را اجرا کنید و ببینید چه پیغام خطایی دریافت می‌کنید.

**تمرین ۳ :**فراخوانی تابع را دوباره به پایین بازگردانید و تعریف تابع `print_lyrics` را بعد از تعریف تابع `repeat_lyrics` قرار دهید. هنگام اجرای برنامه چه اتفاقی می‌افتد؟

## ۴,۸ جریان اجرا

برای اطمینان از اینکه یک تابع قبل از اولین استفاده‌اش تعریف شده است، باید ترتیب اجرای دستورات را بدانید، که به آن **جریان اجرا** گفته می‌شود.

اجرای برنامه همیشه از اولین دستور آن آغاز می‌شود. دستورات یکی یکی و به ترتیب از بالا به پایین اجرا می‌شوند. تعریف توابع جریان اجرای برنامه را تغییر نمی‌دهد، اما به یاد داشته باشید که دستورات داخل تابع تا زمانی که تابع فراخوانی نشود، اجرا نمی‌شوند.

فراخوانی یک تابع مانند یک انحراف موقت در جریان اجراست. به جای رفتن به دستور بعدی، جریان اجرا به بدنه‌ی تابع می‌پرد، تمام دستورات آن را اجرا می‌کند و سپس به همان جایی که رها شده بود بازمی‌گردد.

این موضوع به‌ظاهر ساده است، تا زمانی که به یاد بیاورید یک تابع می‌تواند تابع دیگری را فراخوانی کند. در حالی که در وسط اجرای یک تابع هستیم، ممکن است برنامه نیاز داشته باشد که دستورات تابع دیگری را اجرا کند. اما در حین اجرای آن تابع جدید، ممکن است نیاز به اجرای تابع سومی هم باشد!

خوشبختانه پایتون در پیگیری محل اجرای برنامه بسیار خوب عمل می‌کند؛ بنابراین هر بار که یک تابع به پایان می‌رسد، برنامه دقیقاً از همان‌جایی که در تابع فراخواننده متوقف شده بود، ادامه پیدا می‌کند. وقتی به انتهای برنامه می‌رسد، خاتمه می‌یابد.

نتیجه‌ی این داستان چیست؟ وقتی یک برنامه را می‌خوانید، همیشه لازم نیست آن را از بالا به پایین بخوانید. گاهی بهتر است مسیر جریان اجرا را دنبال کنید.

## ۴,۹ پارامترها و آرگومان ها

برخی از توابع داخلی که تاکنون دیده‌ایم نیاز به آرگومان دارند. برای مثال، وقتی تابع math.sin را فراخوانی می‌کنید، یک عدد را به عنوان آرگومان به آن می‌دهید. برخی از توابع بیش از یک آرگومان می‌گیرند: برای مثال، math.pow دو آرگومان می‌گیرد، پایه و توان.

داخل تابع، آرگومان‌ها به متغیرهایی به نام **پارامترها** اختصاص داده می‌شوند. در اینجا یک مثال از یک تابع کاربرتعریف شده است که یک آرگومان می‌گیرد:

```
def print_twice(bruce):
    print(bruce)
    print(bruce)
```

این تابع آرگومان را به پارامتر با نام **bruce** اختصاص می‌دهد. وقتی تابع فراخوانی می‌شود، مقدار پارامتر (هر چه که باشد) را دو بار چاپ می‌کند. این تابع با هر مقداری که قابل چاپ باشد، کار می‌کند.

```
>>> print_twice('Spam')
Spam
Spam
>>> print_twice(17)
17
17
>>> import math
>>> print_twice(math.pi)
3.141592653589793
3.141592653589793
```

قوانین ترکیب مشابهی که برای توابع داخلی اعمال می‌شوند، برای توابع تعریف شده توسط کاربر نیز اعمال می‌شوند، بنابراین می‌توانیم از هر نوع عبارت به عنوان آرگومان برای تابع **print_twice** استفاده کنیم:

```
>>> print_twice('Spam '*4)
Spam Spam Spam Spam
Spam Spam Spam Spam
>>> print_twice(math.cos(math.pi))
-1.0
-1.0
```

آرگومان قبل از فراخوانی تابع ارزیابی می‌شود، بنابراین در مثال‌ها، عبارات `'Spam '*4` و `math.cos(math.pi)` تنها یکبار ارزیابی می‌شوند.

همچنین می‌توانید از یک متغیر به عنوان آرگومان استفاده کنید:

```
>>> michael = 'Eric, the half a bee.'
>>> print_twice(michael)
Eric, the half a bee.
Eric, the half a bee.
```

نام متغیری که به عنوان آرگومان می‌فرستیم (michael) هیچ ربطی به نام پارامتر (bruce) ندارد. اهمیتی ندارد که نام مقدار در جایی دیگر (در فراخوانی تابع) چه بوده باشد؛ در داخل تابع print_twice ، به همه می‌گوییم."bruce"

## ۴,۱۰ توابع پربازده و توابع بدون بازگشت

برخی از توابعی که ما استفاده می‌کنیم، مانند توابع ریاضی، نتایجی را تولید می‌کنند؛ که به آن‌ها توابع پربازده می‌گوییم. توابع دیگری مثل print_twice عملیاتی انجام می‌دهند اما مقدار برنگردانده و بدون بازگشت هستند. این‌ها توابع بدون بازگشت نامیده می‌شوند.

هنگامی که یک تابع پربازده را فراخوانی می‌کنید، تقریباً همیشه می‌خواهید با نتیجه‌ای که به دست می‌آید کاری انجام دهید؛ به عنوان مثال، می‌توانید آن را به یک متغیر اختصاص دهید یا از آن در بخشی از یک عبارت استفاده کنید:

```
x = math.cos(radians)
golden = (math.sqrt(5) + 1) / 2
```

وقتی یک تابع را در حالت تعاملی فراخوانی می‌کنید، پایتون نتیجه را نمایش می‌دهد:

```
>>> math.sqrt(5)
2.23606797749979
```

اما در یک اسکریپت، اگر یک تابع پربازده را فراخوانی کنید و نتیجه تابع را در یک متغیر ذخیره نکنید، مقدار بازگشتی ناپدید می‌شود!

```
math.sqrt(5)
```

این اسکریپت ریشه مربع ۵ را محاسبه می‌کند، اما از آنجا که نتیجه را در یک متغیر ذخیره نمی‌کند یا نتیجه را نمایش نمی‌دهد، خیلی مفید نیست.

توابع void ممکن است چیزی را روی صفحه نمایش دهند یا اثر دیگری داشته باشند، اما مقدار بازگشتی ندارند. اگر سعی کنید نتیجه را به یک متغیر انتساب دهید، یک مقدار خاص به نام None دریافت خواهید کرد.

```
>>> result = print_twice('Bing')
Bing
Bing
>>> print(result)
None
```

مقدار None مانند رشته "None" نیست. این یک مقدار خاص است که نوع خاص خود را دارد:

```
>>> print(type(None))
<class 'NoneType'>
```

برای بازگشت یک نتیجه از یک تابع، از دستور return در تابع خود استفاده می‌کنیم. به عنوان مثال، می‌توانیم یک تابع بسیار ساده به نام addtwo ایجاد کنیم که دو عدد را با هم جمع کرده و یک نتیجه باز می‌گرداند.

```
def addtwo(a, b):
    added = a + b
    return added

x = addtwo(3, 5)
print(x)

# Code: https://www.py4e.com/code3/addtwo.py
```

وقتی این اسکریپت اجرا می‌شود، دستور print مقدار "۸" را چاپ می‌کند زیرا تابع addtwo با آرگومان‌های ۳ و ۵ فراخوانی شده است. در داخل تابع، پارامترهای a و b به ترتیب ۳ و ۵ بودند. تابع مجموع این دو عدد را محاسبه کرده و در متغیر محلی تابع به نام added ذخیره کرده است. سپس از دستور return برای ارسال مقدار محاسبه شده به کد فراخوانی استفاده کرده است که به متغیر x اختصاص داده شده و چاپ می‌شود.

## ۴,۱۱ چرا از توابع استفاده کنیم؟

ممکن است واضح نباشد که چرا تقسیم یک برنامه به توابع ارزش دارد. دلایل مختلفی برای این کار وجود دارد:

- ایجاد یک تابع جدید به شما این امکان را می‌دهد که گروهی از دستورات را نام‌گذاری کنید که برنامه شما را خواناتر، قابل‌فهم‌تر و اشکال‌زدایی پذیرتر می‌کند.
- توابع می‌توانند برنامه را کوچک‌تر کنند زیرا کد تکراری را حذف می‌کنند. بعداً اگر تغییراتی لازم باشد، فقط باید آن را در یک مکان انجام دهید.
- تقسیم یک برنامه طولانی به توابع این امکان را می‌دهد که هر بخش را به صورت جداگانه اشکال‌زدایی کنید و سپس آن‌ها را در یک کل کارا ترکیب کنید.
- توابع طراحی‌شده به خوبی معمولاً برای بسیاری از برنامه‌ها مفید هستند. هنگامی که یک تابع را بنویسید و اشکال‌زدایی کنید، می‌توانید از آن دوباره استفاده کنید.

در ادامه کتاب، اغلب از تعریف یک تابع برای توضیح یک مفهوم استفاده خواهیم کرد. بخشی از مهارت در ایجاد و استفاده از توابع این است که یک تابع به درستی یک ایده مانند "یافتن کوچکترین مقدار در یک لیست از مقادیر" را ارائه کند. بعدها، کدی را به شما نشان خواهیم داد که کوچکترین مقدار را در یک لیست از مقادیر پیدا می‌کند و آن را به‌عنوان تابعی به نام `min` به شما ارائه خواهیم کرد که لیستی از مقادیر را به‌عنوان آرگومان دریافت کرده و کوچکترین مقدار در آن لیست را باز می‌گرداند.

## ۴,۱۲ اشکال زدایی

اگر از یک ویرایشگر متن برای نوشتن اسکریپت‌های خود استفاده می‌کنید، ممکن است با مشکلاتی در استفاده از فاصله‌ها و تب‌ها مواجه شوید. بهترین راه برای جلوگیری از این مشکلات استفاده از فاصله‌ها به‌طور انحصاری است (بدون تب). بیشتر ویرایشگرهای متنی که از پایتون پشتیبانی می‌کنند به‌طور پیش‌فرض این کار را انجام می‌دهند، اما برخی از آن‌ها این‌گونه نیستند.

تب‌ها و فاصله‌ها معمولاً قابل مشاهده نیستند، که این امر اشکال‌زدایی را دشوار می‌کند. بنابراین سعی کنید ویرایشگری پیدا کنید که مدیریت تورفتگی (indentation) را برای شما انجام دهد.

همچنین فراموش نکنید که قبل از اجرای برنامه آن را ذخیره کنید. برخی از محیط‌های توسعه به‌طور خودکار این کار را انجام می‌دهند، اما برخی دیگر این‌طور نیستند. در این صورت، برنامه‌ای که در ویرایشگر متن می‌بینید همان برنامه‌ای نیست که در حال اجرا است.

اشکال‌زدایی ممکن است زمانی طولانی ببرد اگر مدام برنامه نادرست را دوباره و دوباره اجرا کنید! مطمئن شوید که کدی که مشاهده می‌کنید همان کدی است که اجرا می‌کنید. اگر مطمئن نیستید، چیزی مانند `print("hello")` را در ابتدای برنامه قرار دهید و دوباره آن را اجرا کنید. اگر "hello" را ندیدید، پس برنامه درستی را اجرا نمی‌کنید!

## ۴,۱۳ واژه نامه

**الگوریتم**: یک فرآیند عمومی برای حل یک دسته از مشکلات.

**آرگومان**: مقداری که هنگام فراخوانی یک تابع به آن ارسال می‌شود. این مقدار به پارامتر مربوطه در تابع اختصاص داده می‌شود.

**بدن(Body)** : دنباله‌ای از دستورات داخل یک تعریف تابع.

**ترکیب(Composition)** : استفاده از یک عبارت به‌عنوان بخشی از یک عبارت بزرگتر یا یک دستور به‌عنوان بخشی از یک دستور بزرگتر.

**قطعی(Deterministic)** : مربوط به برنامه‌ای که هر بار با ورودی‌های یکسان همان کار را انجام می‌دهد.

**نشان‌گذاری نقطه‌ای(Dot Notation)** : سینتاکسی برای فراخوانی یک تابع از یک ماژول دیگر که با مشخص کردن نام ماژول و سپس نقطه (period) و نام تابع انجام می‌شود.

**جریان اجرای برنامه(Flow of Execution)** : ترتیب اجرای دستورات در هنگام اجرای یک برنامه.

**تابع میوه‌ای(Fruitful Function)** : تابعی که مقداری را باز می‌گرداند.

**تابع(Function)** : دنباله‌ای از دستورات نام‌گذاری‌شده که عملی مفید را انجام می‌دهد. توابع ممکن است آرگومان بگیرند و یا نتیجه‌ای تولید کنند و ممکن است این‌گونه نباشند.

**فراخوانی تابع(Function Call)** : دستوری که یک تابع را اجرا می‌کند. این دستور شامل نام تابع و فهرست آرگومان‌ها است.

**تعریف تابع(Function Definition)** : دستوری که یک تابع جدید می‌سازد و نام، پارامترها و دستورات اجرایی آن را مشخص می‌کند.

**شی تابع(Function Object)** : مقداری که توسط یک تعریف تابع ایجاد می‌شود. نام تابع یک متغیر است که به شی تابع اشاره می‌کند.

**هدینگ(Header)** : اولین خط یک تعریف تابع.

**دستورimport** : دستوری که یک فایل ماژول را می‌خواند و یک شی ماژول ایجاد می‌کند.

**شی ماژول(Module Object)** : مقداری که توسط دستور import ایجاد می‌شود و دسترسی به داده‌ها و کدهای تعریف‌شده در یک ماژول را فراهم می‌کند.

**پارامتر(Parameter)** : نامی که در داخل یک تابع برای ارجاع به مقداری که به‌عنوان آرگومان ارسال می‌شود، استفاده می‌شود.

**پسو-تصادفی(Pseudorandom)** : مربوط به دنباله‌ای از اعداد که به‌نظر تصادفی می‌آیند، اما توسط یک برنامه قطعی تولید می‌شوند.

**مقدار بازگشتی(Return Value)** : نتیجه یک تابع. اگر فراخوانی تابع به‌عنوان یک عبارت استفاده شود، مقدار بازگشتی همان مقدار عبارت است.

**تابع بدون بازگشت(Void Function)** : تابعی که مقداری باز نمی‌گرداند.

## ۴,۱۴ تمرین ها

**تمرین ۴** :هدفِ واژه **def** در پایتون چیست؟

الف) این یک اصطلاح عامیانه است و به این معنی است که «کدِ بعدی واقعاً عالی است.»
ب) این کلمه آغاز یک تابع را مشخص می‌کند.
پ) این کلمه نشان می‌دهد که بخش تورفتگی‌یافته بعدیِ کد باید برای استفاده بعدی ذخیره شود.
ت)هم b و هم c درست هستند.
ث)هیچ‌یک از موارد بالا.

**تمرین ۵**: برنامه پایتون زیر چه چیزی را چاپ خواهد کرد؟

```
def fred():
   print("Zap")

def jane():
   print("ABC")

jane()
fred()
jane()
```

a) Zap ABC jane fred jane
b) Zap ABC Zap
c) ABC Zap jane
d) ABC Zap ABC
e) Zap Zap Zap

**تمرین ۶** :محاسبه‌ی حقوق خود را با درنظرگرفتن «اضافه‌کار با نرخ یک‌ونیم برابر» بازنویسی کنید و یک تابع به نام **computepay** ایجاد کنید که دو پارامتر(**ساعت‌ها و نرخ**)را دریافت می‌کند.

```
Enter Hours: 45
Enter Rate: 10
Pay: 475.0
```

**تمرین ۷** :برنامه تعیین نمره از فصل قبل را بازنویسی کنید به‌طوری‌که از تابعی به نام **computegrade** استفاده کند؛ این تابع یک **score** را به‌عنوان ورودی بگیرد و نمره مربوطه را به‌صورت **رشته** بازگرداند.

```
 Score   Grade
>= 0.9     A
>= 0.8     B
>= 0.7     C
>= 0.6     D
 < 0.6     F
Enter score: 0.95
A
Enter score: perfect
Bad score
Enter score: 10.0
Bad score
Enter score: 0.75
C
Enter score: 0.5
F
```

برنامه را چندین بار اجرا کنید تا مقادیر مختلف ورودی را آزمایش کنید.

# فصل پنجم

# تکرار

## ۵,۱ به روز رسانی متغیرها

یک الگوی رایج در دستورات انتساب، دستور انتسابی است که یک متغیر را بروزرسانی می‌کند، جایی که مقدار جدید متغیر بستگی به مقدار قدیمی آن دارد.

```
x = x + 1
```

این به این معناست که "مقدار فعلی x را بگیر، ۱ را به آن اضافه کن، و سپس x را با مقدار جدید بروزرسانی کن".

اگر بخواهید متغیری را بروزرسانی کنید که وجود ندارد، با خطا مواجه می‌شوید، زیرا پایتون ابتدا طرف راست عبارت را ارزیابی می‌کند قبل از آن که مقداری به x اختصاص دهد:

```
>>> x = x + 1
NameError: name 'x' is not defined
```

قبل از اینکه بتوانید یک متغیر را بروزرسانی کنید، باید آن را مقداردهی اولیه کنید، معمولاً با یک تخصیص ساده:

```
>>> x = 0
>>> x = x + 1
```

بروزرسانی یک متغیر با افزودن ۱ به آن، به نام "افزایش (increment) "شناخته می‌شود؛ در حالی که کم کردن ۱ به نام "کاهش " (decrement)شناخته می‌شود.

## ۵,۲ دستور While

کامپیوترها اغلب برای خودکارسازی وظایف تکراری استفاده می‌شوند. تکرار کارهای مشابه یا یکسان بدون ایجاد اشتباه، کاری است که کامپیوترها به خوبی انجام می‌دهند، در حالی که افراد به سختی این کار را انجام می‌دهند. به همین دلیل، از آنجا که تکرار بسیار رایج است، پایتون ویژگی‌های مختلفی برای تسهیل این کار فراهم کرده است.

یکی از اشکال تکرار در پایتون، دستور `while` است. در اینجا یک برنامه ساده است که از پنج شروع کرده هر عدد را در خروجی نمایش می دهد تا به صفر برسد و سپس به "Blastoff!" می‌رسد.

```
n = 5
while n > 0:
    print(n)
```

```
    n = n - 1
print('Blastoff!')
```

شما می‌توانید دستور while را تقریباً به همان شیوه‌ای که به زبان انگلیسی می‌خوانید، بخوانید. این جمله به این معنی است: "تا زمانی که n بزرگتر از ۰ است، مقدار n را نمایش بده و سپس مقدار n را یکی کاهش بده. وقتی به ۰ رسیدی، از دستور while خارج شو و کلمه 'Blastoff!' را نمایش بده".

به طور رسمی‌تر، جریان اجرای دستور while به صورت زیر است:

۱. شرایط را ارزیابی کن تا نتیجه True یا False بدست آید.
۲. اگر نتیجه False باشد، از دستور while خارج شو و اجرای برنامه را در دستور بعدی ادامه بده.
۳. اگر نتیجه True باشد، بدنه را اجرا کن و سپس به مرحله ۱ بازگرد .این نوع جریان به نام "حلقه (loop) "شناخته می‌شود زیرا در مرحله سوم، دوباره به بالای حلقه باز می‌گردیم. هر بار که بدنه حلقه اجرا می‌شود، آن را یک "تکرار " (iteration) می‌نامیم. برای حلقه بالا، می‌گوییم: "این حلقه پنج تکرار داشت"، که به این معنی است که بدنه حلقه پنج بار اجرا شد.

بدنه حلقه باید مقداری از متغیرها را تغییر دهد تا در نهایت شرط False شده و حلقه متوقف شود. متغیری که در هر بار اجرای حلقه تغییر می‌کند و زمان خاتمه حلقه را کنترل می‌کند، "متغیر تکرار (iteration variable) "نامیده می‌شود. اگر چنین متغیری وجود نداشته باشد، حلقه به طور بی‌پایان تکرار خواهد شد که منجر به یک "حلقه بی‌نهایت" می‌شود.

## ۵,۳ حلقه های بی پایان

یک منبع بی‌پایان از سرگرمی برای برنامه‌نویسان دستورالعمل شامپو است، "Lather, rinse, repeat" (شستشو، آبکشی، تکرار)، یک حلقه بی‌پایان است زیرا هیچ متغیر تکراری وجود ندارد که به شما بگوید چند بار باید حلقه را اجرا کنید.

در مورد شمارش معکوس، ما می‌توانیم ثابت کنیم که حلقه به اتمام می‌رسد، زیرا می‌دانیم که مقدار n محدود است و می‌توانیم ببینیم که مقدار n هر بار که حلقه اجرا می‌شود کوچک‌تر می‌شود، بنابراین در نهایت به صفر خواهیم رسید. در موارد دیگر، یک حلقه به وضوح بی‌پایان است زیرا اصلاً متغیر تکرار وجود ندارد.

گاهی اوقات شما نمی‌دانید که چه زمانی باید حلقه را تمام کنید، مگر اینکه نصف مسیر بدنه حلقه را طی کنید. در این حالت، می‌توانید عمداً یک حلقه بی‌پایان بنویسید و سپس از دستور break برای خروج از حلقه استفاده کنید .این حلقه به وضوح یک حلقه بی‌پایان است زیرا عبارت منطقی در دستور while صرفاً ثابت منطقی True است:

```
n = 10
while True:
    print(n, end=' ')
    n = n - 1
print('Done!')
```

اگر شما اشتباه کنید و این کد را اجرا کنید، به سرعت خواهید آموخت که چگونه یک فرآیند در حال فرار پایتون را در سیستم خود متوقف کنید یا کجا دکمه خاموش را در کامپیوتر خود پیدا کنید. این برنامه برای همیشه اجرا خواهد شد یا تا زمانی که باتری شما تمام شود، زیرا عبارت منطقی در بالای حلقه همیشه درست است به دلیل این که عبارت ثابت منطقی True است.

در حالی که این یک حلقه بی‌پایان ناکارآمد است، هنوز می‌توانیم از این الگو برای ساخت حلقه‌های مفید استفاده کنیم به شرطی که به دقت کدی را به بدنه حلقه اضافه کنیم تا به طور صریح با استفاده از دستور break زمانی که به شرایط خروج رسیدیم، از حلقه خارج شویم.

برای مثال، فرض کنید شما می‌خواهید ورودی از کاربر بگیرید تا زمانی که آنها کلمه "done" را وارد کنند. شما می‌توانید اینطور کدنویسی کنید:

```
while True:
    line = input('> ')
    if line == 'done':
        break
    print(line)
print('Done!')

# Code: https://www.py4e.com/code3/copytildone1.py
```

شرط حلقه True است، که همیشه درست است، بنابراین حلقه به طور مداوم اجرا می‌شود تا زمانی که به دستور break برسد.

هر بار که از حلقه عبور می‌کند، از کاربر خواسته می‌شود که ورودی را وارد کند. اگر کاربر کلمه "done" را وارد کند، دستور break حلقه را خاتمه می‌دهد. در غیر این صورت، برنامه هر چیزی که کاربر وارد کرده است را بازتاب می‌کند و به ابتدای حلقه برمی‌گردد. در اینجا یک نمونه اجرای برنامه آورده شده است:

```
> hello there
hello there
> finished
finished
> done
Done!
```

این روش نوشتن حلقه‌های while رایج است زیرا می‌توانید شرط را در هر جایی از حلقه بررسی کنید (نه فقط در ابتدا) و می‌توانید شرط توقف را به صورت مثبت بیان کنید ("متوقف شو وقتی که این اتفاق بیفتد")، به جای منفی ("ادامه بده تا وقتی که این اتفاق بیفتد").

## ۵,۴ به اتمام تکرارها با استفاده از دستور continue

گاهی اوقات شما در یک تکرار از یک حلقه قرار دارید و می‌خواهید تکرار جاری را تمام کرده و بلافاصله به تکرار بعدی بروید. در این صورت می‌توانید از دستور continue استفاده کنید تا تکرار جاری را نادیده بگیرید و به تکرار بعدی بروید بدون اینکه بدنه حلقه برای تکرار جاری تمام شود.

در اینجا یک مثال از یک حلقه آورده شده که ورودی خود را کپی می‌کند تا زمانی که کاربر "done" را وارد کند، اما خطوطی که با علامت هش (مربوط به کامنت‌های پایتون) شروع می‌شوند را نادیده می‌گیرد:

```
while True:
    line = input('> ')
    if line[0] == '#':
        continue
    if line == 'done':
        break
    print(line)
print('Done!')

# Code: https://www.py4e.com/code3/copytildone2.py
```

در اینجا یک اجرای نمونه از این برنامه جدید با دستور continue اضافه شده است:

```
> hello there
hello there
> # don't print this
> print this!
print this!
> done
Done!
```

تمامی خطوط چاپ می‌شوند مگر خطی که با علامت هش شروع می‌شود، زیرا زمانی که دستور continue اجرا می‌شود، تکرار جاری خاتمه می‌یابد و به دستور while باز می‌گردد تا تکرار بعدی شروع شود، بدین ترتیب دستور print نادیده گرفته می‌شود.

## ۵,۵ حلقه های قطعی با استفاده از for

گاهی اوقات می‌خواهیم از یک مجموعه از اشیاء مانند لیستی از کلمات، خطوط یک فایل یا لیستی از اعداد عبور کنیم. زمانی که یک مجموعه از اشیاء برای تکرار داریم، می‌توانیم یک حلقه قطعی با استفاده از دستور for بسازیم. دستور while را حلقه نامحدود می‌نامیم زیرا تا زمانی که یک شرط به False تبدیل شود تکرار می‌کند، در حالی که حلقه for از یک مجموعه شناخته شده از اشیاء عبور می‌کند و به اندازه تعداد اشیاء در مجموعه تکرار می‌شود.

نحو دستور for مشابه حلقه while است که شامل یک دستور for و بدنه حلقه می‌باشد:

```
friends = ['Joseph', 'Glenn', 'Sally']
for friend in friends:
    print('Happy New Year:', friend)
print('Done!')
```

از نظر پایتون، متغیر friends یک لیست از سه رشته است و حلقه for از طریق این لیست عبور می‌کند و بدنه حلقه را برای هر یک از سه رشته موجود در لیست یک‌بار اجرا می‌کند که نتیجه‌ی آن خروجی زیر خواهد بود:

```
Happy New Year: Joseph
Happy New Year: Glenn
Happy New Year: Sally
Done!
```

ترجمه‌ی این حلقه for به زبان انگلیسی به‌طور مستقیم به اندازه‌ی حلقه while نیست، اما اگر به friends به‌عنوان یک مجموعه نگاه کنید، به این صورت است: "دستورات داخل بدنه‌ی حلقه for را برای هر عضو از مجموعه‌ی به نام friends یک‌بار اجرا کن". با نگاهی به حلقه for، کلمات کلیدی forو in را می بینید که از کلمات رزرو شده در پایتون هستند، و friend و friendsمتغیرها هستند.

```
for friend in friends:
    print('Happy New Year:', friend)
```

نکته، friend متغیر تکرار در حلقه for است. این متغیر در هر بار تکرار حلقه تغییر می‌کند و کنترل می‌کند که حلقه for کی تمام شود. متغیر تکرار به‌طور متوالی از میان سه رشته ذخیره‌شده در متغیر friends عبور می‌کند.

## ۵,۶ الگوهای حلقه

معمولاً ما از حلقه‌های for یا while برای عبور از یک لیست از موارد یا محتوای یک فایل استفاده می‌کنیم و به دنبال چیزی مانند بزرگ‌ترین یا کوچک‌ترین مقدار داده‌هایی هستیم که از آن‌ها عبور می‌کنیم .این حلقه‌ها معمولاً به‌طور زیر ساخته می‌شوند:

- مقداردهی اولیه به یک یا چند متغیر قبل از شروع حلقه
- انجام محاسباتی روی هر مورد در بدنه حلقه، که ممکن است متغیرها را در بدنه حلقه تغییر دهد
- بررسی متغیرهای نتیجه‌گیری شده زمانی که حلقه تمام می‌شود

ما از یک لیست از اعداد برای نمایش مفاهیم و ساخت این الگوهای حلقه استفاده خواهیم کرد.

### ۵,۶,۱ حلقه‌های شمارش و جمع زدن

برای مثال، برای شمارش تعداد موارد در یک لیست، ما حلقه `for` زیر را می‌نویسیم:

```
count = 0
for itervar in [3, 41, 12, 9, 74, 15]:
    count = count + 1
print('Count: ', count)
```

ما متغیر `count` را قبل از شروع حلقه صفر قرار می‌دهیم، سپس یک حلقه `for` می‌نویسیم تا از روی لیست اعداد عبور کند. متغیر تکرار ما `itervar` نام دارد و در حالی که از آن در بدنه حلقه استفاده نمی‌کنیم، اما همانطور که حلقه را کنترل می‌کند و باعث اجرای بدنه حلقه یک‌بار برای هر کدام از مقادیر موجود در لیست می‌شود.

در بدنه حلقه، برای هر کدام از مقادیر موجود در لیست، یک واحد به مقدار فعلی `count` اضافه می‌کنیم. در حین اجرای حلقه، مقدار `count` تعداد مقادیری است که تاکنون دیده‌ایم .پس از اتمام حلقه، مقدار `count` تعداد کل موارد است. عدد کل در انتهای حلقه به دست می‌آید. ما حلقه را طوری می‌سازیم که زمانی که حلقه تمام شد، آنچه که می‌خواهیم به دست آید .حلقه مشابه دیگری که مجموع مجموعه‌ای از اعداد را محاسبه می‌کند به صورت زیر است:

```
total = 0
for itervar in [3, 41, 12, 9, 74, 15]:
    total = total + itervar
print('Total: ', total)
```

در این حلقه، از متغیر تکرار استفاده می‌کنیم. به جای اضافه کردن یک به شمارش مانند حلقه قبلی، عدد واقعی (۳، ۴۱، ۱۲ و غیره) را به مجموع جاری در هر تکرار حلقه اضافه می‌کنیم. اگر به متغیر `total` فکر کنید، این متغیر "مجموع مقادیر تاکنون" را نگه می‌دارد. بنابراین قبل از شروع حلقه، `total` صفر است چون هنوز هیچ مقداری را ندیده‌ایم، در طول حلقه، `total` مجموع جاری است و در پایان حلقه، `total` مجموع کلی تمام مقادیر موجود در لیست است.

هنگامی که حلقه اجرا می‌شود، `total` مجموع عناصر را انباشته می‌کند؛ متغیری که این‌گونه استفاده می‌شود گاهی به نام "انباشته‌کننده" شناخته می‌شود.

نه حلقه شمارش و نه حلقه جمع‌بندی در عمل چندان مفید نیستند چون توابع داخلی `len()` و `sum()` وجود دارند که به ترتیب تعداد اقلام موجود در یک لیست و مجموع اقلام آن لیست را محاسبه می‌کنند.

### ۵,۶,۲ حلقه‌های بیشترین و کمترین مقدار

برای پیدا کردن بزرگترین مقدار در یک لیست یا دنباله، حلقه زیر را می‌سازیم:

```
largest = None
print('Before:', largest)
for itervar in [3, 41, 12, 9, 74, 15]:
```

```
    if largest is None or itervar > largest :
        largest = itervar
    print('Loop:', itervar, largest)
print('Largest:', largest)
```

زمانی که برنامه اجرا شود خروجی به صورت زیر خواهد بود

```
Before: None
Loop: 3 3
Loop: 41 41
Loop: 12 41
Loop: 9 41
Loop: 74 74
Loop: 15 74
Largest: 74
```

متغیر largest بهتر است به عنوان "بزرگترین مقداری که تاکنون دیده‌ایم" در نظر گرفته شود. قبل از شروع حلقه، ما largest را به مقدار ثابت None تنظیم می‌کنیم .یک مقدار ثابت ویژه است که می‌توانیم آن را در یک متغیر ذخیره کنیم تا نشان دهیم متغیر "خالی" است.

قبل از شروع حلقه، بزرگترین مقداری که تاکنون دیده‌ایم None است چون هنوز هیچ مقداری را مشاهده نکرده‌ایم. در حین اجرای حلقه، اگر largest برابر با None باشد، اولین مقداری که می‌بینیم را به عنوان بزرگترین مقدار تاکنون در نظر می‌گیریم. در اولین تکرار، وقتی که مقدار itervar برابر ۳ است، چون largest برابر با None است، بلافاصله largestرا برابر با ۳ قرار می‌دهیم.

بعد از اولین تکرار، largest دیگر برابر با None نیست، بنابراین بخش دوم عبارت منطقی ترکیبی که بررسی می‌کند آیا itervar بزرگتر از largest است تنها زمانی فعال می‌شود که مقداری بزرگتر از "بزرگترین مقدار تاکنون" ببینیم. زمانی که مقدار جدیدی که "بزرگتر" است مشاهده می‌کنیم، آن مقدار جدید را به عنوان largest در نظر می‌گیریم. می‌توانید در خروجی برنامه مشاهده کنید که مقدار largest از ۳ به ۴۱ و سپس به ۷۴ تغییر می‌کند.

در پایان حلقه، ما تمام مقادیر را بررسی کرده‌ایم و متغیر largest حالا بزرگترین مقدار موجود در لیست را در خود دارد.

برای محاسبه کوچکترین عدد، کد بسیار مشابه است با یک تغییر کوچک:

```
smallest = None
print('Before:', smallest)
for itervar in [3, 41, 12, 9, 74, 15]:
    if smallest is None or itervar < smallest:
        smallest = itervar
    print('Loop:', itervar, smallest)
print('Smallest:', smallest)
```

دوباره، `smallest` به عنوان "کوچکترین مقدار تاکنون" قبل، در حین، و پس از اجرای حلقه عمل می‌کند. زمانی که حلقه به پایان می‌رسد، `smallest` کوچکترین مقدار موجود در لیست را در خود نگه می‌دارد.

دوباره مانند شمارش و جمع کردن، توابع داخلی `max()` و `min()` نوشتن این نوع حلقه‌ها را غیرضروری می‌کنند.

در زیر نسخه ساده‌ای از تابع داخلی `min()` در پایتون آورده شده است:

```
def min(values):
    smallest = None
    for value in values:
        if smallest is None or value < smallest:
            smallest = value
    return smallest
```

در نسخه تابعی کد `smallest`، تمام دستورات `print` حذف شده‌اند تا معادل تابع داخلی `min` که قبلاً در پایتون وجود دارد، باشد.

## ۵,۷ اشکال زدایی

هنگامی که شروع به نوشتن برنامه‌های بزرگتر می‌کنید، ممکن است متوجه شوید که وقت زیادی را صرف اشکال‌زدایی می‌کنید. کد بیشتر به معنای فرصت‌های بیشتر برای ایجاد خطا و مکان‌های بیشتر برای پنهان شدن باگ‌ها است.

یکی از روش‌های کاهش زمان اشکال‌زدایی، "اشکال‌زدایی با تقسیم" است. به عنوان مثال، اگر ۱۰۰ خط در برنامه‌تان وجود دارد و شما آنها را یکی یکی بررسی می‌کنید، این کار ۱۰۰ قدم طول می‌کشد.

در عوض، سعی کنید مشکل را نصف کنید. به وسط برنامه یا نزدیک به آن نگاه کنید تا مقداری میانه‌ای که می‌توانید بررسی کنید، پیدا کنید. یک دستور `print`( یا چیزی دیگر که تأثیر قابل تأیید داشته باشد) اضافه کنید و برنامه را اجرا کنید.

اگر بررسی در نقطه میانه نادرست باشد، مشکل باید در نیمه اول برنامه باشد. اگر درست باشد، مشکل در نیمه دوم است.

هر بار که چنین بررسی‌ای انجام می‌دهید، تعداد خطوطی که باید جستجو کنید را نصف می‌کنید. پس از شش قدم (که بسیار کمتر از ۱۰۰ است)، شما به یک یا دو خط کد خواهید رسید، حداقل در تئوری.

در عمل، همیشه مشخص نیست که "وسط برنامه" کجا است و همیشه ممکن نیست که آن را بررسی کنید. منطقی نیست که خطوط را بشمارید و نقطه دقیق میانه را پیدا کنید. در عوض، به مکان‌هایی در برنامه فکر کنید که احتمال خطا وجود دارد و مکان‌هایی که بررسی آن آسان است. سپس نقطه‌ای را انتخاب کنید که احتمالاً برابر است که باگ قبل یا بعد از آن بررسی باشد.

## ۵,۸ واژه نامه

**جمع‌کننده**: متغیری که در یک حلقه برای جمع‌آوری یا انباشتن نتیجه استفاده می‌شود.

**شمارنده**: متغیری که در یک حلقه برای شمارش تعداد دفعاتی که چیزی اتفاق افتاده است استفاده می‌شود. یک شمارنده را با صفر مقداردهی اولیه می‌کنیم و سپس هر بار که بخواهیم چیزی را "شمارش" کنیم، شمارنده را افزایش می‌دهیم.

**کاهش**: به‌روزرسانی که مقدار یک متغیر را کاهش می‌دهد.

**مقداردهی اولیه**: یک انتساب که به متغیری که قرار است به‌روزرسانی شود، مقدار اولیه می‌دهد.

**افزایش**: به‌روزرسانی که مقدار یک متغیر را افزایش می‌دهد (معمولاً به میزان یک)

**حلقه بی‌پایان**: حلقه‌ای که شرط خاتمه آن هرگز برآورده نمی‌شود یا حلقه‌ای که هیچ شرطی برای خاتمه ندارد.

**تکرار**: اجرای مکرر یک مجموعه از دستورات با استفاده از یک تابع که خودش را فراخوانی می‌کند یا با استفاده از یک حلقه.

## ۵,۹ تمرین ها

**تمرین ۱:** برنامه‌ای بنویسید که به‌صورت مکرر اعداد صحیح را بخواند تا زمانی‌که کاربر «done» را وارد کند. پس از وارد شدن «done»، مجموع، تعداد، و میانگین اعداد صحیح واردشده را چاپ کنید. اگر کاربر چیزی غیر از یک عدد صحیح وارد کرد، با استفاده از `try` و `except` خطای او را تشخیص داده، یک پیام خطا چاپ کنید و به سراغ عدد صحیح بعدی بروید.

```
Enter a number: 4
Enter a number: 5
Enter a number: bad data
Invalid input
Enter a number: 7
Enter a number: done
16 3 5.333333333333333
```

**تمرین ۲:** یک برنامه‌ی دیگر بنویسید که همانند تمرین قبل، برای دریافت فهرستی از اعداد از کاربر درخواست کند و در پایان، به‌جای میانگین، **بیشترین** (**maximum**) و **کمترین** (**minimum**) عدد واردشده را چاپ کند.

# فصل ششم

# رشته‌ها

## ۶,۱ رشته یک دنباله از حروف است

رشته یک دنباله از حروف است. شما می‌توانید حروف را یکی‌یکی با استفاده از عملگر براکت ([]) دسترسی پیدا کنید.

```
>>> fruit = 'banana'
>>> letter = fruit[1]
```

بیانیه دوم، کاراکتر در موقعیت ایندکس ۱ را از متغیر `fruit` استخراج کرده و به متغیر `letter` نسبت می‌دهد. عبارت داخل براکت‌ها به نام ایندکس شناخته می‌شود. ایندکس مشخص می‌کند که کدام کاراکتر را از دنباله می‌خواهید (به همین دلیل نامش ایندکس است).

اما ممکن است آنچه انتظار دارید را دریافت نکنید:

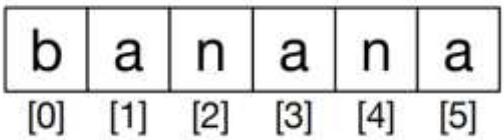
```
>>> print(letter)
a
```

برای بیشتر افراد، حرف اول "banana" باید "b" باشد، نه "a". اما در پایتون، ایندکس یک جابجایی از ابتدای رشته است و جابجایی حرف اول صفر است.

```
>>> letter = fruit[0]
>>> print(letter)
b
```

بنابراین "b" حرف ۰ام ("صفرم") از "banana" است، "a"حرف ۱ام ("اول") و "n" حرف ۲ام ("دوم") است. شما می‌توانید از هر عبارت، از جمله متغیرها و عملگرها، به عنوان ایندکس استفاده کنید، اما مقدار ایندکس باید یک عدد صحیح باشد. در غیر این صورت، با خطا مواجه خواهید شد:

```
>>> letter = fruit[1.5]
TypeError: string indices must be integers
```



## ۶,۲ دریافت طول یک رشته با استفاده از len

`len` یک تابع داخلی است که تعداد کاراکترهای موجود در یک رشته را باز می‌گرداند:

```
>>> fruit = 'banana'
>>> len(fruit)
6
```

برای به دست آوردن آخرین حرف یک رشته، ممکن است وسوسه شوید چیزی شبیه به این را امتحان کنید :

```
>>> length = len(fruit)
>>> last = fruit[length]
IndexError: string index out of range
```

دلیل بروز خطای IndexError این است که در رشته "banana" هیچ حرفی با ایندکس ۶ وجود ندارد. از آنجا که از صفر شروع به شماره‌گذاری کرده‌ایم، شش حرف با شماره‌های ۰ تا ۵ مشخص شده‌اند. برای به دست آوردن آخرین حرف، باید ۱ از طول رشته کم کنید:

```
>>> last = fruit[length-1]
>>> print(last)
a
```

به طور جایگزین، می‌توانید از ایندکس‌های منفی استفاده کنید که از انتهای رشته به عقب شمارش می‌کنند. عبارت `fruit[-1]` آخرین حرف را برمی‌گرداند، `fruit[-2]` دومین حرف از انتها را می‌دهد و به همین ترتیب.

## ۶,۳ عبور از یک رشته با استفاده از یک حلقه

بسیاری از محاسبات شامل پردازش یک رشته به صورت تک‌تک حروف است. معمولاً پردازش از ابتدا شروع می‌شود، هر حرف به نوبت انتخاب می‌شود، روی آن عملی انجام می‌شود و این فرآیند ادامه می‌یابد تا به انتهای رشته برسد. این الگوی پردازش به نام "عبور (Traversal) "شناخته می‌شود. یکی از روش‌های نوشتن این نوع پردازش با استفاده از یک حلقه while است:

```
index = 0
while index < len(fruit):
    letter = fruit[index]
    print(letter)
    index = index + 1
```

این حلقه رشته را عبور می‌دهد و هر حرف را در یک خط جداگانه نمایش می‌دهد. شرط حلقه `index < len(fruit)` است، بنابراین وقتی که مقدار `index` برابر طول رشته می‌شود، شرط نادرست شده و بدنه حلقه اجرا نمی‌شود. آخرین حرفی که دسترسی پیدا می‌کند، حرفی است که ایندکس آن `len(fruit)-1` است، که همان آخرین حرف رشته است.

**تمرین ۱:** یک حلقه while بنویسید که از آخرین حرف رشته شروع کرده و به سمت اولین حرف در رشته حرکت کند و هر حرف را در یک خط جداگانه چاپ کند، به جز اینکه به صورت معکوس باشد.

روش دیگری برای نوشتن یک traversal با استفاده از حلقه `for` وجود دارد:

```
for char in fruit:
    print(char)
```

هر بار که از حلقه عبور می‌کنیم، حرف بعدی در رشته به متغیر `char` اختصاص داده می‌شود. حلقه ادامه پیدا می‌کند تا هیچ کاراکتری باقی نماند.

## ۶,۴ برش های رشته

یک بخش از یک رشته به نام برش (slice) شناخته می‌شود. انتخاب یک برش مشابه انتخاب یک کاراکتر است:

```
>>> s = 'Monty Python'
>>> print(s[0:5])
Monty
>>> print(s[6:12])
Python
```

عملگر `[n:m]` قسمتی از رشته را از کاراکتر-n" ام" تا کاراکتر-m" ام" برمی‌گرداند، که اولین کاراکتر را شامل می‌شود ولی آخرین کاراکتر را شامل نمی‌شود. اگر اندیس اول (قبل از دو نقطه) را حذف کنید، برش از ابتدای رشته شروع می‌شود. اگر اندیس دوم را حذف کنید، برش تا انتهای رشته ادامه پیدا می‌کند:

```
>>> fruit = 'banana'
>>> fruit[:3]
'ban'
>>> fruit[3:]
'ana'
```

اگر اندیس اول بزرگتر یا مساوی با اندیس دوم باشد، نتیجه یک رشته خالی خواهد بود که با دو کوتیشن مارک نشان داده می‌شود:

```
>>> fruit = 'banana'
>>> fruit[3:3]
''
```

یک رشته خالی هیچ کاراکتری ندارد و طول آن صفر است، اما از نظر عملکرد مشابه هر رشته دیگری است.

**تمرین ۲:** با توجه به اینکه fruit یک رشته است، fruit[:] چه معنایی دارد؟

## ۶,۵ رشته ها غیرقابل تغییر هستند

ممکن است وسوسه شوید که یک کارکتر از یک رشته را با دستور انتساب تغییر دهید. به عنوان مثال:

```
>>> greeting = 'Hello, world!'
>>> greeting[0] = 'J'
TypeError: 'str' object does not support item assignment
```

"شیء" در اینجا رشته است و "مورد" همان کاراکتری است که شما سعی کردید به آن انتساب دهید. تا اینجا، یک شیء همان چیزی است که یک مقدار نامیده می‌شود، اما ما این تعریف را بعداً دقیق‌تر خواهیم کرد. یک مورد یکی از مقادیری است که در یک دنباله وجود دارد.

دلیل خطای رخ داده این است که رشته‌ها غیرقابل تغییر هستند، به این معنی که شما نمی‌توانید یک رشته موجود را تغییر دهید. بهترین کاری که می‌توانید انجام دهید این است که یک رشته جدید بسازید که نسخه تغییریافته از نسخه اصلی باشد:

```
>>> greeting = 'Hello, world!'
>>> new_greeting = 'J' + greeting[1:]
>>> print(new_greeting)
Jello, world!
```

این مثال یک حرف جدید به ابتدا از رشته‌ی greeting می‌چسباند. این عمل هیچ تأثیری روی رشته‌ی اصلی ندارد.

## ۶,۶ حلقه زنی و شمارش

برنامه‌ی زیر تعداد دفعاتی که حرف "a" در یک رشته ظاهر می‌شود را می‌شمارد:

```
word = 'banana'
count = 0
for letter in word:
    if letter == 'a':
        count = count + 1
print(count)
```

این برنامه الگوی دیگری از محاسبات را به نام شمارنده نشان می‌دهد. متغیر count با ۰ مقداردهی اولیه می‌شود و سپس هر بار که حرف "a" پیدا می‌شود، مقدار آن افزایش می‌یابد. زمانی که حلقه تمام می‌شود، مقدار count نتیجه را در خود دارد: مجموع تعداد "a"ها.

**تمرین ۳:** این کد را در یک تابع به نام count قرار دهید و آن را عمومی کنید به‌طوری‌که رشته و حرف را به عنوان آرگومان دریافت کند.

## ۶,۷ عملگر in

کلمه‌ی in یک عملگر بولی است که دو رشته را می‌گیرد و اگر اولین رشته به عنوان یک زیررشته در رشته‌ی دوم ظاهر شود، True برمی‌گرداند:

```
>>> 'a' in 'banana'
True
>>> 'seed' in 'banana'
False
```

## ۶,۸ مقایسه رشته ها

عملیات مقایسه بر روی رشته‌ها نیز کار می‌کنند. برای اینکه بتوانید بررسی کنید آیا دو رشته برابر هستند:

```
if word == 'banana':
    print('All right, bananas.')
```

عملیات مقایسه دیگری نیز برای قرار دادن کلمات در ترتیب الفبایی مفید هستند:

```
if word < 'banana':
    print('Your word,' + word + ', comes before banana.')
elif word > 'banana':
    print('Your word,' + word + ', comes after banana.')
else:
    print('All right, bananas.')
```

پایتون حروف بزرگ و کوچک را به آن شکلی که مردم انجام می‌دهند، مدیریت نمی‌کند. تمام حروف بزرگ قبل از تمام حروف کوچک می‌آیند، بنابراین: کلمه‌ی، Pineapple، قبل از banana می‌آید.

یک روش معمول برای حل این مشکل این است که رشته‌ها را به یک فرمت استاندارد(مانند تمام حروف کوچک) تبدیل کنید، قبل از اینکه بخواهید مقایسه‌ای انجام دهید.

## ۶,۹ متدهای رشته‌ای

رشته‌ها نمونه‌ای از اشیاء پایتون هستند. یک شیء هم داده‌ها (خود رشته واقعی) و هم متدها را در بر دارد که به طور مؤثر توابعی هستند که درون شیء ساخته شده‌اند و برای هر نمونه از آن شیء قابل دسترسی هستند.

پایتون یک تابع به نام dir دارد که متدهای موجود برای یک شیء را فهرست می‌کند. تابع type نوع یک شیء را نشان می‌دهد و تابع dir متدهای موجود را نشان می‌دهد.

```
>>> stuff = 'Hello world'
>>> type(stuff)
<class 'str'>
>>> dir(stuff)
[... 'capitalize', 'casefold', 'center', 'count', 'encode',
'endswith', 'expandtabs', 'find', 'format', 'format_map',
'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit',
'isidentifier', 'islower', 'isnumeric', 'isprintable',
'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower',
'lstrip', 'maketrans', 'partition', 'replace', 'rfind',
'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip',
'split', 'splitlines', 'startswith', 'strip', 'swapcase',
'title', 'translate', 'upper', 'zfill']
>>> help(str.capitalize)
Help on method_descriptor:

capitalize(self, /)
    Return a capitalized version of the string.

    More specifically, make the first character have upper
    case and the rest lower case.
>>>
```

در حالی که تابع `dir` متدهای موجود را فهرست می‌کند و شما می‌توانید از `help` برای دریافت مستندات ساده‌ای در مورد یک متد استفاده کنید، منبع بهتر برای مستندات متدهای رشته‌ها، وب‌سایت زیر است

https://docs.python.org/library/stdtypes.html#string-methods.

فراخوانی یک متد مشابه به فراخوانی یک تابع است (ورودی‌هایی می‌گیرد و یک مقدار برمی‌گرداند)، اما نحو آن متفاوت است. برای فراخوانی یک متد، نام متد را به متغیر اضافه می‌کنیم و از نقطه (.) به عنوان جداکننده استفاده می‌کنیم.

به عنوان مثال، متد `upper` یک رشته را می‌گیرد و یک رشته جدید با تمام حروف بزرگ برمی‌گرداند: به جای نوشتن تابع به شکل `upper(word)`، از نحو متد به صورت `word.upper()` استفاده می‌کنیم.

```
>>> word = 'banana'
>>> new_word = word.upper()
>>> print(new_word)
BANANA
```

این فرم از نحو نقطه‌ای، نام متد upper و نام رشته‌ای که باید متد بر روی آن اعمال شود، یعنی word، را مشخص می‌کند. پرانتزهای خالی نشان می‌دهند که این متد هیچ آرگومانی دریافت نمی‌کند.

فراخوانی یک متد را "اجرا" یا "invocation" می‌نامیم؛ در این مورد، می‌گوییم که ما متد upper را بر روی کلمه word اجرا می‌کنیم.

برای مثال، یک متد رشته‌ای به نام find وجود دارد که موقعیت یک رشته را درون رشته‌ای دیگر جستجو می‌کند.

```
>>> word = 'banana'
>>> index = word.find('a')
>>> print(index)
1
```

در این مثال، ما متد find را بر روی word اجرا کرده و حرفی را که به دنبال آن هستیم به عنوان پارامتر به آن می‌دهیم. متد find می‌تواند زیررشته‌ها را نیز پیدا کند، نه فقط حروف:

```
>>> word.find('na')
2
```

این متد می‌تواند به عنوان آرگومان دوم، ایندکسی که باید جستجو از آنجا شروع شود را بپذیرد :

```
>>> word.find('na', 3)
4
```

یکی از وظایف رایج این است که فضای سفید (فاصله‌ها، تب‌ها یا خطوط جدید) را از ابتدای و انتهای یک رشته با استفاده از متد strip حذف کنیم:

```
>>> line = '  Here we go  '
>>> line.strip()
'Here we go'
```

برخی از متدها مانند startswith مقادیر بولی بازمی‌گردانند.

```
>>> line = 'Have a nice day'
>>> line.startswith('Have')
True
>>> line.startswith('h')
False
```

شما متوجه خواهید شد که startswith نیاز به تطابق حساس به حروف بزرگ و کوچک دارد، بنابراین گاهی اوقات یک خط را می‌گیریم و آن را قبل از انجام هرگونه بررسی با استفاده از متد lower به تمام حروف کوچک تبدیل می‌کنیم.

```
>>> line = 'Have a nice day'
>>> line.startswith('h')
False
>>> line.lower()
'have a nice day'
>>> line.lower().startswith('h')
True
```

در مثال آخر، متد lower فراخوانی می‌شود و سپس از startswith استفاده می‌کنیم تا بررسی کنیم آیا رشته‌ی تبدیل‌شده به حروف کوچک با حرف "h" شروع می‌شود. به شرطی که ترتیب را درست رعایت کنیم، می‌توانیم چندین فراخوانی متد را در یک عبارت واحد انجام دهیم.

**تمرین ۴:** یک متد رشته‌ای به نام count وجود دارد که مشابه تابع در تمرین قبلی است. مستندات این متد را در لینک زیر بخوانید:

https://docs.python.org/library/stdtypes.html#string-methods

یک فراخوانی بنویسید که تعداد دفعاتی که حرف "a" در "banana" تکرار می‌شود را بشمارد.

## ۶,۱۰ تجزیه رشته‌ها

اغلب، ما می‌خواهیم درون یک رشته نگاه کنیم و یک زیررشته پیدا کنیم. به عنوان مثال، اگر یک سری خطوط با فرمت زیر به ما ارائه شده باشد:

```
From stephen.marquard@ uct.ac.za Sat Jan  5 09:14:16 2008
```

و اگر بخواهیم تنها نیمه دوم آدرس (یعنی (uct.ac.za) را از هر خط استخراج کنیم، می‌توانیم این کار را با استفاده از متد find و برش رشته انجام دهیم.

ابتدا موقعیت علامت @ را در رشته پیدا می‌کنیم. سپس موقعیت اولین فاصله بعد از علامت @ را پیدا می‌کنیم. و سپس از برش رشته برای استخراج بخش مورد نظر استفاده می‌کنیم.

```
>>> data = 'From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008'
>>> atpos = data.find('@')
>>> print(atpos)
21
>>> sppos = data.find(' ',atpos)
```

```
>>> print(sppos)
31
>>> host = data[atpos+1:sppos]
>>> print(host)
uct.ac.za
>>>
```

ما از نسخه‌ای از متد `find` استفاده می‌کنیم که به ما این امکان را می‌دهد که موقعیتی را در رشته مشخص کنیم که می‌خواهیم متد `find` از آن نقطه شروع به جستجو کند. هنگام برش، ما کاراکترها را از "یک مکان بعد از علامت @ تا قبل از کاراکتر فاصله" استخراج می‌کنیم. مستندات متد `find` در دسترس است در:

https://docs.python.org/library/stdtypes.html#string-methods.

