# پایتون برای همه

## کاوش داده در پایتون ۳

این کتاب برای آموزش برنامه‌نویسی، حتی به کسانی که هیچ تجربه‌ی پیشینی نداشته‌اند نوشته شده است. از این کتاب جهت آموزش برنامه‌نویسی به هر کسی می‌توانید استفاده کنید.

نوشته: دکتر چارلز سورنس
ترجمه: ایمان امینی

# پایتون برای همه

## کاوش داده در پایتون ۳

نوشته: **دکتر چارلز سورنس**

ترجمه: **ایمان امینی**

**زمستان ۹۶**

www.ilola.ir

www.py4e.ir

# فهرست مطالب

# مقدمه‌ای بر ترجمه

«پایتون برای همه» مقدمه‌ایست بر برنامه‌نویسی با پایتون ۳ با تمرکز روی استفاده‌ی عملی و کاربردی از آن. این کتاب در تلاش است تا ساختاری را فراهم کند تا چه دانش‌آموزانی که به دنبال استفاده از پایتون در کارهای آکادمیک‌اند و چه کسانی که مطالعه‌ی جدیِ علوم کامپیوتر را در دستور کار دارند از آن بهره ببرند.

ترجمه‌ی فارسی این اثر توسط ایمان امینی با حمایت مالی موسسه تحقیقاتی رامونا تحت گواهی کریییتیو کامنز BY - NC ارائه می‌شود. شما می‌توانید با توجه به مفاد این گواهی از آن استفاده کنید.

اگر کتاب حاضر اولین تلاش شما برای یادگیری زبان‌برنامه‌نویسی‌ست، به کلمه‌های ناآشنای زیادی بر خواهید خورد. اگر معادل مناسبی برای کلمات در زبان فارسی وجود داشته باشد، ما از آن معادل‌ها استفاده خواهیم کرد، در غیر اینصورت کلمه‌ی اصلی را با یک توضیح کوچک برای درک آن به کار خواهیم برد. درست مانند یادگیری لغاتِ یک زبان جدید، ممکن است کلمات در ابتدا برای‌تان گیج‌کننده و مبهم باشند؛ لازم است که به خودتان و مغزتان فرصت بدهید تا کم‌کم تفسیر درستی از آن کلمات انجام داده و معنی قابل درکی برای‌تان تداعی کند. به عبارتی دست به گیرنده‌ها نزنید، فقط کمی به خودتان زمان دهید تا مفاهیم را یاد بگیرید.

همچنین توصیه می‌کنیم که زمان یادگیری و مطالعه را درست قبل از خواب قرار دهید. دلیلش این است که زمانی‌که شما درگیر مفاهیم جدید شوید، مغز شما نیاز به تفکر در حالت پراکنده[I] برای پردازش مفاهیم دارد. اگر شما بعد از مطالعه بخوابید، این فرصت برای مغزتان فراهم می‌شود که پردازش روی کلمات را انجام دهد. بعد از بیدار

[I] Defuse

شدن، شما درک واضح‌تری از آنچه آموختید را خواهید داشت؛ و البته وقتی بلافاصله بعد از مطالعه می‌خوابید، با احتمال بیشتری مغزتان به سراغ واکاوی آنچه درست پیش از خوابیدن در حال کاوشش بودید، می‌رود.

# فصل ۱

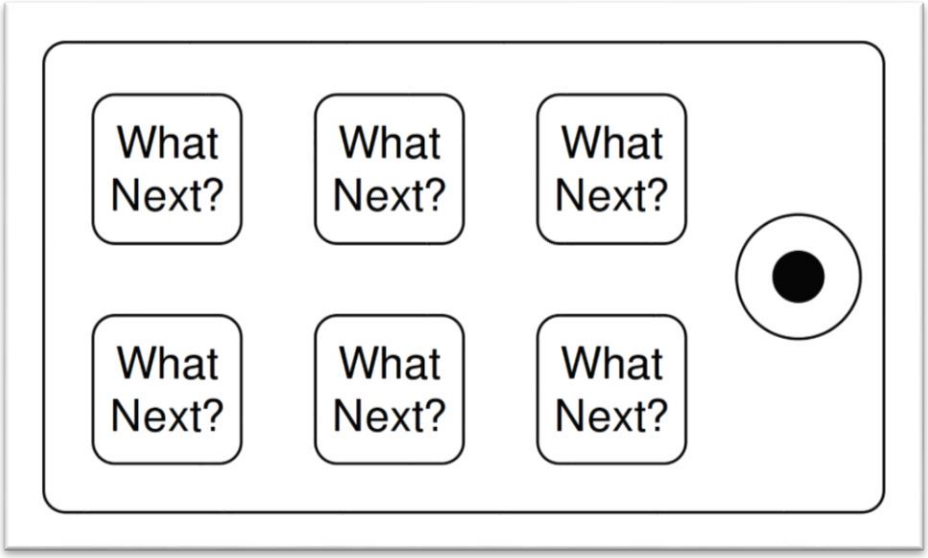
## مقدمه و معرفی

### چرا اصلا باید برنامه‌نویسی یاد بگیرید؟

نوشتن برنامه یا همان برنامه‌نویسی، یک فعالیتی‌ست که ارزشش را دارد و در جایی به خودتان می‌آیید و می‌بینید که خسته‌کننده نیست، حوصله‌تان هم سر نمی‌رود، و تازه از انجامش لذت هم می‌برید. هر کسی دلایل خودش را برای برنامه‌نویسی دارد که اتفاقا کم هم نیستند. شاید بخواهید از طریقش کسب درآمد کنید و یا با استفاده از آن مساله‌ی سختی را در خصوص آنالیز داده‌ها حل کنید؛ شاید تنها به قصد تفریح برنامه‌نویسی کنید و در این بین مسائل دیگران را با آن حل کنید. به هر حال این کتاب فرض می‌کند که همه نیاز دارند که برنامه‌نویسی را یاد بگیرند. وقتی که مهارتش را کسب کردید، می‌توانید به این فکر کنید که خب حالا چه استفاده‌ای از آن کنم.

ما با کامپیوترها محاصره شده‌ایم. حالا این کامپیوترها فقط آن‌هایی نیستند که جعبه‌ی بزرگی دارند و یک مانیتور روی میز جا خوش کرده، بلکه از لپ‌تاپ گرفته تا گوشی موبایلی که همیشه در جیبتان است، یک کامپیوتر به حساب می‌آیند. این کامپیوترهای کوچک و بزرگ کارهای زیادی را برای ما و به جای ما انجام می‌دهند.

سخت‌افزاری که در کامپیوترهای امروزی به کار رفته به طریقی ساخته شده که دائم از شما می‌پرسد «حالا باید چه کاری انجام دهم رئیس؟»




کار برنامه‌نویسان اضافه کردن سیستم‌عامل و دسته‌ای از ابزارها و برنامه‌ها به سخت‌افزار است. چیزی که در نهایت تبدیل به دستیار شخصی دیجیتالی می‌شود. این دستیار شخصی دیجیتال، قادر به انجام کارهای زیادی برای ماست.

کامپیوترهای ما سریع و قدرتمندند و از حافظه‌های انبوهی برخوردارند. اگر ما فقط زبان حرف زدن با آن‌ها و توضیح مسائل برایشان را یاد بگیریم، می‌فهمیم که چقدر می‌توانند کمک‌دست ما باشند. کافیست به کامپیوتر بگویید که «چه کاری در ادامه انجام دهد». وقتی زبانشان را یاد گرفتید پتانسیل واگذاری کارهای کسل‌کننده و تکراری به آن‌ها را با خود دارید. کافیست که کمی با او حرف بزنید و خب او برای‌تان بدون وقفه و خستگی، کار را انجام خواهد داد.

به عنوان مثال به سه پاراگراف ابتدایی این فصل نگاه کنید و بگویید که چه کلمه‌ای بیشترین تکرار را دارد و تعداد این تکرارها چقدر است. درست است که کلمه‌های این کتاب را می‌خوانید و متوجه می‌شوید، ولی شمردن کلمات و محاسبه‌ی کلمه‌ای که

بیشترین تکرار را داشته یک کار سختِ طاقت‌فرسا و کسل‌کننده برای ماست. در اصل مغز ما برای این کار طراحی نشده است. برای کامپیوترها برعکسش صادق است. یعنی خواندن کلمات از روی کاغذ و فهمیدنشان برای کامپیوتر سخت است، در عوض شمردن کلمات و نشان دادن تعدادِ تکرار هر کلمه برایش بسیار ساده است. مثلا در نسخه‌ی انگلیسی این کتاب کدام کلمه بیشتر تکرار شده و به چه تعداد؟ کافیست که برنامه‌ی زیر را که قبلا نوشته‌ایم، روی پاراگراف‌های نخست این فصل اجرا کنیم:

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

دستیار آنالیز اطلاعات شخصیِ» ما در کسری از ثانیه کار را تمام کرد و گفت که کلمه‌ی to شانزده بار در سه پاراگراف اول تکرار شده است.

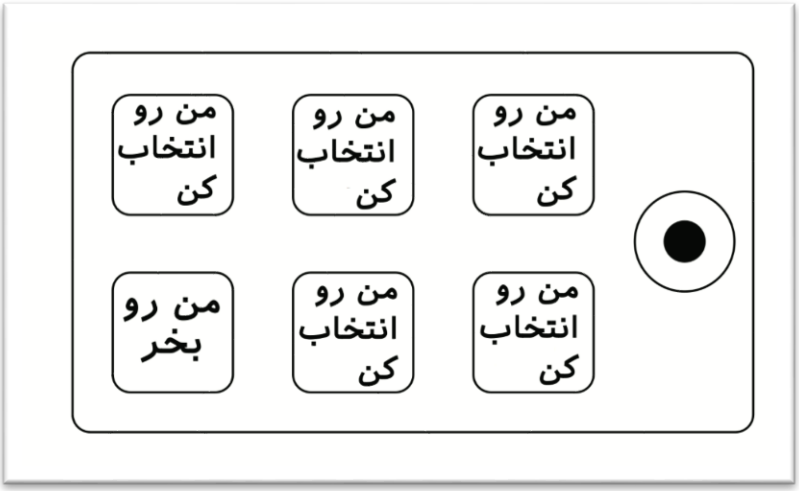
این اصل که کامپیوتر در کارهایی که برای انسان سخت است، سریع و دقیق عمل می‌کند، یک نکته را به ما یادآوری می‌کند: چرا زبانشان را یاد نگیریم؟ ما نیاز داریم که مهارت حرف زدن به زبان کامپیوتر را یاد بگیریم. وقتی که این زبان جدید را یاد گرفتید، می‌توانید کارهای حوصله‌سربر را به شریک خودتان بسپارید و وقت بیشتری را برای کارهایی که مناسب شماست بخرید. شما در این رابطه و شراکت با کامپیوتر چه در چنته دارید؟ خلاقیت، شهود و قوه‌ی خلق و اختراع را از جیبتان خارج کنید.

## خلاقیت و انگیزه

برنامه‌نویسی حرفه‌ای یک شغل ارزشمند است که نه‌تنها انجامش به شما حس خوبی را می‌دهد که از نظر اقتصادی هم کار مناسبی به حساب می‌آید. البته این کتاب برای یادگیری برنامه‌نویسی حرفه‌ای نگاشته نشده است، ولی به هر حال بایستی از جایی شروع کنید. ساخت برنامه‌های هوشمندانه، زیبا و مفید برای دیگران، یک فعالیت خلاقانه محسوب می‌شود. دستیار شخصی دیجیتال (PDA) یا کامپیوترِ شما مملو از برنامه‌های‌ست که برنامه‌نویسان مختلفی نوشته‌اند. هر کدام هم توجه شما به خودشان را می‌طلبند. تلاش این برنامه‌ها برطرف کردن نیازهای شماست. در این بین اگر تجربه‌ی

خوبی از روند اجرای برنامه برای شما به ارمغان بیاورند و توجه شما را جلب کنند، صاف به هدف زده‌اند. به عبارتی تلاش می‌کنند که تجربه‌ی خوبی برای شما بسازند. در برخی موارد با توجه به مجموع تجربه و رضایت شما، جلب نظرتان برای برنامه‌نویس خیر مادی خواهد داشت.

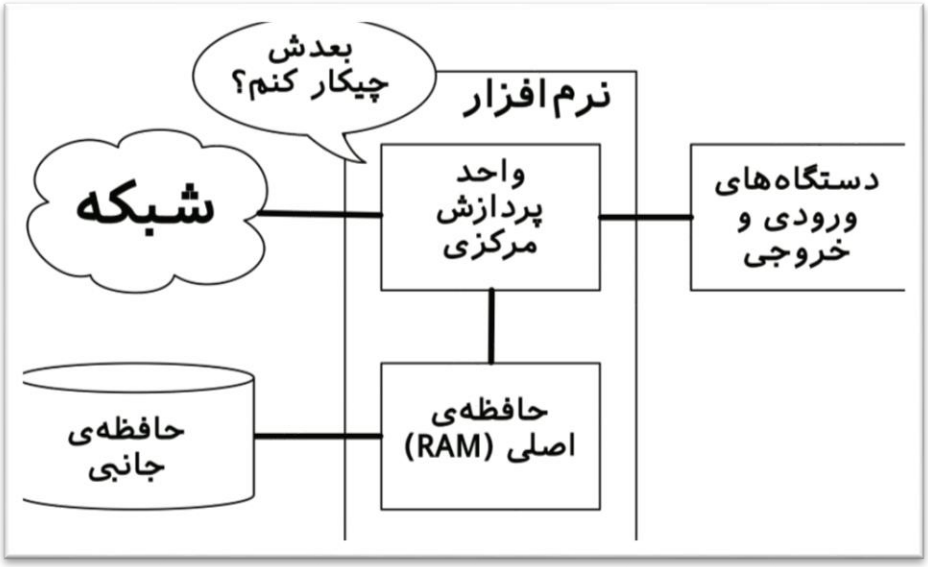
اگر به برنامه‌ها به عنوان خروجی خلاقیت یک دسته برنامه‌نویس نگاه کنیم، تصویر زیر قیاس خوبی‌ست برای نشان دادن آن چیزی که بر صفحه‌ی موبایل شما نمایش داده می‌شود. تصویری که در پنج حالت می‌گوید «مرا انتخاب کن» و در یک حالت می‌گوید «مرا بخر» که خب با خرید شما، خیر مادی به برنامه‌نویسان خواهد رسید.




برای من و شما اکنون پول در آوردن یا خوشحال کردن کاربر نهایی مطرح نیست بلکه می‌خواهیم سر و کله زدن با اطلاعاتی که در زندگی روزمره با آن‌ها سر و کار داریم را به نحوی بهتر و کاراتر یاد بگیریم. اکنون که در ابتدای راهیم، شما هم برنامه‌نویس‌اید و هم کاربر نهایی برنامه‌تان. هرچقدر مهارت بیشتری در زمینه‌ی برنامه‌نویسی کسب کنید نوشتن برنامه برای‌تان خلاقانه‌تر خواهد شد و کم‌کم به این نتیجه می‌رسید که بد نیست برنامه نوشتن برای دیگران را آغاز کنید.

## معماریِ سخت‌افزار کامپیوتر

قبل از اینکه زبان سخن گفتن با کامپیوتر برای اجرای دستورالعمل‌هایی را، که از او می‌خواهیم، یاد بگیریم، بد نیست کمی در خصوص چگونگی ساخت و کارکرد خود کامپیوتربیاموزیم. اگر قرار باشد که کامپیوتر یا موبایل خودتان را به قطعه‌های کوچکی تقسیم کنید، وقتی که درِ کیس را باز می‌کنید با این قطعه‌های اصلی روبرو خواهید شد:




معنای سطح بالای این قسمت‌ها به شرح زیر است:

- واحد پردازش مرکزی (CPU) قسمتی از کامپیوتر است که به صورت وسواس‌گونه و پشت سر هم می‌پرسد «بعدش چیه؟ حالا چیکار کنم؟». اگر پردازنده‌ی سیستم شما سرعتش ۳/۰ گیگا هرتز است، این بدان معناست که CPU سیستم شما این سوال را سه میلیارد بار در ثانیه می‌پرسد. به عبارتی بهتر است یاد بگیرید که خیلی سریع حرف بزنید تا به گرد پای پردازنده‌تان برسید.

- حافظه‌ی اصلی یا RAM اطلاعات را برای استفاده‌ی CPU که خیلی هم عجله دارد نگه‌داری و مهیا می‌کند. حافظه‌ی اصلی هم سرعتی نزدیک به پردازنده دارد؛ ولی مشکلش این است که تا کامپیوتر را خاموش می‌کنید، اطلاعات موجود در آن هم دود می‌شوند و به هوا می‌روند.

- حافظه‌ی جانبی نیز به مانند حافظه‌ی اصلی، اطلاعات را در خود ذخیره می‌کند ولی با این تفاوت که خیلی کُندتر است. مزیتش چیست؟ حتی با خاموش کردن کامپیوتر، اطلاعات روی حافظه‌ی جانبی باقی می‌مانند. هارددیسک‌ها، حافظه‌های فلش[I] از انواع حافظه‌های جانبی به حساب می‌آیند.

- دستگاه‌های ورودی و خروجی شامل صفحه‌ی نمایش، چاپگر، موشواره[II]، کیبورد، میکروفن، بلندگو، تاچ‌پد و غیره می‌شوند. دستگاه‌هایی که به نحوی تعامل شما با کامپیوتر را برقرار می‌کنند، دستگاه‌های ورودی یا خروجی‌اند.

- این روزها، اکثر کامپیوترها یک اتصال به شبکه را برای دریافت اطلاعات از طریق شبکه با خود دارند. ما می‌توانیم به شبکه به عنوان مکانی نگاه کنیم که داده‌ها را از طریق آن ذخیره و دریافت می‌کنیم. البته سرعت نقل و انتقال داده‌ها کم است و همیشه هم در دسترس نیستند. به عبارتی شبکه، حافظه‌ای جانبی‌ست که کُند است و نمی‌شود با اطمینان بر آن تکیه زد.

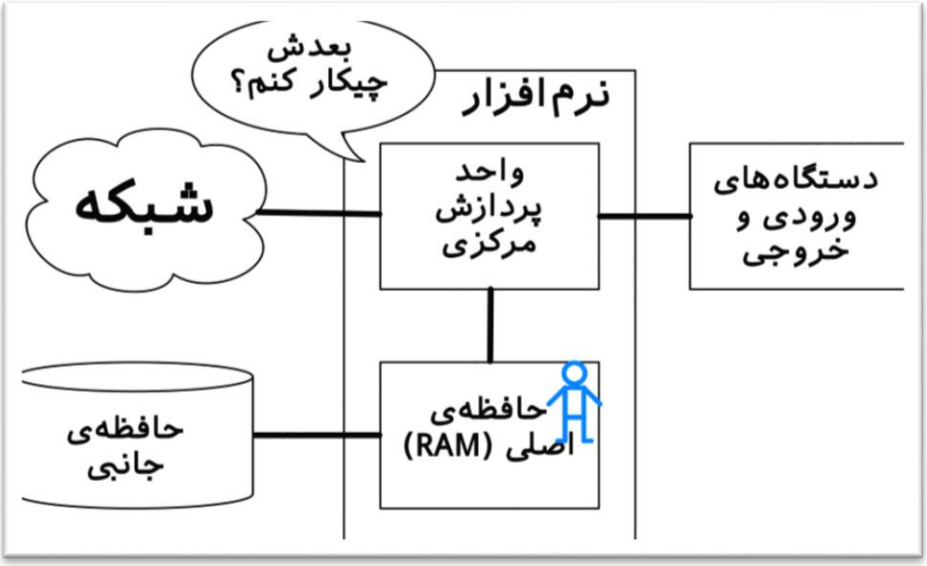
جزئیات ساخت این قطعات را دیگر به متخصصان این امر بسپارید. مهم بود که در خصوص اصطلاحات آن کمی بدانید تا بهتر بتوانید با سیستم خود ارتباط برقرار کنید.

---

[I] فلش‌مموری

[II] ماوس

به عنوان یک برنامه‌نویس، کار شما استفاده از این قطعات در یک هارمونی برای حل مسائلی‌ست که نیازی را برطرف می‌کنند. مثلا پاسخ به یک سوال یا آنالیز داده‌ها برای رسیدن به جواب. اکثر کار شما در حرف زدن با CPU خلاصه می‌شود. اینکه به




اون بگویید که خب قدم بعدی چیست و چه کاری باید انجام دهد. گاهی به CPU می‌گویید که با حافظه‌ی اصلی یا حافظه‌ی جانبی یا شبکه یا دستگاه‌های ورودی و خروجی وارد بحث و گفتگو شود.

شما بایستی همان شخصی باشید که به سوال «بعدش چیه؟» CPU پاسخ می‌دهید. ولی خب اگر قرار باشد تبدیل به یک آدمک ۵ میلی‌متری شده و در کامپیوتر برای تعامل با آن جاسازی شوید تازه نوبت به آن می‌رسد که هر ثانیه سه میلیارد بار سوال بپرسید. برای رفع این مشکلات عدیده، ما یاد می‌گیریم که چطور دستورالعمل بنویسیم. این دستورالعمل‌های ذخیره شده را برنامه، و فعالیتی که منجر به نوشتن این راهنماها و تصحیح‌شان برای اجرای درست می‌شود را برنامه‌نویسی می‌نامیم.

## درک برنامه‌نویسی

در ادامه‌ی این کتاب ما تلاش می‌کنیم تا شما را تبدیل به شخصی کنیم که در هنر برنامه‌نویسی مهارت لازم را کسب کرده است. در انتها شما یک برنامه‌نویس خواهید بود – حالا نه برنامه‌نویس حرفه‌ای ولی حداقل مهارت بررسی یک مساله با آنالیز داده‌ها و اطلاعات و توسعه‌ی برنامه‌ای برای حل آن را پیدا خواهید.

در کل شما به دو مهارت برای تبدیل شدن به یک برنامه‌نویس نیاز خواهید داشت:

I. اول، بایستی یک زبان برنامه‌نویسی را یاد بگیرید (در این کتاب: پایتون) – باید که با فرهنگ لغات و اصول نگارش آن آشنا شوید. بایستی بتوانید که کلمه‌ها را درست بنویسید و جمله‌های خوبی با آن‌ها به زبان جدید بسازید.

II. دوم، باید یک داستان سر هم کنید. داستان‌سرایی چیست؟ ترکیب کلمه‌ها و جمله‌ها و پرده برداشتن از ایده‌ای برای خواننده. هنر و مهارت نقش اساسی را در ساخت یک داستان ایفا می‌کنند. مهارتِ داستان‌نویسی با دست به قلم شدن و دریافت بازخورد از خواننده‌ها یا حتی خودتان با مرور دوباره میسر می‌شود. در برنامه‌نویسی، برنامه‌ی ما همان داستان است و مساله‌ای که قرار است حل کنیم، ایده‌ی این داستان.

زمانی‌که یک زبان برنامه‌نویسی مثل پایتون را یاد گرفتید، فرا گرفتن دومین و چندین زبان برنامه‌نویسی مثل جاوااسکریپت یا C++ بسیار ساده‌تر می‌شود. درست است که در زبان جدید بایستی که کلمه‌ها و دستور زبان آن را مجددا یاد بگیرید، ولی مهارت اصلی حل مساله در بین همه‌ی زبان‌های برنامه‌نویسی یکسان است.

لغات و جملات پایتون را خیلی سریع یاد خواهید گرفت ولی برای نوشتن برنامه‌های منسجم که مسائل جدید را حل کنند به زمان بیشتری احتیاج خواهید داشت. در این

کتاب ما سعی می‌کنیم که برنامه‌نویسی را شبیه به نویسندگی به شما یاد دهیم. ابتدا شروع به خواندن و توضیح برنامه‌ها می‌کنیم، سپس برنامه‌های ساده‌ای خواهیم نوشت و بعد از آن به پیچیدگی برنامه‌هایی که می‌نویسیم اضافه می‌کنیم. در نهایت به جایی می‌رسید که سبک خودتان را در برنامه‌نویسی و دیدن الگوها پیدا خواهید کرد. مسائل را به روش خودتان می‌بینید و برنامه‌ای برای حل آن به شیوه‌ی ویژه‌ی خودتان خواهید نوشت؛ و آن موقع است که به خودتان می‌آیید و می‌بینید که چقدر نوشتن برنامه‌ها برای‌تان لذتبخش است و روند خلاقانه‌ای دارد.

ما با لغات و ساختار برنامه‌های پایتون شروع می‌کنیم. ممکن است که در ابتدا کمی سادگیِ مسائل، حوصله‌سربر باشد و خاطره‌ی سال‌های اول ابتدایی را برای‌تان زنده کند، ولی بایستی که صبور باشید و آرام آرام آن‌ها را فرا بگیرید.

## کلمات و جمله‌ها

برخلاف زبان انسان، تعداد لغات پایتون بسیار محدودند. ما به این «لغات» کلمه‌های «رزرو شده» می‌گوییم. این کلمات برای پایتون معنای ویژه‌ای دارند. زمانی‌که پایتون به هر کدام از کلمه‌های رزرو شده برمی‌خورد، یک و فقط یک معنی از آن‌ها استنتاج می‌کند. شما در حین برنامه‌نویسی لغاتی به اسم «متغیر» به برنامه اضافه خواهید کرد. متغیرها، لغاتی در برنامه‌اند که ساخت و پرداخته ذهن شما برای منظور خاصی به شمار می‌روند. هر کدام از آن‌ها نشان‌دهنده‌ی یک متغیر است و معنای ویژه‌ای برای شما خواهد داشت. ولی برای پایتون فقط کلمات رزرو شده‌اند که معنای خاص دارند و بایستی بدانید که شما از کلمات رزرو شده نمی‌توانید به عنوان متغیر استفاده کنید.

سگی را در نظر بگیرید که تعداد خاصی از کلمات را آموخته است. مثلا «بشین»، «همونجا بمون» یا «بیارش برام» کلماتی‌ست که سگِ آموزش‌دیده‌ی شما می‌فهمد. وقتی که از کلماتی غیر از آن‌ها استفاده می‌کنید و سگ‌تان را مخاطب قرار می‌دهید، سگ با نگاهی که «این بابا چی می‌گه» به شما خیره خواهد شد و عکس‌العملی نشان نخواهد داد. چرا که از کلمات رزرو شده برای سگ استفاده نکرده‌اید. مثلا فرض کنید که به سگ بگویید، «به چارلی گفتم که کمی بدو چون ورزش و دوندگی برای سلامتیت خوبه». سگ

چه چیزی می‌شنود؟ «فلان فلان فلان فلان فلان بدو فلان فلان فلان فلان فلان فلان فلان». در بین کلمات سگ فقط «بدو» را می‌شناسد و بس، چرا که آن کلمه، در زبانِ سگی رزرو شده است. خیلی‌ها می‌گویند که این موضوع برای گربه‌ها صادق نیست. البته به ما هم مربوط نیست چون سگ کار ما را برای توضیح مفهوم راه انداخت.

کلمه‌های رزرو شده که توسط آن‌ها با پایتون ارتباط برقرار کرده و حرف می‌زنیم به شرح زیر است:

```
and       del       global      not       with
as        elif      if          or        yield
assert    else      import      pass
break     except    in          raise
class     finally   is          return
continue  for       lambda      try
def       from      nonlocal    while
```

تمامش همین‌هایند و بر خلاف سگ که نیاز به آموزش دارد، پایتون همه‌ی این‌ها را خودش بلد است. وقتی به او می‌گویید try آن کاری که مرتبط با این فرمان است را بدون شک انجام می‌دهد و هیچوقت اشتباه نمی‌کند.

ما این کلمه‌ها را به مرور و در زمان خودشان یاد خواهیم گرفت. تمرکز الان ما در حرف زدن با پایتون به مانند همان حرف زدن انسان با سگ است. ولی می‌دانستید که پایتون می‌تواند با شما حرف بزند؟ جالبی کار این است که برای به حرف آوردن پایتون کافی‌ست که پیام را در قالب خاصی به او بدهید و پایتون شروع به حرف زدن خواهد کرد. بهتر است با نمونه‌ی «سلام جهان» شروع کنیم:

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

کلمات بالا را روی کاغذ یا در یک ویرایشگر متن بنویسید. تبریک می‌گویم، اولین برنامه به زبان پایتون با محتویات درست و معنادار برای او را نوشتید. جمله‌ی ما با «تابع» print برای چاپ یک رشته از متن که داخل نقل‌قول قرار گرفته شروع می‌شود. علامت نقل‌قول در لاتین به صورت تکی و دوگانه نوشته می‌شود. وقتی در این کتاب به

نقل‌قول تکی اشاره می‌کنیم، منظورمان این علامت «'» و وقتی به نقل‌قول دوگانه اشاره می‌کنیم منظورمان این نشان «"» است. در جمله‌ی بالا، رشته‌ی ما یعنی عبارت Hello world در داخل نقل‌قول تکی قرار گرفته است.

## صحبت با پایتون

حالا که چند کلمه‌ی پایتونی یاد گرفتیم بایستی یاد بگیریم که چطور با استفاده از کلمات و جمله‌ها شروع به صحبت کردن با پایتون کنیم. پس بد نیست که توانایی خودمان در زبان جدید را به بوته‌ی آزمایش بگذاریم.

قبل از اینکه بتوانید با او صحبت کنید، بهتر است که به خانه‌تان دعوتش کنید. پس لازم است که مراحل نصب آن روی کامپیوترتان را با هم مرور کنیم.

ما در این قسمت در خصوص نصب پایتون و راه‌اندازی آن روی ویندوز صحبت خواهیم کرد. کاربران لینوکس حتما می‌دانند که چطور پایتون رو از طریق مخازن نرم‌افزاری خود به سادگی نصب کنند.

اول از همه لازم است که پایتون را دانلود کنید. برای اینکار به سایت https://python.org مراجعه کنید. دو نسخه‌ی ۶۴ و ۳۲ بیتی برای ویندوز در دسترس است. با توجه به معماری سیستم خود، یکی از آن‌ها را دریافت کنید.

ما در این دوره از پایتون ۳ استفاده می‌کنیم پس مطمئن باشید که نسخه‌ی مرتبط را دانلود می‌کنید. در زمان نوشتن این کتاب نسخه‌ی آن ۳/۶/۱ است.

وقتی نصاب ظاهر شد گزینه‌ی Add Python to PATH را تیک بزنید و نصب را ادامه دهید. حالا کافی‌ست که از منوی استارت ویندوز CMD را اجرا کنید. CMD که باز شد، عبارت python را بنویسید و وارد شوید:

```
C:\Users\eman> python


Python 3.5.1 (v3.5.1:37a07cee5969, Dec  6 2015, 01:54:25)
[MSC v.1900 64 bibbt (AMD64)] on win32
```

```
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

اعلانِ مفسر پایتون که با نماد <<< مشخص می‌شود، نشان دهنده‌ی آمادگی پایتون برای پرسیدن سوال از شماست. همان سوال معروف «حالا که چی؟» به بیان ساده‌تر پایتون آماده‌ی صحبت کردن با شماست. تمام آن چیزی که اکنون نیاز دارید، صحبت کردن به زبان پایتونی‌ست.

فرض کنیم که شما ساده‌ترین کلمه‌ها و جملات پایتون را بلد نیستید. شاید شبیه به فضانوردانی که روی یک سیاره‌ی دیگر فرود آمده‌اند و می‌خواهند با موجودات آنجا ارتباط برقرار کنند، با مشکل برخورد کنید:

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

اوضاع خوب پیش نمی‌رود. احتمال دارد اگر همین‌گونه به کار ادامه دهید، ساکنان آن سیاره اول خنجری به شما فرو کرده و سپس منقل را با ذغال، گرم و در نهایت خوب روی آن سرختان کنند و شما را میل فرمایند.

خوشبختانه شما یک نسخه از این کتابی که در دست دارید را با خودتان آورده‌اید و حالا سعی می‌کنید که با زبان خودشان با آن‌ها صحبت کنید:

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

اکنون کنترل اوضاع دست شماست. بد نیست کمی بیشتر با آن‌ها صحبت کنید:

```
>>> 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: Missing parentheses in call to 'print'
>>>
```

مکالمه خوب پیش می‌رفت تا اینکه یک اشتباه خیلی کوچک کردید و پایتون باز نیزه‌اش را به سمتتان گرفت.

درست است که پایتون بسیار پیچیده و قدرتمند است و در انتخاب متن مناسب برای پردازش سختگیری می‌کند ولی زمانی‌که با او صحبت می‌کنید این را در نظر داشته باشید که پایتون باهوش نیست. کوچکترین خطای شما، برای او مشکل در تفسیر حرف شما را باعث می‌شود. در مثال بالا، ما در اصل یک مکالمه با خودمان با استفاده از سینتکس/متن قابل فهم برای پایتون داشتیم.

وقتی شما از برنامه‌ای که توسط شخص دیگری نوشته شده استفاده می‌کنید، در اصل مکالمهْ بین شما و آن برنامه‌نویسان دیگر است و در این بین پایتون یک رابط به حساب می‌آید. پایتون وسیله‌ای‌ست که به سازندگان برنامه‌ها اجازه می‌دهد که نشان دهند این مکالمه بایستی چطور باشد و به چه سمتی پیش برود؛ فقط چند فصل باقیمانده تا شما هم به دسته‌ی برنامه‌نویسانی که زبان پایتونی می‌فهمند ملحق شوید.

قبل از اینکه با مفسر پایتون خداحافظی کنید، باید روش درست خداحافظی با او را یاد بگیرید. به هر حال ساکنان سیاره‌ی پایتون معنای «خداحافظ» را نمی‌فهمند:

```
>>> 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: invalid syntax
>>> quit()
```

اگر دقت کرده باشید می‌بینید که خطاهای متفاوتی را داده است. در تلاش اول NameError داد و در تلاش دوم خطای «متن غیرمعتبر». در تلاش دوم، if یک کلمه‌ی رزرو شده است و زمانی‌که پایتون این کلمه را می‌بیند فکر می‌کند که شما می‌خواهید چیزی در خصوص آن به پایتون بگویید. ولی وقتی با ادامه‌ی جمله مواجه می‌شود و می‌بینید ساختار درستی ندارد، خطا را صادر می‌کند.

راه درست خارج شدن از پایتون استفاده از ()quit بعد از نماد <<< است. به هر حال این کتاب کمک می‌کند که بهتر بفهمید چطور می‌شود با زبان پایتون با او مکالمه کرد و به سادگی خارج شد.

## اصطلاحات: مفسر و کامپایلر

پایتون یک زبان سطح بالا به حساب می‌آید. هدف پایتون، راحتی و قابل فهم بودن آن برای انسان جهت خواندن و نوشتنِ متنِ برنامه است، به صورتی که کامپیوتر هم به واسطه‌ی پایتون بتواند همان فایل را بخواند و پردازش کند. زبان‌های برنامه‌نویسی سطح بالا بسیار زیادند. از معروف‌ترین آن‌ها می‌توان به جاوا، C++، Ruby، PHP، Basic، Prel و جاوااسکریپت اشاره کرد. پردازنده‌ی کامپیوتر (CPU) نمی‌تواند که زبان هیچکدام از این برنامه‌ها را بفهمد.

پردازنده زبانی را می‌فهمد که به آن زبانِ ماشین می‌گوییم. زبان ماشین بسیار ساده است. اگر بخواهیم رک و رو راست باشیم نوشتن به زبان ماشین بسیار حوصله‌سربر خواهد بود چرا که فقط از صفر و یک تشکیل شده است:

```
00101000111010010010101000000011111
11100110000011101010010101101101
...
```

وقتی به ظاهر قضیه نگاه می‌کنید، می‌بینید که زبان ماشین بسیار ساده و بی‌آلایش می‌باشد چرا که کُل‌ش تشکیل شده از دو حرف است ولی متن آن از متن زبانی مثل پایتون بسیار پیچیده‌تر است. برای همین تعداد قلیلی از برنامه‌نویسان تا به حال به این زبان برنامه نوشته‌اند. در عوض مترجم‌های مختلفی برای نوشتن برنامه وجود دارد. این مترجم‌ها اجازه‌ی نوشتن در برنامه‌های سطح بالایی مثل پایتون یا جاوا اسکریپت را به برنامه‌نویس می‌دهند. در نهایت کدِ قابل فهم این برنامه‌ها را گرفته و به زبان ماشین تبدیلش می‌کنند. اکنون این کد توسط CPU قابل پردازش است.

از آنجایی که زبان ماشین به سخت‌افزار کامپیوتر گره خورده است، این زبان نمی‌تواند که بین سخت‌افزارهای مختلف جابجا شود. به عبارتی قابلیت حمل از سخت‌افزاری به سخت‌افزار دیگر را ندارد. برنامه‌هایی که توسط زبان‌های سطح بالا نوشته شده‌اند می‌توانند که بین کامپیوترهای مختلف با استفاده از مفسرهای مختلف جابجا شوند. به عبارتی شما می‌توانید توسط مفسر یا کامپایلر برنامه‌ی نوشته شده را برای سخت‌افزار جدید قابل فهم کنید.

مترجم‌های زبان‌های برنامه‌نویسی در دو گروه اصلی خلاصه می‌شوند: ۱) مفسرها و ۲) کامپایلرها.

یک مفسر (interpreter) کد منبع را – همانگونه که برنامه توسط برنامه‌نویس نوشته شده – می‌خواند، تجزیه می‌کند و دستورالعمل (کد برنامه) را روی هوا تفسیر می‌کند. زمانی‌که ما از حالت تعاملی مفسر پایتون استفاده می‌کنیم، یک خط یا جمله‌ی

پایتونی می‌نویسیم و پایتون در جا آن را خوانده، پردازش کرده و از ما طلب خط بعدی را می‌کند. البته این روند به میزان پردازش لازم، برای آن خط کد، نیز بستگی دارد.

برخی خطوط در پایتون، به او می‌گوید که «بایستی مقادیری را برای استفاده‌ی بعدی به خاطر بسپاری». برای این کار ما نامی برای آن مقدار برمی‌گزینیم که بتوانیم در آینده نیز آن نام را به خاطر آورده و مقدار متغیر را از آن خارج کنیم. اسم برچسب‌هایی که به این مقادیر می‌زنیم variable یا متغیر است. متغیرها، مقادیر را در خودشان نگه می‌دارند. به مثال زیر دقت کنید که متغیر x مقدار 6 را در خود دارد و متغیر y مقدار x * 7 را:

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

در این مثال، ما از پایتون خواستیم که مقدار شش را به خاطر داشته باشد و برچسب x روی آن بزند. به این صورت ما در آینده می‌توانیم مقدار را توسط این برچسب احضار کنیم. ما با استفاده از print مطمئن می‌شویم که پایتون با فراخوانی برچسب، مقدار آن را نمایش می‌دهد. سپس از پایتون می‌خواهیم که مقدار برچسب خورده با x را بگیرد در هفت ضرب کند و در نهایت مقدار محاسبه شده را در y قرار دهد. در نهایت از پایتون می‌خواهیم که مقدار فعلی y را چاپ کند.

در اینجا، دستورات و گزاره‌ها را یک به یک به پایتون دادیم. با این حال پایتون به آن‌ها به عنوان یک سلسله گزاره نگاه می‌کند که گزاره‌های بعدی می‌توانند گزاره‌های قبلی را به خاطر داشته باشند و داده‌هایی که به آن‌ها مربوط است را درخواست نمایند. اکنون ما اولین پاراگراف چهار خطی‌مان را در چهار جمله منطقی و قابل فهم نوشتیم. شما هم دست به کار شوید و همین‌ها را وارد مفسر پایتون کنید. حتی اگر احساس

می‌کنید که این گزاره‌ها آنقدر ساده‌اند که نیازی به وارد کردنشان نیست، باز هم این لطف را در حق خودتان کنید و گزاره‌ها را خودتان بنویسید.

طبیعت یک مفسر این است که بتواند یک گفتگوی تعاملی مانند چیزی که در مثال بالا دیدید را داشته باشد. برای کامپایلر اوضاع فرق می‌کند. شما بایستی کل برنامه را در قالب فایل به او تحویل بدهید. سپس کامپایلر شروع به خواندن و پردازش کدِ منبع سطح بالا کرده و به زبان ماشین تبدیلش می‌کند. نتیجه یک فایلِ قابل اجرا خواهد بود. به عبارتی این فایل قابلیت اجرا توسط سخت‌افزار شما را خواهد داشت.

اگر از سیستم ویندوزی استفاده می‌کنید، این برنامه‌ها که به زبانِ ماشین‌اند و قابلیت اجرا روی کامپیوتر را دارند، پسوندی مثل «exe» یا «dll» را خواهند داشت که اولی به معنای «قابل اجرا بودن» و دومی «کتابخانه پیوند پویا» است. در لینوکس و مکینتاش پسوندی که نشان دهنده‌ی فایل قابل اجرا باشد وجود ندارد.

اگر شما یک فایل اجرایی را در ویرایشگر متن خود – مثلا Notepad – باز کنید، چیزی که نمایش داده خواهد شد، یک متن کاملا ناخوانا و اجق وجق خواهد بود:

```
^?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
....
```

مشخص است که خواندن و نوشتن به زبان ماشین، چندان ساده نیست و اینجاست که مفسرها و کامپایلرهایی که به ما اجازه‌ی نوشتن کد در زبان سطح بالا را می‌دهند، وارد گود می‌شوند.

وسط بحث ما در خصوص کامپایلر و مفسر شاید این فکر به ذهن شما خطور کند که خب، خودِ مفسر پایتون چطور نوشته شده است. آیا پایتون در یک زبان کامپایل‌شده نوشته شده؟ وقتی ما در خط فرمان می‌نویسیم python دقیقا چه اتفاقی می‌افتد؟

پایتون با زبان سطح بالایی به اسم C نوشته شده است. می‌توانید از طریق سایت www.python.org کد منبع آن را مشاهده کنید. در اصل پایتون خودش یک برنامه‌ای‌ست که به صورت زبان ماشین کامپایل شده است. زمانی‌که پایتون را روی سیستم خود نصب می‌کنید شما یک رونوشت از برنامه‌ی کامپایل شده و به زبان ماشین درآمده‌ی پایتون را روی سیستم خود قرار می‌دهید. در ویندوز، کدِ ماشین پایتون که قابلیت اجرا داشته باشد احتمالا در مسیر زیر قرار گرفته است:

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

این اطلاعات بیشتر از آن چیزی بود که برای برنامه‌نویسی نیاز داشتید. در اصل دانستن یا ندانستنشان زیاد تاثیری در روند برنامه‌نویسی شما ندارد ولی گاهی دانستن این موارد به شما کمک می‌کند تا درک بهتری از روند کار داشته باشید.

## نوشتن یک برنامه

کار کردن با حالت تعاملی راه خوبی برای بهره بردن و آزمودن ویژگی‌های پایتون است ولی برای حل مسائل پیچیده، این روش توصیه نمی‌شود.

فرض کنید که می‌خواهید تمام دستورات لازم برای نوشتن یک برنامه را داخل یک فایل بگذارید. واضح است که به یک ویرایشگر متن – چیزی شبیه به نوت‌پد در ویندوز – نیاز خواهید داشت. البته Notepad ویندوز، گزینه‌ی جالبی نیست. ما در این کتاب دو گزینه را برای این منظور به شما معرفی می‌کنیم. اول اتم است. سری به سایت https://atom.io بزنید و از طریق آن، ویرایشگر مخصوص برنامه‌نویسی اتم را دریافت کنید. گزینه‌ی دوم Notepadd++ است. برای دریافت آن نیز می‌توانید به سایت https://notepad-plus-plus.org مراجعه کنید.

به فایلی که بعد از نوشتن دستورالعمل‌ها ذخیره می‌کنیم، اسکریپت می‌گوییم. نام اسکریپت‌های پایتون به py. ختم می‌شود. به نمونه‌ی زیر توجه کنید:

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

کلمه‌ی csev$ اعلانِ سیستم‌عامل من است. عبارت "cat hello.py" محتویات فایل hello.py را نمایش می‌دهد که فعلا تنها یک خط است.

در اینجا ما مفسر اعلان را احضار کرده و به او می‌گوییم که کد منبع را بخواند. با آوردن اسم فایل پایتون می‌فهمد که کد را بایستی از کجا بخواند. به این صورت پایتون کد را از فایل مورد نظر خوانده و پردازش می‌کند، بدون اینکه نیازی به حالت تعاملی باشد.

ولی فایل ما در انتهای خود quit() را برای خارج شدن نداشت. چرا؟ زمانی‌که پایتون به انتها فایل کد می‌رسد، خودش می‌فهمد که وقت خداحافظی‌ست و بایستی خارج شود.

## برنامه چیست

ابتدایی‌ترین تعریفی که برای یک برنامه می‌توان داشت عبارت است از: مجموعه دستوراتی که برای انجام کار خاصی نوشته شده است. حتی اسکریپت ساده‌ی hello.py در پایه‌ای‌ترین حالتْ یک برنامه به حساب می‌آید. درست است که کار مفید و خاصی انجام نمی‌دهد، ولی چاپ عبارت «سلام دنیا» یک وظیفه است که برنامه‌نویس به آن واگذار کرده است.

برای درک آسان مفهوم یک برنامه، مسأله‌ای را در نظر بگیرید که برنامه برای حل آن ساخته شده است. برنامه همان چیزی‌ست که وظیفه‌ی حل مسأله را بر عهده دارد.

فرض کنید که در حال محاسبه برای تحقیق روی پست‌های فیس‌بوک‌اید. مثلا می‌خواهید کلمه‌ای که بیشترین استفاده را در دسته‌ای از پست‌ها دارد، بیرون بکشید.

می‌توانید استریم (جریان جاری روی صفحه‌ی فیس‌بوک) را چاپ کنید، سپس کلمه به کلمه بگردید تا ببینید پراستفاده‌ترین کلمه کدام است. این روش هم زمان زیادی می‌برد، هم احتمال خطا بالاست. برای این کار می‌توانید یک برنامه‌ی پایتون بنویسید تا انجام سریع و دقیق این وظیفه را بر عهده بگیرد و بقیه وقت خود را به خوشگذرانی طی کنید.

به عنوان مثال نگاهی به نوشته‌ی زیر بیندازید. به نظرتان کدام کلمه بیشتر از بقیه تکرار شده؟ چند بار؟

```
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 همین کار را روی متنی با میلیون‌ها خط انجام دهید. در حقیقت اگر شروع به یادگیری پایتون کنید و برنامه‌ای برای این کار بنویسید، سریع‌تر به جواب خواهید رسید.