## ۶,۱۱ رشته‌های قالب‌بندی شده

یک رشته قالب‌بندی شده (که معمولاً به آن f-string گفته می‌شود) این امکان را می‌دهد که عبارات پایتون درون رشته‌های معمولی استفاده شوند. این کار با اضافه کردن حرف f به ابتدای رشته و قرار دادن عبارات داخل آکولادها {} انجام می‌شود .به عنوان مثال، قرار دادن نام یک متغیر داخل آکولادها در یک f-string باعث می‌شود که با مقدار آن متغیر جایگزین شود:

```
>>> camels = 42
>>> f'{camels}'
'42'
```

نتیجه رشته «۴۲» است که نباید با مقدار صحیح ۴۲ اشتباه گرفته شود .یک عبارت می‌تواند در هر جایی از رشته ظاهر شود، بنابراین می‌توانید یک مقدار را در داخل یک جمله قرار دهید:

```
>>> camels = 42
>>> f'I have spotted {camels} camels.'
'I have spotted 42 camels.'
```

چندین عبارت می‌توانند در داخل یک رشته قالب‌بندی‌شده (f-string) گنجانده شوند تا رشته‌های پیچیده‌تری ایجاد کنند.

```
>>> years = 3
>>> count = .1
>>> species = 'camels'
>>> f'In {years} years I have spotted {count} {species}.'
'In 3 years I have spotted 0.1 camels.'
```

رشته‌های قالب‌بندی‌شده (f-string) قدرتمند هستند و حتی بیشتر از آنچه که در اینجا پوشش داده شده می‌توانند انجام دهند. می‌توانید بیشتر در مورد آن‌ها در این لینک بخوانید:

https://docs.python.org/3/tutorial/inputoutput.html#formatted-string-literals.

## ۶,۱۲ اشکال‌زدایی

یک مهارت که باید هنگام برنامه‌نویسی در خود پرورش دهید این است که همیشه از خود بپرسید، «چه چیزی ممکن است اینجا اشتباه پیش برود؟» یا به طور متناوب، «چه کار عجیبی ممکن است کاربر ما انجام دهد که باعث کرش شدن برنامه (به نظر بی‌عیب و نقص) ما شود؟» برای مثال، به برنامه‌ای که برای نشان دادن حلقه while در فصل تکرار استفاده کردیم نگاه کنید:

```
while True:
    line = input('> ')
    if line[0] == '#':
        continue
    if line == 'done':
        break
    print(line)
print('Done!')

# Code: https://www.py4e.com/code3/copytildone2.py
```

ببینید چه اتفاقی می‌افتد وقتی کاربر یک خط ورودی خالی وارد می‌کند:

```
> hello there
hello there
> # don't print this
> print this!
print this!
>
Traceback (most recent call last):
  File "copytildone.py", line 3, in <module>
    if line[0] == '#':
IndexError: string index out of range
```

کد تا زمانی که خطی خالی به آن ارائه نشود به درستی کار می‌کند. اما وقتی خط خالی می‌شود، هیچ کاراکتر صفرمی وجود ندارد و خطای traceback ایجاد می‌شود. دو راه برای این که خط سوم "ایمن" باشد حتی اگر خط خالی باشد، وجود دارد.

یک امکان این است که از متد `startswith` استفاده کنیم که در صورتی که رشته خالی باشد، مقدار False باز می‌گرداند.

```
if line.startswith('#'):
```

راه دیگر این است که دستور if را به‌طور ایمن با استفاده از الگوی نگهبان (guardian pattern) بنویسید و اطمینان حاصل کنید که تنها در صورتی که حداقل یک کاراکتر در رشته وجود داشته باشد، عبارت منطقی دوم ارزیابی شود:

```
if len(line) > 0 and line[0] == '#':
```

## ۶,۱۳ واژه‌نامه

**شمارنده(counter)** : یک متغیر که برای شمارش چیزی استفاده می‌شود، معمولاً با صفر مقداردهی اولیه می‌شود و سپس هر بار که چیزی شمارش می‌شود، افزایش می‌یابد.

**رشته خالی(empty string)** : رشته‌ای که هیچ کاراکتری ندارد و طول آن صفر است و با دو علامت نقل قول نشان داده می‌شود.

**فلگ(flag)** : یک متغیر بولی که برای نشان دادن اینکه یک شرط درست یا غلط است، استفاده می‌شود.

**فراخوانی(invocation)** : بیانیه‌ای که یک متد را فراخوانی می‌کند.

**غیرقابل تغییر(immutable)** : خصوصیتی از یک توالی که اقلام آن قابل تخصیص نیستند.

**شاخص(index)** : یک مقدار صحیح که برای انتخاب یک آیتم از یک توالی استفاده می‌شود، مانند یک کاراکتر در یک رشته.

**آیتم(item)** : یکی از مقادیر موجود در یک توالی.

**متد(method)** : یک تابع که به یک شیء تعلق دارد و با استفاده از نشانه‌گذاری نقطه‌ای فراخوانی می‌شود.

**شیء(object)** : چیزی که یک متغیر می‌تواند به آن اشاره کند. در حال حاضر، می‌توانید از "شیء" و "مقدار" به صورت متناوب استفاده کنید.

**جستجو(search)** : یک الگوی پیمایش که زمانی که چیزی را که به دنبالش است پیدا می‌کند، متوقف می‌شود.

**توالی(sequence)** : یک مجموعه مرتب؛ یعنی مجموعه‌ای از مقادیر که هر مقدار با یک شاخص عدد صحیح شناسایی می‌شود.

**برش(slice)** : بخشی از یک رشته که توسط یک بازه از شاخص‌ها مشخص شده است.

**پیمایش(traverse)** : برای تکرار روی اقلام یک توالی و انجام عملی مشابه بر روی هر یک از آن‌ها.

## ۶,۱۴ تمرین ها

**تمرین ۵:** برش زدن رشته‌ها . کد پایتون زیر را در نظر بگیرید که یک رشته را ذخیره می‌کند:

```
str = 'X-DSPAM-Confidence: 0.8475'
```

با استفاده از تابع find و برش رشته‌ای(string slicing) ، بخشِ رشته بعد از علامت دونقطه را استخراج کنید و سپس با استفاده از تابع float آن بخشِ استخراج‌شده را به یک عدد اعشاری (floating point) تبدیل کنید .

**تمرین ۶:** متدهای رشته مستندات مربوط به متدهای رشته را در این آدرس بخوانید:

https://docs.python.org/library/stdtypes.html#string-methods.

ممکن است بخواهید با برخی از این متدها آزمایش کنید تا مطمئن شوید نحوهٔ کار آن‌ها را درک کرده‌اید. متدهای `strip` و `replace` به‌ویژه کاربردی هستند. مستندات از نحوی استفاده می‌کنند که ممکن است برایتان گیج‌کننده باشد. برای مثال، در `find(sub[, start[, end]])` براکت‌ها نشان‌دهندهٔ آرگومان‌های اختیاری هستند. بنابراین، `sub` لازم است، اما `start`اختیاری است و اگر `start` را وارد کنید، در آن صورت `end` نیز اختیاری خواهد بود.

# فصل هفتم

# فایل‌ها

## ۷,۱ پایداری

تا به اینجا، یاد گرفته‌ایم که چگونه برنامه‌هایی بنویسیم و اهداف خود را با استفاده از اجرای شرطی، توابع و تکرارها به واحد پردازش مرکزی (CPU) منتقل کنیم. همچنین یاد گرفته‌ایم که چگونه داده‌ها را در حافظه اصلی ایجاد و استفاده کنیم. CPU و حافظه جایی هستند که نرم‌افزار ما در آنجا کار می‌کند و اجرا می‌شود. اینجا جایی است که همه «تفکر» انجام می‌شود .اما اگر به یاد داشته باشید از بحث‌های معماری سخت‌افزار، زمانی که برق قطع شود، هر چیزی که در CPU یا حافظه اصلی ذخیره شده باشد، پاک می‌شود. بنابراین تا کنون، برنامه‌های ما فقط تمرین‌های موقتی برای یادگیری پایتون بوده‌اند.

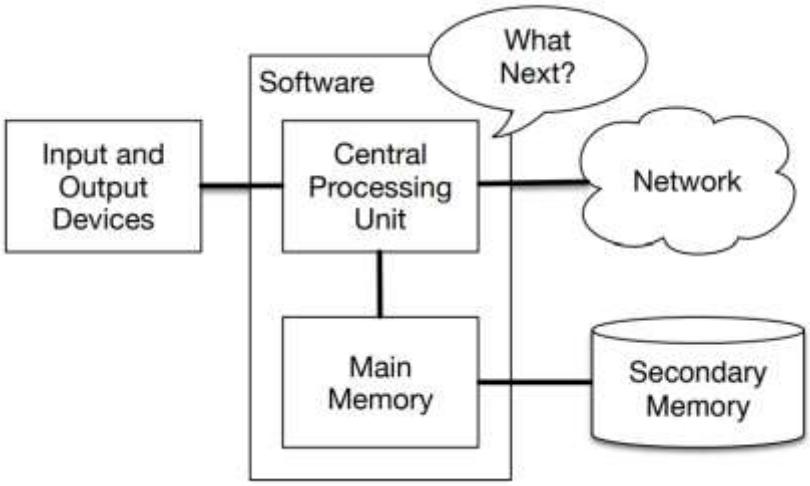
حافظه ثانویه در این فصل، شروع به کار با حافظه ثانویه (یا فایل‌ها) خواهیم کرد. حافظه ثانویه زمانی که برق قطع شود، پاک نمی‌شود. یا در صورت استفاده از یک درایو فلشUSB ، داده‌هایی که از برنامه‌هایمان می‌نویسیم می‌توانند از سیستم حذف شده و به سیستم دیگری منتقل شوند .ما عمدتاً بر خواندن و نوشتن فایل‌های متنی تمرکز خواهیم کرد، مانند فایل‌هایی که در یک ویرایشگر متن ایجاد می‌کنیم. در ادامه، خواهیم دید که چگونه با فایل‌های پایگاه داده که فایل‌های باینری هستند و به طور خاص برای خواندن و نوشتن از طریق نرم‌افزار پایگاه داده طراحی شده‌اند، کار کنیم.

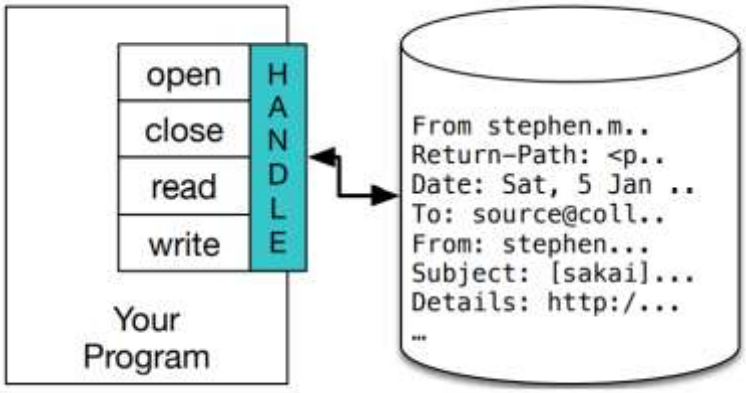
## ۷,۲ باز کردن فایل‌ها

زمانی که می‌خواهیم یک فایل را بخوانیم یا بنویسیم (مثلاً روی هارد دیسک خود)، ابتدا باید فایل را باز کنیم. باز کردن فایل ارتباطی با سیستم عامل شما برقرار می‌کند که می‌داند داده‌های هر فایل در کجا ذخیره شده است. زمانی که یک فایل را باز می‌کنید، از سیستم عامل می‌خواهید که فایل را با نام آن پیدا کند و اطمینان حاصل کند که فایل وجود دارد. در این مثال، فایل mbox.txt را باز می‌کنیم که باید در همان پوشه‌ای که وقتی پایتون را اجرا می‌کنید، قرار داشته باشد. شما می‌توانید این فایل را از

www.py4e.com/code3/mbox.txt

```
>>> fhand = open('mbox.txt')
>>> print(fhand)
<_io.TextIOWrapper name='mbox.txt' mode='r' encoding='cp1252'>
```




اگر فراخوانی تابع `open` موفق باشد، سیستم‌عامل یک «دسته فایل» (file handle) به ما بازمی‌گرداند. دسته فایل، خودِ داده‌های درون فایل نیست، بلکه نوعی «دستگیره» است که از طریق آن می‌توانیم داده‌های فایل را بخوانیم. شما تنها در صورتی این دستگیره را دریافت می‌کنید که فایل مورد نظر وجود داشته باشد و شما مجوزهای لازم برای خواندن آن را داشته باشید.

اگر فایل وجود نداشته باشد، فراخوانی `open` با یک پیام خطا (traceback) شکست خواهد خورد و در نتیجه شما به داده‌های درون فایل دسترسی نخواهید داشت.

```
>>> fhand = open('stuff.txt')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'stuff.txt'
```

بعداً از ساختار try و except استفاده خواهیم کرد تا زمانی که سعی می‌کنیم فایلی را باز کنیم که وجود ندارد، به شکل بهتری با این وضعیت برخورد کنیم.

## ۷,۳ فایل‌های متنی و خطوط

یک فایل متنی را می‌توان به‌صورت دنباله‌ای از خطوط در نظر گرفت، درست مانند اینکه یک رشته (string) در پایتون را می‌توان به‌صورت دنباله‌ای از کاراکترها در نظر گرفت. برای مثال، این نمونه‌ای از یک فایل متنی است که فعالیت‌های ایمیل افراد مختلف در یک تیم توسعه پروژه متن‌باز را ثبت کرده است:

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
Return-Path: <postmaster@collab.sakaiproject.org>
Date: Sat, 5 Jan 2008 09:12:18 -0500
To: source@collab.sakaiproject.org
From: stephen.marquard@uct.ac.za
Subject: [sakai] svn commit: r39772 - content/branches/
Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
...
```

کل فایل تعاملات ایمیل از طریق لینک زیر در دسترس است.

www.py4e.com/code3/mbox.txt

و نسخه کوتاه‌شده‌ای از فایل از طریق این لینک در دسترس است.

www.py4e.com/code3/mbox-short.txt

این فایل‌ها در قالب استانداردی برای فایل‌هایی هستند که شامل چندین پیام ایمیل می‌باشند. خطوطی که با "From" شروع می‌شوند، پیام‌ها را از یکدیگر جدا می‌کنند و خطوطی که با "From:" آغاز می‌شوند، بخشی از محتوای پیام‌ها هستند. برای اطلاعات بیشتر درباره ی قالب mbox می‌توانید به لینک زیر مراجعه کنید:

https://en.wikipedia.org/wiki/Mbox

برای تقسیم فایل به خطوط، کاراکتر خاصی وجود دارد که نمایانگر "پایان خط" است و به آن **کاراکتر خط جدید** (newline character) گفته می‌شود.

در پایتون، این کاراکتر را با `\n` در ثابت‌های رشته‌ای نمایش می‌دهیم. با اینکه این نمایش شبیه دو کاراکتر است، در حقیقت یک کاراکتر منفرد می‌باشد. زمانی که متغیری مانند `"stuff"` را در مفسر پایتون وارد می‌کنیم، `\n` درون رشته به ما نمایش داده می‌شود، اما زمانی که از دستور `print` برای نمایش رشته استفاده می‌کنیم، این کاراکتر باعث می‌شود رشته به دو خط شکسته شود و ما دو خط مجزا مشاهده کنیم.

```
>>> stuff = 'Hello\nWorld!'
>>> stuff
'Hello\nWorld!'
>>> print(stuff)
```

```
Hello
World!
>>> stuff = 'X\nY'
>>> print(stuff)
X
Y
>>> len(stuff)
3
```

شما همچنین می‌توانید ببینید که طول رشته `X\nY` سه کاراکتر است، زیرا کاراکتر خط جدید (`\n`) یک کاراکتر واحد است.

بنابراین زمانی که به خطوط یک فایل نگاه می‌کنیم، باید تصور کنیم که یک کاراکتر ویژه نامرئی به نام خط جدید (newline) در انتهای هر خط وجود دارد که نشان‌دهنده پایان خط است. کاراکتر خط جدید، کاراکترهای فایل را به خطوط تقسیم می‌کند.

## ۷,۴ خواندن فایل‌ها

در حالی که شیء فایل (file handle) داده‌های فایل را در خود نگه نمی‌دارد، بسیار آسان است که یک حلقه `for` بسازیم تا از طریق خطوط فایل بخوانیم و هر خط را شمارش کنیم:

```
fhand = open('mbox-short.txt')
count = 0
for line in fhand:
    count = count + 1
print('Line Count:', count)

# Code: https://www.py4e.com/code3/open.py
```

ما می‌توانیم از هندل فایل به عنوان دنباله‌ای در حلقه `for` خود استفاده کنیم. حلقه `for` به سادگی تعداد خطوط فایل را شمرده و آن‌ها را چاپ می‌کند. ترجمه تقریبی حلقه `for` به زبان انگلیسی این است: "برای هر خط در فایل که توسط هندل فایل نمایندگی می‌شود، یک واحد به متغیر شمارش اضافه کن".

دلیل اینکه تابع `open` تمام فایل را نمی‌خواند این است که ممکن است فایل بسیار بزرگ باشد و داده‌های زیادی را در بر داشته باشد. دستور `open` زمان مشابهی را برای باز کردن فایل صرف می‌کند، فارغ از اندازه فایل. در واقع، حلقه `for` باعث می‌شود داده‌ها از فایل خوانده شوند.

زمانی که فایل به این شیوه با استفاده از حلقه `for` خوانده می‌شود، پایتون مسئول تقسیم داده‌ها در فایل به خطوط جداگانه با استفاده از کاراکتر خط جدید (`\n`) است. پایتون هر خط را از طریق خط جدید می‌خواند و این کاراکتر خط جدید را به عنوان آخرین کاراکتر در متغیر `line` در هر بار تکرار حلقه `for` وارد می‌کند.

چون حلقه for داده‌ها را یک خط در هر بار می‌خواند، این روش به طور کارآمد می‌تواند خطوط فایل‌های بسیار بزرگ را بخواند و شمارش کند، بدون اینکه از حافظه اصلی برای ذخیره داده‌ها استفاده زیادی کند. برنامه فوق می‌تواند خطوط هر فایل را با هر اندازه‌ای شمارش کند و از حافظه بسیار کمی استفاده کند، زیرا هر خط خوانده شده، شمارش شده و سپس دور انداخته می‌شود.

اگر بدانید که فایل نسبتاً کوچک است و با اندازه حافظه اصلی شما تطابق دارد، می‌توانید کل فایل را با استفاده از متد read روی هندل فایل به یک رشته واحد بخوانید.

```
>>> fhand = open('mbox-short.txt')
>>> inp = fhand.read()
>>> print(len(inp))
94626
>>> print(inp[:20])
From stephen.marquar
```

در این مثال، محتوای کامل (تمام ۹۴،۶۲۶ کاراکتر) فایل mbox-short.txt مستقیماً در متغیر inp خوانده می‌شود. ما از برش رشته‌ای (string slicing) برای چاپ ۲۰ کاراکتر اول داده‌های رشته‌ای که در inp ذخیره شده است، استفاده می‌کنیم.

زمانی که فایل به این روش خوانده می‌شود، تمام کاراکترها از جمله تمام خطوط و کاراکترهای خط جدید به یک رشته بزرگ در متغیر inp تبدیل می شوند. این روش را بهتر ا ست که در یک متغیر ذخیره کنید، زیرا هر بار که تابع read فراخوانی می شود، منبع به پایان می‌رسد و دیگر نمی‌توان داده‌ها را از فایل خواند.

```
>>> fhand = open('mbox-short.txt')
>>> print(len(fhand.read()))
94626
>>> print(len(fhand.read()))
0
```

به یاد داشته باشید که این نوع استفاده از تابع open فقط زمانی مناسب است که داده‌های فایل به‌راحتی در حافظه اصلی (RAM) کامپیوتر شما جا شوند. اگر فایل خیلی بزرگ باشد و در حافظه نگنجد، باید برنامه‌تان را طوری بنویسید که فایل را به‌صورت قطعه‌ای (chunk-by-chunk) با استفاده از حلقه‌های for یا while بخواند.

## ۷،۵ جست‌وجو در فایل

در بسیاری از مواقع که می‌خواهید در داده‌های یک فایل جست‌وجو کنید، الگوی رایجی وجود دارد: خواندن خط به خط فایل و نادیده گرفتن اکثر خطوط، فقط پردازش آن‌هایی که شرایط خاصی را دارند. می‌توانیم الگوی خواندن فایل را با متدهای رشته‌ای ترکیب کنیم تا مکانیزم‌های جست‌وجوی ساده‌ای بسازیم.

برای مثال اگر بخواهیم فقط خطوطی را چاپ کنیم که با پیشوند "From:" شروع می‌شوند، می‌توانیم از متد startswith استفاده کنیم:

```
fhand = open('mbox-short.txt')
for line in fhand:
    if line.startswith('From:'):
        print(line)

# Code: https://www.py4e.com/code3/search1.py
```

زمانی که برنامه اجرا می شود خروجی زیر را دریافت خواهیم کرد:

```
From: stephen.marquard@uct.ac.za

From: louis@media.berkeley.edu

From: zqian@umich.edu

From: rjlowe@iupui.edu
...
```

خروجی عالی به نظر می‌رسد، زیرا تنها خطوطی را می‌بینیم که با "From:" شروع می‌شوند. اما چرا خطوط خالی اضافی مشاهده می‌کنیم؟ این به خاطر نویسه نامرئی پایان خط (newline) است. هر خط با یک newline به پایان می‌رسد، بنابراین دستور print، رشته‌ی موجود در متغیر line را که شامل newline است چاپ می‌کند و سپس خود print یک newline دیگر اضافه می‌کند، که باعث ایجاد فاصله دوبرابری بین خطوط می‌شود.

می‌توانیم از برش رشته برای چاپ همه‌چیز به‌جز آخرین نویسه استفاده کنیم، اما روش ساده‌تر استفاده از متد rstrip است که فاصله‌های سفید را از سمت راست رشته حذف می‌کند، مانند زیر:

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if line.startswith('From:'):
        print(line)

# Code: https://www.py4e.com/code3/search2.py
```

When this program runs, we get the following output:

```
From: stephen.marquard@uct.ac.za
From: louis@media.berkeley.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
From: cwen@iupui.edu
...
```

همان‌طور که برنامه‌های پردازش فایل شما پیچیده‌تر می‌شوند، ممکن است بخواهید حلقه‌های جستجوی خود را با استفاده از continue سازمان‌دهی کنید. ایده‌ی اصلی در حلقه‌ی جستجو این است که به دنبال خطوط "جالب" می‌گردید و عملاً از خطوط "غیرجالب" صرف‌نظر می‌کنید. سپس، زمانی که یک خط جالب پیدا کردید، کاری با آن انجام می‌دهید.

ما می‌توانیم ساختار حلقه را به شکلی تنظیم کنیم که الگوی پرش از خطوط غیرجالب را به صورت زیر دنبال کند:

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    # Skip 'uninteresting lines'
    if not line.startswith('From:'):
        continue
    # Process our 'interesting' line
    print(line)

# Code: https://www.py4e.com/code3/search3.py
```

خروجی این برنامه همان است. به زبان ساده، خطوطی که با "From:" شروع نمی‌شوند، خطوطی هستند که برای ما جالب نیستند، و با استفاده از دستور continue آن‌ها را رد می‌کنیم. اما برای خطوط "جالب" (یعنی آن‌هایی که با "From:" شروع می‌شوند)، پردازش را انجام می‌دهیم.

ما می‌توانیم از متد find در رشته‌ها برای شبیه‌سازی جستجوی یک ویرایشگر متنی استفاده کنیم، به‌طوری که خطوطی را پیدا کنیم که رشته‌ی جستجو در هر جای آن‌ها ظاهر شده باشد. از آن‌جایی که find به دنبال وقوع یک رشته درون رشته‌ی دیگر می‌گردد و موقعیت آن را بازمی‌گرداند (یا اگر رشته پیدا نشود، مقدار ۱- برمی‌گرداند)، می‌توانیم حلقه‌ی زیر را بنویسیم تا خطوطی را نمایش دهد که شامل رشته‌ی "@uct.ac.za" هستند (یعنی از دانشگاه کیپ‌تاون در آفریقای جنوبی آمده‌اند):

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if line.find('@uct.ac.za') == -1: continue
    print(line)

# Code: https://www.py4e.com/code3/search4.py
```

که خروجی زیر را تولید می کند:

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
X-Authentication-Warning: set sender to stephen.marquard@uct.ac.za using -f
From: stephen.marquard@uct.ac.za
Author: stephen.marquard@uct.ac.za
From david.horwitz@uct.ac.za Fri Jan  4 07:02:32 2008
X-Authentication-Warning: set sender to david.horwitz@uct.ac.za using -f
From: david.horwitz@uct.ac.za
```

```
Author: david.horwitz@uct.ac.za
...
```

در اینجا ما از شکل خلاصه‌شده‌ی دستور `if` استفاده می‌کنیم، جایی که دستور `continue` در همان خط قرار دارد. این شکل خلاصه دقیقاً همان عملکردی را دارد که اگر `continue` در خط بعدی و با تو رفتگی نوشته شده بود.

## ۷,۶ امکان انتخاب نام فایل توسط کاربر

واقعاً نمی‌خواهیم هر بار که می‌خواهیم یک فایل متفاوت را پردازش کنیم، کد پایتون را ویرایش کنیم. بهتر است برنامه ما به گونه‌ای باشد که هر بار که اجرا می‌شود، از کاربر بخواهد نام فایل را وارد کند تا بدون نیاز به تغییر کد پایتون، بتوانند از برنامه روی فایل‌های مختلف استفاده کنند.

این کار با استفاده از تابع `input` برای خواندن نام فایل از کاربر بسیار ساده انجام می‌شود، به این صورت:

```
fname = input('Enter the file name: ')
fhand = open(fname)
count = 0
for line in fhand:
    if line.startswith('Subject:'):
        count = count + 1
print('There were', count, 'subject lines in', fname)

# Code: https://www.py4e.com/code3/search6.py
```

ما نام فایل را از کاربر می‌خوانیم و آن را در متغیری به نام `fname` قرار می‌دهیم و سپس آن فایل را باز می‌کنیم. با این کار، می‌توانیم برنامه را چندین بار روی فایل‌های مختلف اجرا کنیم.

```
python search6.py
Enter the file name: mbox.txt
There were 1797 subject lines in mbox.txt

python search6.py
Enter the file name: mbox-short.txt
There were 27 subject lines in mbox-short.txt
```

پیش از آن‌که به بخش بعدی نگاه کنید، به برنامه بالا دقت کنید و از خود بپرسید: «چه چیزی ممکن است در اینجا اشتباه پیش برود؟» یا «کاربر خوش‌نیت ما ممکن است چه کاری انجام دهد که باعث شود برنامه کوچک و مرتب ما با یک خطای غیرمنتظره (traceback) از کار بیفتد و ما را در چشم کاربران، حرفه‌ای جلوه ندهد؟»

## ۷,۷ استفاده از try ، except و open

اگر کاربر ما چیزی وارد کند که نام یک فایل نباشد، چه اتفاقی می‌افتد؟ در این صورت، برنامه تلاش می‌کند فایل نادرست یا غیر موجود را باز کند، و چون موفق نمی‌شود، یک خطای traceback نمایش می‌دهد و به طور ناخوشایند متوقف می‌شود. اینجاست که باید از ساختار try و except استفاده کنیم تا برنامه‌مان به‌جای سقوط، با ظرافت خطا را مدیریت کرده و پیامی مناسب به کاربر بدهد.

```
python search6.py
Enter the file name: missing.txt
Traceback (most recent call last):
  File "search6.py", line 2, in <module>
    fhand = open(fname)
FileNotFoundError: [Errno 2] No such file or directory: 'missing.txt'

python search6.py
Enter the file name: na na boo boo
Traceback (most recent call last):
  File "search6.py", line 2, in <module>
    fhand = open(fname)
FileNotFoundError: [Errno 2] No such file or directory: 'na na boo boo'
```

کاربران هر کاری که می‌توانند انجام خواهند داد تا برنامه‌های شما را خراب کنند، چه به اشتباه و چه با نیت مخرب. در حقیقت، یک قسمت مهم از هر تیم توسعه نرم‌افزار، شخص یا گروهی به نام تضمین کیفیت (یا به اختصارQA ) است که وظیفه‌شان انجام دادن عجیب‌ترین کارها برای خراب کردن نرم‌افزاری است که برنامه‌نویس ساخته است.

تیم QA مسئول یافتن نقص‌ها در برنامه‌ها قبل از تحویل آن به کاربران نهایی است، کسانی که ممکن است نرم‌افزار را خریداری کنند یا حقوق برنامه‌نویس را برای نوشتن نرم‌افزار پرداخت کنند. بنابراین تیم QA بهترین دوست برنامه‌نویس است.

حالا که نقص برنامه را مشاهده کردیم، می‌توانیم به طور زیبا آن را با استفاده از ساختار try/except اصلاح کنیم. ما باید فرض کنیم که فراخوانی open ممکن است شکست بخورد و کد بازیابی را هنگام شکست open به شکل زیر اضافه کنیم:

```
fname = input('Enter the file name: ')
try:
    fhand = open(fname)
except:
    print('File cannot be opened:', fname)
    exit()
count = 0
for line in fhand:
    if line.startswith('Subject:'):
        count = count + 1
print('There were', count, 'subject lines in', fname)

# Code: https://www.py4e.com/code3/search7.py
```

تابع exit برنامه را خاتمه می‌دهد. این یک تابع است که آن را فراخوانی می‌کنیم و دیگر باز نمی‌گردد. حالا وقتی که کاربر (یا تیم QA) اشتباهات یا نام فایل‌های نادرست وارد می‌کند، ما این اشتباهات را "دریافت" کرده و به طور مؤدبانه از آنها بازیابی می‌کنیم:

```
python search7.py
Enter the file name: mbox.txt
There were 1797 subject lines in mbox.txt

python search7.py
Enter the file name: na na boo boo
File cannot be opened: na na boo boo
```

محافظت از دستور `open` نمونه‌ای عالی از استفاده صحیح از `try` و `except` در یک برنامه پایتون است. ما از واژه "Pythonic" زمانی استفاده می‌کنیم که کاری را به شیوه "پایتونی" انجام می‌دهیم. ممکن است بگوییم که مثال بالا، روش پایتونی برای باز کردن یک فایل است.

وقتی که شما در پایتون ماهرتر می‌شوید، می‌توانید در مکالمات با سایر برنامه‌نویسان پایتون وارد بحث شوید تا ببینید کدام یک از دو راه‌حل معادل برای یک مشکل "پایتونی‌تر" است. هدف از "پایتونی‌تر بودن" این است که به این نکته اشاره کنیم که برنامه‌نویسی هم بخشی مهندسی است و هم بخشی هنری. ما همیشه فقط به این فکر نمی‌کنیم که چیزی را کارا بسازیم، بلکه می‌خواهیم راه‌حل‌مان زیبا باشد و توسط همکاران‌مان به عنوان یک راه‌حل زیبا درک شود.

## ۷,۸ نوشتن در فایل ها

برای نوشتن در یک فایل، باید آن را با حالت "w" به عنوان پارامتر دوم باز کنید.

```
>>> fout = open('output.txt', 'w')
>>> print(fout)
<_io.TextIOWrapper name='output.txt' mode='w' encoding='cp1252'>
```

اگر فایل از قبل وجود داشته باشد، باز کردن آن در حالت نوشتن (write) داده‌های قدیمی را پاک می‌کند و از نو شروع می‌کند، پس مراقب باشید! اگر فایل وجود نداشته باشد، یک فایل جدید ایجاد می‌شود.

متد write از شیء handle فایل داده‌ها را در فایل قرار می‌دهد و تعداد کاراکترهای نوشته شده را برمی‌گرداند. حالت پیش‌فرض نوشتن، حالت متنی است که برای نوشتن (و خواندن) رشته‌ها استفاده می‌شود.

```
>>> line1 = "This here's the wattle,\n"
>>> fout.write(line1)
24
```

دوباره، شیء فایل محل فعلی را ردیابی می‌کند، بنابراین اگر دوباره از متد write استفاده کنید، داده جدید را به انتهای فایل اضافه می‌کند.

ما باید مطمئن شویم که هنگام نوشتن به فایل، خطوط به درستی مدیریت شوند و کاراکتر newline را به طور صریح وارد کنیم تا پایان خط را مشخص کنیم. دستور print به طور خودکار یک newline به انتهای داده اضافه می‌کند، اما متد write این کار را به صورت خودکار انجام نمی‌دهد.

```
>>> line2 = 'the emblem of our land.\n'
>>> fout.write(line2)
24
```

وقتی که نوشتن تمام شد، باید فایل را ببندید تا مطمئن شوید آخرین قسمت داده‌ها به‌طور فیزیکی به دیسک نوشته می‌شود و در صورتی که برق قطع شود، داده‌ها از دست نمی‌روند.

```
>>> fout.close()
```

ما می‌توانیم فایل‌هایی که برای خواندن باز می‌کنیم را نیز ببندیم، اما اگر فقط چند فایل باز کرده باشیم، می‌توانیم کمی بی‌دقت باشیم زیرا پایتون اطمینان حاصل می‌کند که تمام فایل‌های باز شده زمانی که برنامه تمام می‌شود، بسته می‌شوند. اما وقتی فایل‌ها را می‌نویسیم، باید به‌طور صریح فایل‌ها را ببندیم تا هیچ چیزی را به شانس واگذار نکنیم .

## ۷,۹ اشکال‌زدایی

وقتی که در حال خواندن و نوشتن فایل‌ها هستید، ممکن است با مشکلاتی در ارتباط با فضای خالی (whitespace) مواجه شوید. این خطاها می‌توانند سخت برای اشکال‌زدایی باشند زیرا فضاها، تب‌ها و خطوط جدید به طور معمول قابل مشاهده نیستند.

```
>>> s = '1 2\t 3\n 4'
>>> print(s)
1 2  3
 4
```

تابع داخلی `repr` می‌تواند کمک‌کننده باشد. این تابع هر شیء را به عنوان ورودی می‌گیرد و یک نمایش رشته‌ای از آن شیء باز می‌گرداند. برای رشته‌ها، این تابع کاراکترهای فضای خالی را با دنباله‌های بک‌اسلش (backslash) نمایش می‌دهد:

```
>>> print(repr(s))
'1 2\t 3\n 4'
```

این موضوع می‌تواند برای اشکال‌زدایی مفید باشد.

یک مشکل دیگر که ممکن است با آن مواجه شوید این است که سیستم‌های مختلف از کاراکترهای متفاوتی برای نشان دادن پایان یک خط استفاده می‌کنند. برخی از سیستم‌ها از کاراکتر خط جدید (`\n`) استفاده می‌کنند. برخی دیگر از کاراکتر بازگشت خط (`\r`) استفاده می‌کنند. برخی نیز از هر دو استفاده می‌کنند. اگر فایل‌ها را بین سیستم‌های مختلف منتقل کنید، این ناسازگاری‌ها ممکن است باعث ایجاد مشکلاتی شوند.

برای بیشتر سیستم‌ها، برنامه‌هایی برای تبدیل فرمت‌ها از یکدیگر وجود دارد. شما می‌توانید این برنامه‌ها را پیدا کنید (و بیشتر در مورد این مسئله بخوانید) در https://www.wikipedia.org/wiki/Newline یا البته، می‌توانید خودتان یکی از این برنامه‌ها را بنویسید.

## ۷,۱۰ واژه‌نامه

**catch** : جلوگیری از خاتمه برنامه به دلیل بروز استثنا با استفاده از دستورات `try` و `except`.

**newline**: یک کاراکتر خاص که در فایل‌ها و رشته‌ها برای نشان دادن پایان یک خط استفاده می‌شود.

**Pythonic**: یک تکنیک که به‌طور زیبا و مناسب در پایتون عمل می‌کند. به عنوان مثال، "استفاده از `try` و `except` روش Pythonic برای بازیابی از فایل‌های گم شده است".

**Quality Assurance**: فرد یا تیمی که بر کیفیت کلی محصول نرم‌افزاری نظارت دارد. تیم QA معمولاً در تست محصول و شناسایی مشکلات قبل از انتشار آن نقش دارد.

**text file**: یک رشته از کاراکترها که در ذخیره‌سازی دائمی مانند هارد دیسک ذخیره شده است.

## ۷,۱۱ تمرین ها

**تمرین ۱**: یک برنامه بنویسید که یک فایل را بخواند و محتوای فایل را (خط به خط) همگی را با حروف بزرگ چاپ کند. اجرای برنامه به شکل زیر خواهد بود:

```
python shout.py
Enter a file name: mbox-short.txt
FROM STEPHEN.MARQUARD@UCT.AC.ZA SAT JAN  5 09:14:16 2008
RETURN-PATH: <POSTMASTER@COLLAB.SAKAIPROJECT.ORG>
RECEIVED: FROM MURDER (MAIL.UMICH.EDU [141.211.14.90])
     BY FRANKENSTEIN.MAIL.UMICH.EDU (CYRUS V2.3.8) WITH LMTPA;
     SAT, 05 JAN 2008 09:14:16 -0500
```

می توانید فایل را در آدرس زیر دانلود کنید

www.py4e.com/code3/mbox-short.txt

**تمرین ۲**: یک برنامه بنویسید که نام یک فایل را درخواست کند، سپس فایل را خوانده و به دنبال خطوطی با فرم زیر بگردد:

```
X-DSPAM-Confidence: 0.8475
```

وقتی به خطی برخورد می‌کنید که با **"X-DSPAM-Confidence:"** شروع می‌شود، خط را تجزیه کنید تا عددِ ممیز شناور موجود در آن خط را استخراج کنید. این خطوط را شمارش کنید و سپس مجموع مقادیر «اطمینان اسپم» موجود در این خطوط را محاسبه نمایید. زمانی که به انتهای فایل رسیدید، میانگین مقدار اطمینان اسپم را چاپ کنید.

```
Enter the file name: mbox.txt
Average spam confidence: 0.894128046745

Enter the file name: mbox-short.txt
Average spam confidence: 0.750718518519
```

Test your file on the *mbox.txt* and *mbox-short.txt* files.

**تمرین ۳:** گاهی اوقات وقتی برنامه‌نویسان کسل می‌شوند یا می‌خواهند کمی سرگرمی ایجاد کنند، یک *ایستر اِگ* بی‌ضرر در برنامه خود قرار می‌دهند. برنامه‌ای را که از کاربر نام فایل را می‌پرسد طوری تغییر دهید که وقتی کاربر دقیقاً نام فایل **"na na boo boo"** را وارد می‌کند، یک پیام خنده‌دار چاپ کند. برنامه باید برای تمام فایل‌های دیگر چه وجود داشته باشند و چه وجود نداشته باشند به صورت عادی رفتار کند. در اینجا یک اجرای نمونه از برنامه آورده شده است:

```
python egg.py
Enter the file name: mbox.txt
There were 1797 subject lines in mbox.txt

python egg.py
Enter the file name: missing.tyxt
File cannot be opened: missing.tyxt

python egg.py
Enter the file name: na na boo boo
NA NA BOO BOO TO YOU - You have been punk'd!
```

ما شما را به افزودن ایستر اِگ در برنامه‌هایتان تشویق نمی‌کنیم؛ این فقط یک تمرین است.

# فصل هشتم

# فهرست‌ها(Lists)

## ۸,۱ یک فهرست، یک دنباله است.

مانند رشته‌ها، فهرست نیز یک دنباله از مقادیر است. در یک رشته، مقادیر کاراکتر هستند؛ اما در یک فهرست، می‌توانند از هر نوعی باشند. به مقادیر درون فهرست‌ها عناصر (elements) یا گاهی اقلام (items) گفته می‌شود.

راه‌های مختلفی برای ایجاد یک فهرست جدید وجود دارد؛ ساده‌ترین روش آن است که عناصر را درون کروشه‌ها (یعنی "[" و "]") قرار دهیم:

```
[10, 20, 30, 40]
['crunchy frog', 'ram bladder', 'lark vomit']
```

اولین مثال، فهرستی از چهار عدد صحیح (integer) است. مثال دوم، فهرستی از سه رشته (string) است. عناصر یک فهرست لازم نیست همگی از یک نوع باشند. فهرست زیر شامل یک رشته، یک عدد اعشاری(float) ، یک عدد صحیح، و یک فهرست دیگر است:

```
['spam', 2.0, 5, [10, 20]]
```

فهرستی که درون فهرستی دیگر قرار دارد، یک فهرست تودرتو (nested) نامیده می‌شود.

فهرستی که هیچ عنصری ندارد، «فهرست خالی» نامیده می‌شود؛ شما می‌توانید آن را با استفاده از براکت‌های خالی ([]) ایجاد کنید.

همان‌طور که انتظار دارید، می‌توانید مقادیر فهرست را به متغیرها اختصاص دهید:

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> numbers = [17, 123]
>>> empty = []
>>> print(cheeses, numbers, empty)
['Cheddar', 'Edam', 'Gouda'] [17, 123] []
```

## ۸,۲ فهرست‌ها قابل تغییر هستند(mutable)

نحو (syntax) دسترسی به عناصر یک فهرست همانند دسترسی به کاراکترهای یک رشته است: با استفاده از عملگر براکت. عبارت درون براکت‌ها، شاخص (index) عنصر مورد نظر را مشخص می‌کند. به یاد داشته باشید که شاخص‌ها از ۰ شروع می‌شوند:

```
>>> print(cheeses[0])
Cheddar
```

بر خلاف رشته‌ها، فهرست‌ها قابل تغییر (mutable) هستند، زیرا می‌توان ترتیب آیتم‌ها را در یک فهرست تغییر داد یا مقدار یک آیتم را دوباره تخصیص داد. زمانی که عملگر براکت در سمت چپ یک عبارت انتساب قرار می‌گیرد، عنصری از فهرست را مشخص می‌کند که مقدار جدید به آن اختصاص داده خواهد شد.

```
>>> numbers = [17, 123]
>>> numbers[1] = 5
>>> print(numbers)
[17, 5]
```

عنصر یکم (index 1) از فهرست numbers که پیش‌تر مقدار ۱۲۳ داشت، اکنون **۵** شده است.

می‌توانید یک فهرست را مانند رابطه‌ای میان شاخص‌ها (indices) و عناصر (elements) در نظر بگیرید. این رابطه **نگاشت** (mapping) نامیده می‌شود؛ هر شاخص به یکی از عناصر «نگاشت» می‌شود.

شاخص‌های فهرست همانند شاخص‌های رشته‌ها کار می‌کنند:

- **هر عبارت عدد صحیح** می‌تواند به عنوان شاخص استفاده شود.
- اگر تلاش کنید عنصری را بخوانید یا تغییر دهید که وجود ندارد، با خطای **IndexError** مواجه می‌شوید.
- اگر شاخص مقدار منفی داشته باشد، شمارش از انتهای فهرست به سمت ابتدای فهرست انجام می‌شود.

عملگر in نیز روی فهرست‌ها کار می‌کند.

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in cheeses
True
>>> 'Brie' in cheeses
False
```

## ۸,۳ عبور از یک لیست

رایج‌ترین روش برای عبور از عناصر یک لیست استفاده از حلقه for است. نحو آن همانند نحوی که برای رشته‌ها استفاده می‌شود، می‌باشد:

```
for cheese in cheeses:
    print(cheese)
```

این روش خوب کار می‌کند اگر تنها نیاز به خواندن عناصر لیست داشته باشید. اما اگر بخواهید عناصر را بنویسید یا به‌روزرسانی کنید، به اندیس‌ها نیاز خواهید داشت. یک روش رایج برای این کار ترکیب توابع `range` و `len` است :

```
for i in range(len(numbers)):
    numbers[i] = numbers[i] * 2
```

این حلقه لیست را پیمایش کرده و هر عنصر را به‌روزرسانی می‌کند. تابع `len` تعداد عناصر لیست را برمی‌گرداند. تابع `range` لیستی از اندیس‌ها از ۰ تا n-1 را برمی‌گرداند، که در آن n طول لیست است. هر بار که از حلقه عبور می‌کنیم، متغیر `i` اندیس عنصر بعدی را دریافت می‌کند. دستور انتساب در بدنه از `i` برای خواندن مقدار قدیمی عنصر و اختصاص مقدار جدید استفاده می‌کند.

یک حلقه `for` که بر روی یک لیست خالی پیمایش می‌کند، بدنه آن اجرا نخواهد شد.

```
for x in empty:
    print('This never happens.')
```

اگرچه یک لیست می‌تواند شامل یک لیست دیگر باشد، اما لیست تو در تو همچنان به‌عنوان یک عنصر واحد در نظر گرفته می‌شود. طول این لیست برابر با چهار است :

```
['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```

## ۸,۴ عملیات روی لیست‌ها

عملگر + لیست‌ها را به هم متصل می‌کند:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print(c)
[1, 2, 3, 4, 5, 6]
```

به‌طور مشابه، عملگر * یک لیست را به تعداد معینی تکرار می‌کند :

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

مثال اول چهار بار تکرار می‌شود. مثال دوم لیست را سه بار تکرار می‌کند.

## ۸,۵ عملگر برش (slice)

عملگر برش همچنین روی لیست‌ها عمل می‌کند:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']
>>> t[:4]
['a', 'b', 'c', 'd']
>>> t[3:]
['d', 'e', 'f']
```

اگر اندیس اول را حذف کنید، برش از ابتدای لیست شروع می‌شود. اگر اندیس دوم را حذف کنید، برش تا انتهای لیست ادامه می‌یابد. بنابراین، اگر هر دو اندیس را حذف کنید، برش کپی کاملی از کل لیست خواهد بود .

```
>>> t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

از آنجا که لیست‌ها قابل تغییر هستند، اغلب مفید است که قبل از انجام عملیات‌هایی که لیست‌ها را تغییر می‌دهند یا اصلاح می‌کنند، یک کپی از آن‌ها تهیه کنید. اپراتور برش در سمت چپ یک دستور انتساب می‌تواند چندین عنصر را به‌روزرسانی کند.

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3] = ['x', 'y']
>>> print(t)
['a', 'x', 'y', 'd', 'e', 'f']
```

## ۸,۶ روش‌های لیست

پایتون متدهایی را ارائه می‌دهد که بر روی لیست‌ها عمل می‌کنند. برای مثال، متد append یک عنصر جدید را به انتهای یک لیست اضافه می‌کند:

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> print(t)
['a', 'b', 'c', 'd']
```

متد extend یک لیست را به عنوان آرگومان می‌گیرد و تمام عناصر آن را به انتهای لیست اصلی اضافه می‌کند :

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> print(t1)
['a', 'b', 'c', 'd', 'e']
```

این مثال لیست t2 را بدون تغییر باقی می‌گذارد.
متد sort عناصر لیست را از کم به زیاد مرتب می‌کند:

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> print(t)
['a', 'b', 'c', 'd', 'e']
```

بیشتر متدهای لیست به صورت void هستند؛ یعنی لیست را تغییر می‌دهند و None را باز می‌گردانند. اگر به اشتباه بنویسید t = t.sort()، از نتیجه ناامید خواهید شد.

## ۸,۷ حذف عناصر

روش‌های مختلفی برای حذف عناصر از یک لیست وجود دارد. اگر از اندیس عنصری که می‌خواهید حذف کنید، اطلاع دارید، می‌توانید از متد pop استفاده کنید:

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> print(t)
['a', 'c']
>>> print(x)
b
```

متد pop لیست را تغییر می‌دهد و عنصری که حذف شده را باز می‌گرداند. اگر اندیسی را مشخص نکنید، آخرین عنصر لیست حذف و باز می‌گردد.

اگر نیازی به مقداری که حذف شده ندارید، می‌توانید از دستور del استفاده کنید:

```
>>> t = ['a', 'b', 'c']
>>> del t[1]
>>> print(t)
['a', 'c']
```

اگر شما عنصری که می‌خواهید حذف کنید را می‌دانید (اما نه اندیس آن را)، می‌توانید از متد remove استفاده کنید :

```
>>> t = ['a', 'b', 'c']
>>> t.remove('b')
>>> print(t)
['a', 'c']
```

مقدار برگشتی از متد remove برابر با None است.

برای حذف بیش از یک عنصر، می‌توانید از دستور del با اندیس برش استفاده کنید:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del t[1:5]
>>> print(t)
['a', 'f']
```

همانطور که معمول است، برش تمام عناصر تا اندیس دوم را انتخاب می‌کند، اما خود اندیس دوم را شامل نمی‌شود.

## ۸,۸ فهرست‌ها و توابع

تعداد زیادی توابع داخلی وجود دارند که می‌توان از آن‌ها برای فهرست‌ها استفاده کرد و به شما این امکان را می‌دهند که سریعاً از طریق یک لیست جستجو کنید بدون اینکه نیاز به نوشتن حلقه‌های خودتان داشته باشید:

```
>>> nums = [3, 41, 12, 9, 74, 15]
>>> print(len(nums))
6
>>> print(max(nums))
74
>>> print(min(nums))
3
>>> print(sum(nums))
154
>>> print(sum(nums)/len(nums))
25
```

تابع sum() فقط زمانی کار می‌کنند که عناصر لیست اعداد باشند. توابع دیگر (مانند max() ، len() و غیره) با لیست‌هایی از رشته‌ها و سایر انواع داده‌ای که قابل مقایسه هستند، کار می‌کنند.

ما می‌توانیم برنامه قبلی که میانگین یک لیست از اعداد وارد شده توسط کاربر را محاسبه می‌کند، بازنویسی کنیم و از لیست استفاده کنیم.

ابتدا برنامه‌ای که میانگین را بدون استفاده از لیست محاسبه می‌کند را مشاهده می کنید:

```
total = 0
count = 0
while (True):
    inp = input('Enter a number: ')
    if inp == 'done': break
    value = float(inp)
    total = total + value
    count = count + 1
```

```
average = total / count
print('Average:', average)

# Code: https://www.py4e.com/code3/avenum.py
```

در این برنامه، ما از متغیرهای `count` و `total` برای نگهداری تعداد و مجموع اعداد وارد شده توسط کاربر استفاده می‌کنیم، در حالی که به طور مکرر از کاربر خواسته می‌شود یک عدد وارد کند .ما می‌توانیم به سادگی هر عدد را همانطور که کاربر وارد می‌کند به یاد بیاوریم و از توابع داخلی برای محاسبه مجموع و تعداد در انتها استفاده کنیم.

```
numlist = list()
while (True):
    inp = input('Enter a number: ')
    if inp == 'done': break
    value = float(inp)
    numlist.append(value)

average = sum(numlist) / len(numlist)
print('Average:', average)

# Code: https://www.py4e.com/code3/avelist.py
```

ما قبل از شروع حلقه یک لیست خالی می‌سازیم، سپس هر بار که یک عدد داریم، آن را به لیست اضافه می‌کنیم. در انتهای برنامه، به سادگی مجموع اعداد موجود در لیست را محاسبه کرده و آن را بر تعداد اعداد موجود در لیست تقسیم می‌کنیم تا میانگین به دست آید.

## ۸,۹ لیست‌ها و رشته‌ها

یک رشته یک توالی از کاراکترها است و یک لیست یک توالی از مقادیر است، اما یک لیست از کاراکترها با یک رشته یکسان نیست. برای تبدیل از رشته به یک لیست از کاراکترها، می‌توانید از تابع `list` استفاده کنید.

```
>>> s = 'spam'
>>> t = list(s)
>>> print(t)
['s', 'p', 'a', 'm']
```

چون `list` نام یک تابع ساخته‌شده است، باید از استفاده آن به عنوان نام متغیر خودداری کنید. من همچنین از حرف "l" اجتناب می‌کنم چون خیلی شبیه به عدد "۱" به نظر می‌رسد. به همین دلیل من از "t" استفاده می‌کنم.

تابع `list` یک رشته را به حروف جداگانه تقسیم می‌کند. اگر بخواهید یک رشته را به کلمات تقسیم کنید، می‌توانید از متد `split` استفاده کنید .

```
>>> s = 'pining for the fjords'
>>> t = s.split()
>>> print(t)
['pining', 'for', 'the', 'fjords']
>>> print(t[2])
the
```

زمانی که از split برای تقسیم کردن یک رشته به یک لیست از کلمات استفاده کردید، می‌توانید از عملگر ایندکس (براکت مربع) برای مشاهده یک کلمه خاص در لیست استفاده کنید. می‌توانید split را با یک آرگومان اختیاری به نام delimiter فراخوانی کنید که مشخص می‌کند کدام کاراکترها به عنوان مرزهای کلمه استفاده شوند. مثال زیر از خط تیره به عنوان delimiter استفاده می‌کند :

```
>>> s = 'spam-spam-spam'
>>> delimiter = '-'
>>> s.split(delimiter)
['spam', 'spam', 'spam']
```

join معکوس split است. این متد یک لیست از رشته‌ها را می‌گیرد و عناصر آن را به هم متصل می‌کند .join یک متد از نوع رشته است، بنابراین شما باید آن را روی delimiter فراخوانی کنید و لیست را به عنوان پارامتر به آن بدهید:

```
>>> t = ['pining', 'for', 'the', 'fjords']
>>> delimiter = ' '
>>> delimiter.join(t)
'pining for the fjords'
```

در این حالت، delimiter یک کاراکتر فاصله است، بنابراین join یک فاصله بین کلمات قرار می‌دهد. برای ترکیب رشته‌ها بدون فاصله، می‌توانید از رشته خالی " " به عنوان delimiter استفاده کنید.

## ۸,۱۰ تحلیل خطوط

معمولاً وقتی که داریم فایلی را می‌خوانیم، می‌خواهیم کاری به جز چاپ کل خط انجام دهیم. اغلب می‌خواهیم "خطوط جالب" را پیدا کرده و سپس آن خط را تجزیه کنیم تا بخش جالبی از آن را پیدا کنیم. حالا اگر می‌خواستیم روز هفته را از خطوطی که با "From"شروع می‌شوند چاپ کنیم، چه می‌شد؟

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
```

متد split زمانی که با چنین مشکلی مواجه می‌شویم بسیار مؤثر است. می‌توانیم یک برنامه کوچک بنویسیم که به دنبال خطوطی بگردد که با "From" شروع می‌شوند، آن خطوط را تقسیم کند و سپس کلمه سوم در خط را چاپ کند:

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
```

```
    if not line.startswith('From '): continue
    words = line.split()
    print(words[2])

# Code: https://www.py4e.com/code3/search5.py
```

این برنامه خروجی زیر را تولید خواهد کرد

```
Sat
Fri
Fri
Fri
...
```

بعداً، ما تکنیک‌های پیچیده‌تری برای انتخاب خطوطی که باید روی آن‌ها کار کنیم و نحوه تجزیه آن خطوط برای یافتن دقیق‌ترین اطلاعاتی که به دنبال آن هستیم، خواهیم آموخت.

## ۸,۱۱ اشیاء و مقادیر

اگر این دستورات انتساب را اجرا کنیم:

```
a = 'banana'
b = 'banana'
```

ما می‌دانیم که a و b هر دو به یک رشته ارجاع دارند، اما نمی‌دانیم آیا آن‌ها به یک رشته یکسان ارجاع می‌دهند یا خیر. دو وضعیت ممکن وجود دارد: متغیرها و اشیاء در یک حالت، a و b به دو شی مختلف ارجاع می‌دهند که مقدار یکسانی دارند. در حالت دوم، آن‌ها به همان شی ارجاع می‌دهند.

برای بررسی اینکه آیا دو متغیر به همان شی ارجاع می‌دهند، می‌توانید از عملگر `is` استفاده کنید.

```
>>> a = 'banana'
>>> b = 'banana'
>>> a is b
True
```

در این مثال، پایتون تنها یک شی رشته‌ای ایجاد کرده است و هر دو متغیر a و b به آن ارجاع می‌دهند.
اما وقتی که دو لیست ایجاد می‌کنید، دو شی مختلف به وجود می‌آید:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
```

در این حالت، ما می‌گوییم که دو لیست معادل هستند، زیرا عناصر مشابهی دارند، اما یکسان نیستند زیرا آن‌ها شیء یکسانی نیستند. اگر دو شیء یکسان باشند، آن‌ها همچنین معادل هستند، اما اگر معادل باشند، لزوماً یکسان نیستند.

تا اینجا، ما از واژه‌های "شیء" و "مقدار" به‌طور معادل استفاده کرده‌ایم، اما دقیق‌تر است که بگوییم یک شیء دارای مقدار است. اگر شما a = [1,2,3] را اجرا کنید، a به یک شیء لیست ارجاع می‌دهد که مقدار آن یک دنباله خاص از عناصر است. اگر لیست دیگری همان عناصر را داشته باشد، ما می‌گوییم که آن مقدار مشابهی دارد.

## ۸,۱۲ استفاده از نام مستعار

اگر a به یک شیء ارجاع دهد و شما b = a را نسبت دهید، آنگاه هر دو متغیر به همان شیء ارجاع می‌دهند:

```
>>> a = [1, 2, 3]
>>> b = a
>>> b is a
True
```

اتصال یک متغیر به یک شیء را ارجاع (reference) می‌نامند. در این مثال، دو ارجاع به همان شیء وجود دارد. شیءای که بیش از یک ارجاع دارد، نام‌های متعددی دارد، بنابراین می‌گوییم شیء نام مستعار دارد یا آلیاس (aliased) است.

اگر شیء نام مستعار گرفته (آلیاس) شده قابل تغییر (mutable) باشد، تغییراتی که با یک آلیاس انجام می‌شود، بر روی دیگری هم تأثیر می‌گذارد:

```
>>> b[0] = 17
>>> print(a)
[17, 2, 3]
```

اگرچه این رفتار می‌تواند مفید باشد، اما خطاپذیر است. به طور کلی، بهتر است هنگام کار با اشیاء قابل تغییر (mutable) از ایجاد آلیاس (aliasing) اجتناب کنید.

برای اشیاء غیرقابل تغییر (immutable) مانند رشته‌ها، آلیاسینگ مشکل چندانی ایجاد نمی‌کند. در این مثال:

```
a = 'banana'
b = 'banana'
```

تقریباً هیچ‌گاه فرقی نمی‌کند که آیا a و b به همان رشته اشاره می‌کنند یا نه .

## ۸,۱۳ آرگومان‌های لیست

زمانی که یک لیست را به یک تابع می‌فرستید، تابع یک مرجع به آن لیست دریافت می‌کند. اگر تابع یک پارامتر لیست را تغییر دهد، فراخوانی‌کننده تغییرات را می‌بیند. به عنوان مثال، delete_head اولین عنصر را از یک لیست حذف می‌کند:

```
def delete_head(t):
    del t[0]
```

Here's how it is used:

```
>>> letters = ['a', 'b', 'c']
>>> delete_head(letters)
>>> print(letters)
['b', 'c']
```

پارامتر t و متغیر letters ارجاعات به همان شیء هستند.

مهم است که تفاوت بین عملیات‌هایی که لیست‌ها را تغییر می‌دهند و عملیات‌هایی که لیست‌های جدید ایجاد می‌کنند را تشخیص دهید. به عنوان مثال، متد append یک لیست را تغییر می‌دهد، اما عملگر + یک لیست جدید ایجاد می‌کند :

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> print(t1)
[1, 2, 3]
>>> print(t2)
None

>>> t3 = t1 + [3]
>>> print(t3)
[1, 2, 3]
>>> t1 is t3
False
```

این تفاوت زمانی مهم است که شما توابعی می‌نویسید که قرار است لیست‌ها را اصلاح کنند. به عنوان مثال، این تابع سر یک لیست را حذف نمی‌کند:

```
def bad_delete_head(t):
    t = t[1:]               # WRONG!
```

عملگر برش یک لیست جدید ایجاد می‌کند و دستور اختصاص باعث می‌شود که متغیر t به آن ارجاع دهد، اما هیچ‌کدام از این‌ها تاثیری بر لیستی که به عنوان آرگومان منتقل شده است ندارد.

یک گزینه دیگر این است که تابعی بنویسید که یک لیست جدید ایجاد کرده و آن را بازگرداند. به عنوان مثال، تابع tail همه عناصر جز اولین عنصر یک لیست را باز می‌گرداند:

```
def tail(t):
    return t[1:]
```

این تابع لیست اصلی را بدون تغییر باقی می‌گذارد. در اینجا نحوه استفاده از آن آمده است:

```
>>> letters = ['a', 'b', 'c']
>>> rest = tail(letters)
>>> print(rest)
['b', 'c']
```

تمرین ۱: یک تابع به نام chop بنویسید که یک لیست را گرفته و آن را اصلاح کند، به طوری که اولین و آخرین عناصر را حذف کند و None را بازگشت دهد. سپس یک تابع به نام middle بنویسید که یک لیست را گرفته و یک لیست جدید برمی‌گرداند که شامل همه عناصر به جز اولین و آخرین آن‌ها باشد.

## ۸,۱۴ اشکال‌زدایی

استفاده بی‌دقت از لیست‌ها (و دیگر اشیاء قابل تغییر) می‌تواند منجر به ساعات طولانی اشکال‌زدایی شود. در اینجا برخی از مشکلات رایج و روش‌های اجتناب از آن‌ها آمده است:

۱. **فراموش نکنید که بیشتر متدهای لیست، آرگومان را تغییر می‌دهند و None را برمی‌گردانند. این برعکس متدهای رشته‌ای است که یک رشته جدید بازمی‌گردانند و نسخه اصلی را دست‌نخورده می‌گذارند.**

اگر شما به نوشتن کدی شبیه به این برای رشته‌ها عادت کرده‌اید:

```
word = word.strip()
```

It is tempting to write list code like this:

```
t = t.sort()              # WRONG!
```

چون sort مقدار None را برمی‌گرداند، عملیات بعدی که با t انجام می‌دهید احتمالاً با شکست مواجه خواهد شد.

قبل از استفاده از متدها و عملگرهای لیست، بهتر است مستندات آن‌ها را با دقت مطالعه کنید و سپس آن‌ها را در حالت تعاملی تست کنید. متدها و عملگرهایی که لیست‌ها با دیگر توالی‌ها (مانند رشته‌ها) به اشتراک می‌گذارند، در مستندات در آدرس زیر قابل مشاهده هستند:

docs.python.org/library/stdtypes.html#common-sequence-operations

روش‌ها و عملگرهایی که فقط برای دنباله‌های قابل تغییر (mutable sequences) اعمال می‌شوند، در این آدرس مستند شده‌اند:

docs.python.org/library/stdtypes.html#mutable-sequence-types

**۲. یک عبارت یا روش را انتخاب کنید و به آن پایبند باشید.**

بخشی از مشکل لیست‌ها این است که راه‌های زیادی برای انجام کارها وجود دارد. برای مثال، برای حذف یک عنصر از لیست، می‌توانید از متدهای pop، remove، del یا حتی تخصیص برش (slice assignment) استفاده کنید .برای افزودن یک عنصر، می‌توانید از متد append یا عملگر + استفاده کنید. اما فراموش نکنید که این روش‌ها صحیح هستند:

```
t.append(x)
t = t + [x]
And these are wrong:
t.append([x])          # WRONG!
t = t.append(x)        # WRONG!
t + [x]                # WRONG!
t = t + x              # WRONG!
```

هر یک از این مثال‌ها را در حالت تعاملی امتحان کنید تا مطمئن شوید که عملکرد آنها را درک کرده‌اید. توجه داشته باشید که تنها مورد آخر باعث ایجاد خطای زمان اجرا می‌شود؛ سه مورد دیگر قانونی هستند، اما کار اشتباهی انجام می‌دهند.

**۳. برای جلوگیری از آلیاسینگ، نسخه‌های کپی ایجاد کنید.**

اگر می‌خواهید از متدی مانند sort که آرگومان را تغییر می‌دهد استفاده کنید، اما نیاز دارید که لیست اصلی را حفظ کنید، می‌توانید یک نسخه کپی از آن ایجاد کنید.

```
orig = t[:]
t.sort()
```

در این مثال، شما همچنین می‌توانید از تابع داخلی sorted استفاده کنید که یک لیست جدید مرتب‌شده باز می‌گرداند و لیست اصلی را دست‌نخورده می‌گذارد. اما در این حالت باید از استفاده از sorted به عنوان نام متغیر خودداری کنید!

**۴ .فهرست‌ها، split و فایل‌ها**

هنگامی که فایل‌ها را می‌خوانیم و تجزیه (parse) می‌کنیم، احتمال زیادی وجود دارد که با ورودی‌هایی برخورد کنیم که می‌توانند برنامه‌مان را دچار خطا کنند؛ بنابراین ایده‌ی خوبی است که هنگام نوشتن برنامه‌هایی که فایل را خط‌به‌خط می‌خوانند و به دنبال «سوزنی در انبار کاه» هستند، الگوی نگهبان (guardian pattern) را دوباره مرور کنیم.

بیایید دوباره به برنامه‌ای که به دنبال روز هفته در خطوطی است که با From آغاز می‌شوند، نگاهی بیندازیم:

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
```

از آن‌جایی که این خط را به واژه‌ها تقسیم می‌کنیم، می‌توانیم از استفاده از تابع startswith صرف‌نظر کنیم و به‌جای آن، تنها واژه ی اول خط را بررسی کنیم تا تعیین کنیم که آیا به آن خط علاقه‌مند هستیم یا نه. می‌توانیم از دستور continue برای رد کردن خطوطی که واژه ی اول آن‌ها «From» نیست استفاده کنیم، به‌صورت زیر:

```
fhand = open('mbox-short.txt')
for line in fhand:
    words = line.split()
    if words[0] != 'From' : continue
    print(words[2])
```

این بسیار ساده‌تر به نظر می‌رسد و حتی نیازی به استفاده از تابع `rstrip` برای حذف نویسه ی پایان خط (newline) در انتهای فایل نداریم. اما آیا این روش بهتر است؟

```
python search8.py
Sat
Traceback (most recent call last):
  File "search8.py", line 5, in <module>
    if words[0] != 'From' : continue
IndexError: list index out of range
```

تا حدی برنامه کار می‌کند و روز هفته از اولین خط (Sat) را می‌بینیم، اما سپس برنامه با یک خطای traceback متوقف می‌شود. چه مشکلی پیش آمد؟ چه داده ی نادرستی باعث شد برنامه ی ما که به‌طور زیبا، هوشمندانه و کاملاً پایتونیک نوشته شده بود، شکست بخورد؟

می‌توانید مدت‌ها به آن خیره شوید و سعی کنید مشکل را کشف کنید یا از کسی کمک بگیرید، اما رویکردی سریع‌تر و هوشمندانه‌تر این است که یک دستور print اضافه کنید. بهترین جا برای اضافه کردن دستورprint ، دقیقاً قبل از خطی است که برنامه در آن شکست خورده و متغیری را که به نظر می‌رسد باعث بروز خطا شده چاپ کنید.

البته این روش ممکن است خطوط زیادی را چاپ کند، اما دست‌کم بلافاصله سرنخی از مشکل به‌دست خواهید آورد. بنابراین، دستور print را برای متغیر `words` درست قبل از خط پنجم اضافه می‌کنیم. حتی یک پیشوند «Debug:» نیز به خط اضافه می‌کنیم تا خروجی معمولی را از خروجی دیباگ متمایز کنیم.

```
for line in fhand:
    words = line.split()
    print('Debug:', words)
    if words[0] != 'From' : continue
    print(words[2])
```

وقتی برنامه را اجرا می‌کنیم، حجم زیادی از خروجی از روی صفحه عبور می‌کند، اما در انتها خروجی دیباگ و پیام خطای traceback را می‌بینیم، بنابراین متوجه می‌شویم چه اتفاقی درست قبل از رخ دادن خطا افتاده است.

```
Debug: ['X-DSPAM-Confidence:', '0.8475']
Debug: ['X-DSPAM-Probability:', '0.0000']
Debug: []
Traceback (most recent call last):
  File "search9.py", line 6, in <module>
```

```
    if words[0] != 'From' : continue
IndexError: list index out of range
```

هر خط دیباگ (debug) در حال چاپ لیستی از کلماتی است که هنگام تقسیم خط به کلمات به دست می‌آوریم. زمانی که برنامه دچار خطا می‌شود، لیست کلمات خالی است .[] :اگر فایل را در یک ویرایشگر متن باز کنیم و در آن نقطه به فایل نگاه کنیم، به شکل زیر خواهد بود:

```
X-DSPAM-Result: Innocent
X-DSPAM-Processed: Sat Jan  5 09:14:16 2008
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000

Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
```

خطا زمانی رخ می‌دهد که برنامه ی ما با یک خط خالی مواجه می‌شود! بدیهی است که در یک خط خالی هیچ کلمه‌ای وجود ندارد. چرا هنگام نوشتن کد به این موضوع فکر نکردیم؟ زمانی که کد تلاش می‌کند اولین کلمه (یعنی words[0]) را بررسی کند تا ببیند آیا با "From" مطابقت دارد یا نه، با خطای "index out of range" ایندکس خارج از محدوده) مواجه می‌شویم.

این دقیقاً موقعیت مناسبی برای افزودن کدی محافظ (guardian code) است تا از بررسی اولین کلمه در صورتی که وجود ندارد، جلوگیری شود. روش‌های زیادی برای محافظت از این کد وجود دارد؛ ما تصمیم می‌گیریم پیش از بررسی اولین کلمه، تعداد کلمات را بررسی کنیم:

```
fhand = open('mbox-short.txt')
count = 0
for line in fhand:
    words = line.split()
    # print('Debug:', words)
    if len(words) == 0 : continue
    if words[0] != 'From' : continue
    print(words[2])
```

ابتدا، ما دستور چاپ اشکال‌زدایی (debug print) را کامنت کردیم به‌جای آن‌که آن را حذف کنیم، تا در صورتی که تغییرات‌مان با شکست مواجه شد، بتوانیم برای اشکال‌زدایی دوباره از آن استفاده کنیم. سپس، یک جمله ی نگهبان (guardian statement) اضافه کردیم که بررسی می‌کند آیا تعداد کلمات صفر است یا نه، و اگر چنین بود، با استفاده از دستور continue به خط بعدی فایل می‌رویم.

می‌توانیم دو دستور continue را به‌گونه‌ای در نظر بگیریم که به ما در پالایش مجموعه ی خطوطی که برای‌مان "جالب" هستند و می‌خواهیم بیشتر آن‌ها را پردازش کنیم، کمک می‌کنند. خطی که هیچ کلمه‌ای ندارد برای ما «بی‌اهمیت» است، پس آن را نادیده می‌گیریم. خطی که اولین کلمه‌اش "From" نیست نیز برای ما بی‌اهمیت است، پس آن را هم نادیده می‌گیریم. برنامه ی تغییر داده‌شده به‌درستی اجرا می‌شود، پس شاید درست باشد. جمله ی نگهبان ما تضمین می‌کند که words[0] هرگز شکست نمی‌خورد، اما شاید این کافی نباشد. وقتی برنامه‌نویسی می‌کنیم، باید همیشه در فکر این باشیم که: «چه چیزی ممکن است اشتباه پیش برود؟»

**تمرین ۲ :**مشخص کنید کدام خط از برنامه ی بالا هنوز به‌درستی محافظت نشده است. سعی کنید فایلی متنی بسازید که باعث شود برنامه از کار بیفتد و سپس برنامه را طوری تغییر دهید که آن خط نیز به‌درستی محافظت شود و مطمئن شوید که برنامه فایل جدید شما را نیز به‌درستی پردازش می‌کند.

**تمرین ۳ :**کد نگهبان موجود در مثال بالا را بازنویسی کنید، به‌جای استفاده از دو دستورif ، از یک عبارت منطقی ترکیبی با عملگر `or` در یک دستور if واحد استفاده کنید.

## ۸,۱۵ واژه‌نامه

**اشاره‌گر مشترک(aliasing)** : حالتی که در آن دو یا چند متغیر به یک شیء واحد اشاره می‌کنند.

**جداکننده(delimiter)** : یک نویسه یا رشته‌ای از نویسه‌ها که برای تعیین محل تقسیم یک رشته به کار می‌رود.

**عنصر(element)** : یکی از مقادیر موجود در یک لیست (یا توالی دیگر)؛ همچنین به آن "آیتم" نیز گفته می‌شود.

**معادل(equivalent)** : دارای همان مقدار.

**اندیس(index)** : یک مقدار عدد صحیح که مکان یک عنصر در لیست را مشخص می‌کند.

**یکسان(identical)** :اشاره به یک شیء واحد (که دلالت بر معادل بودن نیز دارد).

**لیست(list)** : یک توالی از مقادیر.

**پیمایش لیست(list traversal)** : دستیابی متوالی به هر عنصر در یک لیست.

**لیست تو در تو(nested list)** : لیستی که یکی از عناصر آن خود یک لیست دیگر است.

**شیء(object)** : چیزی که یک متغیر می‌تواند به آن اشاره کند. یک شیء دارای نوع و مقدار است.

**ارجاع(reference)** : رابطه بین یک متغیر و مقدار آن.

## ۸,۱۶ تمرین ها

**تمرین ۴: یافتن تمام واژه‌های منحصربه‌فرد در یک فایل**

شکسپیر در آثار خود بیش از ۲۰۰۰۰۰ واژه به کار برده است. اما چگونه می‌توان این موضوع را تعیین کرد؟ چگونه می‌توان فهرست همه واژه‌هایی را که شکسپیر استفاده کرده است تولید کرد؟ آیا باید تمام آثار او را دانلود کنید، بخوانید و همه واژه‌های منحصربه‌فرد را دستی دنبال کنید؟

اجازه دهید به‌جای این کار از پایتون استفاده کنیم. تمام واژه‌های منحصربه‌فرد ذخیره‌شده در فایلی به‌نام **romeo.txt** که شامل بخشی از آثار شکسپیر است را فهرست کنید و آن‌ها را به ترتیب الفبایی مرتب کنید.

برای شروع، یک نسخه از فایل زیر را دانلود کنید:

www.py4e.com/code3/romeo.txt

یک **لیست از واژه‌های منحصربه‌فرد** ایجاد کنید که قرار است نتیجه نهایی در آن ذخیره شود. برنامه‌ای بنویسید که فایل **romeo.txt** را باز کرده و خط‌به‌خط بخواند. برای هر خط، آن را با استفاده از تابع `split` به یک فهرست از واژه‌ها تقسیم کنید. برای هر واژه، بررسی کنید که آیا پیش‌تر در فهرست واژه‌های منحصربه‌فرد وجود داشته است یا خیر. اگر آن واژه در فهرست وجود نداشت، آن را به فهرست اضافه کنید.

وقتی اجرای برنامه کامل شد، فهرست واژه‌های منحصربه‌فرد را **مرتب کرده و به ترتیب الفبایی چاپ کنید.**

```
Enter file: romeo.txt
['Arise', 'But', 'It', 'Juliet', 'Who', 'already',
'and', 'breaks', 'east', 'envious', 'fair', 'grief',
'is', 'kill', 'light', 'moon', 'pale', 'sick', 'soft',
'sun', 'the', 'through', 'what', 'window',
'with', 'yonder']
```

**تمرین ۵: کاربر ایمیل مینیمال**

فرمت **MBOX**(میل‌باکس) یک قالب محبوب برای ذخیره و به‌اشتراک‌گذاری مجموعه‌ای از ایمیل‌ها است. این فرمت در اوایل کارِ سرورها و برنامه‌های دسکتاپ ایمیل استفاده می‌شد. بدون ورود به جزئیات زیاد، باید بدانید که MBOX یک فایل متنی است که ایمیل‌ها را پشت سر هم ذخیره می‌کند. ایمیل‌ها توسط یک خط ویژه که با **From**(دقت کنید: بدون دونقطه) شروع می‌شود از هم جدا می‌شوند. نکته‌ی مهم: خطوطی که با **From:**(با دونقطه) شروع می‌شوند، بخشی از محتوای ایمیل هستند و *به‌عنوان جداکننده عمل نمی‌کنند*.

تصور کنید شما یک برنامه‌ی بسیار ساده‌ی مدیریت ایمیل نوشته‌اید که ایمیل‌های موجود در صندوق ورودی کاربر را فهرست می‌کند و تعداد ایمیل‌ها را می‌شمارد.

یک برنامه بنویسید که فایل داده‌ی ایمیل (mail box) را خوانده و هر زمان که به خطی رسیدید که با **"From"**شروع می‌شود، آن خط را با استفاده از تابع `split` به کلمات جدا کنید. در این تمرین، هدف ما یافتن فردی است که پیام را ارسال کرده است؛ که این اطلاعات در **دومین کلمه** از خط From قرار دارد:

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
```

برنامه باید خط‌های From را پردازش کرده و **دومین کلمه** را برای هر خطی که با From شروع می‌شود چاپ کند. سپس تعداد خطوطی را که با **From**(و نه From:) شروع شده‌اند بشمارد و در پایان چاپ کند.

در زیر یک نمونه خروجی مناسب ارائه شده است (چند خط آن حذف شده است):

```
python fromcount.py
Enter a file name: mbox-short.txt
stephen.marquard@uct.ac.za
louis@media.berkeley.edu
zqian@umich.edu

[...some output removed...]

ray@media.berkeley.edu
cwen@iupui.edu
cwen@iupui.edu
cwen@iupui.edu
There were 27 lines in the file with From as the first word
```

**تمرین ۶:** برنامه‌ای را بازنویسی کنید که از کاربر فهرستی از اعداد را می‌پرسد و هنگامی که کاربر «done» را وارد می‌کند، بیشترین و کمترین عدد را در انتهای برنامه چاپ می‌کند. برنامه را طوری بنویسید که اعدادی را که کاربر وارد می‌کند در یک فهرست ذخیره کند و پس از پایان حلقه از توابع `max()` و `min()` برای محاسبه بیشترین و کمترین عدد استفاده نماید.

```
Enter a number: 6
Enter a number: 2
Enter a number: 9
Enter a number: 3
Enter a number: 5
Enter a number: done
Maximum: 9.0
Minimum: 2.0
```

## فصل نهم

# دایره‌المعارف‌ها(Dictionaries)

یک دایره‌المعارف (یا دیکشنری) شبیه به یک لیست است، اما کلی‌تر. در یک لیست، موقعیت‌های شاخص (index) باید اعداد صحیح باشند؛ اما در یک دیکشنری، شاخص‌ها می‌توانند (تقریباً) از هر نوعی باشند.

می‌توانید یک دیکشنری را به صورت نگاشتی (mapping) بین مجموعه‌ای از شاخص‌ها (که کلید یا key نامیده می‌شوند) و مجموعه‌ای از مقادیر (values) در نظر بگیرید. هر کلید به یک مقدار نگاشت می‌شود. ارتباط میان یک کلید و مقدارش، جفت کلید-مقدار (key-value pair) یا گاهی یک "آیتم (item) " نامیده می‌شود.

برای مثال، ما یک دیکشنری می‌سازیم که کلمات انگلیسی را به معادل اسپانیایی‌شان نگاشت می‌دهد؛ بنابراین کلیدها و مقادیر هر دو رشته (string) هستند.

تابع `dict` یک دیکشنری جدید بدون هیچ آیتمی می‌سازد. از آنجا که `dict` نام یک تابع داخلی در پایتون است، باید از استفاده از آن به عنوان نام متغیر خودداری کنید.

```
>>> eng2sp = dict()
>>> print(eng2sp)
{}
```

آکولادهای باز و بسته `{}` یک دیکشنری خالی را نمایش می‌دهند. برای افزودن آیتم‌ها به دیکشنری، می‌توانید از براکت‌های مربعی استفاده کنید :

```
>>> eng2sp['one'] = 'uno'
```

این خط یک آیتم ایجاد می‌کند که کلید `'one'` را به مقدار `"uno"` نگاشت می‌دهد. اگر دوباره دیکشنری را چاپ کنیم، یک جفت کلید-مقدار می‌بینیم که بین کلید و مقدار، دو نقطه (:) قرار دارد:

```
>>> print(eng2sp)
{'one': 'uno'}
```

این قالب خروجی همچنین به عنوان قالب ورودی نیز استفاده می‌شود. برای مثال، می‌توانید یک دیکشنری جدید با سه آیتم ایجاد کنید.

```
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
>>> print(eng2sp)
{'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

در پایتون از نسخه‌ی ۳٫۷ به بعد، ترتیب جفت‌های کلید-مقدار در دیکشنری همان ترتیبی است که در ورودی وارد شده‌اند، یعنی دیکشنری‌ها اکنون ساختارهای مرتب‌شده‌ای هستند. اما این موضوع اهمیت چندانی ندارد، چون عناصر یک دیکشنری هرگز با اندیس‌های عدد صحیح فراخوانی نمی‌شوند. در عوض، شما از کلیدها برای جستجوی مقدارهای مربوطه استفاده می‌کنید:

```
>>> print(eng2sp['two'])
'dos'
```

کلید `'two'` همیشه به مقدار `"dos"` نگاشته می‌شود، بنابراین ترتیب آیتم‌ها اهمیتی ندارد. اگر کلیدی در دیکشنری وجود نداشته باشد، یک استثناء (خطا) ایجاد می‌شود :

```
>>> print(eng2sp['four'])
KeyError: 'four'
```

تابع `len` روی دیکشنری‌ها نیز کار می‌کند؛ این تابع تعداد جفت‌های کلید-مقدار را بازمی‌گرداند :

```
>>> len(eng2sp)
3
```

عملگر `in` روی دیکشنری‌ها نیز کار می‌کند؛ این عملگر به شما می‌گوید که آیا چیزی به عنوان یک کلید در دیکشنری وجود دارد یا نه (وجود داشتن به‌عنوان یک مقدار کافی نیست)

```
>>> 'one' in eng2sp
True
>>> 'uno' in eng2sp
False
```

برای مشاهده اینکه آیا چیزی به عنوان مقدار در یک دیکشنری وجود دارد یا خیر، می‌توانید از متد `values` استفاده کنید که مقادیر را به صورت نوعی باز می‌گرداند که می‌توان آن را به یک لیست تبدیل کرد، و سپس از عملگر `in` استفاده کنید :

```
>>> vals = list(eng2sp.values())
>>> 'uno' in vals
True
```

عملگر `in` برای لیست‌ها و دیکشنری‌ها از الگوریتم‌های مختلفی استفاده می‌کند. برای لیست‌ها، از الگوریتم جستجوی خطی استفاده می‌کند. با طولانی‌تر شدن لیست، زمان جستجو به‌طور مستقیم متناسب با طول لیست افزایش می‌یابد. برای دیکشنری‌ها، پایتون از الگوریتمی به نام **جدول هش (hash table)** استفاده می‌کند که ویژگی قابل‌توجهی دارد: زمان اجرای عملگر `in` تقریباً ثابت است و وابسته به تعداد آیتم‌های موجود در دیکشنری نیست. من توضیح نمی‌دهم که چرا توابع هش این‌قدر جادویی هستند، اما می‌توانید برای مطالعه بیشتر در این مورد به [منبع] مراجعه کنید[۱].wikipedia.org/wiki/Hash_table.

[۱] اگر مایلید اطلاعات بیشتری درباره جدول‌های درهم‌سازی (Hash Tables) کسب کنید، می‌توانید در دوره‌ای که در وب‌سایت https://www.cc4e.com ارائه شده است شرکت کنید. این دوره به بررسی نحوه پیاده‌سازی دیکشنری در زبان برنامه‌نویسی پایتون، با استفاده از زبان برنامه‌نویسی C می‌پردازد.

**تمرین یک:** یک کپی از فایل را دانلود کنید.

www.py4e.com/code3/words.txt

برنامه‌ای بنویسید که واژه‌های موجود در فایل words.txt را بخواند و آن‌ها را به‌عنوان کلیدهای یک دیکشنری ذخیره کند. مقدارهای اختصاص‌یافته به این کلیدها اهمیتی ندارند. سپس می‌توانید از عملگر in به‌عنوان روشی سریع برای بررسی وجود یک رشته در دیکشنری استفاده کنید.

## ۹,۱ دیکشنری به‌عنوان مجموعه‌ای از شمارنده‌ها

فرض کنید یک رشته به شما داده شده و می‌خواهید تعداد دفعاتی که هر حرف در آن ظاهر می‌شود را بشمارید. چندین راه برای انجام این کار وجود دارد:

۱. می‌توانید ۲۶ متغیر ایجاد کنید، یکی برای هر حرف از الفبا. سپس می‌توانید رشته را پیمایش کرده و برای هر کاراکتر، شمارنده‌ی مربوطه را افزایش دهید، احتمالاً با استفاده از یک حلقه شرط زنجیره‌ای.
۲. می‌توانید یک لیست با ۲۶ عنصر بسازید. سپس می‌توانید هر کاراکتر را به یک عدد تبدیل کرده ( با استفاده از تابع داخلی ord)، عدد را به عنوان ایندکس در لیست استفاده کنید و شمارنده‌ی مناسب را افزایش دهید.
۳. می‌توانید یک دیکشنری با کاراکترها به عنوان کلیدها و شمارنده‌ها به عنوان مقادیر متناظر ایجاد کنید. اولین بار که یک کاراکتر را می‌بینید، یک آیتم به دیکشنری اضافه می‌کنید. پس از آن، مقدار یک آیتم موجود را افزایش می‌دهید.

هر یک از این گزینه‌ها همان محاسبه را انجام می‌دهند، اما هرکدام این محاسبه را به شیوه‌ای متفاوت پیاده‌سازی می‌کنند.

یک پیاده‌سازی، روشی برای انجام محاسبه است؛ برخی از پیاده سازی‌ها بهتر از دیگران هستند. برای مثال، مزیت پیاده سازی دیکشنری این است که نیازی به دانستن از پیش نداریم که کدام حروف در رشته ظاهر می‌شوند و فقط باید فضایی برای حروفی که واقعاً ظاهر می‌شوند اختصاص دهیم.

در ادامه، کد ممکن است به این شکل باشد:

```
word = 'brontosaurus'
d = dict()
for c in word:
    if c not in d:
        d[c] = 1
    else:
        d[c] = d[c] + 1
print(d)
```

ما در واقع در حال محاسبه یک **هیستوگرام** هستیم، که یک اصطلاح آماری برای مجموعه‌ای از شمارنده‌ها (یا فراوانی‌ها) است.

حلقه‌ی for رشته را پیمایش می‌کند. هر بار که حلقه اجرا می‌شود، اگر کاراکتر c در دیکشنری نباشد، یک آیتم جدید با کلید c و مقدار اولیه ۱ ایجاد می‌کنیم (چون این حرف را یک‌بار دیده‌ایم). اگر c قبلاً در دیکشنری وجود داشته باشد، مقدار d['c'] را افزایش می‌دهیم.

در ادامه، خروجی برنامه به این شکل است:

```
{'b': 1, 'r': 2, 'o': 2, 'n': 1, 't': 1, 's': 2, 'a': 1, 'u': 2}
```

هیستوگرام نشان می‌دهد که حروف "a" و "b" هرکدام یک‌بار ظاهر شده‌اند؛ حرف "o" دو بار ظاهر شده و به همین ترتیب.

دیکشنری‌ها متدی به نام get دارند که یک کلید و یک مقدار پیش‌فرض می‌گیرد. اگر کلید در دیکشنری وجود داشته باشد، get مقدار متناظر با آن را باز می‌گرداند؛ در غیر این صورت، مقدار پیش‌فرض را باز می‌گرداند. برای مثال:

```
>>> counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
>>> print(counts.get('jan', 0))
100
>>> print(counts.get('tim', 0))
0
```

ما می‌توانیم از متد get برای نوشتن حلقه‌ی هیستوگرام خود به‌طور خلاصه‌تر استفاده کنیم. زیرا متد get به‌طور خودکار وضعیت‌هایی را که یک کلید در دیکشنری وجود ندارد مدیریت می‌کند، بنابراین می‌توانیم چهار خط کد را به یک خط کاهش داده و دستور شرطی if را حذف کنیم.

```
word = 'brontosaurus'
d = dict()
for c in word:
    d[c] = d.get(c,0) + 1
print(d)
```

استفاده از متد get برای ساده‌سازی این حلقه‌ی شمارش، به یک "اصطلاح" بسیار متداول در پایتون تبدیل می‌شود و ما در ادامه‌ی کتاب بارها از آن استفاده خواهیم کرد. بنابراین، بهتر است لحظه‌ای وقت بگذارید و حلقه‌ای که از دستور شرطی if و عملگر in استفاده می‌کند را با حلقه‌ای که از متد get استفاده می‌کند مقایسه کنید. هر دو دقیقاً همان کار را انجام می‌دهند، اما یکی مختصرتر است.

## ۹,۲ دیکشنری‌ها و فایل‌ها

یکی از استفاده های رایج دیکشنری، شمارش تعداد وقوع کلمات در یک فایل با متن نوشته‌شده است. بیایید با یک فایل بسیار ساده از کلمات که از متن "رومئو و ژولیت" گرفته شده شروع کنیم. برای مجموعه‌ی اول مثال‌ها، از یک نسخه‌ی کوتاه شده و ساده شده از متن استفاده خواهیم کرد که هیچ نشانه‌گذاری ندارد. بعدها با متن صحنه‌ای که شامل نشانه گذاری باشد، کار خواهیم کرد.

```
But soft what light through yonder window breaks
It is the east and Juliet is the sun
Arise fair sun and kill the envious moon
Who is already sick and pale with grief
```

ما یک برنامه پایتون می‌نویسیم تا از میان خطوط فایل عبور کند، هر خط را به یک لیست از کلمات تقسیم کند، و سپس از میان هر یک از کلمات در آن خط عبور کرده و هر کلمه را با استفاده از یک دیکشنری شمارش کند.

خواهید دید که دو حلقه `for` داریم. حلقه بیرونی در حال خواندن خطوط فایل است و حلقه درونی در حال پیمایش هر یک از کلمات در آن خط خاص است. این یک مثال از الگویی به نام **حلقه‌های تو در تو** است، زیرا یکی از حلقه‌ها حلقه بیرونی و دیگری حلقه درونی است.

از آنجا که حلقه درونی تمام تکرارهای خود را هر بار که حلقه بیرونی یک تکرار انجام می‌دهد اجرا می‌کند، ما حلقه درونی را به عنوان حلقه‌ای که "سریع‌تر" اجرا می‌شود و حلقه بیرونی را به عنوان حلقه‌ای که کندتر اجرا می‌شود در نظر می‌گیریم.

ترکیب این دو حلقه تو در تو تضمین می‌کند که هر کلمه در هر خط از فایل ورودی شمارش شود.

```
fname = input('Enter the file name: ')
try:
    fhand = open(fname)
except:
    print('File cannot be opened:', fname)
    exit()

counts = dict()
for line in fhand:
    words = line.split()
    for word in words:
        if word not in counts:
            counts[word] = 1
        else:
            counts[word] += 1

print(counts)

# Code: https://www.py4e.com/code3/count1.py
```

در دستور `else` ما، از روش مختصرتر برای افزایش مقدار یک متغیر استفاده می‌کنیم `counts[word] += 1`. معادل `counts[word] = counts[word] + 1` است. هر کدام از این روش‌ها می‌تواند برای تغییر مقدار یک متغیر به هر میزان دلخواه استفاده شود. روش‌های مشابهی برای `*=`, `-=` و `/=` نیز وجود دارند.

هنگامی که برنامه را اجرا می‌کنیم، خروجی خامی از تمام شمارش‌ها را در ترتیب هش نامرتب مشاهده می‌کنیم. (فایل `romeo.txt` در دسترس است) www.py4e.com/code3/romeo.txt

```
python count1.py
Enter the file name: romeo.txt
{'But': 1, 'soft': 1, 'what': 1, 'light': 1, 'through': 1, 'yonder': 1,
'window': 1, 'breaks': 1, 'It': 1, 'is': 3, 'the': 3, 'east': 1, 'and': 3,
'Juliet': 1, 'sun': 2, 'Arise': 1, 'fair': 1, 'kill': 1, 'envious': 1,
'moon': 1, 'Who': 1, 'already': 1, 'sick': 1, 'pale': 1, 'with': 1,
'grief': 1}
```

مشاهده‌ی دیکشنری برای پیدا کردن رایج‌ترین کلمات و تعداد دفعات آن‌ها کمی ناراحت‌کننده است، بنابراین باید کد بیشتری به پایتون اضافه کنیم تا خروجی‌ای به دست آوریم که مفیدتر باشد.

## ۹,۳ حلقه‌ها و دیکشنری‌ها

اگر از یک دیکشنری به‌عنوان توالی در یک عبارت `for` استفاده کنید، این حلقه کلیدهای دیکشنری را پیمایش می‌کند. این حلقه هر کلید و مقدار متناظر با آن را چاپ می‌کند:

```
counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
for key in counts:
    print(key, counts[key])
```

Here's what the output looks like:

```
chuck 1
annie 42
jan 100
```

دوباره، کلیدها به ترتیب مرتب شده‌اند. ما می‌توانیم از این الگو برای پیاده‌سازی انواع اصطلاحات حلقه‌ای که پیش‌تر توضیح داده‌ایم استفاده کنیم. برای مثال، اگر بخواهیم تمام ورودی‌های دیکشنری که مقدار آن‌ها بیشتر از ده است را پیدا کنیم، می‌توانیم کد زیر را بنویسیم:

```
counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
for key in counts:
    if counts[key] > 10 :
        print(key, counts[key])
```

حلقه‌ی `for` روی کلیدهای دیکشنری پیمایش می‌کند، بنابراین باید از عملگر اندیس‌گذاری (index operator) استفاده کنیم تا مقدار متناظر با هر کلید را بازیابی کنیم. در ادامه، خروجی حاصل از این فرآیند آورده شده است:

```
annie 42
jan 100
```

ما فقط ورودی‌هایی را می‌بینیم که مقدار آن‌ها بیشتر از ۱۰ است.

اگر بخواهید کلیدها را به ترتیب الفبایی چاپ کنید، ابتدا باید با استفاده از متد `keys` که برای اشیاء دیکشنری در دسترس است، یک فهرست از کلیدهای دیکشنری بسازید. سپس این فهرست را مرتب کرده و با استفاده از یک حلقه روی فهرست مرتب‌شده پیمایش کنید. در هر گام، مقدار متناظر با هر کلید را جستجو کرده و جفت کلید-مقدار را به ترتیب مرتب‌شده چاپ نمایید، مانند نمونه‌ی زیر:

```
counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
lst = list(counts.keys())
print(lst)
lst.sort()
print(lst)
for key in lst:
    print(key, counts[key])
```

Here's what the output looks like:

```
['chuck', 'annie', 'jan']
['annie', 'chuck', 'jan']
annie 42
chuck 1
jan 100
```

ابتدا فهرست کلیدها را مشاهده می‌کنید که به ترتیبی غیر الفبایی و با استفاده از متد `keys` به دست آمده‌اند. سپس جفت‌های کلید-مقدار را می‌بینید که به ترتیب الفبایی توسط حلقه‌ی `for` نمایش داده شده‌اند.

## ۹,۴ پردازش پیشرفته‌ی متن

در مثال بالا که از فایل `romeo.txt` استفاده کردیم، فایل را تا حد امکان ساده کردیم و تمام علائم نگارشی را به‌صورت دستی حذف کردیم. اما متن اصلی دارای علائم نگارشی فراوانی است، همان‌طور که در نمونه‌ی زیر مشاهده می‌شود:

```
But, soft! what light through yonder window breaks?
It is the east, and Juliet is the sun.
Arise, fair sun, and kill the envious moon,
Who is already sick and pale with grief,
```

از آن‌جایی که تابع split در پایتون به دنبال فاصله‌ها می‌گردد و کلمات را به عنوان توکن‌هایی که با فاصله از هم جدا شده‌اند در نظر می‌گیرد، ما کلمات «soft!» و «soft» را دو کلمه‌ی متفاوت تلقی خواهیم کرد و برای هر کدام یک ورودی جداگانه در دیکشنری ایجاد خواهد شد. همچنین، چون فایل دارای حروف بزرگ و کوچک است، کلمات «who» و «Who» نیز به عنوان دو کلمه‌ی متفاوت با شمارش جداگانه در نظر گرفته می‌شوند.

ما می‌توانیم هر دو این مشکل را با استفاده از متدهای رشته‌ای lower، punctuation و translate حل کنیم. در این میان، متد translate ظریف‌ترین آن‌هاست. در ادامه، مستندات مربوط به متد translate آمده است:

```
line.translate(str.maketrans(fromstr, tostr, deletestr))
```

*Replace the characters in `fromstr` with the character in the same position in `tostr` and delete all characters that are in `deletestr`. The `fromstr` and `tostr` can be empty strings and the `deletestr` parameter can be omitted.*

ما پارامتر tostr را مشخص نخواهیم کرد، اما از پارامتر deletestr استفاده خواهیم کرد تا تمام علائم نگارشی را حذف کنیم. حتی این اجازه را به پایتون خواهیم داد که خودش فهرست نویسه‌هایی را که به عنوان «علائم نگارشی» در نظر می‌گیرد، به ما بدهد:

```
>>> import string
>>> string.punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

پارامترهایی که توسط تابع translate استفاده می‌شدند، در پایتون نسخه ۲,۰ متفاوت بودند. ما تغییرات زیر را در برنامه‌مان اعمال می‌کنیم:

```
import string

fname = input('Enter the file name: ')
try:
    fhand = open(fname)
except:
    print('File cannot be opened:', fname)
    exit()

counts = dict()
for line in fhand:
    line = line.rstrip()
    # First two parameters are empty strings
    line = line.translate(line.maketrans("", "", string.punctuation))
    line = line.lower()
    words = line.split()
    for word in words:
        if word not in counts:
            counts[word] = 1
        else:
            counts[word] += 1

print(counts)
```

```
# Code: https://www.py4e.com/code3/count2.py
```

بخشی از یادگیری «هنر پایتون‌نویسی» یا «تفکر پایتونی» این است که دریابید پایتون اغلب برای بسیاری از مسائل رایج در تحلیل داده‌ها، قابلیت‌هایی داخلی دارد. با گذشت زمان، آن‌قدر نمونه کد خواهید دید و آن‌قدر مستندات را خواهید خواند که یاد می‌گیرید کجا را جستجو کنید تا ببینید آیا کسی پیش از شما چیزی نوشته است که کارتان را بسیار ساده‌تر کند یا نه.

در ادامه، نسخه‌ی خلاصه‌شده‌ای از خروجی آورده شده است:

```
Enter the file name: romeo-full.txt
{'romeo': 40, 'and': 42, 'juliet': 32, 'act': 1, '2': 2, 'scene': 2,
'ii': 1, 'capulets': 1, 'orchard': 2, 'enter': 1, 'he': 5, 'jests': 1,
'at': 9, 'scars': 1, 'that': 30, 'never': 2, 'felt': 1, 'a': 24,
'wound': 1, 'appears': 1, 'above': 6, 'window': 2, 'but': 18,
'soft': 1, 'what': 11, 'light': 5, 'through': 2, 'yonder': 2,
'breaks': 1, ...}
```

بررسی این خروجی هنوز هم دشوار و زمان‌بر است، اما می‌توانیم از پایتون استفاده کنیم تا دقیقاً همان چیزی را که به دنبالش هستیم در اختیارمان قرار دهد. با این حال، برای انجام این کار، باید با **تاپل‌ها (tuples)** در پایتون آشنا شویم. پس از یادگیری تاپل‌ها، دوباره به این مثال بازخواهیم گشت.

## ۹,۵ اشکال‌زدایی

هنگامی که با مجموعه داده‌های بزرگ‌تری کار می‌کنید، اشکال‌زدایی با استفاده از چاپ و بررسی دستی داده‌ها می‌تواند دشوار و زمان‌بر شود. در اینجا چند پیشنهاد برای اشکال‌زدایی در مواجهه با داده‌های حجیم ارائه شده است:

**ورودی را کوچک‌تر کنید**

اگر ممکن است، اندازه‌ی مجموعه‌داده را کاهش دهید. برای مثال، اگر برنامه یک فایل متنی را می‌خواند، با فقط ۱۰ خط اول آن شروع کنید یا از کوچک‌ترین مثالی که می‌توانید بیابید استفاده نمایید. می‌توانید یا فایل‌ها را مستقیماً ویرایش کنید، یا (ترجیحاً) برنامه را به‌گونه‌ای تغییر دهید که فقط n خط اول را بخواند. اگر خطایی وجود دارد، می‌توانید مقدار n را به کوچک‌ترین عددی کاهش دهید که هنوز خطا را ایجاد می‌کند، سپس با افزایش تدریجی آن، خطاها را یافته و اصلاح کنید.

**خلاصه‌ها و نوع‌ها را بررسی کنید**

به‌جای چاپ و بررسی کل داده‌ها، خلاصه‌هایی از داده‌ها را چاپ کنید؛ برای مثال، تعداد آیتم‌های موجود در یک دیکشنری یا مجموع مقادیر یک فهرست عددی. یکی از دلایل رایج بروز خطا در زمان اجرا، وجود مقداری با نوع نادرست است. برای اشکال‌زدایی چنین خطاهایی، اغلب کافی‌ست نوع یک مقدار را چاپ کنید.

**برنامه‌های خودسنج بنویسید**

گاهی می‌توانید کدی بنویسید که به‌صورت خودکار خطاها را بررسی کند. برای نمونه، اگر میانگین یک فهرست عددی را محاسبه می‌کنید، می‌توانید بررسی کنید که نتیجه بزرگ‌تر از بزرگ‌ترین عنصر فهرست یا کوچک‌تر از کوچک‌ترین نباشد. به این کار «بررسی سلامت» (sanity check) می‌گویند، زیرا نتایجی را که کاملاً غیرمنطقی‌اند شناسایی می‌کند.

نوع دیگری از بررسی، مقایسه‌ی نتایج دو محاسبه‌ی متفاوت است تا ببینید آیا با یکدیگر سازگار هستند یا خیر. به این روش «بررسی سازگاری» (consistency check) گفته می‌شود.

**خروجی را با قالب‌بندی مناسب نمایش دهید**

قالب‌بندی خروجی مربوط به اشکال‌زدایی می‌تواند باعث شود خطاها راحت‌تر دیده شوند. دوباره باید گفت که صرف وقت برای ساخت ابزارهای کمکی (scaffolding) می‌تواند زمان لازم برای اشکال‌زدایی را کاهش دهد.

## ۹,۶ واژه‌نامه

**Dictionary فرهنگ‌واژه یا دیکشنری**: یک نگاشت از مجموعه‌ای از کلیدها به مقادیر متناظر آن‌ها.

**Hashtable جدول درهم‌سازی**: الگوریتمی که برای پیاده‌سازی دیکشنری‌های پایتون استفاده می‌شود.

**hash function تابع درهم‌ساز**: تابعی که توسط جدول درهم‌سازی برای محاسبه‌ی محل ذخیره‌سازی یک کلید استفاده می‌شود.

**Histogram نمودار فراوانی**: مجموعه‌ای از شمارنده‌ها.

**Implementation پیاده‌سازی**: روشی برای انجام یک محاسبه.

**Item مورد یا آیتم**: نام دیگر برای یک جفت کلید-مقدار.

**Key کلید:** شیئ که به عنوان بخش اول از یک جفت کلید-مقدار در دیکشنری ظاهر می‌شود.

**key-value pair جفت کلید-مقدار**:نمایش نگاشتی از یک کلید به یک مقدار.

**Lookup جستجو**: عملیاتی در دیکشنری که یک کلید را دریافت کرده و مقدار متناظر آن را پیدا می‌کند.

**nested loops حلقه‌های تو در تو**: زمانی که یک یا چند حلقه درون یک حلقه‌ی دیگر قرار دارند. حلقه‌ی درونی در هر بار اجرای حلقه‌ی بیرونی، به‌طور کامل اجرا می‌شود.

**Value مقدار** : شی که به عنوان بخش دوم از یک جفت کلید-مقدار در دیکشنری ظاهر می‌شود. این تعریف خاص‌تر از کاربرد پیشین ما از واژه‌ی «مقدار» است.

## ۹,۷ تمرین ها

**تمرین ۲:**برنامه‌ای بنویسید که هر پیام ایمیل را بر اساس اینکه «تعهد(commit) » در کدام روز هفته انجام شده، دسته‌بندی کند. برای این کار، به دنبال خطوطی بگردید که با **"From"** شروع می‌شوند، سپس **سومین کلمه** را در خط پیدا کنید و یک شمارشِ پیوسته برای هر یک از روزهای هفته نگه دارید. در پایانِ برنامه، محتوای دیکشنری خود را چاپ کنید (ترتیب مهم نیست).

خط نمونه :

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
```

نمونه اجرا شده:

```
python dow.py
Enter a file name: mbox-short.txt
{'Fri': 20, 'Thu': 6, 'Sat': 1}
```

**تمرین ۳:** یک برنامه بنویسید که یک گزارش (log) ایمیل را بخواند، با استفاده از یک دیکشنری یک هیستوگرام بسازد تا شمارش کند چه تعداد پیام از هر نشانی ایمیل آمده است، و سپس دیکشنری را چاپ کند.

```
Enter file name: mbox-short.txt
{'gopal.ramasammycook@gmail.com': 1, 'louis@media.berkeley.edu': 3,
'cwen@iupui.edu': 5, 'antranig@caret.cam.ac.uk': 1,
'rjlowe@iupui.edu': 2, 'gsilver@umich.edu': 3,
'david.horwitz@uct.ac.za': 4, 'wagnermr@iupui.edu': 1,
'zqian@umich.edu': 4, 'stephen.marquard@uct.ac.za': 2,
'ray@media.berkeley.edu': 1}
```

**تمرین ۴:**

به برنامه بالا کدی اضافه کنید تا مشخص کند چه کسی بیشترین تعداد پیام را در فایل دارد. پس از اینکه تمام داده‌ها خوانده شد

و دیکشنری ساخته شد، با استفاده از یک حلقه بیشینه (حداکثر)—به فصل ۵: حلقه‌های بیشینه و کمینه مراجعه کنید—در میان دیکشنری جست‌وجو کنید تا پیدا کنید چه کسی بیشترین پیام را دارد و تعداد پیام‌های آن شخص را چاپ کنید.

```
Enter a file name: mbox-short.txt
cwen@iupui.edu 5

Enter a file name: mbox.txt
zqian@umich.edu 195
```

**تمرین ۵ :**این برنامه به‌جای ثبت آدرس کاملِ ایمیلی که پیام از آن ارسال شده است، **نام دامنه**‌ای را که پیام از آن فرستاده شده ثبت می‌کند. در پایان برنامه، **محتوای دیکشنری خود را چاپ کنید.**

```
python schoolcount.py
Enter a file name: mbox-short.txt
{'media.berkeley.edu': 4, 'uct.ac.za': 6, 'umich.edu': 7,
'gmail.com': 1, 'caret.cam.ac.uk': 1, 'iupui.edu': 8}
```

# فصل دهم

# تاپل‌ها

## ۱۰,۱ تاپل‌ها تغییرناپذیرند

تاپل (Tuple) دنباله‌ای از مقادیر است، بسیار شبیه به لیست. مقادیری که در یک تاپل ذخیره می‌شوند می‌توانند از هر نوعی باشند و با استفاده از اعداد صحیح (integer) اندیس‌گذاری می‌شوند. تفاوت مهم آن با لیست‌ها این است که تاپل‌ها تغییرناپذیر (immutable) هستند.

تاپل‌ها همچنین قابل مقایسه (comparable) و هش‌پذیر (hashable) هستند، بنابراین می‌توانیم لیست‌هایی از تاپل‌ها را مرتب سازی کنیم و از تاپل‌ها به عنوان کلید در دیکشنری‌های پایتون استفاده کنیم.