حتی می‌توانید از برنامه‌ای که من برای این منظور نوشتم استفاده کنید و وقت بیشتری برای خودتان بخرید:

```
name = input('Enter file:')
handle = open(name, 'r')
text = handle.read()
words = text.split()
counts = dict()
for word in words:
    counts[word] = counts.get(word,0) + 1


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


print(bigword, bigcount)


# Code: http://www.pythonlearn.com/code3/words.py
# Or select Download from this trinket's left-hand menu
```

برای استفاده از این برنامه حتی لازم نیست که برنامه‌نویسی را با پایتون بلد باشید. برای درک تکنیک‌هایی که در این برنامه به کار بردیم، تا فصل ده این کتاب همراه ما باشید. فعلا به عنوان کاربر نهایی، از برنامه استفاده کنید، سرعت و قدرت انجام وظیفه‌ای حوصله‌سربر را نظاره کنید و زمان و انرژی زیادی برای خود بخرید. می‌توانید که کدهای بالا را دستی وارد یک ویرایشگر کنید و یا اینکه به آدرس زیر رفته و آن‌ها را دانلود کنید:

```
http://www.pythonlearn.com/code3
```

اگر از این برنامه استفاده کنید، در اصل شما به عنوان کاربر نهایی و من به عنوان برنامه‌نویس از طریق واسطی به اسم پایتون با هم ارتباط برقرار کرده‌ایم. پایتون راهی برای تبادل مجموعه دستورالعمل‌های مفید (مانند برنامه‌ها) با هر کسی است که آن را

روی سیستم خود نصب کرده است. ما با پایتون حرف نمی‌زنیم؛ در اصل در حال حرف زدن با یکدیگر از طریق پایتون‌ایم.

## اجزاء یک برنامه

در چند فصل بعدی، در خصوص لغات، ساختار جمله‌ها، ساختار پاراگراف، و ساختار داستانی که قرار است با پایتون بگویید صحبت خواهیم کرد. در خصوص قابلیت‌های قدرتمند پایتون به صحبت خواهیم نشست و به شما روش ترکیب کردن این ویژگی‌ها برای مهار قدرت پایتون و ساخت برنامه‌های کاربردی را یاد خواهیم داد.

ما از چند الگوی مفهومیِ سطح پایین برای ساخت برنامه‌ها استفاده می‌کنیم. این زیرساخت، فقط مربوط به برنامه‌های پایتونی نیست؛ بلکه جزئی از تمامی زبان‌های برنامه‌نویسی به حساب می‌آید؛ از زبان ماشین گرفته تا زبان‌های سطح بالا شامل این زیرساخت می‌شوند.

### ورودی

دریافت داده از «دنیای بیرون». شاید خواندن داده از روی یک فایل یا از طریق یک سنسور مثل GPS یا میکروفن. در برنامه‌های ابتداییِ ما، این داده‌ها مستقیما از کیبورد بوسیله‌ی تایپ کردن می‌آید.

### خروجی

نمایش نتیجه‌ی برنامه روی صفحه‌ی نمایش یا قرار دادنش در یک فایل یا شاید نوشتن و ارسال آن به دستگاهی مثل اسپیکر برای پخش موزیک یا خواندن یک متن.

### اجرای ترتیبی

دستوراتی که در یک اسکریپت نوشته شده اگر به ترتیب و یکی پس از دیگری اجرا شود، به آن اجرای ترتیبی می‌گوییم. به عبارتی همانطور که معمول است برنامه از ابتدای اسکریپت شروع به خواندن می‌کند و به انتها می‌رسد.

### اجرای شرطی

شاید گاهی اجرای دستور یا خطی از کد، مشروط بر محقق شدن شرطی باشد. فکر می‌کنم آنقدر واضح است که نیاز به توضیح نباشد ولی بگذارید که یک مثال کوتاه بزنم. مثلا شرط می‌کنید که اگر برنامه ورودی Dr-Chuck را از کیبورد گرفت روی صفحه‌ی نمایش عبارت Hello Master را چاپ و اگر ورودی غیر از Dr-Chuck را گرفت، عبارت Access Denied را چاپ کند. ما در برنامه هم خط مربوط به چاپ Hello Master و هم خط مربوط به چاپ Access Denied را داریم که بنا به ورودی یکی از آن‌ها چاپ می‌شود. به عبارت ساده در برنامه شرطی وجود دارد که با محقق شدن یا نشدنش، برنامه دو رفتار متفاوت را از خود نشان می‌دهد.

### اجرای تکرارشونده

یک سری از خطوط برنامه چندین و چند بار اجرا می‌شوند. معمولا متغیر یا متغیرها در این حالت حضور دارند.

### استفاده دوباره یا reuse

یک سری از دستورالعمل‌ها را می‌نویسید و از آن‌ها در طول برنامه‌ی خود استفاده‌ی دوباره می‌کنید. یعنی لازم نیست که یک دستورالعمل را در جای جای برنامه بنویسید. یک بار می‌نویسید و باز استفاده می‌کنید.

موارد فوق تشکیل‌دهنده‌ی ساختمان یک برنامه به حساب می‌آیند. خب همه‌چیز خیلی ساده به نظر می‌رسد. یک ورودی می‌گیریم، بر اساس چیزی که می‌خواهیم دستورات را تایپ می‌کنیم تا به ترتیب رو به پایین اجرا شوند. در بعضی جاها نیاز است شرطی اعمال کنیم. گاهی لازم است که خطی را به دفعات تکرار کنیم. در جاهایی هم شاید

بخواهیم از قطعه‌هایی از کد چندین و چند بار استفاده کنیم. در نهایت هم خروجی روی صفحه ظاهر شود. ظاهرش ساده است. مثل این است که به کسی بگویید پایت را بگذار روی گاز و برو و وقتی به مانعی رسیدی پایت را بگذار روی ترمز. شاید در تئوری ساده باشد ولی کد نوشتن، مثل داستان‌نویسی، نقاشی، یا حتی رانندگی، یک هنر است. شما از مفاهیم و تجهیزات بسیار ساده شروع می‌کنید و یک اثر را خلق می‌کنید.

## چه مشکلی ممکن است پیش بیاید؟

همانطور که پیشتر دیدیم، در حین صحبت با پایتون باید بی‌نهایت دقیق باشیم. جابجا شدن یک حرف در یک برنامه، می‌تواند نتیجه‌ش بی‌خیال‌شدن پایتون باشد.

برنامه‌نویسان تازه‌کار وقتی با این سرسختی پایتون در مقابله با خطاهایشان روبرو می‌شوند با خودشان می‌گویند که «پایتون خیلی سخت‌گیر، خشن، و نفرت‌انگیز است. به نظر می‌رسد که پایتون با بقیه دوست و رفیق باشد، ولی حقیقتش حسودی می‌کند و از ما کینه به دل دارد. به خاطر همین موضوع هم برنامه‌های دقیق ما رو می‌گیرد و بعد به اسم «مناسب نیست» آن‌ها را به صورتمان می‌کوبد.» البته اگر از آن دسته افرادی باشید که می‌خواهید دائم مشکل را به گردن دیگری بیندازید، این پاراگراف صدق می‌کند وگرنه پایتون دقیق و سخت‌گیر است، ولی حسودی نمی‌کند بلکه کارها را به روش خود انجام می‌دهد.

یک نمونه از خطا در برنامه را می‌توانید در اینجا مشاهده کنید:

```
>>> 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


>>> 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
>>>
```

ولی در جر و بحث کردن با پایتون چیزی نصیبتان نمی‌شود، چون پایتون در گفتن (SyntaxError) صبور است و اگر لازم باشد تا آخر دنیا، این ترکیب را به شما می‌گوید. پس آرامش خودتان را حفظ کنید و به فکر پیدا کردن خطا بیفتید.

شاید بتوانیم پایتون را به یک سگ تشبیه کنیم. عاشق شماست ولی صرفا یک سری کلمه‌های رزرو شده را می‌داند. وقتی با حروفی اشتباه با او حرف می‌زنید دمش را تکان می‌دهد و به شما نشان می‌دهد که منظور شما را نمی‌فهمد ولی مشتاقانه منتظر این است که شما منظورتان را با حروف درستی بیان کنید. برای همین است که <<< را نمایش می‌دهد.

به مرور که کار با پایتون را بهتر و بهتر یاد بگیرید، با سه نوعِ کلی از خطا مواجه خواهید شد که از قرار زیرند:

### Syntax errors

خطاهای مربوط به متن همانقدر که راحت اتفاق می‌افتند، راحت هم اصلاح می‌شوند. کافیست که ببینید کجای برنامه یک خطای گرامری یا املایی کرده‌اید. پایتون تمام تلاشش را می‌کند که به شما بگوید دقیقا در چه خطی از برنامه این خطا را مرتکب شده‌اید. البته گاهی خطا در خطوط قبلی برنامه رخ داده است. در این صورت بایستی که خطی که پایتون به آن اشاره می‌کند را به عنوان سرنخ در دست بگیرید و جلو بروید.

### خطاهای منطقی یا Logic errors

یک خطای منطقی زمانی اتفاق می‌افتد که برنامه از نظر متن مشکلی ندارد ولی ترتیب خطوط و دستورات به گونه‌ای که باید باشد، نیست. یا یک خط به اشتباه به خط دیگری پیوند خورده است. به عنوان یک مثالِ واضح، این برنامه (داستان) را با هم مرور می‌کنیم:

I. یک بطری آب بردار
II. بگذارش داخل کوله‌پشتی
III. پیاده به کتابخانه برو
IV. درِ بطری آب را بگذار

مشکلی با متن نداریم ولی یکمی دیر به فکر گذاشتن درِ بطری افتادیم و کل وسایل داخل کوله‌پشتی خیس شده است!

### خطاهای معنایی یا Semantic errors

یک خطای معنایی زمانی رخ می‌دهد که شما متن برنامه را به خوبی نوشته باشید، ترتیب اجرای دستورات و خطوط هم درست باشد، ولی برنامه آن کاری که شما از او انتظار دارید را انجام ندهد. یک مثال ساده: فرض کنید می‌خواهید به یک دوست آدرس رستوران را بدهید و بگویید:

«زمانی‌که به چهار راهی که پمپ بنزین است رسیدی، بپیچ به چپ، و یک مایل برو. یک ساختمان قرمز سمت چپت خواهد بود که همان رستوران است.»

خب تا اینجا همه‌چیز خوب است. الان دوستِ شما زنگ می‌زند و می‌گوید که آن‌ها پشت یک طویله‌اند و نشانی از رستوران نیست. بعد شما می‌گویید: «وقتی به پمپ بنزین رسیدی رفتی چپ یا راست؟» و او می‌گوید: «من دقیقا همان چیزی که گفتی رو اجرا کردم. رفتم به سمت چپ» و شما جواب می‌دهید: «واقعا شرمنده، راهنمای من از نظر متن و منطق درست بود ولی یک اشتباه معنایی خیلی کوچیک داشت که باعث شد شما به خطا بیفتی و از جای اشتباهی سر در بیاری.»

در هر سه مورد بالا، پایتون تمام تلاشش را می‌کند تا کاری که شما از آن می‌خواهید را به بهترین شکل ممکن انجام دهد. این شمایید که باید بهتر با او کنار بیایید. پایتون در کارش دقیق است.

## مسیر یادگیری

گاهی در مسیر یادگیری با خود فکر می‌کنید که «مطالب این کتاب با هم جور در نمی‌آید؛ چرا سر در نمی‌آورم؟» درست مانند یادگیری «حرف‌زدن» است. یکی دو سالی طول کشید تا فقط یاد گرفتید که صدا در بیاورید. سپس شش ماهی طول کشید که از یک کلمه‌ی ساده به یک جمله برسید و شاید ۵ الی ۶ سال زمان برد تا از آن جمله‌های کوتاه یک پاراگراف بسازید. در نهایت هم باز چند سالی وقت صرف کردید تا بالاخره توانستید یک داستان کوتاهِ جذاب را بنویسید.

ولی هدف ما این است که شما پایتون را خیلی سریع‌تر یاد بگیرید برای همین در چند فصل ابتدایی همه‌ی این موارد را دَرهم به شما یاد خواهیم داد. ولی مانند یادگیری یک زبان جدید، درک مفاهیم و اُخت گرفتن با آن‌ها زمان می‌برد. به همین خاطر ممکن است گاهی با دیدن تصویر بزرگ گیج شوید. ما مدام بین سرفصل‌ها جابجا می‌شویم و قطعه‌های کوچک پازل را کنار هم می‌چینیم تا بالاخره تصویر اصلی پایتون برای شما قابل درک شود. این کتاب به صورت خطی نوشته شده است ولی این موضوع نبایستی مانع این شود که بین سرفصل‌ها جابجا شوید. اگه احساس کردید لازم است اکنون از

فصل پنج به فصل یک برگردید و یک موضوع را دو مرتبه از نظر بگذرانید، اصلا شک به خودتان راه ندهید. نگاهی به فصل‌های جلوتر بیندازید، حتی اگر درست متوجه محتوای آن نمی‌شوید. در نهایت این عقب و جلو شدن‌ها بین سرفصل‌های مختلف به شما دید بهتری در خصوص «چرایی» برنامه‌نویسی خواهد داد. با بازخوانی اطلاعات قبلی – یا حتی انجام دوباره‌ی تمرینات – خواهید دید که مطالبی که زمانی برای‌تان غیرقابل درک به نظر می‌رسید را خیلی بهتر و کامل‌تر می‌فهمید.

معمولا در یادگیری زبان‌برنامه‌نویسی اول، زمان‌هایی وجود دارد که شما بلند می‌گویید «آها، حالا گرفتم چی شد!» این زمان‌ها شما یک قدم به عقب برمی‌دارید و می‌بینید، تیشه‌هایی که به این سنگ بزرگ زده‌اید، کم‌کم شکل یک مجسمه و تصویر واقعی را به خود می‌گیرد.

گاهی مشکلْ حل نشدنی به نظر می‌آید. سخت است که تمام شب را بیدار بمانید و هیچ پیشرفتی در کارتان نکنید. کافی‌ست که به خودتان استراحت دهید، چرت بزنید یا خودتان را مهمان یک میان‌وعده کنید. گاهی توضیح مشکل برای یک شخص دیگر (حتی سگ‌تان) به شما دید بهتری برای حل آن می‌دهد. همه‌ی این‌ها باعث می‌شود تا دیدی تازه‌تر به موضوع پیدا کنید. به شما اطمینان می‌دهیم که با تمام کردن این کتاب، وقتی آن را دوباره در دست گرفتید، می‌بینید که چقدر مفاهیمْ شسته رفته و قشنگ در جای خود قرار گرفته‌اند و چقدر مفهوم همه چیز را خوب درک می‌کنید. فقط کمی زمان نیاز دارید تا آن‌ها را جذب کنید.

## واژگان فصل

**باگ / Bug:**
خطایی در برنامه.

**سی‌پی‌یو / واحد پردازش مرکزی / Central Processing Unit:**
قلب هر کامپیوتر. جایی که نرم‌افزاری که ما می‌نویسم اجرا می‌شود.

**کامپایل / Compile:**
ترجمه برنامه از یک زبان سطح بالا به یک زبان سطح پایین. این کار برنامه را برای اجرا آماده می‌کند.

**زبان سطح بالا / High-Level Language:**
یک زبان برنامه‌نویسی مثل پایتون که خواندن و نوشتنش برای انسان ساده است.

**حالت تعاملی / Interactive Mode:**
راهی برای استفاده از مفسر پایتون. به صورتی که دستورات را مستقیم در خط فرمان می‌نویسیم.

**تفسیر / Interpret:**
اجرای یک زبان سطح بالا با استفاده از ترجمه یک خط در زمان.

**زبان سطح پایین / Low-Level Language:**
یک زبان برنامه‌نویسی که برای اجرا توسط کامپیوتر بهینه و طراحی شده است. همچنین به آن «کد ماشین» یا «زبان اسمبلی» می‌گویند.

**کد ماشین / Machine Code:**
سطح پایین‌ترین زبان برای نرم‌افزار. زبانی که مستقیما توسط سی‌پی‌یو اجرا می‌شود.

**حافظه‌ی اصلی / Main Memory:**
جایی که داده‌ها و برنامه‌ها نگهداری می‌شوند. حافظه‌ی اصلی به محضی که برق برود یا سیستم خاموش و یا ری‌استارت بشود پاک می‌شود.

**تجزیه / parse:**
آزمایش و آنالیز کردن متن یک برنامه.

**قابلیت حمل / Portability:**
ویژگی‌ای که یک برنامه را قابل اجرا روی بیش از یک رایانه می‌کند.

**چاپ گزاره / Print Statement:**
دستورالعملی که به مفسر پایتون می‌گوید تا یک مقدار را روی صفحه، نمایش دهد.

**حل مساله / Problem Solving:**
پروسه‌ی فرمول‌بندی یک مساله برای پیدا کردن راه‌حل و بیان آن راه‌حل.

**برنامه / Program:**
مجموعه‌ای از دستورالعمل‌ها که یک محاسبه‌ی خاص را انجام می‌دهد.

**اعلان / Prompt:**
زمانی‌که یک برنامه، پیامی را نمایش می‌دهد و از کاربر برای ادامه‌ی کار ورودی می‌خواهد.

**حافظه‌ی جانبی / Secondary Memory:**
محل ذخیره‌ی داده و برنامه‌ها. بر خلاف حافظه‌ی اصلی، اطلاعات حتی با خاموش شدن سیستم روی حافظه‌ی جانبی باقی می‌ماند. کُندتر از حافظه‌ی اصلی است و شامل هارددیسک، فلش‌مموری و غیره می‌شود.

**معنا و مفهوم / Semantics:**
معنای یک برنامه.

**خطای معنایی / Semantic Error:**
خطایی که باعث می‌شود برنامه کاری را کند، که از آن انتظار نمی‌رود و قصد برنامه‌نویس آن نبوده است.

**کد منبع / Source Code:**
کدهای یک برنامه به زبان برنامه‌نویسی سطح بالا.

## تمرین‌ها

**تمرین ۱:** کار کردن حافظه‌ی جانبی در یک کامپیوتر چیست؟

الف) تمام محاسبات و روابط منطقی یک برنامه را اجرا کند.

ب) اطلاعات صفحات وب را از اینترنت بگیرد.

ج) اطلاعات را برای بلند مدت نگه‌داری کند — حتی بدون وجود برق.

د) از کاربر، اطلاعات ورودی بگیرد.

**تمرین ۲:** برنامه چیست؟

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

**تمرین ۴:** کدامیک از موارد زیر شامل کد ماشین می‌شود؟

الف) مفسر پایتون

ب) کیبورد

ج) فایل کد منبع پایتون

د) یک سند متنی

**تمرین ۵:** مشکلِ کد زیر چیست؟

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

**تمرین ۶:** متغیرها، مثل X در مثال زیر بعد از اجرای آن در پایتون، در کجای کامپیوتر ذخیره می‌شوند؟

الف) واحد پردازش مرکزی

ب) حافظه اصلی

ج) حافظه جانبی

د) دستگاه‌های ورودی

هـ) دستگاه‌های خروجی

**تمرین ۷:** کد زیر چه چیزی را در خروجی چاپ خواهد کرد؟

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

الف) 43

ب) 44

ج) x + 1

د) چون از نظر ریاضی معادله‌ی x = x + 1 ممکن نیست، خطا می‌دهد

**تمرین ۸:** هر کدام از موارد زیر را با توجه به یکی از قابلیت‌ها و یا اعضای ما انسان‌ها توضیح دهید:

الف) واحد پردازش مرکزی

ب) حافظه‌ی اصلی

ج) حافظه‌ی جانبی

د) دستگاه ورودی

هـ) دستگاه خروجی

به عنوان مثال معادلِ انسانی یک واحد پردازش مرکزی چیست؟

**تمرین ۹:** چطور می‌توانیم خطای "Syntax Error" را رفع کنیم؟

# فصل ۲

# متغیرها، عبارات و گزاره‌ها

## مقادیر و انواع آن‌ها

یک مقدار – مثل یک حرف یا یک عدد – پایه‌ای‌ترین چیزی‌ست که برنامه با آن کار می‌کند. مقادیری که تا اینجای کار مشاهده نموده‌ایم شما 1 و 2 و "Hello World!" می‌شوند.

این مقادیر به گروه‌های خاصی تعلق دارند. مثلا 2 یک عدد صحیح (Integer) و "Hello World!" یک رشته (String) است. ولی چرا "رشته"؟ چون شامل رشته‌ای از حروف می‌شود. حالا شما یا مفسر چگونه تشخیص می‌دهید که "Hello World!" یک رشته است؟ پاسخ ساده است: رشته‌ها بین علامت‌های نقل‌قول قرار می‌گیرند.

گزاره یا دستور print چه برای رشته چه برای عدد صحیح به خوبی کار می‌کند. با استفاده از python مفسر را اجرا می‌کنیم:

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

اگر می‌خواهید نوع یک مقدار را از مفسر بپرسید، می‌توانید از type استفاده کنید:

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

واضح است که رشته متعلق به نوع str و عدد صحیح یا integer متعلق به int است. اعداد اعشاری نیز با نام float شناخته می‌شوند.

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

خب بیایید کمی پیچیده‌ترش کنیم. نظرتان در خصوص "17" و "3.2" چیست؟ اعداد صحیح و اعشاری‌اند ولی داخل علامت نقل‌قول قرار گرفته‌اند. بد نیست از مفسرِ پایتون بپرسیم:

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

پایتون می‌گوید که رشته‌اند.

معمولا وقتی که اعداد صحیحِ بزرگی را تایپ می‌کنیم، با استفاده از کاما عدد را برای خوانایی بیشتر می‌شِکنیم. درست است که این فرمت برای پایتون قابل درک است، ولی چیزی که پایتون از آن برداشت می‌کند کاملا متفاوت است.

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

چیزی که پایتون به ما نشان می‌دهد، با چیزی که ما انتظار داریم، متفاوت است. مفسر پایتون ۱,۰۰۰,۰۰۰ را به عنوان سلسله‌ای از اعداد صحیح که با کاما جدا شده‌اند تفسیر می‌کند به همین خاطر بینش یک فاصله قرار می‌دهد و سپس چاپش می‌کند.

این اولین خطای معنایی‌ست که مشاهده می‌کنید. کد بدون برگرداندن هیچگونه خطایی به کارش ادامه داد ولی آن چیزی را که ما انتظار داشتیم برنگرداند. یعنی کارش را "درست" به معنای چیزی که ما می‌خواهیم، انجام نداد.

## متغیرها

یکی از برجسته‌ترین ویژگی‌های یک زبان برنامه‌نویسی، قابلیت کار با متغیرهاست. یک متغیر، نامی است که به یک مقدار اشاره می‌کند.

یک Assignment Statement یا گزاره‌ی گمارشی، یک متغیر را می‌سازد؛ البته اگر پیشتر موجود نباشد؛ سپس مقداری را به آن نسبت یا اختصاص می‌دهد.

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

در مثال بالا ما سه مقدار را به سه متغیر اختصاص داده‌ایم. اولین خط، یک پیغام را به متغیر message، دومی عدد ۱۷ را به n و سومی مقدار تقریبی عدد پی را به متغیر pi اختصاص داده است.

برای نمایش مقدارِ یک متغیر می‌توانید از دستور print استفاده کنید:

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

نوع یک متغیر برابر با نوع مقداری‌ست که به آن متغیر نسبت داده شده است. مثلا در مثال فوق 17 یک عدد صحیح یا int است. نوع n نیز بر اساس این مقدار int یا عدد صحیح خواهد بود:

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

## انتخاب نام برای متغیر و کلمات کلیدی

معمولاً برنامه‌نویس‌ها اسامی معنادار برای متغیر خود انتخاب می‌کنند و سپس اطلاعات مربوط به آن متغیر را داخل سند مکتوب می‌نمایند. چرا اسامی معنادار؟ به این خاطر که بهتر است هر متغیر نشانی را، از مقداری که در خود دارد، داشته باشد. مثلا شما برای ذخیره‌ی دمای اتاق که از طریق سنسور دریافت کرده‌اید، بهتر است نامی مرتبط، و با معنی را – مثل temp – انتخاب کنید تا هر جای دیگری از برنامه که خواستید از مقدار دما استفاده کنید، بین اسامی بی‌معنیِ متغیرها گیج و سر در گم نشوید.

طول نام یک متغیر می‌تواند خیلی بلند، و شامل اعداد و حروف باشد ولی نمی‌تواند که با یک عدد شروع شود. به عنوان مثال 6temp یک نام غیرمجاز برای متغیر است. نام یک متغیر می‌تواند با حرف بزرگ شروع شود، ولی بهتر است که از حرف کوچک به عنوان اولین حرف در نام یک متغیر استفاده شود (دلیلش را در آینده برای‌تان خواهیم گفت).

استفاده از کاراکتر آندراِسکور یا آندرلاین (_) نیز مجاز است و در اسامی با ترکیب چند کلمه مثل my_name یا airspeed_of_unladen_swallow استفاده می‌شود. اسامی متغیرها می‌تواند با کاراکتر آندراِسکور شروع شود، ولی بهتر است که از این کار – به غیر از موارد خاص نوشتن کد کتابخانه – اجتناب شود.

اگر شما برای یک متغیر، نام غیرمجازی استفاده کنید، یک خطای متن یا syntax error دریافت خواهید کرد:

```
>>> 76trombones = 'big parade'
  File "<stdin>", line 1
    76trombones = 'big parade'
               ^
SyntaxError: invalid syntax
>>> more@ = 1000000
  File "<stdin>", line 1
    more@ = 1000000
          ^
SyntaxError: invalid syntax
>>> class = 'Advanced Theoretical Zymurgy'
  File "<stdin>", line 1
    class = 'Advanced Theoretical Zymurgy'
          ^
SyntaxError: invalid syntax
>>>
```

نام trombon۷۶غیرمجاز است چرا که با عدد شروع شده؛ more@ غیرمجاز است چون شامل کاراکتر غیرمجاز @ می‌شود. ولی مشکل class چیست؟

نام class به این دلیل که یکی از کلمات کلیدی و رزرو شده‌ی پایتون است، نمی‌تواند به عنوان نام متغیر انتخاب شود. مفسر از این کلمات کلیدی (Keyword) برای شناسایی ساختار یک برنامه استفاده می‌کند و زمانی‌که می‌خواهید از آن‌ها به عنوان نام متغیر استفاده کنید، برنامه برای جلوگیری از گیج شدن، آن را غیرمجاز اعلام می‌کند. پایتون ۳۳ کلمه‌ی رزرو شده دارد:

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

شاید بد نباشد که این لیست را کنار دست خود داشته باشید و اگر دیدید که مفسر در خصوص نام متغیر ایراد می‌گیرد، نگاهی به لیست بیندازید و مطمئن شوید که آن اسم در بین کلمات رزرو شده نیست.

## گزاره‌ها

یک گزاره، واحدی از کُد است که مفسر پایتون توانایی اجرایش را داشته باشد. ما تا اینجا دو نوع گزاره دیده‌ایم: print برای چاپ یک عبارت و گزاره‌ی مربوط به گمارش یا اختصاص چیزی به متغیر (Assignment).

وقتی شما یک گزاره را در حالت تعاملی وارد می‌کنید، مفسر آن را اجرا و سپس نتیجه را نمایش می‌دهد. البته اگر نتیجه‌ی قابل مشاهده‌ای داشته باشد.

یک اسکریپت معمولا شامل یک سری گزاره می‌شود. اگر بیش از یک گزاره در اسکریپت داشته باشیم، نتیجه در زمانی‌که گزاره در داخل اسکریپت اجرا می‌شود، نمایش داده می‌شود. به عنوان مثال:

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

خروجی زیر را خواهد داشت:

```
1
2
```

به دو نکته در اسکریپت بالا توجه داشته باشید. اول اینکه گزاره‌ها به ترتیب از بالا به پایین خوانده شدند و دوم اینکه دومین گزاره یعنی x = 2 در اسکریپت بالا یک گزاره‌ی گمارشی‌ست و خروجی قابل نمایش ندارد.

## عملوند و عملگرها

پیش از هر چیز تعریف عملوند و عملگر ضروری به نظر می‌رسد.

عملگر عبارت است از نشانه‌های ویژه‌ای که برای محاسبه به کار می‌رود. مانند جمع یا ضرب. مقادیری که عملگر روی آن‌ها اعمال می‌شود، را عملوند می‌گویند.

**عملوند:** در ریاضیات و برنامه‌نویسی رایانه، یک عملوند هدف یک عملیات ریاضی است. هر عبارت که بین دو عملگر قرار بگیرد و یا بعد از یک عملگر بیاید یک عملوند محسوب می گردد.

**عملگر:** در ریاضیات، یک عملگر تابعی است که بر تابعی دیگر اعمال می‌شود. در مواردی ممکن است یک عملگر را یک عمل ریاضی گویند مانند «عمل جمع» که در اصل عملگر جمع می‌باشد. در علوم کامپیوتر پس از تعریف متغیرها و مقدار دادن به آنها/ عملیاتی روی آنها انجام می‌شود. انجام عملیات توسط عملگر صورت می‌گیرد.

عملگرهای + و - و * و / و // و ** به ترتیب جمع، تفریق، ضرب، تقسیم، تقسیم مسطح و به‌توان‌رساندن را انجام می‌دهند. به مثال‌های زیر توجه کنید:

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

بین عملگر تقسیم در پایتون 2.x و پایتون 3.x تفاوت‌هایی وجود دارد. در پایتون 3.x نتیجه‌ی تقسیمْ یک عدد اعشاری خواهد بود.

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

در پایتون 2.x اگر دو عدد صحیح را بر هم تقسیم کنیم، نتیجه هم یک عدد صحیح خواهد بود:

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

برای اینکه در پایتون 3.x به نتیجه‌ی مشابه پایتون 2.x یعنی یک عدد صحیح برسید، از عملگر // استفاده کنید:

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

در پایتون 3.x نتیجه‌ی تقسیم بیشتر به چیزی که از یک حساب‌کتاب درست و ماشین‌حساب انتظار دارید شبیه است.

## عبارت و عبارت‌ها در برنامه‌نویسی

یک عبارت، ترکیبی از مقادیر، متغیرها و عملگرهاست. یک مقدار، یا یک متغیر به خودی خود، یک عبارت‌اند. به بیان دیگر هر کدام از خطوط زیر یک عبارت به حساب می‌آیند (البته با این پیش‌فرض که متغیر x شامل یک مقدار است):

```
17
x
x + 17
```

اگر شما یک عبارت را در حالت تعاملی پایتون وارد کنید، مفسر آن را حل کرده و نتیجه را روی خروجی چاپ می‌کند:

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

ولی در یک اسکریپت اوضاع متفاوت است. یکی از جاهایی که تازه‌واردها گیج می‌شوند همین تفاوتِ رفتار مفسر پایتون در اسکریپت و حالت تعاملی است.

**تمرین ۱:** عبارت‌های زیر را یکی یکی در حالت تعاملی پایتون وارد کنید. مفسر بعد از هر عبارت چه چیزی را نمایش می‌دهد؟

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

## ترتیب عملیات‌ها

زمانی‌که بیش از یک عملگر در عبارت ظاهر می‌شود، ترتیب محاسبه و عملیات روی آن عبارت بر طبق قوانین خاصی انجام می‌پذیرد. به آن قوانین اولویت می‌گوییم. برای عملگرهای ریاضی، پایتون از عرف ریاضی معمول پیروی می‌کند.

I. پرانتز، بالاترین اولویت را دارد. به همین خاطر می‌توانید از آن برای تحمیلِ ترتیبِ انجام عملیات به صورتی که می‌خواهید استفاده کنید. از آن‌جایی که عبارات داخل پرانتز اول از همه بررسی می‌شوند، نتیجه‌ی 2 * (۳ - ۱) می‌شود 4 و (1 + 1) ** (۵ - ۲) می‌شود 8. حتی می‌توانید از پرانتز برای آسان‌تر و خواناتر ساختن یک عبارت استفاده کنید. مثلا این عبارت (minute * 100) / 60 بدون پرانتز هم نتیجه‌ی یکسانی می‌دهد ولی خواناتر است.

```
>>> 2 * (3 - 1)
4
>>> (1 + 1) ** (5 - 2)
8
>>> minute = 120
>>> minute * 100 / 60
200.0
>>> (minute * 100) / 60
200.0
>>>
```

II. به توان رساندن در رده‌ی دوم اولویت‌ها قرار می‌گیرد. 1 + 1 ** 2 می‌شود 3 و 3 ** 1 * 3 می‌شود 3، و نه ۲۷.

```
>>> 2 ** 1 + 1
3
>>> 3 * 1 ** 3
3
>>>
```

III. در رده‌ی سوم اولویت، ضرب و تقسیم با هم قرار می‌گیرند.

IV. در رده‌ی چهارم جمع و تفریق در کنار هم جای می‌گیرند و اولویت یکسانی دارند.

V. زمانی‌که عملگرها با اولویت یکسان در یک عبارت استفاده می‌شوند، به ترتیب از سمت چپ به راست محاسبه می‌گردند. در نتیجه 5 - 3 - 1 می‌شود 1، و نه عدد 3 چرا که اول 3 از 5 کسر می‌شود و سپس 1 از باقیمانده‌ی آن.

```
>>> 5 - 3 - 1
1
>>>
```

زمانی‌که شک داشتید که عملیات چطور انجام می‌شود، از پرانتز استفاده کنید تا خیالتان راحت شود. استفاده از پرانتز همانطور که پیشتر گفتیم هم به خوانایی کد اضافه می‌کند، هم احتمال خطا را کاهش می‌دهد.

## عملگر پیمانه

عملگر پیمانه، روی اعداد صحیح اعمال شده، و باقیمانده عملوند اول بر دوم را به دست می‌آورد. گیج‌کننده بود؟ به زبان ساده‌تر، فرض کنید که دو عدد ۵ و ۲ را در اختیار دارید؛ حاصل تقسیم پنج بر دو، می‌شود ۲ به عنوان خارج قسمت، و ۱ به عنوان باقیمانده. عملگر پیمانه در برنامه‌نویسی، وظیفه‌ی نشان دادن باقیمانده را به عهده می‌گیرد.

از آنجا که اگر یکی از عملوندها عدد اعشاری باشد، خارج قسمت نیز اعشاری خواهد شد، و عملا باقیمانده نخواهیم داشت، این عملگر تنها روی اعداد صحیح قابل اجراست.

نشانه‌ای که برای عملگر پیمانه در پایتون و البته بسیاری دیگر از زبان‌های برنامه‌نویسی استفاده می‌شود، علامت درصد % است. به مثال زیر توجه کنید:

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

خب هفت تقسیم بر سه، دو می‌شود به علاوه‌ی یک به عنوان باقیمانده که در مثال بالا دیدید. عملگر پیمانه از آنچه که در آینه به نظر می‌رسد به شما نزدیک‌تر است و در آینده خواهید فهمید که این عملگر چقدر به کارتان خواهد آمد. مثلا فرض کنید می‌خواهید ببینید که X بر Y بخش‌پذیر است یا خیر. اینجا با استفاده از عملگر پیمانه و بررسی خروجی آن به راحتی می‌توانید نتیجه را بیرون بکِشید.

نمونه‌ی دیگر استفاده از عملگر پیمانه، بیرون کشیدن عدد یا عددهای سمت راست از یک مقدار است. چگونه؟ مثلا ما عدد ۴۵۸ را داریم که رقم سمت راست آن ۸ است. حالا با استفاده از عملگر پیمانه آن را اینگونه استخراج می‌کنیم:

```
>>> 458 % 10
8
```

حالا فرض کنید که دو رقم سمت راست را نیاز دارید. به مثال زیر دقت کنید:

```
>>> 458 % 100
58
```

## عملیات روی رشته

عملگر + برای استرینگ‌ها یا رشته‌ها هم کارایی دارد ولی این کارایی متفاوت با نوع ریاضی آن است. + رشته‌ها را بهم پیوند می‌دهد یا به عبارتی زنجیره‌بندی می‌کند. بگذارید با مثالی به شما نشان دهم زمانی‌که از + برای زنجیره‌بندی استفاده می‌کنید چه اتفاقی می‌افتد.

```
>>> first = 10
>>> second = 15
>>> print first + second
25
>>> type(first)
<type 'int'>
>>> type(second)
<type 'int'>
>>> first = '10'
>>> second = '15'
>>> print first + second
1015
>>> type(first)
<type 'str'>
>>> type(second)
<type 'str'>
>>>
```

در مثال بالا ما دو بار ۱۰ و ۱۵ را با هم جمع بستیم. مرتبه‌ی اول آن‌ها را بدون علامت نقل‌قول به متغیرهایشان شناساندیم و در مرتبه‌ی دوم با استفاده از علامت نقل‌قول. علامت نقل‌قول به متغیر می‌گوید که نوع این مقدار رشته یا استرینگ است. در مثال اول، پایتون با دو عدد ریاضی سر و کار دارد و آن‌ها را به صورت ریاضی با هم جمع می‌زند. در مثال دوم، پایتون با دو رشته سر و کار دارد و آن‌ها را زنجیره‌بندی می‌کند. یعنی آخر اولی را به اول بعدی و همین‌طور تا آخر همه را به هم پیوند می‌دهد. در مثال ما ۱۰ را با ۱۵ پیوند داد و خروجی ۱۰۱۵ را چاپ کرد.

## درخواست ورودی از کاربر

گاهی لازم است که مقدار یک متغیر را از کاربر از طریق کیبورد بگیریم. پایتون از یک تابع توکاری-شده یا درونی، به این معنی که این تابع به صورت پیش‌فرض با پایتون همراه است، برای این کار استفاده می‌کند. اسم این تابع input است و ورودی خود را از کیبورد می‌گیرد. وقتی این تابع فراخوانده می‌شود، برنامه متوقف شده و منتظر کاربر می‌شود تا چیزی تایپ کند. زمانی‌که کاربر Enter را می‌زند، برنامه به ادامه‌ی کار خود

می‌پردازد و چیزی که کاربر تایپ کرده بود را به عنوان یک رشته یا استرینگ در متغیر مورد نظر ذخیره می‌کند. مثال زیر را در حالت تعاملی پایتون وارد کنید تا خودتان هم ببینید:

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

در مثال بالا دیدید که تابع ما منتظر شد تا کاربر چیزی را وارد کند، ولی هیچ نشانه‌ای از اینکه منتظر است وجود نداشت. یعنی اگر کسی غیر از ما پشت سیستم نشسته بود، به اعلانِ خط فرمان نگاه می‌کرد و می‌گفت: «هان! که چی؟» پس شاید بد نباشد که خط اعلان چیزی را به ما اعلام کند، مثلا «کاربر محترم لطفا یک ورودی با کیبورد وارد کن». برای اینکار بایستی داخل پرانتز بعد از input یک رشته قرار دهید. به مثال زیر دقت کنید:

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

آن \n در انتهای رشته بیانگر این است که ورودی را در «خط جدید» بگیرد و احتمالا مخفف newline است. این نویسه مختص شکستن خط و رفتن به خط بعدی‌ست؛ و به همین خاطر است که ورودی کاربر در خط جدید ظاهر می‌شود. به خط سوم در کد بالا نگاه کنید.

اگر از کاربر یک عدد صحیح بخواهید چه؟ input ورودی را به صورت رشته، و نه عدد صحیح یا integer می‌گیرد. برای این کار از تابع ()int استفاده می‌کنیم:

```
>>> 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
```

حالا اگر به جای یک عدد صحیح، یک رشته وارد تابع int کنیم، چه می‌شود؟ بیایید ببینیم:

```
>>> 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:
```

همانطور که مشاهده می‌کنید زمانی‌که رشته یک عدد صحیح نباشد تابع int خطا می‌دهد. در آینده به بررسی و نحوه‌ی برطرف کردن این نوع اشکالات خواهیم پرداخت.

## کامنت یا نظر روی برنامه

به مرور که برنامه بزرگ‌تر می‌شود، پیچیده‌تر خواهد شد. با پیچیده‌تر شدن، خواندنش سخت‌تر می‌شود و ممکن است کار به جایی برسد که شما با خواندن یک قطعه کد متوجه نشوید که قرار است چه کاری را انجام دهد.

به همین دلیل نوشتن کامنت یا نظر روی کدها – به زبان آدمیزاد – ایده‌ی خوبی به حساب می‌آید. در پایتون کامنت‌ها با # شروع می‌شوند.

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

در مثال بالا، کامنت در یک خط جدا در اسکریپت قرار داشت. ولی شما می‌توانید آن را به انتهای خط نیز اضافه کنید بدون اینکه اختلالی در کارکرد برنامه ایجاد کند:

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

هر چیزی که بعد از # قرار می‌گیرد، نادیده گرفته خواهد شد و هیچ تاثیری در برنامه نخواهد داشت. کامنت‌ها زمانی‌که یک ویژگی نه‌چندان واضح را به کد اضافه می‌کنید، بسیار کاربردی‌اند.

معمولا کاربر می‌فهمد که این قطعه کد چه کاری را انجام می‌دهد، ولی چرایی‌اش مبهم خواهد بود در نتیجه بهتر است در کامنت‌ها چراییِ قرار دادن قطعه کد را تشریح کنید. یک کد خوب بایستی به خوبی کامنت شده باشد.

به عنوان مثال کامنت زیر یک کامنت غیرضروری و اضافه است، چون بدون وجود این کامنت هم کاربر می‌فهمد که در این خط کد عدد پنج به متغیر v اختصاص پیدا می‌کند:

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

ولی کامنت زیر اطلاعات خوبی را در اختیار کسی که در حال خواندن کد است می‌گذارد:

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

البته اسامی خوب برای متغیر هم می‌تواند نیاز به کامنت کردن را بهینه کند؛ از طرفی اسامی بلند هم نه تنها امکان اشتباه را بالا می‌برد که می‌تواند برای مقاصدی که عبارت طولانی‌ست کار را پیچیده کند. در نتیجه انتخاب نام متغیر و نحوه‌ی کامنت کردن آن بایستی هوشمندانه باشد.

## انتخاب نام‌های به خاطر ماندنی

تا زمانی‌که از قوانین ساده‌ی نام‌گذاری متغیرها استفاده و از کلمه‌های رزرو شده اجتناب می‌کنید، پایتون به شما برای انتخاب نام متغیر گیر نخواهد داد. ولی این انتخاب می‌تواند چه در زمان نوشتن و چه در زمان خواندن برنامه باعث سردرگمی شما و دیگران شود. البته منظور از دیگران، کسانی است که کد را می‌خوانند وگرنه برای پایتون و برنامه‌ی نهایی این موارد مهم نیستند. به سه برنامه‌ی زیر نگاه کنید. این سه برنامه دقیقا برای کاربر نهایی یک کار را انجام می‌دهند و کاربر نهایی اصلا متوجه اینکه نام متغیرها چیستند نخواهد شد:

```
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 سه این برنامه‌ها را دقیقا یکی می‌داند ولی ما آدم‌ها آن را به شکل‌های متفاوت می‌بینیم. معمولا یک انسان نیتِ برنامه را در قطعه کد دوم، راحت‌تر حدس می‌زند. ساعت، نرخ و پرداخت تداعی‌کننده‌ی این است که برنامه قرار است «ساعت» کاری را در «نرخی» ضرب و مقدار را به عنوان «پرداخت» برگرداند. دلیلش هم ساده است: برنامه‌نویس از نام‌های به جا و خوبی برای برنامه‌اش استفاده کرده است.

ما اسامی‌ای که به صورت زیرکانه برای متغیرها انتخاب شده‌اند را "mnemonic variable names" یا «نام‌های به خاطر ماندنی» می‌نامیم. در برنامه‌نویسی این نام‌ها به ما کمک می‌کنند تا منظور هر متغیر را از اسم آن تشیخص دهیم.

مشکلی که ممکن است در انتخاب نام‌های به خاطر ماندنی برای تازه‌کاران پیش بیاید، اشتباه گرفتن آن‌ها با نام‌های رزرو شده است. مثلا به برنامه‌ی زیر نگاه کنید:

```
for word in words:
    print word
```

کد بالا چه کاری را انجام می‌دهد؟ کدام یکی از کلمه‌های for و word و in و words کلمه‌های رزرو شده‌اند و کدامیک اسامی‌ای که توسط برنامه‌نویس انتخاب شده‌اند؟ آیا پایتون می‌فهمد که کلمه‌ی word به چه منظوری به کار برده شده است؟ خواندن برنامه‌ی بالا برای یک تازه‌کاری که هنوز کلمه‌های رزرو شده را از بَر نشده، می‌تواند این شُبهه را ایجاد کند که word هم یک کلمه‌ی رزرو شده است، یا in نام یک متغیر است.

کد زیر، دقیقا همان کاری را انجام می‌دهد که کد بالا:

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

تفکیک کلمات رزرو شده‌ای که برای پایتون دارای معنی خاص‌اند و نام متغیرها که توسط برنامه‌نویس انتخاب شده‌اند در مثال بالا، برای یک تازه‌کار ساده‌تر است. کاملا واضح است که پایتون در خصوص پیتزا و اسلایس‌های آن چیزی نمی‌داند.

ولی اگر کار برنامه‌ی ما بررسی داده‌ها برای پیدا کردن کلمات باشد، آیا استفاده از اسم پیتزا برای متغیر کار اشتباهی نیست؟ انتخاب آن‌ها به عنوان نام متغیر، برنامه‌نویس را گیج و متن برنامه را از آنچه هدفش است دور خواهد کرد.

```
for word in words: print word
```

در مثال بالا قسمت‌هایی از کد که برای پایتون با معنی‌اند بولد شده است و نام متغیرها به صورت عادی نمایش داده شده. البته در این خصوص به خودتان هم زیاد سخت نگیرید. بسیاری از ویرایشگرهای متنی که با سینتکس پایتون آشنایی دارند، به خوبی کلمات رزرو شده را تشخیص می‌دهند و لازم نیست که نگران آن باشید. بعد از مدت زمان کوتاهی و البته با تکرار و تمرین کم‌کم کلمات رزرو شده را به راحتی از کلمه‌های با معنی، که برای متغیرها انتخاب شده است، تشخیص خواهید داد.

## دیباگ کردن / اشکال زدایی

تا اینجای کار با پیام‌هایی که پایتون به عنوان SyntaxError صادر می‌کند آشنا شده‌ایم. خطاهایی که ممکن است مرتکب شوید شامل انتخاب نام‌های رزرو شده برای متغیر مثل class و yield یا استفاده از کاراکترهای غیرمجاز مثل odd~job یا US$ و یا حتی استفاده از فاصله در بین نام‌ها می‌شود. زمانی‌که بین اسم متغیری فاصله می‌اندازید، پایتون فکر می‌کند که شما دو عملوند بدون هیچ عملگری دارید و در نهایت خطا صادر می‌کند:

```
>>> bad name = 5
SyntaxError: invalid syntax
>>> month = 09
  File "<stdin>", line 1
    month = 09
            ^
SyntaxError: invalid token
```

پیام‌هایی که SyntaxError می‌دهد اطلاعات زیادی را در اختیار ما نمی‌گذارد. رایج‌ترین پیام SyntaxError: invalid syntax و SyntaxError: invalid token است که هیچ کدام حاوی اطلاعات به درد بخوری نیستند. اینجاست که باید کمی فسفر بسوزانید و ببینید که کجای کد اشتباهات رایجی که بالاتر به آن‌ها اشاره شد را انجام داده‌اید. البته سرنخ‌هایی هم این پیغام خطا به شما می‌دهد.

```
>>> bad name = 5
SyntaxError: invalid syntax
>>> month = 09
  File "<stdin>", line 1
    month = 09
            ^
SyntaxError: invalid token
```

یک خطای ران‌تایم یا خطای زمان اجرا چه زمانی اتفاق می‌افتد؟ اغلب زمانی‌که شما قبل از تعیین چیزی، آن را فراخوانی کنید؛ مثلا از متغیری که هنوز مقداری ندارد استفاده کنید. خب فرض کنید که متغیر todayTemp را ساخته‌اید حالا در جای دیگری از برنامه می‌خواهید از آن استفاده کنید. به جای todayTemp به اشتباه todayTmep را تایپ می‌کنید. پایتون نام دوم را غلط املایی نمی‌داند، بلکه آن را اسم یک متغیر جدید در نظر می‌گیرد و چون مقداری در آن وجود ندارد، خطا صادر می‌کند:

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

شما rate را تعریف نکرده بودید در نتیجه وقتی می‌خواهید از آن استفاده کنید، پایتون خطای NameError را صادر می‌کند.

یکی دیگر از خطاهای رایج تازه‌کاران استفاده از نام‌ها با حروف کوچک و بزرگ اشتباهی‌ست. برای پایتون متغیر Latex از latex متفاوت است، چرا که به کوچکی و بزرگی حروف حساس است.

یکی از خطاهای معمول دیگر که یک خطای معنایی به حساب می‌آید، ترتیب و اولویت اجرای عملیات است. به عنوان مثال اگر معادله‌ی ۱/۲π را به همین صورت بنویسید:

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

اول از همه تقسیم انجام می‌شود. در نتیجه چیزی که برنامه به شما می‌دهد π/2 است که با عددی که ما انتظار داریم متفاوت است. پایتون به هر حال نمی‌فهمد که منظور شما از معادله چیست، و خطایی هم صادر نمی‌کند. فقط جواب اشتباه تحویلتان می‌دهد. همیشه ترتیب و اولویت عملیات‌ها را بررسی کنید و در جایی که شک دارید از پرانتز استفاده کنید.

## واژگان فصل

**گمارش / Assignment:**
یک گزاره‌ای که یک مقدار را به یک متغیر اختصاص می‌دهد.

**هم‌بند کردن / Concatenate:**
بستن دو عملوند به یکدیگر.

**نظر / کامنت / Comment:**
اطلاعاتی در یک برنامه که برای دیگر برنامه‌نویسان که به کد منبع دسترسی دارند، در برنامه قرار داده می‌شود و روی نحوه‌ی اجرای برنامه هیچ تاثیری ندارد.

**عددیابی / ارزیابی / Evaluate:**
ساده‌سازی یک عبارت با استفاده از عملگرها برای دریافت یک مقدار.

**عبارت / Expression:**
ترکیبی از متغیرها، عملگرها و مقادیر که بیانگر یک مقدار است.

**اعشار / Floating Point:**
نوعی از اعداد که شامل قسمت اعشاری نیز می‌شود.

**تقسیم گردشده رو به پایین / Floor Division:**
تقسیم گردشده رو به پایین. زمانی‌که عددی را تقسیم می‌کنید، اگر قسمت اعشاری آن را حذف نمایید به این کار Floor Division می‌گویند.

**عدد صحیح / Integer:**
نوعی که بیانگر اعداد کامل می‌شود.

**کلمه کلیدی / Keyword:**
یک کلمه‌ی رزرو شده توسط کامپایلر که برای پارس/تجزیه و تحلیل کردن یک برنامه استفاده می‌شود. مثلا شما نمی‌توانید از کلماتی شبیه به if یا def یا while به منظور استفاده برای اسامی متغیرها استفاده کنید چرا که این کلمه‌ها کی‌وُرد یا رزرو شده‌اند.

**به خاطر ماندنی / Mnemonic:**
ما به متغیرها نام‌هایی می‌دهیم که بدانیم مقداری که در آن‌ها ذخیره شده بیانگر چیست. استفاده از نام‌های به خاطر ماندنی به ما و دیگر برنامه‌نویسانی که به کد منبع دسترسی دارند کمک شایانی می‌کند.

**عملگر پیمانه / Modulus Operator:**
عملگری که با % نمایش داده می‌شود و مقدار باقیمانده از یک تقسیم را برمی‌گرداند. به عنوان مثال عملگر پیمانه در عبارت 5 % 13 عدد 3 خواهد بود.

**عملوند / Operand:**
مقداری که عملگر روی آن عملیات انجام می‌دهد.

**عملگر / Operator:**
نمادی خاص که نشانگر محاسبه‌ی ساده‌ای مثل جمع، تفریق، ضرب یا هم‌بند کردن یک استرینگ/ رشته است.

**قوانین اولویت‌ها / Rules of Precedence:**
قوانینی که به برنامه می‌گوید کدام محاسبه در یک عبارتی با چند عملگر و عملوند، بر دیگری مقدم است و بایستی زودتر انجام پذیرد.

**گزاره / دستور / Statement:**
قسمتی از کد که نمایانگر یک دستور یا عملی باشد. تا اینجای کتاب، گزاره‌ها به اختصاص دادن مقادیر یا چاپ کردن محدود بودند.

**رشته / String:**
نوعی که توالی‌ای از کاراکترها را نمایش می‌دهد.

**نوع / Type:**
دسته‌ای که یک مقدار در آن قرار می‌گیرد. مثلا تا اینجای کار با نوعِ عدد صحیح (type int) نوعِ عدد اعشاری (type float) و نوع رشته (type str) آشنا شدیم.

**مقدار / Value:**
یکی از واحدهای پایه‌ی داده، شبیه به یک عدد یا رشته که برنامه با آن کار می‌کند و آن را دستکاری می‌نماید.

**متغیر / Variable:**
نامی که به یک «مقدار» اشاره می‌کند.

## تمرین‌ها

**تمرین ۲:** برنامه‌ای بنویسید که از input استفاده کرده و نام کاربر را طلب کند. سپس با استفاده از متغیر مورد نظر، به کاربر سلام دهد. خروجی باید چیزی شبیه به این باشد:

```
Enter your name: Iman
Hello Iman
```

**تمرین ۳:** برنامه‌ای بنویسید که از کاربر ساعاتی که کار کرده و نرخ مزدش را بپرسد و در نهایت دستمزدش را محاسبه کند.

```
Enter Hours: 35
Enter Rate: 2.75
Pay: 96.25
```

در خصوص اینکه، عدد بدست آمده حتما دو عدد اعشار داشته باشد نگران نباشید. می‌توانید از تابع درونی‌شده‌ی پایتون به اسم round برای گرد کردن عدد و تنظیم تعداد اعشار آن استفاده کنید.

**تمرین ۴:** فرض کنید که گزاره‌های زیر را اجرا کرده‌اید:

```
width = 17
height = 12.0
```

برای هرکدام از عبارت‌های زیر، مقدار و نوعِ این مقدار را یادداشت کنید:

```
•    width//2
•    width/2.0
•    height/3
•    1 + 2 \* 5
```

بعد از محاسبه‌ی دستی، از مفسر پایتون برای بررسی پاسخ‌های‌تان استفاده کنید.

**تمرین ۵:** برنامه‌ای بنویسید که از کاربر درجه‌ی سانتی‌گراد را بگیرد و آن را به فارنهایت تبدیل کند و مقدار تبدیل شده را در خروجی چاپ نماید.

# فصل ۳

# اجرای مشروط

## بولی و عبارت‌های بولی

یک عبارت Boolean چیست؟ خب به فارسی به آن بولی می‌گوییم و عبارتی‌ست که یا غلط است یا صحیح. مثال زیر از علمگر == برای مقایسه‌ی دو عملوند استفاده می‌کند تا ببیند آن‌ها با هم برابرند و True را صادر کند یا در غیر این‌صورت False را.

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

True یا False مقادیر ویژه‌ای به حساب می‌آیند که به نوع bool تعلق دارند. لازم است که باز هم تکرار کنم که آن‌ها رشته یا استرینگ نیستند بلکه bool اند.

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

علمگر == یکی از علمگرهای مقایسه است. بقیه عبارتند از:

```
x != y
```

x با y برابر نیست؛

```
x > y
```

x از y بزرگتر است؛

```
x < y
```

x از y کوچکتر است؛

```
x >= y
```

x بزرگتر یا مساوی با y است؛

```
x <= y
```

x کوچکتر یا مساوی با y است؛

```
x is y
```

x درست همان y است؛

```
x is not y
```

x همان y نیست.

احتمالا شما با این عملگرها از قبل آشنایی داشته‌اید، با این حال نمادهایی که پایتون استفاده می‌کند می‌تواند گاهی متفاوت باشد. یک خطای رایج استفاده از = به جای == برای عمل مقایسه‌ی تساوی دو چیز است. خاطرتان باشد که = برای اختصاص مقدار است و یک Assignment Operator یا عملگر گمارشی است و == یک

علمگر برای مقایسه؛ و همینطور <= و >= هم در پایتون معنی ندارند. علامت مساوی باید بعد از > و < بیاید.

## عملگرهای منطقی

سه عملگر منطقی وجود دارد. این عملگرهای منطقی عبارتند از «و/and» و «یا/or» و «نه/not». معنای این علمگرها دقیقا برابر با معنی لفظی آن‌هاست. مثلا عبارت زیر را در نظر بگیرید:

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

این عبارت اگر ایکس از صفر بزرگ‌تر باشد و همزمان از ده کوچک‌تر باشد صحیح است. یعنی هر دو طرف «and» بایستی صحیح باشد که نتیجه True شود و در هر سه حالت دیگر False پاسخ برنامه خواهد بود. یک مثال دیگر:

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

این عبارت اگر شرط اول یا دوم درست باشد، مقدار «صحیح» یا «True» را برمی‌گرداند. به عبارتی اگر n بر ۲ یا ۳ بخش‌پذیر باشد عبارت صحیح از آب در می‌آید. یعنی کافی‌ست که یکی از دو طرف عبارت «or» صحیح باشد تا نتیجه True باشد. و مثال آخر مربوط به عملگر «نه» است.

```
not (x > y)
```

این عبارت زمانی صحیح است که عبارت x > y صحیح نباشد. یعنی مقدار این عبارت زمانی صحیح است که x کوچکتر یا مساوی y باشد. در عبارت‌هایی که عملگر not وجود دارد، مقدار عبارت (بدون در نظر گرفتن not) هرچه باشد، خروجی برعکس خواهد بود. اگر True باشد False و اگر False باشد True.

اگر بخواهیم خیلی مته به خشخاش بگذاریم، عملوندهای یک عملگر منطقی بایستی که عبارت‌های بولی باشند. ولی پایتون در این زمینه زیاد سختگیر نیست و تمام عددهای غیر از صفر را نیز به عنوان True یا صحیح تفسیر می‌کند. به مثال زیر دقت کنید:

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

این انعطاف در جاهایی می‌تواند به درد بخورد ولی از طرفی می‌تواند گمراه‌کننده هم باشد. بهتر است که تا قبل از زمانی‌که می‌دانید دارید چه می‌کنید، از اعداد به این شکل استفاده نکنید.

## اجرای شرطی

برای ساختن برنامه‌های به درد بخور، معمولا همیشه بایستی از شروط استفاده کنیم. بررسی کنیم که اگر فلان، در نتیجه بیسار و برنامه عکس‌العمل و رفتار مناسبی را در مواقع لازم نشان دهد. ولی چگونه؟ گزاره‌های شرطی به ما این امکان را می دهد.

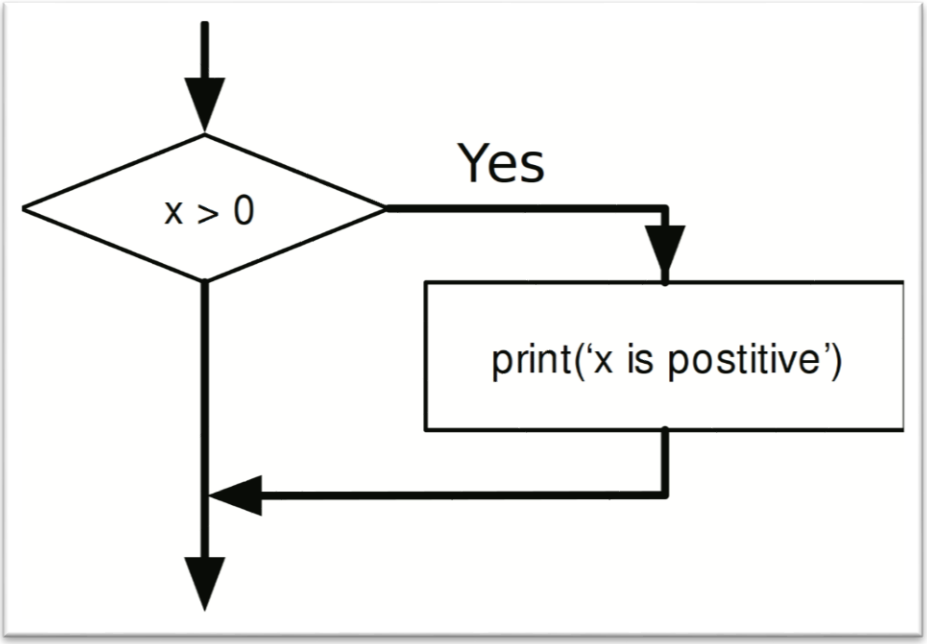
به یک مثال بسیار ساده از if نگاه کنید:

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

عبارت بولی بعد از if، شرط خوانده می‌شود. ما گزاره‌ی if را با کلون یا دو نقطه (:) به پایان می‌رسانیم و سپس خطوط مربوط به آن شرط را می‌نویسم. خطوط بعد از if که زیرمجموعه‌ای از آن شرط به حساب می‌آیند مقداری به سمت داخل می‌روند (تورفته‌اند).

**پایتون به این تورفتگی‌ها بسیار حساس است.** در اصل شما به این صورت به برنامه می‌فهمانید که بدنه‌ی if شامل چه کدهایی‌ست. برای ایجاد این تورفتگی حتما بایستی از فاصله یا اسپیس استفاده کنید. یک اشتباه رایج استفاده از Tab برای این

کار است که پایتون را گیج می‌کند. ما برای هر تو رفتگی چهار اسپس را در نظر گرفته‌ایم؛ البته تعداد اسپیس‌ها کاملا بستگی به خودتان دارد ولی بایستی تمام گزاره‌های داخل بدنه‌ی if تعداد مشابهی اسپیس داشته باشد. در نظر داشته باشید وقتی کدی، تو در تو می‌شود، آن قسمت تو رفته بایستی شامل فاصله‌های بیشتری شود که در ادامه‌ی این کتاب به آن‌ها خواهیم پرداخت.




اگر شرطِ منطقی درست باشد، در نتیجه گزاره‌ی تورفته اجرا می‌شود. در غیر این صورت برنامه از روی آن گزاره پرش می‌کند.

گزاره‌های if ساختار مشابهی با حلقه for و یا function definitionها دارند. ولی شاید این سوال برای‌تان پیش بیاید که function definition چیست؟ در حقیقت function definition چیزی به غیر از یک تابع نیست که برای پایتون توسط شما یا هر کس دیگری تعریف شده است.

اگر گزاره‌های بعد از شرط، بیش از یک خط باشند، کل بدنه‌ی شرط دو قسمت می‌شود. بخش اول هدر که خودِ شرطِ منطقی‌ست و با دو نقطه پایان می‌یابد؛ و بخش دوم گزاره‌هایی که با تورفتگی نسبت به هدر مشخص می‌شوند. به این گزاره‌ها یی که بیش از یک خط‌اند Compound Statements یا گزاره‌های مرکب می‌گویند.

محدودیتی برای تعداد گزاره‌هایی که در بدنه‌ی یک شرط منطقی می‌تواند جای بگیرد وجود ندارد ولی مقدار کمینه‌ی آن یک گزاره است. یعنی بایستی حداقل یک گزاره بعد از شرطِ ما وجود داشته باشد وگرنه اصلا شرط برای چه؟

البته گاهی لازم است که برای رزرو حالتی یک شرط خالی بنویسید. که در آینده از آن استفاده کنید. پایتون برای رد شدن از این شرط دستور pass را در خود دارد. به مثال زیر نگاه کنید:

```
if x < 0;
    pass        # need to handle negative values!
```

اگر شما یک گزاره‌ی شرطی را درحالت تعاملیِ مفسر پایتون وارد کنید، خط اعلان به سه نقطه تغییر شکل می‌دهد. به این صورت پایتون به شما می‌گوید که در درون یک بلاک از کد قرار دارید. مثال زیر را ببینید (تو رفتگی گزاره‌ی print را فراموش نکنید):

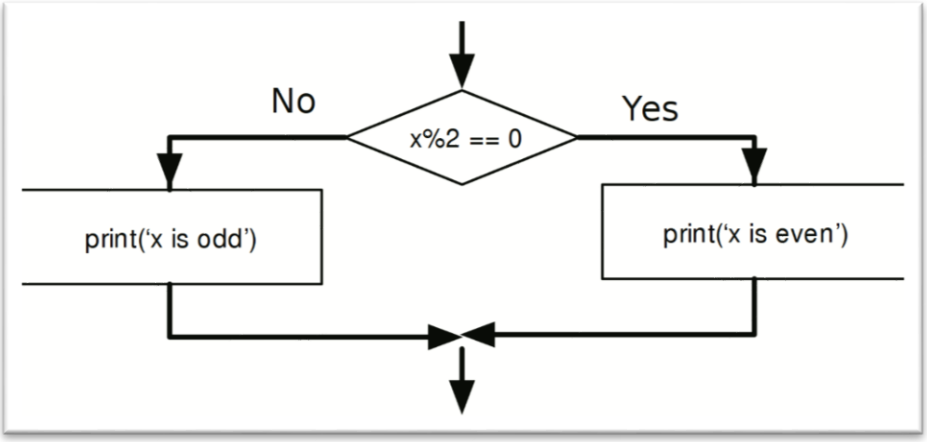
```
>>> x = 3
>>> if x < 10:
...     print('Small')
...
Small
>>>
```

## اجرای ثانوی

فرم دیگری از گزاره‌ی if فرم اجرای ثانوی‌ست. در این حالت «اگر / if» گزاره‌ی شرطی درست نباشد از آن شرط پرش کرده و به «در غیر این صورت / else» رفته و گزاره‌ی مربوط به آن را اجرا می‌کند. به متن زیر دقت کنید:

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

در مثال بالا اگر باقی‌مانده‌ی x تقسیم بر ۲ برابر با صفر باشد، شرط محقق شده و گزاره‌ی print('x is even') اجرا می‌شود، در غیر این صورت (هر اتفاقی بیفتد که شرط محقق نشود) گزاره‌ی print('x is odd') اجرا می‌شود. به دیاگرام زیر نگاه کنید:



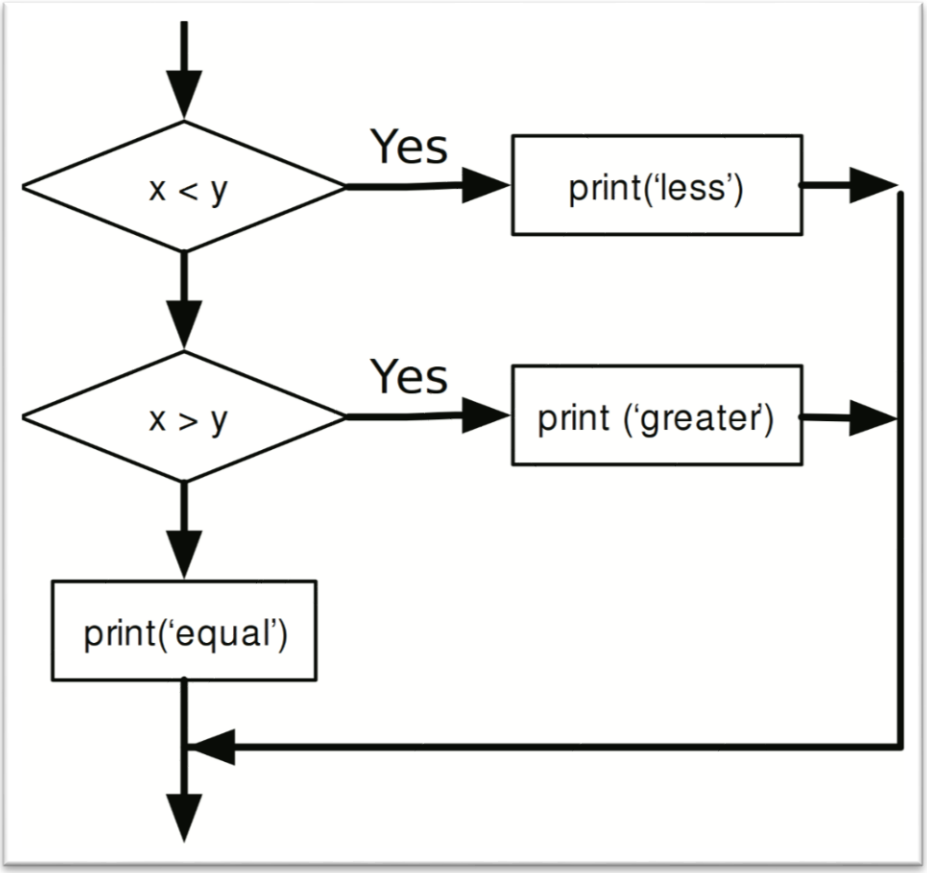

از آنجایی که شرط یا محقق می‌شود یا نمی‌شود، یکی از گزاره‌ها اجرا خواهد شد. یا گزاره‌ای که می‌گوید شرط درست است (اول) یا گزاره‌ای که می‌گوید «در غیر این صورت» (گزاره‌ی دوم). این حالت، branches یا شاخه‌ها نیز خوانده می‌شود چرا که در جریان اجرا، برنامه به شاخه‌های مختلف تقسیم می‌شود.

## شرط‌های زنجیروار

گاهی بیش از دو احتمال وجود دارد و ما نیاز به شاخه‌های بیشتری در برنامه‌مان داریم. یکی از راه‌های نوشتن این مدل شرط‌ها استفاده از شرط‌های زنجیروار یا زنجیره‌ای یا Chained Conditional 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 خلاصه‌شده‌ی else if است. در اینجا نیز تنها یکی از شاخه‌ها اجرا می‌شوند. هر کدام که صحیح از آب در بیاید، همان اجرا شده و برنامه از بدنه‌ی شرط خارج می‌شود.




محدودیتی برای استفاده از elif وجود ندارد. اگر else یا «درغیراین‌صورت» وجود دارد، بایستی آن را در انتهای بدنه‌ی شرط بیاورید.

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

هر شرط به ترتیب از بالا به پایین بررسی می‌شود و در صورت غلط بودن، شرط بعدی مورد آزمون قرار می‌گیرد. زمانی‌که هر یک از شرط‌ها صحیح باشد، گزاره‌ی موجود در آن اجرا و برنامه از بدنه‌ی شرط خارج می‌شود.

حتی اگر دو شرط در یک بدنه‌ی اجرای شرطی درست از آب در بیاید، اولی اجرا شده و برنامه از بدنه‌ی شرط خارج می‌شود. به مثال زیر دقت کنید:

```
x = 4
if x < 10:
    print ('x is less than 10')
elif x < 5:
    print ('x is greater than 5')
else:
    print ('nothing')
```

مقدار x برابر با 4 است، در نتیجه شرط اول و دوم درست‌اند، ولی برنامه فقط گزاره‌ی print ('x is less than 10') را چاپ خواهد کرد.

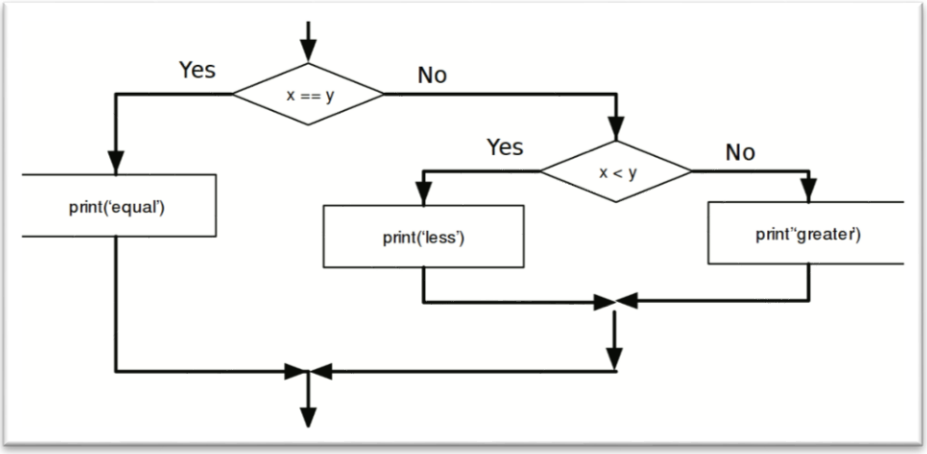
## شرط‌های تو در تو

یک شرط می‌تواند در داخل یک شرط دیگر نیز قرار بگیرد. به مثال زیر با سه شاخه‌ی مجزا دقت کنید:

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

شرط بیرونی‌تر دارای دو شاخه‌ی مجزاست. شاخه‌ی اول شامل یک گزاره‌ی ساده می‌شود. شاخه‌ی دوم خود شامل یک شرط دیگر می‌شود، که دارای دو شاخه است. دو

شاخه‌ی دیگر خود شامل گزاره‌های ساده‌ای می‌شوند ولی آن‌ها هم می‌توانند حاوی شرط‌های بیشتر یا گزاره‌های پیچیده‌تر شوند.




با وجود اینکه تو رفتگی ساختار متن کد، شروط داخلی را مشخص می‌کند، با این حال شرط‌های تو در تو به سرعت خوانایی خود را از دست می‌دهند. در کل بهتر است که تا زمانی‌که احتیاج به شرط‌های تو در تو ندارید، از آن‌ها استفاده نکنید. به عنوان مثال، کد زیر را ببینید:

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

گزاره‌ی print تنها در صورتی که هر دو شرط صحیح باشند اجرا می‌شود. این کد را می‌توانید به صورت زیر و با استفاده از عملگر and بازنویسی کنید تا از شروط تو در تو اجتناب کرده باشید.

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

پیشتر اشاره کرده بودیم که بایستی خطوط داخل بدنه if با فاصله‌های مساوی نسبت به کد اصلی مشخص شوند. در خصوص شرط‌های تو در تو، بایستی که گزاره‌های شرطِ داخلی، مقدار بیشتری تو رفتگی داشته باشند تا پایتون بفهمد که گزاره‌ها مربوط به بدنه‌ی کدام if است. این موضوع برای while و for و... که نیاز به تو رفتگی دارند، نیز صادق است.

## استفاده از try و except برای استثناها

پیشتر یاد گرفتیم که با input از کاربر، ورودی بگیریم و سپس با استفاده از تابع int آن را به عدد صحیح تبدیل کنیم. یادتان می‌آید که ورودی input به عنوان نوع «استرینگ» ذخیره می‌شد. حالا به مثال زیر نگاه کنید:

```
>>> prompt = "What...is the airspeed velocity of an unladen
swallow?\n"
>>> 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:
>>>
```

زمانی‌که ما در حال اجرای این نمونه گزاره‌ها در حالت تعاملی باشیم، خب خطا را می‌بینیم و مفسر به ادامه‌ی کارش می‌پردازد، ولی در اسکریپت اوضاع فرق می‌کند و برنامه متوقف شده و با نشان دادن یک تریس‌بْک از انجام گزاره‌ی بعدی باز می‌ماند.

تریس‌بْک چیست؟ گزارشی که برنامه در خصوص وضعیتش نشان می‌دهد را تریس‌بک می‌گوییم.

به برنامه‌ی زیر که قرار است مقدار فارنهایت را به سانتی‌گراد تبدیل کرده و چاپ نماید، دقت کنید:

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


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

اگر ما این کد را اجرا کرده و مقدار غیرمجازی به ورودی بدهیم، با یک پیامِ نه‌چندان دلچسب، برنامه به کار خود پایان می‌دهد.

```
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'
```

برای رفع مشکلِ برخورد به خطاهای «قابل پیش‌بینی» و همچنین «غیرمترقبه» چه کاری می‌توان انجام داد؟ پایتون ساختاری برای مقابله با این حالت‌ها با خود به همراه دارد که به آن except/try می‌گوییم. ایده‌ی try و except چیست؟ زمانی‌که احساس می‌کنید ممکن است سلسله‌ای از دستورالعمل‌ها در حین اجرا، به مشکل بر بخورند، می‌توانید گزاره‌هایی را تعیین کنید تا آن حالت خاص را مدیریت کنند. این گزاره‌ها که زیر except قرار می‌گیرند، تنها در صورتی اجرا می‌شوند که برنامه به خطا بر بخورد، در غیر این صورت پایتون از روی آن‌ها پرش می‌کند.

می‌توانید به try و except به مثابه یک ضمانت‌نامه نگاه کنید. ضمانت اجرای پاره‌ای از کدها.

از آنجایی که بهتر است دست به کار شویم تا درک بهتری از این قابلیت پیدا کنیم، برنامه‌ی بالا را از نو و این بار با try و except بازنویسی می‌کنیم:

```
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: http://www.py4e.com/code3/fahren2.py
```

پایتون شروع به اجرای بلوک try می‌کند. اگر همه چیز به خوبی پیش برود از بخش except پرش کرده و به اجرای ادامه‌ی اسکریپت می‌پردازد. ولی اگر مورد خاصی در بلوک try حادث شود، از آن قطعه کد خارج شده و به اجرای سلسله گزاره‌های بلوک except می‌پردازد.

```
python fahren2.py
Enter Fahrenheit Temperature:72
22.22222222222222
```

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

مدیریت استثناها با استفاده از گزاره‌ی try، «گرفتن استثنا» یا catching an exception نامیده می‌شود. در مثال ما، بلوک except خطایی را چاپ می‌کند و می‌گوید «لطفا یک عدد وارد کنید». در کل، گرفتن استثنا امکان رفع مشکل را به ما می‌دهد. یا ما را به تصحیح مشکل و تلاش مجدد وا می‌دارد و یا اینکه برنامه را با وقار به پایان می‌رساند.

## میانبر در بررسی عبارت‌های منطقی

زمانی‌که پایتون یک عبارت منطقی را پردازش می‌کند، این ارزیابی از سمت چپ به راست صورت می‌گیرد. مثلا عبارت منطقی زیر را در نظر بگیرید:

```
x >= 2 and (x/y) > 2
```

در این مثال اگر قسمت اول عبارت (قبل از and) غلط از آب در بیاید، پایتون قسمت دوم عبارت یعنی (x/y) > 2 را نادیده می‌گیرد، چرا که به هر حال – با توجه به and – پاسخ کل عبارت «False» خواهد بود.

زمانی‌که پایتون تشخیص می‌دهد که محاسبه‌ی ادامه‌ی عبارت، تاثیری در نتیجه نهایی عبارت ندارد، از حساب کردن ادامه‌ی عبارت منطقی دست می‌کشد. به این حالت short-circuiting یا «میانبر زدن در ارزیابی» می‌گویند.

شاید بگویید که خب این چیز خاصی نیست، یک محاسبه‌ی کمتر شاید مساله‌ی بزرگی برای یک برنامه و روند اجرایش نباشد، به هر حال این یک رفتار منطقی از پایتون است. ولی شما با استفاده از همین قابلیت پایتون می‌توانید الگویی به اسم «الگوی گاردین» یا «الگوی محافظ» بسازید. چطور؟ این کدها را در مفسر پایتون وارد 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
>>>
```

چه اتفاقی افتاد؟ بخش اولِ قطعه کد سوم صحیح است پس پایتون به سراغ قسمت دوم می‌رود و با معادله‌ای روبرو می‌شود که از نظر منطقی جور در نمی‌آید. تقسیم عدد بر صفر! و خب خطا صادر می‌شود. در قطعه کد اول x >= 2 صحیح است و قسمت دوم یعنی (x/y) > 2 هم همینطور، در نتیجه جواب این عبارت‌های منطقی هم صحیح است. در قطعه کد دوم، قسمت اول عبارت غلط است و پایتون بدون در نظر گرفتن قسمت دوم، حکم «غلط» بودن کلِ عبارت را صادر می‌کند. اینجاست که پایتون در ارزیابی عبارت، میانبر می‌زند.