از نظر نحوی(syntax) ، تاپل یک لیست از مقادیر است که با ویرگول از هم جدا شده‌اند:

```
>>> t = 'a', 'b', 'c', 'd', 'e'
```

اگرچه این کار ضروری نیست، اما رایج است که تاپل‌ها را در پرانتز قرار دهیم تا هنگام مشاهده کد پایتون، بتوانیم سریع‌تر آن‌ها را به‌عنوان تاپل تشخیص دهیم:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
```

برای ایجاد یک تاپل با تنها یک عنصر، باید یک ویرگول در انتهای آن قرار دهید :

```
>>> t1 = ('a',)
>>> type(t1)
<type 'tuple'>
```

بدون وجود ویرگول، پایتون عبارت (`'a'`) را به عنوان یک عبارت معمولی با یک رشته درون پرانتز در نظر می‌گیرد که نتیجه‌ی آن یک رشته خواهد بود:

```
>>> t2 = ('a')
>>> type(t2)
<type 'str'>
```

روش دیگری برای ساختن یک تاپل استفاده از تابع داخلی `tuple` است. اگر بدون هیچ آرگومانی فراخوانی شود، یک تاپل خالی ایجاد می‌کند :

```
>>> t = tuple()
>>> print(t)
()
```

اگر آرگومان یک دنباله (رشته، لیست یا تاپل) باشد، نتیجه‌ی فراخوانی تابع `tuple`، یک تاپل خواهد بود که شامل عناصر همان دنباله است:

```
>>> t = tuple('lupins')
>>> print(t)
('l', 'u', 'p', 'i', 'n', 's')
```

از آنجا که `tuple` نام یک سازنده (constructor) در پایتون است، باید از استفاده از آن به عنوان نام یک متغیر خودداری کنید.

بیشتر عملگرهایی که روی لیست‌ها کار می‌کنند، روی تاپل‌ها نیز قابل استفاده هستند. عملگر براکت (`[]`) برای دسترسی به عناصر با اندیس به‌کار می‌رود :

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> print(t[0])
'a'
```

و عملگر برش (slice) یک بازه از عناصر را انتخاب می‌کند.

```
>>> print(t[1:3])
('b', 'c')
```

اما اگر سعی کنید یکی از عناصر تاپل را تغییر دهید، با یک خطا مواجه خواهید شد:

```
>>> t[0] = 'A'
TypeError: object doesn't support item assignment
```

شما نمی‌توانید عناصر یک تاپل را تغییر دهید، اما می‌توانید یک تاپل را با تاپل دیگری جایگزین کنید.

```
>>> t = ('A',) + t[1:]
>>> print(t)
('A', 'b', 'c', 'd', 'e')
```

## ۱۰,۲ مقایسه ی تاپل‌ها

عملگرهای مقایسه در پایتون با تاپل‌ها و دیگر دنباله‌ها (sequences) کار می‌کنند. پایتون ابتدا اولین عنصر از هر دنباله را با هم مقایسه می‌کند. اگر این دو عنصر برابر باشند، به عنصر بعدی می‌رود و این روند ادامه می‌یابد تا جایی که به دو عنصر متفاوت برسد. در این صورت، نتیجه ی مقایسه بر اساس همین دو عنصر متفاوت تعیین می‌شود. اگر به دو عنصر متفاوت برسد، دیگر عناصر باقی مانده (حتی اگر خیلی بزرگ یا مهم باشند) بررسی نمی‌شوند.

```
>>> (0, 1, 2) < (0, 3, 4)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
```

تابع `sort` به همان شیوه عمل می‌کند. ابتدا بر اساس عنصر اول مرتب سازی می‌کند، اما در صورت برابر بودن عنصر اول، به سراغ عنصر دوم می‌رود و به همین ترتیب ادامه می‌دهد.

این ویژگی منجر به الگویی به نام **DSU** می‌شود که مخفف مراحل زیر است:

- **تزئین(Decorate)**
  یک دنباله را با ساختن فهرستی از تاپل‌ها تزئین می‌کنیم؛ به‌طوری‌که یک یا چند کلید مرتب‌سازی قبل از عناصر اصلی دنباله قرار می‌گیرند.
- **مرتب‌سازی(Sort)**
  فهرست تاپل‌ها را با استفاده از تابع `sort` پایتون مرتب می‌کنیم.
- **برداشتن تزئین(Undecorate)**
  عناصر مرتب شده ی اصلی را از فهرست استخراج می‌کنیم.

برای مثال، فرض کنید فهرستی از کلمات دارید و می‌خواهید آن‌ها را از طولانی‌ترین تا کوتاه‌ترین مرتب کنید:

```
txt = 'but soft what light in yonder window breaks'
words = txt.split()
t = list()
for word in words:
    t.append((len(word), word))

t.sort(reverse=True)

res = list()
for length, word in t:
    res.append(word)

print(res)

# Code: https://www.py4e.com/code3/soft.py
```

حلقه ی اول یک فهرست از تاپل‌ها ایجاد می‌کند؛ در هر تاپل، ابتدا طول یک کلمه و سپس خود آن کلمه قرار دارد. تابع `sort` ابتدا عناصر اول تاپل‌ها (یعنی طول کلمات) را با هم مقایسه می‌کند و تنها در صورت تساوی این مقادیر، به سراغ مقایسه ی عنصر دوم (کلمه) می‌رود. پارامتر `reverse=True` به تابع `sort` می‌گوید که فهرست را به ترتیب نزولی مرتب کند.

حلقه ی دوم فهرست تاپل‌ها را پیمایش کرده و فهرستی از کلمات می‌سازد که به ترتیب نزولی طول مرتب شده‌اند. در بین کلمات چهارحرفی، ترتیب به صورت حروف الفبای معکوس است؛ به همین دلیل، «what» پیش از «soft» ظاهر می‌شود.

خروجی برنامه به صورت زیر است:

```
['yonder', 'window', 'breaks', 'light', 'what',
'soft', 'but', 'in']
```

البته زمانی که یک خط شعر به یک فهرست پایتونی تبدیل شود و سپس بر اساس طول کلمات به صورت نزولی مرتب گردد، بخش زیادی از زیبایی شاعرانه ی خود را از دست می‌دهد.

## ۱۰,۳ تخصیص تاپل(Tuple Assignment)

یکی از ویژگی‌های نحوی منحصر به فرد زبان پایتون، امکان استفاده از تاپل در سمت چپ و یک دنباله (sequence) در سمت راست یک دستور انتساب است. این ویژگی به شما اجازه می‌دهد تا در یک دستور، چند متغیر را به صورت هم زمان به مقادیر موجود در یک دنباله نسبت دهید.

در مثال زیر، یک تاپل دو عضوی داریم و به‌سادگی در یک عبارت می‌توانیم عنصر اول و دوم این تاپل را به متغیرهای `x` و `y` نسبت دهیم.

```
>>> m = ( 'have', 'fun' )
>>> x, y = m
>>> x
'have'
>>> y
'fun'
>>>
```

این از انتساب تاپل به تاپل عمومی‌تر است. هم تاپل‌ها و هم لیست‌ها دنباله‌ها (sequences) هستند، بنابراین این نحو با یک لیست دو عنصری نیز کار می‌کند.

```
>>> m = [ 'have', 'fun' ]
>>> x, y = m
>>> x
'have'
>>> y
'fun'
>>>
```

این جادو نیست، پایتون به طور تقریبی نحو انتساب تاپل را به صورت زیر ترجمه می‌کند:

```
>>> m = ( 'have', 'fun' )
>>> x = m[0]
>>> y = m[1]
>>> x
'have'
>>> y
'fun'
>>>
```

از نظر سبکی، وقتی از یک تاپل در سمت چپ عبارت انتساب استفاده می‌کنیم، پرانتزها را حذف می‌کنیم، اما دستور زیر نیز به عنوان یک نحو معتبر است:

```
>>> m = ( 'have', 'fun' )
>>> (x, y) = m
>>> x
'have'
>>> y
'fun'
>>>
```

یک کاربرد هوشمندانه از انتساب تاپلی این است که می‌توانیم مقادیر دو متغیر را تنها در یک دستور با هم جابجا کنیم:

```
>>> a, b = b, a
```

هر دو طرف این دستور تاپل هستند، اما سمت چپ یک تاپل از متغیرهاست و سمت راست یک تاپل از عبارت‌ها. هر مقدار در سمت راست به متغیر مربوط به خودش در سمت چپ اختصاص داده می‌شود. همه ی عبارت‌های سمت راست پیش از انجام هرگونه انتساب ارزیابی می‌شوند.

تعداد متغیرها در سمت چپ و تعداد مقادیر در سمت راست باید برابر باشند.

```
>>> a, b = 1, 2, 3
ValueError: too many values to unpack
```

به طور کلی، سمت راست می‌تواند از هر نوع دنباله‌ای (رشته، لیست یا تاپل) باشد. به عنوان مثال، برای تقسیم یک آدرس ایمیل به نام کاربری و دامنه، می‌توانید بنویسید :

```
>>> addr = 'monty@python.org'
>>> uname, domain = addr.split('@')
```

مقدار بازگشتی از متد split یک لیست با دو عنصر است؛ عنصر اول به متغیر uname و عنصر دوم به متغیر domain اختصاص داده می‌شود.

```
>>> print(uname)
monty
>>> print(domain)
python.org
```

## ۱۰,۴ دیکشنری ها و تاپل ها

دیکشنری‌ها متدی به نام items دارند که یک لیست از تاپل‌ها را برمی‌گرداند، که هر تاپل یک جفت کلید-مقدار است:

```
>>> d = {'b':1, 'a':10, 'c':22}
>>> t = list(d.items())
>>> print(t)
[('b', 1), ('a', 10), ('c', 22)]
```

همانطور که از یک دیکشنری انتظار دارید، موارد در این دیکشنری به ترتیب الفبایی نیستند. با این حال، از آنجایی که لیست تاپل‌ها یک لیست است و تاپل‌ها قابل مقایسه هستند، اکنون می‌توانیم لیست تاپل‌ها را مرتب کنیم. تبدیل یک دیکشنری به یک لیست از تاپل‌ها راهی است برای اینکه محتوای دیکشنری را به ترتیب کلید چاپ کنیم:

```
>>> d = {'b':1, 'a':10, 'c':22}
>>> t = list(d.items())
>>> t
[('b', 1), ('a', 10), ('c', 22)]
>>> t.sort()
>>> t
[('a', 10), ('b', 1), ('c', 22)]
```

لیست جدید به ترتیب حروف الفبای صعودی بر اساس مقدار کلید مرتب شده است.

## ۱۰,۵ تخصیص چندگانه با دیکشنری‌ها

با ترکیب متد items، تخصیص تاپل و حلقه for، می‌توان یک الگوی کد خوب برای پیمایش کلیدها و مقادیر یک دیکشنری در یک حلقه واحد مشاهده کرد:

```
d = {'a':10, 'b':1, 'c':22}
for key, val in d.items():
    print(val, key)
```

این حلقه دارای دو متغیر تکرار است زیرا متد items یک لیست از تاپل‌ها را برمی‌گرداند و key, val یک انتساب تاپل است که به‌طور متوالی از هر یک از زوج‌های کلید-مقدار در دیکشنری عبور می‌کند. برای هر بار تکرار در حلقه، هم key و هم val به زوج کلید-مقدار بعدی در دیکشنری (هنوز به ترتیب هش) منتقل می‌شوند. خروجی این حلقه به‌صورت زیر خواهد بود:

```
10 a
1 b
22 c
```

باز هم می‌بینیم که داده‌ها به ترتیب کلیدهای هش‌شده چاپ می‌شوند (یعنی بدون هیچ ترتیب خاصی). اگر این دو تکنیک را با یکدیگر ترکیب کنیم، می‌توانیم محتوای یک دیکشنری را بر اساس مقدار (value) ذخیره شده در هر زوج کلید-مقدار به‌صورت مرتب شده چاپ کنیم.

برای انجام این کار، ابتدا فهرستی از تاپل‌ها می‌سازیم که در آن هر تاپل به شکل (مقدار، کلید) است. متد items فهرستی از تاپل‌های (کلید، مقدار) به ما می‌دهد، اما این بار می‌خواهیم بر اساس مقدار مرتب کنیم، نه کلید. پس از آنکه فهرستی با تاپل‌های مقدار-کلید ساختیم، مرتب‌سازی این فهرست به ترتیب معکوس (نزولی) و چاپ فهرست مرتب‌شده، کار ساده‌ای خواهد بود.

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> l = list()
>>> for key, val in d.items() :
...     l.append( (val, key) )
...
>>> l
[(10, 'a'), (1, 'b'), (22, 'c')]
>>> l.sort(reverse=True)
>>> l
[(22, 'c'), (10, 'a'), (1, 'b')]
>>>
```

با ساخت دقیق فهرستی از تاپل‌ها به‌گونه‌ای که مقدار (value) به عنوان نخستین عنصر هر تاپل قرار گیرد، می‌توانیم فهرست تاپل‌ها را مرتب کنیم و محتوای دیکشنری خود را بر اساس مقدارها مرتب شده دریافت کنیم.

## ۱۰,۶ رایج‌ترین واژه‌ها

در بازگشت به مثال مداوم خود از متن نمایشنامه رومئو و ژولیت، پرده ی دوم، صحنه ی دوم، می‌توانیم برنامه‌مان را به‌گونه‌ای گسترش دهیم که با استفاده از این تکنیک، ده واژه ی پرکاربرد در متن را به‌صورت زیر چاپ کند:

```
import string
fhand = open('romeo-full.txt')
counts = dict()
for line in fhand:
    line = line.translate(str.maketrans('', '', string.punctuation))
    line = line.lower()
    words = line.split()
    for word in words:
        if word not in counts:
            counts[word] = 1
        else:
            counts[word] += 1

# Sort the dictionary by value
```

```
lst = list()
for key, val in list(counts.items()):
    lst.append((val, key))

lst.sort(reverse=True)

for key, val in lst[:10]:
    print(key, val)

# Code: https://www.py4e.com/code3/count3.py
```

قسمت اول برنامه که فایل را می‌خواند و دیکشنری‌ای می‌سازد که هر واژه را به تعداد تکرارش در سند نگاشت می‌کند، بدون تغییر باقی مانده است. اما به‌جای اینکه فقط تعداد تکرارها را چاپ کنیم و برنامه را تمام کنیم، لیستی از تاپل‌های (مقدار، کلید) می‌سازیم و سپس این لیست را به‌صورت معکوس مرتب می‌کنیم.

از آنجایی که مقدار در ابتدا قرار دارد، برای مقایسه‌ها از آن استفاده خواهد شد. اگر بیش از یک تاپل مقدار یکسانی داشته باشند، برنامه به عنصر دوم (کلید) نگاه می‌کند، بنابراین تاپل‌هایی با مقدار یکسان به صورت الفبایی بر اساس کلید مرتب خواهند شد.

در پایان، یک حلقه‌ی for زیبا می‌نویسیم که از تکرار با انتساب چندگانه استفاده می‌کند و با پیمایش یک بخش از لیست (`lst[:10]`) ده واژه ی پرتکرار را چاپ می‌کند. اکنون خروجی برنامه دقیقاً همان چیزی است که برای تحلیل فراوانی واژه‌ها مدنظر داشتیم.

```
61 i
42 and
40 romeo
34 to
34 the
32 thou
32 juliet
30 that
29 my
24 thee
```

اینکه چنین پردازش و تحلیلی بر داده‌های پیچیده را می‌توان تنها با یک برنامه‌ی ۱۹ خطی ساده و قابل درک در پایتون انجام داد، یکی از دلایلی است که پایتون را به انتخابی مناسب برای کاوش در داده‌ها تبدیل می‌کند.

## ۱۰,۷ استفاده از تاپل به عنوان کلید در دیکشنری‌ها

از آن‌جا که تاپل‌ها قابل هش (hashable) هستند اما لیست‌ها نه، اگر بخواهیم یک کلید ترکیبی برای استفاده در دیکشنری بسازیم، باید از تاپل به عنوان کلید استفاده کنیم.

با یک کلید ترکیبی زمانی روبه‌رو می‌شویم که مثلاً بخواهیم یک دفترچه تلفن ایجاد کنیم که جفت‌های «نام خانوادگی، نام» را به شماره تلفن‌ها نگاشت کند. فرض کنیم متغیرهای last، first، و number را تعریف کرده‌ایم. در این صورت می‌توانیم یک عبارت انتساب به دیکشنری به شکل زیر بنویسیم:

```
directory[last,first] = number
```

عبارتی که در داخل کروشه‌ها قرار دارد، یک تاپل است. ما می‌توانیم از تخصیص تاپل (tuple assignment) در یک حلقه‌ی for استفاده کنیم تا در این دیکشنری پیمایش کنیم.

```
for last, first in directory:
    print(first, last, directory[last,first])
```

این حلقه از روی کلیدهای موجود در دیکشنری عبور می‌کند که این کلیدها تاپل هستند. در هر تکرار، عناصر هر تاپل به متغیرهای last و first اختصاص داده می‌شود و سپس نام کامل به همراه شماره تلفن مربوطه چاپ می‌گردد.

## ۱۰,۸ دنباله‌ها: رشته‌ها، لیست‌ها، و تاپل‌ها - اوه خدای من!

من تمرکز را روی لیست‌هایی از تاپل‌ها گذاشته‌ام، اما تقریباً تمام مثال‌های این فصل با لیست‌هایی از لیست‌ها، تاپل‌هایی از تاپل‌ها، و تاپل‌هایی از لیست‌ها نیز کار می‌کنند. برای اینکه نیاز به شمردن تمام ترکیب‌های ممکن نباشد، گاهی ساده‌تر است که فقط درباره‌ی "دنباله‌هایی از دنباله‌ها" صحبت کنیم.

در بسیاری از موقعیت‌ها، انواع مختلف دنباله‌ها (رشته‌ها، لیست‌ها، و تاپل‌ها) می‌توانند به‌جای یکدیگر استفاده شوند. پس چگونه و چرا باید یکی را به جای دیگری انتخاب کرد؟

برای شروع از نکته‌ی واضح، رشته‌ها نسبت به دیگر دنباله‌ها محدودتر هستند زیرا عناصر آن‌ها باید کاراکتر باشند. آن‌ها همچنین غیر قابل تغییر (immutable) هستند. اگر نیاز دارید بتوانید کاراکترهای درون یک رشته را تغییر دهید (نه اینکه رشته‌ای جدید بسازید)، ممکن است بخواهید به‌جای رشته، از یک لیست از کاراکترها استفاده کنید.

لیست‌ها رایج‌تر از تاپل‌ها هستند، عمدتاً چون قابل‌تغییر (mutable) هستند. اما در چند حالت ممکن است تاپل‌ها را ترجیح دهید:

۱ .در برخی موقعیت‌ها، مانند بازگرداندن مقدار در یک عبارت return، ایجاد یک تاپل از نظر نحوی ساده‌تر از یک لیست است. در موقعیت‌های دیگر ممکن است لیست را ترجیح دهید.

۲ .اگر بخواهید از یک دنباله به‌عنوان کلید در یک دیکشنری استفاده کنید، باید از نوعی غیرقابل‌تغییر مثل تاپل یا رشته استفاده کنید.

۳ .اگر بخواهید یک دنباله را به‌عنوان آرگومان به یک تابع بدهید، استفاده از تاپل‌ها احتمال بروز رفتارهای غیرمنتظره ناشی از اشاره‌گرهای مشترک (aliasing) را کاهش می‌دهد.

از آن‌جا که تاپل‌ها غیرقابل‌تغییرند، متدهایی مانند `sort` و `reverse` که لیست‌ها را تغییر می‌دهند، برای تاپل‌ها وجود ندارند. با این حال، پایتون توابع داخلی `sorted` و `reversed` را ارائه می‌دهد که هر دنباله‌ای را به‌عنوان ورودی می‌پذیرند و یک دنباله‌ی جدید با همان عناصر، ولی با ترتیبی متفاوت، بازمی‌گردانند.

## ۱۰,۹ درک لیست‌ها(List Comprehension)

گاهی می‌خواهید یک دنباله ایجاد کنید با استفاده از داده‌هایی که از دنباله‌ای دیگر گرفته شده‌اند. می‌توانید این کار را با نوشتن یک حلقه‌ی `for` انجام دهید و هر بار یک عنصر را به دنباله‌ی جدید اضافه کنید.

برای مثال، اگر بخواهید فهرستی از رشته‌ها – که هرکدام عددی را به صورت رشته در خود دارند – به عدد تبدیل کنید تا بتوانید مجموع آن‌ها را محاسبه کنید، می‌توانید چنین بنویسید:

```
list_of_ints_in_strings = ['42', '65', '12']
list_of_ints = []
for x in list_of_ints_in_strings:
    list_of_ints.append(int(x))

print(sum(list_of_ints))
```

با استفاده از لیست (List Comprehension)، کد بالا را می‌توان به شکلی فشرده‌تر و خواناتر نوشت:

```
list_of_ints_in_strings = ['42', '65', '12']
list_of_ints = [ int(x) for x in list_of_ints_in_strings ]
print(sum(list_of_ints))
```

## ۱۰,۱۰ اشکال‌زدایی(Debugging)

لیست‌ها، دیکشنری‌ها و تاپل‌ها به‌طور کلی با عنوان ساختارهای داده شناخته می‌شوند؛ در این فصل، ما با ساختارهای داده مرکب آشنا می‌شویم، مانند لیستی از تاپل‌ها یا دیکشنری‌هایی که تاپل به عنوان کلید و لیست به عنوان مقدار دارند. ساختارهای داده مرکب بسیار مفید هستند، اما مستعد نوعی خطا هستند که من آن را «خطای شکل» (shape error) می‌نامم؛ یعنی خطاهایی که ناشی از نوع، اندازه یا ترکیب اشتباه یک ساختار داده هستند، یا ممکن است شما کدی بنویسید و شکل ساختار داده‌تان را فراموش کنید و باعث بروز خطا شوید.

برای مثال، اگر انتظار داشته باشید که یک لیست شامل یک عدد صحیح دریافت کنید، اما به جای آن فقط یک عدد صحیح معمولی (و نه داخل لیست) به شما داده شود، کد شما کار نخواهد کرد.

## ۱۰,۱۱ واژه‌نامه

**قابل مقایسه(comparable)** : نوعی داده که در آن می‌توان بررسی کرد آیا یک مقدار از نوعی خاص بزرگ‌تر، کوچک‌تر یا مساوی مقدار دیگری از همان نوع است. انواع قابل مقایسه را می‌توان در یک لیست قرار داد و مرتب‌سازی کرد.

**ساختار داده(data structure)** : مجموعه‌ای از مقادیر مرتبط که معمولاً در قالب لیست‌ها، دیکشنری‌ها، تاپل‌ها و غیره سازمان‌دهی می‌شوند.

**DSU**: مخفف "تزئین، مرتب‌سازی، حذف تزئین(decorate-sort-undecorate) "، الگویی که شامل ساخت لیستی از تاپل‌ها، مرتب‌سازی آن، و استخراج بخشی از نتیجه است.

**جمع‌آوری(gather)** : عمل ترکیب آرگومان‌هایی با طول متغیر در قالب یک تاپل.

**دارای هش(hashable)** : نوعی که دارای تابع هش است. انواع غیرقابل تغییر مانند اعداد صحیح، اعشاری و رشته‌ها قابل هش شدن هستند؛ انواع قابل تغییر مانند لیست‌ها و دیکشنری‌ها قابل هش شدن نیستند.

**پراکنده‌سازی(scatter)** : عمل در نظر گرفتن یک دنباله به عنوان لیستی از آرگومان‌ها.

**شکل (shape) ساختار داده**: خلاصه‌ای از نوع، اندازه و ترکیب یک ساختار داده.

**تک‌عنصری(singleton)** : لیست (یا دنباله‌ای دیگر) که تنها شامل یک عنصر است.

**تاپل(tuple)** : دنباله‌ای غیرقابل تغییر از عناصر.

**تخصیص تاپلی(tuple assignment)** : تخصیصی که در آن یک دنباله در سمت راست و یک تاپل از متغیرها در سمت چپ قرار دارد. سمت راست ارزیابی شده و عناصر آن به متغیرهای سمت چپ اختصاص داده می‌شوند.

## ۱۰,۱۲ تمرین ها

**تمرین ۱:** برنامه‌ای را بازنویسی کنید که به شکل زیر باشد: خطوطی که با "From" شروع می‌شوند را بخوانید و آدرس‌های ایمیل را از آن خطوط استخراج کنید. تعداد پیام‌هایی را که از هر شخص آمده است با استفاده از یک دیکشنری بشمارید.

پس از خواندن تمام داده‌ها، فردی را که بیشترین کامیت (commit) را دارد چاپ کنید؛ به این صورت که از دیکشنری یک لیست از تاپل‌های (تعداد، ایمیل) ایجاد کنید. سپس این لیست را به صورت معکوس مرتب کنید و فردی را چاپ کنید که بیشترین تعداد کامیت را داشته است.

```
Sample Line:
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008

Enter a file name: mbox-short.txt
cwen@iupui.edu 5

Enter a file name: mbox.txt
zqian@umich.edu 195
```

**تمرین ۲:** این برنامه توزیع ساعتِ روز را برای هر یک از پیام‌ها می‌شمارد. می‌توانید ساعت را از خطی که با **"From"** شروع می‌شود استخراج کنید؛ به این صورت که ابتدا رشته زمان را پیدا کرده و سپس آن را با استفاده از نویسه دونقطه (:) به بخش‌های مختلف تقسیم کنید. پس از آنکه شمارش برای هر ساعت جمع‌آوری شد، شمارش‌ها را ــ هر کدام در یک خط ــ بر اساس ساعت و به ترتیب مرتب‌شده، مانند نمونه زیر چاپ کنید.

```
python timeofday.py
Enter a file name: mbox-short.txt
04 3
06 1
07 1
09 2
10 3
11 6
14 1
15 2
16 4
17 2
18 1
19 1
```

**تمرین ۳**: برنامه‌ای بنویسید که یک فایل را بخواند و حروف را بر اساس میزان فراوانی‌شان به ترتیب نزولی چاپ کند. برنامه شما باید تمام ورودی را به حروف کوچک تبدیل کند و فقط حروف **a** تا **z** را بشمارد. برنامه نباید فاصله‌ها، اعداد، علائم نگارشی یا هر چیز غیر از حروف a تا z را بشمارد. نمونه متن‌هایی از چند زبان مختلف پیدا کنید و ببینید که فراوانی حروف چگونه بین زبان‌ها تفاوت دارد. نتایج خود را با جدول‌های موجود در این صفحه مقایسه کنید:
https://wikipedia.org/wiki/Letter_frequencies

# فصل یازدهم

# عبارات منظم

تا به حال ما در حال خواندن فایل‌ها بودیم، به دنبال الگوها می‌گشتیم و بخش‌های مختلف خطوطی را که جالب می‌یافتیم استخراج می‌کردیم. از روش‌های رشته‌ای مانند `split` و `find` استفاده کرده و از لیست‌ها و برش رشته‌ها برای استخراج بخش‌هایی از خطوط استفاده کرده‌ایم.

این کار جستجو و استخراج آنقدر رایج است که پایتون یک ماژول بسیار قدرتمند به نام عبارات منظم (regular expressions) دارد که بسیاری از این وظایف را به شکلی بسیار زیبا و کارآمد انجام می‌دهد. دلیل اینکه ما عبارات منظم را زودتر در کتاب معرفی نکردیم این است که اگرچه آنها بسیار قدرتمند هستند، اما کمی پیچیده‌اند و نحو (syntax) آنها نیاز به کمی عادت دارد.

عبارات منظم تقریباً یک زبان برنامه‌نویسی مستقل برای جستجو و تجزیه رشته‌ها هستند. در واقع، کتاب‌های کاملی در مورد عبارات منظم نوشته شده است. در این فصل، ما فقط اصول پایه‌ای عبارات منظم را پوشش خواهیم داد. برای جزئیات بیشتر در مورد عبارات منظم، به موارد زیر مراجعه کنید:

https://en.wikipedia.org/wiki/Regular_expression

https://docs.python.org/library/re.html

ماژول عبارات منظم `re` باید قبل از استفاده به برنامه شما وارد شود. ساده‌ترین استفاده از ماژول عبارات منظم، تابع `search()` است. برنامه زیر یک استفاده ساده از تابع `search()` را نمایش می‌دهد.

```
# Search for lines that contain 'From'
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('From:', line):
        print(line)

# Code: https://www.py4e.com/code3/re01.py
```

ما فایل را باز می‌کنیم، در هر خط حلقه می‌زنیم و از تابع `search()` عبارات منظم برای چاپ تنها خطوطی که شامل رشته "From:" هستند استفاده می‌کنیم. این برنامه از قدرت واقعی عبارات منظم استفاده نمی‌کند، زیرا به راحتی می‌توانستیم از `line.find()` برای رسیدن به همان نتیجه استفاده کنیم.

قدرت عبارات منظم زمانی نمایان می‌شود که ما کاراکترهای ویژه‌ای را به رشته جستجو اضافه می‌کنیم که به ما اجازه می‌دهند دقیق‌تر کنترل کنیم که کدام خطوط با رشته مطابقت دارند. اضافه کردن این کاراکترهای ویژه به عبارات منظم ما را قادر می‌سازد تا تطبیق و استخراج پیچیده‌ای انجام دهیم در حالی که کد بسیار کمی می‌نویسیم.

برای مثال، کاراکتر caret (^) در عبارات منظم برای مطابقت با "ابتدای" یک خط استفاده می‌شود. ما می‌توانیم برنامه خود را تغییر دهیم تا فقط خطوطی که "From:" در ابتدای آن‌ها است را مطابقت دهد به این صورت:

```
# Search for lines that start with 'From'
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^From:', line):
        print(line)

# Code: https://www.py4e.com/code3/re02.py
```

حالا تنها خطوطی که با رشته "From:" شروع می‌شوند را مطابقت می‌دهیم. این هنوز یک مثال بسیار ساده است که می‌توانستیم به طور معادل با استفاده از متد `startswith()` از ماژول رشته‌ها انجام دهیم. اما این مثال به معرفی این مفهوم کمک می‌کند که عبارات منظم شامل کاراکترهای خاصی هستند که کنترل بیشتری روی مطابقت‌ها به ما می‌دهند.

## ۱۱,۱ تطبیق کاراکترها در عبارات منظم

تعدادی کاراکتر خاص دیگر وجود دارند که به ما این امکان را می‌دهند تا عبارات منظم قدرتمندتری بسازیم. رایج‌ترین کاراکتر خاص نقطه یا نقطه کامل (period) است که با هر کاراکتری مطابقت دارد.

در مثال زیر، عبارت منظم `F..m:` با هر یک از رشته‌های "From:" ، "Fxxm:"، "F12m:"، یا "F!@m:" مطابقت خواهد داشت چون کاراکترهای نقطه در عبارت منظم با هر کاراکتری مطابقت دارند.

```
# Search for lines that start with 'F', followed by
# 2 characters, followed by 'm:'
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^F..m:', line):
        print(line)

# Code: https://www.py4e.com/code3/re03.py
```

این ویژگی به ویژه زمانی قدرت بیشتری پیدا می‌کند که با قابلیت نشان دادن اینکه یک کاراکتر می‌تواند هر تعداد دفعات تکرار شود، ترکیب شود. با استفاده از کاراکترهای * یا + در عبارت منظم خود، می‌توانیم مشخص کنیم که به جای مطابقت با یک کاراکتر واحد در رشته جستجو، آن‌ها با صفر یا بیشتر از کاراکترها (در صورت استفاده از ستاره) یا یک یا بیشتر از کاراکترها (در صورت استفاده از علامت بعلاوه) مطابقت دارند.

ما می‌توانیم با استفاده از کاراکتر wild card تکراری، دامنه خطوطی که مطابقت می‌دهیم را بیشتر محدود کنیم. در مثال زیر:

```
# Search for lines that start with From and have an at sign
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^From:.+@', line):
        print(line)

# Code: https://www.py4e.com/code3/re04.py
```

رشته جستجو ^From:.+@ موفقیت‌آمیز خطوطی را که با "From:" شروع می‌شوند، دنبال یک یا بیشتر کاراکتر (.+)، و سپس یک علامت @ مطابقت می‌دهد. بنابراین این رشته با خط زیر مطابقت خواهد داشت

```
From: stephen.marquard@uct.ac.za
```

می‌توانید به wild card .+ به عنوان گسترش دهنده‌ای فکر کنید که تمام کاراکترها را بین کاراکتر کولن و علامت @ مطابقت می‌دهد.

```
From:.+@
```

خوب است که کاراکترهای بعلاوه و ستاره را به‌عنوان "پرخاشگر (pushy)" در نظر بگیرید. به‌عنوان مثال، رشته زیر آخرین علامت @ در رشته را مطابقت خواهد داد، زیرا .+ به‌طور پیوسته به سمت بیرون فشار می‌آورد، همانطور که در زیر نشان داده شده است:

```
From: stephen.marquard@uct.ac.za, csev@umich.edu, and cwen @iupui.edu
```

می‌توان با اضافه کردن کاراکتر دیگر به علامت ستاره یا بعلاوه، از رفتار "حریصانه (greedy)" آن‌ها جلوگیری کرد. برای اطلاعات بیشتر در مورد غیرفعال کردن این رفتار حریصانه، به مستندات دقیق‌تر مراجعه کنید.

## ۱۱,۲ استخراج داده‌ها با استفاده از عبارات منظم

اگر بخواهیم داده‌ای را از یک رشته در پایتون استخراج کنیم، می‌توانیم از متد `findall()` استفاده کنیم تا تمام زیر رشته‌هایی که با یک عبارت منظم مطابقت دارند، استخراج شوند. بیایید از مثال استخراج هر چیزی که شبیه به یک آدرس ایمیل است از هر خطی، بدون توجه به قالب آن استفاده کنیم. به عنوان مثال، می‌خواهیم آدرس‌های ایمیل را از هر یک از خطوط زیر استخراج کنیم:

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
Return-Path: <postmaster@collab.sakaiproject.org>
          for <source@collab.sakaiproject.org>;
Received: (from apache@localhost)
Author: stephen.marquard@uct.ac.za
```

ما نمی‌خواهیم برای هر نوع خط کد بنویسیم و به‌طور مختلف برای هر خط از تقسیم و برش استفاده کنیم. برنامه زیر از `findall()` برای یافتن خطوطی که حاوی آدرس‌های ایمیل هستند و استخراج یک یا چند آدرس از هر یک از آن خطوط استفاده می‌کند .

```
import re
s = 'A message from csev@umich.edu to cwen@iupui.edu about meeting @2PM'
lst = re.findall(r'\S+@\S+', s)
print(lst)

# Code: https://www.py4e.com/code3/re05.py
```

متد `findall()` رشته را در آرگومان دوم جستجو می‌کند و فهرستی از تمام رشته‌هایی که شبیه به آدرس‌های ایمیل هستند، برمی‌گرداند. ما از یک توالی دوکاراکتری که یک کاراکتر غیر فاصله‌ای را مطابقت می‌دهد (\S) استفاده می‌کنیم.

خروجی برنامه به این شکل خواهد بود:

```
['csev@umich.edu', 'cwen@iupui.edu']
```

با ترجمه عبارت منظم، ما به دنبال زیر رشته‌هایی هستیم که حداقل یک کاراکتر غیر فاصله‌ای داشته باشند، سپس یک علامت @، و پس از آن حداقل یک کاراکتر غیر فاصله‌ای دیگر. عبارت `\S+` به تعداد ممکن کاراکترهای غیر فاصله‌ای را تطبیق می‌دهد.

عبارت منظم دو بار مطابقت خواهد کرد (برای "csev@umich.edu" و cwen@iupui.edu )، اما برای رشته "@۲PM" مطابقت نخواهد کرد، زیرا هیچ کاراکتر غیر فاصله‌ای قبل از علامت @ وجود ندارد. ما می‌توانیم از این عبارت منظم در برنامه‌ای برای خواندن تمام خطوط یک فایل و چاپ هر چیزی که شبیه به آدرس ایمیل باشد، به شکل زیر استفاده کنیم:

```
# Search for lines that have an at sign between characters
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall(r'\S+@\S+', line)
    if len(x) > 0:
        print(x)

# Code: https://www.py4e.com/code3/re06.py
```

ما هر خط را می‌خوانیم و سپس تمام زیررشته‌هایی که با عبارت منظم ما مطابقت دارند را استخراج می‌کنیم. از آنجا که متد `findall()` یک لیست برمی‌گرداند، ما فقط بررسی می‌کنیم که آیا تعداد عناصر موجود در لیست بازگشتی بیشتر از صفر است یا خیر، تا تنها خطوطی را چاپ کنیم که حداقل یک زیررشته شبیه به آدرس ایمیل پیدا کرده‌ایم.

اگر برنامه را روی فایل mbox-short.txt اجرا کنیم، خروجی به شکل زیر خواهد بود:

```
...
['<source@collab.sakaiproject.org>;']
['<source@collab.sakaiproject.org>;']
['apache@localhost)']
['source@collab.sakaiproject.org;']
['cwen@iupui.edu']
['source@collab.sakaiproject.org']
['cwen@iupui.edu']
['cwen@iupui.edu']
['wagnermr@iupui.edu']
```

برای حل مشکل کاراکترهای اشتباهی مانند "<" یا ";" که در ابتدای یا انتهای برخی از آدرس‌های ایمیل وجود دارند، می‌خواهیم فقط بخشی از رشته را که با یک حرف یا عدد شروع می‌شود و با یک حرف یا عدد تمام می‌شود استخراج کنیم.

برای این کار، از ویژگی دیگری در عبارات منظم استفاده می‌کنیم. پرانتزهای مربعی برای نشان دادن مجموعه‌ای از کاراکترهای قابل قبول استفاده می‌شوند که مایل به تطابق با آن‌ها هستیم. به نوعی، \S از ما می‌خواهد که مجموعه‌ای از "کاراکترهای غیر فضای سفید" را تطابق بدهیم. حالا می‌خواهیم کمی بیشتر مشخص کنیم که کدام کاراکترها را می‌خواهیم تطابق دهیم.

در اینجا عبارت منظم جدید ما را مشاهده می کنید:

```
[a-zA-Z0-9]\S*@\S*[a-zA-Z]
```

این موضوع کمی پیچیده‌تر می‌شود و می‌توانید شروع به مشاهده کنید که چرا عبارات منظم به‌طور خود به خودی به یک زبان خاص خود تبدیل می‌شوند. ترجمه این عبارت منظم، به این صورت است که ما به دنبال زیر رشته‌هایی هستیم که با یک حرف کوچک، حرف بزرگ یا عدد "[a-zA-Z0-9]" شروع می‌شوند، سپس صفر یا بیشتر از کاراکترهای غیر فضای سفید (\S*) قرار دارند، بعد یک علامت "@" آمده است، و سپس صفر یا بیشتر از کاراکترهای غیر فضای سفید (\S*) آمده و در نهایت یک حرف بزرگ یا کوچک وجود دارد. توجه داشته باشید که ما از + به * تغییر داده‌ایم تا نشان‌دهنده "صفر یا بیشتر از کاراکترهای غیر سفید" باشد، زیرا [a-zA-Z0-9] خود یک کاراکتر غیر سفید است. به یاد داشته باشید که * یا + فقط به کاراکتر بلافاصله سمت چپ آن‌ها اعمال می‌شود.

اگر از این عبارت در برنامه خود استفاده کنیم، داده‌های ما خیلی تمیزتر خواهند بود.

```
# Search for lines that have an at sign between characters
# The characters must be a letter or number
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall(r'[a-zA-Z0-9]\S*@\S*[a-zA-Z]', line)
    if len(x) > 0:
        print(x)
```

```
# Code: https://www.py4e.com/code3/re07.py
...
['wagnermr@iupui.edu']
['cwen@iupui.edu']
['postmaster@collab.sakaiproject.org']
['200801032122.m03LMFo4005148@nakamura.uits.iupui.edu']
['source@collab.sakaiproject.org']
['source@collab.sakaiproject.org']
['source@collab.sakaiproject.org']
['apache@localhost']
```

توجه داشته باشید که در خطوط source@collab.sakaiproject.org، عبارت منظم ما دو حرف در انتهای رشته (";>") را حذف کرده است. این به این دلیل است که وقتی ما [a-zA-Z] را به انتهای عبارت منظم خود اضافه می‌کنیم، می‌خواهیم که هر رشته‌ای که پارسر عبارت منظم پیدا می‌کند، باید با یک حرف تمام شود. بنابراین زمانی که ">" را در انتهای sakaiproject.org>; می‌بیند، به سادگی در آخرین حرف "مطابقت" یافته‌ای که پیدا کرده متوقف می‌شود (یعنی، "g" آخرین تطابق صحیح بود). همچنین توجه داشته باشید که خروجی برنامه یک لیست پایتون است که رشته‌ای را به عنوان تنها عنصر در آن لیست دارد.

## ۱۱,۳ ترکیب جستجو و استخراج

اگر بخواهیم اعداد را در خطوطی که با رشته "X-" شروع می‌شوند پیدا کنیم، مانند:

```
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000
```

ما فقط نمی‌خواهیم هر عدد اعشاری را از هر خط استخراج کنیم. ما فقط می‌خواهیم اعداد را از خطوطی استخراج کنیم که دارای نحو فوق باشند. ما می‌توانیم عبارت منظم زیر را بسازیم تا خطوط مورد نظر را انتخاب کنیم:

```
^X-.*: [0-9.]+
```

ترجمه این عبارت به این معنی است که ما می‌خواهیم خطوطی که با X- شروع می‌شوند، دنبال کنیم. سپس، صفر یا بیشتر کاراکتر (.*) بعد از آن آمده، سپس دو نقطه (:) و یک فاصله می‌آید. پس از آن، ما به دنبال یک یا چند کاراکتر هستیم که یا یک عدد (۰-۹) یا یک نقطه (.) هستند، که این بخش با [0-9.]+ مشخص می‌شود. توجه داشته باشید که داخل براکت‌های مربعی، نقطه یک نقطه واقعی را مطابقت می‌دهد (یعنی در داخل براکت‌ها نقطه یک wildcard نیست).

این یک عبارت منظم بسیار دقیق است که تقریباً فقط خطوطی را که به دنبال آن‌ها هستیم، مطابقت می‌دهد.

```
# Search for lines that start with 'X' followed by any non
# whitespace characters and ':'
# followed by a space and any number.
# The number can include a decimal.
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search(r'^X\S*: [0-9.]+', line):
        print(line)

# Code: https://www.py4e.com/code3/re10.py
```

وقتی برنامه را اجرا می‌کنیم، داده‌ها به خوبی فیلتر می‌شوند و فقط خطوطی که به دنبالشان هستیم نمایش داده می‌شوند.

```
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000
X-DSPAM-Confidence: 0.6178
X-DSPAM-Probability: 0.0000
...
```

اما حالا باید مشکل استخراج اعداد را حل کنیم. در حالی که استفاده از متد `split` کار ساده‌ای است، می‌توانیم از ویژگی دیگری از عبارات منظم برای جستجو و تجزیه‌ی خط به طور همزمان استفاده کنیم.

پرانتزها یکی دیگر از کاراکترهای ویژه در عبارات منظم هستند. زمانی که پرانتزها را به یک عبارت منظم اضافه می‌کنید، آنها در هنگام تطبیق رشته نادیده گرفته می‌شوند. اما زمانی که از `findall()` استفاده می‌کنید، پرانتزها نشان می‌دهند که در حالی که می‌خواهید کل عبارت تطبیق داده شود، فقط علاقه‌مند به استخراج بخشی از زیررشته‌ای هستید که با عبارت منظم مطابقت دارد.

بنابراین تغییرات زیر را در برنامه خود ایجاد می‌کنیم:

```
# Search for lines that start with 'X' followed by any
# non whitespace characters and ':' followed by a space
# and any number. The number can include a decimal.
# Then print the number if it is greater than zero.
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall(r'^X\S*: ([0-9.]+)', line)
    if len(x) > 0:
        print(x)

# Code: https://www.py4e.com/code3/re11.py
```

به جای فراخوانی متد `search()`، پرانتزهایی را در اطراف قسمتی از عبارت منظم که نشان‌دهنده عدد اعشاری است قرار می‌دهیم تا به `findall()` بگوییم فقط بخش عدد اعشاری از رشته تطبیق یافته را به ما بازگرداند. خروجی این برنامه به شکل زیر است:

```
['0.8475']
['0.0000']
['0.6178']
['0.0000']
['0.6961']
['0.0000']
...
```

اعداد هنوز در یک لیست هستند و باید از رشته به اعداد اعشاری تبدیل شوند، اما ما از قدرت عبارات منظم برای جستجو و استخراج اطلاعاتی که برایمان جالب بود استفاده کرده‌ایم. به عنوان مثال دیگر از این تکنیک، اگر به فایل نگاه کنید، تعدادی خط به شکل زیر وجود دارد:

```
Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
```

اگر بخواهیم تمام شماره‌های اصلاح (عدد صحیح در انتهای این خطوط) را با استفاده از همان تکنیک که در بالا ذکر شد استخراج کنیم، می‌توانیم برنامه زیر را بنویسیم:

```
# Search for lines that start with 'Details: rev='
# followed by numbers
# Then print the number if one is found
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('^Details:.*rev=([0-9]+)', line)
    if len(x) > 0:
        print(x)

# Code: https://www.py4e.com/code3/re12.py
```

ترجمه عبارت منظم ما این است که به دنبال خطوطی هستیم که با کلمه "Details:" شروع می‌شوند، سپس هر تعداد کاراکتر (.*) آمده و بعد از آن "rev=" قرار دارد و سپس یک یا چند عدد وجود دارد. ما می‌خواهیم خطوطی که با این عبارت منظم مطابقت دارند را پیدا کنیم، اما فقط عدد صحیح انتهای خط را استخراج کنیم، بنابراین [۰-۹]+ را در داخل پرانتز قرار می‌دهیم.

هنگامی که برنامه را اجرا می‌کنیم، خروجی زیر را دریافت می‌کنیم:

```
['39772']
['39771']
['39770']
['39769']
...
```

به یاد داشته باشید که عبارت `[0-9]+` به صورت "حریصانه" عمل می‌کند و سعی می‌کند بزرگ‌ترین رشته‌ای از اعداد را که ممکن است استخراج کند، قبل از این که آن اعداد را استخراج کند. این رفتار "حریصانه" است که باعث می‌شود تمام پنج رقم را برای هر عدد دریافت کنیم. ماژول عبارات منظم در هر دو جهت گسترش می‌یابد تا زمانی که به یک کاراکتر غیر عددی یا شروع یا پایان خط برخورد کند. حال می‌توانیم از عبارات منظم برای تکرار یک تمرین از بخش‌های قبلی کتاب استفاده کنیم که در آن به زمان روز هر پیام ایمیل علاقه داشتیم. ما به دنبال خطوطی از این دست می‌گشتیم:

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
```

و می‌خواستیم ساعت روز را برای هر خط استخراج کنیم. قبلاً این کار را با دو فراخوانی به متد `split` انجام دادیم. ابتدا خط به کلمات تقسیم می‌شد و سپس کلمه پنجم استخراج می‌شد و دوباره بر اساس کاراکتر دو نقطه (colon) تقسیم می‌شد تا دو کاراکتری که به آن‌ها نیاز داشتیم استخراج شوند. در حالی که این روش کار می‌کند، در واقع منجر به کدی شکننده می‌شود که فرض می‌کند خطوط به درستی فرمت شده‌اند. اگر بخواهید مقدار زیادی چک خطا (یا یک بلوک try/except بزرگ) اضافه کنید تا اطمینان حاصل کنید که برنامه شما هرگز زمانی که خطوط نادرست فرمت شده‌اند شکست نمی‌خورد، کد به ۱۵-۱۰ خط خواهد رسید که خواندن آن سخت است.

ما می‌توانیم این کار را به روشی بسیار ساده‌تر با استفاده از عبارت منظم زیر انجام دهیم:

```
^From .* [0-9][0-9]:
```

ترجمه این عبارت منظم این است که ما به دنبال خطوطی هستیم که با کلمه "From" ( توجه داشته باشید که فاصله وجود دارد) شروع شوند، سپس هر تعداد کاراکتر (.*) دنبال شوند، سپس یک فاصله، سپس دو رقم [۰-۹][۰-۹] بیایند، و در نهایت کاراکتر دو نقطه (colon) بیاید. این تعریف نوع خطوطی است که ما به دنبال آن‌ها هستیم.

برای استخراج فقط ساعت با استفاده از `findall()`، پرانتزهایی اطراف دو رقم می‌زنیم به شکل زیر:

```
^From .* ([0-9][0-9]):
```

این به برنامه زیر منجر می‌شود:

```
# Search for lines that start with From and a character
# followed by a two digit number between 00 and 99 followed by ':'
# Then print the number if one is found
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('^From .* ([0-9][0-9]):', line)
    if len(x) > 0: print(x)

# Code: https://www.py4e.com/code3/re13.py
```

لطفاً متن مورد نظر را برای ترجمه ارسال کنید.

```
['09']
['18']
['16']
['15']
...
```

## ۱۱,۴ ویژگی فرار(Escape character)

از آنجا که ما از کاراکترهای ویژه در عبارات منظم برای مطابقت با آغاز یا پایان خط یا تعیین کاراکترهای wildcard استفاده می‌کنیم، نیاز داریم که روشی برای نشان دادن این که این کاراکترها "عادی" هستند و می‌خواهیم خود کاراکتر واقعی مانند علامت دلار یا caret را مطابقت دهیم، داشته باشیم. ما می‌توانیم نشان دهیم که می‌خواهیم فقط یک کاراکتر خاص را مطابقت دهیم، با پیشوند دادن آن کاراکتر با یک بک‌اسلش. به عنوان مثال، می‌توانیم مقادیر پولی را با عبارت منظم زیر پیدا کنیم.

```
import re
x = 'We just received $10.00 for cookies.'
y = re.findall('\$[0-9.]+',x)
```

از آنجا که ما علامت دلار را با بک‌اسلش پیشوند می‌دهیم، در واقع علامت دلار را در رشته ورودی مطابقت می‌دهیم به جای آن که با "پایان خط" مطابقت داشته باشد. باقی‌مانده عبارت منظم یک یا بیشتر از ارقام یا کاراکتر نقطه را مطابقت می‌دهد. نکته: داخل براکت‌های مربع، کاراکترها "ویژه" نیستند. بنابراین وقتی می‌گوییم `[0-9.]`، این واقعاً به معنی ارقام یا یک نقطه است. خارج از براکت‌های مربع، نقطه کاراکتر "واحد-نشان" است و هر کاراکتری را مطابقت می‌دهد. داخل براکت‌ها، نقطه واقعاً یک نقطه است.

## ۱۱,۵ خلاصه

اگرچه این فقط سطحی از عبارات منظم را بررسی کرد، اما ما کمی در مورد زبان عبارات منظم آموخته‌ایم. آن‌ها رشته‌های جستجو با کاراکترهای ویژه‌ای هستند که خواسته‌های شما را به سیستم عبارت منظم منتقل می‌کنند تا مشخص کنند که «مطابقت» چیست و چه چیزی از رشته‌های مطابقت داده شده استخراج می‌شود. در اینجا برخی از این کاراکترها و دنباله‌های کاراکتری آورده شده است:

^شروع خط را مطابقت می‌دهد.
$پایان خط را مطابقت می‌دهد.
.هر کاراکتری را مطابقت می‌دهد.(wildcard)
\sیک کاراکتر فاصله‌گذاری (whitespace) را مطابقت می‌دهد.
\Sیک کاراکتر غیر فاصله‌گذاری (غیر \s) را مطابقت می‌دهد.

به کاراکترهای پیشین اعمال می‌شود و نشان می‌دهد که باید صفر یا بیشتر بار مطابقت داده شود.
?* به کاراکترهای پیشین اعمال می‌شود و نشان می‌دهد که باید صفر یا بیشتر بار در «حالت غیر حریص» مطابقت داده شود.

به کاراکترهای پیشین اعمال می‌شود و نشان می‌دهد که باید یک یا بیشتر بار مطابقت داده شود.
?+ به کاراکترهای پیشین اعمال می‌شود و نشان می‌دهد که باید یک یا بیشتر بار در «حالت غیر حریص» مطابقت داده شود.
? به کاراکترهای پیشین اعمال می‌شود و نشان می‌دهد که باید صفر یا یک بار مطابقت داده شود.
?? به کاراکترهای پیشین اعمال می‌شود و نشان می‌دهد که باید صفر یا یک بار در «حالت غیر حریص» مطابقت داده شود.
[aeiou] یک کاراکتر را مطابقت می‌دهد به شرطی که آن کاراکتر در مجموعه مشخص شده باشد. در این مثال، این کاراکتر می‌تواند«a» ، «e»، «i»، «o»، یا «u» باشد، اما هیچ کاراکتر دیگری را مطابقت نمی‌دهد.
[a-z0-9] می‌توانید رنج‌هایی از کاراکترها را با استفاده از خط تیره مشخص کنید. این مثال یک کاراکتر است که باید یک حرف کوچک یا یک عدد باشد.
[^A-Za-z] وقتی که کاراکتر اول در نماد مجموعه یک caret باشد، منطق معکوس می‌شود. این مثال یک کاراکتر را مطابقت می‌دهد که هر چیزی غیر از حرف بزرگ یا کوچک باشد.
( )زمانی که پرانتز به یک عبارت منظم اضافه می‌شود، برای مطابقت نادیده گرفته می‌شود، اما به شما این امکان را می‌دهد که یک زیرمجموعه خاص از رشته مطابقت داده شده را استخراج کنید نه کل رشته را، زمانی که از ()findall استفاده می‌کنید.
\bرشته خالی را مطابقت می‌دهد، اما فقط در شروع یا پایان یک کلمه.
\Bرشته خالی را مطابقت می‌دهد، اما نه در شروع یا پایان یک کلمه.
\dهر عدد ده‌دهی را مطابقت می‌دهد؛ معادل مجموعه [۰-۹]

\Dهر کاراکتر غیر عددی را مطابقت می‌دهد؛ معادل مجموعه [^۰-۹]

## ۱۱,۶ بخش اضافی برای کاربران یونیکس / لینوکس

پشتیبانی از جستجوی فایل‌ها با استفاده از عبارات منظم از دهه ۱۹۶۰ در سیستم‌عامل یونیکس ساخته شده است و تقریباً در تمام زبان‌های برنامه‌نویسی به نوعی در دسترس است.

در حقیقت، یک برنامه خط فرمان به نام ( `grep` مخفف"Generalized Regular Expression Parser" ) در یونیکس وجود دارد که تقریباً همان کاری را که مثال‌های تابع `search()` در این فصل انجام می‌دهند، انجام می‌دهد. بنابراین، اگر سیستم‌عامل شما مکینتاش یا لینوکس باشد، می‌توانید دستورات زیر را در پنجره خط فرمان خود امتحان کنید.

```
$ grep '^From:' mbox-short.txt
From: stephen.marquard@uct.ac.za
From: louis@media.berkeley.edu
```

```
From: zqian@umich.edu
From: rjlowe@iupui.edu
```

این دستور به grep می‌گوید که خطوطی را که با رشته "From:" شروع می‌شوند در فایل mbox-short.txt نشان دهد. اگر کمی با دستور grep آزمایش کنید و مستندات grep را بخوانید، متوجه خواهید شد که تفاوت‌های جزئی بین پشتیبانی از عبارات منظم در Python و grep وجود دارد. به عنوان مثال، grep از کاراکتر غیر خالی \S پشتیبانی نمی‌کند، بنابراین باید از نوتیشن مجموعه‌ای کمی پیچیده‌تر [ ^ ] استفاده کنید که به سادگی به این معنی است که کاراکتری را پیدا کنید که هیچ چیز جز فضای خالی نباشد.

## ۱۱,۷ اشکال‌زدایی

پایتون مستندات ساده و ابتدایی‌ای دارد که می‌تواند در مواقعی که نیاز به یادآوری سریع اسم دقیق یک متد خاص دارید، بسیار مفید باشد. این مستندات را می‌توان در مفسر پایتون در حالت تعاملی مشاهده کرد. شما می‌توانید از سیستم راهنمای تعاملی با استفاده از دستور help() استفاده کنید.

```
>>> help()

help> modules
```

اگر می‌دانید که می‌خواهید از کدام ماژول استفاده کنید، می‌توانید از دستور dir() برای پیدا کردن متدهای موجود در آن ماژول به شرح زیر استفاده کنید:

```
>>> import re
>>> dir(re)
[.. 'compile', 'copy_reg', 'error', 'escape', 'findall',
'finditer', 'match', 'purge', 'search', 'split', 'sre_compile',
'sre_parse', 'sub', 'subn', 'sys', 'template']
```

شما همچنین می‌توانید با استفاده از دستور help به همراه متد مورد نظر، مقداری مستندات مختصر در مورد یک متد خاص به دست آورید.

```
>>> help (re.search)
Help on function search in module re:

search(pattern, string, flags=0)
    Scan through string looking for a match to the pattern, returning
    a match object, or None if no match was found.
>>>
```

مستندات داخلی خیلی گسترده نیست، اما زمانی که عجله دارید یا به مرورگر وب یا موتور جستجو دسترسی ندارید، می‌تواند مفید باشد.

## ۱۱,۸ واژه‌نامه

**کد شکننده**: کدی که زمانی که داده ورودی در یک فرمت خاص باشد، کار می‌کند، اما اگر از فرمت صحیح منحرف شود، به راحتی خراب می‌شود. به این کد "شکننده" گفته می‌شود زیرا به راحتی خراب می‌شود.

**مطابقت حریصانه**: مفهوم اینکه کاراکترهای + و * در یک عبارت منظم به‌طور گسترده‌ای به بیرون گسترش می‌یابند تا بزرگترین رشته ممکن را مطابقت دهند.

**Grep**: یک فرمان موجود در اکثر سیستم‌های یونیکس که در فایل‌های متنی جستجو می‌کند تا خطوطی که با عبارات منظم مطابقت دارند را پیدا کند. نام این فرمان مخفف "Generalized Regular Expression Parser" است.

**عبارت منظم**: زبانی برای بیان رشته‌های جستجوی پیچیده‌تر. یک عبارت منظم ممکن است شامل کاراکترهای خاصی باشد که نشان می‌دهند جستجو تنها در ابتدای یا انتهای یک خط مطابقت دارد یا ویژگی‌های مشابه دیگر.

**وایلدکارد**: یک کاراکتر خاص که با هر کاراکتری مطابقت دارد. در عبارات منظم، کاراکتر وایلدکارد نقطه (.) است.

## ۱۱,۹ تمرین ها

**تمرین ۱:** یک برنامه‌ی ساده بنویسید تا عملکرد دستور **grep** در یونیکس را شبیه‌سازی کند. از کاربر بخواهید یک الگوی عبارت منظم (regular expression) وارد کند و سپس تعداد خطوطی را که با این عبارت منظم تطابق دارند، شمارش کند.

```
$ python grep.py
Enter a regular expression: ^Author
mbox.txt had 1798 lines that matched ^Author

$ python grep.py
Enter a regular expression: ^X-
mbox.txt had 14368 lines that matched ^X-

$ python grep.py
Enter a regular expression: java$
mbox.txt had 4175 lines that matched java$
```

**تمرین ۲:** برنامه‌ای بنویسید که به دنبال خطوطی با این فرم بگردد:

```
New Revision: 39772
```

عدد را از هر یک از خطوط با استفاده از یک عبارت منظم و متد ()findall استخراج کنید. میانگین این اعداد را محاسبه کرده و میانگین را به‌صورت یک عدد صحیح چاپ کنید.

```
Enter file:mbox.txt
38549

Enter file:mbox-short.txt
39756
```

# فصل دوازدهم

# برنامه‌های شبکه‌ای

در حالی که بسیاری از مثال‌های این کتاب بر روی خواندن فایل‌ها و جستجو برای داده‌ها در آن‌ها تمرکز کرده است، منابع مختلفی برای اطلاعات وجود دارد زمانی که به اینترنت فکر می‌کنیم. در این فصل ما به عنوان یک مرورگر وب عمل خواهیم کرد و صفحات وب را با استفاده از پروتکل انتقال ابرمتن (HTTP) بازیابی خواهیم کرد. سپس داده‌های صفحه وب را خوانده و آن‌ها را تجزیه و تحلیل خواهیم کرد.

## ۱۲,۱ پروتکل انتقال ابرمتن HTTP

پروتکل شبکه‌ای که وب را راه‌اندازی می‌کند در واقع بسیار ساده است و در پایتون پشتیبانی داخلی به نام سوکت وجود دارد که این امکان را می‌دهد تا ارتباطات شبکه‌ای را برقرار کرده و داده‌ها را از طریق این سوکت‌ها در یک برنامه پایتون بازیابی کنیم. سوکت مشابه یک فایل است، با این تفاوت که یک سوکت واحد یک ارتباط دوطرفه بین دو برنامه فراهم می‌کند. شما می‌توانید هم از سوکت بخوانید و هم به آن بنویسید. اگر چیزی به سوکت بنویسید، به برنامه‌ای که در طرف دیگر سوکت قرار دارد ارسال می‌شود. اگر از سوکت بخوانید، داده‌هایی که برنامه دیگر ارسال کرده است را دریافت می‌کنید.

اما اگر سعی کنید از سوکت بخوانید و برنامه در طرف دیگر سوکت هیچ داده‌ای ارسال نکرده باشد، شما فقط منتظر می‌مانید. اگر برنامه‌ها در هر دو انتهای سوکت فقط منتظر داده‌ای بمانند بدون اینکه چیزی ارسال کنند، برای مدت طولانی منتظر خواهند ماند، بنابراین بخش مهمی از برنامه‌هایی که از طریق اینترنت با یکدیگر ارتباط برقرار می‌کنند، داشتن نوعی پروتکل است. پروتکل مجموعه‌ای از قواعد دقیق است که مشخص می‌کند چه کسی ابتدا باید اقدام کند، چه کاری باید انجام دهد، پاسخ‌ها به آن پیام چه هستند و چه کسی باید بعد از آن اقدام کند و به همین ترتیب. به نوعی، دو برنامه در هر دو انتهای سوکت در حال انجام یک رقص هستند و اطمینان حاصل می‌کنند که روی پای یکدیگر پا نمی‌گذارند.

اسناد زیادی وجود دارند که این پروتکل‌های شبکه‌ای را توصیف می‌کنند. پروتکل انتقال ابرمتن در سند زیر توضیح داده شده است:

https://www.w3.org/Protocols/rfc2616/rfc2616.txt

این یک سند طولانی و پیچیده به حجم ۱۷۶ صفحه است که جزئیات زیادی دارد. اگر این موضوع برای شما جالب است، می‌توانید همه آن را بخوانید. اما اگر به صفحه ۳۶ از RFC2616 نگاه کنید، خواهید دید که نحو درخواست GET توضیح داده شده است. برای درخواست یک سند از یک سرور وب، ما یک اتصال برقرار می‌کنیم، به عنوان مثال به سرور www.pr4e.orgدر پورت ۸۰، و سپس یک خط از این نوع را ارسال می‌کنیم:

```
GET http://data.pr4e.org/romeo.txt HTTP/1.0
```

جایی که پارامتر دوم صفحه وبی است که ما درخواست می‌کنیم، و سپس یک خط خالی ارسال می‌کنیم. سرور وب با اطلاعات هدر مربوط به سند و یک خط خالی پاسخ می‌دهد و سپس محتوای سند را ارسال می‌کند.

## ۱۲,۲ ساده‌ترین مرورگر وب در جهان

شاید ساده‌ترین راه برای نشان دادن نحوه عملکرد پروتکل HTTP این باشد که یک برنامه پایتون بسیار ساده بنویسیم که به یک سرور وب متصل شود و طبق قوانین پروتکل HTTP یک سند درخواست کند و آنچه را که سرور باز می‌گرداند نمایش دهد.

```
import socket

mysock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
mysock.connect(('data.pr4e.org', 80))
cmd = 'GET http://data.pr4e.org/romeo.txt HTTP/1.0\r\n\r\n'.encode()
mysock.send(cmd)

while True:
    data = mysock.recv(512)
    if len(data) < 1:
        break
    print(data.decode(),end='')

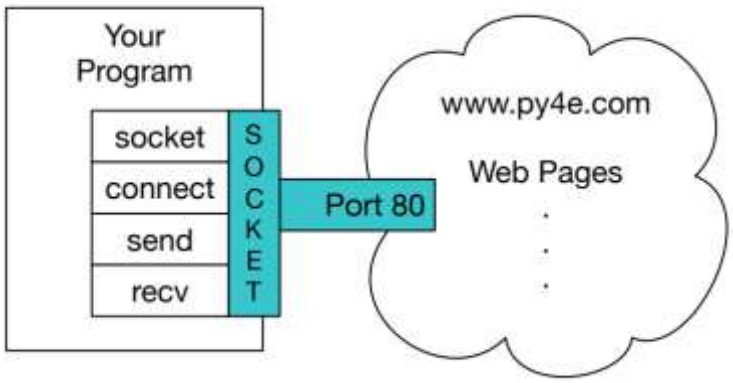
mysock.close()

# Code: https://www.py4e.com/code3/socket1.py
```

ابتدا برنامه یک اتصال به پورت ۸۰ سرور www.pr4e.comبرقرار می‌کند. از آنجا که برنامه ما نقش "مرورگر وب" را ایفا می‌کند، پروتکل HTTP می‌گوید که ما باید دستور GET را ارسال کنیم و سپس یک خط خالی ارسال نماییم \r\n. نشان‌دهنده پایان خط (EOL) است، بنابراین \r\n\r\n نشان‌دهنده یک خط خالی بین دو ترتیب EOL می‌باشد. این معادل یک خط خالی است.

پس از ارسال آن خط خالی، یک حلقه می‌نویسیم که داده‌ها را به صورت قطعات ۵۱۲ کاراکتری از سوکت دریافت کرده و داده‌ها را تا زمانی که دیگر داده‌ای برای خواندن نباشد (یعنی تابع recv() یک رشته خالی بازگرداند) چاپ می‌کند. این برنامه خروجی زیر را تولید می‌کند:

```
HTTP/1.1 200 OK
```




```
Date: Wed, 11 Apr 2018 18:52:55 GMT
Server: Apache/2.4.7 (Ubuntu)
Last-Modified: Sat, 13 May 2017 11:22:22 GMT
ETag: "a7-54f6609245537"
Accept-Ranges: bytes
Content-Length: 167
Cache-Control: max-age=0, no-cache, no-store, must-revalidate
Pragma: no-cache
Expires: Wed, 11 Jan 1984 05:00:00 GMT
Connection: close
Content-Type: text/plain

But soft what light through yonder window breaks
It is the east and Juliet is the sun
Arise fair sun and kill the envious moon
Who is already sick and pale with grief
```

خروجی با هدرهایی شروع می‌شود که وب‌سرور برای توصیف سند ارسال می‌کند. به عنوان مثال، هدر Content-Type نشان می‌دهد که سند یک سند متن ساده (text/plain) است. پس از اینکه سرور هدرها را ارسال کرد، یک خط خالی اضافه می‌کند تا پایان هدرها را نشان دهد، و سپس داده‌های واقعی فایل romeo.txt را ارسال می‌کند. این مثال نشان می‌دهد که چگونه می‌توان یک اتصال شبکه سطح پایین با سوکت‌ها برقرار کرد. سوکت‌ها می‌توانند برای ارتباط با یک وب‌سرور، سرور ایمیل یا بسیاری از انواع دیگر سرورها استفاده شوند. تنها چیزی که نیاز است، پیدا کردن سندی است که پروتکل را توصیف می‌کند و نوشتن کدی برای ارسال و دریافت داده‌ها بر اساس آن پروتکل. با این حال، از آنجا که پروتکلی که بیشتر استفاده می‌کنیم، پروتکل HTTP وب است، پایتون یک کتابخانه ویژه طراحی کرده است که به طور خاص از پروتکل HTTP برای بازیابی اسناد و داده‌ها از طریق وب پشتیبانی می‌کند.

یکی از نیازهای استفاده از پروتکلHTTP ، ارسال و دریافت داده‌ها به عنوان اشیاء بایت است، به جای رشته‌ها. در مثال قبلی، متدهای ()encode و ()decode رشته‌ها را به اشیاء بایت تبدیل کرده و برعکس. مثال بعدی از نوتیشن "b استفاده می‌کند تا مشخص کند که یک متغیر باید به عنوان یک شیء بایت ذخیره شود. متدهای ()encode و "b معادل هستند.

```
>>> b'Hello world'
b'Hello world'
>>> 'Hello world'.encode()
b'Hello world'
```

## ۱۲,۳ دریافت یک تصویر از طریق HTTP

در مثال قبلی، ما یک فایل متنی ساده که دارای خطوط جدید بود را بازیابی کردیم و داده‌ها را به سادگی به صفحه نمایش کپی کردیم در حین اجرای برنامه. می‌توانیم از یک برنامه مشابه برای دریافت یک تصویر از طریق HTTP استفاده کنیم. به جای کپی کردن داده‌ها به صفحه نمایش در حین اجرای برنامه، داده‌ها را در یک رشته جمع‌آوری می‌کنیم، هدرها را برش می‌دهیم و سپس داده‌های تصویر را به یک فایل ذخیره می‌کنیم به صورت زیر:

```
import socket
import time

HOST = 'data.pr4e.org'
PORT = 80
mysock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
mysock.connect((HOST, PORT))
mysock.sendall(b'GET http://data.pr4e.org/cover3.jpg HTTP/1.0\r\n\r\n')
count = 0
picture = b""

while True:
    data = mysock.recv(5120)
    if len(data) < 1: break
    #time.sleep(0.25)
    count = count + len(data)
    print(len(data), count)
    picture = picture + data

mysock.close()

# Look for the end of the header (2 CRLF)
pos = picture.find(b"\r\n\r\n")
print('Header length', pos)
print(picture[:pos].decode())

# Skip past the header and save the picture data
picture = picture[pos+4:]
fhand = open("stuff.jpg", "wb")
fhand.write(picture)
fhand.close()

# Code: https://www.py4e.com/code3/urljpeg.py
```

زمانی که برنامه اجرا می‌شود، خروجی زیر را تولید می‌کند:

```
$ python urljpeg.py
5120 5120
5120 10240
4240 14480
5120 19600
...
5120 214000
3200 217200
5120 222320
5120 227440
3167 230607
Header length 393
HTTP/1.1 200 OK
Date: Wed, 11 Apr 2018 18:54:09 GMT
Server: Apache/2.4.7 (Ubuntu)
Last-Modified: Mon, 15 May 2017 12:27:40 GMT
ETag: "38342-54f8f2e5b6277"
Accept-Ranges: bytes
Content-Length: 230210
Vary: Accept-Encoding
Cache-Control: max-age=0, no-cache, no-store, must-revalidate
Pragma: no-cache
Expires: Wed, 11 Jan 1984 05:00:00 GMT
Connection: close
Content-Type: image/jpeg
```

شما می‌توانید ببینید که برای این URL ، هدر Content-Type نشان می‌دهد که بدنه سند یک تصویر است .(image/jpeg) پس از اتمام برنامه، می‌توانید داده‌های تصویر را با باز کردن فایل stuff.jpg در یک بیننده تصویر مشاهده کنید. همانطور که برنامه اجرا می‌شود، مشاهده می‌کنید که هر بار که متد ()recv را فراخوانی می‌کنیم، به اندازه ۵۱۲۰ کاراکتر دریافت نمی‌کنیم. ما به اندازه‌ای که داده‌ها توسط وب سرور به شبکه منتقل شده است، دریافت می‌کنیم. در این مثال، هر بار که درخواست می‌کنیم تا ۵۱۲۰ کاراکتر داده را دریافت کنیم، ممکن است به اندازه ۳۲۰۰ کاراکتر دریافت کنیم. نتایج شما بسته به سرعت شبکه ممکن است متفاوت باشد. همچنین توجه داشته باشید که در آخرین فراخوانی به متد()recv ، ۳۱۶۷ بایت دریافت می‌کنیم که انتهای جریان است، و در فراخوانی بعدی به()recv ، یک رشته با طول صفر دریافت می‌کنیم که نشان می‌دهد سرور اتصال را از سمت خود قطع کرده است و داده‌ای باقی نمانده است.

ما می‌توانیم با باز کردن دستور()time.sleep ، فراخوانی‌های ()successive recv خود را کندتر کنیم. به این ترتیب، پس از هر فراخوانی، یک چهارم ثانیه منتظر می‌مانیم تا سرور بتواند "از ما جلوتر بیفتد" و داده‌های بیشتری را برای ما ارسال کند قبل از اینکه دوباره متد ()recv را فراخوانی کنیم. با این تاخیر، برنامه به شکل زیر اجرا می‌شود:

```
$ python urljpeg.py
5120 5120
5120 10240
```

```
5120 15360
...
5120 225280
5120 230400
207 230607
Header length 393
HTTP/1.1 200 OK
Date: Wed, 11 Apr 2018 21:42:08 GMT
Server: Apache/2.4.7 (Ubuntu)
Last-Modified: Mon, 15 May 2017 12:27:40 GMT
ETag: "38342-54f8f2e5b6277"
Accept-Ranges: bytes
Content-Length: 230210
Vary: Accept-Encoding
Cache-Control: max-age=0, no-cache, no-store, must-revalidate
Pragma: no-cache
Expires: Wed, 11 Jan 1984 05:00:00 GMT
Connection: close
Content-Type: image/jpeg
```

حالا غیر از اولین و آخرین فراخوانی‌ها به متد `recv()`، هر بار که درخواست داده جدیدی می‌کنیم، ۵۱۲۰ کاراکتر دریافت می‌کنیم.
یک بافر بین سرور که درخواست‌های `send()` را ارسال می‌کند و برنامه ما که درخواست‌های `recv()` را انجام می‌دهد وجود دارد. زمانی که برنامه را با تاخیر اجرا می‌کنیم، در یک نقطه سرور ممکن است بافر موجود در سوکت را پر کرده و مجبور شود تا زمانی که برنامه ما شروع به خالی کردن بافر نکند، توقف کند. توقف یکی از برنامه‌های فرستنده یا دریافت کننده داده‌ها به نام "کنترل جریان" شناخته می‌شود.

## ۱۲,۴ بازیابی صفحات وب با `urllib`

در حالی که می‌توانیم داده‌ها را به صورت دستی از طریق پروتکل HTTP با استفاده از کتابخانه `socket` ارسال و دریافت کنیم، راه ساده‌تری برای انجام این کار در پایتون وجود دارد که استفاده از کتابخانه `urllib` است. با استفاده از `urllib`، شما می‌توانید یک صفحه وب را مانند یک فایل ببینید. تنها کافی است که صفحه وب مورد نظر خود را مشخص کنید و `urllib` تمام جزئیات مربوط به پروتکل HTTP و هدرها را مدیریت خواهد کرد.

کد معادل برای خواندن فایل `romeo.txt` از وب با استفاده از `urllib` به شکل زیر است:

```
import urllib.request

fhand = urllib.request.urlopen('http://data.pr4e.org/romeo.txt')
for line in fhand:
    print(line.decode().strip())
```

```
# Code: https://www.py4e.com/code3/urllib1.py
```

زمانی که صفحه وب با استفاده از `urllib.request.urlopen` باز می‌شود، می‌توانیم آن را مانند یک فایل ببینیم و از یک حلقه `for` برای خواندن محتویات آن استفاده کنیم.

زمانی که برنامه اجرا می‌شود، تنها محتوای فایل را مشاهده می‌کنیم. هدرها هنوز ارسال می‌شوند، اما کد `urllib` هدرها را مصرف کرده و فقط داده‌ها را به ما باز می‌گرداند.

```
But soft what light through yonder window breaks
It is the east and Juliet is the sun
Arise fair sun and kill the envious moon
Who is already sick and pale with grief
```

برای این که بتوانیم برنامه‌ای بنویسیم که داده‌های فایل `romeo.txt` را بازیابی کرده و فراوانی هر کلمه در فایل را محاسبه کنیم، می‌توانیم از کد زیر استفاده کنیم:

```
import urllib.request, urllib.parse, urllib.error

fhand = urllib.request.urlopen('http://data.pr4e.org/romeo.txt')

counts = dict()
for line in fhand:
    words = line.decode().split()
    for word in words:
        counts[word] = counts.get(word, 0) + 1
print(counts)

# Code: https://www.py4e.com/code3/urlwords.py
```

مجدداً، پس از اینکه صفحه وب را باز کردیم، می‌توانیم آن را مانند یک فایل محلی بخوانیم.

## ۱۲,۵ خواندن فایل‌های باینری با استفاده از urllib

گاهی اوقات شما ممکن است بخواهید یک فایل غیر متنی (یا باینری) مانند یک تصویر یا فایل ویدیویی را بازیابی کنید. داده‌های این فایل‌ها معمولاً برای چاپ مفید نیستند، اما به راحتی می‌توانید یک کپی از URL را به یک فایل محلی روی دیسک سخت خود با استفاده از urllib ایجاد کنید.

الگوی کار به این صورت است که URL را باز کرده و از متد `read` برای دانلود محتوای کامل سند به یک متغیر رشته‌ای (مثل `img`) استفاده می‌کنید و سپس آن اطلاعات را به یک فایل محلی می‌نویسید به شکل زیر:

```
import urllib.request, urllib.parse, urllib.error

img = urllib.request.urlopen('http://data.pr4e.org/cover3.jpg').read()
fhand = open('cover3.jpg', 'wb')
fhand.write(img)
fhand.close()

# Code: https://www.py4e.com/code3/curl1.py
```

این برنامه تمام داده‌ها را یک‌جا از طریق شبکه می‌خواند و در متغیر `img` در حافظه اصلی کامپیوتر شما ذخیره می‌کند، سپس فایل `cover.jpg` را باز کرده و داده‌ها را به دیسک شما می‌نویسد. آرگومان `wb` برای متد `open()` فایل باینری را برای نوشتن باز می‌کند. این برنامه زمانی که اندازه فایل از حافظه کامپیوتر شما بزرگتر نباشد، به خوبی کار می‌کند.

با این حال، اگر این فایل یک فایل صوتی یا ویدیویی بزرگ باشد، این برنامه ممکن است کرش کند یا حداقل وقتی حافظه کامپیوتر شما تمام می‌شود، به طرز فوق‌العاده‌ای کند اجرا شود. به‌منظور جلوگیری از تمام شدن حافظه، ما داده‌ها را به‌صورت بلوک‌ها (یا بافرها) دریافت می‌کنیم و سپس هر بلوک را قبل از دریافت بلوک بعدی به دیسک می‌نویسیم. به این ترتیب، برنامه می‌تواند هر اندازه فایلی را بدون استفاده از تمام حافظه موجود در کامپیوتر شما بخواند.

```
import urllib.request, urllib.parse, urllib.error

img = urllib.request.urlopen('http://data.pr4e.org/cover3.jpg')
fhand = open('cover3.jpg', 'wb')
size = 0
while True:
    info = img.read(100000)
    if len(info) < 1: break
    size = size + len(info)
    fhand.write(info)

print(size, 'characters copied.')
fhand.close()

# Code: https://www.py4e.com/code3/curl2.py
```

در این مثال، ما فقط ۱۰۰،۰۰۰ کاراکتر را در هر بار می‌خوانیم و سپس آن کاراکترها را به فایل `cover3.jpg` می‌نویسیم قبل از اینکه ۱۰۰،۰۰۰ کاراکتر بعدی را از وب دریافت کنیم.

این برنامه به‌طور زیر اجرا می‌شود:

```
python curl2.py
230210 characters copied.
```

## ۱۲,۶ تحلیل HTML و استخراج داده از وب

یکی از کاربردهای رایج قابلیت `urllib` در پایتون، استخراج داده از وب است. استخراج داده از وب به این معنی است که برنامه‌ای بنویسیم که خود را به‌جای یک مرورگر وب قرار دهد و صفحات را بازیابی کرده و سپس داده‌های آن صفحات را برای جستجوی الگوها بررسی کند.

برای مثال، یک موتور جستجو مانند گوگل به منبع یک صفحه وب نگاه می‌کند و لینک‌ها به صفحات دیگر را استخراج کرده و آن صفحات را بازیابی کرده و لینک‌ها را استخراج می‌کند و این فرآیند را ادامه می‌دهد. با استفاده از این تکنیک، گوگل به‌طور مداوم به صفحات مختلف وب سر می‌زند.

گوگل همچنین از فراوانی لینک‌های موجود در صفحات به یک صفحه خاص به‌عنوان یکی از معیارها برای تعیین اهمیت صفحه و رتبه آن در نتایج جستجو استفاده می‌کند.

## ۱۲,۷ تحلیل HTML با استفاده از عبارات منظم

یک روش ساده برای تحلیل HTML استفاده از عبارات منظم است که به‌طور مداوم به جستجوی زیررشته‌هایی که با یک الگوی خاص مطابقت دارند می‌پردازد.
در اینجا یک صفحه وب ساده آورده شده است:

```
<h1>The First Page</h1>
<p>
If you like, you can switch to the
<a href="http://www.dr-chuck.com/page2.htm">
Second Page</a>.
</p>
```

ما می‌توانیم یک عبارت منظم ساختار یافته خوب برای تطابق و استخراج مقادیر لینک‌ها از متن بالا به شرح زیر بسازیم:

```
href="http[s]?://.+?"
```

عبارت منظم ما به دنبال رشته‌هایی می‌گردد که با `href="http://` یا `href="https://` شروع می‌شوند، سپس با یک یا چند کاراکتر ادامه می‌یابند (`.+?`) و در نهایت با یک علامت نقل‌قول دوتایی دیگر بسته می‌شوند. علامت سؤال پس از `http[s]?` نشان می‌دهد که باید به‌دنبال رشته‌ای بگردیم که با "http" شروع شده و صفر یا یک بار "s" در ادامه داشته باشد.

علامت سؤال اضافه شده به .+? مشخص می‌کند که تطبیق باید به‌صورت "غیر حریصانه (non-greedy) "انجام شود، نه "حریصانه.(greedy) " در تطبیق غیر حریصانه، موتور عبارت منظم تلاش می‌کند کوچک‌ترین رشته ممکن را که با الگو مطابقت دارد بیابد، در حالی که در تطبیق حریصانه، بزرگ‌ترین رشته ممکن را انتخاب می‌کند.

ما در عبارت منظم از پرانتز استفاده می‌کنیم تا مشخص کنیم کدام بخش از رشته‌ی تطبیق داده‌شده را می‌خواهیم استخراج کنیم و برنامه‌ی زیر را تولید می‌کنیم:

```
# Search for link values within URL input
import urllib.request, urllib.parse, urllib.error
import re
import ssl

# Ignore SSL certificate errors
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

url = input('Enter - ')
html = urllib.request.urlopen(url, context=ctx).read()
links = re.findall(b'href="(http[s]?://.*?)"', html)
for link in links:
    print(link.decode())

# Code: https://www.py4e.com/code3/urlregex.py
```

کتابخانه‌ی ssl به این برنامه اجازه می‌دهد تا به وب‌سایت‌هایی دسترسی پیدا کند که به‌طور سخت‌گیرانه از پروتکل HTTPS استفاده می‌کنند. متد read کد منبع HTML را به‌صورت یک شیء از نوع bytes بازمی‌گرداند، نه یک شیء از نوع HTTPResponse.

متد findall از عبارات منظم (regular expression) لیستی از تمام رشته‌هایی را که با الگوی ما مطابقت دارند بازمی‌گرداند و تنها متن لینک بین علامت‌های نقل‌قول دوتایی را استخراج می‌کند.

زمانی که برنامه را اجرا کرده و یک URL وارد می‌کنیم، خروجی زیر را دریافت خواهیم کرد:

```
Enter - https://docs.python.org
https://docs.python.org/3/index.html
https://www.python.org/
https://docs.python.org/3.8/
https://docs.python.org/3.7/
https://docs.python.org/3.5/
```

```
https://docs.python.org/2.7/
https://www.python.org/doc/versions/
https://www.python.org/dev/peps/
https://wiki.python.org/moin/BeginnersGuide
https://wiki.python.org/moin/PythonBooks
https://www.python.org/doc/av/
https://www.python.org/
https://www.python.org/psf/donations/
http://sphinx.pocoo.org/
```

عبارات منظم (Regular expressions) زمانی بسیار خوب عمل می‌کنند که HTML شما به‌خوبی قالب‌بندی شده و قابل پیش‌بینی باشد. اما از آنجایی که صفحات HTML زیادی وجود دارند که «خراب» هستند، یک راه‌حل که فقط از عبارات منظم استفاده کند ممکن است برخی از لینک‌های معتبر را نادیده بگیرد یا در نهایت داده‌های نادرستی به‌دست آورد.

این مشکل را می‌توان با استفاده از یک کتابخانه‌ی قدرتمند برای تجزیه‌ی HTML حل کرد.

## ۱۲,۸ تجزیه‌ی HTML با استفاده ازBeautifulSoup

با اینکه HTML شبیه به XML به نظر می‌رسد و برخی صفحات با دقت مانند XML ساخته شده‌اند، اکثر صفحات HTML معمولاً به گونه‌ای خراب هستند که باعث می‌شود یک تجزیه‌گر XML کل صفحه را به عنوان ساختار نادرست رد کند.

کتابخانه‌های متعددی در Python وجود دارند که می‌توانند به شما کمک کنند تا HTML را تجزیه کرده و داده‌ها را از صفحات استخراج کنید. هر یک از این کتابخانه‌ها مزایا و معایب خاص خود را دارند و می‌توانید بسته به نیازتان یکی را انتخاب کنید.

به عنوان نمونه، ما تنها یک ورودی HTML را تجزیه می‌کنیم و لینک‌ها را با استفاده از کتابخانه‌ی BeautifulSoup استخراج خواهیم کرد .BeautifulSoup می‌تواند HTML های بسیار خراب را تحمل کرده و همچنان به شما اجازه دهد داده‌های مورد نیازتان را به‌آسانی استخراج کنید. شما می‌توانید کد BeautifulSoup را از آدرس زیر دانلود و نصب کنید:

https://pypi.python.org/pypi/beautifulsoup4

اطلاعات مربوط به نصب BeautifulSoup با استفاده از ابزار Python Package Index (pip) در آدرس زیر در دسترس است:

https://packaging.python.org/tutorials/installing-packages/

ما از کتابخانه‌ی `urllib` برای خواندن صفحه‌ی وب استفاده خواهیم کرد و سپس با کمک `BeautifulSoup`، ویژگی‌های `href` را از تگ‌های لنگر (`a`) استخراج می‌کنیم .

```
# To run this, download the BeautifulSoup zip file
# http://www.py4e.com/code3/bs4.zip
# and unzip it in the same directory as this file
```

```
import urllib.request, urllib.parse, urllib.error
from bs4 import BeautifulSoup
import ssl

# Ignore SSL certificate errors
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

url = input('Enter - ')
html = urllib.request.urlopen(url, context=ctx).read()
soup = BeautifulSoup(html, 'html.parser')

# Retrieve all of the anchor tags
tags = soup('a')
for tag in tags:
    print(tag.get('href', None))

# Code: https://www.py4e.com/code3/urllinks.py
```

این برنامه از کاربر یک آدرس وب درخواست می‌کند، سپس صفحه‌ی وب را باز می‌کند، داده‌ها را می‌خواند و آن‌ها را به تجزیه‌گر BeautifulSoup (parser) می‌سپارد، و سپس تمام تگ‌های لنگر (anchor) را بازیابی کرده و مقدار ویژگی href را برای هر تگ چاپ می‌کند.

زمانی که برنامه اجرا می‌شود، خروجی زیر را تولید می‌کند:

```
Enter - https://docs.python.org
genindex.html
py-modindex.html
https://www.python.org/
#
whatsnew/3.6.html
whatsnew/index.html
tutorial/index.html
library/index.html
reference/index.html
using/index.html
howto/index.html
installing/index.html
distributing/index.html
extending/index.html
c-api/index.html
faq/index.html
py-modindex.html
genindex.html
glossary.html
search.html
```

```
contents.html
bugs.html
about.html
license.html
copyright.html
download.html
https://docs.python.org/3.8/
https://docs.python.org/3.7/
https://docs.python.org/3.5/
https://docs.python.org/2.7/
https://www.python.org/doc/versions/
https://www.python.org/dev/peps/
https://wiki.python.org/moin/BeginnersGuide
https://wiki.python.org/moin/PythonBooks
https://www.python.org/doc/av/
genindex.html
py-modindex.html
https://www.python.org/
#
copyright.html
https://www.python.org/psf/donations/
bugs.html
http://sphinx.pocoo.org/
```

این فهرست بسیار طولانی‌تر است زیرا برخی از تگ‌های لنگر (anchor) در HTML مسیرهای نسبی هستند (مانند `tutorial/index.html`) یا ارجاع‌های درون‌صفحه‌ای (مانند `'#'`) که شامل `http://` یا `https://` نیستند، در حالی که این پیش‌نیاز در عبارت منظم (regular expression) ما وجود داشت.

شما همچنین می‌توانید از BeautifulSoup استفاده کنید تا بخش‌های مختلف هر تگ را استخراج کنید:

```
# To run this, download the BeautifulSoup zip file
# http://www.py4e.com/code3/bs4.zip
# and unzip it in the same directory as this file

from urllib.request import urlopen
from bs4 import BeautifulSoup
import ssl

# Ignore SSL certificate errors
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

url = input('Enter - ')
html = urlopen(url, context=ctx).read()
soup = BeautifulSoup(html, "html.parser")
```

```
# Retrieve all of the anchor tags
tags = soup('a')
for tag in tags:
    # Look at the parts of a tag
    print('TAG:', tag)
    print('URL:', tag.get('href', None))
    print('Contents:', tag.contents[0])
    print('Attrs:', tag.attrs)

# Code: https://www.py4e.com/code3/urllink2.py
python urllink2.py
Enter - http://www.dr-chuck.com/page1.htm
TAG: <a href="http://www.dr-chuck.com/page2.htm">
Second Page</a>
URL: http://www.dr-chuck.com/page2.htm
Content: ['\nSecond Page']
Attrs: [('href', 'http://www.dr-chuck.com/page2.htm')]
```

`html.parser` یک تجزیه‌گر HTML (parser) است که در کتابخانه استاندارد پایتون ۳ گنجانده شده است. اطلاعات مربوط به دیگر تجزیه‌گرهای HTML در آدرس زیر در دسترس است:

http://www.crummy.com/software/BeautifulSoup/bs4/doc/#installing-a-parser

این مثال‌ها تنها آغاز قدرت کتابخانه‌ی BeautifulSoup را در هنگام تجزیه‌ی کد HTML نشان می‌دهند.

## ۱۲,۹ بخش اضافی برای کاربرانUnix / Linux

اگر از یک رایانه‌یLinux ، Unix یا Macintosh استفاده می‌کنید، احتمالاً دستورات از پیش‌نصب‌شده‌ای در سیستم‌عامل خود دارید که می‌توانند فایل‌های متنی ساده یا باینری را با استفاده از پروتکل HTTP یا FTP بازیابی کنند. یکی از این دستورات، دستور **curl** است.

```
$ curl -O http://www.py4e.com/cover.jpg
```

دستور curl مخفف "copy URL" (کپی آدرس اینترنتی) است، و به همین دلیل دو مثالی که پیش‌تر برای بازیابی فایل‌های باینری با استفاده از urllib آورده شد، با نام‌های هوشمندانه curl1.py و curl2.py در سایت www.py4e.com/code3 نام‌گذاری شده‌اند، چرا که عملکردی مشابه دستور curl دارند. همچنین یک نمونه برنامه‌ی curl3.py نیز وجود دارد که این کار را کمی مؤثرتر انجام می‌دهد، در صورتی که واقعاً بخواهید از این الگو در برنامه‌ای که می‌نویسید استفاده کنید. دستور دوم که عملکردی بسیار مشابه دارد، wget است.

```
$ wget http://www.py4e.com/cover.jpg
```

هر دوی این دستورات انجام کار بازیابی صفحات وب و فایل‌های راه دور را ساده می‌کنند.

## ۱۲,۱۰ واژه نامه

**BeautifulSoup**: کتابخانه‌ای در پایتون برای تجزیه اسناد HTML و استخراج داده‌ها از اسناد HTML که نقص‌های موجود در HTML که معمولاً توسط مرورگرها نادیده گرفته می‌شود را جبران می‌کند. می‌توانید کد BeautifulSoup را از سایت www.crummy.comدانلود کنید.

**Port**: شماره‌ای که معمولاً نشان‌دهنده برنامه‌ای است که شما در زمان برقراری ارتباط با یک سرور از طریق سوکت به آن دسترسی دارید. به عنوان مثال، ترافیک وب معمولاً از پورت ۸۰ استفاده می‌کند، در حالی که ترافیک ایمیل از پورت ۲۵ استفاده می‌کند.

**Scrape**: زمانی که یک برنامه نقش یک مرورگر وب را بازی می‌کند و یک صفحه وب را بازیابی کرده و سپس محتوای صفحه وب را بررسی می‌کند. اغلب برنامه‌ها از پیوندهای موجود در یک صفحه برای پیدا کردن صفحه بعدی استفاده می‌کنند تا بتوانند یک شبکه از صفحات یا شبکه اجتماعی را جستجو کنند.

**Socket**: اتصال شبکه‌ای بین دو برنامه که در آن برنامه‌ها می‌توانند داده‌ها را در هر دو جهت ارسال و دریافت کنند.

**Spider**: عملی که در آن یک موتور جستجوی وب یک صفحه را بازیابی کرده و سپس تمام صفحات لینک شده از آن صفحه را بررسی می‌کند و به همین ترتیب ادامه می‌دهد تا تقریباً تمام صفحات موجود در اینترنت را جمع‌آوری کرده و از آن‌ها برای ساخت ایندکس جستجو استفاده می‌کند.

## ۱۲,۱۱ تمرین ها

**تمرین یک**: برنامه‌ی sokcet1.py را تغییر دهید تا از کاربر برای وارد کردن URL درخواست کند تا بتواند هر صفحه‌ی وبی را بخواند.
می‌توانید از `split('/')` برای شکستن URL به اجزای مختلف آن استفاده کنید تا بتوانید نام میزبان (host name) را برای فراخوانی `socket connect` استخراج کنید. افزودن بررسی خطا با استفاده از `try` و `except` برای رسیدگی به شرایطی که کاربر یک URL نادرست یا غیرموجود وارد می‌کند، لازم است.

**تمرین دو:** برنامه‌ی socket خود را تغییر دهید تا تعداد کاراکترهایی را که دریافت کرده است بشمارد و پس از نمایش ۳۰۰۰ کاراکتر، نمایش متن را متوقف کند. برنامه باید کل سند را بازیابی کند و تعداد کل کاراکترها را بشمارد و این تعداد را در پایان سند نمایش دهد.

**تمرین سه**: از urllib استفاده کنید تا تمرین قبلی را تکرار کنید:
(۱ ) دریافت سند از یکURL ،
(۲ ) نمایش حداکثر ۳۰۰۰ کاراکتر،
(۳ ) شمارش تعداد کل کاراکترهای موجود در سند.
برای این تمرین نگران هدرها نباشید، فقط ۳۰۰۰ کاراکتر اول محتوای سند را نمایش دهید.

**تمرین چهار**: برنامه‌ی urllinks.py را تغییر دهید تا تگ‌های پاراگراف (p) را از سند HTML واکشی‌شده استخراج و شمارش کند و تعداد پاراگراف‌ها را به‌عنوان خروجی برنامه نمایش دهد. متن پاراگراف‌ها را نمایش ندهید، فقط آن‌ها را بشمارید. برنامه را روی چند صفحه‌ی کوچک و همچنین چند صفحه‌ی بزرگ آزمایش کنید.

**تمرین پنج:** برنامه‌ی socket را تغییر دهید تا فقط داده‌هایی را نمایش دهد که **پس از دریافت هدرها و یک خط خالی** آمده‌اند. به یاد داشته باشید که تابع `recv` کاراکترها را دریافت می‌کند (همراه با کاراکترهای خط جدید)، نه خطوط را.

# فصل سیزدهم

# استفاده از خدمات وب

زمانی که دریافت و تجزیه ی اسناد از طریق HTTP با استفاده از برنامه‌ها آسان شد، مدت زیادی طول نکشید تا رویکردی توسعه پیدا کند که در آن شروع به تولید اسنادی کردیم که به‌طور خاص برای استفاده توسط برنامه‌های دیگر طراحی شده بودند (یعنی اسنادی که برای نمایش در مرورگر نبودن).

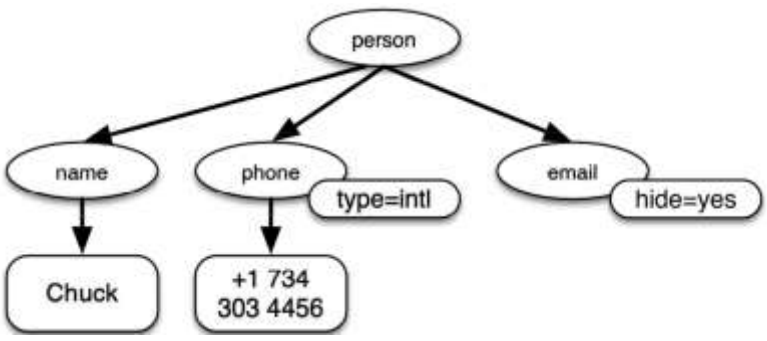
دو فرمت رایج برای تبادل داده از طریق وب وجود دارد. زبان نشانه‌گذاری توسعه‌پذیر (XML) مدت زمان زیادی است که مورد استفاده قرار می‌گیرد و برای تبادل داده‌هایی با ساختار سند گونه مناسب است. زمانی که برنامه‌ها صرفاً می‌خواهند دیکشنری‌ها، لیست‌ها یا اطلاعات داخلی دیگر را با هم تبادل کنند، از فرمت JavaScript Object Notation (JSON) استفاده می‌کنند (به سایت www.json.org مراجعه کنید). ما به هر دو فرمت خواهیم پرداخت.

## ۱۳,۱ زبان نشانه‌گذاری توسعه‌پذیر XML

XML از لحاظ ظاهری بسیار شبیه به HTML است، اما ساختارمندتر از HTML می‌باشد. در اینجا نمونه‌ای از یک سند XML آورده شده است:

```
<person>
  <name>Chuck</name>
  <phone type="intl">
    +1 734 303 4456
  </phone>
  <email hide="yes" />
</person>
```

هر جفت تگ باز (مثلاً <person>) و تگ بسته (مثلاً </person>) یک عنصر یا گره را با همان نام تگ (مثلاً person) نشان می‌دهد. هر عنصر می‌تواند شامل مقداری متن، برخی ویژگی‌ها (attributes) مانند hide ، و عناصر تودرتو یا تودرتوهای دیگر باشد. اگر یک عنصر XML خالی باشد (یعنی محتوایی نداشته باشد)، می‌توان آن را با یک تگ خودبسته نمایش داد (مثلاً <email />). اغلب مفید است که به یک سند XML به عنوان یک ساختار درختی نگاه کنیم؛ جایی که یک عنصر بالادست (در اینجا person :) وجود دارد، و تگ‌های دیگر (مثلاًphone ) به عنوان فرزندان عناصر والدشان ترسیم می‌شوند.



## ۱۳,۲ تجزیه‌ی XML

در اینجا یک برنامه‌ی ساده آورده شده که مقداری XML را تجزیه کرده و برخی عناصر داده را از آن استخراج می‌کند:

```
import xml.etree.ElementTree as ET

data = '''
<person>
  <name>Chuck</name>
  <phone type="intl">
    +1 734 303 4456
  </phone>
  <email hide="yes" />
</person>'''

tree = ET.fromstring(data)
print('Name:', tree.find('name').text)
print('Attr:', tree.find('email').get('hide'))

# Code: https://www.py4e.com/code3/xml1.py
```

سه تا کوتیشن تکی (''') یا سه تا کوتیشن دوتایی (""") به ما این امکان را می‌دهند که رشته‌هایی بسازیم که در چندین خط نوشته می‌شوند.

فراخوانی تابع `fromstring`، رشته‌ی متنی XML را به یک "درخت" از عناصر XML تبدیل می‌کند. وقتی XML به صورت یک درخت نمایش داده می‌شود، مجموعه‌ای از متدها در اختیار داریم که می‌توانند بخش‌هایی از داده را از رشته‌ی XML استخراج کنند. تابع `find`در درخت XML جست‌وجو می‌کند و عنصری را که با تگ مشخص‌شده مطابقت دارد، بازیابی می‌کند.

```
Name: Chuck
Attr: yes
```

استفاده از یک تجزیه‌گر XML مانند `ElementTree` این مزیت را دارد که هرچند XML این مثال ساده است، ولی در واقع قوانین زیادی درباره‌ی نگارش درست XML وجود دارد، و استفاده از `ElementTree` به ما کمک می‌کند بدون نگرانی درباره‌ی قواعد نحوی XML ، داده‌ها را از آن استخراج کنیم.

## ۱۳,۳ حلقه زدن روی گره‌ها

اغلب XML شامل چندین گره است و ما نیاز داریم که با استفاده از یک حلقه همه‌ی گره‌ها را پردازش کنیم. در برنامه‌ی زیر، از طریق تمام گره‌های `user` حلقه می‌زنیم:

```
import xml.etree.ElementTree as ET

input = '''
<stuff>
  <users>
    <user x="2">
      <id>001</id>
      <name>Chuck</name>
    </user>
    <user x="7">
      <id>009</id>
      <name>Brent</name>
    </user>
  </users>
</stuff>'''

stuff = ET.fromstring(input)
lst = stuff.findall('users/user')
print('User count:', len(lst))

for item in lst:
    print('Name', item.find('name').text)
    print('Id', item.find('id').text)
    print('Attribute', item.get('x'))

# Code: https://www.py4e.com/code3/xml2.py
```

متد `findall` یک لیست پایتونی از زیردرخت‌هایی که ساختارهای `user` را در درخت XML نمایش می‌دهند، بازیابی می‌کند. سپس می‌توانیم با استفاده از یک حلقه‌ی `for` به هر یک از گره‌های `user` نگاه کنیم و عناصر متنی `name` و `id` را همراه با ویژگی `x` از گره‌ی `user` چاپ کنیم.

```
User count: 2
Name Chuck
Id 001
Attribute 2
Name Brent
Id 009
Attribute 7
```

مهم است که در دستور `findall` تمام عناصر والد (parent) به‌جز عنصر سطح بالای XML (برای مثال `users/user` :) را درج کنیم. در غیر این صورت، پایتون هیچ یک از گره‌های مورد نظر را پیدا نخواهد کرد.

```
import xml.etree.ElementTree as ET

input = '''
<stuff>
  <users>
    <user x="2">
```

```
      <id>001</id>
      <name>Chuck</name>
    </user>
    <user x="7">
      <id>009</id>
      <name>Brent</name>
    </user>
  </users>
</stuff>'''

stuff = ET.fromstring(input)

lst = stuff.findall('users/user')
print('User count:', len(lst))

lst2 = stuff.findall('user')
print('User count:', len(lst2))
```

`lst` تمام عناصری از نوع `user` را ذخیره می‌کند که درون عنصر والد `users` قرار دارند. در مقابل، `lst2` به دنبال عناصری از نوع `user` می‌گردد که درون عنصر سطح بالای `stuff` قرار نگرفته‌اند، در حالی که چنین عناصری در ساختار وجود ندارند.

```
User count: 2
User count: 0
```

## ۱۳,۴ جیسون (JavaScript Object Notation – JSON)

فرمت JSON از ساختار اشیاء و آرایه‌ها در زبان JavaScript الهام گرفته شده است. اما از آنجایی که زبان Python پیش از JavaScript ابداع شده بود، نحوه نگارش دیکشنری‌ها و لیست‌ها در Python بر نحوه نگارش JSON نیز تأثیر گذاشته است. بنابراین، ساختار JSON تقریباً مشابه ترکیبی از لیست‌ها و دیکشنری‌های Python است.

در ادامه، یک نمونه کد JSON آورده شده که تقریباً معادل همان نمونه ساده XML بالا است:

```
{
  "name" : "Chuck",
  "phone" : {
    "type" : "intl",
    "number" : "+1 734 303 4456"
   },
   "email" : {
     "hide" : "yes"
   }
}
```

تفاوت‌هایی که مشاهده خواهید کرد این است که اولاً درXML ، می‌توانیم ویژگی‌هایی مانند "intl" را به تگ "phone" اضافه کنیم. اما درJSON ، ما به سادگی از جفت‌های کلید-مقدار استفاده می‌کنیم. همچنین تگ "person" در XML حذف شده و با یک مجموعه از آکولادهای خارجی جایگزین شده است.

به طور کلی، ساختارهای JSON ساده‌تر از XML هستند زیرا JSON قابلیت‌های کمتری نسبت به XML دارد. اما مزیت JSON این است که به طور مستقیم با ترکیبی از دیکشنری‌ها و لیست‌ها مرتبط است. و از آنجا که تقریباً تمام زبان‌های برنامه‌نویسی چیزی معادل دیکشنری‌ها و لیست‌های Python دارند، JSON فرمت بسیار طبیعی برای تبادل داده بین دو برنامه همکار است. JSON به سرعت به فرمت انتخابی برای تبادل داده بین برنامه‌ها تبدیل شده است، به دلیل سادگی نسبی آن نسبت به XML

## ۱۳,۵ تجزیه Jason

ما JSON خود را با تو در تو کردن دیکشنری‌ها و لیست‌ها به صورت مورد نیاز می‌سازیم. در این مثال، یک لیست از کاربران را نشان می‌دهیم که هر کاربر مجموعه‌ای از جفت‌های کلید-مقدار است (یعنی یک دیکشنری). بنابراین، ما یک لیست از دیکشنری‌ها داریم.

در برنامه زیر، از کتابخانه داخلی json برای تجزیه JSON و خواندن داده‌ها استفاده می‌کنیم. این را با داده‌ها و کد معادل XML بالا مقایسه کنید .JSON جزئیات کمتری دارد، بنابراین باید از پیش بدانیم که یک لیست داریم و این لیست مربوط به کاربران است و هر کاربر مجموعه‌ای از جفت‌های کلید-مقدار است .JSON مختصرتر است (یک مزیت) اما همچنین خود توصیف کمتری دارد (یک معایب)

```
import json

data = '''
[
  { "id" : "001",
    "x" : "2",
    "name" : "Chuck"
  } ,
  { "id" : "009",
    "x" : "7",
    "name" : "Brent"
  }
]'''

info = json.loads(data)
print('User count:', len(info))

for item in info:
    print('Name', item['name'])
    print('Id', item['id'])
    print('Attribute', item['x'])

# Code: https://www.py4e.com/code3/json2.py
```

اگر کد برای استخراج داده‌ها از JSON و XML را مقایسه کنید، خواهید دید که آنچه از json.loads() دریافت می‌کنیم یک لیست پایتون است که با استفاده از یک حلقه for از آن عبور می‌کنیم، و هر آیتم درون این لیست یک دیکشنری پایتون است. پس از تجزیه JSON ، می‌توانیم از اپراتور اندیس پایتون برای استخراج بخش‌های مختلف داده‌ها برای هر کاربر استفاده کنیم. نیازی به استفاده از کتابخانه JSON برای جستجو در JSON تجزیه شده نیست، زیرا داده‌های برگشتی تنها ساختارهای بومی پایتون هستند.
خروجی این برنامه دقیقاً مشابه نسخه XML بالا است.