حالا کمی فسفر بسوزانیم. چطور می‌توانیم از این رفتار پایتون، مانعی برای برخوردن به خطا بسازیم؟ یعنی به پایتون بگوییم که اگر y برابر با صفر بود، به سراغ حل معادله دوم نرود و نتیجه را صادر کند؟ خب کافی‌ست که در سمت چپِ قسمتِ دومِ عبارت بالا، همین شرط را اضافه کنیم:

```
>>> 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 عبارت منطقیِ اول x >= 2 غلط است در نتیجه ارزیابی در همانجا و رسیدن به عملگر and پایان یافته و نتیجه‌ی «غلط» صادر می‌شود. در دومین عبارت منطقی x >= 2 صحیح، ولی y != 0 غلط است در نتیجه پایتون به ارزیابی بخش سوم عبارت یعنی (x/y) نمی‌رسد و مقدار «غلط» را بازمی‌گرداند.

در عبارت منطقی سوم y != 0 بعد از (x/y) قرار گرفته است در نتیجه ارزیابی عبارت با رسیدن پایتون به (x/y) به مشکل برخورد کرده و خطا صادر می‌شود.

در قطعه کد دوم، ما از y != 0 به عنوان محافظ برای اجرای صحیح و بدون خطای (x/y) استفاده می‌کنیم تا مطمئن شویم که y برابر با 0 نیست و در نتیجه برنامه به خطا برخورد نمی‌کند.

## واژگان فصل

**بدنه کد / body:**
سلسله گزاره‌هایی که در یک گزاره‌ی مرکب قرار دارند.

**بولی / boolan expression:**
عبارتی که مقدار آن یا True است یا False.

**شاخه / branch:**
سلسله‌ای از گزاره‌هایی که بخشی از گزاره‌ی شرطی را تشکیل می‌دهد.

**شروط زنجیروار / chained conditional:**
یک گزاره‌ی شرطی با چندین شاخه‌ی مجزا.

**عملگر مقایسه / comparison operator:**
یکی از عملگرهایی که عملوندهای خود را مقایسه می‌کند: == و =! و < و > و =< و =>

**گزاره‌ی شرطی / conditional statement:**
گزاره‌ای که کنترل جریان اجرا را با توجه به شرایط در دست می‌گیرد.

**شرط / condition:**
یک عبارت بولی در یک گزاره‌ی شرطی که تعیین کننده‌ی اجرا شدن یا نشدن یک شاخه است.

**گزاره‌های مرکب / compound statement:**
یک گزاره که شامل یک هدر و یک بدنه می‌شود. هدر با دو نقطه (:) تمام می‌شود و بدنه نسبت به گزاره تو رفتگی دارد.

**الگوی محافظ / guardian pattern:**
گاهی ما یک عبارت منطقی را با توجه به رفتار «میانبر» در پایتون برای جلوگیری از ایجاد مشکل در برنامه استفاده می‌کنیم. این عبارت منطقیِ اضافی، الگوی محافظ را می‌سازد.

**عملگرهای منطقی / logical operator:**
یکی از عملگرهایی که با یک عبارت بولی ترکیب می‌شود: and یا or یا not.

**شرط تو در تو / nested conditional:**
یک گزاره شرطی که در یک شاخه از یک گزاره‌ی شرطی دیگر ظاهر می‌شود.

**تِریسْبک / traceback:**
خروجی چاپ‌شده در زمان وقوع یک استثنا، از لیست توابعی که اجرا شده است.

**میانبر / short circuit:**
گاهی ممکن است پایتون در میانه‌ی راه محاسبه‌ی یک عبارت منطقی، بدون بررسی کلِ عبارت به نتیجه‌ی نهایی برسد. در اینجا پایتون با دانستن مقدار نهایی، باقی عبارت را نادیده می‌گیرد. مثلا در یک عبارت منطقی که با and به هم متصل شده کافیست که یک طرف عبارت، «غلط» از آب در بیاید که نتیجه نیز «غلط» شود. به همین خاطر اگر پایتون بخش اول عبارت را «غلط» ارزیابی کند، از بررسی ادامه‌ی عبارت اجتناب کرده و مقدار را برمی‌گرداند.

## تمرین‌ها

**تمرین ۱:** برنامه‌ای که پیش‌تر برای تعیین دستمزد نوشتید را بازنویسی کنید به صورتی که کارمندهایی که بیش از ۴۰ ساعت کار کرده‌اند به ازای هر ساعت اضافه، یک و نیم برابر حقوق دریافت کنند. به عبارتی اگر ۴۵ ساعت کار کرده، ۴۰ ساعت حقوق معمول و ۵ ساعت حقوق با ضریب یک و نیم دریافت کند.

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

**تمرین ۲:** حالا با استفاده از try و expect برنامه را برای دریافت ورودی غیر از عدد و نشان دادن پیغام درست به کاربر بهینه کنید. خروجی غیر عددی مثل nine بایستی شبیه به خطوط زیر باشد:

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

**تمرین ۳:** برنامه‌ای بنویسید که عددی بین 0.0 و 1.0 دریافت کند. اگر عدد خارج از این محدوده بود، خطا صادر نماید و برای اعداد بین این محدوده نمره به سبک زیر چاپ کند:

```
Score   Grade
>= 0.9     A
>= 0.8     B
>= 0.7     C
>= 0.6     D
< 0.6    F
~~~
```

خروجی با توجه به ورودی‌های نمونه بایستی به این صورت باشد:

```
Enter score: perfect
Bad score
Enter score: 10.0
Bad score
Enter score: 0.75
C
Enter score: 0.5
F
```

برنامه‌ها را با ورودی‌های مختلف اجرا کنید و مطمئن شوید که درست کار می‌کنند.

# فصل ۴

# توابع

## احضار توابع

در برنامه‌نویسی، یک فانکشن یا تابع، سلسله‌ای از گزاره‌ها تحت نام یا عنوانی‌ست که یک محاسبه‌ی خاص را انجام می‌دهد. زمانی‌که شما یک تابع را تعریف می‌کنید، در حقیقت یک نامی برای آن انتخاب کرده و سپس تحت آن نام یک سری گزاره را می‌گنجانید. در ادامه می‌توانید آن‌ها را توسط همان نام «کال» یا «فراخوانی» کنید. ما پیشتر فراخوانی توابع را دیده‌ایم. به مثال زیر دقت کنید:

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

نام تابع در مثال بالا type است. عبارتی که در پرانتز آمده «آرگویمنت» آن تابع به حساب می‌آید. اگر آرگویمنت شامل یک متغیر یا مقدار شود، ما آن‌ها را به تابع – به مانند مثال بالا – ارسال می‌کنیم. نتیجه‌ی احضار تابع type در مثال بالا، نشان دادن نوع داده‌ی آرگویمنتی‌ست که به آن ارسال کرده‌ایم.

معمولا می‌گویند که یک تابع آرگیومنتی را «می‌گیرد» ونتیجه‌ای را «برمی‌گرداند.»

## توابع توکاری شده

پایتون تعدادی از توابع مهم را در خودش دارد و لازم نیست که برای احضارشان، ابتدا آن‌ها را تعریف کنیم. سازنده‌ی پایتون، یک سری تابع برای حل مسائل معمول نوشته و در داخل پایتون برای استفاده‌ی ما قرار داده است.

مثلا تابع max و min بیشترین و کمترین مقدار موجود در یک لیست را برمی‌گرداند.

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

تابع max به ما بزرگترین کاراکتر موجود در رشته را نشان می‌دهد و کاراکتر min کوچیکترین را؛ که در مثال بالا یک فاصله یا space است.

یک تابع بسیار پرکاربرد دیگر len نام دارد. این تابع آیتم‌های موجود در آرگیومنت را شمارش می‌کند و نتیجه را برمی‌گرداند. مثلا تعداد کاراکترهای یک رشته را شمرده و به صورت یک عدد صحیح آن را برمی‌گرداند:

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

این توابع فقط محدود به رشته‌ها نیستند و می‌توانند روی هر مقداری محاسبات لازم را انجام و جواب پس بدهند.

به یاد داشته باشید که اسم توابع توکاری شده را به عنوان نام متغیر استفاده نکنید. به عنوان مثال اسم متغیر خود را max نگذارید.

## تابع‌های تبدیل نوع

پایتون شامل تابع‌هایی برای تبدیل یک مقدار به «نوع» دیگر می‌شود. برای نمونه تابع int یک مقدار را از شما گرفته، و در صورت واجد شرایط بودنش، یک عدد صحیح تحویل شما می‌دهد. اگر هم مقدار واجد شرایط نباشد، این تابع به شما در خصوص مشکل گزارش خواهد داد:

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

تابع int می‌تواند یک عدد اعشاری را نیز به عدد صحیح تبدیل کند. البته کارش گِرد کردن نیست، بلکه قسمت اعشاری را می‌بُرد:

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

تابع float یک عدد صحیح یا رشته را گرفته و به عدد اعشاری تبدیل می‌کند:

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

در نهایت تابع str یک آرگیومنت را دریافت و به رشته مبدل می‌کند:

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

## اعداد تصادفی

ورودی یکسانی به برنامه‌های کامپیوتری بدهید و هر بار خروجی یکسانی تحویل بگیرید. قطعی بودن پاسخ، معمولا یک ویژگی خوب به حساب می‌آید، چرا که ما انتظار محاسبات یکسانی از عملیات‌های کامپیوتری داریم. ولی گاهی برای برخی از برنامه‌ها نیاز به رفتار غیرقابل پیش‌بینی احساس می‌شود. مثلا بازی‌ها یک مثال واضح از این امرند.

ساخت یک برنامه‌ای که کاملا در مقابل یک محاسبه‌ی جبری و قطعی قرار بگیرد کار ساده‌ای نخواهد بود. حداقل به نظر می‌آید که نیست. یکی از این روش‌ها استفاده از الگوریتم‌هایی‌ست که اعداد شبه‌تصادفی ایجاد می‌کنند. اعداد شبه‌تصادفی، واقعا تصادفی نیستند و بر اساس محاسبات جبری به دست می‌آیند ولی برای ما و شما، تشخیص غیرتصادفی بودن آن‌ها غیرممکن است.

ماژول random، توابعی را فراهم می‌کند که پایتون بتواند این اعداد شبه تصادفی را ایجاد نماید. ما در اینجا به اعداد شبه تصادفی، اعداد تصادفی می‌گوییم.

تابع random در پایتون، یک عدد اعشاری بین ۰/۰ و 1.0 ایجاد می‌کند که شامل خود عدد ۱/۰ نمی‌شود. هر زمان که random را فراخوانی کنید، یک عدد تصادفیِ بلند در این محدوده دریافت می‌کنید. برای نمونه نگاهی به اسکریپت زیر بیندازید (می‌توانید این کدها را در حالت تعاملی پایتون نیز وارد کنید):

```
import random


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

با اجرای این اسکریپت ده عدد تصادفی بین ۰/۰ و ۱/۰ ایجاد خواهد شد. در جدول پایین اسکریپت به همراه خروجی برنامه را آورده‌ایم:

```
>>> import random
>>> for i in range(10):
...     x = random.random()
...     print(x)
...
0.597816151859746
0.14302601179310193
0.21502653599314525
0.6998553060747957
0.9475634141321392
0.014420883720896227
0.03784440862132332
0.08296168390036773
0.09072806706955494
0.07216573611640587
>>>
```

**تمرین ۱:** برنامه را روی سیستم خود اجرا کنید و ببینید چه اعدادی دریافت خواهید کرد. چند بار این کار را تکرار کنید و نگاهی به اعدادی که دریافت کرده‌اید بیندازید.

تابع random یکی از چندین توابعی‌ست که کارهای مرتبط با اعداد تصادفی را انجام می‌دهد. تابع randint پارامترهای low و high را گرفته و یک عدد صحیح بین آن‌ها برمی‌گرداند:

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

برای انتخاب یک عنصر از میان یک سلسله آیتم می‌توانید از choice استفاده کنید:

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

ماژول random همچنین توابعی را برای ساخت مقادیر تصادفی از توزیع‌های پیوسته – شامل توزیع نرمال، نمایی، گاما و چند نمونه‌ی دیگر – در خود دارد.

## توابع ریاضی

پایتون یک ماژول به اسم math دارد که توابع ریاضی پر استفاده را در اختیار شما قرار می‌دهد. قبل از اینکه از آن استفاده کنید، لازم است که درون‌ریزی‌اش کنید:

```
>>> import math
>>>
```

اینکار یک ماژل مقصد به اسم math می‌سازد. اگر شما ماژل مقصد را چاپ کنید، اطلاعاتی در خصوص آن در خروجی ظاهر خواهد شد:

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

ماژول مقصد شامل توابع و مقادیر مشخص شده‌ایست که دسترسی به آن‌ها را از طریق اسم خاصِ ماژول و اسم خاص تابع ممکن می‌سازد. این دو اسم با یک نقطه از هم جدا می‌شوند. به این قالب dot notation یا «نماد نقطه‌ای» می‌گویند.

```
>>> ratio = signal_power / noise_power
>>> decibels = 10 * math.log10(ratio)
>>> radians = 0.7
>>> height = math.sin(radians)
```

اولین مثال، لگاریتم بر پایه‌ی ۱۰ ــِ نسبت سیگنال به نویز را محاسبه می‌کند. ماژول math تابعی به اسم log را نیز مهیا می‌کند که لگاریتم بر پایه‌ی e می‌گیرد.

مثال دوم سینوس radians را محاسبه می‌کند. نام متغیر radians در اصل می‌گوید که sin و بقیه توابع مربوط به مثلثات (cos و tan و غیره)، آرگویمنت را به رادیان می‌گیرند.

برای تغییر درجه به رادیان می‌توانید آن را بر ۳۶۰ تقسیم و در pi۲ ضرب نمایید:

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

عبارت math.pi متغیر pi را از ماژول math می‌گیرد. مقدار این متغیر تقریبا برابر با عدد پی است و تا حدود ۱۵ رقم اعشار دقت دارد.

اگر از مثلثات سر در می‌آورید، نتایج پیشین را با مربع ریشه دو، تقسیم بر دو مقایسه کنید.

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

## اضافه کردن یک تابع

تا اینجای کار از توابعی که در خود پایتون وجود داشت استفاده کردیم، ولی پایتون این امکان را به ما می‌دهد که توابع جدید نیز اضافه کنیم. یک شناسانش تابع، نام یک تابع و گزاره‌های آن را تعیین می‌کند. این گزاره‌ها زمانی‌که تابع فراخوانده می‌شود اجرا می‌گردند. زمانی‌که یک تابع را تعریف کردید، می‌توانید از آن به تعداد دفعاتی که عشق‌تان کشید دوباره و دوباره استفاده کنید بدون اینکه مجبور باشید کد آن را راه‌به‌راه بنویسید.

مثال:

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

کلمه‌ی def در ابتدای کار، یک کلمه‌ی کلیدی (keyword) برای تعریف تابع است. سپس اسم تابع را مشخص می‌کنیم. مثلا در مثال بالا print_lyrics اسم تابع است. قوانین انتخاب اسم متغیرها، برای توابع نیز صادقند. حروف، اعداد و بعضی از علائم مجازند ولی اولین کاراکتر نمی‌تواند عدد باشد. از کلمات کلیدی برای اسم تابع نبایستی استفاده کنید و بایستی از انتخاب نام یکسان برای یک متغیر و تابع در یک برنامه اجتناب کنید.

پرانتزهای خالی در جلوی یک تابع نشان می‌دهد که این تابع آرگویمنتی را دریافت نمی‌کند. در آینده توابعی خواهیم ساخت که به عنوان ورودی آرگیومنت دریافت می‌کنند.

اولین خط یک تابع را به اسم هِدِر و بقیه آن را بادی یا بدنه‌ی تابع می‌نامیم. هدر با دو نقطه تمام می‌شود و بدنه با مقداری تورفتگی نسبت هدر در ادامه آورده می‌شود. رسم این است که تورفتگی از چهار فاصله (Space) تشکیل شده باشد. بدنه می‌تواند شامل هر مقدار گزاره که نیاز است شود. در اصل محدودیتی در کار نیست.

رشته‌هایی که در گزاره‌ی پرینت آورده می‌شوند بین علامت نقل‌قول قرار می‌گیرند. علامت نقل‌قول به دو صورت 'رشته' و "رشته" استفاده می‌شود. البته از سینگل‌کوت (') بیشتر استفاده می‌کنند. گاهی خودِ رشته در درون خود سینگل‌کوت دارد. مثلاً عبارت It's me در درون خود یک سینگل‌کوت دارد. این سینگل‌کوت باعث می‌شود که پایتون گیج شده و با رسیدن به آن، رشته را تمام شده فرض کند. اینجاست که از دابل‌کوت (") بایستی استفاده کنیم.

اگر شما یک تابع را در حالت تعاملی پایتون وارد کنید، مفسر علامت (...) را چاپ می‌کند تا به شما نشان دهد نوبت به وارد کردن بدنه‌ی کد در تابع است.

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

برای تمام کردن بدنه در حالت تعاملی، Enter را در یک سطر خالی فشار دهید. یعنی بعد از نوشتن آخرین خط از بدنه، دو بار اینتر را بزنید.

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

ما به مقدار print_lyrics فانکشن آبجکت (function object) می‌گوییم. فانکشن آبجکت از نوع تابع به حساب می‌آید.

دستورِ فراخوانی یک تابع دقیقا برابر با نام خود تابع است:

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

زمانی‌که یک تابع را تعریف کردید، می‌توانید از آن در داخل توابع دیگر نیز بهره ببرید. مثلا می‌توانیم یک تابع با اسم repeat_lyrics بنویسیم و از print_lyrics در آن استفاده کنیم.

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

حالا تابع repeat_lyrics را فراخوانی می‌کنیم:

```
>>> 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.
```

البته درست است که آهنگ این مدلی نبود ولی خب حداقل یاد گرفتید که یک تابع چطوری کار می‌کند.

## تعریف تابع و استفاده از آن

کدهای قسمتِ قبل را که سر هم کنیم، کل برنامه چیزی شبیه به این 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: http://www.py4e.com/code3/lyrics.py
```

برنامه شامل دو تابعِ تعریف شده به اسم‌های print_lyrics و repeat_lyrics می‌شود. توابعِ تعریف شده دقیقا مثل سایر گزاره‌ها اجرا می‌شوند ولی نتیجه‌ی اجرایشان ساخته شدنِ فانکشن‌آبجکت‌هاست. گزاره‌های داخل تابع تا زمانی‌که تابع فراخوانی نشود، اجرا نخواهند شد و خودِ تابع هیچ خروجی‌ای را نمی‌سازد.

همانگونه که انتظار می‌رود، قبل از اجرای یک تابع، بایستی که آن را تعریف کرده باشید. به عبارت دیگر، تعریف تابع بایستی که قبل از اولین باری که فراخوانده می‌شود آورده شود.

**تمرین ۲:** آخرین خط این برنامه یعنی repeat_lyrics() را بِبُرید و قبل از تعریف توابع قرار دهید. حالا برنامه را اجرا کنید. چه خطایی دریافت می‌کنید؟

آشنا شدن با نوع و خروجی خطاها، باعث می‌شود که شما در آینده و در مواجه به آن‌ها سریع‌تر مشکل را تشخیص دهید.

**تمرین ۳:** فراخوانی تابع را به پایین اسکریپت برگردانید. حالا تابع print_lyrics را زیر تابع repeat_lyrics قرار دهید. برنامه را اجرا کرده. چه اتفاقی می‌افتد؟

## جریان اجرا

برای اینکه تشخیص دهید تابع را در کجای یک برنامه تعریف کنید، بایستی بدانید که در چه زمانی برای اولین بار فراخوانده می‌شود. به عبارتی بایستی از جریانِ اجرای یک برنامه مطلع باشید.

اجرا همیشه از اولین خط برنامه آغاز و در هر لحظه یک گزاره به ترتیب از بالا به پایین اجرا می‌شود.

توابعْ جریان و روند اجرای یک برنامه را دستکاری نمی‌کنند ولی گزاره‌های داخل یک تابع، تا زمانی‌که آن تابع فراخوانی نشود، اجرا نخواهند شد.

فراخوانی یک تابع شبیه به یک راه فرعی در درون برنامه و جریان اجراست. به جای اینکه روند اجرای برنامه به خط بعد از تابع برود، مسیرش را به سمت تابع کج کرده و به داخل بدنه‌ی آن می‌پرد. سپس گزاره‌های موجود در بدنه‌ی تابع را اجرا نموده و در نهایت به مکانی که مسیرش را کج کرده بود برمی‌گردد و جریانِ اجرا را از سر می‌گیرد.

به نظر ساده می‌رسد. ولی در جریان اجرای یک تابع ممکن است یک تابع دیگر نیز فراخوانده شود. اوضاع وقتی پیچیده‌تر می‌شود که در جریان اجرای تابع دوم نیز اصلا بعید نیست که یک تابع دیگر فراخوانده و جریان اجرا به سمت آن کج شود.

خوشبختانه پایتون راه‌بلدِ خوبی است و دقیقا می‌داند که هر لحظه در کجای نقشه‌ی برنامه ایستاده و مسیرش به کدام طرف است. هر زمان که کار با یک تابع به سر رسید، پایتون به سر راه قبل از اجرای آن تابع رفته و ادامه‌ی مسیر را از سر می‌گیرد. زمانی هم که به پایان برنامه رسید، آن را خاتمه می‌دهد.

حالا هدف از این قصه‌ی سر در گم ما چه بود؟ خواستیم بگوییم که زمانی‌که یک برنامه را می‌خوانید همیشه از بالا به پایین و به ترتیب نخوانید، بلکه بهتر است که جریان اجرا را دنبال کنید. مثلا لازم نیست گزاره‌های یک تابع را تا قبل از فراخوانی آن، صرفا به خاطر اینکه در بالای برنامه قرار گرفته، بخوانید و آنالیز کنید.

## پارامترها و آرگویمنت‌ها

بعضی از توابع توکاری‌شده‌ای که تاکنون استفاده کرده‌ایم، به آرگیومنت احتیاج داشتند. به عنوان نمونه زمانی‌که math.sin را فراخوانی می‌کنیم، یک عدد به عنوان آرگیومنت به آن می‌فرستیم. گاهی توابع بیش از یک آرگیومنت را دریافت می‌کنند. مثلا math.pow دو آرگیومنت می‌گیرد: ۱) پایه و ۲) توان.

در داخل تابع، آرگیومنت‌ها به متغیرهایی – که آن‌ها را پارامتر می‌نامیم – اختصاص داده می‌شوند. بهتر است نگاهی به مثال زیر بیندازیم. در این مثال کاربر یک تابع که یک آرگیومنت دریافت می‌کند را تعریف کرده است:

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

این تابع یک آرگیومنت را به پارامتری که نامش bruce است تخصیص داده است. در مثال بالا زمانی‌که تابع فراخوانده می‌شود، مقدار پارامتر (هر چیزی که باشد) دو بار چاپ می‌شود.

این تابع با هر مقداری که قابل چاپ باشد، کار خواهد کرد.

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

قوانینی که برای توابعِ توکاری‌شده صادق است، برای توابعی که کاربر تعریف کرده نیز صدق می‌کند، در نتیجه از هر عبارتی به عنوان آرگیومنت برای تابع print_twice می‌توانیم بهره ببریم.

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

آرگیومنت قبل از اینکه تابع فراخوانده شود، بررسی می‌شود. مثلا در مثال ما 'Spam *4' و همچنین math.cos(math.pi) تنها یک بار وارسی می‌شوند.

همچنین می‌توانید از متغیر به عنوان یک آرگیومنت بهره ببربید:

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

نام متغیری که ما به عنوان آرگیومنت (michael) به تابع می‌فرستیم، هیچ دخل و تصرفی در نام پارامتر ما (bruce) ندارد. به عبارتی مقداری که فراخوانی می‌شود – در اینجا michael – مهم نیست که چه باشد، چون تابع با پارامتری که با آن تعریف شده – در اینجا bruce – کار می‌کند. به عبارت ساده در تابع print_twice ما همه را bruce می‌نامیم، حالا هر آرگیومنتی که می‌خواد وارد تابع شود تفاوتی ایجاد نمی‌کند، چرا که نامش در داخل تابع، bruce خواهد بود.

## توابع نتیجه‌ده و توابع بی‌نتیجه

برخی از توابع، مانند تابع ریاضی، نتیجه را صادر کرده و برمی‌گرداند. من به این توابع، «توابع باثمر» یا «نتیجه‌ده» می‌گویم. باقی توابع، مانند print_twice، کاری را انجام می‌دهند، ولی نتیجه‌ای را برنمی‌گردانند. به آن‌ها «توابع بی‌نتیجه» یا «وُید» می‌گویند.

زمانی‌که یک تابع نتیجه‌ده را احضار می‌کنید، تقریبا همیشه قرار است با نتیجه‌ای که برمی‌گرداند کاری را انجام دهید. به عنوان مثال آن را به یک متغیر اختصاص دهید یا به عنوان بخشی از عبارت از آن بهره ببرید:

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

زمانی‌که یک تابع را در حالت تعاملی فرا می‌خوانید، پایتون نتیجه‌ی برگردانده شده را نمایش می‌دهد:

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

ولی در یک اسکریپت اوضاع متفاوت است. اگر در یک اسکریپت شما اطلاعات برگردانده شده توسط تابع را در داخل یک متغیر ذخیره نکنید، هیچ چیزی از آن تابع دستگیرتان نمی‌شود و تمام اطلاعاتش پودر شده و به هوا می‌رود:

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

در اسکریپت بالا، ریشه‌ی دوم عدد ۵ گرفته می‌شود ولی از آنجایی که این مقدار در هیچ متغیری ذخیره نمی‌شود یا روی خروجی چاپ نمی‌گردد، این محاسبه به هیچ کاری نمی‌آید.

توابع بی‌نتیجه شاید چیزی را روی صفحهْ نمایش دهند یا تاثیرات دیگری بر روند برنامه بگذارند ولی آن‌ها نتیجه‌ای را برنمی‌گردانند. اگر سعی کنید که نتیجه را به یک متغیر اختصاص دهید، مقداری که دریافت می‌کنید None خواهد بود. به مثال زیر دقت کنید:

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

مقدار None برابر با رشته‌ی "None" نیست بلکه مقدار ویژه‌ای‌ست که نوع خاصِ خود را دارد.

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

برای بازگرداندن یک مقدار از یک تابع، ما از گزاره return در تابعِ خود استفاده می‌کنیم. به عنوان مثال می‌توانیم یک تابع بسیار ساده برای جمع دو عدد با نام addtwo به شکل زیر بنویسیم که دو عدد را به عنوان آرگیومنت گرفته و سپس پارامترهایش را با هم جمع زده و مقدارِ حاصل از این معادله را برمی‌گرداند:

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


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


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

زمانی‌که این اسکریپت اجرا می‌شود، گزاره‌ی print عدد 8 را چاپ خواهد کرد چرا که تابع addtwo با دو آرگیومنت 5 و 3 فراخوانی شده است. در درون تابع پارامترهای a و b به ترتیب 3 و 5 می‌شوند. تابعْ جمع دو عدد را محاسبه می‌کند و در متغیر محلی خود با نام added ذخیره می‌کند. منظور از «متغیر محلی» متغیری‌ست که در درون تابع تعریف می‌شود و بیرون از تابع اعتباری ندارد. سپس از گزاره‌ی return استفاده کرده و مقدار محاسبه شما را به کدِ فراخوان به عنوان نتیجه‌ی تابع باز می‌گرداند. کدِ فرخوان همان عبارتیست که تابع در آن فراخوانی شده است. سپس نتیجه‌ی برگرداننده شده به متغیر x اختصاص داده شده و در نهایت چاپ می‌شود.

## چرا از توابع استفاده می‌کنیم؟

خُرد کردن برنامه به توابع مختلف چه مزیت‌هایی دارد؟ در اینجا به دلایلی که به خاطرش ما یک برنامه را به تابع‌ها می‌شکنیم، مرور می‌کنیم:

- ساخت یک تابع جدید به شما امکان نام‌گذاری روی دسته‌ای از گزاره‌ها را می‌دهد. به این صورت برنامه‌ی شما خواناتر شده، بهتر درک می‌شود و راحت‌تر می‌تواند اشکال‌زدایی شود.
- توابع، اندازه‌ی برنامه‌ی ما را کوچکتر می‌کنند. عدم نیاز به تکرار یک سری گزاره که تحت نامی به عنوان تابع در برنامه تعریف کرده‌ایم، به ما این امکان را می‌دهد که برنامه‌های خود را بدون از دست دادن کارایی

کوچکتر کنیم. در آینده نیز اگر لازم باشد که سلسله گزاره‌ها را دستکاری کنیم، فقط کافیست که همان تابع را تغییر دهیم و نتیجه‌ی این تغییر را در سرتاسر برنامه ببینیم.

- تقسیم یک برنامه‌ی بزرگ به توابع به شما امکان اشکال‌زدایی قسمت‌های مختلف را در آن واحد می‌دهد. سپس می‌توانید این قسمت‌ها را سر هم کنید و در نهایت کل بدنه‌ی برنامه را بسازید.
- توابعی که خوب طراحی شده‌اند، می‌توانند توسط برنامه‌های زیادی مورد استفاده قرار بگیرند. زمانی‌که یک تابع را نوشتید و اشکال‌زدایی کردید، می‌توانید با خیال راحت از آن در سایر برنامه‌های خود نیز استفاده کنید.

در ادامه‌ی این کتاب، ما اغلب از توابع برای توضیح یک مفهوم استفاده می‌کنیم. بخشی از مهارت ساخت و استفاده از توابع، تبیین یک ایده یا مفهوم توسط آن است. مثلا شما یک ایده دارید: «پیدا کرد کوچکترین مقدار از یک لیست». در اینجا ما به مهارتی برای ساخت یک تابع – که این ایده را در عمل پیاده کند – احتیاج داریم. در ادامه به شما کد مربوط به چنین تابعی را – که ما min می‌خوانیمش – نشان خواهیم داد. این تابع لیستی از مقادیر را به عنوان آرگیومنت گرفته و کوچکترین مقدار آن لیست را برمی‌گرداند.

## اشکال‌زدایی

اگر از یک برنامه‌ی ویرایش متن برای نوشتن اسکریپت‌های خود استفاده می‌کنید، احتمالا به مشکل استفاده از اسپیس یا Tab برخواهید خورد. اکثر برنامه‌ها وقتی در حال نوشتن اسکریپت پایتون‌اید، این موضوع را تشخیص می‌دهند و جای Tab را با فاصله عوض می‌کنند ولی بعضی خیر. برای همین بهتر است که خودتان دست به کار شوید و به جای Tab از فاصله استفاده کنید.

اسپیس و Tab معمولا قابل مشاهده نیستند و همین موضوع اشکال‌زدایی در باب این مشکل را سخت می‌کند. بهتر است از ویرایشگری استفاده کنید که وظیفه‌ی تو رفتگی گزاره‌ها را خودش برعهده می‌گیرد.

اشکال‌زدایی – به‌خصوص زمانی‌که قسمت اشتباهی از برنامه را اجرا می‌کنید – زمان زیادی را از شما خواهد گرفت. مطمئن شوید که در حال اصلاح کدی هستید که اجرایش می‌کنید. اگر مطمئن نیستید که اشکال از کدام قسمتِ کد است می‌توانید چیزی مثل print("hello") را در آن قسمت کد قرار دهید تا مطمئن شوید جای درستی را ویرایش می‌کنید. اگر hello را در حین اجرا نمی‌بینید، پس جای غلطی را اشکال‌زدایی می‌کنید.

## واژگال فصل

**الگوریتم / Algorithm:**
یک پروسه‌ی عمومی برای حل دسته‌ای از مسائل.

**آرگیومنت / Argument:**
مقداری که به یک تابع ارسال می‌شود. این مقدار به پارامتر متناظر در تابع اختصاص پیدا می‌کند.

**بدنه / Body:**
سلسله گزاره‌های موجود در یک تابع.

**ترکیب / Composition:**
استفاده از یک عبارت به عنوان بخشی از یک عبارت بزرگ‌تر، یا یک گزاره به عنوان بخشی از یک گزاره‌ی بزرگ‌تر.

**قطعی / Deterministic:**
قطعی بودن پاسخ هر برنامه در برابر ورودی یکسان.

**نماد نقطه‌ای / Dot Notation:**
متن یا سینتکسی که توسط آن یک تابع را از یک ماژول دیگر فراخوانی می‌کنیم. به این صورت که ابتدا نام ماژول را آورده و سپس بعد از نقطه، نام تابع موجود در آن را می‌آوریم.

**جریانِ اجرا / Flow of Execution:**
ترتیبی که در آن گزاره‌ها در جریان راه‌اندازی برنامه، اجرا می‌شوند.

**تابع نتیجه‌ده / Fruitful Function:**
تابعی که یک مقدار را برمی‌گرداند.

**تابع / Function:**
سلسله‌ای از گزاره‌ها تحت عنوانِ یک نام که عملیات به‌دردبخوری را انجام می‌دهد. یک تابع ممکن است که آرگیومنت بگیرد یا نگیرد. همچنین ممکن است نتیجه‌ای را برگرداند یا برنگرداند.

**فراخوانی تابع / Function Call:**
گزاره‌ای که یک تابع را اجرا می‌کند. این گزاره شامل نام تابع و لیست آرگیومنت‌های احتمالی که تابع دریافت می‌کند می‌شود.

**تعریف تابع / Function Definition:**
گزاره‌ای که یک تابع جدید را با اختصاص نام، پارامتر و گزاره‌هایی که قرار است اجرا شوند، می‌سازد.

**فاکنشن آبجکت / Function Object:**
مقداری که توسط یک تابع ساخته می‌شود. نام تابع، یک متغیر است که به یک فانکشن آبجکت اشاره دارد.

**هدر / Header:**
اولین خط در تعریف یک تابع.

**درون‌ریزی گزاره / Import Statement:**
گزاره‌ای که فایل ماژول را می‌خواند و یک ماژول آبجکت می‌سازد.

**ماژول آبجکت / Module Object:**
مقداری که توسط گزاره‌ی import ساخته می‌شود. این مقدار دسترسی را به داده‌ها و کدهای تعریف شده در یک ماژول مقدور می‌سازد.

**پارامتر / Parameter:**
نامی که در داخل یک تابع استفاده می‌شود و به مقداری که از طریق آرگیومنت به تابع داده شده اشاره دارد.

**شبه تصادفی / Pseudorandom:**
سلسله‌ای از اعداد که به نظر تصادفی می‌آیند ولی توسط عوامل قطعی در برنامه ساخته شده‌اند.

**مقدار بازگردانده شده / Return Value:**
نتیجه‌ی یک تابع. اگر فراخوانی یک تابع به عنوان یک عبارت استفاده شده باشد، مقداری بازگردانده شده توسط تابع، مقدار عبارت خواهد بود.

**تابع بی‌نتیجه / Void Function:**
تابعی که مقداری را برنمی‌گرداند.

## تمرین‌ها

**تمرین ۴:** هدف کلیدواژه def در پایتون چیست؟

الف) یک اصطلاح که معنی آن «عجب کد باحالی‌ـه» است.

ب) نشان‌دهنده شروع یک تابع است.

پ) نشان‌دهنده این موضوع است که کدهای تو رفته‌ی بعدی برای آینده ذخیره می‌شوند.

ج) گزینه‌ی ب و پ هر دو صحیح است.

د) هیچکدام از موارد بالا.

**تمرین ۵:** برنامه‌ی زیر، چه عبارتی را چاپ خواهد کرد؟

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


def jane():
print("ABC")


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

الف) Zap ABC jane fred jane

ب) Zap ABC Zap

پ) ABC Zap jane

ج) ABC Zap ABC

د) Zap Zap Zap

**تمرین ۶:** برنامه پرداخت اضافه برای کسانی که بیشتر کار کرده‌اند را بازنویسی کرده و تابعی با نام computepay بسازید که دو پارامتر hours و rate را دریافت و سپس مقدار را محاسبه می‌کند.

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

**تمرین ۷:** برنامه نمره‌دهیِ فصل قبل را دو مرتبه بنویسید و از یک تابع به اسم computegrade استفاده کنید. این تابع بایستی که نمره را به عنوان پارامتر دریافت کرده و نتیجه را به صورت A یا B یا ... برگرداند.

```
Score Grade
> 0.9 A
> 0.8 B
> 0.7 C
> 0.6 D
<= 0.6 F
Program Execution:
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
```

# فصل ۵

# تکرار

## به‌روز رسانی متغیرها

یک الگوی رایج در گزاره‌های گمارشی، دستور آپدیت‌کردن یک متغیر است؛ به صورتی که مقدار جدید یک متغیر، به مقدار قبلی آن وابسته خواهد بود.

```
x = x + 1
```

گزاره‌ی بالا به این معنی‌ست که مقدار فعلی متغیر x را بگیر، عدد ۱ را به آن اضافه کن و در نهایت مقدار x را با مقدار جدید که x + 1 باشد، جایگزین کن.

اگر سعی کنید که یک متغیری که وجود خارجی ندارد را به‌روز کنید، خطایی دریافت خواهید کرد. ولی چرا خطا؟ دلیلش این است که پایتون ابتدا قسمت راست معادله را بررسی و حساب می‌کند؛ سپس مقدار جدید را به قسمت چپ معادله نسبت می‌دهد. وقتی x برای محاسبه در کار نباشد، خطا صادر می‌کند:

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

قبل از اینکه بتوانید یک متغیر را به‌روز کنید، بایستی که آن را تعریف کرده باشید. یا به عبارت تخصصی‌تر، یک مقدار اولیه به آن اختصاص داده شده باشد. مثلا در گزاره‌های زیر، ابتدا x را تعریف کرده‌ایم و مقدار ۰ را به آن اختصاص داده‌ایم:

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

به‌روز رسانی یک متغیر با اضافه کردن عدد یک، «افزونش» یا «افزودن پله‌ای» خوانده می‌شود. اگر مقدار عدد یک را از متغیر کم کنید به آن «نزول» یا «کاستن پله‌ای» می‌گوییم.

## گزاره while

کامپیوترها اغلب برای انجامِ خودکار کارهای تکراری استفاده می‌شوند. تکرارِ وظایف مشابه، بدون خطا کردن، کاری‌ست که انسان به خوبی از پسش برنمی‌آید، ولی کامپیوتر چرا، برمی‌آید و خیلی خوب هم برمی‌آید! تکرار در همه‌کاری بسیار رایج است، به همین خاطر پایتون به چندین ویژگی مختلف، برای سهولت در آن مجهز شده است.

یک نمونه از تکرار، استفاده از گزاره while است. به مثال زیر نگاه کنید؛ یک برنامه‌ی ساده‌ای که از عدد پنج تا به صفر می‌شمارد و سپس عبارت !Blastoff را چاپ می‌کند:

```
n = 5
while n > 0:
    print(n)
    n = n - 1
print('Blastoff!')
```

گزاره‌ی while را همانگونه که انگلیسی می‌خوانید (به معنای «تا زمانی که») می‌توانید استفاده کنید. مثال بالا می‌گوید: تا زمانی‌که n از ۰ بزرگتر است، مقدار n را چاپ و سپس عدد یک را از آن کسر کن. زمانی‌که به ۰ رسیدی، از حلقه‌ی while خارج شو و کلمه‌ی !Blastoff را چاپ کن.

حالا اگر بخواهیم جریان اجرای گزاره while را به صورت رسمی‌تر بیان کنیم، سه مرحله‌ی اصلی در پیش رو داریم:

I. محاسبه‌ی شرط برای دریافت True یا False. به عبارت ساده‌تر، گزاره‌ی while شرط را وارسی کرده و می‌بینید که نتیجه‌ی آن «صحیح» یا «غلط» است.
II. اگر محاسبه‌ی بالا False یا «غلط» بود، از گزاره‌ی while خارج شده و به اجرای گزاره‌ی بعد از بدنه‌ی آن می‌پردازد.
III. اگر محاسبه‌ی اولیه True یا «صحیح» بود، بدنه‌ی while اجرا می‌شود و دو مرتبه به سراغ محاسبه‌ی شرط گزاره‌ی while (مرحله اول) می‌رود.

ما به این جریان اجرا، loop یا حلقه می‌گوییم. ولی چرا حلقه؟ چون در مرحله‌ی سوم، جریان اجرا دور زده و باز به مرحله‌ی اول باز می‌گردد. به هر بار که بدنه‌ی این حلقه را اجرا می‌کنیم یک «تکرار» می‌گوییم. در مثال بالا حلقه‌ای داشتیم که در آن پنج تکرار انجام شد. به عبارت ساده‌تر حلقه‌ی ما پنج بار اجرا شد.

بدنه‌ی حلقه بایستی که مقدار یک یا چند متغیر را تغییر دهد؛ به این خاطر که در نهایت نتیجه‌ی شرط (مرحله‌ی اول) «غلط» شود، و برنامه بتواند از بدنه‌ی حلقه خارج شده و به حلقه پایان بخشد. ما به متغیری که با هر بار اجرای حلقه، تغییر می‌کند، و کنترل جریان حلقه را در دست دارد، «متغیر تکرار» می‌گوییم. اگر «متغیر تکرار» در کار نباشد، این حلقه برای همیشه تکرار خواهد شد و نتیجه‌ی آن یک «حلقه بی‌نهایت» خواهد بود.

## حلقه‌های بی‌نهایت

برای یک برنامه‌نویس، دستورالعمل یک شامپو هم سرگرم‌کننده است. چرا؟ دستور العمل شامپو می‌گوید: ۱) شامپو را به سرتان بزنید، ۲) آب بکشید، ۳) تکرار کنید. ولی اثری از متغیرِ تکرار در کار نیست که شخص بداند بایستی چند بار این عمل را تکرار کند.

در حالت شمارش معکوس، ما می‌دانیم که n یک عدد متناهی است، در نتیجه با کم شدن مقدار از آن، در نهایت به صفر رسیده و حلقه به پایان می‌رسد. در حالت‌هایی که متغیر تکرار در کار نیست، حلقه، یک حلقه‌ی بی‌نهایت خواهد بود. البته حلقه‌ی بی‌نهایت می‌تواند با وجود متغیر تکرار نیز اتفاق بیفتد. مثلاً فرض کنید که در مثال پیشین به اشتباه به جای کم کردن مقدار n، به آن اضافه کنید. در این سناریو n هیچ‌وقت به عدد صفر نخواهد رسید و حلقه برای همیشه و تا زمانی‌که کامپیوتر ظرفیتش را دارد، تکرار خواهد شد.

## حلقه‌های بی‌نهایت و break

گاهی تا زمانی‌که به اواسط بدنه‌ی حلقه نرسید، نمی‌دانید که باید این حلقه را تمام کنید یا نه. در این زمان می‌توانید که یک حلقه‌ی بی‌نهایت را از عمد بنویسید و سپس با استفاده از گزاره break از حلقه بیرون بپرید.

حلقه‌ی زیر، یک حلقه‌ی بی‌نهایت است چرا که در جلوی گزاره‌ی while عبارت منطقی بی‌تغییر True نوشته شده و این شرط همیشه «صحیح» است:

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

اگر کد بالا را اجرا کنید، خواهید فهمید که حلقه‌ی بی‌نهایت تا زمانی‌که برنامه را متوقف نکنید، تکرار خواهد شد. باید راهی برای خاتمه‌دادن به برنامه پیدا کنید، یا به دکمه‌ی پاور کامپیوتر متوسل شوید. به این دلیل که شرط while همیشه «صحیح» است این برنامه تا همیشه اجرا خواهد شد.