```
User count: 2
Name Chuck
Id 001
Attribute 2
Name Brent
Id 009
Attribute 7
```

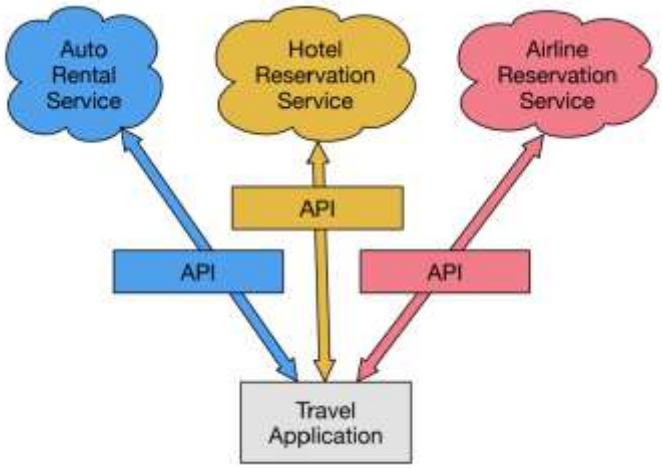
به طور کلی، یک روند صنعتی وجود دارد که به سمت استفاده از JSON به جای XML برای خدمات وب حرکت می‌کند. زیرا JSON ساده‌تر است و به طور مستقیم‌تر به ساختارهای داده‌ای بومی که در زبان‌های برنامه‌نویسی داریم، نگاشت می‌شود، بنابراین کد تجزیه و استخراج داده‌ها معمولاً هنگام استفاده از JSON ساده‌تر و مستقیم‌تر است. اما XML بیشتر خود توصیفی است و بنابراین در برخی برنامه‌ها، XML هنوز مزیتی دارد. به عنوان مثال، بیشتر پردازشگرهای کلمات اسناد را به صورت داخلی با استفاده از XML ذخیره می‌کنند نهJSON

## ۱۳,۶ واحدهای رابط برنامه‌نویسی کاربردی(APIs)

حالا که توانایی تبادل داده بین برنامه‌ها از طریق پروتکل انتقال ابرمتن (HTTP) را داریم و همچنین راهی برای نمایش داده‌های پیچیده‌ای که بین این برنامه‌ها ارسال و دریافت می‌شود با استفاده از زبان نشانه‌گذاری توسعه‌پذیر (XML) یا قالب شیء نشانه‌گذاری جاوااسکریپت(JSON) ، گام بعدی این است که "قراردادها" بین برنامه‌ها را با استفاده از این تکنیک‌ها تعریف و مستند کنیم. نام عمومی این قراردادها که بین برنامه‌ها تعریف می‌شود، رابط‌های برنامه‌نویسی کاربردی (API) است.

وقتی از یک API استفاده می‌کنیم، معمولاً یک برنامه مجموعه‌ای از خدمات را برای استفاده سایر برنامه‌ها فراهم می‌آورد و API ها (یعنی "قوانین") که باید برای دسترسی به خدمات ارائه شده توسط برنامه دنبال شوند، منتشر می‌کند.

زمانی که شروع به ساخت برنامه‌های خود می‌کنیم و عملکرد برنامه ما شامل دسترسی به خدماتی است که توسط برنامه‌های دیگر ارائه شده، این رویکرد را معماری مبتنی بر سرویس (SOA) می‌نامیم. رویکرد SOA به این صورت است که برنامه کلی ما از خدمات برنامه‌های دیگر استفاده می‌کند. در حالی که رویکرد غیرSOA به این معناست که برنامه یک برنامه مستقل است که تمام کدهای لازم برای پیاده سازی آن را در خود دارد.




ما بسیاری از نمونه‌های SOA را زمانی می‌بینیم که از وب استفاده می‌کنیم. به عنوان مثال، می‌توانیم به یک سایت واحد برویم و بلیت هواپیما، هتل و خودرو را همه از یک سایت رزرو کنیم. داده‌های هتل‌ها در سیستم‌های هواپیمایی ذخیره نمی‌شود. بلکه، سیستم‌های هواپیمایی از طریق سرویس‌ها با سیستم‌های هتل تماس می‌گیرند و داده‌های هتل را دریافت کرده و به کاربر نمایش می‌دهند. وقتی کاربر تصمیم به رزرو هتل از طریق سایت هواپیمایی می‌گیرد، سایت هواپیمایی از سرویس دیگری در سیستم‌های هتل برای انجام رزرو استفاده می‌کند. و زمانی که زمان شارژ کردن کارت اعتباری برای کل تراکنش فرا می‌رسد، کامپیوترهای دیگری در این فرآیند درگیر می‌شوند.

معماری مبتنی بر سرویس مزایای زیادی دارد که شامل موارد زیر می‌شود: (۱) ما همیشه فقط یک نسخه از داده‌ها را نگهداری می‌کنیم ( این موضوع به‌ویژه برای مواردی مانند رزرو هتل مهم است، جایی که نمی‌خواهیم به طور اضافی اطلاعات را وارد کنیم ) و (۲) مالکان داده‌ها می‌توانند قوانین استفاده از داده‌های خود را تعیین کنند. با این مزایا، سیستم SOA باید به‌طور دقیق طراحی شود تا عملکرد خوبی داشته باشد و نیازهای کاربران را برآورده کند.

وقتی یک برنامه مجموعه‌ای از خدمات خود را در API خود به صورت آنلاین در دسترس قرار می‌دهد، این خدمات را وب سرویس می‌نامیم.

## ۱۳,۷ امنیت و استفاده ازAPI

بسیار رایج است که برای استفاده از API یک ارائه دهنده، نیاز به یک کلید API داشته باشید. ایده کلی این است که آن‌ها می‌خواهند بدانند که چه کسی از خدمات آن‌ها استفاده می‌کند و هر کاربر چقدر از خدمات آن‌ها استفاده کرده است. ممکن است آن‌ها سطوح رایگان و پرداختی برای خدمات خود داشته باشند یا سیاستی داشته باشند که محدودیت‌هایی برای تعداد درخواست‌هایی که یک فرد می‌تواند در یک دوره زمانی خاص ارسال کند، اعمال کنند.

گاهی اوقات، پس از دریافت کلید API ، شما به سادگی این کلید را به عنوان بخشی از داده‌های POST یا به عنوان یک پارامتر در URL هنگام فراخوانی API قرار می‌دهید.

در مواقع دیگر، ارائه دهنده می‌خواهد از منبع درخواست‌ها اطمینان بیشتری داشته باشد و از شما انتظار دارد که پیام‌ها را با استفاده از کلیدها و رمزهای مشترک به‌طور رمزنگاری شده امضا کنید. یکی از فناوری‌های بسیار رایجی که برای امضای درخواست‌ها در اینترنت استفاده می‌شود، OAuth نام دارد. شما می‌توانید اطلاعات بیشتری در مورد پروتکل OAuth در وب‌سایت www.oauth.net بخوانید.

خوشبختانه، کتابخانه‌های رایگان و راحتی برای OAuth وجود دارند که به شما این امکان را می‌دهند که از نوشتن یک پیاده‌سازی OAuth از ابتدا خودداری کنید. این کتابخانه‌ها پیچیدگی‌های متفاوتی دارند و از نظر قابلیت‌ها متفاوت هستند. وب‌سایت OAuth اطلاعاتی در مورد کتابخانه‌های مختلف OAuth ارائه می‌دهد.

## ۱۳,۸ واژه‌نامه

**API**: رابط برنامه‌نویسی کاربردی - قراردادی بین برنامه‌ها که الگوهای تعامل بین دو مولفه برنامه را تعریف می‌کند.

**ElementTree**: یک کتابخانه داخلی پایتون برای تجزیه داده‌های XML.

**JSON**: فرمت نشانه‌گذاری شیء جاوااسکریپت - فرمت مورد استفاده برای نشانه‌گذاری داده‌های ساختار یافته بر اساس سینتکس شیء‌های جاوااسکریپت.

**SOA**: معماری مبتنی بر سرویس - هنگامی که یک برنامه از مولفه‌هایی تشکیل شده است که از طریق شبکه به هم متصل هستند.

**XML**: زبان نشانه‌گذاری قابل توسعه - فرمت مورد استفاده برای نشانه‌گذاری داده‌های ساختار یافته.

# فصل چهاردهم

# برنامه‌نویسی شیءگرا

## ۱۴,۱ مدیریت برنامه‌های بزرگتر

در ابتدای این کتاب، چهار الگوی اصلی برنامه‌نویسی را معرفی کردیم که برای ساخت برنامه‌ها از آن‌ها استفاده می‌کنیم:

- کد ترتیبی
- کد شرطی (دستوراتif )
- کد تکراری (حلقه‌ها)
- ذخیره‌سازی و استفاده مجدد (توابع)

در فصول بعدی، متغیرهای ساده و همچنین ساختارهای داده‌ای مجموعه‌ای مانند لیست‌ها، تاپل‌ها و دیکشنری‌ها را بررسی کردیم. هنگام ساخت برنامه‌ها، ما ساختارهای داده‌ای را طراحی کرده و کدی برای دستکاری آن‌ها می‌نویسیم. راه‌های مختلفی برای نوشتن برنامه‌ها وجود دارد و تا به حال احتمالاً برنامه‌هایی نوشته‌اید که "چندان زیبا" نیستند و برنامه‌هایی که "زیباتر" هستند. حتی اگر برنامه‌های شما کوچک باشند، شروع به دیدن این می‌کنید که نوشتن کد مقداری هنر و زیبایی خاص خود را دارد. با این حال، وقتی برنامه‌ها به میلیون‌ها خط کد می‌رسند، نوشتن کدی که قابل فهم باشد، بیشتر اهمیت پیدا می‌کند. اگر شما در حال کار روی یک برنامه میلیون خطی باشید، نمی‌توانید تمام برنامه را به یکباره در ذهن خود نگه دارید. به روش‌هایی نیاز داریم تا برنامه‌های بزرگ را به قطعات کوچک‌تر تقسیم کنیم تا وقتی مشکلی را حل می‌کنیم، یک باگ را اصلاح می‌کنیم یا ویژگی جدیدی اضافه می‌کنیم، تنها بخشی از برنامه را بررسی کنیم. به نوعی، برنامه‌نویسی شیءگرا یک روش برای سازماندهی کد است تا بتوانید روی ۵۰ خط کد تمرکز کنید و آن را درک کنید و در عین حال از بررسی ۹۹۹،۹۵۰ خط دیگر کد در این لحظه چشم‌پوشی کنید.

## ۱۴,۲ شروع کار

مانند بسیاری از جنبه‌های برنامه‌نویسی، لازم است که مفاهیم برنامه‌نویسی شیءگرا را یاد بگیرید تا بتوانید از آن‌ها به‌طور مؤثر استفاده کنید. شما باید این فصل را به‌عنوان روشی برای یادگیری برخی اصطلاحات و مفاهیم و کار از طریق چند مثال ساده به‌منظور ایجاد پایه‌ای برای یادگیری‌های آینده در نظر بگیرید.

نتیجه کلیدی این فصل این است که درک پایه‌ای از نحوه ساخت و عملکرد اشیاء داشته باشید و مهم‌تر از همه، چگونه از قابلیت‌های اشیاء که توسط پایتون و کتابخانه‌های پایتون به ما ارائه می‌شود، استفاده کنیم.

## ۱۴,۳ استفاده از اشیاء

همانطور که مشخص می‌شود، ما از ابتدا در این کتاب از اشیاء استفاده کرده‌ایم. پایتون بسیاری از اشیاء از پیش ساخته شده را برای ما فراهم می‌کند. در اینجا کدی ساده داریم که در آن چند خط اول باید برای شما بسیار ساده و طبیعی به نظر برسد.

```
stuff = list()
stuff.append('python')
stuff.append('chuck')
stuff.sort()
print (stuff[0])
print (stuff.__getitem__(0))
print (list.__getitem__(stuff,0))

# Code: https://www.py4e.com/code3/objects.py
```

به‌جای تمرکز بر این‌که این خطوط چه کاری انجام می‌دهند، بیایید از دیدگاه برنامه‌نویسی شیءگرا بررسی کنیم که در واقع چه رخ می‌دهد. نگران نباشید اگر پاراگراف‌های زیر در نخستین خواندن چندان قابل‌فهم نبودند؛ زیرا هنوز تمامی این اصطلاحات را تعریف نکرده‌ایم.

خط اول یک شیء از نوع **list** می‌سازد، خط دوم و سوم متد **append()** را فراخوانی می‌کنند، خط چهارم متد **sort()** را فراخوانی می‌کند، و خط پنجم، عنصر موجود در **موقعیت صفر** را بازیابی می‌کند.

خط ششم، متد `__getitem__()` را در فهرست **stuff** با پارامتر **صفر** فراخوانی می‌کند.

```
print (stuff.__getitem__(0))
```

خط هفتم، روشی حتی طولانی‌تر و پرجزئیات‌تر برای بازیابی عنصر صفرم فهرست است.

```
print (list.__getitem__(stuff,0))
```

در این قطعه‌کد، ما متد `__getitem__` را در کلاس فهرست فراخوانی می‌کنیم و فهرست و عنصری را که قصد بازیابی آن را داریم، به عنوان پارامتر به آن ارسال می‌کنیم.

سه خط پایانی برنامه با یکدیگر معادل‌اند، اما استفاده از **علامت‌گذاری براکت‌ها** برای دسترسی به عنصری در موقعیت مشخص، بسیار ساده‌تر و مناسب‌تر است.

برای بررسی قابلیت‌های یک شیء، می‌توانیم خروجی تابع `dir()` را مشاهده کنیم:

```
>>> stuff = list()
>>> dir(stuff)
['__add__', '__class__', '__contains__', '__delattr__',
'__delitem__', '__dir__', '__doc__', '__eq__',
```

```
'__format__', '__ge__', '__getattribute__', '__getitem__',
'__gt__', '__hash__', '__iadd__', '__imul__', '__init__',
'__iter__', '__le__', '__len__', '__lt__', '__mul__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__reversed__', '__rmul__', '__setattr__',
'__setitem__', '__sizeof__', '__str__', '__subclasshook__',
'append', 'clear', 'copy', 'count', 'extend', 'index',
'insert', 'pop', 'remove', 'reverse', 'sort']
>>>
```

بخش‌های باقی‌مانده  این فصل تمامی اصطلاحات یادشده را تعریف خواهند کرد؛ بنابراین، پس از پایان فصل حتماً به پاراگراف‌های بالا بازگردید و آن‌ها را دوباره بخوانید تا میزان درک خود را بررسی کنید.

## ۱۴,۴ شروع کار با برنامه‌ها

یک برنامه در ساده‌ترین شکل خود، مقداری ورودی دریافت می‌کند، پردازشی روی آن انجام می‌دهد، و سپس خروجی تولید می‌کند. برنامه تبدیل طبقه آسانسور که پیش‌تر دیدیم، نمونه‌ای بسیار کوتاه اما کامل است که هر سه مرحله را نشان می‌دهد.

```
usf = input('Enter the US Floor Number: ')
wf = int(usf) - 1
print('Non-US Floor Number is',wf)

# Code: https://www.py4e.com/code3/elev.py
```

اگر کمی بیشتر درباره این برنامه بیندیشیم، درمی‌یابیم که «جهان بیرون» وجود دارد و برنامه نیز در برابر آن قرار می‌گیرد. بخش‌های ورودی و خروجی همان نقاطی هستند که برنامه از طریق آن‌ها با جهان بیرون تعامل می‌کند. درون برنامه، کد و داده‌ای قرار دارد که برای انجام وظیفه‌ای که برنامه برای حل آن طراحی شده است، مورد استفاده قرار می‌گیرد.




یکی از روش‌های اندیشیدن به برنامه‌نویسی شیءگرا این است که آن را به چندین «ناحیه» تقسیم می‌کند. هر ناحیه شامل مقداری کد و داده است (مانند یک برنامه کوچک) و تعاملات مشخص و دقیقی با جهان بیرون و با دیگر نواحی درون برنامه دارد.

اگر به کاربرد استخراج پیوندها برگردیم، جایی که از کتابخانه BeautifulSoup استفاده کردیم، می‌توانیم برنامه‌ای را ببینیم که با اتصال چندین شیء مختلف به یکدیگر ساخته شده است تا یک وظیفه واحد را انجام دهد:

```
# To run this, download the BeautifulSoup zip file
# http://www.py4e.com/code3/bs4.zip
# and unzip it in the same directory as this file

import urllib.request, urllib.parse, urllib.error
from bs4 import BeautifulSoup
import ssl

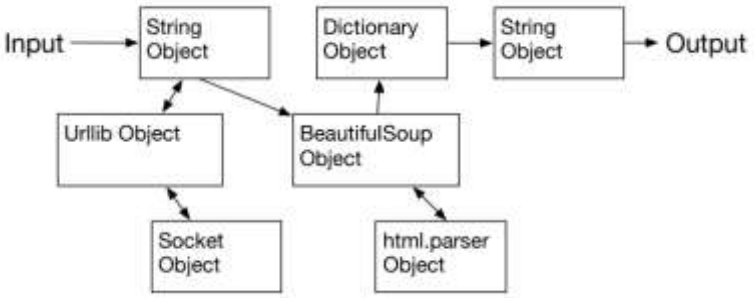
# Ignore SSL certificate errors
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

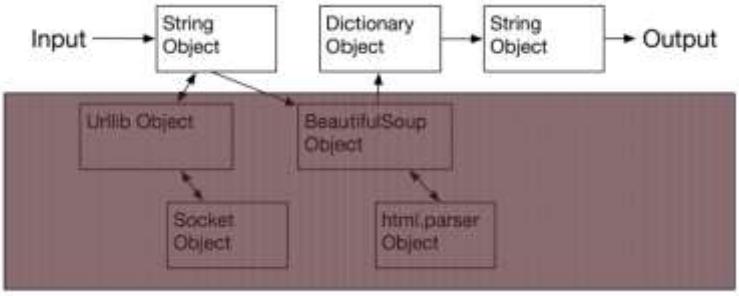
url = input('Enter - ')
html = urllib.request.urlopen(url, context=ctx).read()
soup = BeautifulSoup(html, 'html.parser')

# Retrieve all of the anchor tags
tags = soup('a')
for tag in tags:
    print(tag.get('href', None))

# Code: https://www.py4e.com/code3/urllinks.py
```

ما نشانی اینترنتی (URL) را در قالب یک رشته می‌خوانیم و سپس آن را به کتابخانه **urllib** ارسال می‌کنیم تا داده‌ها را از وب بازیابی کند. کتابخانه urllib برای برقراری اتصال واقعی شبکه و دریافت داده، از کتابخانه **socket** استفاده می‌کند. رشته‌ای را که urllib بازمی‌گرداند، به **BeautifulSoup** می‌سپاریم تا آن را تجزیه کند .BeautifulSoup از شیء **html.parser** بهره می‌گیرد و شیئی را بازمی‌گرداند. سپس متد **()tags** را روی شیء بازگشتی فراخوانی می‌کنیم که یک فرهنگ (دیکشنری) شامل اشیای برچسب (tag objects) را برمی‌گرداند. در ادامه، در میان این برچسب‌ها حلقه می‌زنیم و برای هر برچسب، متد **()get** را فراخوانی می‌کنیم تا مقدار ویژگی **href** را چاپ کنیم. می‌توانیم تصویری از این برنامه و نحوه همکاری اشیا ترسیم کنیم.



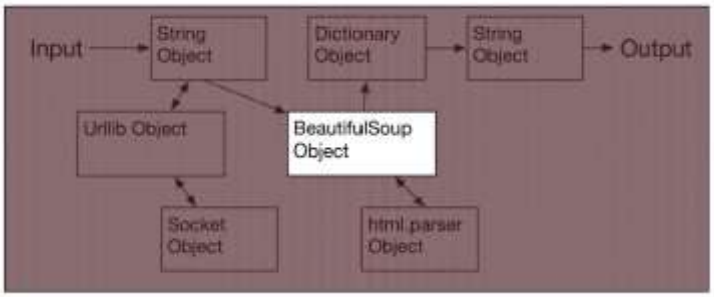



نکته اصلی این نیست که دقیقاً بدانیم این برنامه چگونه کار می‌کند، بلکه این است که ببینیم چگونه شبکه‌ای از اشیای در تعامل با یکدیگر ایجاد می‌کنیم و جابه‌جایی اطلاعات بین این اشیا را «هماهنگ» می‌سازیم تا یک برنامه شکل بگیرد. همچنین مهم است توجه کنیم که وقتی چند فصل قبل به آن برنامه نگاه می‌کردید، می‌توانستید آنچه در برنامه رخ می‌دهد را کاملاً درک کنید، بدون اینکه حتی متوجه شوید برنامه در حال «هماهنگ سازی انتقال داده بین اشیا» است. تنها خطوطی از کد بودند که کار مورد نظر را انجام می‌دادند.

## ۱۴,۵ بخش‌بندی یک مسئله

یکی از مزایای رویکرد شیءگرا این است که می‌تواند پیچیدگی را پنهان کند. برای مثال، در حالی که لازم است بدانیم چگونه از کدهای urllib و BeautifulSoup استفاده کنیم، نیازی نیست از نحوه کار داخلی این کتابخانه‌ها آگاه باشیم. این موضوع به ما اجازه می‌دهد بر بخش‌هایی از مسئله که باید حل کنیم تمرکز کنیم و سایر بخش‌های برنامه را نادیده بگیریم.

این توانایی که فقط بر بخشی از برنامه که برای ما اهمیت دارد تمرکز کنیم و باقی جزئیات را نادیده بگیریم، برای توسعه‌دهندگان اشیایی که از آن‌ها استفاده می‌کنیم نیز مفید است. برای مثال، برنامه‌نویسانی که BeautifulSoup را توسعه می‌دهند، نیازی ندارند بدانند یا برایشان اهمیتی ندارد که ما چگونه صفحه HTML را بازیابی می‌کنیم، کدام بخش‌ها را می‌خواهیم بخوانیم، یا قصد داریم با داده‌های استخراج‌شده از صفحه وب چه کاری انجام دهیم.



## ۱۴,۶ اولین شیء پایتونی ما

در سطحی پایه‌ای، یک شیء در واقع مجموعه‌ای از کد و ساختارهای داده است که از یک برنامه کامل کوچک‌ترند. تعریف یک تابع به ما امکان می‌دهد بخشی از کد را ذخیره کنیم، نامی برای آن تعیین کنیم، و سپس بعدها با استفاده از آن نام، آن بخش از کد را فراخوانی کنیم.

یک شیء می‌تواند شامل چندین تابع باشد (که آن‌ها را متد می‌نامیم) و نیز شامل داده‌ای باشد که این متدها از آن استفاده می‌کنند. داده‌هایی که جزئی از یک شیء هستند، *ویژگی‌ها* (attributes) نامیده می‌شوند.

ما از واژه کلیدی **class** برای تعریف داده و کدی استفاده می‌کنیم که هر یک از اشیا را تشکیل خواهد داد. واژه کلیدی `class` نام کلاس را تعیین می‌کند و سپس یک بلوک تورفتگی شده آغاز می‌کند که در آن ویژگی‌ها (داده) و متدها (کد) را قرار می‌دهیم.

```
class PartyAnimal:

   def __init__(self):
     self.x = 0

   def party(self) :
     self.x = self.x + 1
     print("So far",self.x)

an = PartyAnimal()
an.party()
an.party()
an.party()
```

```
# Code: https://www.py4e.com/code3/party2.py
```

هر متد شبیه یک تابع است؛ با واژه کلیدی **def** آغاز می‌شود و از یک بلوک تورفتگی‌شده کد تشکیل شده است.

نخستین متد، متدی با نام ویژه `__init__()` است. این متد هنگام ایجاد شیء فراخوانی می‌شود تا هرگونه تنظیم اولیه برای داده‌هایی را که قصد داریم در شیء ذخیره کنیم، انجام دهد. در این کلاس، ویژگی **x** را با استفاده از *نشانه‌گذاری نقطه‌ای* تخصیص می‌دهیم و مقدار اولیه آن را صفر قرار می‌دهیم.

```
self.x = 0
```

متد دیگر با نام **party** تعریف شده است. تمامی متدها پارامتر نخست ویژه‌ای دارند که بنا بر قرارداد آن را **self** می‌نامیم. این پارامتر نخست به ما امکان می‌دهد به نمونه شیء دسترسی داشته باشیم تا بتوانیم با استفاده از نشانه‌گذاری نقطه‌ای، ویژگی‌ها را تنظیم کرده و متدها را فراخوانی کنیم.

همان‌گونه که واژه کلیدی **def** موجب اجرای کد تابع نمی‌شود، واژه کلیدی **class** نیز یک شیء ایجاد نمی‌کند. بلکه واژه class، الگویی را تعریف می‌کند که مشخص می‌کند هر شیء از نوع **PartyAnimal** شامل چه داده‌ها و چه کدی خواهد بود.

کلاس مانند **قالب شیرینی‌پزی** است و اشیایی که با استفاده از کلاس ساخته می‌شوند مانند **شیرینی‌ها** هستند. شما روی قالب شیرینی خامه نمی‌ریزید؛ بلکه خامه را روی خود شیرینی‌ها می‌ریزید و می‌توانید بر روی هر شیرینی خامه‌ای متفاوت بگذارید.



اگر ادامه این برنامه نمونه را دنبال کنیم، نخستین خط قابل‌اجرای کد را مشاهده خواهیم کرد:

```
an = PartyAnimal()
```

در این نقطه، ما به پایتون دستور می‌دهیم که یک شیء یا نمونه‌ای از کلاس **PartyAnimal** بسازد (یعنی ایجاد کند). این کار ظاهری شبیه به یک فراخوانی تابع دارد، با این تفاوت که فراخوانی مستقیماً روی خود کلاس انجام می‌شود. پایتون شیء را با داده‌ها و متدهای مناسب می‌سازد و سپس آن را بازمی‌گرداند؛ شیئی که در ادامه به متغیر **an** اختصاص داده می‌شود.

به یک معنا، این فرایند بسیار شبیه خط زیر است که از ابتدای کار از آن استفاده کرده‌ایم:

```
counts = dict()
```

در این‌جا به پایتون دستور می‌دهیم که با استفاده از قالب **dict** (که از پیش در پایتون وجود دارد) یک شیء بسازد، نمونه دیکشنری را بازگرداند، و آن را به متغیر **counts** اختصاص دهد.

وقتی از کلاس **PartyAnimal** برای ساختن یک شیء استفاده می‌کنیم، متغیر **an** برای اشاره به آن شیء به‌کار می‌رود. ما از **an** برای دسترسی به کد و داده‌های مربوط به آن نمونه خاص از کلاس PartyAnimal استفاده می‌کنیم.

هر شیء یا نمونه PartyAnimal شامل یک متغیر به نام **x** و یک متد یا تابع به نام **party** است. متد **party** را در این خط فراخوانی می‌کنیم:

```
an.party()
```

هنگامی که متد **party** فراخوانی می‌شود، پارامتر نخست آن (که بنا بر قرارداد **self** نامیده می‌شود) به همان نمونه مشخص از شیء **PartyAnimal** اشاره می‌کند که متد party از آن فراخوانی شده است. درون متد **party**، خط زیر را مشاهده می‌کنیم:

```
self.x = self.x + 1
```

این ساختار که از عملگر نقطه استفاده می‌کند، در واقع به معنای «متغیر x درون self » است. هر بار که متد **party()** فراخوانی می‌شود، مقدار داخلی **x** یک واحد افزایش می‌یابد و سپس این مقدار چاپ می‌شود.

خط زیر، روش دیگری برای فراخوانی متد **party** درون شیء **an** است:

```
PartyAnimal.party(an)
```

در این نسخه جایگزین، ما به کد درون کلاس دسترسی پیدا می‌کنیم و به‌طور صریح اشاره‌گر شیء **an** را به‌عنوان پارامتر نخست (یعنی **self** درون متد) ارسال می‌کنیم.

می‌توان **an.party()** را شکل کوتاه‌شده خط بالا در نظر گرفت.

هنگامی که برنامه اجرا می‌شود، خروجی زیر را تولید می‌کند:

```
So far 1
So far 2
So far 3
So far 4
```

شیء ساخته می‌شود و متد **party** چهار بار فراخوانی می‌گردد؛ در هر فراخوانی، مقدار متغیر **x** درون شیء **an** یک واحد افزایش می‌یابد و چاپ می‌شود.

## ۱۴,۷ کلاس‌ها به‌عنوان نوع‌ها

همان‌گونه که دیده‌ایم، در پایتون تمامی متغیرها دارای یک **نوع** هستند. می‌توانیم از تابع داخلی **dir** برای بررسی قابلیت‌های یک متغیر استفاده کنیم. همچنین می‌توانیم از **type** و **dir** برای کلاس‌هایی که خودمان ایجاد می‌کنیم نیز بهره ببریم.

```
class PartyAnimal:

   def __init__(self):
     self.x = 0

   def party(self) :
     self.x = self.x + 1
     print("So far",self.x)

an = PartyAnimal()
print ("Type", type(an))
print ("Dir ", dir(an))
print ("Type", type(an.x))
```

```
print ("Type", type(an.party))

# Code: https://www.py4e.com/code3/party3.py
```

وقتی این برنامه اجرا می‌شود، خروجی زیر را تولید می‌کند:

```
Type <class '__main__.PartyAnimal'>
Dir  ['__class__', '__delattr__', ...
'__sizeof__', '__str__', '__subclasshook__',
'__weakref__', 'party', 'x']
Type <class 'int'>
Type <class 'method'>
```

می‌توانید ببینید که با استفاده از کلمه کلیدیclass ، یک نوع جدید ایجاد کرده‌ایم. از خروجی دستورdir ، می‌توان دید که هم ویژگی عددی x و هم روش party در شیء موجود هستند.

## ۱۴,۸ چرخه حیات شیء

در مثال‌های قبلی، یک کلاس (الگو) تعریف کرده‌ایم، از آن کلاس برای ایجاد یک نمونه از آن کلاس (شیء) استفاده کرده‌ایم و سپس از آن نمونه استفاده کرده‌ایم. هنگامی که برنامه تمام می‌شود، تمام متغیرها دور ریخته می‌شوند. معمولاً زیاد به ایجاد و تخریب متغیرها فکر نمی‌کنیم، اما اغلب زمانی که اشیاء ما پیچیده‌تر می‌شوند، نیاز داریم که در داخل شیء اقداماتی انجام دهیم تا هنگام ساخت شیء، آن را تنظیم کنیم و شاید هنگام دور ریختن شیء، تمیزکاری‌های لازم را انجام دهیم.

اگر بخواهیم شیء ما از این لحظات ساخت و تخریب آگاه باشد، باید روش‌های خاصی را به شیء خود اضافه کنیم.

```
class PartyAnimal:

   def __init__(self):
     self.x = 0
     print('I am constructed')

   def party(self) :
     self.x = self.x + 1
     print('So far',self.x)

   def __del__(self):
     print('I am destructed', self.x)

an = PartyAnimal()
an.party()
an.party()
an = 42
```

```
print('an contains',an)

# Code: https://www.py4e.com/code3/party4.py
```

وقتی این برنامه اجرا می‌شود، خروجی زیر را تولید می‌کند:

```
I am constructed
So far 1
So far 2
I am destructed 2
an contains 42
```

هنگامی که پایتون شیء ما را می‌سازد، متد __init__ را فراخوانی می‌کند تا این امکان را در اختیار ما قرار دهد که مقادیر اولیه یا پیش‌فرضی برای شیء تنظیم کنیم. زمانی که پایتون به خط زیر می‌رسد:

```
an = 42
```

در واقع، "شیء ما را دور می‌اندازد" تا بتواند از متغیر an برای ذخیره مقدار ۴۲ استفاده کند. درست در لحظه‌ای که شیء an در حال "تخریب" است، کد ویرایش‌گر (destructor) ما **(del)** فراخوانی می‌شود. نمی‌توانیم از تخریب متغیر جلوگیری کنیم، اما می‌توانیم قبل از اینکه شیء ما دیگر وجود نداشته باشد، هرگونه تمیزکاری ضروری را انجام دهیم.

هنگام توسعه اشیاء، رایج است که یک سازنده (constructor) به شیء اضافه کنیم تا مقادیر اولیه برای شیء تنظیم شود. نسبتاً نادر است که برای یک شیء به یک ویرایش‌گر (destructor) نیاز داشته باشیم.

## ۱۴,۹ چندین نمونه

تا اینجا، ما یک کلاس تعریف کرده‌ایم، یک شیء واحد ساخته‌ایم، از آن شیء استفاده کرده‌ایم و سپس آن را دور انداخته‌ایم. با این حال، قدرت واقعی در برنامه‌نویسی شی‌گرا زمانی است که ما چندین نمونه از کلاس خود بسازیم.

هنگامی که چندین شیء از کلاس خود می‌سازیم، ممکن است بخواهیم مقادیر اولیه متفاوتی برای هر یک از اشیاء تنظیم کنیم. می‌توانیم داده‌ها را به سازنده‌ها ارسال کنیم تا هر شیء مقدار اولیه متفاوتی داشته باشد.

```
class PartyAnimal:

   def __init__(self, nam):
     self.x = 0
     self.name = nam
     print(self.name,'constructed')

   def party(self) :
```

```
        self.x = self.x + 1
        print(self.name,'party count',self.x)

s = PartyAnimal('Sally')
s.party()
j = PartyAnimal('Jim')

j.party()
s.party()

# Code: https://www.py4e.com/code3/party5.py
```

سازنده (constructor) هم پارامتر **self** را دارد که به نمونه شیء اشاره می‌کند و هم پارامترهای اضافی را که در هنگام ایجاد شیء به سازنده ارسال می‌شوند:

```
s = PartyAnimal('Sally')
```

در درون سازنده، خط دوم پارامتر **(nam)** را که به سازنده ارسال شده است، در ویژگی **name** درون نمونه شیء قرار می‌دهد.

```
self.name = nam
```

خروجی برنامه نشان می‌دهد که هر یک از اشیاء ( **s** و **j**) نسخه‌های مستقل خود را از متغیرهای **x** و **nam** در اختیار دارند:

```
Sally constructed
Sally party count 1
Jim constructed
Jim party count 1
Sally party count 2
```

## ۱۴,۱۰ وراثت

یکی دیگر از ویژگی‌های قدرتمند برنامه‌نویسی شی‌گرا این است که می‌توانیم یک کلاس جدید ایجاد کنیم که از یک کلاس موجود گسترش یافته باشد. وقتی یک کلاس را گسترش می‌دهیم، کلاس اصلی را کلاس والد و کلاس جدید را کلاس فرزند می‌نامیم.

برای این مثال، کلاس `PartyAnimal` را به فایل جداگانه‌ای منتقل می‌کنیم. سپس می‌توانیم کلاس `PartyAnimal` را در فایل جدید «وارد» کرده و آن را گسترش دهیم، به این صورت:

```
from party import PartyAnimal

class CricketFan(PartyAnimal):

   def __init__(self, nam) :
```

```
        super().__init__(nam)
        self.points = 0

    def six(self):
        self.points = self.points + 6
        self.party()
        print(self.name,"points",self.points)

s = PartyAnimal("Sally")
s.party()
j = CricketFan("Jim")
j.party()
j.six()
print(dir(j))

# Code: https://www.py4e.com/code3/party6.py
```

وقتی کلاس CricketFan را تعریف می‌کنیم، مشخص می‌کنیم که در حال گسترش کلاس PartyAnimal هستیم. این بدان معناست که تمامی متغیرها (مانند x و متدها مانند party) از کلاس PartyAnimal به کلاس CricketFan ارث برده می‌شوند. برای مثال، در متد six در کلاس CricketFan، متد party از کلاس PartyAnimal فراخوانی می‌شود.

ما از یک سینتکس خاص در متد __init__() در کلاس CricketFan استفاده می‌کنیم تا مطمئن شویم که متد __init__() در کلاس PartyAnimal فراخوانی می‌شود، به طوری که هر نوع تنظیمات مورد نیاز در کلاس PartyAnimal انجام شود، علاوه بر تنظیماتی که برای گسترش‌های کلاس CricketFan نیاز داریم.

```
    def __init__(self, nam) :
        super().__init__(nam)
        self.points = 0
```

دستور super() به پایتون می‌گوید که متد __init__ را در کلاسی که در حال گسترش آن هستیم فراخوانی کند. کلاس PartyAnimal کلاس والد (super) است و کلاس CricketFan کلاس فرزند (sub) می‌باشد.

هنگامی که برنامه اجرا می‌شود، ما s و j را به عنوان نمونه‌های مستقل از PartyAnimal و CricketFan ایجاد می‌کنیم. شیء j قابلیت‌های اضافی نسبت به شیء s دارد.

```
Sally constructed
Sally party count 1
Jim constructed
Jim party count 1
Jim party count 2
Jim points 6
['__class__', '__delattr__', ... '__weakref__',
'name', 'party', 'points', 'six', 'x']
```

در خروجی دستور dir برای شیء j (نمونه‌ای از کلاس CricketFan)، مشاهده می‌کنیم که این شیء ویژگی‌ها و متدهای کلاس والد را دارد، همچنین ویژگی‌ها و متدهایی که هنگام گسترش کلاس برای ایجاد کلاس CricketFan به آن اضافه شده است.

## ۱۴,۱۱ خلاصه

این یک معرفی سریع به برنامه نویسی شی‌گرا است که عمدتاً بر روی اصطلاحات و نحو تعریف و استفاده از اشیاء متمرکز است. بیایید به سرعت کدی را که در ابتدای فصل بررسی کردیم مرور کنیم. در این مرحله شما باید کاملاً متوجه باشید که چه چیزی در حال انجام است.

```
stuff = list()
stuff.append('python')
stuff.append('chuck')
stuff.sort()
print (stuff[0])
print (stuff.__getitem__(0))
print (list.__getitem__(stuff,0))

# Code: https://www.py4e.com/code3/objects.py
```

خط اول یک شیء لیست ایجاد می‌کند. زمانی که پایتون شیء لیست را ایجاد می‌کند، متد سازنده **(init)** فراخوانی می‌شود تا ویژگی‌های داده داخلی که برای ذخیره داده‌های لیست استفاده می‌شوند، تنظیم شوند. ما هیچ پارامتری به سازنده ارسال نکرده‌ایم. زمانی که سازنده برمی‌گردد، از متغیر stuff برای اشاره به نمونه بازگشتی از کلاس لیست استفاده می‌کنیم. خطوط دوم و سوم متد append را با یک پارامتر فراخوانی می‌کنند تا یک آیتم جدید را به انتهای لیست اضافه کنند و ویژگی‌های داخل stuff به روز شوند. سپس در خط چهارم، متد sort را بدون پارامتر فراخوانی می‌کنیم تا داده‌های داخل شیء stuff مرتب شوند.

سپس اولین آیتم لیست را با استفاده از براکت‌های مربع (که یک میانبر برای فراخوانی متد __getitem__ در داخل stuff هستند) چاپ می‌کنیم. این معادل فراخوانی متد __getitem__ در کلاس لیست است که شیء stuff را به عنوان اولین پارامتر و موقعیتی که به دنبالش هستیم را به عنوان دومین پارامتر می‌فرستد.در انتهای برنامه، شیء stuff دور ریخته می‌شود، اما نه قبل از فراخوانی ویرانگر **(del)** تا شیء بتواند هرگونه تمیزکاری لازم را انجام دهد.

این‌ها اصول اولیه برنامه‌نویسی شی‌گرا هستند. جزئیات بیشتری در مورد نحوه استفاده بهینه از رویکردهای شی‌گرا هنگام توسعه برنامه‌های بزرگ و کتابخانه‌ها وجود دارد که از حوصله این فصل خارج است.

## ۱۴,۱۲ واژه‌نامه

**ویژگی(attribute)** : یک متغیر که جزئی از یک کلاس است.

**کلاس(class)** : الگویی که برای ساخت یک شیء می‌توان از آن استفاده کرد. ویژگی‌ها و متدهایی که شیء را تشکیل می‌دهند، در کلاس تعریف می‌شوند.

**کلاس فرزند(child class)** : کلاسی جدید که هنگام گسترش یک کلاس والد ایجاد می‌شود. کلاس فرزند تمام ویژگی‌ها و متدهای کلاس والد را به ارث می‌برد.

**سازنده(constructor)** : یک متد ویژه با نام خاص **(init)** که در لحظه استفاده از یک کلاس برای ساخت یک شیء فراخوانی می‌شود. معمولاً از این متد برای تنظیم مقادیر اولیه شیء استفاده می‌شود.

**ویرانگر(destructor)** : یک متد ویژه با نام خاص **(del)** که درست قبل از از بین رفتن یک شیء فراخوانی می‌شود. ویرانگرها به ندرت استفاده می‌شوند.

**ارث‌بری(inheritance)** : هنگامی که یک کلاس جدید (فرزند) با گسترش یک کلاس موجود (والد) ایجاد می‌شود. کلاس فرزند تمام ویژگی‌ها و متدهای کلاس والد را دارد و علاوه بر آن ویژگی‌ها و متدهای اضافی تعریف شده توسط کلاس فرزند را نیز شامل می‌شود.

**متد(method)** : یک تابع که در داخل یک کلاس و اشیایی که از آن کلاس ساخته شده‌اند قرار دارد. برخی الگوهای شی‌گرای برنامه‌نویسی از واژه «پیام» (message) به جای «متد» برای توصیف این مفهوم استفاده می‌کنند.

**شیء(object)** : یک نمونه ساخته شده از یک کلاس. شیء شامل تمام ویژگی‌ها و متدهایی است که توسط کلاس تعریف شده‌اند. برخی مستندات شی‌گرای برنامه‌نویسی از واژه «نمونه» (instance) به جای «شیء» به صورت متناوب استفاده می‌کنند.

**کلاس والد(parent class)** : کلاسی که برای ایجاد یک کلاس فرزند جدید گسترش می‌یابد. کلاس والد تمام متدها و ویژگی‌های خود را به کلاس فرزند جدید منتقل می‌کند.

# فصل پانزدهم

# استفاده از پایگاه‌داده‌ها وSQL

## ۱۵,۱ پایگاه‌داده چیست؟

پایگاه‌داده، فایلی است که برای ذخیره‌سازی داده‌ها سازمان‌دهی شده است. اغلب پایگاه‌داده‌ها در ساختاری مشابه یک «دیکشنری» سازمان‌دهی می‌شوند؛ بدین معنا که داده‌ها را از طریق «کلیدها» به «مقادیر» نگاشت می‌کنند. بزرگ‌ترین تفاوت این است که پایگاه‌داده روی دیسک (یا سایر حافظه‌های دائمی) نگه‌داری می‌شود، بنابراین **پس از پایان اجرای برنامه نیز باقی می‌ماند.** از آن‌جا که پایگاه‌داده روی حافظه دائمی ذخیره می‌شود، می‌تواند بسیار بیشتر از یک دیکشنری داده ذخیره کند، در حالی‌که دیکشنری محدود به حافظه RAM رایانه است.

همانند دیکشنری، نرم‌افزار پایگاه‌داده طوری طراحی شده است که **درج و بازیابی داده** حتی در صورت وجود مقادیر بسیار زیاد، بسیار سریع انجام شود.این کار از طریق **ایجاد ایندکس‌ها** هنگام افزودن داده به پایگاه‌داده انجام می‌شود، تا رایانه بتواند مستقیماً و با سرعت بالا به یک ورودی خاص دسترسی پیدا کند.سامانه‌های پایگاه‌داده متعددی برای کاربردهای گوناگون وجود دارند؛ از جمله: Oracle، MySQL، Microsoft SQL Server، PostgreSQLو.SQLite در این کتاب بر **SQLite** تمرکز می‌کنیم، زیرا بسیار رایج است و از پیش در پایتون تعبیه شده است SQLite .برای استفاده درون سایر برنامه‌ها طراحی شده تا پشتیبانی پایگاه‌داده را در قلب همان برنامه فراهم کند.

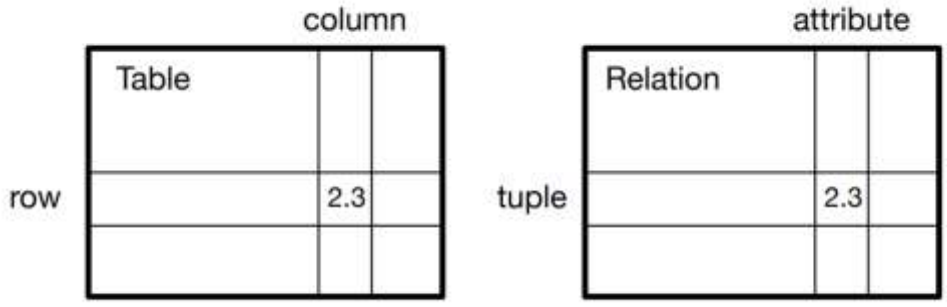
به‌عنوان مثال، مرورگر **Firefox** نیز از SQLite درونی استفاده می‌کند،همچنین بسیاری از نرم‌افزارهای دیگر نیز همین‌گونه‌اند.

http://sqlite.org/

SQLite برای برخی از مسائل مربوط به پردازش داده که در حوزه انفورماتیک مشاهده می‌کنیم، بسیار مناسب است.

## ۱۵,۲ مفاهیم پایگاه‌داده

در نگاه نخست، یک پایگاه‌داده شبیه یک صفحه‌گسترده با چندین برگه به‌نظر می‌رسد. ساختارهای اصلی داده در یک پایگاه داده عبارت‌اند از :**جدول‌ها، سطرها و ستون‌ها.**




در توصیف‌های فنی پایگاه‌داده‌های رابطه‌ای، مفاهیم **جدول، سطر و ستون** به‌ترتیب با واژه‌های رسمی‌تر **رِلیشن(relation)** ، **تاپل (tuple)** و **ویژگی (attribute)** شناخته می‌شوند. در این فصل از همان اصطلاحات غیررسمی‌تر استفاده خواهیم کرد.

## ۱۵,۳ مرورگر پایگاه‌داده برایSQLite

در حالی که در این فصل تمرکز ما بر استفاده از پایتون برای کار با داده‌ها در فایل‌های پایگاه‌داده SQLite خواهد بود، بسیاری از عملیات را می‌توان به‌صورت راحت‌تر با استفاده از نرم‌افزاری به نام **Database Browser for SQLite** انجام داد که به صورت رایگان در دسترس است از نشانی:

http://sqlitebrowser.org/

با استفاده از مرورگر پایگاه‌داده می‌توان به‌آسانی جدول ایجاد کرد، داده وارد نمود، داده‌ها را ویرایش کرد یا کوئری‌های ساده SQL را بر روی داده‌های موجود در پایگاه‌داده اجرا کرد.

به نوعی، مرورگر پایگاه داده مشابه یک ویرایشگر متن هنگام کار با فایل‌های متنی است. وقتی می‌خواهید یک یا چند عملیات محدود روی یک فایل متنی انجام دهید، کافی است آن را در یک ویرایشگر متن باز کرده و تغییرات موردنظر را اعمال کنید. اما زمانی که نیاز به انجام تعداد زیادی تغییر دارید، معمولاً یک برنامه ساده پایتون می‌نویسید. همین الگو هنگام کار با پایگاه داده‌ها نیز دیده می‌شود: عملیات ساده را در مدیریت پایگاه داده انجام می‌دهید و عملیات پیچیده‌تر را بهتر است با پایتون پیاده‌سازی کنید.

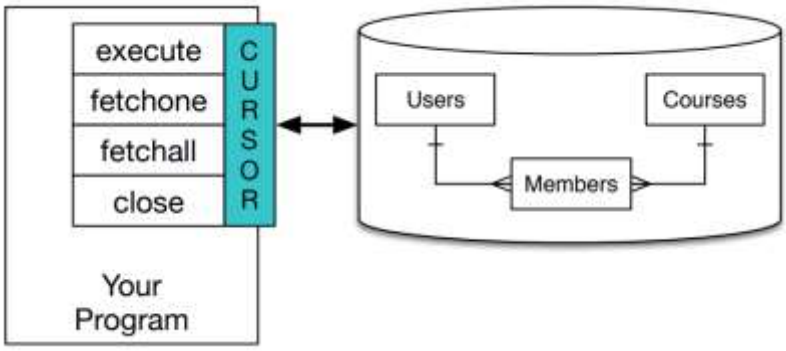
## ۱۵,۴ ایجاد یک جدول پایگاه‌داده

پایگاه داده‌ها نسبت به فهرست‌ها (lists) یا دیکشنری‌های پایتون به ساختار تعریف‌شده‌تری نیاز دارند. وقتی یک جدول پایگاه داده ایجاد می‌کنیم، باید از پیش نام هر یک از ستون‌ها و نوع داده‌ای را که قصد ذخیره‌سازی آن در هر ستون داریم، به پایگاه‌داده اعلام کنیم.

وقتی نرم‌افزار پایگاه‌داده نوع داده هر ستون را بداند، می‌تواند کارآمدترین روش را برای ذخیره‌سازی و بازیابی داده بر اساس نوع داده انتخاب کند. می‌توانید انواع داده‌های پشتیبانی‌شده در SQLite را در نشانی زیر مشاهده کنید:

http://www.sqlite.org/datatypes.html

تعریف ساختار داده‌ها در ابتدا ممکن است تا حدودی نامناسب یا دشوار به‌نظر برسد، اما نتیجه آن **دسترسی بسیار سریع به داده‌ها** است، حتی هنگامی که پایگاه‌داده حجم زیادی از اطلاعات را در خود جای داده باشد.




کدی که برای ایجاد یک فایل پایگاه‌داده و ساخت جدولی به نام **Track** با دو ستون در پایگاه‌داده استفاده می‌شود، به صورت زیر است:

```
import sqlite3

conn = sqlite3.connect('music.sqlite')
cur = conn.cursor()

cur.execute('DROP TABLE IF EXISTS Track')
cur.execute('CREATE TABLE Track (title TEXT, plays INTEGER)')

conn.close()

# Code: https://www.py4e.com/code3/db1.py
```

عملیات **connect** یک «اتصال» به پایگاه‌داده‌ای برقرار می‌کند که در فایل **music.sqlite** در پوشه فعلی ذخیره شده است. اگر این فایل وجود نداشته باشد، ایجاد خواهد شد. دلیل آن‌که این عملیات «اتصال» نامیده می‌شود این است که گاهی پایگاه‌داده روی یک «سرور پایگاه‌داده» جداگانه قرار دارد، سروری که از سروری که برنامه ما روی آن اجرا می‌شود مستقل است. در مثال‌های ساده ما، پایگاه‌داده تنها یک فایل محلی در همان پوشه‌ای است که کد پایتون در آن قرار دارد.

**Cursor** یا «مکان‌نما» شبیه یک دسته فایل (file handle) است که می‌توانیم از آن برای انجام عملیات روی داده‌های ذخیره‌شده در پایگاه‌داده استفاده کنیم. فراخوانی تابع **cursor()** از نظر مفهومی بسیار شبیه فراخوانی تابع **open()** هنگام کار

با فایل‌های متنی است. هنگامی که مکان‌نما را در اختیار داشته باشیم، می‌توانیم با استفاده از متد **execute()** شروع به اجرای فرمان‌ها بر روی محتوای پایگاه‌داده کنیم.

فرمان‌های پایگاه‌داده در قالب زبانی خاص بیان می‌شوند که در میان بسیاری از تولیدکنندگان پایگاه‌داده استاندارد شده است تا ما بتوانیم تنها یک زبان پایگاه‌داده یاد بگیریم. این زبان **Structured Query Language** نام دارد که به اختصار **SQL** گفته می‌شود.

http://en.wikipedia.org/wiki/SQL

در مثال ما، دو فرمان SQL را در پایگاه‌داده اجرا می‌کنیم. مطابق یک قرارداد رایج، واژه‌های کلیدی SQL را با **حروف بزرگ** نمایش خواهیم داد و بخش‌هایی از فرمان که خودمان اضافه می‌کنیم (مانند نام جدول‌ها و ستون‌ها) را با **حروف کوچک** نشان می‌دهیم.

نخستین فرمانSQL ، جدول **Track** را در صورتی که وجود داشته باشد از پایگاه‌داده حذف می‌کند. این الگو تنها برای آن است که بتوانیم برنامه نمونه را بارها اجرا کنیم تا جدول Track را ایجاد کنیم، بدون آن‌که با خطا مواجه شویم. توجه داشته باشید که فرمان **DROP TABLE**، جدول و تمام محتوای آن را از پایگاه‌داده حذف می‌کند؛ یعنی هیچ «عملیات بازگردانی» (undo) در این مورد وجود ندارد.

```
cur.execute('DROP TABLE IF EXISTS Track ')
```

فرمان دوم، جدولی به نام **Track** ایجاد می‌کند که دارای ستونی متنی با نام **title** و ستونی عددی با نام **plays** است.

```
cur.execute('CREATE TABLE Track (title TEXT, plays INTEGER)')
```

اکنون که جدولی به نام **Track** ایجاد کرده‌ایم، می‌توانیم با استفاده از عملگر **INSERT** در SQL مقداری داده در آن درج کنیم. بار دیگر، از برقراری اتصال به پایگاه داده و دریافت مکان نما آغاز می‌کنیم. سپس می‌توانیم با استفاده از مکان نما فرمان‌های SQLرا اجرا کنیم.

فرمان **INSERT** در SQL مشخص می‌کند که در حال استفاده از کدام جدول هستیم و سپس یک سطر جدید را تعریف می‌کند؛ این کار از طریق فهرست‌کردن ستون‌هایی که می‌خواهیم مقداردهی کنیم **(title, plays)** و سپس تعیین **VALUES** مربوط به آن‌ها انجام می‌شود. ما مقادیر را به‌صورت علامت سؤال **(?, ?)** مشخص می‌کنیم تا نشان دهیم که مقادیر واقعی، به‌صورت یک **تاپل** (`('My Way', 15)`) به‌عنوان پارامتر دوم به فراخوانی `execute()` ارسال می‌شوند.

```
import sqlite3

conn = sqlite3.connect('music.sqlite')
cur = conn.cursor()

cur.execute('INSERT INTO Track (title, plays) VALUES (?, ?)',
    ('Thunderstruck', 20))
cur.execute('INSERT INTO Track (title, plays) VALUES (?, ?)',
    ('My Way', 15))
```

```
conn.commit()

print('Track:')
cur.execute('SELECT title, plays FROM Track')
for row in cur:
     print(row)

cur.execute('DELETE FROM Track WHERE plays < 100')
conn.commit()

cur.close()

# Code: https://www.py4e.com/code3/db2.py
```

ابتدا با استفاده از دستور **INSERT**، دو سطر در جدول وارد می‌کنیم و با فراخوانی **commit()** داده‌ها را مجبور می‌کنیم که در فایل پایگاه‌داده نوشته شوند.

Tracks

| title | plays |
|---|---|
| Thunderstruck | 20 |
| My Way | 15 |

سپس از فرمان **SELECT** برای بازیابی سطرهایی که به‌تازگی درج کرده‌ایم، استفاده می‌کنیم. در فرمان SELECT مشخص می‌کنیم که کدام ستون‌ها را می‌خواهیم **(title, plays)** و از کدام جدول می‌خواهیم داده‌ها را دریافت کنیم. پس از اجرای عبارت SELECT ، مقدار بازگشتی مکان‌نما (cursor) شیئی است که می‌توانیم آن را در یک حلقه `for` پیمایش کنیم. برای افزایش کارایی، مکان‌نما هنگام اجرای فرمان SELECT ، تمام داده‌ها را به‌طور کامل از پایگاه‌داده نمی‌خواند. بلکه داده‌ها به‌صورت «در لحظه» و به‌تدریج، هم‌زمان با پیمایش سطرها در حلقه `for` خوانده می‌شوند.

خروجی برنامه به صورت زیر است:

```
Track:
('Thunderstruck', 20)
('My Way', 15)
```

حلقه `for` ما دو سطر پیدا می‌کند، و هر سطر به‌صورت یک **تاپل پایتونی** است که مقدار نخست آن عنوان **(title)** و مقدار دوم تعداد پخش **(plays)** است. در انتهای برنامه، یک فرمان SQL برای **DELETE** اجرا می‌کنیم تا سطرهایی را که تازه ایجاد کرده‌ایم حذف کنیم و بتوانیم برنامه را چندین بار اجرا کنیم. فرمان DELETE نمونه‌ای از استفاده عبارت **WHERE** را نشان می‌دهد که به ما امکان می‌دهد معیار انتخابی تعیین کنیم، به‌طوری که پایگاه‌داده فقط سطرهایی را که با این معیار مطابقت دارند

تحت تأثیر قرار دهد.در این مثال، معیار انتخاب‌شده با همه سطرها مطابقت دارد؛ بنابراین جدول را خالی می‌کنیم تا بتوانیم برنامه را بارها اجرا کنیم. پس از اجرای DELETE ، متد **commit()** را نیز فراخوانی می‌کنیم تا حذف داده‌ها به‌طور کامل در پایگاه‌داده اعمال شود.

## ۱۵,۵ خلاصه‌ای از زبان SQL

تا این‌جا در مثال‌های پایتون این کتاب از **Structured Query Language (SQL)** استفاده کرده‌ایم و بسیاری از مفاهیم پایه‌ای مربوط به دستورات SQL را پوشش داده‌ایم. در این بخش، زبان SQL را به‌طور مشخص بررسی می‌کنیم و مروری بر ساختار نحوی آن ارائه می‌دهیم.

از آن‌جا که تعداد بسیار زیادی سیستم پایگاه‌داده از شرکت‌های مختلف وجود دارد، زبان SQL استانداردسازی شد تا بتوانیم به‌صورت قابل‌انتقال و مشترک با پایگاه‌داده‌های گوناگون ارتباط برقرار کنیم.

یک پایگاه‌داده رابطه‌ای از **جدول‌ها، سطرها و ستون‌ها** تشکیل می‌شود. ستون‌ها معمولاً دارای یک نوع داده هستند مانند **text**، **numeric**، یا **date** . هنگام ایجاد یک جدول، نام و نوع ستون‌ها را مشخص می‌کنیم:

```
CREATE TABLE Track (title TEXT, plays INTEGER)
```

برای وارد کردن یک سطر جدید به یک جدول، از دستور **INSERT** در SQL استفاده می‌کنیم:

```
INSERT INTO Track (title, plays) VALUES ('My Way', 15)
```

دستور **INSERT** نام جدول را مشخص می‌کند، سپس فهرستی از فیلدها یا ستون‌هایی را که می‌خواهیم در سطر جدید مقداردهی کنیم بیان می‌کند، و بعد از کلیدواژه **VALUES**، فهرستی از مقادیر متناظر برای هر یک از آن فیلدها می‌آید.

دستور **SELECT** در SQL برای بازیابی سطرها و ستون‌ها از پایگاه‌داده به‌کار می‌رود. عبارت SELECT به شما اجازه می‌دهد تعیین کنید کدام ستون‌ها را می‌خواهید دریافت کنید و همچنین از یک عبارت **WHERE** برای مشخص کردن سطرهایی که مایل به مشاهده آن‌ها هستید استفاده کنید. این دستور همچنین یک عبارت اختیاری **ORDER BY** ارائه می‌دهد تا ترتیب مرتب‌سازی سطرهای بازگردانده شده را کنترل کنید.

```
SELECT * FROM Track WHERE title = 'My Way'
```

استفاده از علامت ***** به این معناست که شما از پایگاه‌داده می‌خواهید **تمام ستون‌ها** را برای هر سطری که با عبارت WHERE مطابقت دارد برگرداند.

توجه داشته باشید که برخلاف پایتون، در عبارت **WHERE** در SQL برای آزمون برابری از **علامت مساوی یک‌تایی (=)** استفاده می‌شود، نه مساوی دوتایی.(==) دیگر عملگرهای منطقی قابل استفاده در عبارت WHERE عبارت‌اند از: <, >, <=, >=

=! ,=< و همچنین واژه‌های منطقی **AND**و **OR**و نیز استفاده از پرانتز برای ساختن عبارت‌های منطقی پیچیده. می‌توانید درخواست کنید که سطرهای بازگردانده‌شده بر اساس یکی از فیلدها مرتب شوند، به صورت زیر:

```
SELECT title,plays FROM Track ORDER BY title
```

می‌توان با استفاده از دستور **UPDATE** درSQL ، یک یا چند ستون را در یک یا چند سطر از یک جدول به‌روزرسانی کرد، به این صورت:

```
UPDATE Track SET plays = 16 WHERE title = 'My Way'
```

دستور **UPDATE** نام جدول را مشخص می‌کند و سپس پس از کلیدواژه **SET** فهرستی از فیلدها و مقادیر جدیدی را که باید تغییر کنند ارائه می‌دهد. پس از آن، یک عبارت اختیاری **WHERE** می‌آید که تعیین می‌کند کدام سطرها باید به‌روزرسانی شوند.

یک دستور UPDATE تمام سطرهایی را که با معیار WHERE مطابقت دارند تغییر خواهد داد. اگر عبارت WHERE مشخص نشود، این دستور **تمام سطرهای جدول** را به‌روزرسانی می‌کند. برای حذف یک سطر، باید در دستور **DELETE** یک عبارت WHERE وجود داشته باشد. عبارت WHERE تعیین می‌کند کدام سطرها باید حذف شوند:

```
DELETE FROM Track WHERE title = 'My Way'
```

این چهار دستور پایه‌ای درSQL— **INSERT** ، **SELECT**، **UPDATE**و**DELETE** چهار عملیاتی را فراهم می‌کنند که برای ایجاد و نگهداری داده لازم‌اند. برای اشاره جمعی به این چهار عملیات، از اصطلاح **CRUD** (Create, Read, Update, Delete)استفاده می‌کنیم.

## ۱۵,۶ چندین جدول و مدل‌سازی داده پایه

قدرت واقعی یک پایگاه‌داده رابطه‌ای زمانی آشکار می‌شود که چندین جدول ایجاد کرده و بین آن‌ها ارتباط برقرار کنیم. عملِ تصمیم‌گیری درباره این‌که چگونه داده‌های یک برنامه را به چند جدول تقسیم کنیم و روابط میان جدول‌ها را مشخص نماییم، **مدل‌سازی داده** نام دارد. سندی که جدول‌ها و روابط میان آن‌ها را نمایش می‌دهد، **مدل داده** (Data Model) نامیده می‌شود.

مدل‌سازی داده مهارتی نسبتاً پیشرفته است و در این بخش تنها به ابتدایی‌ترین مفاهیم مدل‌سازی رابطه‌ای اشاره خواهیم کرد. برای اطلاعات بیشتر درباره مدل‌سازی داده می‌توانید از این منبع شروع کنید:

http://en.wikipedia.org/wiki/Relational_model

فرض کنیم در پایگاه‌داده مربوط به قطعه‌های موسیقی، بخواهیم علاوه بر عنوان هر قطعه و تعداد پخش آن، **نام هنرمند** هر قطعه را نیز ثبت کنیم. یک رویکرد ساده این است که ستونی جدید با نام **artist** به جدول اضافه کنیم و نام هنرمند را در آن ستون قرار دهیم، به‌صورت زیر:

```
DROP TABLE IF EXISTS Track;
CREATE TABLE Track (title TEXT, plays INTEGER, artist TEXT);
```

سپس می‌توانیم چند قطعه موسیقی را در جدول خود وارد کنیم.

```
INSERT INTO Track (title, plays, artist)
    VALUES ('My Way', 15, 'Frank Sinatra');
INSERT INTO Track (title, plays, artist)
    VALUES ('New York', 25, 'Frank Sinatra');
```

اگر با اجرای دستور **SELECT * FROM Track** به داده‌های خود نگاه کنیم، به‌نظر می‌رسد کار را به‌خوبی انجام داده‌ایم.

```
sqlite> SELECT * FROM Track;
My Way|15|Frank Sinatra
New York|25|Frank Sinatra
sqlite>
```

ما در مدل‌سازی داده خود یک خطای بسیار جدی مرتکب شده‌ایم. ما قوانین **نرمال‌سازی پایگاه‌داده** را نقض کرده‌ایم.

https://en.wikipedia.org/wiki/Database_normalization

نرمال‌سازی پایگاه‌داده در ظاهر بسیار پیچیده به‌نظر می‌رسد و شامل توجیه های ریاضی فراوانی است، اما فعلاً می‌توانیم همه آن را به یک قانون ساده که از آن پیروی خواهیم کرد کم کنیم. ما هرگز نباید یک داده متنی یکسان را بیش از یک‌بار در یک ستون قرار دهیم. اگر به آن داده بیش از یک‌بار نیاز داریم، باید یک **کلید عددی** برای آن داده ایجاد کنیم و هر بار با استفاده از این کلید به داده اصلی ارجاع دهیم به‌ویژه در حالتی که چندین ورودی به یک شیء واحد اشاره می‌کنند.

برای نشان دادن این‌که با اختصاص‌دادن ستون‌های متنی به مدل پایگاه‌داده خود، در چه **مسیر لغزنده‌ای** قرار گرفته‌ایم، تصور کنید بخواهیم **رنگ چشم هنرمندان** را نیز در داده‌های خود نگه داریم. آیا چنین کاری خواهیم کرد؟

```
DROP TABLE IF EXISTS Track;
CREATE TABLE Track (title TEXT, plays INTEGER,
    artist TEXT, eyes TEXT);
INSERT INTO Track (title, plays, artist, eyes)
    VALUES ('My Way', 15, 'Frank Sinatra', 'Blue');
INSERT INTO Track (title, plays, artist, eyes)
    VALUES ('New York', 25, 'Frank Sinatra', 'Blue');
```

از آن‌جا که فرانک سیناترا بیش از ۱۲۰۰ ترانه ضبط کرده است، آیا واقعاً قرار است رشته 'Blue' را در ۱۲۰۰ سطر در جدول Track تکرار کنیم؟ و اگر روزی تصمیم بگیریم که رنگ چشم او در واقع 'Light Blue' است چه خواهد شد؟ چیزی در این روش درست به‌نظر نمی‌رسد.

راه‌حل صحیح این است که جدولی جداگانه برای **هر هنرمند (Artist)** ایجاد کنیم و تمام داده‌های مربوط به هنرمند را در آن جدول ذخیره کنیم. سپس به روشی باید بین یک سطر در جدول **Track** و یک سطر در جدول **Artist** ارتباط برقرار کنیم. شاید بتوان این «پیوند» میان دو «جدول» را یک «رابطه» بین دو جدول نامید. و دقیقاً همین نامی است که کارشناسان پایگاه‌داده برای این پیوندها انتخاب کرده‌اند.

حالا بیایید یک جدول Artist به شکل زیر بسازیم:

```
DROP TABLE IF EXISTS Artist;
CREATE TABLE Artist (name TEXT, eyes TEXT);
INSERT INTO Artist (name, eyes)
   VALUES ('Frank Sinatra', 'blue');
```

اکنون دو جدول در اختیار داریم، اما برای مرتبط‌کردن سطرهای این دو جدول به یکدیگر، نیاز به روشی برای ایجاد پیوند داریم. برای این کار، به چیزی نیاز داریم که آن را «**کلید**» می‌نامیم. این کلیدها صرفاً عددهای صحیحی هستند که می‌توانیم از آن‌ها برای یافتن یک سطر در جدول دیگر استفاده کنیم.

اگر قرار است به سطرهای یک جدول ارجاع دهیم، باید **کلید اصلی (Primary Key)** به سطرهای آن جدول اضافه کنیم. طبق یک قرارداد رایج، معمولاً نام ستون کلید اصلی را **id** قرار می‌دهیم. بنابراین جدول **Artist** ما به این شکل درخواهد آمد:

```
DROP TABLE IF EXISTS Artist;
CREATE TABLE Artist (id INTEGER, name TEXT, eyes TEXT);
INSERT INTO Artist (id, name, eyes)
   VALUES (42, 'Frank Sinatra', 'blue');
```

اکنون یک سطر در جدول برای **Frank Sinatra** (به‌همراه رنگ چشم او) داریم و یک **کلید اصلی** با مقدار **۴۲** که می‌توانیم از آن برای مرتبط‌کردن قطعه‌های موسیقی به او استفاده کنیم. بنابراین جدول **Track** را به شکل زیر تغییر می‌دهیم:

```
DROP TABLE IF EXISTS Track;
CREATE TABLE Track (title TEXT, plays INTEGER,
    artist_id INTEGER);
INSERT INTO Track (title, plays, artist_id)
    VALUES ('My Way', 15, 42);
INSERT INTO Track (title, plays, artist_id)
    VALUES ('New York', 25, 42);
```

ستون **artist_id** یک عدد صحیح است و طبق نام‌گذاری متداول، یک **کلید خارجی (Foreign Key)** محسوب می‌شود که به **کلید اصلی (Primary Key)** در جدول **Artist** اشاره می‌کند. این ستون را «کلید خارجی» می‌نامیم زیرا به سطری در **جدولی دیگر** اشاره دارد.

اکنون ما از قوانین **نرمال‌سازی پایگاه‌داده** پیروی می‌کنیم، اما هنگام استخراج داده از پایگاه‌داده، نمی‌خواهیم مقدار **۴۲** را ببینیم؛ ما می‌خواهیم **نام و رنگ چشم هنرمند** را مشاهده کنیم. برای انجام این کار، از کلیدواژه **JOIN** در عبارت SELECT استفاده می‌کنیم.

```
SELECT title, plays, name, eyes
FROM Track JOIN Artist
ON Track.artist_id = Artist.id;
```

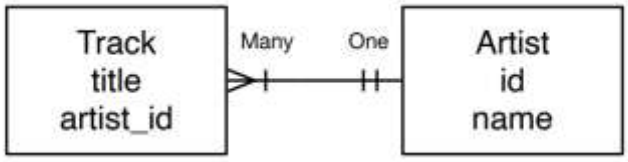
عبارت **JOIN** دارای یک شرط **ON** است که مشخص می‌کند سطرهای دو جدول چگونه باید به یکدیگر متصل شوند. برای هر سطر در جدول **Track**، داده‌های مربوط به جدول **Artist** از آن سطری اضافه می‌شود که مقدار **artist_id** در جدول Trackبا مقدار **id** در جدول Artist مطابقت داشته باشد. خروجی به این صورت خواهد بود:

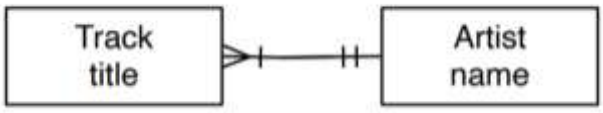
```
My Way|15|Frank Sinatra|blue
New York|25|Frank Sinatra|blue
```

اگرچه این روش در ابتدا کمی دست‌وپاگیر به‌نظر می‌رسد و شاید غر instinct شما این باشد که نگه‌داشتن تمام داده‌ها در یک جدول سریع‌تر خواهد بود، اما در واقع **بزرگ‌ترین محدودیت کارایی پایگاه‌داده، میزان داده‌ای است که هنگام اجرای یک پرس‌وجو باید اسکن شود.** هرچند جزئیات بسیار پیچیده‌اند، اما **اعداد صحیح (integers)** بسیار کوچک‌تر از رشته‌ها—به‌ویژه رشته‌های یونیکد—هستند و **جابجایی و مقایسه آن‌ها بسیار سریع‌تر** انجام می‌شود.

## ۱۵,۷ نمودارهای مدل داده

در حالی که طراحی پایگاه‌داده ما برای **Track**و **Artist** ساده است—با تنها دو جدول و یک رابطه یک‌به‌چند—مدل‌های داده می‌توانند به‌سرعت پیچیده شوند. در چنین شرایطی، درک این مدل‌ها زمانی بسیار آسان‌تر می‌شود که یک **نمایش گرافیکی** از مدل داده خود داشته باشیم.






اگرچه روش‌های گرافیکی زیادی برای نمایش مدل‌های داده وجود دارند، در این‌جا از یکی از روش‌های «کلاسیک» استفاده می‌کنیم که **نمودار پای کلاغی** (*Crow's Foot Diagrams*) نام دارد، همان‌گونه که در شکل نشان داده شده است.

در این روش، هر جدول در قالب یک مستطیل نمایش داده می‌شود که شامل **نام جدول** و **ستون‌های آن** است. سپس هرجا که بین دو جدول رابطه‌ای وجود داشته باشد، **یک خط** آن‌ها را به یکدیگر متصل می‌کند و در انتهای این خط از **نمادهای مخصوص** استفاده می‌شود تا **ماهیت رابطه** مشخص شود.

https://en.wikipedia.org/wiki/Entity-relationship_model

در این مثال، «تعداد زیادی» قطعه موسیقی می‌تواند به هر هنرمند مرتبط باشد. بنابراین در نمودار، سمت مربوط به **Track** با نماد پای کلاغی باز شده نمایش داده می‌شود که نشان‌دهنده انتهای «چند» است. سمت مربوط به **Artist** با یک خط عمودی نشان داده می‌شود که به معنای «یک» است.

در عمل ممکن است هنرمندان بسیاری وجود داشته باشند، اما نکته مهم این است که **برای هر هنرمند، تعداد زیادی قطعه موسیقی وجود خواهد داشت** و هر هنرمند ممکن است با چندین قطعه مرتبط باشد. نکته دیگری که مشاهده می‌شود این است که **ستونی که کلید خارجی** مانند `artist_id` را نگه می‌دارد **در سمت «چند»** قرار دارد، و **کلید اصلی** در سمت «یک».

از آن‌جا که الگوی جای‌گذاری کلیدهای اصلی و خارجی بسیار ثابت است و دقیقاً مطابق سرِ «چند» و سرِ «یک» نمودار قرار می‌گیرد، معمولاً در نمودار مدل داده—همان‌گونه که در نمودار دومِ شکل نشان داده شده—**هیچ‌کدام از ستون‌های کلید اصلی یا کلید خارجی را درج نمی‌کنیم.**

این ستون‌ها به‌عنوان **جزئیات اجرایی** (Implementation Detail) تلقی می‌شوند که ماهیت رابطه را ثبت می‌کنند، نه به‌عنوان بخش اصلی داده‌ای که مدل می‌شود.

## ۱۵,۸ ایجاد خودکار کلیدهای اصلی

در مثال پیشین، ما به‌طور دلخواه مقدار **۴۲** را به‌عنوان کلید اصلیِ فرانک اختصاص دادیم. اما هنگامی که میلیون‌ها سطر وارد پایگاه‌داده می‌کنیم، بهتر است پایگاه‌داده **به‌طور خودکار** مقدار ستون **id** را تولید کند. برای این کار، ستون **id** را به‌صورت **PRIMARY KEY** تعریف می‌کنیم و هنگام درج سطر جدید، مقدار id را مشخص نمی‌کنیم، به‌صورت زیر:

```
DROP TABLE IF EXISTS Artist;
CREATE TABLE Artist (id INTEGER PRIMARY KEY,
```

```
    name TEXT, eyes TEXT);
INSERT INTO Artist (name, eyes)
   VALUES ('Frank Sinatra', 'blue');
```

اکنون به پایگاه‌داده دستور داده‌ایم که برای سطر مربوط به **Frank Sinatra** یک مقدار یکتا را به‌صورت خودکار اختصاص دهد. اما پس از درج سطر، نیاز داریم **راهی برای به‌دست آوردن مقدار id سطر تازه وارد شده** داشته باشیم.

یکی از روش‌ها این است که با استفاده از یک دستور **SELECT**، مقدار را از تابع داخلی SQLite با نام **last_insert_rowid()** بازیابی کنیم.

```
sqlite> DROP TABLE IF EXISTS Artist;
sqlite> CREATE TABLE Artist (id INTEGER PRIMARY KEY,
   ...>     name TEXT, eyes TEXT);
sqlite> INSERT INTO Artist (name, eyes)
   ...>    VALUES ('Frank Sinatra', 'blue');
sqlite> select last_insert_rowid();
1
sqlite> SELECT * FROM Artist;
1|Frank Sinatra|blue
sqlite>
```

به‌محض آن‌که مقدار **id** مربوط به سطر «Frank Sinatra» را بدانیم، می‌توانیم هنگام **INSERT** کردنِ قطعه‌ها در جدول **Track** از آن استفاده کنیم. به‌عنوان یک **راهکار کلی**، هرگاه جدولی ایجاد می‌کنیم، یک ستون **id** نیز به آن اضافه می‌کنیم:

```
sqlite> DROP TABLE IF EXISTS Track;
sqlite> CREATE TABLE Track (id INTEGER PRIMARY KEY,
   ...>     title TEXT, plays INTEGER, artist_id INTEGER);
```

توجه داشته باشید که مقدار **artist_id** همان سطر جدیدی است که به‌صورت خودکار در جدول **Artist** اختصاص داده شده است. همچنین با وجود این‌که ستون **INTEGER PRIMARY KEY** را به جدول **Track** اضافه کرده‌ایم، در دستورات **INSERT** مربوط به جدولTrack ، ستون **id** را در فهرست فیلدها قرار ندادیم.

این کار دوباره به پایگاه‌داده اعلام می‌کند که **برای ستون id یک مقدار یکتای خودکار** انتخاب کند.

```
sqlite> INSERT INTO Track (title, plays, artist_id)
   ...>     VALUES ('My Way', 15, 1);
sqlite> select last_insert_rowid();
1
sqlite> INSERT INTO Track (title, plays, artist_id)
   ...>     VALUES ('New York', 25, 1);
sqlite> select last_insert_rowid();
2
sqlite>
```

می‌توان پس از هر بارINSERT ، دستور **SELECT last_insert_rowid();** را فراخوانی کرد تا مقدار id که پایگاه‌داده برای سطر تازه‌افزوده‌شده تعیین کرده است، بازیابی شود. بعدتر، هنگامی که با پایتون کدنویسی می‌کنیم، می‌توانیم مقدار id را در کد درخواست کنیم و آن را برای استفاده‌های بعدی در یک متغیر ذخیره کنیم.

## ۱۵,۹ کلیدهای منطقی برای جست‌وجوی سریع

اگر جدولی پر از هنرمندان و جدولی پر از قطعه‌های موسیقی داشته باشیم—که هر سطر از جدول Track با استفاده از یک کلید خارجی به سطری در جدول Artist مرتبط شده باشد—و بخواهیم **تمام قطعه‌هایی را که «Frank Sinatra» اجرا کرده است** فهرست کنیم، به شکل زیر عمل خواهیم کرد:

```
SELECT title, plays, name, eyes
FROM Track JOIN Artist
ON Track.artist_id = Artist.id
WHERE Artist.name = 'Frank Sinatra';
```

از آن‌جا که دو جدول داریم و یک کلید خارجی این دو جدول را به هم مرتبط می‌کند، داده ما **خوب مدل‌سازی شده** است. اما اگر قرار باشد میلیون‌ها رکورد در جدول **Artist** داشته باشیم و بخواهیم جست‌وجوهای زیادی بر اساس **نام هنرمند** انجام دهیم، بهتر است به پایگاه‌داده **سرنخی درباره نحوه استفاده خود از ستون name** بدهیم. این کار را با افزودن یک **ایندکس (index)** به ستونی متنی که قصد داریم در عبارت‌های **WHERE** از آن استفاده کنیم انجام می‌دهیم:

```
CREATE INDEX artist_name ON Artist(name);
```

وقتی به پایگاه‌داده اعلام می‌کنیم که یک **ایندکس** روی ستونی از یک جدول لازم است، پایگاه‌داده اطلاعات اضافی ذخیره می‌کند تا بتواند با استفاده از فیلد ایندکس‌شده (در این مثال، **name**) سطرها را سریع‌تر پیدا کند.

پس از درخواست ایجاد ایندکس، نیازی نیست در دستورات SQL کار خاصی انجام دهیم. پایگاه‌داده **به‌صورت خودکار** ایندکس را هنگام **درج، حذف یا به‌روزرسانی داده‌ها** به‌روز نگه می‌دارد و هرگاه تشخیص دهد که استفاده از ایندکس سرعت اجرای پرس‌وجو را افزایش می‌دهد، آن را به‌طور **خودکار** به کار می‌گیرد. ستون‌های متنی که برای یافتن سطرها بر اساس اطلاعات دنیای واقعی—برای مثال نام یک هنرمند— استفاده می‌شوند، **کلیدهای منطقی (Logical Keys)** نامیده می‌شوند.

## ۱۵,۱۰ افزودن قیود (Constraints) به پایگاه‌داده

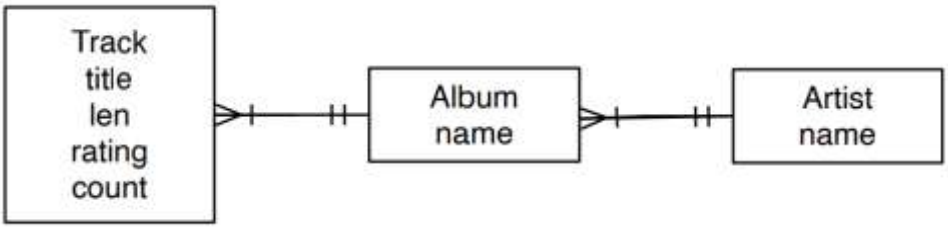
ما همچنین می‌توانیم از ایندکس برای اعمال **قیدها** (rules) در عملیات پایگاه‌داده استفاده کنیم. رایج‌ترین قید، **قید یکتایی (uniqueness constraint)** است که تضمین می‌کند تمام مقادیر موجود در یک ستون **منحصربه‌فرد** باشند. می‌توانیم با افزودن کلیدواژه **UNIQUE** (که اختیاری است) به دستور **CREATE INDEX** به پایگاه‌داده اعلام کنیم که این قید را در

اجرای دستورات SQL رعایت کند. برای اعمال این قید، می‌توانیم ایندکس **artist_name** را حذف کرده و آن را با قید **UNIQUE** دوباره ایجاد کنیم، به‌صورت زیر:

```
DROP INDEX artist_name;
CREATE UNIQUE INDEX artist_name ON Artist(name);
```

اگر تلاش کنیم نام **'Frank Sinatra'** را برای بار دوم وارد کنیم، عملیات با **خطا** مواجه خواهد شد.

```
sqlite> SELECT * FROM Artist;
1|Frank Sinatra|blue
sqlite> INSERT INTO Artist (name, eyes)
   ...>    VALUES ('Frank Sinatra', 'blue');
Runtime error: UNIQUE constraint failed: Artist.name (19)
sqlite>
```




می‌توانیم به پایگاه‌داده اعلام کنیم که خطاهای مربوط به **کلید تکراری** را نادیده بگیرد. برای این کار، کلیدواژه **IGNORE** را به دستور **INSERT** اضافه می‌کنیم، به شکل زیر:

```
sqlite> INSERT OR IGNORE INTO Artist (name, eyes)
   ...>     VALUES ('Frank Sinatra', 'blue');
sqlite> SELECT id FROM Artist WHERE name='Frank Sinatra';
1
sqlite>
```

با ترکیب دو دستور **INSERT OR IGNORE** و **SELECT** می‌توانیم— اگر نام از قبل در پایگاه‌داده وجود نداشته باشد— رکورد جدیدی وارد کنیم و چه رکورد از پیش وجود داشته باشد چه نه، **کلید اصلی (primary key)** آن رکورد را بازیابی کنیم.

```
sqlite> INSERT OR IGNORE INTO Artist (name, eyes)
   ...>      VALUES ('Elvis', 'blue');
sqlite> SELECT id FROM Artist WHERE name='Elvis';
2
sqlite> SELECT * FROM Artist;
1|Frank Sinatra|blue
2|Elvis|blue
sqlite>
```

از آن‌جا که برای ستون مربوط به **رنگ چشم** هیچ قید یکتایی (Uniqueness Constraint) تعریف نکرده‌ایم، وجود چندین مقدار **«Blue»**در این ستون هیچ مشکلی ایجاد نمی‌کند.

## ۱۵,۱۱ یک نمونه برنامه چند جدولی

یک برنامه نمونه با نام **tracks_csv.py** نشان می‌دهد چگونه می‌توان این ایده‌ها را با یکدیگر ترکیب کرد تا داده‌های متنی تحلیل (parse) شوند و سپس با استفاده از یک **مدل داده صحیح** شامل **روابط بین جدول‌ها**، در چندین جدول بارگذاری شوند.

این برنامه، فایل جداشده با ویرگول (**tracks.csv**) را—که بر اساس خروجی کتابخانه iTunes دکتر چاک تهیه شده است—می‌خواند و پردازش می‌کند.

```
Another One Bites The Dust,Queen,Greatest Hits,55,100,217103
Asche Zu Asche,Rammstein,Herzeleid,79,100,231810
Beauty School Dropout,Various,Grease,48,100,239960
Black Dog,Led Zeppelin,IV,109,100,296620
...
```

ستون‌های موجود در این فایل عبارت‌اند از **title** عنوان، **artist** هنرمند، **album** آلبوم، **number of plays** تعداد پخش، **rating** امتیاز بین ۰ تا ۱۰۰، و **length** مدت زمان قطعه بر حسب میلی‌ثانیه. مدل داده ما در شکل نشان داده شده است و در قالب دستورات SQL به شکل زیر تعریف می‌شود:

```
DROP TABLE IF EXISTS Artist;
DROP TABLE IF EXISTS Album;
DROP TABLE IF EXISTS Track;

CREATE TABLE Artist (
    id INTEGER PRIMARY KEY,
    name TEXT UNIQUE
);

CREATE TABLE Album (
    id INTEGER PRIMARY KEY,
    artist_id  INTEGER,
    title TEXT UNIQUE
);

CREATE TABLE Track (
    id INTEGER PRIMARY KEY,
    title TEXT UNIQUE,
    album_id INTEGER,
    len INTEGER, rating INTEGER, count INTEGER
);
```

ما در حال افزودن کلیدواژه **UNIQUE** به ستون‌های متنی (TEXT) هستیم که می‌خواهیم روی آن‌ها قید یکتایی برقرار باشد، تا بتوانیم از این قید در دستورات **INSERT IGNORE** استفاده کنیم. این روش، نسبت به نوشتن دستورات جداگانه **CREATE INDEX**، کوتاه‌تر و خلاصه‌تر است، اما **همان تأثیر** را دارد.

با آماده بودن این جدول‌ها، کد **tracks_csv.py** را به‌صورت زیر می‌نویسیم تا داده‌ها را تجزیه (parse) کرده و در جدول‌ها وارد کند:

```
import sqlite3

conn = sqlite3.connect('trackdb.sqlite')
cur = conn.cursor()

handle = open('tracks.csv')

for line in handle:
    line = line.strip();
    pieces = line.split(',')
    if len(pieces) != 6 : continue

    name = pieces[0]
    artist = pieces[1]
    album = pieces[2]
    count = pieces[3]
    rating = pieces[4]
    length = pieces[5]

    print(name, artist, album, count, rating, length)

    cur.execute('''INSERT OR IGNORE INTO Artist (name)
        VALUES ( ? )''', ( artist, ) )
    cur.execute('SELECT id FROM Artist WHERE name = ? ', (artist, ))
    artist_id = cur.fetchone()[0]

    cur.execute('''INSERT OR IGNORE INTO Album (title, artist_id)
        VALUES ( ?, ? )''', ( album, artist_id ) )
    cur.execute('SELECT id FROM Album WHERE title = ? ', (album, ))
    album_id = cur.fetchone()[0]

    cur.execute('''INSERT OR REPLACE INTO Track
        (title, album_id, len, rating, count)
        VALUES ( ?, ?, ?, ?, ? )''',
        ( name, album_id, length, rating, count ) )

    conn.commit()
```

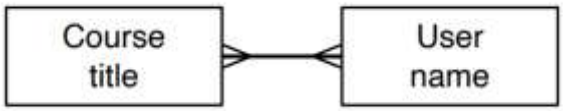
می‌بینید که ما الگوی **INSERT OR IGNORE** به‌همراه یک دستور **SELECT** را بارها تکرار می‌کنیم تا مقدارهای مناسب **artist_id** و **album_id** را به‌دست آوریم و در دستورات INSERT بعدی از آن‌ها استفاده کنیم. از جدول **Artist** شروع می‌کنیم، زیرا برای درج در جدول **Album** به مقدار **artist_id** نیاز داریم و برای درج در جدول **Track** نیز به مقدار **album_id** نیازمندیم.

اگر به جدول **Album** نگاه کنیم، می‌بینیم که رکوردها در هنگام پردازش داده‌ها اضافه شده‌اند و در صورت نیاز، یک **کلید اصلی (primary key)**به آن‌ها اختصاص داده شده است. همچنین می‌توانیم برای هر سطر در جدولAlbum ، **کلید خارجی (foreign key)**را ببینیم که به سطری در جدول **Artist** اشاره می‌کند.

```
sqlite> .mode column
sqlite> SELECT * FROM Album LIMIT 5;
id  artist_id  title
--  ---------  ----------------
1   1          Greatest Hits
2   2          Herzeleid
3   3          Grease
4   4          IV
5   5          The Wall [Disc 2]
```

ما می‌توانیم تمام داده‌های جدول **Track** را با دنبال‌کردن روابط میان جدول‌ها، و استفاده از عبارت‌های **JOIN** و **ON** دوباره بازسازی کنیم. در خروجی زیر، می‌توانید **هر دو انتهای دو رابطه موجود** را که هر ردیف را به جدول‌های دیگر متصل می‌کنند مشاهده کنید.

```
sqlite> .mode line
sqlite> SELECT * FROM Track
   ...> JOIN Album ON Track.album_id = Album.id
   ...> JOIN Artist ON Album.artist_id = Artist.id
   ...> LIMIT 2;
       id = 1
    title = Another One Bites The Dust
 album_id = 1
      len = 217103
   rating = 100
    count = 55
       id = 1
artist_id = 1
    title = Greatest Hits
       id = 1
     name = Queen
```




```
       id = 2
    title = Asche Zu Asche
 album_id = 2
      len = 231810
   rating = 100
    count = 79
       id = 2
artist_id = 2
    title = Herzeleid
       id = 2
     name = Rammstein
```

این مثال سه جدول و دو رابطه یک‌به‌چند میان این جدول‌ها را نشان می‌دهد. همچنین توضیح می‌دهد که چگونه می‌توان با استفاده از **ایندکس‌ها** و **محدودیت‌های یکتایی(uniqueness constraints)** ، جدول‌ها و روابط میان آن‌ها را به‌صورت برنامه‌نویسی‌شده ایجاد کرد.

https://en.wikipedia.org/wiki/One-to-many_(data_model)

در ادامه نگاهی به **رابطه‌های چندبه‌چند** در مدل‌های داده خواهیم داشت.

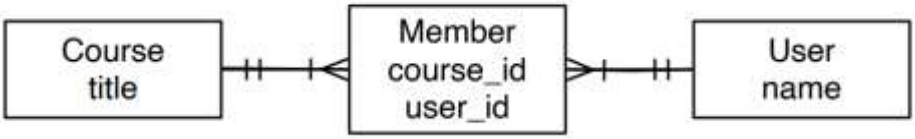
## ۱۵,۱۲ رابطه‌های چندبه‌چند در پایگاه‌داده‌ها

برخی از رابطه‌های داده‌ای را نمی‌توان با یک رابطه ساده یک‌به‌چند مدل‌سازی کرد .برای مثال فرض کنید قصد داریم برای یک سامانه مدیریت درس، یک مدل داده طراحی کنیم. در این سامانه **درس‌ها(courses)** ، **کاربران(users)** ، و **فهرست حضور (rosters)** وجود خواهند داشت .یک کاربر می‌تواند در فهرست چندین درس قرار داشته باشد و یک درس نیز می‌تواند در فهرست خود چندین کاربر داشته باشد.

رسم یک رابطه چندبه‌چند بسیار ساده است—همان‌طور که در شکل نشان داده شده است. تنها کافی است دو جدول را رسم کنیم و آن‌ها را با خطی که در هر دو انتها علامت «چند» دارد به یکدیگر وصل کنیم .مشکل اصلی اینجاست که **چگونه این رابطه را با استفاده از کلیدهای اصلی و خارجی پیاده‌سازی کنیم.**

پیش از آن‌که بررسی کنیم چگونه رابطه‌های چندبه‌چند را در عمل پیاده‌سازی کنیم، ببینیم آیا می‌توانیم با گسترش یک رابطه یک‌به‌چند، راه‌حلی موقتی (hack) بسازیم .اگر SQL از ساختار **آرایه‌ها** پشتیبانی می‌کرد، شاید تلاش می‌کردیم رابطه را این‌گونه تعریف کنیم:

```
CREATE TABLE Course (
    id      INTEGER PRIMARY KEY,
    title   TEXT UNIQUE
    student_ids ARRAY OF INTEGER;
);
```




متأسفانه، با وجود آن‌که این ایده وسوسه‌انگیز است، **SQL از آرایه‌ها پشتیبانی نمی‌کند**. یا می‌توانیم یک رشته طولانی بسازیم و تمام کلیدهای اصلی کاربران را با جداکننده ویرگول در آن به هم متصل کنیم.

```
CREATE TABLE Course (
    id      INTEGER PRIMARY KEY,
    title   TEXT UNIQUE
    student_ids ARRAY OF INTEGER;
);

INSERT INTO Course (title, student_ids)
VALUES( 'si311', '1,3,4,5,6,9,14');
```

این کار **بسیار ناکارآمد** خواهد بود، زیرا با افزایش اندازه فهرستِ یک درس (roster) و افزایش تعداد درس‌ها، تعیین این‌که کدام درس‌ها دانشجوی شماره ۱۴ را در فهرست خود دارند، بسیار پرهزینه و دشوار می‌شود.

به‌جای استفاده از هیچ‌یک از روش‌های قبلی، رابطه **چندبه‌چند** را با یک جدول اضافی مدل‌سازی می‌کنیم که آن را *جدول واسط (junction table)*، *through table*، *connector table* یا *join table* می‌نامیم، همان‌طور که در شکل نشان داده شده است. هدف این جدول، ثبت ارتباط بین یک **درس** و یک **دانشجو** است. به‌یک معنا این جدول **میانِ جدول Course و جدول User قرار می‌گیرد** و با هر یک از آن‌ها یک رابطه **یک‌به‌چند** دارد.

با استفاده از یک جدول واسط، یک رابطه چندبه‌چند را به **دو رابطه یک‌به‌چند** تبدیل می‌کنیم. پایگاه‌داده‌ها در مدل‌سازی و پردازش روابط یک‌به‌چند بسیار توانمند هستند. یک نمونه جدول **Member** به شکل زیر خواهد بود:

```
CREATE TABLE User (
    id      INTEGER PRIMARY KEY,
    name    TEXT UNIQUE
);

CREATE TABLE Course (
    id      INTEGER PRIMARY KEY,
    title   TEXT UNIQUE
);
```

```
CREATE TABLE Member (
    user_id     INTEGER,
    course_id   INTEGER,
    PRIMARY KEY (user_id, course_id)
);
```

با پیروی از قرارداد نام‌گذاری خود، ستون‌های **Member.user_id**و **Member.course_id** کلیدهای خارجی هستند که به سطرهای متناظر در جدول‌های **User**و **Course** اشاره می‌کنند. هر ورودی در جدول Member یک سطر در جدول User را از طریق جدول Member به یک سطر در جدول Course متصل می‌کند.

ما مشخص می‌کنیم که ترکیب **course_id**و **user_id**، کلید اصلی (**PRIMARY KEY**) جدول Member باشد؛ این کار یک قید یکتایی نیز برای ترکیب user_id / course_id ایجاد می‌کند. حالا فرض کنید لازم است تعدادی دانشجو را در فهرست (roster)چندین درس وارد کنیم. فرض می‌کنیم این داده‌ها در قالب یک فایل JSON به‌صورت رکوردهایی مانند زیر به دست ما می‌رسند:

```
[
  [ "Charley", "si110"],
  [ "Mea", "si110"],
  [ "Hattie", "si110"],
  [ "Keziah", "si110"],
  [ "Rosa", "si106"],
  [ "Mea", "si106"],
  [ "Mairin", "si106"],
  [ "Zendel", "si106"],
  [ "Honie", "si106"],
  [ "Rosa", "si106"],
...
]
```

می‌توانیم کدی به شکل زیر بنویسیم تا فایل JSON را بخواند و اعضای هر فهرست درس را با استفاده از کد زیر در پایگاه‌داده درج کند.

```
import json
import sqlite3

conn = sqlite3.connect('rosterdb.sqlite')
cur = conn.cursor()

str_data = open('roster_data_sample.json').read()
json_data = json.loads(str_data)

for entry in json_data:

    name = entry[0]
```

```
    title = entry[1]

    print((name, title))

    cur.execute('''INSERT OR IGNORE INTO User (name)
        VALUES ( ? )''', ( name, ) )
    cur.execute('SELECT id FROM User WHERE name = ? ', (name, ))
    user_id = cur.fetchone()[0]

    cur.execute('''INSERT OR IGNORE INTO Course (title)
        VALUES ( ? )''', ( title, ) )
    cur.execute('SELECT id FROM Course WHERE title = ? ', (title, ))
    course_id = cur.fetchone()[0]

    cur.execute('''INSERT OR REPLACE INTO Member
        (user_id, course_id) VALUES ( ?, ? )''',
        ( user_id, course_id ) )

    conn.commit()
```

همان‌طور که در یک مثال قبلی دیدیم، ابتدا مطمئن می‌شویم که برای کاربر مربوطه یک رکورد در جدول User داریم و کلید اصلی آن رکورد را می‌دانیم، و همین‌طور برای درس مربوطه یک رکورد در جدول Course داریم و از کلید اصلی آن آگاه هستیم. از الگوی «INSERT OR IGNORE»و «SELECT» استفاده می‌کنیم تا کد ما، چه رکورد از قبل در جدول وجود داشته باشد و چه وجود نداشته باشد، به‌درستی کار کند.

درج ما در جدول Member در ساده‌ترین حالت تنها وارد کردن دو عدد صحیح است؛ به‌صورت یک سطر جدید یا استفاده از یک سطر موجود، بسته به قیدی که تعریف کرده‌ایم تا مطمئن شویم برای یک ترکیب مشخص از user_id و course_id ورودی‌های تکراری در جدول Member نخواهیم داشت. برای بازسازی داده‌هایمان در هر سه جدول، دوباره از JOIN / ON استفاده می‌کنیم تا یک پرس‌وجوی SELECT بسازیم؛

```
sqlite> SELECT * FROM Course
   ...> JOIN Member ON Course.id = Member.course_id
   ...> JOIN User ON Member.user_id = User.id;
+----+-------+---------+-----------+----+---------+
| id | title | user_id | course_id | id |  name   |
+----+-------+---------+-----------+----+---------+
| 1  | si110 | 1       | 1         | 1  | Charley |
| 1  | si110 | 2       | 1         | 2  | Mea     |
| 1  | si110 | 3       | 1         | 3  | Hattie  |
| 1  | si110 | 4       | 1         | 4  | Lyena   |
| 1  | si110 | 5       | 1         | 5  | Keziah  |
| 1  | si110 | 6       | 1         | 6  | Ellyce  |
| 1  | si110 | 7       | 1         | 7  | Thalia  |
| 1  | si110 | 8       | 1         | 8  | Meabh   |
| 2  | si106 | 2       | 2         | 2  | Mea     |
```

```
| 2  | si106 | 10      | 2         | 10 | Mairin  |
| 2  | si106 | 11      | 2         | 11 | Zendel  |
| 2  | si106 | 12      | 2         | 12 | Honie   |
| 2  | si106 | 9       | 2         | 9  | Rosa    |
+----+-------+---------+-----------+----+---------+
sqlite>
```

می‌توانیم سه جدول را از چپ به راست ببینیمCourse :. Memberو User و همچنین می‌توانیم ارتباط‌ها بین کلیدهای اصلی و کلیدهای خارجی را در هر سطرِ خروجی مشاهده کنیم.

## ۱۵,۱۳ مدل‌سازی داده در نقطه اتصالِ چندبه‌چند

در حالی که تاکنون «جدول اتصال» (join table) را به‌صورت جدولی با دو کلید خارجی که بین سطرهای دو جدول ارتباط برقرار می‌کنند معرفی کرده‌ایم، این ساده‌ترین شکل یک جدول اتصال است. در عمل، بسیار رایج است که بخواهیم خودِ این ارتباط را نیز با داده‌هایی توصیف کنیم. با ادامه‌دادن مثال کاربران، دوره‌ها و فهرست‌های ثبت‌نام برای مدل‌سازی یک سامانه ساده مدیریت یادگیری، لازم است **نقشی** را که هر کاربر در هر دوره بر عهده دارد نیز درک و ثبت کنیم.

اگر ابتدا تلاش کنیم این مسئله را با افزودن یک فلگ (پرچم) به نام «instructor» به جدول User حل کنیم، متوجه می‌شویم که این کار جواب نمی‌دهد؛ زیرا ممکن است یک کاربر در یک دوره **مدرس** و در دوره‌ای دیگر **دانشجو** باشد. اگر ستونی به نام instructor_idبه جدول Course اضافه کنیم باز هم مشکل حل نمی‌شود، چون یک دوره می‌تواند چندین مدرس داشته باشد. و هیچ حقه یک‌به‌چندی وجود ندارد که بتواند با این واقعیت کنار بیاید که تعداد نقش‌ها به‌مرور گسترش می‌یابد و نقش‌هایی مانند Teaching Assistantیا Parent نیز اضافه می‌شوند.

اما اگر به‌سادگی ستونی به نام role به جدول Member اضافه کنیم، می‌توانیم دامنه وسیعی از نقش‌ها، ترکیب نقش‌ها و غیره را نمایش دهیم.بیایید جدول Member خود را به شکل زیر تغییر دهیم:

```
DROP TABLE Member;

CREATE TABLE Member (
    user_id     INTEGER,
    course_id   INTEGER,
    role        INTEGER,
    PRIMARY KEY (user_id, course_id)
);
```

برای سادگی، تصمیم می‌گیریم که عدد **صفر** در ستون **role** به معنای «دانشجو» باشد و عدد **یک** در ستون **role** به معنای «مدرس».(instructor) دفرض می‌کنیم داده JSON ما به شکل زیر با فیلد **role** تکمیل شده است.

```
[
  [ "Charley", "si110", 1],
  [ "Mea", "si110", 0],
```

```
  [ "Hattie", "si110", 0],
  [ "Keziah", "si110", 0],
  [ "Rosa", "si106", 0],
  [ "Mea", "si106", 1],
  [ "Mairin", "si106", 0],
  [ "Zendel", "si106", 0],
  [ "Honie", "si106", 0],
  [ "Rosa", "si106", 0],
...
]
```

می‌توانیم برنامه roster.py بالا را تغییر دهیم تا نقش (role) را نیز در آن بگنجانیم.

```
for entry in json_data:

    name = entry[0]
    title = entry[1]
    role = entry[2]

    ...

    cur.execute('''INSERT OR REPLACE INTO Member
        (user_id, course_id, role) VALUES ( ?, ?, ? )''',
        ( user_id, course_id, role ) )
```

در یک سامانه واقعی، احتمالاً یک جدول Role ایجاد می‌کردیم و ستون role در جدول Member را به‌صورت یک کلید خارجی به جدول Role در می‌آوردیم، به‌صورت زیر:

```
DROP TABLE Member;

CREATE TABLE Member (
    user_id     INTEGER,
    course_id   INTEGER,
    role_id     INTEGER,
    PRIMARY KEY (user_id, course_id, role_id)
);

CREATE TABLE Role (
    id          INTEGER PRIMARY KEY,
    name        TEXT UNIQUE
);

INSERT INTO Role (id, name) VALUES (0, 'Student');
INSERT INTO Role (id, name) VALUES (1, 'Instructor');
```

توجه کنید که از آنجا که ستون id را در جدول Role به‌عنوان PRIMARY KEY اعلام کرده‌ایم، می‌توانیم آن را در دستور INSERTحذف کنیم. اما همچنین می‌توانیم مقدار id را خودمان انتخاب کنیم، به شرطی که این مقدار از قبل در ستون id وجود نداشته باشد و مغایر با محدودیت ضمنی UNIQUE برای کلیدهای اصلی نباشد.

## ۱۵,۱۴ خلاصه

این فصل مطالب گسترده‌ای را پوشش داد تا یک دید کلی از اصول اولیه استفاده از پایگاه‌داده در پایتون به شما بدهد. نوشتن کدی که از یک پایگاه‌داده برای ذخیره‌سازی داده‌ها استفاده کند، پیچیده‌تر از استفاده از دیکشنری‌ها یا فایل‌های ساده‌ی متنی در پایتون است. بنابراین، تنها زمانی باید از پایگاه‌داده استفاده کرد که برنامه شما واقعاً به قابلیت‌های پایگاه‌داده نیاز داشته باشد. موقعیت‌هایی که یک پایگاه‌داده می‌تواند بسیار مفید باشد عبارتند از:

۱) زمانی که برنامه شما نیاز دارد تعداد زیادی به‌روزرسانی کوچک و تصادفی روی یک مجموعه داده بزرگ انجام دهد،
۲) زمانی که داده‌های شما آن‌قدر بزرگ هستند که در یک دیکشنری جا نمی‌شوند و باید به‌طور مکرر اطلاعات را جست‌وجو کنید،
۳) زمانی که یک فرآیند طولانی‌مدت دارید که می‌خواهید بتوانید آن را متوقف و دوباره اجرا کنید و داده‌ها از یک اجرا تا اجرای بعدی حفظ شوند.

می‌توانید یک پایگاه‌داده ساده با یک جدول برای بسیاری از نیازهای برنامه بسازید، اما بیشتر مسائل به چندین جدول و پیوندها/روابط میان ردیف‌های جداول مختلف نیاز دارند. وقتی شروع به ساختن پیوند میان جداول می‌کنید، مهم است که طراحی دقیقی انجام دهید و قوانین نرمال‌سازی پایگاه‌داده را رعایت کنید تا بهترین استفاده از قابلیت‌های پایگاه‌داده را داشته باشید. از آنجا که انگیزه اصلی استفاده از پایگاه‌داده معمولاً وجود حجم زیادی از داده‌هاست، بسیار مهم است که مدل داده‌های خود را کارآمد طراحی کنید تا برنامه‌هایتان بیشترین سرعت را داشته باشند.

## ۱۵,۱۵ اشکال‌زدایی

یک الگوی رایج هنگام توسعه یک برنامه پایتون برای اتصال به یک پایگاه‌داده SQLite این است که برنامه پایتون را اجرا کنید و سپس با استفاده از *Database Browser for SQLite* نتایج را بررسی کنید. این مرورگر به شما اجازه می‌دهد خیلی سریع بررسی کنید که آیا برنامه‌تان درست کار می‌کند یا خیر.

اما باید مراقب باشید؛ زیرا SQLite به‌طور جدی تلاش می‌کند تا از تغییر هم‌زمان داده‌ها توسط دو برنامه جلوگیری کند. برای مثال، اگر یک پایگاه‌داده را در مرورگر باز کنید و تغییری در پایگاه‌داده ایجاد کنید، و هنوز دکمه «Save» را در مرورگر فشار نداده باشید، مرورگر فایل پایگاه‌داده را «قفل» می‌کند و مانع از دسترسی هر برنامه دیگر به آن فایل می‌شود. به‌ویژه، اگر پایگاه‌داده قفل باشد، برنامه پایتون شما نمی‌تواند به آن دسترسی پیدا کند.

راه‌حل این است که قبل از اجرای برنامه پایتون، یا مرورگر پایگاه‌داده را ببندید، یا از منوی *File* گزینه *Close Database* را انتخاب کنید تا پایگاه‌داده در مرورگر بسته شود. این کار مانع از آن می‌شود که کد پایتون شما به دلیل قفل بودن پایگاه‌داده با شکست مواجه شود.

## ۱۵,۱۶ واژه‌نامه

**Attribute**: یکی از مقادیر موجود در یک تاپل. در کاربرد رایج‌تر به آن «ستون» یا «فیلد» گفته می‌شود.

**Constraint**: قیدی که به پایگاه داده اعلام می‌کنیم تا قانونی را در مورد یک فیلد یا ردیف اعمال کند. یک قید رایج این است که اصرار داشته باشیم هیچ مقدار تکراری در یک فیلد وجود نداشته باشد (یعنی تمام مقادیر باید یکتا باشند)

**Cursor**: کرسر امکانی است که اجازه می‌دهد در یک پایگاه داده دستورات SQL را اجرا کرده و داده‌ها را از پایگاه داده بازیابی کنید. کرسر مشابه یک سوکت یا دسته فایل برای اتصال‌های شبکه‌ای و فایل‌ها است.

**database browser**: نرم‌افزاری که به شما اجازه می‌دهد مستقیماً به یک پایگاه داده متصل شوید و بدون نوشتن برنامه، داده‌های آن را به‌طور مستقیم مشاهده یا ویرایش کنید.

**foreign key**: یک کلید عددی که به کلید اصلی (primary key) یک ردیف در جدول دیگر اشاره می‌کند. کلیدهای خارجی ارتباط بین ردیف‌های ذخیره‌شده در جدول‌های مختلف را برقرار می‌کنند.

**Index**: داده اضافی که سیستم پایگاه داده هنگام درج ردیف‌ها در جدول نگهداری می‌کند تا جست‌وجوها را بسیار سریع‌تر انجام دهد.

**logical key**: کلیدی که «جهان بیرون» برای یافتن یک ردیف خاص از آن استفاده می‌کند. برای مثال، در جدولی از حساب‌های کاربری، نشانی ایمیل یک فرد می‌تواند یک کلید منطقی مناسب برای داده‌های مربوط به آن کاربر باشد.

**Normalization**: طراحی یک مدل داده به‌گونه‌ای که هیچ داده‌ای تکرار نشود. هر داده تنها در یک نقطه از پایگاه داده ذخیره می‌شود و در دیگر جاها با استفاده از کلید خارجی به آن ارجاع داده می‌شود.

**primary key**: یک کلید عددی که به هر ردیف اختصاص داده می‌شود و برای ارجاع به آن ردیف از جدول دیگر به کار می‌رود. اغلب پایگاه داده طوری پیکربندی می‌شود که این کلیدها را به‌طور خودکار هنگام درج ردیف‌ها تولید کند.

**Relation**: بخشی از یک پایگاه داده که شامل تاپل‌ها و ویژگی‌ها (attributes) است. در اصطلاح رایج‌تر به آن «جدول» گفته می‌شود.

**Tuple**: یک ورودی واحد در یک جدول پایگاه داده که مجموعه‌ای از ویژگی‌ها را تشکیل می‌دهد. در کاربرد رایج‌تر به آن «ردیف» گفته می‌شود.

# فصل شانزدهم

# تجسم‌سازی داده‌ها

تاکنون ما زبان پایتون و سپس نحوه استفاده از پایتون، شبکه و پایگاه‌داده‌ها را برای مدیریت داده‌ها آموخته‌ایم. در این فصل، سه برنامه کامل را بررسی می‌کنیم که همه این موارد را کنار هم قرار می‌دهند تا داده‌ها را مدیریت و تجسم‌سازی کنند. ممکن است از این برنامه‌ها به‌عنوان کد نمونه استفاده کنید تا به شما کمک کنند یک مسئله واقعی را در دنیای واقعی حل کنید. هر یک از این برنامه‌ها یک فایل ZIP است که می‌توانید آن را دانلود کرده، از حالت فشرده خارج کنید و روی رایانه خود اجرا نمایید.

## ۱۶,۱ ساخت یک نقشه OpenStreetMap از داده‌های ژئوکد شده

در این پروژه، ما از **API ژئوکدینگ OpenStreetMap** استفاده می‌کنیم تا مجموعه‌ای از نام دانشگاه‌ها که کاربران به‌صورت آزاد و بدون ساختار وارد کرده‌اند، پاک‌سازی و استانداردسازی کنیم. سپس داده‌های پاک‌سازی‌شده را روی یک **نقشه واقعی OpenStreetMap** قرار می‌دهیم.




برای شروع، برنامه را از این آدرس دانلود کنید:

www.py4e.com/code3/opengeo.zip

اولین مشکلی که باید حل کنیم این است که این API های ژئوکدگذاری دارای محدودیت نرخ هستند و تنها تعداد مشخصی درخواست در هر روز پذیرفته می‌شود. اگر داده زیادی داشته باشید، ممکن است لازم باشد فرآیند جست‌وجو را چندین بار متوقف و دوباره شروع کنید. بنابراین ما مسئله را به دو مرحله تقسیم می‌کنیم.

در مرحله اول، داده‌های ورودی «نظرسنجی» را از فایل **where.data** می‌گیریم و آن را خط‌به‌خط می‌خوانیم، و اطلاعات ژئوکدشده را از گوگل بازیابی کرده و در پایگاه‌داده **geodata.sqlite** ذخیره می‌کنیم. پیش از آنکه برای هر موقعیت واردشده توسط کاربر از API ژئوکدگذاری درخواست کنیم، ابتدا بررسی می‌کنیم که آیا قبلاً داده مربوط به همان خط ورودی را در پایگاه‌داده داریم یا خیر. پایگاه‌داده در اینجا نقش یک «کش محلی» را برای داده‌های ژئوکد شده بازی می‌کند تا مطمئن شویم هرگز یک درخواست را دوبار برای گوگل ارسال نکنیم.

هر زمان می‌توانید با حذف کردن فایل **geodata.sqlite** این فرآیند را دوباره از ابتدا آغاز کنید. برنامه **geoload.py** را اجرا کنید. این برنامه خطوط ورودی فایل **where.data** را می‌خواند و برای هر خط بررسی می‌کند که آیا پیش‌تر در پایگاه‌داده موجود است یا نه. اگر داده‌ای برای آن موقعیت نداشته باشیم، برنامه API ژئوکدگذاری را فراخوانی کرده، داده را بازیابی می‌کند و در پایگاه‌داده ذخیره می‌کند.

در اینجا یک نمونه اجرا را مشاهده می‌کنید که پس از آن انجام شده که مقداری داده از قبل در پایگاه‌داده وجود داشته است:

```
Found in database AGH University of Science and Technology

Found in database Academy of Fine Arts Warsaw Poland

Found in database American University in Cairo

Found in database Arizona State University

Found in database Athens Information Technology

Retrieving https://py4e-data.dr-chuck.net/
   opengeo?q=BITS+Pilani
Retrieved 794 characters {"type":"FeatureColl

Retrieving https://py4e-data.dr-chuck.net/
   opengeo?q=Babcock+University
Retrieved 760 characters {"type":"FeatureColl

Retrieving https://py4e-data.dr-chuck.net/
   opengeo?q=Banaras+Hindu+University
Retrieved 866 characters {"type":"FeatureColl
...
```

پنج موقعیت اول از قبل در پایگاه داده موجود هستند و بنابراین نادیده گرفته می‌شوند. برنامه داده‌ها را تا جایی اسکن می‌کند که به موقعیت‌های جدید برسد و سپس شروع به واکشی آن‌ها می‌کند.

برنامه‌ی **geoload.py** را می‌توان در هر زمانی متوقف کرد، و یک شمارنده وجود دارد که می‌توانید از آن برای محدود کردن تعداد فراخوانی‌ها به API زمین‌کدگذاری (geocoding) در هر اجرا استفاده کنید. با توجه به اینکه فایل **where.data** تنها چند صد داده دارد، شما نباید با محدودیت روزانه نرخ فراخوانی مواجه شوید، اما اگر داده بیشتری داشتید، ممکن بود چندین اجرا در چندین روز طول بکشد تا پایگاه داده شما تمام داده‌های زمین‌کدگذاری‌شده موردنیاز ورودی‌تان را دریافت کند.

وقتی مقداری داده در فایل **geodata.sqlite** بارگذاری کردید، می‌توانید داده‌ها را با استفاده از برنامه **geodump.py** بصری‌سازی کنید. این برنامه پایگاه داده را می‌خواند و فایل **where.js** را با مکان، عرض جغرافیایی و طول جغرافیایی در قالب کد قابل اجرا در جاوااسکریپت ایجاد می‌کند. اجرای برنامه **geodump.py** به صورت زیر است:

```
AGH University of Science and Technology, Czarnowiejska,
Czarna Wieś, Krowodrza, Kraków, Lesser Poland
Voivodeship, 31-126, Poland 50.0657 19.91895

Academy of Fine Arts, Krakowskie Przedmieście,
Northern Śródmieście, Śródmieście, Warsaw, Masovian
Voivodeship, 00-046, Poland 52.239 21.0155
...
260 lines were written to where.js
Open the where.html file in a web browser to view the data.
```

فایل **where.html** شامل کد **HTML** و **JavaScript** است که برای نمایش یک نقشه گوگل استفاده می‌شود. این فایل، تازه‌ترین داده‌های موجود در **where.js** را می‌خواند تا اطلاعاتی را که باید روی نقشه نمایش داده شوند، دریافت کند. در ادامه، قالب یا ساختار فایل **where.js** آمده است:

```
myData = [
[50.0657,19.91895,
'AGH University of Science and Technology, Czarnowiejska,
Czarna Wieś, Krowodrza, Kraków, Lesser Poland
Voivodeship, 31-126, Poland '],
[52.239,21.0155,
'Academy of Fine Arts, Krakowskie Przedmieściee,
Śródmieście Północne, Śródmieście, Warsaw,
Masovian Voivodeship, 00-046, Poland'],
   ...
];
```

این یک متغیر جاوااسکریپت است که شامل فهرستی از فهرست‌ها است. نحو (Syntax) مربوط به ثابت‌های فهرست در جاوااسکریپت بسیار شبیه پایتون است، بنابراین این ساختار باید برای شما آشنا باشد. به‌سادگی فایل **where.html** را در یک مرورگر باز کنید تا مکان‌ها را ببینید. شما می‌توانید با قرار دادن نشانگر ماوس روی هر پین روی نقشه، موقعیتی را مشاهده کنید که API مکان‌یابی (geocoding)برای ورودی واردشده توسط کاربر برگردانده است. اگر هنگام باز کردن فایل **where.html** نتوانستید هیچ داده‌ای ببینید، ممکن است لازم باشد کنسول جاوااسکریپت یا کنسول توسعه‌دهنده (Developer Console) مرورگر خود را بررسی کنید.

## ۱۶,۲ تصویری‌سازی شبکه‌ها و ارتباطات

در این کاربرد، ما برخی از عملکردهای یک موتور جستجو را انجام می‌دهیم. ابتدا بخش کوچکی از وب را خزیده (spider) و نسخه‌ای ساده‌شده از الگوریتم رتبه‌بندی صفحه گوگل (PageRank) را اجرا می‌کنیم تا تعیین کنیم کدام صفحات بیشترین میزان اتصال را دارند، و سپس رتبه صفحه و میزان ارتباط صفحات در گوشه کوچکی از وب که بررسی کرده‌ایم را تجسم می‌کنیم. برای تولید خروجی بصری، از کتابخانه JavaScript به نام **D3** با آدرس: http://d3js.org/ استفاده خواهیم کرد.

شما می‌توانید این برنامه را از لینک زیر دانلود و استخراج کنید:

www.py4e.com/code3/pagerank.zip




برنامه اول (**spider.py**) یک وب‌سایت را خزش (crawl) می‌کند و مجموعه‌ای از صفحات را وارد پایگاه‌داده (**spider.sqlite**) کرده و پیوندهای میان صفحات را ثبت می‌کند. می‌توانید هر زمان که بخواهید با حذف فایل **spider.sqlite** و اجرای دوباره **spider.py** این فرآیند را از نو آغاز کنید.