پس به نظر می‌رسد که استفاده از عبارت منطقیِ بی‌تغییر در حلقه، کار بیهوده‌ای است. با این حال شما می‌توانید حلقه‌های مفید با استفاده از این الگو بسازید. ولی

چگونه؟ با استفاده از کدی که جریانِ اجرا را از حلقه خارج کند. شما می‌توانید از break، البته با دقت و به جا، استفاده کرده و یک حلقه‌ی بی‌نهایت را بشکنید.

برای نمونه فرض کنید که شما می‌خواهید که کاربر مدام ورودی به برنامه بفرستد و این کار تنها زمانی به پایان برسد که کاربر عبارت done را تایپ کند. کد زیر را ببینید:

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


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

شرطِ اصلیِ حلقه True است و به این خاطر که یک مقدار ثابت است، حلقه‌ی ما بی‌نهایت خواهد بود. ولی زمانی‌که شرط داخل بدنه‌ی حلقه درست از آب در بیاید، گزاره‌ی break اجرا شده و ما از حلقه خارج می‌شویم.

در برنامه‌ی بالا، پایتون مدام از کاربر ورودی می‌خواهد. اگر کاربر عبارت done را تایپ نماید، با اجرا شدن گزاره‌ی break جریان اجرا به خارج از حلقه پرش می‌کند. در غیر این صورت، برنامه هر چیزی که کاربر تایپ کند را تکرار می‌کند:

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

این روشِ نوشتن حلقه‌های while بسیار رایج است، چرا که شما می‌توانید در هر جای حلقه که خواستید، شرط لازم برای ادامه‌ی آن را تعریف کنید. همچنین می‌توانید به

برنامه به جای اینکه بگویید «آنقدر حلقه را ادامه بده تا این اتفاق بیفتد» بگویید «وقتی این اتفاق افتاد، در جا حلقه را تمام کن».

## پایان دادن به تکرار با استفاده از continue

گاهی در حین تکرار حلقه می‌خواهید که تکرارِ جاری را تمام کرده و سریع به سراغ تکرار بعدی بروید، بدون اینکه گزاره‌های بعدی موجود در حلقه، در آن تکرار اجرا شوند. اینجاست که از گزاره continue استفاده می‌کنیم.

بهتر است قضیه را با یک مثال روشن‌تر کنیم. کد زیر را در نظر بگیرید. این کد، ورودی کاربر را گرفته و تا زمانی‌که کاربر done را وارد کند، آن‌ها را چاپ می‌کند. ولی یک استثنا این وسط وجود دارد. اگر کاربر رشته‌ای وارد کند که با # شروع شده باشد، برنامه آن را چاپ نخواهد کرد و به سراغ دریافت مجدد ورودی از کاربر می‌رود.

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


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

برنامه را بنویسید و اجرا کنید. یک نمونه از اجرای آن را در پایین آورده‌ایم:

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

تمام خطوط غیر از آن خطی که با علامت هش شروع شده بود، چاپ شدند. دلیلش این است که آن رشته، شرط موجود در بدنه‌ی حلقه را محقق کرده و گزاره‌ی continue اجرا می‌شد. با اجرا شدن continue، جریان اجرا بدون اجرا کردن ادامه‌ی گزاره‌های حلقه، به سراغ تکرار بعدی می‌رود. به همین خاطر در آن مورد خاص، برنامه از اجرای گزاره‌ی print باز ماند.

## تعریف حلقه‌ها با for

گاهی لازم است که بین دسته‌ای و محتویاتش دور بزنیم؛ مثلا بین لیستی از کلمه‌ها، یا خطوط در یک فایل، و یا شماره‌ها در یک فهرست. زمانی‌که لیستی از چیزی‌هایی داریم که لازم است بین‌شان بگردیم، از یک حلقه‌ی معین با استفاده از گزاره‌ی for بهره می‌بریم. چرا حلقه «معین»؟ اگر یادتان باشد ما از گزاره‌ی while برای حلقه‌های نامعین استفاده می‌کردیم، چرا که حلقه while به سادگی تا زمانی‌که شرطِ اجرای آن «غلط» شود، تکرار می‌شود، ولی for در بین یک دسته‌ی «مشخص» و «معین» از چیزها حلقه می‌زند. به عبارتی آنقدر این حلقه را تکرار می‌کند که بازه‌ی مورد نظر در لیست را گشته باشد.

متن یا سینتکس حلقه‌ی for مشابه با حلقه‌ی while است. به صورت کلی ما گزاره‌ی for و بدنه‌ی آن حلقه را با یک تورفتگی داریم:

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

مثال بالا را در نظر بگیرید. در زبان پایتونی متغیر friends یک لیستی‌ست که شامل سه رشته می‌شود. حلقه‌ی for بین این لیست دور زده و بدنه‌ی حلقه را برای هر کدام از این سه رشته به صورت جداگانه اجرا می‌کند. در زیر خروجی برنامه‌ی بالا را مشاهده کنید:

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

ترجمه‌ی تحت‌الفظی while دقیقا همان چیزی بود که در پایتون مورد استفاده قرار می‌گرفت، ولی برای حلقه‌ی for اوضاع فرق می‌کند. بیایید حلقه‌ی مثال قبل را با هم به زبان آدمیزاد شرح دهیم: «گزاره‌های موجود در بدنه‌ی حلقه را برای هر کدام از دوستان موجود در لیست friends اجرا کن».

در کد مربوط به حلقه‌ی for کلمه‌های for و in کلمات رزرو شده‌اند و کلمات friend و friends متغیراند:

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

در این موردِ خاص، friend «متغیر تکرار» برای حلقه‌ی for محسوب می‌شود. متغیر friend در هر تکرار تغییر کرده و حلقه for را تا انتها کنترل می‌کند. به این صورت که با رسیدن متغیر به آخرین آیتم موجود در لیست، حلقه پایان می‌پذیرد. در مثال بالا، متغیر تکرار که همان friend باشد، پشت سر هم و به ترتیب سه رشته‌ی موجود در متغیر friends را در حلقه اجرا می‌کند.

## الگوی حلقه

اغلب ما از حلقه‌ی for یا while استفاده کرده تا روی لیستی از آیتم‌ها یا محتوای یک فایل کار کنیم. مانند زمانی‌که به دنبال بزرگترین یا کوچکترین مقدار در داده‌ای که اسکن کرده‌ایم می‌باشیم.

به طور کلی، حلقه‌ها از سه جزء تشکیل شده‌اند:

- تعریف یک یا چند مقدار، قبل از شروع حلقه؛
- اجرای محاسبات روی هر کدام از آیتم‌ها، و تغییر احتمالی متغیرهای موجود در بدنه‌ی حلقه؛
- بررسی نتایج متغیرها زمانی‌که حلقه به پایان رسید.

در ادامه ما لیستی از شماره‌ها را – برای نشان دادن مفهوم و ساختار الگوی حلقه‌های رایج – برایتان آورده‌ایم.

## حلقه‌هایی که می‌شمارند و جمع می‌زنند

برای نمونه، اگر قرار باشد شماره‌ی آیتم‌های موجود در یک لیست را بشماریم، ما از حلقه‌ی for استفاده می‌کنیم:

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

قبل از اینکه حلقه شروع شود، مقدار متغیر count را صفر قرار می‌دهیم. سپس یک حلقه for می‌نویسیم. این حلقه تک‌تک آیتم‌های لیست را دور زده و به ازای هر کدام از آن‌ها یک بار گزاره‌ی موجود در بدنه را اجرا می‌کند. در اینجا نامِ متغیر تکرار itervar است که کنترل حلقه را در دست دارد.

در بدنه‌ی حلقه ما عدد یک را به مقدار فعلی متغیر count به ازای هر کدام از مقادیر موجود در لیست اضافه می‌کنیم. زمانی‌که حلقه اجرا می‌شود، مقدار count برابر با آخرین مقداری‌ست که داشته است. مثلا در اجرای اول count برابر با صفر است ولی در تکرار دوم حلقه با توجه به اجرای اول، برابر با یک است که با توجه به گزاره‌ی موجود در بدنه‌ی حلقه به دو تبدیل می‌شود. به همین ترتیب با هر بار اجرای حلقه، مقدار count یک عدد افزایش پیدا می‌کند.

در نهایت مقدار count برابر با مجموع تعداد آیتم‌های موجود در لیست خواهد بود. این حلقه، خروجی خاصی را چاپ نمی‌کند، ولی مقداری را در نهایت به count اختصاص می‌دهد که هدف ما از ایجاد و ساخت آن حلقه بوده است.

در مثال زیر حلقه‌ای را داریم که مجموع اعداد موجود در لیست را محاسبه کرده و سپس چاپ می‌کند:

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

در این حلقه ما از متغیر تکرار استفاده کرده‌ایم. به جای اضافه کردن عدد یک به count – به مانند مثال قبل – ما عدد موجود در لیست (۳،۴۱، ۱۲، ۹ و...) را به مقدار کل، در حین اجرای آن تکرار اضافه نموده‌ایم. به عبارتی متغیر total حاوی مقداری برابر با جمع مقادیر تا آنجای کار می‌شود. قبل از اینکه حلقه شروع شود مقدار total صفر بود، چرا که هنوز مقدار اختصاص داده شده اولیه را در خود داشت و عملیاتی روی آن صورت نگرفته بود. در حین اجرای حلقه، مقدار total برابر با حاصل جمع مقدارهای موجود در لیست می‌شود و در پایان، مقدار total برابر با جمع تمامی مقادیر موجود در لیست خواهد بود.

همینطور که حلقه اجرا می‌شود، total جمع عناصر موجود در معادله را حساب می‌کند؛ یک متغیر که به این صورت مورد استفاده قرار می‌گیرد، accumulator یا انباشتگر خوانده می‌شود.

نه حلقه‌ی شمارش و نه حلقه‌ی انباشتگر، در آینده به کار شما نخواهند آمد چرا که توابع توکاری‌شده‌ای به اسم‌های len() و sum() به ترتیب شماره‌ی آیتم‌ها، و جمع آیتم‌های موجود در یک لیست را به ما می‌دهند و شما نیاز به حلقه‌های فوق ندارید.

## حلقه برای یافتن مقدار بیشینه و کمینه

برای پیدا کردن بزرگترین مقدار در یک لیست یا سلسله‌ای از اعداد ما حلقه‌ی زیر را ساخته‌ایم:

```
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)
```

زمانی‌که برنامه اجرا می‌شود، خروجی زیر ظاهر خواهد شد:

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

متغیر largest عبارت است از بزرگترین مقداری که تا اینجای کار به آن رسیده‌ایم. قبل از حلقه، ما مقدار largest را None قرار می‌دهیم. None یک مقدار ویژه‌ی غیرقابل تغییر است که می‌تواند به متغیرها اختصاص داده شود. این مقدار می‌گوید که متغیر «خالی» است.

قبل از شروع حلقه، بزرگترین مقداری که ما تا آنجای کار با توجه به جریان اجرا دیده‌ایم، None است. در حقیقت مقدار دیگری غیر از None را ندیده‌ایم. در خروجی بالا مشخص است که درست زمانی‌که حلقه اجرا می‌شود، مقدار ۳ جایگزین مقدار اولیه‌ی متغیر largest می‌شود.

بعد از اولین تکرار، largest دیگر None نیست در نتیجه قسمت دوم عبارت منطقیِ ترکیبی – که مقایسه‌ی itervar > largest را بررسی می‌کند – تنها زمانی درست از آب در می‌آید که به عددی بزرگتر از مقدار فعلی largest برخورد کند. وقتی ما در جریان اجرا به مقداری بزرگتر از largest برخورد می‌کنیم، مقدار largest تغییر کرده و آن مقدار جدید را به خود می‌گیرد. در مثال بالا می‌بینید که مقدار largest از ۳ به ۴۱ و سپس به ۷۴ تغییر پیدا کرد.

در پایان حلقه، ما یک به یک مقادیر را بررسی کرده‌ایم. largest حاوی بزرگترین مقدار موجود در لیست خواهد بود.

برای محاسبه‌ی کمترین مقدار موجود در لیست، کد قبلی تنها یک تغییر کوچک می‌کند:

```
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)
```

به شکل مشابه smallest حاوی کوچکترین عددِ در جریان اجراست. زمانی‌که حلقه پایان می‌پذیرد، این متغیر حاوی کوچکترین عدد کل لیست خواهد بود.

به مانند شمارش و انباشتگر، توابع توکاری شده‌ای در پایتون با نام‌های ()max و ()min وجود دارد که به ترتیب بزرگترین و کوچکترین مقدار یک لیست را برمی‌گردانند.

کد زیر یک نسخه‌ی ساده شده از تابع توکاری شده‌ی ()min در پایتون است:

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

تابع بالا، همان کد مربوط به محاسبه‌ی کوچکترین عدد بود که گزاره‌های print آن حذف شده است. اکنون این تابع کاری را که تابع توکاری شده python برای تشخیص کوچکترین عدد موجود در لیست، انجام می‌دهد را اجرایی می‌کند.

## اشکال‌زدایی

به محضی که شروع به نوشتن برنامه‌های بزرگتر می‌کنید، تازه می‌بینید که گاهی اشکال‌زدایی زمان بیشتری از نوشتن خودِ کد از شما می‌گیرد. کدهای بیشتر، به معنی احتمالِ بیشترِ خطا و احتمال مخفی شدن باگ‌های بیشتر در برنامه است.

یکی از راه‌های صرفه‌جویی در زمان اشکال‌زدایی استفاده از تکنیک دو نیم کردن است. به عنوان مثال فرض کنید که ۱۰۰ خط کد دارید. اگر قرار باشد برای بررسی کد، خط به خط جلو بروید برای بررسیِ کاملِ کد، شما صد مرحله در پیش رو خواهید داشت.

حالا روش دیگری را استفاده می‌کنیم. شما ابتدا مساله را به نصف می‌شکنید. به وسط برنامه یا یک مکان منطقی در همان حوالی می‌روید و یک مقدار میانی را با استفاده از گزاره‌ی print چاپ می‌کنید. سپس برنامه را اجرا می‌کنید.

با بررسی خروجی گزاره print بررسی می‌کنید که مشکل از نیمه‌ی اول کد بوده یا خیر. چگونه؟ اگر مقدار میانی یک میزانِ اشتباه را برگرداند، خب مشکل از بخش اول کد است.

هر بار که به طریق مشابه کد را بررسی کنید، مقدار خطوط مشکوک، به نصف تقلیل پیدا می‌کند. بعد از شش مرحله، احتمالا به یک یا دو خط کد رسیده‌اید. حداقل در تئوری!

در عمل، وسط یک برنامه همیشه واضح نیست و نمی‌شود که یک قطعه کد در بین آن جا داد و کد را بررسی کرد. در اصل شمارش خطوط و پیدا کردن نقطه‌ی وسط، کار معقولی به نظر نمی‌رسد. در عوض به دنبال قطعه‌هایی از کد که احتمالا خطا در آن‌ها خف کرده بگردید و همان‌ها را بررسی کنید. جایی که احساس می‌کنید خطا از آن ناشی شده را انتخاب کرده و سپس با استفاده از ابزارهایی مثل print سعی کنید از مشکل سر در بیاورید.

## واژگان فصل

**انباشتگر / Accumulator:**
متغیری‌ست در یک حلقه برای افزایش یا انباشتن یک برآیند یا نتیجه.

**شمارنده / Counter:**
متغیری‌ست در یک حلقه که برای شمارش تعداد دفعاتی که اتفاقی می‌افتد، به کار می‌رود. ما «شمارنده» را در ابتدا با صفر مقداردهی می‌کنیم و سپس به آن مقداری را – زمانی‌که می‌خواهیم، چیزی را بشماریم – اضافه می‌کنیم.

**نزول / Decrement:**
به‌روز رسانی‌ای که مقدار یک متغیر را کاهش می‌دهد.

**مقداردهی اولیه / Initialize:**
گزاره‌ای که در آن به یک متغیر، مقدار اولیه داده می‌شود. این متغیر در ادامه به‌روزرسانی خواهد شد.

**صعود / Increment:**
به‌روز رسانی‌ای که مقدار یک متغیر را افزایش می‌دهد.

**حلقه بی‌نهایت / Infinite Loop:**
حلقه‌ای که شرط آن هیچگاه «غلط» نمی‌شود؛ به عبارتی اگر آن حلقه تنها بر مبنای شرط، تکرار شود، آن حلقه برای همیشه، و تا زمان بستن برنامه، تکرار خواهد شد.

**تکرار / Iteration:**
اجرای تکراری دسته‌ای از گزاره‌ها را Iteration یا تکرار می‌گویند. تکرار می‌تواند هم توسط یک تابع با فراخوانی خودش، هم در یک حلقه صورت پذیرد.

## تمرین‌ها

**تمرین ۱:** برنامه‌ای بنویسید که تا زمانی‌که کاربر کلمه‌ی "done" را تایپ کند، به صورت مکرر از او عدد در خواست کند. سپس با ورود "done" سه مقدار «مجموع اعداد»، «تعداد ورودی‌های عددی»، «میانگین اعداد» را چاپ کند. برنامه‌ی شما باید با استفاده از try و except در زمانی‌که کاربر ورودی غیرعددی وارد می‌کند به او پیام خطای «Invalid input» را نشان دهد و به سراغ درخواست مجدد عدد برود. به خروجی برنامه‌ی نمونه‌ی ما دقت کنید:

```
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:** برنامه‌ی دیگری بنویسید که مثل برنامه‌ی بالا یک لیست از اعداد را درخواست کرده، سپس «بزرگ‌ترین» و «کوچک‌ترین» آن‌ها را به جای مقدار «میانگین» چاپ کند.

# فصل ۶

# رشته‌ها

## رشته چیست؟

یک رشته، زنجیره‌ای از کاراکترهاست. شما می‌توانید با استفاده از عملگر قلاب به هر کاراکتر در آنِ واحد دسترسی داشته باشید:

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

گزاره‌ی دوم، کاراکتری که در موقعیت 1 از متغیر fruit قرار دارد را بیرون می‌کشد و آن را به متغیر letter نسبت می‌دهد.

عبارت داخل براکت‌ها را ایندکس یا شاخص می‌گوییم. شاخص نشان می‌دهد که کدام کاراکتر در سلسله‌ی موجود را می‌خواهید.

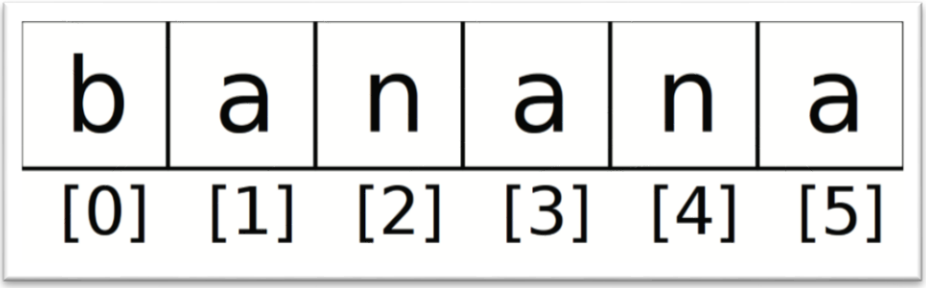
ولی شاید انتظار شما از مقدار موجود در letter چیز دیگری باشد:

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

به صورت شهودی اولین کاراکتر موجود در رشته banana حرف b است، و نه a. ولی چرا شاخص 1 حرف a را نشان می‌دهد؟ در پایتون، شاخص از ابتدای یک رشته شروع و اولین حرف از رشته با شاخص 0 مشخص می‌شود.

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

در نتیجه b صفرمین حرف از رشته banana به حساب می‌آید و a یکمین و n دومین و به همین ترتیب.




شما می‌توانید از هر عبارتی، شامل متغیرها و عملگرها، به عنوان شاخص استفاده کنید، با این استثنا که مقدار شاخص بایستی یک عدد صحیح باشد وگرنه شما چیزی شبیه به این خطا را دریافت می‌کنید:

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

## دریافت طول یک رشته با استفاده از len

یک تابع توکاری شده با اسم len در پایتون وجود دارد که مقدار عددیِ تعداد کاراکترهای موجود در یک استرینگ را برمی‌گرداند:

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

با استفاده از تابع len چطور می‌شود آخرین کاراکتر یک رشته را استخراج کرد؟ شاید بخواهید که از آن به این صورت استفاده کنید:

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

ولی دلیل اینکه خطای IndexError گرفتیم چیست؟ حرفی با شاخص 6 در رشته banana وجود ندارد. از آنجایی که پایتون شمارش کاراکترهای رشته را با عدد 0 آغاز می‌کند، ما در این رشته شاخص‌های 0 تا 5 را داریم. به عبارتی برای دریافت آخرین کاراکتر از یک رشته، بایستی عدد یک را از اندازه‌ی طول آن رشته کم کنیم:

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

همچنین می‌توانید از شاخص‌های منفی به منظور شمارش از آخر به اول استفاده کنید. عبارت fruit[-1] آخرین حرف و عبارت fruit[-2] حرف یکی مانده به آخر را استخراج می‌کند.

## پیمایش در رشته با یک حلقه

بسیاری از محاسبات، با استفاده از پردازش تک‌تک کاراکترهای موجود در یک رشته صورت می‌پذیرد. اغلب از ابتدا آغاز می‌شود، هر کاراکتر را به صورت جداگانه انتخاب می‌کند، کاری روی آن انجام می‌دهد و به سراغ کاراکتر بعدی تا انتهای رشته می‌رود. به این الگوی پردازشی، «پیمایش» می‌گویند. یکی از راه‌های نوشتن یک پیمایش استفاده از حلقه while است:

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

این حلقه، پیمایشی را در رشته آغاز کرده و هر حرف را به صورت مستقل در یک خط به نمایش می‌گذارد. شرطِ حلقه index < len(fruit) است، در نتیجه به محضی که index مساوی با طول رشته شد، شرط غلط از آب در آمده و بدنه‌ی حلقه اجرا نخواهد شد. آخرین کاراکتری که عملیات در این حلقه روی آن انجام می‌شود، با شاخص len(fruit)-1 خواهد بود که خب آخرین کاراکتر رشته نیز محسوب می‌شود.

**تمرین ۱:** یک حلقه while بنویسید که از آخرین کاراکتر موجود در یک رشته شروع کرده و عکسِ حلقه‌ی بالا، به سمت ابتدای حلقه حرکت کند و هر حرف را در یک خط جداگانه به مانند حلقه‌ی بالا چاپ می‌کند. با این تفاوت که چاپ کردنِ حروف از حرف آخر شروع شود و به اول برسد.

یک راه دیگر برای نوشتن یک پیمایش استفاده از حلقه for است:

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

هر باری که حلقه اجرا می‌شود، کاراکتر بعدی به متغیر char نسبت داده خواهد شد. این حلقه تا زمانی‌که کاراکتری باقی نمانده باشد، ادامه پیدا می‌کند.

## قاچ‌هایی از رشته

یک بخش از یک رشته را «قاچ» یا «قطعه» می‌گوییم. انتخاب یک قطعه از رشته، مشابه با انتخاب یک کاراکتر است. به مثال زیر نگاه کنید:

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

عملگر، قسمتی از رشته که از کاراکتر nام شروع شده و به کاراکتر mام می‌رسد را برمی‌گدارند. با این قاعده که حاصلْ شاملِ کاراکتر nام می‌شود، ولی شامل کاراکتر mام نخواهد شد.

اگر شما شاخص اول (قبل از دو نقطه) را از قلم بیندازید، قطعه از ابتدای رشته شروع خواهد شد. اگر شاخص دوم (بعد از دو نقطه) را از قلم بیندازید، قطعه تا پایان رشته خواهد بود:

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

اگر اولین شاخص، بزرگتر و یا برابر با شاخص دوم بود، حاصلْ یک رشته‌ی خالی می‌شود که با دو علامت نقل قول مشخص خواهد شد:

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

یک رشته‌ی خالی، شامل هیچ کاراکتری نشده و طول آن 0 خواهد بود، ولی در سایر موارد شبیه به یک رشته‌ی معمولی می‌باشد.

**تمرین ۲:** فرض کنید که fruit یک رشته است. fruit[:] چه معنی‌ای را می‌دهد؟

## رشته‌ها قابل تغییر نیستند

شاید وسوسه شده باشید که با استفاده از عملگر گمارش (=) یک کاراکتر در رشته را درآورده و تغییر دهید. مثلا:

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

آبجکت در این حالت یک رشته است و آیتم، کاراکتری‌ست که شما می‌خواهید آن را به رشته اختصاص دهید. در اینجا آبجکت مثل یک مقدار است؛ ولی در آینده معنای دقیق‌تری از آن را برای‌تان خواهیم گفت. یک آیتم، یکی از مقادیر موجود در یک زنجیر به حساب می‌آید.

دلیل خطا در مثال بالا، این است که رشته‌ها تغییر ناپذیرند، به این معنی که شما یک رشته‌ی موجود را نمی‌توانید تغییر دهید. بهترین کاری که می‌توانید انجام دهید ساخت یک رشته‌ی جدید با استفاده از رشته‌ی قبلی و جایگزینی کاراکتر با بهره‌گیری از شاخص‌هاست:

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

در این مثال ما حرف اول را با استفاده از قطعه کردن greeting و چسباندن یک کاراکتر به آن، تغییر دادیم. به یاد داشته باشید که رشته‌ی اصلی همان Hello, World سابق است.

## حلقه‌زنی و شمارش

برنامه‌ی زیر، تعداد دفعالتی که حرف a در رشته ظاهر شده را حساب می‌کند:

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

در این برنامه ما الگوی دیگری از محاسبه، به اسم «شمارش» را به شما نشان دادیم. متغیر count با ۰ مقداردهی اولیه شده و سپس هر بار که a در رشته پدیدار شد، یک عدد به آن اضافه گردید. زمانی‌که برنامه از حلقه خارج می‌شود، count حاوی نتیجه‌ای از تعداد aهای موجود در رشته خواهد بود.

**تمرین ۳:** این کد را در تابعی به اسم count قرار دهید که دو آرگیومنت دریافت می‌کند. آرگیومنت‌ها یکی رشته و دیگری حرفی که قرار است تعداد تکرارش را تابع پیدا کند و برگرداند.

## عملگر in

کلمه‌ی in یک عملگر بولی است که دو رشته را می‌گیرد و True را اگر رشته‌ی اول به عنوان قسمتی از رشته‌ی دوم باشد، برمی‌گرداند.

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

## مقایسه رشته

عملگرهای مقایسه روی رشته‌ها نیز عمل می‌کنند. اگر می‌خواهید ببینید که دو رشته با هم برابرند:

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

بقیه‌ی عملگرهای مقایسه برای مرتب کردن کلمات بر اساس الفبا می‌توانند کمک بزرگی به شما باشند:

```
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.')
```

رفتار مردم در مقایسه با پایتون در خصوص حروف بزرگ و کوچک متفاوت است. در پایتون تمام حروف بزرگ پیش از تمام حروف کوچک می‌آید:

```
Your word, Pineapple, comes before banana.
```

یک راه‌حل رایج برای این مساله استفاده از قالب استاندارد است. مثلا می‌توانید تمام حروف را به حروف کوچک تبدیل کنید و سپس عملیات را روی آن انجام دهید.

## متدهای رشته (string)

رشته‌ها یک مثال از آبجکت‌های پایتون به شمار می‌روند. یک آبجکت (شئ) شامل داده و متدها می‌شود. متد، توابعی است که در آبجکت به صورت توکاری‌شده قرار داده شده‌اند و برای تمام نمونه‌های آن آبجکت در دسترسند.

پایتون تابعی با نام dir دارد که لیست متدهای موجود در یک آبجکت را به نمایش می‌گذارد. تابع type نوع یک آبجکت را نشان می‌دهد و تابع dir متدهای موجود در آن آبجکت را.

```
>>> 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(...)
    S.capitalize() -> str


    Return a capitalized version of S, i.e. make the first
character
    have upper case and the rest lower case.
>>>
```

تابع dir لیست متدها را نشان می‌دهد و در کنارش تابع help اطلاعات و اسناد ساده‌ای از یک متد را به نمایش می‌گذارد. اطلاعات بیشتر از متدهای رشته را می‌توانید از اینجا بخوانید:

https://docs.python.org/3.5/library/stdtypes.html#string-methods

فراخوانی یک متد، به مانند فراخوانی یک تابع است، یعنی متد هم آرگیومنت را می‌گیرد و مقداری را برمی‌گرداند؛ ولی متنِ این فراخوانی متفاوت است. برای فراخوانی متد، نامِ آن را به نام متغیر با یک نقطه بینشان می‌چسبانیم.

به عنوان مثال متد upper یک رشته را می‌گیرد و رشته‌ای جدید که تمام حروفش بزرگ است را برمی‌گرداند.

به جای استفاده از متن تابع، که به صورت upper(word) است، از متن مخصوص متد یعنی word.upper() استفاده می‌کنیم.

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

این قالب که به آن «نمادِ نقطه» می‌گوییم، نام متد، upper، و نام رشته را مشخص کرده و متد را روی آن (در مثال ما متغیر word) اعمال می‌کند. پرانتزهای خالی به این معنی‌ست که این متد هیچ آرگیومنتی را دریافت نمی‌کند.

فراخوانی یک متد، invocation یا «احضار» خوانده می‌شود. در این مثال، ما می‌گوییم که upper را روی word احضار کردیم.

برای مثال، یک متد رشته به اسم find وجود دارد که جایگاه یک رشته را در یک رشته‌ی دیگر جستجو می‌کند:

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

در این نمونه، ما find را روی word احضار کرده و حرفی که در پی آن بودیم را به صورت پارامتر برایش فرستادیم.

متد find قادر به پیدا کردنِ زیررشته‌ها به مانند کاراکترهاست:

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

این متد می‌تواند آرگیومنت دوم را نیز دریافت کرده تا بداند که جستجو را از چه مکانی شروع کند:

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

یک کار رایج از بین بردن فاصله‌های خالی (فاصله یا tab یا خطوط جدید) از ابتدا و انتهای یک رشته با استفاده از متد strip است:

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

برخی متدها مثل startswith مقدارهای بولی را برمی‌گردانند:

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

متد startswith به بزرگی و کوچکی حروف حساس است، به همین خاطر شاید بد نباشد که کل یک خط را گرفته و حروفش را به حروف کوچک تبدیل کرده و سپس بررسی را روی آن انجام دهید:

```
>>> 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 احضار شده و سپس از متد startswith استفاده کرده‌ایم. به این صورت متد startswith روی رشته‌ی جدید که تمام حروفش کوچک است عملیات را انجام می‌دهد. اگر مراقب ترتیبِ متدها باشید، حتی می‌توانید چند متد را به صورت ترکیبی در تنها یک عبارت احضار کنید:

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

**تمرین ۴:** یک متد رشته به نام count وجود دارد که شبیه به تابعی که در مثال قبل نوشتیم عمل می‌کند. اسناد مرتبط با این متد را از لینک زیر بخوانید و سپس یک احضاریه بنویسد که تعداد تکرارِ حرف a در banana را برگرداند.

https://docs.python.org/3.5/library/stdtypes.html#string-methods

## تجزیه تحلیل (پارس کردن) رشته‌ها

احتمالاً برای شما هم زیاد پیش می‌آید که به داخل یک رشته نگاه کرده و به دنبال یک زیر-رشته بگردید. مثلا فرض کنید که یک سری خط را داریم که به صورت زیر قالب‌بندی شده‌اند:

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

حالا شاید بخواهیم که قسمت دوم آدرس را از هر خط جدا کرده و برداریم (در اینجا: uct.ac.za). ما می‌توانیم از متد find و روش قاچ زدن در پایتون، استفاده کنیم.

```
>>> 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 یک نسخه از متد find که به ما اجازه مشخص کردن جایگاه کاراکتری را در رشته می‌دهد، استفاده کردیم؛ به این صورت find می‌داند که جستجو را از کجا شروع کند. زمانی‌که شروع به قاچ‌زدن کردیم، ما از کاراکتری که یکی از @ جلوتر بود شروع کردیم. شاخص دوم که نشانگر پایانِ قاچ‌زدن است، کاراکتر فاصله بود که در مثال بالا با نام متغیر sppos مشخص است. حالا ما قاچ‌زدن را از کاراکتر بعد از @ و تا کاراکتر فاصله (ولی نه شامل آن) انجام دادیم.

سند مرتبط به متد find را می‌توانید در لینک زیر بخوانید:

https://docs.python.org/3.5/library/stdtypes.html#string-methods

## عملگر format / قالب / آرایش

عملگر فرمت یا آرایش یعنی % به ما اجازه‌ی ساخت رشته‌ها و جابجا کردن بخشی از آن‌ها را، با داده‌هایی که در متغیرها ذخیره شده است، می‌دهد. زمانی‌که % روی اعداد صحیح به کار گرفته می‌شود، عملگر پیمانه خوانده شده و کارکرد متفاوتی دارد. زمانی‌که اولین عملوند یک رشته باشد، % عملگر آرایش خواهد بود.

اولین عملوند فرمت‌استرینگ یا رشته‌چینش خوانده می‌شود و شامل تعداد یک یا بیشتر از فرمت‌سکوئنس یا «توالی چینش [حروف]» می‌شود. فرمت‌سکوئنس مشخص می‌کند که عملوند نظیر به نظیر چگونه قالب‌بندی یا آرایش شود. نتیجه‌ی حاصله یک رشته خواهد بود.

به عنوان مثال، فرمت‌سکوئنس "%d" به این معنیست که دومین عملوند بایستی که به عنوان یک عدد صحیح قالب‌بندی شود (در اینجا d اولین حرف decimal است).

```
>>> camels = 42
>>> '%d' % camels
'42'
```

نتیجهْ رشته‌ای با مقدار "42" است. مراقب باشید که این "42" با عدد 42 متفاوت است چرا که اولی یک رشته است و دومی یک عدد صحیح.

یک فرمت‌سکوئنس، در هر جایی از یک رشته می‌تواند قرار بگیرد، در نتیجه می‌توانید مقداری را در درون یک جمله فرو کنید. به مثال زیر دقت کنید:

```
>>> camels = 42
>>> 'I have spotted %d camels.' % camels
'I have spotted 42 camels.'
```

اگر بیش از یک فرمت‌سکوئنس در یک رشته وجود داشته باشد، دومین آرگیومنت بایستی یک تاپِل[I] باشد. هر فرمت‌سکوئنس با عنصر موجود در تاپل به ترتیب تطبیق داده می‌شود.

مثال زیر از "%d" برای جایگذاری یک عدد صحیح و "%g" برای جایگذاری یک عدد اعشاری (نپرسید چرا) و "%s" برای جایگذاری یک رشته استفاده کرده است.

```
>>> 'In %d years I have spotted %g %s.' % (3, 0.1, 'camels')
'In 3 years I have spotted 0.1 camels.'
```

شماره‌ی عناصر در تاپل بایستی با شماره فرمت‌سکوئنس در رشته تطابق داشته باشد. مثلاً سومین فرمت‌سکوئنس با سومین عنصر موجود در تاپل پیوند می‌خورد. همچنین نوع عناصر بایستی که با فرمت‌سکوئنس مطابق باشد:

```
>>> '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string
>>> '%d' % 'dollars'
TypeError: %d format: a number is required, not str
```

[I] Tuple

در مثال اول، عناصر کافی برای عبارت وجود ندارد و در مثال دوم عملوند دوم با نوع اشتباهی نظیر به نظیر شده است.

عملگر قالب، عملگر قدرتمندی‌ست با این توضیح که استفاده از آن سخت است. اطلاعات بیشتر را می‌توانید از پیوند زیر بخوانید:

https://docs.python.org/3.5/library/stdtypes.html#printf-style-string-formatting

## اشکال‌زدایی

مهارتی که لازم است در حین یادگیری برنامه‌نویسی بیاموزید، پرسش دائمی این سؤال از خود است: «چه مشکلی اینجا می‌تونه پیش بیاد؟» یا «چه کار یا کارهایی، کاربر نهایی می‌تونه انجام بده که برنامه‌ی "بی‌نقص" ما کرش کنه؟»

برای نمونه، به برنامه‌ای که ما برای نشان دادن حلقه‌ی while در فصل «تکرار» استفاده کردیم دقت کنید:

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


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

اگر کاربر هیچ ورودی وارد نکند و Enter را بزند چه اتفاقی می‌افتد:

```
> 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 به خوبی کار می‌کند، مگر اینکه با یک خط خالی مواجه شود. در این صورت به خاطر اینکه کاراکتر صفرمی وجود ندارد، ما یک تریس‌بک دریافت می‌کنید.

دو راه‌حل برای حل این مشکل و امن کردن خط سوم – حتی اگر ورودی خالی باشد – وجود دارد.

اولین راه استفاده از متد startswith است که مقدار False را برای رشته‌ی خالی برمی‌گرداند:

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

دومین راه‌حل نوشتن یک گزاره‌ی if با استفاده از الگوی محافظ است. به این صورت که مطمئن شویم دومین عبارت منطقی تنها زمانی‌که حداقل یک کاراکتر در رشته وجود دارد بررسی می‌شود.

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

## واژگان فصل

**شمارنده / Counter:**

متغیری که برای شمارش چیزی به کار می‌رود و معمولا در ابتدا با صفر مقداردهی می‌شود و سپس افزایش پیدا می‌کند.

**رشته خالی / Empty String:**
رشته‌ای که کاراکتری نداشته و طول آن 0 است و توسط دو علامت نقل‌قول مشخص می‌شود.

**عملگر قالب یا آرایش / Format Operator:**
یک عملگر، %، که یک «چینش رشته/فرمت‌استرینگ» و یک تاپل را گرفته و سپس یک رشته که شامل عناصر آن تاپل می‌شود را می‌سازد؛ به صورتی که فرمت‌استرینگ تعیین‌کننده‌ی ترتیب قرار گیری تاپل در داخل رشته می‌شود.

**توالی چینش حروف / Format Sequence:**
توالی‌ای از کاراکترها در یک فرمت‌استرینگ، مثل d%، که نشان می‌دهد یک مقدار چطور بایستی شکل بگیرد.

**رشته چینش / Format String:**
رشته‌ای که در کنار عملگر قالب مورد استفاده قرار می‌گیرد و شامل توالی حروف می‌شود.

**پرچم / Flag:**
یک متغیر بولی که برای نشان دادن درست یا غلط بودن یک شرط مورد استفاده قرار می‌گیرد.

**احضار / Invocation:**
یک گزاره که متدی را فراخوانی می‌کند.

**غیرقابل تغییر / Immutable:**
زنجیره‌ای که آیتم‌های آن قابل گمارش نیستند. به عبارتی شما نمی‌توانید آیتم‌های موجود در آن را تغییر دهید و مقدار جدیدی را به آن‌ها اختصاص دهید.

**شاخص / Index:**
یک مقدار صحیح برای انتخاب آیتمی در یک زنجیره، مانند یک کاراکتر در یک رشته.

**آیتم / Item:**
یکی از مقادیر موجود در یک توالی.

**متد / Method:**
یک تابع که با یک آبجکت در ارتباط است و توسط نشانه‌گذاری نقطه‌ای یا نماد نقطه فراخواهی می‌شود.

**آبجکت / Object:**
چیزی که یک متغیر می‌تواند به آن ارجاع پیدا کند. فعلا می‌توانید از آبجکت و متغیر به جای هم استفاده کنید چون تا اینجا تقریبا معنای مشابهی دارند.

**جستجو / Search:**
الگویی از پیمایش برای یافتن چیزی است که بعد از یافتن آن متوقف می‌شود.

**توالی یا زنجیره / Sequence:**
یک مجموعه‌ی مرتب شده از یک دسته از مقدارهاست که هر مقدار توسط یک شاخص صحیح شناسایی می‌شود.

**قاچ یا قطعه / Slice:**
قسمتی از یک رشته که توسط بازه‌ای از شاخص‌ها مشخص شده است.

**پیمایش / Traverse:**
گردش کردن در بین آیتم‌های یک توالی، و انجام عملیات مشابه روی هر کدام از آن‌ها.

## تمرین‌ها

**تمرین ۵:** کد پایتون زیر را بردارید:

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

با استفاده از find و قابلیت قاچ‌کردن رشته، قسمتی از رشته که بعد از دو نقطه آمده را جدا کرده و سپس با استفاده از تابع float آن قسمت جداشده را به یک عدد اعشاری تبدیل کنید.

**تمرین ۶:** اسناد متدهای رشته را از اینجا بخوانید:

https://docs.python.org/3.5/library/stdtypes.html#string-methods

برای اینکه بفهمید بعضی از آن‌ها چطور کار می‌کنند بهتر است که در عمل از این متدها استفاده کنید. متدهای strip و replace از جمله به‌دردبخورها به حساب می‌آیند.

سند، از چینشِ متن خاصی – که ممکن است شما را گیج کند – استفاده می‌کند. برای مثال در find(sub[, start[, end]]) قلاب‌ها نشان‌دهنده‌ی آرگیومنت‌های اختیاری‌اند. در نتیجه sub اجباری‌ست ولی start که در قلاب قرار دارد اختیاری‌ست. همچنین اگر start را مشخص کنید، end نیز می‌تواند به صورت اختیاری مورد استفاده قرار بگیرد.