```
Enter web url or enter: http://www.dr-chuck.com/
['http://www.dr-chuck.com']
How many pages:2
1 http://www.dr-chuck.com/ 12
2 http://www.dr-chuck.com/csev-blog/ 57
How many pages:
```

در این اجرای نمونه، به برنامه گفتیم یک وب‌سایت را خزیده و دو صفحه را بازیابی کند. اگر برنامه را دوباره اجرا کنید و به آن بگویید صفحات بیشتری را بخزد، دیگر صفحاتی را که پیش‌تر در پایگاه داده ذخیره شده‌اند، دوباره خزیده نخواهد کرد. هنگام اجرای مجدد، برنامه به یک صفحه انتخاب‌شده به‌صورت تصادفی که هنوز خزیده نشده است می‌رود و از آن‌جا شروع می‌کند. بنابراین هر اجرای بعدی برنامه spider.py به‌صورت افزایشی عمل می‌کند.

```
Enter web url or enter: http://www.dr-chuck.com/
['http://www.dr-chuck.com']
How many pages:3
3 http://www.dr-chuck.com/csev-blog 57
4 http://www.dr-chuck.com/dr-chuck/resume/speaking.htm 1
5 http://www.dr-chuck.com/dr-chuck/resume/index.htm 13
How many pages:
```

می‌توانید چندین نقطه شروع در یک پایگاه داده داشته باشید—در داخل برنامه، به این نقاط "وب‌ها" گفته می‌شود. اسپایدر به‌صورت تصادفی از میان تمام لینک‌های بازدیدنشده در همه وب‌ها یکی را انتخاب می‌کند و آن را برای خزیدنِ بعدی برمی‌گزیند. اگر بخواهید محتوای فایل *spider.sqlite* را dump کنید، می‌توانید *spdump.py* را به این صورت اجرا کنید:

```
(5, None, 1.0, 3, 'http://www.dr-chuck.com/csev-blog')
(3, None, 1.0, 4, 'http://www.dr-chuck.com/dr-chuck/resume/speaking.htm')
(1, None, 1.0, 2, 'http://www.dr-chuck.com/csev-blog/')
(1, None, 1.0, 5, 'http://www.dr-chuck.com/dr-chuck/resume/index.htm')
4 rows.
```

این خروجی تعداد لینک‌های ورودی، رتبه قبلی صفحه، رتبه جدید صفحه، شناسه صفحه و نشانی اینترنتی آن را نشان می‌دهد. برنامه spdump.py فقط صفحاتی را نمایش می‌دهد که حداقل یک لینک ورودی دارند. پس از اینکه چند صفحه در پایگاه داده داشتید، می‌توانید با اجرای برنامه sprank.py رتبه‌بندی صفحات را انجام دهید. تنها کافی است به آن بگویید که چند بار حلقه رتبه‌بندی صفحه (page rank iterations) را تکرار کند.

تعداد تکرارها: ۲

```
1 0.546848992536
2 0.226714939664
[(1, 0.559), (2, 0.659), (3, 0.985), (4, 2.135), (5, 0.659)]
```

می‌توانید دوباره پایگاه داده را استخراج کنید تا ببینید مقدار Page Rank به‌روزرسانی شده است.

```
(5, 1.0, 0.985, 3, 'http://www.dr-chuck.com/csev-blog')
(3, 1.0, 2.135, 4, 'http://www.dr-chuck.com/dr-chuck/resume/speaking.htm')
(1, 1.0, 0.659, 2, 'http://www.dr-chuck.com/csev-blog/')
(1, 1.0, 0.659, 5, 'http://www.dr-chuck.com/dr-chuck/resume/index.htm')
4 rows.
```

شما می‌توانید sprank.py را هرچقدر که بخواهید اجرا کنید و این برنامه صرفاً هر بار PageRank را دقیق‌تر می‌کند. حتی می‌توانید چند بار sprank.py را اجرا کنید و سپس با spider.py چند صفحه دیگر را crawl کنید و بعد دوباره sprank.py را اجرا کنید تا مقادیر PageRank دوباره همگرا شوند. یک موتور جستجو معمولاً برنامه‌های crawling و ranking را به‌طور مداوم اجرا می‌کند.

اگر می‌خواهید محاسبات PageRank را بدون انجام مجدد عملیات spidering روی صفحات وب از نو شروع کنید، می‌توانید از spreset.pyاستفاده کنید و سپس sprank.py را دوباره اجرا کنید.

تعداد تکرارها: ۵۰

```
1 0.546848992536
2 0.226714939664
3 0.0659516187242
4 0.0244199333
5 0.0102096489546
6 0.00610244329379
...
42 0.000109076928206
43 9.91987599002e-05
44 9.02151706798e-05
45 8.20451504471e-05
46 7.46150183837e-05
47 6.7857770908e-05
48 6.17124694224e-05
49 5.61236959327e-05
50 5.10410499467e-05
[(512, 0.0296), (1, 12.79), (2, 28.93), (3, 6.808), (4, 13.46)]
```

برای هر تکرار الگوریتمPage Rank ، این برنامه میانگین میزان تغییر رتبه صفحه را برای هر صفحه چاپ می‌کند. در ابتدا، شبکه بسیار نامتوازن است و به همین دلیل مقادیر Page Rank مربوط به صفحات مختلف بین تکرارها به‌طور قابل‌توجهی تغییر می‌کنند. اما پس از چند تکرار کوتاه، Page Rankبه حالت پایدار می‌رسد و همگرا می‌شود. شما باید sprank.py را آن‌قدر اجرا کنید تا مقادیر Page Rank به مرحله همگرایی برسند.

اگر بخواهید صفحات برتر فعلی را از نظر Page Rank مشاهده کنید، برنامه spjson.py را اجرا کنید تا داده‌های مربوط به صفحاتی که بیشترین پیوند ورودی را دارند از پایگاه داده خوانده شده و در قالب JSON تولید شود تا بتوان آن را در مرورگر وب مشاهده کرد.

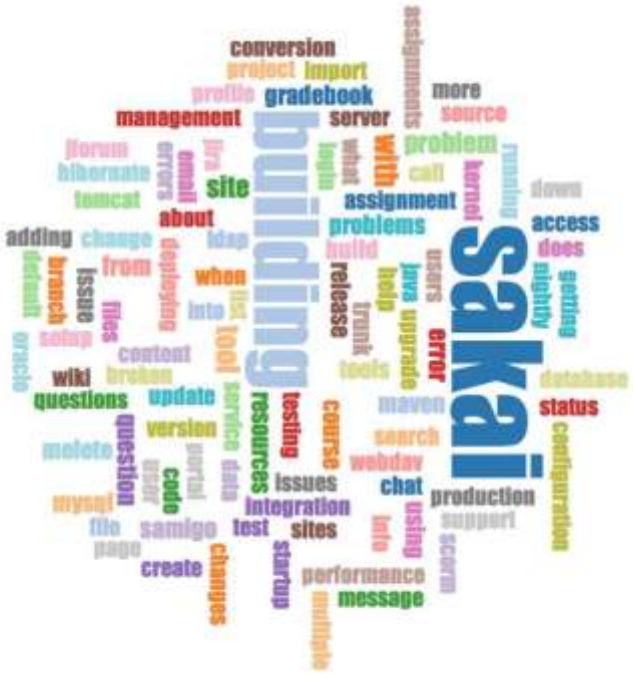
```
Creating JSON output on spider.json...
How many nodes? 30
Open force.html in a browser to view the visualization
```

شما می‌توانید با باز کردن فایل force.html در مرورگر خود، این داده‌ها را مشاهده کنید. این فایل یک چیدمان خودکار از گره‌ها و پیوندها را نمایش می‌دهد. شما می‌توانید روی هر گره کلیک کرده و آن را بکشید، و همچنین می‌توانید روی هر گره دوبار کلیک کنید تا نشانی اینترنتی (URL) مربوط به آن گره را مشاهده کنید. اگر ابزارهای دیگر را دوباره اجرا کنید، فایل spjson.py را دوباره اجرا کرده و دکمه refresh را در مرورگر فشار دهید تا داده‌های جدید از spider.json دریافت شود.

## ۱۶,۳ تصویری‌سازی داده‌های ایمیل

تا این بخش از کتاب، شما کاملاً با فایل‌های داده **mbox-short.txt** و **mbox.txt** آشنا شده‌اید. اکنون زمان آن رسیده است که تحلیل خود از داده‌های ایمیل را به سطح بالاتری ببرید.

در دنیای واقعی، گاهی لازم است داده‌های ایمیل را از سرورها دریافت کنید. این کار ممکن است زمان زیادی ببرد و داده‌ها ممکن است ناسازگار، پر از خطا و نیازمند پاک‌سازی یا اصلاح فراوان باشند. در این بخش، ما با یک برنامه کاربردی کار می‌کنیم که تاکنون پیچیده‌ترین نمونه بوده است، و نزدیک به یک گیگابایت داده را دریافت کرده و آن را تصویری‌سازی می‌کنیم.




**یک ابرِ واژگان از فهرست** Sakai Developer List

می‌توانید این برنامه را از اینجا دانلود کنید: https://www.py4e.com/code3/gmane.zip

ما از داده‌های یک سرویس رایگان آرشیو فهرست‌های ایمیل استفاده خواهیم کرد که نام آن ***gmane*** بود ــ این سرویس اکنون تعطیل شده است و برای اهداف این دوره، بخشی از آرشیو آن در آدرس زیر نگهداری شده است: **http://mbox.dr-chuck.net** سرویس gmane در بین پروژه‌های متن‌باز بسیار محبوب بود، زیرا یک آرشیو قابل جستجوی مناسب از فعالیت‌های ایمیلی آنها فراهم می‌کرد.

http://mbox.dr-chuck.net/export.php

وقتی داده‌های ایمیل مربوط به پروژه ساکای با استفاده از این نرم‌افزار جمع‌آوری (spider) شد، این فرایند نزدیک به یک گیگابایت داده تولید کرد و طی چند روز و در چندین نوبت اجرا انجام شد. فایل README.txt موجود در فایل ZIP بالا ممکن است شامل دستورالعمل‌هایی باشد درباره این‌که چگونه می‌توانید یک نسخه از پیش جمع‌آوری‌شده فایل content.sqlite را برای بخش عمده‌ای از مجموعه ایمیل‌های ساکای دانلود کنید تا مجبور نباشید پنج روز منتظر بمانید تا برنامه‌ها را اجرا کنید. اگر نسخه از پیش جمع‌آوری‌شده را دانلود کردید، همچنان باید فرایند spidering را اجرا کنید تا پیام‌های جدیدتر نیز به‌روزرسانی شوند.

اولین گام، جمع‌آوری (spider) کردن مخزن است. آدرس پایه (base URL) در فایل gmane.py به‌صورت ثابت (hard-coded) تعریف شده و برای فهرست ایمیلی توسعه‌دهندگان ساکای تنظیم شده است. شما می‌توانید با تغییر این آدرس پایه، یک مخزن دیگر را جمع‌آوری کنید. مطمئن شوید که اگر آدرس پایه را تغییر می‌دهید، فایل content.sqlite را حذف کنید.

فایل gmane.py مانند یک spider با نقش کش مسئولانه عمل می‌کند؛ به این معنا که با سرعت پایین اجرا می‌شود و برای جلوگیری از محدودسازی(throttling) ، تنها یک پیام ایمیل در هر ثانیه دریافت می‌کند. این برنامه تمام داده‌های خود را در یک پایگاه‌داده ذخیره می‌کند و می‌توان آن را هر زمان که لازم بود متوقف و دوباره اجرا کرد. ممکن است جمع‌آوری کل داده‌ها چندین ساعت طول بکشد، بنابراین لازم است چندین بار برنامه را مجدداً اجرا کنید.

در این‌جا نمونه‌ای از اجرای gmane.py برای برداشت پنج پیام آخر از فهرست ایمیلی توسعه‌دهندگان ساکای آمده است:

```
How many messages:10
http://mbox.dr-chuck.net/sakai.devel/51410/51411 9460
    nealcaidin@sakaifoundation.org 2013-04-05 re: [building ...
http://mbox.dr-chuck.net/sakai.devel/51411/51412 3379
    samuelgutierrezjimenez@gmail.com 2013-04-06 re: [building ...
http://mbox.dr-chuck.net/sakai.devel/51412/51413 9903
    da1@vt.edu 2013-04-05 [building sakai] melete 2.9 oracle ...
http://mbox.dr-chuck.net/sakai.devel/51413/51414 349265
    m.shedid@elraed-it.com 2013-04-07 [building sakai] ...
http://mbox.dr-chuck.net/sakai.devel/51414/51415 3481
    samuelgutierrezjimenez@gmail.com 2013-04-07 re: ...
http://mbox.dr-chuck.net/sakai.devel/51415/51416 0

Does not start with From
```

این برنامه فایل content.sqlite را از پیام شماره یک تا اولین شماره پیامی که تاکنون اسپایدر نشده اسکن می‌کند و اسپایدر کردن را از همان پیام آغاز می‌کند. اسپایدر کردن ادامه می‌یابد تا زمانی که تعداد پیام‌های مورد نظر اسپایدر شود یا به صفحه‌ای برسد که به‌نظر نمی‌رسد پیام با قالب‌بندی درست باشد.

گاهی مخزن یک پیام را کم دارد. ممکن است مدیران پیام‌ها را حذف کرده باشند یا شاید پیام‌ها گم شده باشند. اگر اسپایدر شما متوقف شد و به‌نظر رسید که به یک پیام گم‌شده خورده است، در SQLite Manager یک ردیف با شناسه پیامِ گمشده اضافه کنید و تمام فیلدهای دیگر را خالی بگذارید و سپس gmane.py را دوباره اجرا کنید. این کار روند اسپایدرینگ را از حالت گیرکرده خارج می‌کند و اجازه می‌دهد ادامه یابد. این پیام‌های خالی در مرحله بعدی پردازش نادیده گرفته خواهند شد.

یکی از نکات خوب این است که زمانی که همه پیام‌ها را اسپایدر کردید و آن‌ها در content.sqlite ذخیره شدند، می‌توانید دوباره gmane.pyرا اجرا کنید تا پیام‌های جدیدی که به لیست ارسال می‌شوند نیز گرفته شوند. داده‌های content.sqlite کاملاً خام هستند، با یک مدل داده‌ای ناکارآمد و بدون فشرده‌سازی. این کار عمدی است زیرا به شما امکان می‌دهد در SQLite Manager به content.sqlite نگاه کنید و مشکلات فرآیند اسپایدرینگ را خطایابی کنید. اجرای هرگونه کوئری روی این پایگاه داده ایده بدی است، زیرا بسیار کند خواهد بود.

مرحله دوم اجرای برنامه gmodel.py است. این برنامه داده خام را از content.sqlite می‌خواند و نسخه‌ای پاک‌سازی‌شده و با مدل‌سازی مناسب از داده‌ها را در فایل index.sqlite تولید می‌کند. این فایل بسیار کوچک‌تر خواهد بود (اغلب ۱۰ برابر کوچک‌تر) زیرا متن هدر و بدنه پیام‌ها را نیز فشرده می‌کند.

هر بار که gmodel.py اجرا می‌شود، فایل index.sqlite را حذف و دوباره ایجاد می‌کند؛ این کار به شما اجازه می‌دهد پارامترهای آن را تنظیم کنید و جدول‌های نگاشت (mapping tables) را در content.sqlite ویرایش کنید تا فرآیند پاک‌سازی داده را اصلاح کنید. این یک نمونه از اجرای gmodel.py است. این برنامه هر ۲۵۰ پیام پردازش‌شده یک خط چاپ می‌کند تا بتوانید مشاهده کنید که پیشرفتی در حال رخ‌دادن است، زیرا ممکن است این برنامه مدت زیادی برای پردازش تقریباً یک گیگابایت داده ایمیل زمان ببرد.

```
Loaded allsenders 1588 and mapping 28 dns mapping 1
1 2005-12-08T23:34:30-06:00 ggolden22@mac.com
251 2005-12-22T10:03:20-08:00 tpamsler@ucdavis.edu
501 2006-01-12T11:17:34-05:00 lance@indiana.edu
751 2006-01-24T11:13:28-08:00 vrajgopalan@ucmerced.edu
...
```

برنامه **gmodel.py** چندین کار مهم در زمینه پاک‌سازی داده را انجام می‌دهد. نام دامنه‌ها برای پسوندهای **com.، org.، edu.**و **net.** به **دو سطح** کاهش داده می‌شوند. سایر دامنه‌ها به **سه سطح** محدود می‌شوند. برای مثال:

- si.umich.edu به umich.edu تبدیل می‌شود.
- caret.cam.ac.ukبه cam.ac.uk تبدیل می‌شود.

همچنین آدرس‌های ایمیل به حروف کوچک تبدیل می‌شوند، و برخی از آدرس‌های @gmane.org مانند نمونه زیر نیز اصلاح می‌گردند:

```
arwhyte-63aXycvo3TyHXe+LvDLADg@public.gmane.org
```

هر زمان که در سایر بخش‌های مجموعه پیام‌ها یک نشانی ایمیل واقعیِ متناظر یافت شود، این نشانی‌های @gmane.org به نشانی واقعی تبدیل می‌شوند. در پایگاه داده *mapping.sqlite* دو جدول وجود دارد که به شما امکان می‌دهد هم نام‌های دامنه و هم نشانی‌های ایمیلی را که در طول دوره فعالیت فهرست ایمیل تغییر کرده‌اند، نگاشت (map) کنید. برای مثال، *استیو گیتنز* در طول مدت حضور خود در فهرست توسعه‌دهندگان ساکای، از نشانی‌های ایمیل زیر استفاده کرده است:

```
s-githens@northwestern.edu
sgithens@cam.ac.uk
swgithen@mtu.edu
```

ما می‌توانیم دو ورودی به جدول *Mapping* در فایل **mapping.sqlite** اضافه کنیم تا برنامه **gmodel.py** هر سه آدرس را به یک آدرس واحد نگاشت کند.

```
s-githens@northwestern.edu ->  swgithen@mtu.edu
sgithens@cam.ac.uk -> swgithen@mtu.edu
```

شما می‌توانید ورودی‌های مشابهی را در جدول **DNSMapping** نیز اضافه کنید، اگر چندین نام دامنه (DNS) وجود داشته باشد که بخواهید همگی به یک نام دامنه واحد نگاشت شوند. نگاشت زیر به داده‌های مربوط به پروژه Sakai افزوده شده است:

```
iupui.edu -> indiana.edu
```

همه حساب‌های کاربری مربوط به پردیس‌های مختلف دانشگاه ایندیانا به‌صورت یک‌جا دنبال می‌شوند. می‌توانی gmodel.py را بارها و بارها اجرا کنی و هنگام بررسی داده‌ها، نگاشت‌های بیشتری اضافه کنی تا داده‌ها تمیزتر و سازگارتر شوند. وقتی کارت تمام شد، یک نسخه کاملاً نمایه‌سازی‌شده از ایمیل‌ها در فایل *index.sqlite* خواهی داشت. این فایل، همان فایلی است که باید برای تحلیل داده از آن استفاده کنی. با وجود این فایل، تحلیل داده بسیار سریع خواهد بود. اولین و ساده‌ترین تحلیل داده این است که مشخص کنیم «چه کسی بیشترین ایمیل را ارسال کرده است؟» و «کدام سازمان بیشترین ایمیل را ارسال کرده است؟». این کار با استفاده از برنامه *gbasic.py* انجام می‌شود.

```
How many to dump? 5
Loaded messages= 51330 subjects= 25033 senders= 1584

Top 5 Email list participants
steve.swinsburg@gmail.com 2657
azeckoski@unicon.net 1742
ieb@tfd.co.uk 1591
csev@umich.edu 1304
david.horwitz@uct.ac.za 1184

Top 5 Email list organizations
gmail.com 7339
```

```
umich.edu 6243
uct.ac.za 2451
indiana.edu 2258
unicon.net 2055
```

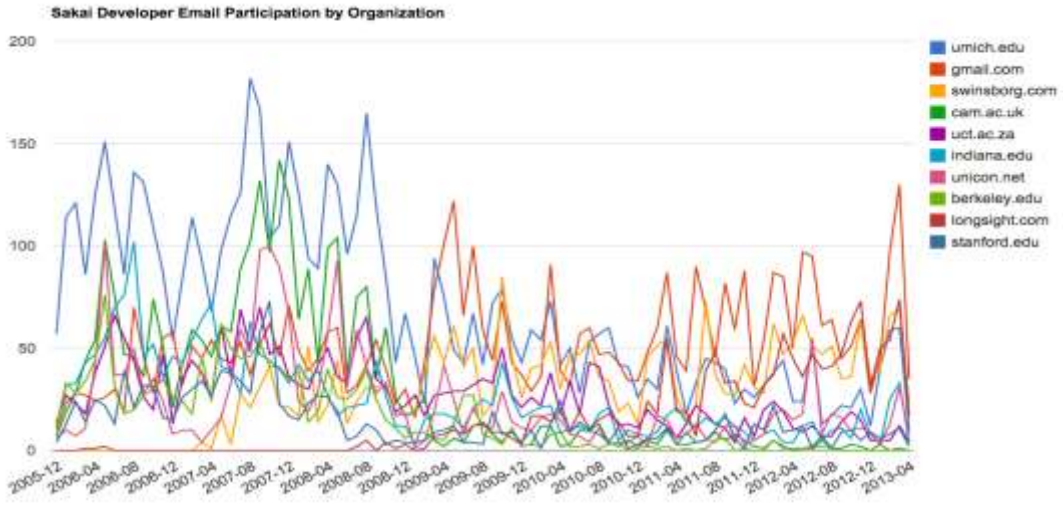
**توجه کنید که gbasic.py با چه سرعت بیشتری نسبت به gmane.py یا حتی gmodel.py اجرا می‌شود.** هر سه برنامه روی یک مجموعه‌داده کار می‌کنند، اما **gbasic.py از داده‌های فشرده و نرمال‌سازی‌شده در فایل index.sqlite استفاده می‌کند.** اگر حجم داده‌های زیادی برای مدیریت داشته باشید، یک فرایند چندمرحله‌ای مانند آنچه در این برنامه به‌کار رفته است، شاید زمان بیشتری برای توسعه نیاز داشته باشد، **اما هنگامی که واقعاً بخواهید داده‌ها را تحلیل و مصورسازی کنید، زمان بسیار زیادی را برای شما صرفه‌جویی خواهد کرد.** شما می‌توانید **یک مصورسازی ساده از فراوانی واژه‌ها در خطوط عنوان ایمیل‌ها** با استفاده از فایل **gword.py** تولید کنید.

```
Range of counts: 33229 129
Output written to gword.js
```

این فایل **gword.js** را تولید می‌کند که می‌توانید آن را با استفاده از **gword.htm** برای ساخت یک «ابرِ واژگان» مشابه آنچه در ابتدای این بخش دیدید، مشاهده کنید. یک مصورسازیِ دوم توسط **gline.py** تولید می‌شود. این برنامه میزان مشارکت ایمیلیِ سازمان‌ها را در طول زمان محاسبه می‌کند.

```
Loaded messages= 51330 senders= 1584
Top 10 Oranizations
['gmail.com', 'umich.edu', 'uct.ac.za', 'indiana.edu',
'unicon.net', 'tfd.co.uk', 'berkeley.edu', 'longsight.com',
'stanford.edu', 'ox.ac.uk']
Output written to gline.js
```

خروجیِ این برنامه در فایل gline.js نوشته می‌شود که با استفاده از gline.htm قابلِ مشاهده است.



این خروجی در فایل **gline.js** نوشته می‌شود که با استفاده از **gline.htm** قابل مشاهده است. این یک برنامه نسبتاً پیچیده و پیشرفته است و قابلیت‌های متعددی برای انجام فرایندهای واقعیِ دریافت داده، پاک‌سازی داده‌ها و مصورسازی اطلاعات دارد.

## تاریخچه چاپ

- **۱ ژانویه ۲۰۲۴** –به‌روزرسانی مثال‌ها برای Python 3.12 ، حذف ارجاعات به API های توییتر، بازنویسی فصل پایگاه‌داده‌ها
- **۲۹ ژوئن ۲۰۲۳** –اعمال بسیاری از اصلاحات، تغییر از API های گوگل به API های OpenStreetMap
- **۵ ژوئیه ۲۰۱۶** –نخستین نسخه کامل برای پایتون ۳٫۰
- **۲۰ دسامبر ۲۰۱۵** –نسخه اولیه و خام تبدیل‌شده به پایتون ۳٫۰

## جزئیات حق‌تألیف

حق‌تألیف ۲۰۰۹ تا کنون – دکتر چارلز آر. سیورِنس. این اثر تحت مجوز **Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported** منتشر شده است. این مجوز در آدرس زیر در دسترس است: http://creativecommons.org/licenses/by-nc-sa/3.0/ شما می‌توانید در پیوست با عنوان **«جزئیات حق‌تألیف»** مشاهده کنید که نویسنده چه نوع استفاده‌هایی را تجاری و غیرتجاری تلقی می‌کند و همچنین چه معافیت‌هایی برای این مجوز در نظر گرفته شده است.

## **پیشگفتار بازآفرینی یک کتابِ باز**

برای اعضای هیئت علمی که دائماً به آن‌ها گفته می‌شود «انتشار بده یا محو شو»، کاملاً طبیعی است که بخواهند همیشه چیزی را از صفر خلق کنند؛ آثاری که ساخته خودشان و حاصل آفرینشی تازه باشد. این کتاب، تلاشی است در جهت *شروع نکردن از صفر*، بلکه «بازترکیب» (Remixing) کتاب **Think Python: How to Think Like a Computer Scientist** که توسط **آلن بی. داونی، جف اکنر، و دیگران** نوشته شده است.

در دسامبر ۲۰۰۹، زمانی که برای پنجمین نیم‌سال متوالی در حال آماده‌سازی تدریس درس **SI502 - برنامه‌نویسی شبکه‌ای** در دانشگاه میشیگان بودم، تصمیم گرفتم زمان آن رسیده است که یک کتاب درسی پایتون بنویسم؛ کتابی که **به‌جای تمرکز بر الگوریتم‌ها و انتزاع‌ها، بر کاوش و تحلیل داده تمرکز کند**. هدف من در SI502 این بود که به دانشجویان **مهارت‌های مادام‌العمر در کار با داده‌ها** با استفاده از پایتون بیاموزم. تعداد کمی از دانشجویانم قصد داشتند برنامه‌نویس حرفه‌ای شوند. در عوض، آن‌ها قرار بود **کتابدار، مدیر، وکیل، زیست‌شناس، اقتصاددان** و... شوند که تنها می‌خواستند از فناوری، به‌صورت ماهرانه و هدفمند، در مسیر حرفه‌ای خود استفاده کنند.

من هیچ‌وقت نتوانستم کتاب پایتونی پیدا کنم که کاملاً مناسب یک دوره‌ی داده‌محور باشد، بنابراین تصمیم گرفتم دقیقاً چنین کتابی را خودم بنویسم. خوشبختانه، در یک جلسه‌ی هیئت‌علمی سه هفته پیش از آن‌که بخواهم در تعطیلات زمستانی کتاب جدیدم را از صفر شروع کنم، دکتر آتول پراکاش کتاب *Think Python* را به من نشان داد؛ کتابی که او همان ترم برای تدریس درس پایتون خود استفاده کرده بود. این کتاب یک متن درسی خوش‌نوشت در علوم کامپیوتر است که بر توضیحات کوتاه، مستقیم و سهولت یادگیری تمرکز دارد.

ساختار کلی کتاب تغییر کرده تا بتوانیم هرچه سریع‌تر به حل مسائل تحلیل داده برسیم و از همان ابتدا مجموعه‌ای از مثال‌ها و تمرین‌های پیوسته درباره‌ی تحلیل داده ارائه شود.

فصل‌های ۲ تا ۱۰ مشابه کتاب *Think Python* هستند، اما تغییرات اساسی داشته‌اند. مثال‌ها و تمرین‌های عددی جای خود را به تمرین‌های داده‌محور داده‌اند. موضوعات به ترتیبی ارائه شده‌اند که برای ساخت راه‌حل‌های تحلیل داده‌ی پیچیده‌تر مورد نیاز است. برخی موضوعات مانند *try* و *except* جلوتر آورده شده و به عنوان بخشی از فصل شرط‌ها ارائه شده‌اند. توابع، تنها زمانی که برای مدیریت پیچیدگی برنامه لازم بوده‌اند، به‌طور مختصر معرفی شده‌اند، نه اینکه به‌عنوان یک درس اولیه درباره‌ی انتزاع مطرح شوند. تقریباً تمام توابع تعریف‌شده توسط کاربر (user-defined functions) از مثال‌ها و تمرین‌ها حذف شده‌اند، به‌جز فصل ۴. واژه‌ی «بازگشت» (recursion) اصلاً در کتاب به کار نرفته است.

در فصل‌های ۱ و ۱۱ تا ۱۶، تمام مطالب کاملاً جدید هستند و بر نمونه‌های واقعی و ساده از کاربردهای پایتون در تحلیل داده تمرکز دارند. این موارد شامل عبارت‌های باقاعده برای جست‌وجو و تجزیه، خودکارسازی وظایف روی رایانه، بازیابی داده از طریق شبکه، استخراج داده از صفحات وب، برنامه‌نویسی شیءگرا، استفاده از خدمات وب، تجزیه داده‌های XML وJSON ، ایجاد و استفاده از پایگاه‌های داده با بهره‌گیری از زبان ساخت‌یافته پرس‌وجو(SQL) ، و بصری‌سازی داده‌ها هستند.

هدف نهایی همه این تغییرات، انتقال تمرکز از علوم کامپیوتر به «اطلاع‌رسانی» (Informatics) و گنجاندن تنها موضوعاتی در یک درس مقدماتی فناوری است که حتی اگر فرد تصمیم بگیرد برنامه‌نویس حرفه‌ای نشود، همچنان برای او مفید باشند.

دانشجویانی که این کتاب برایشان جذاب است و می‌خواهند بیشتر کاوش کنند، باید به کتاب *Think Python* نوشته آلن بی. داونی مراجعه کنند. از آنجا که میان این دو کتاب هم‌پوشانی زیادی وجود دارد، دانشجویان به‌سرعت مهارت‌های بیشتری در حوزه‌های برنامه‌نویسی فنی و تفکر الگوریتمی که در *Think Python* پوشش داده شده‌اند، به دست خواهند آورد. و با توجه به شباهت سبک نگارشی دو کتاب، آن‌ها می‌توانند با کمترین زحمت و با سرعت از *Think Python* عبور کنند.

به‌عنوان دارنده حق‌تألیف *Think Python*، آلن (Allen B. Downey) به من اجازه داده است که مجوز کتاب را برای بخش‌هایی از اثر او که در این کتاب باقی مانده‌اند، از «مجوز مستندات آزاد گنو(GFDL) » به «مجوز جدیدتر Creative Commons Attribution—ShareAlike» تغییر دهم. این تغییر در راستای حرکت کلی اسناد باز از GFDL به CC-BY-SA است (برای مثال، ویکی‌پدیا نیز همین مسیر را پیموده است). استفاده از مجوز CC-BY-SA سنت قدرتمند *copyleft* کتاب را حفظ می‌کند و در عین حال استفاده دوباره و بازآفرینی این محتوا را برای نویسندگان جدید بسیار ساده‌تر و مستقیم‌تر می‌سازد.

من معتقدم این کتاب نمونه‌ای است از اینکه چرا منابع باز برای آینده آموزش بسیار مهم هستند، و می‌خواهم از آلن بی. داونی و انتشارات کمبریج برای تصمیم پیشرو و آینده‌نگرانه‌شان در دسترس قرار دادن این کتاب تحت یک حق‌تألیف باز تشکر کنم. امیدوارم آن‌ها از نتیجه تلاش‌های من خرسند باشند و همچنین امیدوارم شما، خواننده گرامی، نیز از تلاش جمعی ما رضایت داشته باشید.

مایلم از آلن بی. داونی و لورن کولز بابت کمک‌ها، شکیبایی و راهنمایی‌هایشان در رسیدگی و حل مسائل مربوط به حق‌تألیف این کتاب تشکر کنم.

Charles Severance
www.dr-chuck.com
Ann Arbor, MI, USA
September 9, 2013

چارلز سیورِنس استاد بالینی (Clinical Associate Professor) در دانشکده اطلاعات دانشگاه میشیگان است.

## مشارکت‌ها

### ترجمه‌ها

این کتاب به چندین زبان ترجمه شده است که می‌توانید آن‌ها را در نشانی زیر بیابید: https://www.py4e.com/book

- **اسپانیایی** *Python para todos: Explorando la información con Python 3*
  کتاب ترجمه‌شده، خودکارسنج‌ها(autograders) ، منابع و وب‌سایت در: https://es.py4e.com
  **مشارکت‌کنندگان :**خوان کارلوس پرِز کاستِیانوْس، خوان دوگناک، دانیل مِرینو اِچِوِریا، خایمه بِرمئو رامیرِز، و فرناندو تاردیو.
- **ایتالیایی** *Python per tutti: Esplorare dati con Python3*
  **مشارکت‌کنندگان :**آلِساندرو روسِتی و ویتوره زِن.
- **پرتغالی** *Python Para Todos: Explorando Dados com Python 3*
  **مترجم :**یوری لائیا دِ مِدِیروس.
- **لهستانی** *Python dla wszystkich: Odkrywanie danych z Python 3*
  کتاب ترجمه‌شده، خودکارسنج‌ها، منابع و وب‌سایت در: https://py4e.pl
  **مترجم :**آندرژی وویتوویچ (دانشگاه آدم میکِویچ در پوزنان، لهستان)
- **یونانی** –کتاب ترجمه‌شده، خودکارسنج‌ها، منابع و وب‌سایت در: https://gr.py4e.com
  **مترجم :**کونستانتیا کیورتیدو.
- **عربی** – **مترجم**Electronics Go

هر شخصی که اجازه ترجمه کتاب را دریافت کند، موافقت می‌کند که یک نسخه رایگان از کتاب را به‌صورت آنلاین منتشر کند. برخی از ترجمه‌ها نیز نسخه چاپی دارند و برای خرید در دسترس هستند.

## فهرست مشارکت‌کنندگان برای *Python for Everybody*

منبع کتاب در گیت‌هاب نگهداری می‌شود و می‌توانید تعداد زیادی از مشارکت‌کنندگانی را که با ارسال Pull Request ، پیشنهاد اصلاح و رفع اشکال داده‌اند، پیدا کنید. https://github.com/csev/py4e/

به بخش **Insights → Contributors** مراجعه کنید تا همه افرادی را که از طریق گیت‌هاب در این پروژه مشارکت داشته‌اند، مشاهده کنید.

Juan Carlos Perez Castellanos, Juan Dougnac, Daniel Merino Echeverría, Jaime Bermeo Ramírez, Fernando Tardío, Alessandro Rossetti, Vittore Zen, Yuri Loia de Medeiros, Konstantia Kiourtidou, Andrzej Wójtowicz, Elliott Hauser, Stephen Catto, Sue Blumenberg, Tamara Brunnock, Mihaela Mack, Chris Kolosiwsky, Dustin Farley, Jens Leerssen, Naveen KT, Mirza Ibrahimovic, Naveen (@togarnk), Zhou Fangyi, Alistair Walsh, Erica Brody, Jih-Sheng Huang, Louis Luangkesorn, and Michael Fudge

می‌توانید جزئیات مربوط به مشارکت‌ها را در اینجا ببینید: https://github.com/csev/py4e/graphs/contributors

## فهرست مشارکت‌کنندگان برای *Python for Informatics*

Bruce Shields for copy editing early drafts, Sarah Hegge, Steven Cherry, Sarah Kathleen Barbarow, Andrea Parker, Radaphat Chongthammakun, Megan Hixon, Kirby Urner, Sarah Kathleen Barbrow, Katie Kujala, Noah Botimer, Emily Alinder, Mark Thompson-Kular, James Perry, Eric Hofer, Eytan Adar, Peter Robinson, Deborah J. Nelson, Jonathan C. Anthony, Eden Rassette, Jeannette Schroeder, Justin Feezell, Chuanqi Li, Gerald Gordinier, Gavin Thomas Strassel, Ryan Clement, Alissa Talley, Caitlin Holman, Yong-Mi Kim, Karen Stover, Cherie Edmonds, Maria Seiferle, Romer Kristi D. Aranas (RK), Grant Boyer, Hedemarrie Dussan,

## پیش‌گفتار کتاب«Think Python»

**تاریخچه عجیب«Think Python»** *نوشته : آلِن بی. دَونی*

در ژانویه ۱۹۹۹، در حال آماده‌سازی برای تدریس یک درس مقدماتی برنامه‌نویسی با زبان جاوا بودم. سه بار این درس را تدریس کرده بودم و احساس ناکامی می‌کردم. نرخ شکست دانشجویان بسیار بالا بود و حتی در میان دانشجویانی که موفق می‌شدند، میزان کلی یادگیری بسیار پایین بود.

یکی از مشکلاتی که به‌روشنی می‌دیدم، **کتاب‌های درسی** بود. این کتاب‌ها بیش از حد حجیم بودند، جزئیات غیرضروری زیادی درباره جاوا داشتند، و در مقابل، راهنماییِ سطح‌بالای کافی درباره *چگونگی برنامه‌نویسی* ارائه نمی‌کردند. علاوه بر این، همگی گرفتار «اثر درِ تله» (Trap Door Effect) بودند: از آسان شروع می‌شدند، آرام پیش می‌رفتند، و تقریباً حوالی فصل پنجم، ناگهان همه‌چیز از هم می‌پاشید. دانشجویان حجم زیادی از مطالب جدید را در زمانی کوتاه دریافت می‌کردند، و من مجبور بودم باقی ترم را به «جمع کردن تکه‌ها» بپردازم.

دو هفته پیش از شروع کلاس‌ها، تصمیم گرفتم **کتاب خودم** را بنویسم. اهدافم چنین بود:

- کوتاه نگه داشتن کتاب. بهتر است دانشجو ۱۰ صفحه بخواند تا این‌که ۵۰ صفحه را نخواند.
- دقت در واژگان. تلاش کردم اصطلاحات فنی را به حداقل برسانم و هر واژه را هنگام نخستین استفاده تعریف کنم.
- ساخت تدریجی. برای جلوگیری از اثر درِ تله، دشوارترین موضوعات را به مجموعه‌ای از گام‌های کوچک تقسیم کردم.
- تمرکز بر برنامه‌نویسی، نه زبان برنامه‌نویسی. حداقل مجموعه لازم از جاوا را آوردم و بقیه را کنار گذاشتم.

به یک عنوان نیاز داشتم؛ بنابراین از سر شوخی عنوان **چگونه مانند یک دانشمند رایانه‌ای فکر کنیم** را انتخاب کردم.

نسخه نخست، خام بود، اما *کار کرد* .دانشجویان مطالب را می‌خواندند و به اندازه‌ای می‌فهمیدند که من می‌توانستم زمان کلاس را صرف مباحث سخت، موضوعات جذاب و مهم‌تر از همه—تمرین دادن به دانشجویان—کنم. کتاب را تحت مجوز *GNU Free Documentation License* منتشر کردم که اجازه کپی، اصلاح و توزیع کتاب را به خوانندگان می‌دهد.

آنچه بعداً اتفاق افتاد، بخش جذاب ماجراست .**جف الکنر**، یک دبیر دبیرستان در ویرجینیا، کتاب مرا پذیرفت و آن را به زبان پایتون ترجمه کرد. او نسخه ترجمه‌شده را برایم فرستاد و من تجربه عجیبِ یادگیری پایتون را از طریق خواندن **کتاب خودم** پیدا کردم.

جف و من کتاب را بازبینی کردیم، مطالعات موردی نوشته **کریس مایرز** را اضافه کردیم و در سال ۲۰۰۱ کتاب *چگونه مانند یک دانشمند رایانه‌ای فکر کنیم: یادگیری با پایتون* را منتشر کردیم—آن‌هم تحت همان مجوز آزاد. در قالب انتشارات Green Tea Press، نسخه چاپی کتاب را در Amazon و کتاب‌فروشی‌های دانشگاهی عرضه کردم. کتاب‌های دیگر Green Tea Press نیز در greenteapress.com در دسترس هستند.

در سال ۲۰۰۳، تدریس در کالج اولین (Olin College) را آغاز کردم و برای نخستین‌بار موفق شدم پایتون تدریس کنم. تفاوت آن با جاوا *قابل توجه* بود: دانشجویان کمتر مشکل داشتند، بیشتر یاد می‌گرفتند، پروژه‌های جالب‌تری انجام می‌دادند و به‌طور کلی **لذت بیشتری می‌بردند.**

در طول پنج سال اخیر، به توسعه کتاب ادامه دادم—خطاها را اصلاح کردم، برخی مثال‌ها را بهبود بخشیدم و مطالب بیشتری (به‌ویژه تمرین‌ها) اضافه کردم. در سال ۲۰۰۸، کار روی بازنگری عمده کتاب را شروع کردم، و هم‌زمان، ویراستاری از انتشارات دانشگاه کمبریج با من تماس گرفت که علاقه‌مند به انتشار نسخه بعدی بود. زمان‌بندی بسیار خوبی بود.

امیدوارم از کار با این کتاب لذت ببرید و این کتاب کمک کند تا برنامه‌نویسی و اندیشیدن—حداقل اندکی—*همچون یک دانشمند رایانه‌ای* را بیاموزید.

## سپاس‌گزاری‌های کتاب«Think Python»

پیش از هر چیز، از **جف اِلکنر** سپاس‌گزارم که کتاب جاوای من را به پایتون ترجمه کرد—کاری که این پروژه را آغاز کرد و مرا با زبانی آشنا کرد که در نهایت تبدیل به زبان مورد علاقه‌ام شد. از **کریس مایرز** نیز سپاس‌گزارم که چندین بخش به کتاب *چگونه مانند یک دانشمند رایانه‌ای فکر کنیم* اضافه کرد. و از بنیاد نرم‌افزار آزاد (Free Software Foundation) تشکر می‌کنم که مجوز GNU Free Documentation License را ایجاد کرد—مجوزی که همکاری من با جف و کریس را ممکن ساخت.

همچنین از ویراستاران Lulu که روی کتاب *چگونه مانند یک دانشمند رایانه‌ای فکر کنیم* کار کردند، سپاس‌گزارم.

از تمامی دانشجویانی که با نسخه‌های پیشین این کتاب کار کرده‌اند و نیز از تمام مشارکت‌کنندگان (که نامشان در پیوست آمده است) که اصلاحات و پیشنهادهای خود را ارسال کردند، قدردانی می‌کنم. و از همسرم، **لیزا**، برای همکاری‌اش در این کتاب، و برای Green Tea Press، و برای همه چیزهای دیگر نیز سپاس‌گزارم.

**اَلِن بی. دَونی**
**نیدهام، ماساچوست**

## فهرست مشارکت‌کنندگان کتاب«Think Python»

بیش از ۱۰۰ خواننده دقیق و اندیشمند در سال‌های گذشته پیشنهادها و اصلاح‌هایی ارسال کرده‌اند. مشارکت‌ها و اشتیاق آنان نسبت به این پروژه کمک بزرگی بوده است. برای اطلاع از جزئیات مربوط به نقش هر یک از این افراد، به متن کتاب «Think Python» مراجعه کنید.

Lloyd Hugh Allen, Yvon Boulianne, Fred Bremmer, Jonah Cohen, Michael Conlon, Benoit Girard, Courtney Gleason and Katherine Smith, Lee Harr, James Kaylin, David Kershaw, Eddie Lam, Man-Yong Lee, David Mayo, Chris McAloon, Matthew J. Moelter, Simon Dicon Montford, John Ouzts, Kevin Parks, David Pool, Michael Schmitt, Robin Shaw, Paul Sleigh, Craig T. Snydal, Ian Thomas, Keith Verheyden, Peter Winstanley, Chris Wrobel, Moshe Zadka, Christoph Zwerschke, James Mayer, Hayden McAfee, Angel Arnal, Tauhidul Hoque and Lex Berezhny, Dr. Michele Alzetta, Andy Mitchell, Kalin Harvey, Christopher P. Smith, David Hutchins, Gregor Lingl, Julie Peters, Florin Oprina, D. J. Webre, Ken, Ivo Wever, Curtis Yanko, Ben Logan, Jason Armstrong, Louis Cordier, Brian Cain, Rob Black, Jean-Philippe Rey at Ecole Centrale Paris, Jason Mader at George Washington University made a number Jan Gundtofte-Bruun, Abel David and Alexis Dinno, Charles Thayer, Roger Sperberg, Sam Bull, Andrew Cheung, C. Corey Capel, Alessandra, Wim Champagne, Douglas Wright, Jared Spindor, Lin Peiheng, Ray Hagtvedt, Torsten Hübsch, Inga Petuhhov, Arne Babenhauserheide, Mark E. Casida, Scott Tyler, Gordon Shephard, Andrew Turner, Adam Hobart, Daryl Hammond and Sarah Zimmerman, George Sass, Brian Bingham, Leah Engelbert-Fenton, Joe Funke, Chao-chao Chen, Jeff Paine, Lubos Pintes, Gregg Lind and Abigail Heithoff, Max Hailperin, Chotipat Pornavalai, Stanislaw Antol, Eric Pashman, Miguel Azevedo, Jianhua Liu, Nick King, Martin Zuther, Adam Zimmerman, Ratnakar Tiwari, Anurag Goel, Kelli Kratzer, Mark Griffiths, Roydan Ongie, Patryk Wolowiec, Mark Chonofsky, Russell

Coleman, Wei Huang, Karen Barber, Nam Nguyen, Stéphane Morin, Fernando Tardio, and Paul Stoop.

## جزئیات حق نشر

این اثر تحت مجوز **Creative Commons Attribution–NonCommercial–ShareAlike 3.0 Unported License** منتشر شده است. نسخه کامل این مجوز در نشانی زیر در دسترس است:

creativecommons.org/licenses/by-nc-sa/3.0/.

من ترجیح می‌دادم این کتاب را تحت مجوز کمتر محدودکننده **CC-BY-SA** منتشر کنم. اما متأسفانه چند سازمان سودجو وجود دارند که کتاب‌های دارای مجوز آزاد را جست‌وجو می‌کنند و سپس نسخه‌های تقریباً بدون‌تغییر آن‌ها را از طریق سرویس‌های چاپ بر اساس تقاضا، مانند **LuLu** یا **KDP**، منتشر کرده و می‌فروشند. خوشبختانه، KDP سیاستی را اضافه کرده است که خواست‌های دارنده واقعی حق‌مؤلف را بر تلاش افراد غیرمالک برای انتشار آثار دارای مجوز آزاد، مقدم می‌داند. اما سرویس‌های چاپ بر اساس تقاضا بسیارند و تعداد کمی از آن‌ها سیاستی به‌خوبی KDP دارند.

با تأسف، من عنصر **NC** را به مجوز این کتاب افزودم تا در صورتی که فردی تلاش کند کتاب را تکثیر کرده و به‌صورت تجاری بفروشد، امکان پیگیری داشته باشم. متأسفانه افزودن عنصر NC استفاده‌هایی از این اثر را که مایل بودم آزاد باشند محدود می‌کند. به همین دلیل این بخش از سند را اضافه کرده‌ام تا شرایط خاصی را توضیح دهم که در آن‌ها از پیش اجازه می‌دهم مطالب این کتاب در موقعیت‌هایی استفاده شوند که ممکن است از نظر برخی تجاری تلقی شوند.

اگر شما نسخه‌های محدودی از تمام یا بخشی از این کتاب را برای استفاده در یک دوره آموزشی (مثلاً به‌صورت *coursepack*) چاپ می‌کنید، برای این منظور مجوز **CC-BY** به شما اعطا می‌شود. اگر در دانشگاه تدریس می‌کنید و این کتاب را به زبانی غیر از انگلیسی ترجمه کرده و با استفاده از نسخه ترجمه‌شده تدریس می‌کنید، می‌توانید با من تماس بگیرید تا برای انتشار ترجمه شما، مجوز **CC-BY-SA** اعطا کنم. به‌ویژه، به شما اجازه داده می‌شود کتاب ترجمه‌شده را به‌صورت تجاری نیز به فروش برسانید.

اگر قصد ترجمه کتاب را دارید، شاید بهتر باشد با من تماس بگیرید تا مطمئن شویم که همه مواد درسی مرتبط را نیز در اختیار دارید تا بتوانید آن‌ها را ترجمه کنید.

البته، اگر موارد بالا کافی نیستند، می‌توانید آزادانه با من تماس بگیرید و درخواست اجازه کنید. در تمامی موارد، اجازه استفاده و ترکیب مجدد این اثر زمانی داده خواهد شد که در نتیجه کار جدید، **ارزش افزوده روشن یا منفعت قابل توجهی برای دانشجویان یا مدرسان ایجاد شود.**

**چارلز سِورِنس**
**www.dr-chuck.com**
**ان آربر، میشیگان، ایالات متحده**
**۹ سپتامبر ۲۰۱۳**