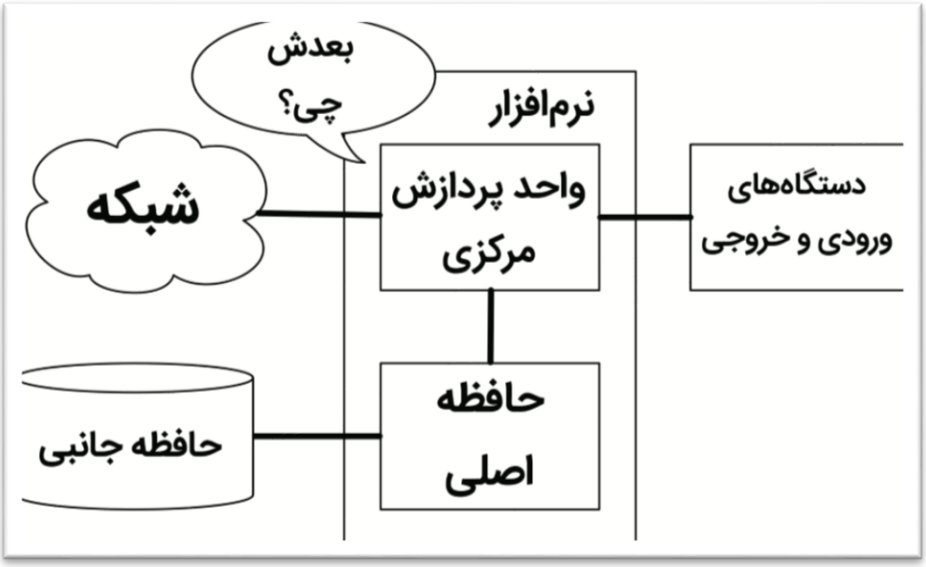
# فصل ۷

# فایل‌ها

## ماندگاری

تا اینجا در خصوص نحوه‌ی نوشتن برنامه‌ها و ارتباط برقرار کردن با واحد پردازش مرکزی با استفاده از اجرای شرطی، توابع، و تکرارها آشنا شدیم. در خصوص چگونگی ساخت و استفاده از ساختار داده در حافظه‌ی اصلی نیز آموختیم. CPU و حافظه جایی‌اند که نرم‌افزار در آن‌ها کار می‌کند و اجرا می‌شود. در اصل این منطقه همان جایی‌ست که "فکر کردن" اتفاق می‌افتد.

ولی مشکل اینجاست که می‌خواهیم بحث میان سخت‌افزارهای ما ادامه داشته باشد، اما همین که برق برَود و سیستم خاموش شود، تمام آنچه در CPU و حافظه اصلی بوده پاک خواهد شد. برنامه‌های پایتونی ما تلاش‌های تفریحیِ ناپایداری خواهند بود که از بین می‌روند.




در این فصل با حافظه‌ی جانبی (یا فایل‌ها) سر و کار داریم. اطلاعات در حافظه‌ی جانبی زمانی‌که سیستم خاموش می‌شود، از بین نمی‌رود؛ یا مثلا برنامه‌ای که نوشته‌ایم می‌تواند روی یک حافظه‌ی فلش کپی شده، از سیستم جدا و سپس به یک سیستم دیگر منتقل و اجرا شود.

تمرکز اصلی ما روی خواندن و نوشتن فایل‌های متنی مانند همان‌هایی که در ویرایشگر متن می‌نویسیم، خواهد بود. در ادامه یاد خواهیم گرفت که چطور با فایل‌های پایگاه داده که فایل‌های باینری‌اند کار کنیم؛ به‌خصوص آن‌ها که برای خوانده و نوشتن شدن توسط نرم‌افزارهای پایگاه طراحی شده‌اند.

## باز کردن فایل‌ها

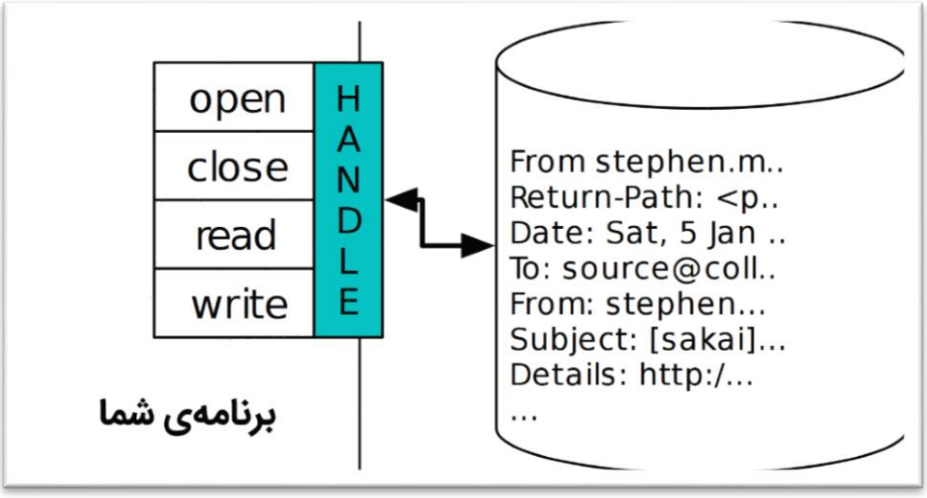
وقتی قرار است که فایلی را بخوانیم یا روی آن بنویسیم (مثلا فایل‌های روی هارد دیسک شما)، ابتدا باید فایل را باز کنیم. باز کردن فایل، برقراری ارتباط با سیستم‌عامل شما به حساب می‌آید. چرا سیستم‌عامل؟ چون سیستم‌عامل است که می‌داند هر فایل

در کجا ذخیره شده است. زمانی‌که شما یک فایل را باز می‌کنید، از سیستم‌عامل می‌پرسید که آن فایل را از طریق نامش پیدا کرده و مطمئن شود که فایل اصلا وجود دارد. در این مثال، ما فایل mbox.txt که در همان فولدر جاری است را باز می‌کنیم؛ منظورمان از فولدر جاری، همان مسیری‌ست که پایتون در آن اجرا شده است. شما می‌توانید فایل را از طریق این لینک دانلود کنید:

http://www.py4e.com/code3/mbox.txt

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

اگر عملیات open موفقیت‌آمیز باشد، سیستم عامل یک file handle یا «دستگیره‌ی فایل» باز می‌گرداند. دستگیره‌ی فایل، خودِ داده‌های موجود در فایل نیست، اما در عوض یک «دستگیره» است که ما می‌توانیم با استفاده از آن، داده‌ها را بخوانیم. اگر فایل درخواست‌شده وجود خارجی داشته باشد، شما یک دستگیره خواهید داشت و اگر مجوزهای لازم را در اختیار داشته باشید، فایل خوانده خواهد شد.



اگر فایل وجود خارجی نداشته باشد، open با شکست روبرو شده و شما را به یک تریس‌بک مهمان خواهد کرد. در نهایت شما دستگیره، برای دسترسی به محتویات فایل، 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'
```

در ادامه ما از try و except برای مدیریت موقرانه این حالت استفاده خواهیم کرد. به این صورت وقتی تلاش کردیم که یک فایلی که وجود ندارد را باز کنیم، با تریس‌بکِ خشنِ پایتون چشم‌درچشم نشویم.

## فایل‌های متنی و خطوط

یک فایل متنی را می‌توان سلسله‌ای از خطوط در نظر گرفت. درست مثل سلسله‌ای از کاراکترها که در پایتون یک رشته خوانده می‌شود. نوشته‌ی زیر، یک فایل متنی‌ست که فعالیت‌های مرتبط با ایمیل را از افراد متفاوت جمع‌آوری می‌کند؛ این فایل از یک پروژه‌ی متن‌باز به عاریت گرفته شده 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 متنی را می‌توانید از لینک زیر دریافت کنید:

http://www.py4e.com/code3/mbox.txt

همچنین یک نسخه‌ی خلاصه‌شده در لینک زیر در دسترس است:

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

فرمت این فایل‌ها استاندارد است و شامل چندین پیام پستی می‌شود. خطوطی که با کلمه"From" آغاز می‌شوند، پیام‌ها را جدا کرده و خطوطی که با "From:" شروع می‌شوند، قسمتی از آن پیام‌ها هستند. برای اطلاعات بیشتر در خصوص فرمت mbox این صفحه را ببینید:

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

برای شکستن فایل به خطوط، کاراکتر ویژه‌ای که نشان‌دهنده‌ی پایان خط است وجود دارد. این کاراکتر newline یا خط‌جدید خوانده می‌شود.

پایتون کاراکتر خط‌جدید را با یک بک‌اسلش و n مشخص می‌کند. اگرچه \n به نظر دو کاراکتر است، ولی در اصل یک کاراکتر خوانده می‌شود. فرض کنید که ما این کاراکتر را در داخل متغیری با نام stuff قرار می‌دهیم. حالا اگر در داخل مفسر stuff را وارد کنیم، این کاراکتر هم نمایش داده می‌شود، ولی اگر از print استفاده کنیم، رشته‌ی ما به دو خط شکسته می‌شود. در اصل در حین چاپ کردن متغیر، شما اثر کاراکتر \n را خواهید دید، و نه خود کاراکتر را. برای شفاف شدن موضوع به مثال زیر دقت کنید:

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

برای اینکه مطمئن شوید که \n تنها یک کاراکتر است بد نیست که طول رشته‌ی X\nY را با پایتون اندازه بگیرید. می‌بینید که پایتون آن را سه کاراکتر در نظر می‌گیرد. در اصل \n یک کاراکتر به حساب آمده است.

به این صورت، و با تصور وجود یک نویسه‌ی نامرعی در پایان هر خط، انتهای خطوط در متن‌ها مشخص می‌شود. کافی‌ست که فقط آن را در ذهن خود تصویر کنید. این نویسه یا کاراکتر ناپیدا، همان \n است.

حرف آخر اینکه، کاراکترِ خط‌جدید کاراکترهای موجود در یک فایل را به خطوط جداگانه می‌شکند.

## خواندن فایل‌ها

دستگیره‌ی فایل، حاوی دیتای فایل نمی‌شود، با این حال با استفاده از یک حلقه‌ی for می‌توانید که به راحتی خطوط فایل را مرور کرده و بشمارید:

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

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

ما از دستگیره‌ی فایل به عنوان یک توالی در حلقه‌ی for استفاده کرده‌ایم. حلقه‌ی for در مثال بالا، تعداد خطوط موجود در فایل را شمرده و نتیجه را چاپ می‌کند. ترجمه‌ی تحت‌الفظی حلقه‌ی for در به زبان آدمیزاد می‌شود: «برای هر خط در فایل – که توسط دستگیره‌ی فایل ارائه شده است – یک عدد به متغیر count اضافه کن.»

دلیل اینکه تابع open تمام فایل را نمی‌خواند این است که احتمال اینکه فایل بسیار بزرگ باشد و حاوی گیگابایت‌ها دیتا شود، وجود دارد. به این صورت گزاره‌ی

open برای باز کردن هر فایل – از کوچک به بزرگ – مقدار یکسانی زمان نیاز دارد. گزاره‌ای که داده‌ی اصلی را از فایل می‌خواند، حلقه‌ی for است.

زمانی‌که ما از حلقه for به این صورت استفاده می‌کنیم، پایتون وظیفه‌ی بخش‌بخش کردن داده‌های داخل فایل را به خطوط جداگانه، با استفاده از کاراکتر newline/خط‌جدید برعهده می‌گیرد. پایتون هر خط را از طریق خط‌جدید خوانده و کاراکتر خط‌جدید را به عنوان آخرین کاراکتر متغیر line برای هر تکرار حلقه‌ی for به حساب می‌آورد.

به این خاطر که حلقه for داده‌ها را به صورت یک خط در یک زمان می‌خواند، به راحتی می‌تواند که خطوط بیشمار در فایل‌های بزرگ را خوانده و بشمارد، بدون اینکه حافظه‌ی اصلی تحت فشار حجم زیاد فایل قرار بگیرد. برنامه‌ی بالا می‌تواند که خطوط را در هر حجمی با استفاده‌ی حداقلی از حافظه‌ی اصلی بخواند چرا که برنامه هر خط را می‌خواند، می‌شمارد و سپس داده‌های آن را دور می‌ریزد و به سراغ خط بعدی می‌رود. به عبارت ساده‌تر تنها برای همان خطِ در جریان اجرا، نیاز به حافظه‌ی اصلی وجود دارد و داده‌هایِ قبلی، حافظه را اشغال نخواهند کرد.

اما اگر می‌دانید فایلی که قرار است پردازش شود، نسبت به حجم حافظه‌ی اصلی، بسیار کوچک است، می‌توانید تمام فایل را با استفاده از متد read و دستگیره‌ی فایل در داخل یک متغیر قرار دهید. به این صورت برای تمام فایل روی حافظه‌ی اصلی فضا اشغال می‌شود:

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

در این مثال، کل محتویات (شامل ۹۴,۶۲۶ کاراکتر) از فایل mbox-short.txt خوانده شده و مستقیما در متغیر inp قرار می‌گیرند. در کدِ بالا، ما از قاچ‌کردن رشته

برای پرینت کردن ۲۰ کاراکتر ابتدایی رشته‌ی داده‌ای که در inp وجود دارد استفاده کرده‌ایم.

زمانی‌که فایلی به این صورت خوانده می‌شود، تمام کاراکترهای خطوط و کاراکترهای خط‌جدید در یک رشته‌ی بزرگ قرار خواهند گرفت. در مثالِ ما، این رشته‌ی بزرگ در متغیر inp ذخیره می‌شود. در حالت تعاملی پایتون می‌توانید با صادر کردن گزاره‌ی inp تمام کاراکترهای این رشته‌ی بزرگ را چاپ کنید. فراموش نکنید که این روش بایستی تنها روی فایل‌هایی اجرا شود که مطمئن باشید با قرارگیری آن‌ها روی حافظه‌ی اصلی مشکلی پیش نمی‌آید و به راحتی روی آن جا خواهند شد.

اگر فایل نسبت به حافظه‌ی اصلی خیلی بزرگ باشد، بایستی با استفاده از حلقه‌های for یا while برنامه‌تان را طوری بنویسید که فایل را به قطعه‌های کوچک‌تر تقسیم کند.

## جستجو در یک فایل

وقتی که در بین داده‌های یک فایل شروع به جستجو می‌کنید، با احتمال زیاد با استفاده از یک الگوی خاص به دنبال خطوط ویژه‌ای برای پردازش خواهید گشت؛ خطوطی که شرایط به‌خصوصی را داشته باشند. همچنین می‌توانید چند الگو را برای خواندن یک فایل با هم ترکیب کنید. برای این کار از متدهای رشته به منظور ساخت مکانیسم‌های ساده‌ای برای جستجو استفاده خواهیم کرد.

به عنوان مثال اگر ما بخواهیم که یک فایل را خوانده و سپس تنها خطوطی که با From: شروع شده‌اند را چاپ کنیم، می‌توانیم از متدِ رشته startswith برای پیدا کردن آن خطوط بهره ببریم:

```
fhand = open('mbox-short.txt')
count = 0
for line in fhand:
    if line.startswith('From:'):
        print(line)


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

زمانی‌که این برنامه اجرا می‌شود، خروجی زیر را دریافت می‌کنیم:

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


From: louis@media.berkeley.edu


From: zqian@umich.edu


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

خروجی، عالی به نظر می‌رسد چرا که تمام خطوطی که با :From شروع شده‌اند در آن چاپ شده است. تنها مشکل وجود خطوط خالی بین آن‌هاست. این مشکل به خاطر کاراکتر نامرئی خط‌جدید/newline به‌وجود آمده است. دلیلش این است که هر خط با یک کاراکتر خط‌جدید تمام می‌شود، در نتیجه آن کاراکتر هم چاپ خواهد شد؛ و باعث می‌شود که در کنار چاپ جداگانه‌ی خطوط، یک خط خالی بین آن‌ها ایجاد شود.

در اینجا می‌توانیم از روش قاچ‌زدن رشته و چاپ تمام کاراکترها به‌جز کاراکتر آخر استفاده کنیم؛ ولی روش ساده‌تر برای اینکار استفاده از متد rstrip است. به این صورت که فاصله‌ی خالیِ سمت راستِ یک رشته را جدا می‌کند:

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


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

با اجرای دوباره‌ی برنامه، 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
From: zqian@umich.edu
From: rjlowe@iupui.edu
From: cwen@iupui.edu
...
```

همینطور که برنامه‌ی پردازش فایل پیچیده‌تر می‌شود، شاید بخواهید ساختار حلقه‌های جستجو را با استفاده از continue بسازید. ایده‌ی اصلی حلقه‌ی جستجو این است که شما به دنبال خطوط «جالب توجه» گشته و از خطوط دیگر که برای شما یا هدف برنامه جذابیتی ندارد (غیرجذاب) پرش نمایید. سپس زمانی‌که شما یک خط جالب توجه پیدا کردید، با آن خط عملیات خاصی را انجام دهید.

ساختار حلقه برای پرش کردن از خطوط غیرجذاب می‌تواند شبیه به الگوی زیر باشد:

```
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: http://www.py4e.com/code3/search3.py
```

خروجی برنامه، تفاوتی با خروجی قبلی ندارد. در این مثال، خطوط غیرجذاب آن‌هایی هستند که با "From:" شروع نمی‌شوند، و ما با استفاده از continue از این خطوط پرش کردیم. برای خطوط جذاب (یعنی آن‌هایی که با "From:" شروع می‌شوند) اوضاع فرق می‌کند و در این مثال ما روی آن‌ها پردازش انجام می‌دهیم.

همچنین می‌توانیم از متد رشته find شبیه به گزینه جستجو در ویرایشگرهای متنی استفاده کنیم. به این صورت که find به دنبال خطوطی بگردد که رشته‌ی مورد نظر را در خود دارند. حالا این رشته نه‌فقط در ابتدا که در هر جایی از این خطوط می‌تواند باشد. از آنجایی که find به دنبال یک رشته در رشته‌ی دیگر می‌گردد و همچنین موقعیت رشته‌ی پیدا شده را با عدد برمی‌گرداند، می‌توانیم از حلقه‌ی زیر، به منظور پیدا کردن خطوطی که حاوی uct.ac.za@ می‌شوند، استفاده کنیم. به یاد داشته باشید اگر خطی حاوی رشته‌ی مورد نظر نشود، مقدار 1- توسط متد find برگردانده خواهد شد.

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if line.find('@uct.ac.za') == -1: continue
    print(line)


# Code: http://www.py4e.com/code3/search4.py
```

برنامه‌ی بالا، خروجی زیر را چاپ خواهد کرد.

```
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
...
```

## انتخاب نام فایل را به کاربر بسپارید

اگر قرار باشد که هر بار برای پردازش یک فایلِ متفاوت، کد برنامه را عوض کنیم، اوضاع جالب نخواهد بود. بهتر است که از کاربر اسم فایلی که قرار است پردازش شود، پرسیده شده و سپس عملیات روی آن فایل صورت پذیرد. به این صورت کد ما می‌تواند روی فایل‌های مختلف، بدون ایجاد تغییر در برنامه‌ی اصلی، عملیات انجام دهد.

این کار ساده‌ای است که با خواندن نام فایل از طریق تابع input انجام می‌پذیرد:

```
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: http://www.py4e.com/code3/search6.py
```

ما اسم فایل را از کاربر می‌گیریم و آن را در متغیر fname قرار می‌دهیم. سپس آن فایل را با استفاده از متغیری که در اختیار داریم باز می‌کنیم. حالا می‌توانید که برنامه را روی فایل‌های مختلف به کار بگیرید.

```
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
```

قبل از اینکه به سراغ بخش بعدی بروید، به برنامه‌ی بالا نگاهی بیندازید و از خود بپرسید که «چه اتفاقی ممکن است بیفتد که برنامه با مشکل مواجه شود؟» یا «کاربر ممکن است چه کاری انجام دهد که برنامه‌ی قشنگ ما با یک خطای زشت به کار خود پایان دهد؟ چیزی که کاربر از دیدنش خوشش نخواهد آمد!»

## استفاده از try و except و open

قرار شد که قبل از مشاهده‌ی این قسمت به سوالات بالا جواب دهید. یک بار دیگر آن سوال‌های پاراگراف قبلی را از خودتان بپرسید.

اگر کاربر نامی را وارد برنامه کند که وجود خارجی نداشته باشد چه اتفاقی می‌افتد؟

```
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'
```

نخندید، واقعا ممکن است کاربر کارهایی را انجام دهد که به نظر شما احتمالش وجود ندارد. کارهایی که برنامه را با مشکل مواجه کند. حالا شاید با قصد، حتی قصد خرابکاری، این کار را انجام دهد. به عنوان یک اصل، بخشی از هر تیم توسعه‌ی نرم‌افزار، افراد یا گروهی هستند که QA یا گروه «اطمینان از کیفیت» خوانده می‌شوند. کار این گروه یا شخص انجام چیزهای، حتی مسخره، برای از کار انداختن برنامه است. به عبارتی آن‌ها به هر چیزی متوسل می‌شوند که برنامه با مشکل مواجه شود و کارش را درست انجام ندهد.

تیم QA مسئول پیدا کردن عیب‌های یک برنامه قبل از انتشار آن و رسیدن به دست کاربر نهایی است. کاربر نهایی هم همان کسی است که قرار است برنامه را بخرد یا حقوق ما را بپردازد. در نتیجه تیم QA بهترین رفیق یک برنامه‌نویس است.

حالا که مشکل در برنامه را کشف کردیم، بایستی که آن را به روش مناسبی با استفاده از try یا except حل کنیم. فرض کنید که باز کردن فایل با open با مشکل مواجه شود (به دلیل ناهماهنگی اسم فایل با فایل‌های موجود در آن مسیر). اینجاست که باید یک کد برای اعلام مشکل 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: http://www.py4e.com/code3/search7.py
```

تابع exit به اجرای برنامه خاتمه می‌دهد. این تابع هیچگاه چیزی را برنمی‌گرداند. حال اگر کاربر ما (یا تیم QA) اسم اشتباه یا مزخرفی را وارد کند، ما مچشان را گرفته و برنامه را با حفظ آبرو می‌بندیم:

```
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
```

حفاظت از فراخوانی open در برنامه، مثال خوبی از استفاده‌ی درست از try و except در یک برنامه‌ی پایتونی است. ما عبارت Pythonic یا پایتونی را زمانی‌که یک کاری را به روش پایتون انجام می‌دهیم استفاده می‌کنیم. به همین خاطر به مثال بالا «یک راه پایتونی برای باز کردن یک فایل» می‌گوییم.

زمانی‌که شما مهارت بیشتری در پایتون کسب کردید و با برنامه‌نویس‌های دیگر بر سر راه‌های حل یک مساله صحبت کردید، می‌توانید بگویید که این راه‌حل پایتونی‌تر است یا خیر. ولی چرا پایتونی‌تر باشد؟ به این خاطر که بخشی از برنامه‌نویسی هنر است و بخشی مهندسی؛ به عبارتی یک راه حل پایتونی‌تر راه‌حل مهندسی-هنری‌تری برای یک مساله است. قرار نیست که فقط برنامه‌ی ما کار کند، بلکه می‌خواهیم که راه‌حلمان زیبا و تحسین‌برانگیز باشد.

## نوشتن فایل‌ها

برای نوشتن یک فایل، بایستی که فایل را در مود w به عنوان پارامتر دوم باز کنید. بهتر است مثال زیر را برای درک بهتر ببینید:

```
>>> fout = open('output.txt', 'w')
>>> print(fout)
<_io.TextIOWrapper name='output.txt' mode='w'
encoding='cp1252'>
```

اگر فایل از پیش موجود باشد، باز کردن فایل در مود یا حالت w داده‌های پیشین را پاک کرده و از نو شروع می‌کند. به همین خاطر لازم است که مراقب باشید. اگر فایل از پیش موجود نباشد، فایل جدیدی ساخته خواهد شد.

متد write از آبجکت دستگیره‌ی فایل، داده‌ها را درون فایل قرار داده و تعداد کاراکترهای نوشته شده را برمی‌گرداند. حالت پیش‌فرضِ نوشتن، متن برای نوشتن (و خواندن) رشته‌ها است.

```
>>> line1 = "This here's the wattle,\n"
>>> fout.write(line1)
24
```

آبجکت فایل، جایی که قرار دارد را رهگیری می‌کند، بنابراین زمانی‌که write را دو مرتبه فراخوانی کنیم، داده‌های جدید به انتهای فایل اضافه می‌شونم.

بایستی مطمئن باشیم که انتهای خطوط را به درستی مدیریت می‌کنیم. برای اینکار لازم است از کاراکتر خط‌جدید در پایان هر خط استفاده کنیم. گزاره‌ی print به صورت خودکار خطوط جدید را الحاق می‌کند ولی متد write این کار را انجام نمی‌دهد.

```
>>> line2 = 'the emblem of our land.\n'
>>> fout.write(line2)
24
```

زمانی‌که کارتان تمام شد، فایل را ببندید و مطمئن شوید که آخرین بیت از داده‌ها روی دیسک به صورت فیزیکی نوشته شده است و با خاموش شدن سیستم از بین نخواهد رفت.

```
>>> fout.close()
```

همچنین ما می‌توانیم که فایلی را که برای خواندن باز کرده‌ایم، ببندیم ولی در این مورد (خواندن) اگر کمی سهل‌انگاری کنیم و تنها چند فایل را باز کرده باشیم مشکلی به وجود نمی‌آید. در اصل پایتون از بسته شدن فایل‌های باز قبل از پایان برنامه مطمئن می‌شود. ولی وقتی در حال نوشتن فایل‌ها هستیم، بایستی که صراحتا به پایتون اعلام کنیم که فایل را ببندد و چیزی را به دست تقدیر و شانس نسپاریم.

## اشکال‌زدایی

زمانی‌که در حال نوشتن و خواندن فایل‌ها هستید، امکان اینکه به مشکل فاصله‌ی سفید برخورد کنید وجود دارد. اشکال‌زدایی این خطاها می‌تواند مشکل باشد چرا که فاصله‌ها، تب‌ها و خط‌جدیدها در حالت عادی نامرئی هستند:

```
>>> s = '1 2\t 3\n 4'
>>> print(s)
1 2  3
 4
```

تابع توکاری شده‌ی repr در اینجا به کمک شما خواهد آمد. این تابع هر آبجکتی را به عنوان یک آرگیومنت می‌گیرد و سپس یک رشته که نشان دهنده‌ی آن آبجکت باشد را برمی‌گرداند. برای رشته‌ها، این تابع فاصله‌های سفید را با پشت‌بند بک‌اسلکش نمایش می‌دهد:

```
>>> print(repr(s))
'1 2\t 3\n 4'
```

این موضوع می‌تواند در اشکال‌زدایی به شما کمک کند.

مشکل دیگری که ممکن است با آن روبرو شوید مربوط به سیستم‌های متفاوت و استفاده از کاراکتر مختلف برای نمایش پایان خطوط است. برخی سیستم‌ها از یک

خط‌جدید که با \n مشخص می‌شود، و برخی سیستم‌ها از کاراکتر بازگشت یا \r به این منظور استفاده می‌کنند. برخی سیستم‌ها هم از هر دو نمونه استفاده می‌کنند. اگر شما فایل‌ها را در بین سیستم‌های مختلف جابجا می‌کنید، این ناهماهنگی‌ها ممکن است که برایتان دردسر درست کند.

برای اکثر سیستم‌ها، برنامه‌هایی برای تبدیل یک فرمت به دیگری وجود دارد. شما می‌توانید اطلاعات بیشتر در خصوص آن‌ها را در پیوند زیر مشاهده کنید. و البته اگر بخواهید می‌توانید دست به کار شده و یکی را خودتان بنویسید.

https://wikipedia.org/wiki/Newline

## واژگان فصل

**گرفتن / Catch:**
به استفاده از try و except برای اجتناب از استثناهایی که برنامه را با یک تریس‌بک خاتمه می‌دهد، گرفتن / Catch گفته می‌شود.

**خط‌جدید / Newline:**
کاراکتر ویژه‌ای که از آن در فایل‌ها و رشته‌ها برای نشان دادن پایان یک خط استفاده می‌شود.

**پایتونی / Pythonic:**
تکنیکی که به خوبی و زیبایی در پایتون کار می‌کند؛ به عبارتی به تکنیکی که چاشنی هنر و مهندسی را برای حل یک مساله داشته باشد تکنیک پایتونی می‌گوییم. «استفاده از try و except یک روش پایتونی برای مدیریت فایل‌هایی است که وجود ندارند.»

**اطمینان از کیفیت / Quality Assurance:**
شخص یا تیمی که تمرکز اصلی‌شان بررسی کیفیت نهایی یک نرم‌افزار است. QA اغلب درگیر آزمودن یک محصول و پیدا کردن مشکلاتْ قبل از انتشار آن است.

**فایل متنی / Text File:**
توالی کاراکترهایی که روی حافظه‌ی دائمی، مثل هارد دیسک، ذخیره شده‌اند.

## تمرین‌ها

**تمرین ۱:** برنامه‌ای بنویسید که یک فایل را بخواند و محتویات آن را تماما با حروف بزرگ چاپ کند (خط به خط). مثال زیر نحوه‌ی اجرا و خروجی این برنامه را نشان می‌دهد:

```
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
```

فایل مورد نظر را می‌توانید از لینک زیر دانلود کنید:

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

**تمرین ۲:** برنامه‌ای بنویسید که نام فایلی را بخواهد و سپس به دنبال خطوطی با الگوی زیر باشد:

```
X-DSPAM-Confidence:0.8475
```

زمانی‌که با خطی که با X-DSPAM-Confidence: شروع می‌شود مواجه شد، خط را بیرون کشیده و سپس عدد اعشاری روبروی آن را به دست بیاورد. سپس این خطوط را شمرده و مقدار کل SPAM-Confidence را محاسبه کند. زمانی‌که برنامه به انتهای فایل رسید، میانگین SPAM-Confidence را چاپ کند.

```
Enter the file name: mbox.txt
Average spam confidence: 0.894128046745
```

```
Enter the file name: mbox-short.txt
Average spam confidence: 0.750718518519
```

برنامه‌ی خود را با دو فایل mbox.txt و mbox-short.txt آزمایش کنید.

**تمرین ۳:** گاهی زمانی‌که حوصله‌ی برنامه‌نویس‌ها سر می‌رود و می‌خواهند که کمی خوش بگذرانند، کمی تخم‌مرغ شب عید به برنامه‌شان اضافه می‌کنند؛ به عبارتی برنامه را به صورتی ویرایش می‌کنند که اگر کاربر اسم خاصی مثل " na na boo boo" را وارد کرد، یک پیغام خنده‌دار چاپ کند. برنامه بایستی که برای تمام فایل‌های دیگر که وجود دارند یا ندارند به روش معمول عمل کند، و تنها بایستی که روی این اسم خاص واکنش غیرعادی نشان دهد. به مثال پایین از اجرای برنامه نگاه کنید:

```
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!
```

به این کار گذاشتن تخم‌مرغ شب عید در برنامه می‌گوییم و ما شما را به انجام آن ترغیب نمی‌کنیم؛ هرچند اکنون برای تمرین هم که شده این کار را انجام دهید.

# فصل ۸

# لیست‌ها

## لیست یک توالی است

درست مانند یک رشته، لیستْ توالی‌ای از مقدارهاست. در یک رشته، مقدارها، کاراکترها هستند؛ در یک لیست مقدارها هر چیزی می‌توانند باشند. مقدارها در یک لیستْ اِلمنت‌ها، آیتم‌ها و یا عناصر لیست خوانده می‌شوند.

چندین راه برای ساخت یک لیست جدید وجود دارد؛ ساده‌ترین راه، قرار دادن عناصر لیست در داخل قلاب است ([ و ]):

```
[10, 20, 30, 40]
['crunchy frog', 'ram bladder', 'lark vomit']
```

مثال اول لیستی از چهار عدد صحیح است. مثال دوم لیستی از سه رشته است. لازم نیست که عناصر یک لیست از نوع یکسانی باشند. به عنوان مثال عناصر لیست زیر شامل رشته، عدد اعشاری، عدد صحیح و یک لیست دیگر می‌شود:

```
['spam', 2.0, 5, [10, 20]]
```

لیستی که در لیست دیگر قرار می‌گیرد، تو در تو خوانده می‌شود.

لیستی که شامل هیچ المنتی نشود، یک لیست خالی خوانده می‌شود. شما به راحتی با یک قلاب خالی می‌توانید آن را بسازید:

```
[]
```

همانطور که انتظار می‌رود شما می‌توانید مقدارهای لیست را به متغیرها نسبت دهید:

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> numbers = [17, 123]
>>> empty = []
>>> print(cheeses, numbers, empty)
['Cheddar', 'Edam', 'Gouda'] [17, 123] []
```

## لیست‌های تغییرپذیرند

متنِ کدْ برای دسترسی به عناصر یک لیست، دقیقا مثل سینتکس برای دسترسی به کاراکترهای یک رشته است: عملگر قلاب. عبارت داخل قلاب، شاخص را تعیین می‌کند. به خاطر داشته باشید که شاخص‌ها با عدد 0 شروع می‌شدند:

```
>>> print(cheeses[0])
Cheddar
```

برخلاف رشته‌ها، لیست‌ها تغییرپذیرند؛ به عبارتی شما می‌توانید که ترتیب آیتم‌های یک لیست را تغییر دهید و یا آیتم‌ها را دوباره به آن اختصاص دهید. زمانی‌که عملگر قلاب در سمت چپ یک گمارش قرار می‌گیرد، عنصری از لیست که قرار است مقدار تازه‌ای بگیرد را مشخص می‌کند. بگذارید با یک مثال کمی روشن‌تر منظورم را بیان کنم:

```
>>> numbers = [17, 123]
>>> numbers[1] = 5
>>> print(numbers)
[17, 5]
```

عنصر یکم در لیست numbers عدد ۱۲۳ بود؛ با استفاده از دستور گمارش که با علامت = مشخص می‌شود، می‌توان این مقدار را تغییر داد. کافیست که در سمت چپ با استفاده از شاخص، عنصر مورد نظر در لیست را مشخص کنیم (در اینجا

numbers[1]) و سپس مقدار جدید را در سمت راست معادله قرار دهیم. به این صورت مقدار جدید با مقدار قبلیِ عنصر در لیست تعویض می‌شود.

یک لیست در چشم ما، یک رابطه بین شاخص و عنصر است. این رابطه را «مپینگ» یا «نمایش مطابقه» یا «ترانگاشت» یا «بازنمایی» می‌نماییم؛ به عبارتی هر شاخص مسیری به یک عنصر را بازنمایی می‌کند.

شاخص‌های لیست به‌مانند شاخص‌های رشته عمل می‌کنند:

- هر عدد صحیح می‌تواند به عنوان یک شاخص استفاده شود.
- اگر شما تلاش کنید یک عنصر که وجود خارجی ندارد را بخوانید یا بنویسید، خطای IndexError را دریافت خواهید کرد.
- اگر یک شاخص مقداری منفی داشته باشد، از انتهای لیست شروع به شمردن می‌کند.

عملگر in برای لیست‌ها نیز کار می‌کند:

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in cheeses
True
>>> 'Brie' in cheeses
False
```

## پیش رفتن در یک لیست

رایج‌ترین راه برای پیش رفتن در عناصر یک لیست استفاده از حلقه for است. در این حالت متنِ کدْ دقیقا شبیه همان چیزی است که در رشته استفاده کردیم:

```
for cheese in cheeses:
    print(cheese)
```

با این حلقه شما عناصر یک لیست را می‌خوانید. اما اگر شما بخواهید که عناصر را آپدیت کنید و چیزی روی آن‌ها بنویسید، نیاز به شاخص خواهید داشت. یک راه‌حل رایج برای اینکار ترکیب دو تابع range و len برای این کار است:

```
for i in range(len(numbers)):
    numbers[i] = numbers[i] * 2
```

این حلقه در یک لیست پیش می‌رود و هر عنصر را به صورت جداگانه به‌روز می‌کند. len برابر با تعداد عناصر یک لیست خواهد بود. range لیستی از شاخص‌ها از ۰ تا n - 1 را برمی‌گرداند. در اینجا n طول یک لیست است. در هر بار چرخش این حلقه i شاخص عنصر بعدی خواهد بود. دستورگمارش در بدنه با استفاده از i شروع به خواندن مقدار قدیمی کرده و سپس مقدار جدید را به آن اختصاص می‌دهد.

در یک لیست خالی بدنه‌ی حلقه for اصلا اجرا نمی‌شود:

```
for x in empty:
    print('This never happens.')
```

با وجود اینکه یک لیست می‌تواند شامل یک لیست دیگر شود، لیست تودرتو به عنوان تنها یک عنصر خوانده خواهد شد. به عنوان مثال، طولِ لیست زیر چهار خواهد بود:

```
['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```

## عملیات‌های لیست

با استفاده از عملگر + می‌توانید لیست‌ها را به هم بچسبانید:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print(c)
[1, 2, 3, 4, 5, 6]
```

با استفاده از عملگر * می‌توانید یک لیست را به تعداد دلخواه تکرار کنید:

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

مثال اول، لیست را چهار بار و مثال دوم لیست را سه بار تکرار کرد.

## قاچ‌زدن لیست (عملگر اسلایس)

عملگر قاچ‌زدن روی لیست‌ها نیز 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 به جای اولین شاخص چیزی ننویسید، قاچ‌زدن از ابتدای لیست شروع می‌شود. اگر دومین شاخص را ننویسید، قاچ‌زدن تا انتهای لیست ادامه پیدا خواهد کرد. اگر هر دو را ننویسید (مثال زیر) قاچِ لیست در اصل یک رونوشت از کل لیست خواهد بود:

```
>>> t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

از آنجایی که لیست‌ها تغییرپذیرند، بهتر است که قبل از اینکه عملیاتی روی آن‌ها انجام دهید، یک رونوشت بگیرید.

عملگر اسلایس در سمت چپ یک گزاره‌ی گمارش می‌تواند به شما کمک کند تا چندین عنصر را به صورت یک جا تغییر دهید:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3] = ['x', 'y']
>>> print(t)
['a', 'x', 'y', 'd', 'e', 'f']
```

## متدهای لیست

پایتون متدهایی را برای کار روی لیست‌ها فراهم می‌کند. به عنوان مثال، append یک عنصر جدید به انتهای لیست اضافه می‌کند:

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> print(t)
['a', 'b', 'c', 'd']
```

متد extend یک لیست را به عنوان آرگیومنت گرفته و تمام عناصر آن را به انتهای لیست قبلی اضافه می‌کند:

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> print(t1)
['a', 'b', 'c', 'd', 'e']
```

در این مثال t2 دست نخورده باقی می‌ماند.

متد sort عناصر یک لیست را از پایین به بالا مرتب می‌کند:

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> print(t)
['a', 'b', 'c', 'd', 'e']
```

اکثر متدهای لیست، بی‌نتیجه یا وُید هستند؛ آن‌ها لیست را تغییر می‌دهند ولی چیزی برنمی‌گردانند. اگر شما به‌صورت تصادفی گزاره t = t.sort() را صادر کردید، نتیجه شما را ناامید خواهد کرد. خودتان امتحان کنید:

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> print (t.sort())
None
```

## پاک کردن عناصر

چندین راه برای پاک کردن عناصر از لیست‌ها وجود دارد. اگر شما بدانید که شاخصِ عنصری که می‌خواهید چند است می‌توانید از pop استفاده کنید:

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> print(t)
['a', 'c']
>>> print(x)
b
```

متد pop لیست را تغییر داده و عنصری که حذف شده را برمی‌گرداند. اگر شما شاخصی را مشخص نکنید، آخرین عنصر را حذف کرده و برمی‌گرداند.

اگر می‌خواهید که یک مقدار را حذف کنید، می‌توانید از عملگر del استفاده کنید:

```
>>> t = ['a', 'b', 'c']
>>> del t[1]
>>> print(t)
['a', 'c']
```

اگر می‌دانید که چه عنصری را می‌خواهید پاک کنید ولی شاخص آن را نمی‌دانید، می‌توانید از remove استفاده کنید:

```
>>> t = ['a', 'b', 'c']
>>> t.remove('b')
>>> print(t)
['a', 'c']
```

مقدار None توسط remove برگردانده می‌شود.

برای بازگرداندن بیش از یک عنصر می‌توانید از del و روش قاچ‌زدن استفاده کنید:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del t[1:5]
>>> print(t)
['a', 'f']
```

همانطور که معمول است، قاچ‌زدن یا اسلایس کردن تمام المنت‌ها را تا شاخص دوم انتخاب می‌کند (ولی شامل شاخص دوم نمی‌شود). در اسلایس کردن از شاخص اول تا شاخص دوم انتخاب می‌شود که شامل شاخص اول می‌شود ولی شامل شاخص دوم نمی‌شود.

## لیست‌ها و توابع

تعدادی تابع توکاری شده برای انجام عملیات‌های سریع – بدون نیاز به نوشتن حلقه‌های خاص هر کدام – وجود دارد. شما می‌توانید برای کار کردن با لیست‌ها از این توابع استفاده کنید:

```
>>> 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 زمانی‌که همه‌ی عناصر یک لیست عدد باشند کاربرد دارد. بقیه توابع مثل (()max، ()len و غیره) روی رشته‌ها و سایر مقادیری که قابل مقایسه هستند کار می‌کند.

اکنون با استفاده از این توابع می‌توانیم یک برنامه که پیش‌تر به آن پرداخته بودیم را بازنویسی کنیم. این برنامه مقدار میانگین لیستی را که کاربر وارد کرده محاسبه می‌کند.

در باکس زیر، برنامه برای محاسبه میانگین بدون به کارگیریِ لیست کار می‌کند:

```
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: http://www.py4e.com/code3/avenum.py
```

در این برنامه ما متغیرهایی نظیر count و total را داریم که اعداد و مقدار کل را در خود نگه می‌دارند. برنامه هر بار از کاربر یک ورودی می‌خواهد و سپس آن مقدار را به total اضافه می‌کند.

حالا با استفاده از توابع توکاری شده می‌توانیم به راحتی هر عددی که کاربر وارد می‌کند را ذخیره کرده و در انتها تعداد و مجموع آن را پیدا 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: http://www.py4e.com/code3/avelist.py
```

در ابتدا، و پیش از شروع حلقه، ما یک لیست خالی ساختیم، سپس هر زمان که یک عدد از کاربر دریافت کردیم، آن را به لیست اضافه نمودیم. در انتها ما جمع عددهای لیست را محاسبه و آن را بر تعداد عددها تقسیم کردیم. به این صورت میانگین اعداد وارد شده در لیست در انتها چاپ خواهد شد.

## لیست و رشته‌ها

یک رشته، یک توالی از کاراکترهاست و یک لیست یک توالی از مقادیر؛ اما یک لیست از کاراکتر برابر با یک رشته نیست. برای تبدیل یک رشته به لیست می‌توانید از تابع لیست استفاده کنید.

```
>>> s = 'spam'
>>> t = list(s)
>>> print(t)
['s', 'p', 'a', 'm']
```

به این خاطر که list نام یک تابع توکاری شده است، بایستی مراقب باشیم که از آن به عنوان اسم متغیر استفاده نکنیم. همچنین من از حرف l به این خاطر که بسیار شبیه به عدد ۱ است استفاده نمی‌کنم. به همین خاطر در مثال بالا از t استفاده کردم.

تابع list رشته را به حروف مجزا می‌شکند. اگر شما بخواهید که یک رشته را به کلمه‌های مجزا بشکنید می‌توانید از متد split استفاده کنید:

```
>>> s = 'pining for the fjords'
>>> t = s.split()
>>> print(t)
['pining', 'for', 'the', 'fjords']
>>> print(t[2])
the
```

یک بار که از split برای شکستن لیست به کلمه‌ها استفاده کردید، می‌توانید از عملگر شاخص (علامت قلاب) برای نشان کردن یک کلمه خاص در لیست بهره ببرید.

همچنین شما می‌توانید در متد split از یک آرگیومنت اختیاری نیز استفاده کنید. به این آرگیومنت حائل[I] می‌گویند. حائل مشخص می‌کند که چه کاراکتری‌هایی به عنوان مرز بین کلمات انتخاب شود. در مثال زیر یک هایفن یا خطِ ربط به عنوان حائل مشخص شده و برنامه کلمه‌ها را بر اساس آن جدا می‌کند:

```
>>> s = 'spam-spam-spam'
>>> delimiter = '-'
>>> s.split(delimiter)
['spam', 'spam', 'spam']
```

[I] delimiter

متد join برعکس split عمل می‌کند. این متد یک لیست را گرفته و عناصر آن را همبند کرده و در نهایت یک رشته تحویل شما می‌دهد. متد join یک متد مرتبط با رشته است و شما بایستی که یک delimiter یا حائل را مشخص کنید. این حائل برای جا گرفتن بین عناصری که قرار است یک رشته را تشکیل دهند مورد استفاده قرار می‌گیرد:

```
>>> t = ['pining', 'for', 'the', 'fjords']
>>> delimiter = ' '
>>> delimiter.join(t)
'pining for the fjords'
```

در این مورد، حائلِ ما یک فاصله‌ی خالی است در نتیجه join آن فاصله را گرفته و بین کلمات قرار می‌دهد. برای هم‌بند کردن رشته بدون فاصله می‌توانید از یک رشته‌ی خالی "" به عنوان حائل استفاده کنید.

## تجزیه و تحلیل خطوط

معمولا زمانی‌که یک فایل را می‌خوانیم، قرار است که روی خطوط آن کاری انجام دهیم. اغلب به دنبال خطوط «جذاب» می‌گردیم و سپس آن خطوط را پارس یا تحلیل می‌کنیم تا قسمت‌های جذاب آن خطوط را پیدا کنیم. خب اگر بخواهیم روز هفته را از خطوطی که با From: شروع می‌شوند در بیاوریم و چاپ کنیم بایستی چکاری انجام دهیم؟

```
From stephen.marquard@uct.ac.zaSatJan 5 09:14:16 2008
```

متد split زمانی‌که با مسائل اینچنینی روبرو شویم بسیار کاربردی خواهد بود. می‌توانیم یک برنامه‌ی کوچک بنویسیم که به دنبال خطوطی که با From شروع می‌شوند بگردد و سومین کلمه آن خطوط 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: http://www.py4e.com/code3/search5.py
```

ما در اینجا یک نسخه‌ی خلاصه شده از گزاره if را استفاده کردیم و continue را در همان خطی که if قرار دارد، جا دادیم. این مدل خلاصه شده از تابع if تفاوتی با نوع معمولی که در آن continue در خط بعدی قرار می‌گیرد ندارد.

برنامه‌ی بالا خروجی پایین را خواهد داشت:

```
Sat
Fri
Fri
Fri
    ...
```

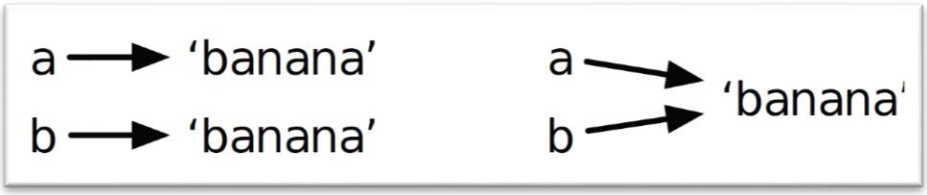
در ادامه یاد می‌گیریم که چطور از تکنیک‌های پیشرفته برای انتخاب خطوط استفاده کرده و همان اطلاعاتی که می‌خواهید را از دلِ خطوطِ جدا شده در بیاوریم.

## آبجکت‌ها و مقادیر

فرض کنید که ما گزاره‌های گمارشی زیر را اجرا کرده‌ایم:

```
a = 'banana'
b = 'banana'
```

ما می‌دانیم که a و b هر دو به یک رشته ارجاع داده می‌شوند، اما نمی‌دانیم که هر دو به یک رشته‌ی واحد ارجاع داده می‌شوند یا خیر. دو وضعیت محتمل است:




در حالت اول a و b به دو آبجکت مختلف ارجاع داده شده‌اند که مقدار یکسانی دارد. در حالت دوم هر دو به یک آبجکتِ واحد ارجاع داده شده‌اند.

برای بررسی اینکه دو متغیر به آبجکت یکسانی ارجاع داده شده‌اند یا خیر می‌توانید از عملگر is استفاده کنید:

```
>>> a = 'banana'
>>> b = 'banana'
>>> a is b
True
```

در این مثال، پایتون تنها یک آبجکت رشته ساخته است و هر دوی a و b به آن ارجاع داده شده‌اند.

اما زمانی‌که ما دو لیست می‌سازیم، در حقیقت دو آبجکت دریافت خواهیم کرد:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
```

در این حالت ما می‌گوییم که دو لیست به این خاطر که عناصر یکسانی دارند، برابر هستند ولی یکسان نیستند، چرا که آن‌ها دو آبجکت مجزا هستند. اگر دو آبجکت یکسان باشند، آن‌ها برابر هم خواهند بود ولی اگر برابر باشند، لزوما یکسان نخواهند بود.

تا اینجا ما از «آبجکت» و «مقدار» تقریبا به یک معنی استفاده کردیم و گاهی از این یکی برای اشاره به آن یکی بهره بردیم ولی اگر قرار است که کمی دقیق‌تر شویم بایستی بگوییم که یک آبجکت در حقیقت یک مقدار دارد. اگر شما a = [1,2,3] را اجرا کنید، a به یک آبجکتِ لیست اشاره می‌کند که مقدارش توالی خاصی از عناصر است. اگر لیست دیگری همین عناصر را داشته باشد ما می‌گوییم که مقادیر یکسانی دارد.

## استفاده از نام مستعار یا الایزینگ

اگر a به یک آبجکت اشاره داشته باشد و شما گمارشِ b = a را صادر کنید، هر دو متغیر به یک آبجکت ارجاع داده می‌شوند:

```
>>> a = [1, 2, 3]
>>> b = a
>>> b is a
True
```

ارتباط یک متغیر با یک آبجکت ارجاع خوانده می‌شود. در این مثال ما دو ارجاع به یک آبجکتِ واحد داریم.

یک آبجکت با بیش از یک ارجاع، بیش از یک نام خواهد داشت. ما می‌گوییم که آبجکت نام مستعار دارد (اِیلیِست آبجکت).

اگر یک ایلیست آبجکت یا آبجکت با نام مستعار قابل تغییر باشد، تغییر بر هر کدام از ایلیس‌ها روی دیگری نیز اثر می‌گذارد:

```
>>> b[0] = 17
>>> print(a)
[17, 2, 3]
```

با وجود اینکه این رفتار می‌تواند در جاهایی به درد ما بخورد، ولی ما را مستعد خطا و اشتباه می‌کند. به طور کلی بهتر است که از الایزینگ زمانی‌که با آبجکت‌های تغییرپذیر کار می‌کنید، اجتناب کنید.

برای آبجکت‌های غیرقابل تغییر مانند رشته‌ها، الایزینگ مشکلی را ایجاد نخواهد کرد. در این مثال:

```
a = 'banana'
b = 'banana'
```

تقریبا هیچ فرقی نمی‌کند که a و b به رشته‌های یکسانی ارجاع بدهند یا خیر.

## آرگیومنت‌های لیست

زمانی‌که لیست را به یک تابع می‌فرستید، تابع یک ارجاع به لیست را دریافت می‌کند. اگر تابع پارامترهای یک لیست را تغییر دهد، فراخوانْ تغییرات را خواهد دید. به عنوان مثال delete_head اولین عنصر از یک لیست را حذف می‌کند:

```
def delete_head(t):
    del t[0]
```

در اینجا با نحوه‌ی کاربرد این تابعی که تعریف کردیم آشنا می‌شویم:

```
>>> letters = ['a', 'b', 'c']
>>> delete_head(letters)
>>> print(letters)
['b', 'c']
```

پارامتر t و متغیر letters نام‌های مستعار (ایلیس‌ها) برای یک آبجکت مشابه هستند.

لازم است که بین عملیاتی که لیست‌ها را تغییر می‌دهد و عملیاتی که یک لیست جدید می‌سازد تفاوت قائل شوید. به عنوان مثال، متد append یک لیست را تغییر می‌دهد ولی عمگر + یک لیست جدید می‌سازد:

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> print(t1)
[1, 2, 3]
>>> print(t2)
None
```

```
>>> t3 = t1 + [3]
>>> print(t3)
[1, 2, 3]
>>> t2 is t3
False
```

این تفاوت زمانی‌که در حال نوشتن توابعی هستید که قرار است لیست‌ها را تغییر دهند، مهم می‌شود. به عنوان مثال این تابع ابتدای لیست را حذف نمی‌کند:

```
def bad_delete_head(t):
    t = t[1:]
```

عملگر اسلایس (قاچ‌زدن) یک لیست جدید می‌سازد و گزاره‌ی گمارش t را به آن نسبت می‌دهد ولی هیچکدام از این‌ها روی لیستی که به عنوان یک آرگیومنت به تابع فرستاده شده تاثیر نمی‌گذارد.

یک راه جایگزین نوشتن تابعی است که یک لیست جدید را ساخته و برمی‌گرداند. به عنوان مثال، tail تمام المنت‌های یک لیست به‌جز اولین‌شان را برمی‌گرداند:

```
def tail(t):
    return t[1:]
```

تابع، لیست اصلی را دست‌نخورده باقی می‌گذارد. نحوه‌ی استفاده از آن را ببینید:

```
>>> letters = ['a', 'b', 'c']
>>> rest = tail(letters)
>>> print(rest)
['b', 'c']
```

**تمرین ۱:** یک تابع به اسم chop بنویسید که یک لیست را گرفته و آن را با حذف اولین و آخرین عنصر تغییر دهد و در نهایت None را برگرداند.

سپس یک تابع با اسم middle بنویسید که یک لیست را گرفته و یک لیست جدید که شامل تمام محتوایات آن به جز عنصرهای اول و آخر می‌شود را برگرداند.

## اشکال‌زدایی

اگر در استفاده از لیست‌ها (و همچنین بقیه‌ی آبجکت‌های تغییرپذیر) بی‌دقتی کنید، ممکن است ساعت‌ها درگیر اشکال‌زدایی شوید. در اینجا به چند نکته که شما را از این اشتباهات دور می‌کند اشاره خواهیم کرد:

### ۱- اغلبِ متدها، آرگویمنت‌شان را دستکاری می‌کنند

بیاد داشته باشید که اغلب متدها، آرگویمنت‌شان را دستکاری می‌کنند و مقدار None را باز می‌گردانند. در خصوص متدهای رشته، عکس این قضیه صادق است و معمولا مقدار جدید بازگرداننده شده و خودِ رشته دست‌نخورده باقی می‌ماند.

اگر شما عادت به نوشتن کد رشته به سبک زیر کرده‌اید:

```
word = word.strip()
```

ممکن است که وسوسه شوید که کدِ لیست را به همین شکل بنویسید (که البته اشتباه است):

```
~~ {.python} t = t.sort() # WRONG! ~~
```

به این خاطر که sort مقدار None را باز می‌گرداند و عملیات بعدی که بر پایه‌ی t ساخته‌اید، با مشکل مواجه خواهد شد.

قبل از استفاده از عملگرها و متدهای لیست، اسناد مربوط به آن‌ها را با دقت بخوانید و سپس آن‌ها را در حالت تعاملی بیازمایید. اسناد متدها و عملگرها که لیست – یا سایر توالی‌ها (مثل رشته‌ها) – مشترکا استفاده می‌کنند در اینجا مکتوب شده‌اند:

https://docs.python.org/2/library/stdtypes.html#string-methods

اسناد عملگرها و متدهایی که تنها روی توالی‌های تغییرپذیر عمل می‌کنند در اینجا مکتوب شده است:

https://docs.python.org/2/library/stdtypes.html#mutable-sequence-types

## ۲- یک اصطلاح را برگزیده و به آن بچسبید

بخشی از مشکل ما با لیست‌ها این است که راه‌های زیادی برای انجام آنچه می‌خواهیم وجود دارد. به عنوان مثال برای حذف یک عنصر از لیست شما می‌توانید از pop یا remove یا del یا حتی از خاصیت اسلایس استفاده کنید.

برای اضافه کردن یک عنصر می‌توانید از متد append یا عملگر + استفاده کنید. فراموش نکنید که این کدها درست است:

```
t.append(x)
t = t + [x]
```

و این کدها اشتباهند:

```
t.append([x])          # WRONG!
t = t.append(x)        # WRONG!
t + [x]                # WRONG!
t = t + x              # WRONG!
```

هر کدام از این مثال‌ها را در حالت تعاملی امتحان کنید تا مطمئن شوید که آنچه ما توضیح می‌دهیم را در عمل می‌فهمید و می‌توانید استفاده کنید. به خاطر داشته باشید که تنها آخرین گزاره خطای در حین اجرا را بازمی‌گرداند و بقیه از نظر پایتون گزاره‌های درستی هستند ولی کار اشتباهی را انجام می‌دهند.

### ۳- کپی برای اجتناب از الایزینگ بگیرید

اگر می‌خواهید که از یک متد، مانند sort که آرگویمنتش را تغییر می‌دهد، استفاده کنید، و همچنین به لیست اصلی نیز نیاز دارید، بهتر است که یک کپی از آن بگیرید:

```
orig = t[:]
t.sort()
```

در این مثال شما می‌توانید از تابع توکاری شده‌ی sorted نیز استفاده کنید که مقدار جدید را باز می‌گرداند و به لیست اصلی دست نمی‌زند. اما در این مورد شما بایستی از sorted به عنوان نام متغیر استفاده نکنید.

### ۴- لیست‌ها، split و فایل‌ها

وقتی که شما فایل‌ها را خوانده و پارس (Parse یا تجزیه و تحلیل) می‌کنید راه‌های زیادی برای ارسال ورودی به برنامه و کرش کردن آن در مواجه با آن‌ها وجود دارد. برای اجتناب از این پیشامدها بهتر است نگاهی دوباره به الگوی محافظ بیندازید.

بیایید یک بار دیگر برنامه‌مان را ببینیم و نگاهی به روزِ هفته در خطوط فایلمان بیندازیم:

```
From stephen.marquard@uct.ac.zaSatJan 5 09:14:16 2008
```

از آنجایی که ما این خط را به کلمات تکه‌تکه می‌کنیم، ممکن است بی‌خیال استفاده از startswith شده و تنها نگاه به اولین کلمه‌ی خط برای یافتن خطوط جذاب بیندازیم. سپس می‌توانیم از continue برای پرش از خطوطی که From را به عنوان اولین کلمه ندارند استفاده کنیم:

```
fhand = open('mbox-short.txt')
for line in fhand:
    words = line.split()
    if words[0] != 'From' : continue
    print(words[2])
```

این برنامه ساده‌تر است و نیازی هم به rstrip برای از بین بردن کاراکتر خط‌جدید از آخر خطوط ندارد. ولی بد نیست نگاهی به خطوط زیر بیندازید:

```
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
```

به نظر که برنامه کار می‌کند و ما Sat را به عنوان اولین خط در خروجی بالا می‌بینیم، ولی در ادامه برنامه با مشکل مواجه می‌شود. چه اتفاقی افتاد؟ چه چیزی باعث بروز این خطا و ایجاد مشکل در برنامه‌ی زیبا و هوشمندانه‌ی ما شد؟ برنامه‌ی خیلی پایتونیِ ما!

هرچقدر برنامه را بالا پایین می‌کنیم می‌بینیم که مشکلی نباید وجود داشته باشد پس ایراد از کجاست؟ یک راه هوشمندانه و سریع استفاده از گزاره print است. بهترین مکان برای قرار دادن آن، درست قبل از خطی است که برنامه در حین اجرای آن به مشکل برمی‌خورد.

این روش ممکن است که خطوط زیادی در خروجی بسازد، با این حال حداقل یک سرنخ در دستان خود دارید. دستورِ چاپ متغیر words را درست قبل از خط پنجم قرار می‌دهیم. یک پیشوند :Debug هم به آن می‌چسبانیم تا آن را از خروجی‌های دیگر جدا کند.

```
for line in fhand:
        words = line.split()
        print('Debug:', words)
        if words[0] != 'From' : continue
        print(words[2])
```

وقتی یک برنامه را اجرا می‌کنیم، مقدار زیادی خروجی روی صفحه‌ی نمایش چاپ خواهد شد ولی در نهایت ما خروجی اشکال‌زدایی و تریس‌بک را داریم که به ما می‌گوید چه اتفاقی 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 خطِ اشکال‌زدایی، لیستی از کلماتی که ما با استفاده از split از خط استخراج شده است را نمایش می‌دهد. دقیقا در مکانی که برنامه خطا می‌دهد، ما با یک لیست خالی مواجه شده‌ایم. اگر فایل را در یک ویرایشگر متن باز کنیم و نگاهی به آن بیندازیم، دقیقا در نقطه‌ای که تحلیل فایل با مشکل مواجه شده ما با یک خط خالی طرف هستیم:

```
  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
```

برنامه در مواجه با یک خط خالی، صفر کلمه در لیست خود خواهد داشت و زمانی‌که کد به دنبال اولین کلمه (word[0]) در این لیست می‌گردد تا برررسی کند که آیا شرط

موجود در حلقه برقرار است یا خیر، با خطای «شاخص خارج از محدوده است» روبرو می‌شویم.

بهترین مکان برای اضافه کردن کد محافظ همینجاست. به این صورت که ابتدا، اولین کلمه را بررسی کنیم و ببینیم اگر وجود خارجی ندارد، گزاره‌ی بعدیِ کد اجرا نشود. راه‌های زیادی برای محافظت از این کد وجود دارد؛ ما در مثال زیر، بررسی شماره‌ی کلماتی که داریم را به عنوان راه‌حل انتخاب کرده‌ایم:

```
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 گزاره‌ی چاپ دیباگ (اشکال‌زدایی) را به جای حذف کردن آن، کامنت می‌کنیم تا اگر تغییرات ما نتیجه‌ی درخور را نداشت، دومرتبه از آن استفاده کنیم. سپس گزاره‌ی محافظ را اضافه کرده و تا اگر ما صفر کلمه داریم، حلقه به دور بعد برود و خط بعدی از فایل را بررسی کند.

به این صورت به قضیه نگاه کنید که گزاره‌های continue به ما کمک می‌کند که خطوط جذاب را پیدا کرده تا بتوانیم روی آن‌ها عملیات انجام دهیم. طبیعتا خطی که کلمه‌ای ندارد، «غیرجذاب» است و ما از آن پرش کرده و به خط بعد می‌رویم. خطی هم که اولین کلمه‌اش From نیست غیر جذاب است و در کد بالا از روی آن نیز پرش می‌کنیم.

بعد از اینکه برنامه را تغییر دادیم، به نظر می‌آید که همه چیز به خوبی کار می‌کند. گزاره‌ی محافظ مطمئن می‌شود که words[0] به مشکل برخورد نمی‌کند. اما آیا این کافی است؟ زمانی‌که برنامه‌نویسی می‌کنیم بایستی همیشه در پس ذهن خود این عبارت را داشته باشیم: «چه مشکلی می‌تواند پیش بیاید؟»

**تمرین ۲:** کدام خط بالا هنوز که هنوز است به خوبی محافظت نشده است؟ ببینید می‌توانید فایلی بسازید که باعث خطا در اجرای برنامه شود؟ سپس آن را با کد محافظ دیگری اصلاح کنید و در نهایت برنامه را با فایل جدید آزمایش کنید.

**تمرین ۳:** کد محافظ بالا را بدون دو گزاره‌ی مجزای if بازنویسی کنید. در عوض از یک عبارت منطقی ترکیبی و عملگر منطقی and به این منظور استفاده کرده و فقط یک گزاره‌ی شرطی بنویسید.

## واژگان فصل

**الایزینگ / Aliasing:**
وضعیتی که دو یا چند متغیر به یک آبجکت مشابه ارجاع داده می‌شوند.

**حائل / Delimiter:**
یک کاراکتر یا رشته که برای مشخص کردن محل جدا شدن رشته مورد استفاده قرار می‌گیرد.

**عنصر / المنت / Element:**
یکی از مقدارهای یک لیست (یا هر توالی دیگر)؛ آیتم نیز خوانده می‌شود.

**برابر /Equivalent:**
داشتن مقدارهای برابر.

**شاخص / Index:**
یک مقدار صحیح که نشان‌دهنده‌ی یک عنصر در یک لیست است.

**همسان / Identical:**
یک آبجکتِ واحد بودن (برابر بودن، جزئی از همسان بودن است، ولی برابر بودن مساوی با همسان بودن نیست).

**لیست / List:**
یک توالی از مقدارها.

**پیمایش یا گذار در لیست / List Traversal:**
دسترسی سلسله‌ای به هر یک از المنت‌های یک لیست.

**لیست تو در تو / Nested List:**
لیستی که خودش عنصری از لیست دیگر است.

**آبجکت / Object:**
چیزی که متغیری می‌تواند به آن ارجاع داده شود. یک آبجکت شامل یک نوع و یک مقدار می‌شود.

**ارجاع / Reference:**
ارتباط بین یک متغیر و مقدارش.

## تمرین‌ها

**تمرین ۴:** یک کپی از این فایل را از لینک زیر دانلود کنید:

www.py4e.com/code3/romeo.txt

**الف)** برنامه‌ای بنویسید که فایل romeo.txt را باز کرده و سپس آن را خط به خط بخواند. برای هر خط، آن را با استفاده از تابع split به لیستی از کلمات تبدیل کند.

**ب)** برای هر کلمه، بررسی کند که آیا کلمه از قبل در لیست وجود دارد یا خیر. اگر کلمه در لیست نیست، آن را به لیست اضافه کند.

**ج)** زمانی‌که کار برنامه تمام شد، نتایج را بر اساس حروف الفبا مرتب کرده و سپس چاپ نماید.

```
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:** برنامه‌ای بنویسید که داده‌های صندوق پستی را خوانده و سپس زمانی‌که خطی که با From شروع می‌شود را کشف کرد، آن را به کلمات مجزا با استفاده از تابع split بشکند. ما می‌خواهیم بدانیم که این پیام‌ها را چه کسی نوشته است. اسم اشخاص در هر خط، دومین کلمه در لیست کلمات ما خواهد بود.

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

ابتدا خط From را تجزیه و تحلیل می‌کنیم و سپس دومین کلمه از هر خط جذاب «= شامل From در ابتدای خود» را چاپ می‌کنیم. همچنین تعداد خطوط که با From و نه :From شروع شده‌اند را نیز چاپ می‌کنیم. خروجی زیر نمونه‌ی خوبی 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 که یک لیست از اعداد را از کاربر می‌خواست و سپس با تایپ کلمه done به دریافت ورودی خاتمه می‌داد و در نهایت مقدارهای بیشینه و کمینه‌ی لیست را چاپ می‌کرد را بازنویسی کنید. برنامه‌ای بنویسید که اعدادی که کاربر وارد می‌کند را در داخل یک لیست قرار داده و سپس با توابع max() و min() مقدار بیشینه و کمینه را بعد از اینکه حلقه تمام شد کشف و در نهایت چاپ کند.

```
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 ۹

# دیکشنری‌ها

## دیکشنری

یک دیکشنری چیزی شبیه به یک لیست - ولی کُلی‌تر - است. در یک لیستْ ما جایگاه‌های شاخص را داریم که با عدد صحیح مشخص می‌شوند؛ در دیکشنری، شاخص‌ها تقریبا هر نوعی – رشته، عدد صحیح و... – می‌توانند باشند.

می‌توانید دیکشنری را نقشه‌ای بین دسته‌ای از شاخص‌ها (که به آن‌ها کلید می‌گوییم) و دسته‌ای از مقادیر در نظر بگیرید. هر کلید، نقشه یا راهی به یک مقدار دارد. ارتباط کلی و مقدار را جفت کلید-مقدار یا گاهی آیتم می‌نامیم.

به عنوان مثال، ما یک دیکشنری ساخته‌ایم که کلمات انگلیسی را به کلمات اسپانیایی متصل می‌کند. در اینجا کلیدها و مقادیر از نوع رشته خواهند بود.

تابع dict یک دیکشنری جدید را – بدون هیچ آیتمی – می‌سازد. به این خاطر که dict نام یک تابع توکاری شده است، نباید از آن به عنوان اسم متغیر استفاده کرد.

```
>>> eng2sp = dict()
>>> print(eng2sp)
{}
```

آکولاد، {}، نمایانگر یک دیکشنری خالی است. برای اضافه کردن آیتم‌ها به دیکشنری بایستی که از قلاب استفاده کنید.

```
>>> eng2sp['one'] = 'uno'
```

خط بالا یک آیتم می‌سازد که نقشه‌ای از کلید 'one' به مقدار 'uno' در خود دارد. اگر ما دیکشنری را مجددا چاپ کنیم، ما جفت کلید-مقدار را – که با دو نقطه از هم جدا شده‌اند – خواهیم دید:

```
>>> print(eng2sp)
{'one': 'uno'}
```

این چینش در خروجی، می‌تواند برای ورودی هم استفاده شود. به عنوان مثال شما می‌توانید یک دیکشنری جدید با سه آیتم بسازید. اما اگر eng2sp را چاپ کنید، احتمالا جا خواهید خورد.

```
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
>>> print(eng2sp)
{'one': 'uno', 'three': 'tres', 'two': 'dos'}
```

ترتیب جفت کلید-مقدار مانند ترتیبی که وارد برنامه کردیم نیست. در اصل اگر شما همین مثال را روی سیستم خود اجرا کنید، احتمال دارد که نتیجه‌ی متفاوتی در خروجی داشته باشید. در حقیقت ترتیب آیتم‌های یک دیکشنری غیرقابل پیش‌بینی است.

این مساله، مشکلی را ایجاد نمی‌کند، چراکه المنت‌های یک دیشکنری هیچگاه با یک شاخصی که عدد صحیح باشد مشخص نمی‌شوند. در عوض شما از کلیدها برای پیدا کردن مقدار مرتبط استفاده می‌کنید.

```
>>> print(eng2sp['two'])
'dos'
```

کلید 'two' همیشه به مقدار 'dos' اشاره دارد در نتیجه اینکه جفتِ 'two' و 'dos' کجای دیکشنری باشند، تفاوتی را ایجاد نمی‌کند.

اگر کلیدی در دیشکنری موجود نباشد، در تلاش برای چاپ آن با خطا روبرو خواهید شد:

```
>>> print(eng2sp['four'])
KeyError: 'four
```

تابع len برای دیکشنری‌ها نیز کاربردی است و تعداد جفت کلید–مقدار را برمی‌گرداند:

```
>>> len(eng2sp)
3
```

عملگر in در دیکشنری‌ها هم مورد استفاده قرار می‌گیرد؛ in به شما می‌گوید که آیا کلید خاصی در دیشکنری وجود دارد یا خیر (به خاطر داشته باشید که in کلیدها را مرور می‌کند؛ اگر عبارتی جزو مقدارهای یک دیکشنری باشد، in توجهی به آن نمی‌کند):

```
>>> 'one' in eng2sp
True
>>> 'uno' in eng2sp
False
```

اگر می‌خواهید که ببینید آیا مقدار خاصی در دیکشنری وجود دارد یا خیر، بایستی از متد values استفاده کنید. این متد، مقادیر را به عنوان یک لیست برمی‌گرداند. در اینجا می‌توانید از in برای یافتن مقدار خاصی در لیست ساخته شده بهره ببرید:

```
>>> vals = list(eng2sp.values())
>>> 'uno' in vals
True
```

عملگر in از الگوریتم‌های متفاوتی برای دیکشنری و لیست‌ها استفاده می‌کند. برای لیست‌ها از الگوریتم جستجوی خطی بهره می‌برد. هرچقدر لیست بزرگتر شود، زمان جستجو طولانی‌تر خواهد شد. به عبارتی رابطه‌ی مستقیمی بین زمان جستجو و طول لیست برقرار است. برای دیکشنری‌ها، پایتون از یک الگوریتم به نام hash table یا

جدول هَش/درهم‌سازی استفاده می‌کند که ویژگی جالب توجهی دارد: عملگر in بدون توجه به تعداد آیتم‌های یک دیکشنری، تقریبا زمان یکسانی را برای جستجو صرف می‌کند. من در اینجا چرایی این موضوع در توابع هش را توضیح نمی‌دهم ولی می‌توانید اطلاعات بیشتر را از صفحه ویکی آن بخوانید:

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

**تمرین ۱:** [wordlist2]

برنامه‌ای بنویسید که کلمه‌ها را از words.txt بخواند و آن‌ها را به عنوان کلیدها در یک دیکشنری ذخیره کند. مهم نیست که مقدار چه چیزی باشد. سپس با استفاده از عملگر in چک کند آیا یک رشته در دیشکنری وجود دارد یا خیر.

## دیشکنری به عنوان دسته‌ای از شمارنده‌ها

فرض کنید که یک رشته را به شما داده‌اند و از شما می‌خواهند که تعداد دفعاتی که هر کاراکتر در این رشته نمایان شده را حساب کنید. چندین راه برای این کار وجود دارد:

I. می‌توانید ۲۶ متغیر بسازید؛ هر کاراکتر یک متغیر. سپس در رشته پیمایش کرده و برای هر کاراکتر یک عدد به متغیر مورد نظر اضافه کنید. احتمالا با استفاده از شرط زنجیروار.

II. می‌توانید لیستی با ۲۶ المنت درست کنید سپس هر کدام از کاراکترها را به یک شماره تغییر دهید (با استفاده از تابع توکاری‌شده‌ی ord) و در نهایت با استفاده از شماره – به عنوان شاخص – شمارنده مورد نظر را یک واحد افزایش دهید.

III. می‌توانید یک دیکشنری بسازید که کلیدهایش، کاراکترها و مقادیرش، شمارنده‌ی مربوط به هر کلید یا کاراکتر باشد. اولین باری که یک کاراکتر را می‌بینید، یک آیتم به دیکشنری اضافه می‌کنید. بعد

از آن در مواجه مجدد با کاراکتر مورد نظر، مقدار آیتم موجود را یک واحد افزایش دهید.

هر کدام از این گزینه‌ها، پاسخ یکسانی به سوال ما می‌دهند ولی پیاده‌سازی روش رسیدن به پاسخ در هر یک متفاوت است.

پیاده‌سازی، راهی برای اجرای یک محاسبه است؛ برخی پیاده‌سازی‌ها از بقیه بهترند. به عنوان مثال، مزیت پیاده‌سازی به روش سوم و با استفاده از دیکشنری این است که ما نیازی به دانستن حروفی که در رشته ظاهر می‌شوند نداریم بلکه خودِ برنامه جای خالی برای آن‌ها دارد. به عبارتی زمانی‌که برنامه به هر کدام از حروف برخورد کند، جای خالی برای آن‌ها ساخته و در درون دیکشنری قرارشان می‌دهد؛ بدون اینکه زحمت تعیین آن‌ها را از قبل کشیده باشیم.

کدِ ما احتمالا این شکلی خواهد بود:

```
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 حال محاسبه‌ی یک هیستوگرام هستیم؛ هیستوگرام به صورت آماری به تعدادی شمارنده یا تکرارها اشاره دارد.

حلقه‌ی for در رشته پیمایش می‌کند. در هر دور اگر متغیر c در درون دیشکنری وجود خارجی نداشته باشد، ما یک آیتم جدید با کلید c ساخته و مقدار اولیه را 1 قرار می‌دهیم. اگر c از قبل در دیکشنری وجود داشته باشد، ما مقدارش (d[c])را یک واحد افزایش می‌دهیم.

این خروجی برنامه است:

```
{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2,
't': 1}
```

هیستوگرام نشان می‌دهد که حروف a و b هر کدام یک بار و حرف o دوبار در رشته ظاهر شده است.

دیکشنری‌ها، متدی با نام get دارند که یک کلید و یک مقدارِ پیش‌فرض را می‌گیرد. اگر کلید در دیکشنری موجود باشد، get مقدارِ نظیربه‌نظیر یا منطبق با آن کلید را برمی‌گرداند؛ در غیر این صورت مقدار پیش‌فرض را برمی‌گرداند. به عنوان مثال:

```
>>> counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
>>> print(counts.get('jan', 0))
100
>>> print(counts.get('tim', 0))
0
```

ما می‌توانیم از get برای نوشته حلقه هیستوگرام خود استفاده کنید. متد get می‌تواند مواردی که کلید در دیکشنری وجود ندارد را به صورت خودکار مدیریت کنید. به همین خاطر در خطوطِ برنامه‌ی ما صرفه‌جویی خواهد شد و یک گزاره‌ی if کمتر خواهیم داشت:

```
word = 'brontosaurus'
d = dict()
for c in word:
    d[c] = d.get(c,0) + 1
print(d)
```

استفاده از متد get حلقه‌ی شمارنده را ساده می‌کند. این متد در پایتون بسیار رایج است و در ادامه کتاب بارها از آن استفاده می‌کنیم. حالا لازم است که شما نمونه‌ی کدی که با گزاره if و عملگر in نوشته شده بود را با متد get مقایسه کنید. هر دو یک کار را می‌کنند ولی یکی مختصرتر و مفیدتر.

## دیشکنری‌ها و فایل‌ها

یکی از موارد رایج در استفاده از دیکشنری‌ها شمردن تعداد کلماتی‌ست که در یک فایل متنی آمد است. بیایید با یک فایل بسیار ساده از کلمات متن رومئو و ژولیت شروع کنیم.

برای دسته‌ی اول از مثال‌ها، ما یک نسخه‌ی کوتاه و ساده شده از متن را انتخاب می‌کنیم که شامل نشانه‌های نقطه‌گذاری نیز نمی‌شود. بعدا ما روی متن‌ها با تمام نشانه‌ها 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
```

حالا یک برنامه پایتونی خواهیم نوشت که خطوط فایل را خوانده و سپس هر کدام را به لیستی از کلمات بشکَند. سپس با حلقه زدن در بین این لیست‌ها هر کلمه را، با استفاده از یک دیکشنری، بشمارد.

خواهید دید که ما دو حلقه‌ی for خواهیم داشت. حلقه‌ی بیرونی، خطوط را می‌خواند و حلقه‌ی درونی در بین کلمات آن خطِ خاصْ چرخ زده و عملیات انجام می‌دهد. این یک نمونه از حلقه‌های تو در تو است. چرا تو در تو؟ چونکه یک حلقه‌ی بیرونی و یک حلقه‌ی درونی داریم.

به این خاطر که در هر تکرارِ حلقه‌ی بیرونی، حلقه‌ی درونی یک دور کامل زده – تا زمانی‌که حلقه آیتم‌های مورد بررسی را تمام کند یا به نحوی بشکند و از دور خارج شود – ما حلقه‌ی درونی را حلقه‌ی سریع‌تر و حلقه‌ی بیرونی را حلقه‌ی کندتر در نظر می‌گیریم.

ترکیب دو حلقه‌ی تو در تو به ما این اطمینان را می‌دهد که تمام کلمات و خطوط فایل ورودی شمرده خواهند شد.

```
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: http://www.py4e.com/code3/count1.py
```

زمانی‌که برنامه را اجرا می‌کنیم، با یک سری کلمه و تکرارهای خامی مواجه می‌شویم که هیچ ترتیب خاصی ندارند. فایل romeo.txt را از اینجا بگیرید:

http://www.py4e.com/code3/romeo.txt

```
python count1.py
Enter the file name: romeo.txt
{'and': 3, 'envious': 1, 'already': 1, 'fair': 1,
'is': 3, 'through': 1, 'pale': 1, 'yonder': 1,
'what': 1, 'sun': 2, 'Who': 1, 'But': 1, 'moon': 1,
'window': 1, 'sick': 1, 'east': 1, 'breaks': 1,
'grief': 1, 'with': 1, 'light': 1, 'It': 1, 'Arise': 1,
'kill': 1, 'the': 3, 'soft': 1, 'Juliet': 1}
```

گشتن در یک دیکشنری برای پیدا کردن کلمات رایج، زمانی‌که ترتیبی وجود ندارد، کار زیاد راحتی نیست. به همین خاطر لازم است که ما مقداری کدهای پایتونی اضافه کنیم که ما را راحت‌تر به نتیجه‌ی دلخواهمان برساند.

## حلقه زدن و دیکشنری‌ها

اگر از یک دیکشنری به عنوان توالی در یک گزاره‌ی for استفاده می‌کنید، گزاره‌ی for در بین کلیدهای دیکشنری پیمایش خواهد کرد. برای درک بهتر موضوع به مثال زیر نگاهی بیندازید؛ حلقه‌ی زیر کلیدها و مقدار مرتبط با آن‌ها را یکی یکی چاپ می‌کند:

```
counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
for key in counts:
    print(key, counts[key])
```

خروجی آن چیزی شبیه به این خواهد بود:

```
jan 100
chuck 1
annie 42
```

البته لازم است تکرار کنم که کلیدها به ترتیب خاصی نیستند.

ما می‌توانیم از این الگو برای پیاده‌سازی حلقه‌های مختلفی که پیش‌تر توضیح دادیم استفاده کنیم. به عنوان مثال اگر ما بخواهیم که تمام مدخل‌های یک دیشکنری را، با مقدار بالاتر از ده، پیدا کنیم، می‌توانیم از کد زیر استفاده کنیم:

```
counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
for key in counts:
    if counts[key] > 10 :
        print(key, counts[key])
```

حلقه‌ی for در بین کلیدهای دیکشنری پیمایش کرده، در نتیجه ما بایستی از عملگر شاخص برای به‌دست آوردن مقدار مرتبط با هر کلید استفاده کنیم. خروجی چیزی شبیه به این خواهد بود:

```
jan 100
annie 42
```

در اینجا تنها مدخل‌هایی که مقداری بیش از ده دارند را خواهیم دید.

اگر بخواهید که کلیدها را به ترتیب الفبا چاپ کنید، بد نیست لیستی از کلیدهای موجود در دیکشنری را با استفاده از متد keys بگیرید. متد keys در آبجکت دیکشنری موجود است. سپس به لیست تازه متولد شده، ترتیبی بدهید و در نهایت با پیمایش در لیست، جفت کلید-مقدار را چاپ کنید:

```
counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
lst = list(counts.keys())
print(lst)
lst.sort()
for key in lst:
    print(key, counts[key])
```

خروجی شبیه به این خواهد بود:

```
['jan', 'chuck', 'annie']
annie 42
chuck 1
jan 100
```

ابتدا با استفاده از متد keys ما لیستی نامرتب از کلیدها را ساختیم. سپس از طریق یک حلقه for ما لیست مرتب کلید-مقدار را چاپ کردیم.

## تحلیل پیشرفته‌ی متن

در مثال بالا ما از متن romeo.txt استفاده کردیم. تا جایی که امکانش بود فایل را ساده کردیم و همه علائم نشانه‌گذاری را حذف نمودیم. متن اصلی شامل مقدار زیادی از این علائم می‌شد:

```
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 پایتون از تابع split به دنبال فاصله‌های خالی گشته و سپس بر اساس فضاهای خالی بین کلمه‌ها آن‌ها را پیدا می‌کند، پس کلمه‌ی soft! و soft دو مدخل مجزا در دیکشنری به خود اختصاص خواهند داد.

همچنین از آن‌جایی که حروف اولِ هر جمله، حروف بزرگ هستند، کلمه‌های یکسان که تنها یکی از حروف آن‌ها در بزرگی و کوچکی متفاوت است به عنوان دو کلمه‌ی مجزا شناخته می‌شوند. مثلا کلمه‌ی Who و who دو مدخل در دیشکنری ما خواهند داشت.

ما می‌توانیم هر دو مشکل را با استفاده از متدهای رشته low و punctuation و translate حل کنیم.

```
line.translate(str.maketrans(fromstr, tostr, deletestr))
```

در مثال بالا ما چند متغیر داریم. به سه متغیر در درونی‌ترین پرانتز می‌پردازیم؛ این خط کد با آن‌ها چه کار می‌کند؟ کاراکتر fromstr را با کاراکتر tostr که در همان موقعیت هست، جابجا می‌کند و تمام کاراکترهایی که در deletestr هست را حذف می‌کند. در ضمن در متد translate همه‌ی این پارامترها اجباری نیستند.

ما table را مشخص نکردیم ولی با استفاده از پارامتر deletechars تمام علائم نشانه‌گذاری را حذف کردیم. حتی از پایتون خواستیم که لیستی از علائم که به عنوان punctuation می‌شناسد را به ما ارائه دهد:

```
>>> import string
>>> string.punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

پارامترهای که توسط translate در پایتون ۲ استفاده می‌شدند، متفاوت بودند.

حالا تغییرات زیر را به برنامه دادیم:

```
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()
    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: http://www.py4e.com/code3/count2.py
```

بخشی از یادگیری «هنر پایتون» یا «فکر کردن به روش پایتونی» درک این موضوع است که پایتون در اغلب موارد دارای قابلیت‌های توکاری شده برای انجام آنالیزِ داده است. به مرور شما مثال‌های کافی از کدها را خواهید دید و اسناد کافی برای دانستن کدهایی که دیگران نوشته‌اند و شما می‌توانید از آن‌ها بهره ببرید خواهید خواند.

خروجی زیر یک نمونه خلاصه شدن از خروجی برنامه‌ی ماست:

```
Enter the file name: romeo-full.txt
{'swearst': 1, 'all': 6, 'afeard': 1, 'leave': 2, 'these':
2,
'kinsmen': 2, 'what': 11, 'thinkst': 1, 'love': 24, 'cloak':
1,
a': 24, 'orchard': 2, 'light': 5, 'lovers': 2, 'romeo': 40,
'maiden': 1, 'whiteupturned': 1, 'juliet': 32, 'gentleman':
1,
'it': 22, 'leans': 1, 'canst': 1, 'having': 1, ...}
```

با نگاه به خروجی می‌بینیم که هنوز که هنوز است موارد مشکل‌دار در بین آن وجود دارد و می‌توانیم با استفاده از پایتون به دنبال آن‌ها گشته و نابودشان کنیم. برای این کار بهتر است که در خصوص تاپل‌ها/tuple یاد بگیریم. زمانی‌که آن‌ها را یاد گرفتیم باز به سراغ این برنامه می‌آییم.

## اشکال‌زدایی

همینطور که با داده‌های بزرگتر کار می‌کنید، اشکال‌زدایی با استفاده از چاپ کردن و بررسی دستی، سخت‌تر می‌شود. در اینجا برای شما چند پیشنهاد برای اشکال‌زدایی داده‌های بزرگ داریم:

### خروجی را کوچک کنید

اگر امکانش وجود دارد، مجموعه داده‌ها را کوچکتر کنید. به عنوان مثال اگر برنامه یک فایل متنی را می‌خواند، با ده خط اول یا مثال‌های کوچکتری که

به چشمتان می‌آید شروع کنید. می‌توانید که خود فایل را ویرایش کنید؛ یا راه بهتر این است که برنامه را به صورتی تغییر دهید که تنها n خط اول را بخواند.

اگر برنامه با مشکل مواجه می‌شود، می‌توانید n را به کوچکترین مقداری که خطا را نشان می‌دهد بشکنید. سپس به مرور اندازه‌ی آن را افزایش بدهید تا بالاخره خطا را یافته و مشکل را حل کنید.

### خلاصه و نوع داده را چک کنید

به جای چاپ کردن کل داده، سعی کنید با خلاصه‌ای از داده دست و پنجه نرم کنید. به عنوان مثال: تعداد آیتم‌های موجود در یک دیکشنری یا تعداد کل عددهای یک لیست.

یک دلیل عمده‌ی «خطاهای در حین اجرا» این است که یک مقدار، «نوع» درستی ندارد. برای اشکال‌زدایی این مدل خطاها، چاپ کردن نوع یک مقدار کافی است. به عنوان مثال: تلاش برای تقسیم یک عدد صحیح بر یک رشته در مثال زیر؛ مقسوم‌علیه به جای اینکه عدد صحیح باشد، رشته است و همین مساله، مشکل ایجاد می‌کند:

```
>>> Divisor = input('Enter divisor: ')
Enter divisor: 3
>>> 15 / Divisor
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for /: 'int' and
'str'
>>> type(Divisor)
<class 'str'>
>>>
```

### کد با قابلیت بررسیِ خود بنویسید

گاهی می‌توانید کدی بنویسید که خطاها را به صورت خودکار بررسی کند. برای نمونه اگه شما مقدار میانگین عددها را محاسبه می‌کنید، می‌توانید بررسی کنید که مقدار محاسبه شده بزرگتر از بزرگترین المنتِ موجود در لیست و کوچکتر از کوچکترین‌شان نباشد. این کار «بررسیِ سلامت» خوانده می‌شود و نتایجی که کاملا پرت هستند را بررسی و کشف می‌کند.

نمونه‌ی دیگر، بررسی نتیجه دو محاسبه‌ی مختلف برای اطمینان از یکسان بودن پاسخ‌ها است. این کار «بررسی انسجامِ» پاسخ نامیده می‌شود.

### چاپ کردن قشنگِ خروجی

با شکل دادن به خروجیِ اشکال‌زدایی، مشاهده‌ی یک خطا را راحت‌تر کنید.

به خاطر داشته باشید، زمانی‌که برای ساخت چارچوب برنامه اختصاص می‌دهید می‌تواند باعث کم شدن زمانی‌که قرار است برای اشکال‌زدایی بگذارید شود، پس فونداسیون را قوی ببندید.

## واژگان فصل

**دیکشنری / Dictionary:**
نقشه‌ی راهی از یک سری کلید به مقادیر مرتبط با آن‌ها.

**جدول درهم‌سازی / جدول هش / Hashtable:**
الگوریتمی که برای پیاده‌سازی دیکشنری‌های پایتون استفاده شده است.

**تابع هش / Hash Function:**
تابعی که توسط یک جدول هش برای محاسبه‌ی مکان یک کلید استفاده می‌شود.

**هیستوگرام / Histogram:**
دسته‌ای از شمارنده‌ها.

**اجرا / پیاده‌سازی / Implementation:**
راهی برای انجام یک محاسبه.

**آیتم / Item:**
نام دیگری برای جفت کلید-مقدار.

**کلید / Key:**
یک آبجکت که در یک دیکشنری به عنوان بخش اولِ جفتِ کلید-مقدار ظاهر می‌شود.

**جفت کلید-مقدار/ Key-Value Pair:**
نمایشی از نقشه‌ی راهِ یک کلید به یک مقدار.

**جستجو / Lookup:**
عملیاتی در دیکشنری که یک کلید را گرفته و مقدار نظیر به نظیر آن را پیدا می‌کند.

**حلقه‌های تو در تو / Nested Loops:**
زمانی‌که یک یا چند حلقه در «داخل» یک حلقه‌ی دیگر قرار دارد، به این نوع حلقه‌ها، حلقه‌های تو در تو می‌گوییم. در هر یک دور حلقه‌ی بیرونی، حلقه‌ی درونی کل پروسه‌ای که باید انجام دهد تا از حلقه خارج شود را یک مرتبه انجام می‌دهد.

**مقدار / Value:**
یک آبجکت که در یک دیشکنری به عنوان بخش دومِ جفتِ کلید-مقدار ظاهر می‌شود. این مورد، نوع ویژه‌ای از «مقدار» است که پیشتر در خصوص آن صحبت می‌کردیم.

## تمرین‌ها

**تمرین ۲:** برنامه‌ای بنویسید که هر پیامِ ایمیل را براساس روزِ هفته دسته‌بندی کند. برای انجام این کار، بایستی چشم بر خطوطی که با From شروع می‌شوند داشته باشید. سپس سومین کلمه را استخراج کنید و برای هر روز هفته شمارنده بگذارید. در انتها برنامه بایستی محتویات دیکشنری را چاپ کند (ترتیب مهم نیست).

خط نمونه:

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

نمونه‌ی اجرا شده:

```
python dow.py
Enter a file name: mbox-short.txt
{'Fri': 20, 'Thu': 6, 'Sat': 1}
```

**تمرین ۳:** برنامه‌ای بنویسید که لاگ یک ایمیل را بخواند، سپس یک هیستوگرام با استفاده از دیکشنری برای شمردنِ تعداد پیام‌هایی که برای هر آدرس ایمیل آمده بسازد. در انتها 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:** کدی به برنامه بالا اضافه کنید که نشان دهد چه کسی بیش از بقیه ایمیل داشته است.

در انتها و پس از پایان کارِ خواندن داده‌ها، از یک حلقه‌ی بیشینه، برای یافتن کسی که بیشترین پیام‌ها را داشته و تعداد این پیام‌ها، استفاده کنید.

```
Enter a file name: mbox-short.txt
cwen@iupui.edu 5
```

```
Enter a file name: mbox.txt
zqian@umich.edu 195
```

**تمرین ۵:** برنامه‌ای بنویسید که نام‌های دامنه (به جای آدرس) را ثبت می‌کند. به عبارتی، جایی که نامه از آن رسیده را به جای شخصی که نامه را فرستاده ثبت کند. در نهایت دیکشنری را چاپ کند.

```
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 ۱۰

# تاپل‌ها

## تاپل‌ها تغییرناپذیرند

تاپل، به‌مانند لیست، یک توالی از مقادیر است. مقادیری که در یک تاپل ثبت شده‌اند، از هر نوعی می‌توانند باشد و جایگاهشان توسط اعداد صحیح مشخص می‌شود. تفاوت مهم آن‌ها با لیست، تغییرناپذیر بودنشان است.

یک تاپل، لیستی از مقادیر است که با کاما از هم جدا شده‌اند:

```
>>> t = 'a', 'b', 'c', 'd', 'e'
```

با وجود اینکه قرار دادن این لیست در پرانتز ضروری نیست، ولی این کار به تشخیص سریعِ تاپل کمک می‌کند، پس بهتر است که آن را در داخل پرانتز قرار دهید:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
```

برای ساخت یک تاپل با یک المنت، بایستی که کامای آخر را اضافه کنید:

```
>>> t1 = ('a',)
>>> type(t1)
<type 'tuple'>
```

اما اگر در گزاره‌ی بالا کاما نگذاریم چه می‌شود؟ بدون کاما، پایتون فکر می‌کند که ('a') یک عبارت با یک رشته در پرانتز است که در نهایت برای پایتون به معنای یک رشته خواهد بود.

```
>>> t2 = ('a')
>>> type(t2)
<type 'str'>
```

راه دیگر برای ساخت تاپل، استفاده از تابع توکاری شده‌ی tuple است. اگر از این تابع بدون هیچ آرگیومنتی استفاده کنید، در نهایت یک تاپل خالی خواهید داشت:

```
>>> t = tuple()
>>> print(t)
()
```

اگر آرگیومنت، یک توالی (مانند رشته، لیست یا تاپل) باشد، نتیجه‌ی فراخوانی tuple یک تاپل با المنت‌هایی از آن توالی خواهد بود:

```
>>> t = tuple('lupins')
>>> print(t)
('l', 'u', 'p', 'i', 'n', 's')
```

به این خاطر که tuple یک نامِ با معنی برای پایتون است، بایستی از قرار دادن آن به عنوان نام متغیر خودداری کنید.

اغلب عملگرها روی تاپل‌ها هم کار می‌کنند. عملگر قلاب، یک المنت را نمایان می‌کند:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> print(t[0])
'a'
```

و عملگر قاچ‌زدن، بازه‌ای از المنت‌ها را انتخاب می‌کند:

```
>>> print(t[1:3])
('b', 'c')
```

اما اگر شما سعی در تغییر یکی از المنت‌ها کنید، با یک پیغام خطا روبرو خواهید شد:

```
>>> t[0] = 'A'
TypeError: object doesn't support item assignment
```

شما نمی‌توانید المنت‌های یک تاپل را تغییر دهید، ولی می‌توانید آن تاپل را با یکی دیگر عوض کنید:

```
>>> t = ('A',) + t[1:]
>>> print(t)
('A', 'b', 'c', 'd', 'e')
```

## مقایسه تاپل‌ها

عملگر مقایسه همانگونه که برای بقیه توالی‌ها کاربرد دارد، برای تاپل هم می‌تواند مورد استفاده قرار بگیرد. پایتون، مقایسه را با اولین المنت از هر توالی شروع می‌کند. اگر آن‌ها مساوی بودند به سراغ المنت بعدی می‌رود و این کار تا زمانی‌که المنت‌های متفاوتی را پیدا کند، ادامه خواهد داشت. ولی به محض رسیدن به یک تفاوت، المنت‌های بعد از آن، هرچقدر هم که بزرگ باشند، دیگر در نظر گرفته نخواهند شد:

```
>>> (0, 1, 2) < (0, 3, 4)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
```

تابع sort نیز به همین روش کار می‌کند و اساسا با اولین المنت ترتیب را انجام می‌دهد ولی زمانی‌که با مقدار برابری مواجه شد، ترتیب را با المنت دوم و الی آخر انجام خواهد داد.

این قابلیت عاریت گرفته شده از الگوریتم DSU است. DSU مخفف Decorate Sort Undecorate است.

**Decorate**
یک توالی که ساخته شده از لیستی از تاپل‌هاست. یک یا چند کلید برای مرتب شدن، بر هر کدام از المنت‌های این توالی مقدم هستند،

**Sort**
لیستی از تاپل‌ها که تابع درونی sort روی آن‌ها اعمال شده و مرتب شده‌اند، و

**Undecorate**
استخراج المنت‌های مرتب شده‌ی این توالی.

کمی گیج‌کننده است. بگذارید مثالی بزنم؛ فرض کنید که شما لیستی از کلمات دارید و می‌خواهید که آن را از بزرگتر به کوچک‌تر 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: http://www.py4e.com/code3/soft.py
```

حلقه‌ی اول، یک لیست از تاپل‌ها می‌سازد. هر تاپل دو جزء دارد. جزء اول، طول کلمه است. ولی کدام کلمه؟ کلمه‌ای که جزء دوم تاپل است. به عبارتی تاپلِ ما تشکیل شده است از: ۱) طول کلمه و ۲) خودِ کلمه. این بخش D از DSU است. شما یک کلید در ابتدای تاپل دارید و هدفتان مرتب کردن لیست با استفاده از این کلیدِ مقدم است.

متد sort در مقایسه این تاپل‌ها، ابتدا اولین المنت که همان طول کلمه باشد را مقایسه می‌کند و اگر طول دو کلمه مساوی بود، به سراغ مقایسه المنت‌های دوم، که همان کلمه باشد، می‌رود. کلمه‌ی کلیدی در آرگیومنت آن یعنی reverse=True به sort می‌گوید که بایستی مقایسه حالت نزولی (از بالا به پایین) را داشته باشد. این بخش S از DSU به حساب می‌آید.

حلقه‌ی دوم در لیست تاپل‌ها پیمایش کرده و لیستی از کلمات، به ترتیبی که طول آن‌ها از زیاد به کم باشد، می‌سازد. در اینجا با دو کلمه‌ی چهار حرفی what و soft طرف هستیم. با توجه به یکسان بودن المنت اول – طول کلمه – تابع sort به سراغ مقایسه المنت دوم می‌رود. w از s بزرگتر است و با توجه به اینکه لیست ما از بزرگ به کوچک است، در نتیجه what قبل از soft می‌آید. ساخت یک لیست از المنت‌های مرتب شده هم بخش U از DSU است.

خروجی برنامه شبیه به این خواهد بود:

```
['yonder', 'window', 'breaks', 'light', 'what',
'soft', 'but', 'in']
```

وزن شعر ما در لیستی که پایتون درست کرد به هم ریخت چرا که پایتون کاری به وزن شعر ندارد، تنها بنا به درخواست ما می‌خواهد که کلمه‌ها را بر اساس طول‌شان از بیشتر به کمتر مرتب کند.

## گمارش تاپل

یکی از ویژگی‌ها مرتبط به متن در زبان پایتون قابلیت داشتن یک تاپل در سمت چپ و یک گزاره‌ی گمارش است. این کار به شما اجازه می‌دهد تا با یک گزاره‌ی گمارش چند متغیر را در آن واحد تغییر دهیم.

در این مثال ما یک لیست با دو المنت داریم (که خود یک توالی به حساب می‌آید). در خط دوم، اولین و دومین المنت – که در دل متغیر m قرار دارند – را به متغیرهای x و y در یک تک گزاره اختصاص می‌دهیم.

```
>>> m = [ 'have', 'fun' ]
>>> x, y = m
>>> x
'have'
>>> y
'fun'
>>>
```

پایتون کد بالا را اینگونه می‌خواند و تفسیر می‌کند:

```
>>> m = [ 'have', 'fun' ]
>>> x = m[0]
>>> y = m[1]
>>> x
'have'
>>> y
'fun'
>>>
```

به عبارتی وقتی ما یک تاپل را در سمت چپ گزاره‌ی گمارش قرار می‌دهیم، پرانتزها را از قلم می‌اندازیم ولی می‌توانیم برای وضوح بیشترِ متن، آن‌ها را اضافه کنیم، بدون اینکه مشکلی برای برنامه پیش بیاید:

```
>>> m = [ 'have', 'fun' ]
>>> (x, y) = m
>>> x
'have'
>>> y
'fun'
>>>
```

یک استفاده‌ی هوشمندانه از گزاره‌ی گمارشِ تاپل‌ها این است که به ما اجازه می‌دهد دو متغیر را در یک گزاره‌ی واحد با هم جابجا کنیم:

```
>>> a, b = b, a
```

هر دو طرف این گزاره تاپل هستند ولی سمت چپ یک تاپل از متغیرها و سمت راست یک تاپل از عبارت‌ها است. هر مقدار در سمت راست به مقدار مرتبط در سمت چپ اختصاص داده می‌شود. تمام عبارت‌ها در سمت راست قبل از اینکه گمارش صورت پذیرد ارزش‌یابی می‌شوند (به عبارتی قبل از گمارش مقدار آن‌ها توسط پایتون مشخص می‌شود و این نوع گمارش خللی در کار برنامه به وجود نمی‌آورد).

تعداد متغیرها در سمت چپ و تعداد مقدارها در سمت راست بایستی که یکسان باشد:

```
>>> a, b = 1, 2, 3
ValueError: too many values to unpack
```

سمت راست می‌تواند هر نوعی از توالی (رشته، لیست یا تاپل) باشد. به عنوان مثال برای بخش کردن یک آدرس ایمیل به نام کاربری و نام دامنه، می‌توانید کد زیر را بنویسید:

```
>>> addr = 'monty@python.org'
>>> uname, domain = addr.split('@')
```

مقدار بازگردانده شده توسط split یک لیست با دو المنت خواهد بود؛ المنت اول به uname و المنت دوم به domain اختصاص پیدا خواهد کرد.

```
>>> print(uname)
monty
>>> print(domain)
python.org
```

## دیشکنری و تاپل‌ها

دیشکنری‌ها یک متدی به نام items دارند که لیستی از تاپل‌ها را برمی‌گرداند. در این لیست، هر تاپل یک جفتِ کلید-مقدار خواهد بود:

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> t = list(d.items())
>>> print(t)
[('b', 1), ('a', 10), ('c', 22)]
```

و همانگونه که از یک دیکشنری انتظار می‌رود، آیتم‌ها ترتیب خاصی نخواهند داشت.

هرچند، از آنجایی که لیست تاپل‌ها خود یک لیست است و تاپل‌ها قابل مقایسه هستند، می‌توانیم این لیست تاپل‌ها را نظم بدهیم. تبدیل یک دیکشنری به لیستی از تاپل‌ها به ما این امکان را می‌دهد که محتویات دیکشنری را با توجه به کلیدهایش مرتب کنیم.

```
>>> d = {'a':10, 'b':1, '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 الفبا مرتب شده است.

## گمارش‌های چندگانه در دیکشنری‌ها

با ترکیب items و گمارش و حلقه for می‌توانیم الگوی زیبایی برای پیمایش در بین کلیدها و مقدارهای یک دیکشنری در یک تک حلقه بنویسیم:

```
for key, val in list(d.items()):
    print(val, key)
```

این حلقه دو متغیر تکرار key و val دارد، چرا که items لیستی از تاپل‌ها را برمی‌گرداند. این تاپل‌ها خود دارای دو جزء کلید و مقدار می‌شوند. متغیر تکرار با توجه به دوتایی بودن المنت‌های لیست ما، می‌تواند از دو قسمت تشکیل شود. ما key را برای بخش اول این المنت‌ها که همان کلیدهای دیکشنری هستند و val را برای بخش دوم این المنت‌ها که همان مقدارهای آن کلید هستند به عنوان متغیرهای تکرار در گزاره‌ی for قرار دادیم. key, val یک گمارش تاپل است که به صورت متوالی جفت کلید-مقدار را وارسی می‌کند (تا به انتهای لیست برسد).

برای هر تکرار در طول حلقه هر دوی key و value پیش از جفت کلید-متغیر بعدی در دیشکنری می‌آیند (در هم).

خروجی حلقه چیزی شبیه به این خواهد بود:

```
10 a
22 c
1 b
```

می‌بینید که درهم است. به عبارتی هیچ ترتیب خاصی ندارد.

اگر ما این دو تکنیک را با هم ترکیب کنیم، می‌توانیم محتویات یک دیکشنری را بر اساس ترتیب مقدارهایش چاپ کنیم.

برای انجام این کار، ابتدا لیستی از تاپل‌ها می‌سازیم که هر تاپل شامل (مقدار، کلید) شود. متد items به ما لیستی از (کلید، مقدار) را می‌دهد، اما ما قرار است که ترتیب را بر اساس «مقدار» تنظیم کنیم. زمانی‌که لیستی از تاپل‌های مقدار-کلید را ساختیم، ترتیب دادن به آن‌ها و چاپ‌شان کار ساده‌تری خواهد بود.

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> l = list()
>>> for key, val in d.items() :
...     l.append( (val, key) )
...
>>> l
[(10, 'a'), (22, 'c'), (1, 'b')]
>>> l.sort(reverse=True)
>>> l
[(22, 'c'), (10, 'a'), (1, 'b')]
>>>
```

اگر مراحل را به درستی جلو برویم، لیستی از تاپل‌ها خواهیم داشت که «مقدار»، جزء اول آن‌ها خواهد بود. در نهایت با ترتیب دادن به لیست، محتویات دیکشنری بر اساس مقدارهایشان منظم خواهند شد.

## رایج‌ترین کلمه

به مثال قبلی و متن رومئو و ژولیت برگردیم، می‌توانیم با استفاده از این تکنیک، ده کلمه‌ی رایج در متن را به صورت زیر چاپ کنیم:

```
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: http://www.py4e.com/code3/count3.py
```

بخشِ اول برنامه، فایل را می‌خواند و به محاسبه‌ی کلمه‌ها و تکرارهایشان در آن می‌پردازد؛ اما به جای چاپ بدون ترتیب آن‌ها، در بخش دوم لیستی از تاپل‌ها (مقدار، کلید) می‌سازد و سپس این لیست را به ترتیب از بالا به پایین مرتب می‌کند.

از آن‌جایی که در این تاپل‌ها، مقدار در ابتدای کار قرار دارد، برای مقایسه از آن استفاده می‌کنیم. اگر بیش از یک تاپل با مقدار برابر وجود داشته باشد، پایتون برای ترتیب دادن به آن‌ها به سراغ المنت دوم (یعنی کلید) می‌رود. در نتیجه تاپل‌هایی که مقدار یکسانی دارند، به ترتیب الفبا چیده خواهند شد.

در انتها ما یک حلقه‌ی for می‌نویسیم که چندین گزاره‌ی گمارش را اجرا کرده و ده کلمه‌ی رایج را چاپ کند. این کار توسط قاچی از لیست اصلی صورت می‌پذیرد (lst[:10]).

خروجی ما در نهایت چیزی شبیه به این خواهد بود:

```
61 i
42 and
40 romeo
34 to
34 the
32 thou
32 juliet
30 that
29 my
24 thee
```

دیدید که تحلیل و آنالیز داده‌های پیچیده به راحتیِ یک برنامه‌ی نوزده خطی در پایتون بود؛ به همین دلیل پایتون برای کاوش اطلاعات، یک زبان برنامه‌نویسی مناسب و انتخاب معقولانه‌ای است.

## استفاده از تاپل‌ها به عنوان کلید در دیکشنری‌ها

با توجه به اینکه تاپل‌ها hashable (به‌طور ساده می‌توان گفت که داده‌های غیرقابل تغییر) هستند اگر بخواهیم یک کلید ترکیبی برای استفاده در دیشکنری بسازیم بایستی که از یک تاپل به عنوان کلید استفاده کنیم.

اگر بخواهیم یک دفتر راهنمای تلفن بسازیم با یک کلید مرکب روبرو خواهیم شد که از فامیل و نام شخص به شماره‌ی تلفن آن مسیر باز می‌کند. فرض کنید که ما متغیرهای last و first و number را مشخص کرده‌ایم. ما می‌توانیم گزاره‌ی گمارش یک دیکشنری را به این صورت بنویسیم:

```
directory[last,first] = number
```

عبارت موجود در براکت یک تاپل است. ما می‌توانیم از گمارش تاپل در یک حلقه for برای پیمایش در این دیکشنری نیز استفاده کنیم:

```
for last, first in directory:
    print(first, last, directory[last,first])
```

حلقه در بین کلیدهای موجود در directory – که تاپل هستند – پیمایش کرده و هر المنت را به last و first اختصاص داده و در نهایت شماره تلفن نظیر به نظیر آن را چاپ می‌کند.

## توالی‌ها: رشته‌ها، لیست‌ها و تاپل‌ها - خدای من!

ما در این بخش روی لیست‌هایی از تاپل‌ها تمرکز کردیم، ولی تقریبا تمام مثال‌های این فصل برای لیست‌هایی از لیست‌ها، تاپل‌هایی از تاپل‌ها و تاپل‌هایی از لیست‌ها نیز صادق است. اما به جای شمردن ترکیب‌های احتمالی بهتر است که کار را ساده کنیم و بگوییم توالی‌هایی از توالی‌ها.

در بسیاری از مواقع، نوع متفاوت توالی‌ها (رشته‌ها، لیست‌ها و تاپل‌ها) می‌توانند در جای هم مورد استفاده قرار بگیرند. ولی چگونه، کدام را برای استفاده انتخاب کنیم؟

برای شروع به سراغ رشته می‌رویم. رشته‌ها محدودتر از بقیه‌ی توالی‌ها هستند به این خاطر که المنت‌ها بایستی که کاراکترها باشند. همچنین رشته‌ها تغییرناپذیرند. اگر نیاز به تغییر در کاراکترهای یک رشته دارید (و نه ساخت یک رشته‌ی جدید) شاید بهتر باشد که به سراغ یک لیست از کاراکترها بروید.

لیست‌ها رایج‌تر از تاپل‌ها هستند، چراکه آن‌ها قابل تغییرند. ولی شرایط خاصی هم وجود دارد که شاید بخواهید از تاپل‌ها استفاده کنید:

I. در برخی جاها مثل وقتی که از گزاره‌ی return استفاده می‌کنیم، ساخت یک تاپل ساده‌تر از لیست است. در بقیه موارد شاید که لیست را ترجیح دهید.

II. اگر بخواهید از یک توالی به عنوان یک کلید دیکشنری استفاده کنید، بایستی از یک نوع غیرقابل تغییر مثل یک تاپل یا رشته استفاده کنید.

III. اگر در حال فرستادن یک توالی به عنوان یک آرگیومنت به یک تابع هستید، استفاده از تاپل‌ها با توجه به الایزینگ و رفتارهای غیرمنتظره مرتبط با آن بهتر خواهد بود.

به این خاطر که تاپل‌ها غیرقابل تغییر هستند، متدهایی مثل sort یا reverse در دسترس شما نخواهند بود. این متدها قابلیت تغییر لیست‌ها را دارند. با این حال پایتون توابع توکاری شده مثل sorted و reversed را در درون خود دارد که می‌تواند هر توالی‌ای را به عنوان پارامتر دریافت کرده و یک توالی جدید با المنت‌ها یکسان ولی ترتیب متفاوت تولید کند.

## اشکال‌زدایی

لیست‌ها، دیکشنری‌ها و تاپل‌ها عموما به عنوان ساختمان یا ساختار داده ( data structure) شناخته می‌شوند؛ در این فصل ما ساختارهای داده مرکب مانند لیست‌های تاپل‌ها و دیکشنری‌هایی که شامل تاپل‌ها به عنوان کلید و لیست‌ها به عنوان مقدار می‌شوند را دیدیم. ساختارهای داده مرکب، مفید و کارا هستند ولی مستعد خطای ظاهری نیز می‌باشند. این خطاها به خاطر استفاده از نوع، اندازه یا ترکیب اشتباهی که ساختار داده دارد اتفاق می‌افتد.

به عنوان مثال اگر شما منتظر یک لیست با یک عدد صحیح هستید، و من به شما یک عدد صحیح خالی (و نه در دلِ یک لیست) تحویل دهم، برنامه کار نخواهد کرد.

زمانی‌که در حال اشکال‌زدایی یک برنامه هستید، به‌خصوص اگر روی یک باگِ سرسخت کار می‌کنید، این چهار مورد را مد نظر داشته باشید:

### خواندن

کد خود را آزمایش کنید و آن را برای خودتان بلند بخوانید. ببینید که کد همان حرفی که منظور شماست را می‌زند یا خیر.

### اجرا

تغییرات کوچکی را به کد خود بدهید و، سپس با توجه به آن تغییرات، کد را اجرا کرده و رفتار برنامه را وارسی کنید. اگر یاد بگیرید که عبارت درستی را در جای درست نمایش دهید، مشکل نمایان می‌شود. با این حال گاهی لازم است زمان صرف ساخت یک چارچوب، برای پیدا کردن مشکل کنید.

### غور کردن

کمی فکر کنید. چه خطایی است؟ سینتکس؟ در حین اجرا؟ معنایی؟ چه اطلاعاتی از پیام خطا یا خروجی برنامه استخراج می‌شود؟ چه نوع خطایی ممکن است باعث به وجود آمدن این مشکل شود؟ آخرین چیزی که تغییر دادید و بعد از آن برنامه به مشکل برخورد کرد چه چیزی بود؟

### عقب‌نشینی کردن

گاهی لازم است که تغییرات اخیر را بازگردانده و عقب‌نشینی کنید تا برنامه به حالت عملیاتیِ قبلش بازگردد. شاید اینگونه و با دوباره ساختن ویژگی جدید بتوانید برنامه را بهتر درک کرده و خطا را ناک اوت کنید.

برنامه‌نویسان تازه‌کار گاهی بین یکی از این موارد گیر می‌افتند و بقیه را فراموش می‌کنند.

به عنوان مثال، خواندن کد، ممکن است که به شما در حل مساله، اگر خطای نوشتاری در برنامه وجود داشته باشد، کمک کند ولی اگر کج‌فهمیِ مفهومی علت مشکل در برنامه باشد، خیر. اگر شما نمی‌فهمید که برنامه‌تان چه کاری انجام می‌دهد، با صد

بار خواندن کد، به ریشه‌ی خطا نخواهید رسید؛ به این خاطر که خطا نه در برنامه که در کله‌ی شماست.

انجام آزمایش می‌تواند به شما کمک کند، به خصوص اگر آزمایش‌های ساده و کوچکی را انجام دهید. ولی اگر آزمایش را بدون فکر کردن و خواندن کد انجام دهید، احتمالا در الگویی که من آن را «برنامه‌نویسیِ درهم و تصادفی» می‌نامم، گرفتار می‌شوید. شما تغییرات تصادفی و در هم انجام می‌دهید تا برنامه به کار بیفتد. البته فکر نکنم که لازم باشد که بگوییم این کارگاهی زمان زیادی را از شما تلف می‌کند.

شما بایستی که زمانی را برای فکر کردن بگذارید. اشکال‌زدایی شبیه به یک آزمایش علمی است. شما بایستی حداقل یک فرضیه در خصوص مشکل داشته باشید. اگر دو یا چند احتمال وجود دارد شروع به انجام آزمایش‌هایی که احتمالات را محدودتر و محدودتر می‌کند، کنید.

کمی استراحت کردن به شما کمک می‌کند که واضح‌تر فکر کنید. همینطور صحبت کردن. اگر مشکل را برای یک شخص دیگر توضیح دهید (حتی خودتان در آینه) گاهی قبل از اینکه پرسش‌تان تمام شود، به راه‌حل می‌رسید.

ولی گاهی بهترین تکنیک‌های اشکال‌زدایی هم – اگر تعداد خطاها بسیار زیاد باشد یا کد بزرگ و پیچیده باشد – با مشکل مواجه می‌شوند. گاهی بهترین گزینه عقب‌نشینی و ساده‌کردن برنامه تا جایی است که کار کند و شما آن را بفهمید.

برای برنامه‌نویسان تازه‌کار، گاهی عقب‌نشینی سخت است چرا که نمی‌خواهند کدهایی که نوشته‌اند را پاک کنند (حتی اگر مشکل‌دار باشد). اگر کپی کردن برنامه در یک فایل دیگر به شما کمک می‌کند که حس بهتری داشته باشید، آن را کپی کرده و سپس خطوط را حذف کنید. بعد از حل مشکل می‌توانید آن را دوباره به برنامه‌ی خودتان بچسبانید.

پیدا کردن یک باگِ سرسخت نیاز به خواندن، اجرا، غور کردن و گاهی عقب‌نشینی دارد. اگر در هر کدام از این موارد گیر افتادید و راه به جایی نبردید، دیگری را امتحان کنید.

## واژگان فصل

**قابل مقایسه / Comparable:**
یک مقدار می‌تواند برای بزرگتر، کوچکتر یا مساوی بودن با مقدار دیگر از همان نوع بررسی شود. انواعی که قابل مقایسه هستند می‌توانند در درون یک لیست قرار داده شده و مرتب شوند. ما به این موارد، قابل مقایسه یا Comparable می‌گوییم.

**ساختمان داده / Data Structure:**
مجموعه‌ای از مقادیر مرتبط که اغلب در قالب لیست‌ها، دیکشنری‌ها، تاپل‌ها و ... چیده می‌شوند.

**دی‌اس‌یو / DSU:**
کوتاه شده‌ی "decorate-sort-undecorate"، الگویی است که لیستی از تاپل‌ها را ساخته، مرتب می‌کند و قسمتی از نتیجه را استخراج می‌کند.

**هشیبل / Hashable:**
نوعی که یک تابع هش دارد. نوع‌های غیرقابل تغییر مثل اعداد صحیح، اعشاری و رشته‌ها هشیبل هستند؛ نوع‌های قابل تغییر مثل لیست‌ها و دیشکنری‌ها هشیبل نیستند.

**پراکنش / Scatter:**
عملیاتِ رفتار با یک توالی به عنوان لیستی از آرگیومنت‌ها.

**شکل / Shape:**
خلاصه‌ای از یک نوع، اندازه و ترکیب یک ساختمان داده.

**تک / Singleton:**
لیست (یا هر توالی دیگری) با یک المنت.

**تاپل / Tuple:**
توالیِ غیرقابل تغییر از المنت‌ها.

**گمارش تاپل / Tuple Assignment:**
گمارشی که با یک توالی در سمت راست و یک تاپل از متغیرها در سمت چپ انجام می‌شود. سمت راست ارزیابی شده و سپس المنت‌های آن به متغیرهای سمت چپ اختصاص داده می‌شوند.

## تمرین‌ها

**تمرین ۱:** برنامه‌ای که پیشتر نوشتیم را به این شکل بازنویسی کنید: خواندن و تحلیل کردن خطوطی که با From شروع می‌شوند و خارج کردن آدرس‌ها از آن خطوط؛ و شمردن تعداد پیام‌ها از هر شخص با استفاده از یک دیکشنری.

بعد از اینکه تمام داده‌ها خوانده شد، با ساختن یک لیست از تاپل (شمارش، ایمیل)، شخصی که بیشترین ایمیل‌ها را فرستاده پیدا کنید. سپس لیست را در حالت از بالا به پایین مرتب کنید و نهایت نام آن شخص و تعداد پیام‌هایش را چاپ کنید:

```
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:** برنامه‌ای بنویسید که پراکندگی ساعت‌هایی که پیام‌ها وارد شده‌اند را بررسی می‌کند. شما می‌توانید از خط From رشته‌ی مرتبط با ساعت را بیرون بکشید، سپس با استفاده از کاراکتر دو نقطه ساعت را استخراج کنید. بعد از آن هر ساعت را بشمارید و تعداد تکرار پیام در آن ساعتِ خاص را چاپ کنید.

نمونه‌ی خروجی برنامه:

```
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:** یک برنامه بنویسید که یک فایل را خوانده و حروف را بر حسب تکرار از بالا به پایین چاپ کند. برنامه‌ی شما بایستی تمام ورودی‌ها را به حروف کوچک تغییر داده و تنها حروف a تا z را بشمارد. برنامه نبایستی فاصله‌ها، اعداد، علائم نشانه‌گذاری یا هر چیزی غیر از a تا z را بشمارد. چندین متن از زبان‌های مختلف پیدا کنید و ببینید که تکرار حروف در آن‌ها چطور است. نتایج خود را با جدول زیر مقایسه کنید:

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

# فصل ۱۱

# عبارت‌های باقاعده

## عبارت‌های باقاعده (RegEx)

تا اینجای کار، فایل را می‌خواندیم، به دنبال الگوهای خاص می‌گشتیم و آن‌ها را استخراج می‌کردیم و در نهایت عملیات‌های خاصی روی آن‌ها انجام می‌دادیم. ما از متدهای رشته مثل split و find و با استفاده از لیست‌ها و قاچ زدن رشته‌ها، قسمت‌هایی از خطوط را خارج می‌کردیم.

کار جستجو و استخراج، به مانند آنچه توصیف کردیم، آنقدر رایج است که پایتون برای آن یک کتابخانه‌ی بسیار قدرتمند به نام Regular Expressions یا عبارت‌های باقاعده دارد. این کتابخانه به زیباترین شکل ممکن بسیاری از این وظایف را مدیریت و اجرایی می‌کند. ما عبارت‌های باقاعده را در ابتدا آموزش ندادیم چون در کنار قدرت زیادشان، کمی پیچیده هستند و برای کاربران، آشنایی با متنِ عبارت‌های باقاعده، کمی زمانبر خواهد بود.

عبارت‌های باقاعده تقریبا زبان کوچک خودشان را برای جستجو در فایل و تجزیه و تحلیل رشته‌ها دارند. در حقیقت کتاب‌هایی هستند که فقط در باب این عبارت‌ها نوشته شده‌اند. در این فصل ما تنها چشمه‌ای از دریای بزرگ آن را آموزش می‌دهیم. برای جزئیات بیشتر به لینک‌های زیر مراجعه کنید:

http://en.wikipedia.org/wiki/Regular_expression
https://docs.python.org/2/library/re.html

کتابخانه‌ی عبارت‌های باقاعده، re نام دارد که بایستی در ابتدا و قبل از استفاده از آن‌ها در برنامه‌ی شما درون‌ریزی یا ایمپورت شود. ساده‌ترین حالت استفاده از کتابخانه عبارت‌های باقاعده، تابع search است. برنامه‌ی زیر نشان می‌دهد که یک بهره‌وریِ ساده از تابع search چگونه است:

```
# 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: http://www.py4e.com/code3/re01.py
```

در مثال بالا ابتدا فایل را باز کردیم و در هر کدام از خطوط فایل حلقه زدیم. سپس با استفاده از عبارت باقاعده ()search تنها خطوطی را که شامل رشته‌ی :From می‌شوند را چاپ کردیم. این برنامه قدرت واقعی عبارت‌های باقاعده را به ما نشان نمی‌دهد چرا که به راحتی و با استفاده از ()line.find ما همین کاری را که re.search کرد، می‌توانستیم انجام دهیم.

قدرت عبارت‌های باقاعده زمانی مشخص می‌شود که ما کاراکترهای خاصی را به رشته‌ای که می‌خواهیم جستجو کنیم اضافه نماییم. این کار به ما اجازه می‌دهد که کنترل خطوط را با دقت بیشتری در دست بگیریم. اضافه کردن این کاراکترهای خاص به عبارت باقاعده، به ما کمک می‌کند تا کارهای پیچیده و حرفه‌ای را – برای تطابق و استخراج رشته‌ها با استفاده از یک قطعه کد بسیار کوچک – پیاده‌سازی کنیم.

به عنوان مثال، کاراکتر کَرِت (^) برای تطابق با ابتدای خط در عبارت‌های باقاعده مورد استفاده قرار می‌گیرد. با استفاده از این قابلیت ما می‌توانیم برنامه‌ی خود را جوری تغییر دهیم که تنها خطوطی که :From را در ابتدای خود دارند تطبیق داده شوند. مثال

زیر را ببینید که چگونه ما آرگیومنت عبارت باقاعده را تغییر دادیم تا به این مهم دست پیدا کنیم:

```
# 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: http://www.py4e.com/code3/re02.py
```

اکنون برنامه تنها خطوطی که با رشته‌ی From: آغاز می‌شوند را تطبیق خواهد داد. این مثالِ خیلی ساده از کارهایی است که می‌توانیم با استفاده از متد startswith() از کتابخانه‌ی رشته انجام دهیم. ولی این پیام را برای ما دارد که عبارت‌های باقاعده شامل کاراکترها با توانایی‌های خاصی هستند که به ما کنترل بیشتر برای انجام کارهای مختلف را می‌دهد.

## تطبیق کاراکتر در عبارت‌های باقاعده

تعداد بیشتری کاراکترهای خاص برای ساختن عبارت‌های باقاعده‌ی قدرتمندتر وجود دارند. پُراستفاده‌ترین کاراکتر خاص «نقطه» یا همان «توقف کامل» است که با هر کارکتری تطبیق پیدا می‌کند. بهتر است که با یک مثال منظور خودم را واضح‌تر بیان کنم.

در مثال زیر، عبارت باقاعده‌ی F..m: با هر رشته‌ای، از From: گرفته تا Fxxm: و F12m: یا F@!m:: جور می‌شود. از آنجایی که «نقطه» یا «.» در یک عبارت باقاعده مطابق با هر کارکتری خواهد بود، فرقی نمی‌کند که بین F و m چه کاراکترهایی باشد، فقط کافیست که تعداد دو کاراکتر بین یک رشته که با F شروع می‌شود و با m: خاتمه

می‌یابد وجود داشته باشد تا عبارت باقاعده‌ی ما آن رشته را، رشته‌ی «جذاب» به حساب بیاورد.

```
# 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: http://www.py4e.com/code3/re03.py
```

این روش، زمانی‌که با قابلیت تکرار به تعداد نامعلوم ترکیب شود، قدرت بسیار بیشتری پیدا می‌کند. شما با استفاده از کاراکترها * و + در عبارت باقاعده‌ی خود، به برنامه می‌گویید که فلان کاراکتر می‌تواند به تعداد نامعلومی تکرار شود، و این مورد اشکالی در تطبیق دادن آن‌ها ایجاد نمی‌کند.

شاید کمی گیج شده باشید. بگذارید کمی واضح‌تر مساله را بیان کنم. در مثال بالا ما دو نقطه بین F و m داشتیم که نشان می‌داد بایستی بین F و m فقط و فقط دو کاراکتر (البته فرق نمی‌کند با چه شکل و شمایلی، بلکه تعدادشان مهم است) وجود داشته باشد تا رشته، جور در بیاید. حالا اگر به جای دو عدد نقطه از یک نقطه و ستاره (F.*m:) استفاده کنیم، عبارت باقاعده‌ی ما اینگونه معنی می‌شود: هر رشته‌ای که با F شروع شود و با m: خاتمه یابد، با عبارت ما جور یا مطابق است. در حقیقت *. می‌گوید که مهم نیست بین F و m صفر یا صد میلیون کاراکتر باشد، تنها اگر رشته‌ای باشد که با F شروع شود و با m: خاتمه یابد همه چیز حل است.

ولی علامت به‌علاوه یا + چه کاری می‌کند؟ این علامت دقیقا کاری که ستاره انجام می‌دهد را می‌کند، با این تفاوت که در حالت ستاره حتی اگر کاراکتری هم بین F و m وجود نداشته باشد، عبارت ما جور در می‌آید (Fm: مثلا در حالت ستاره جور در می‌آید)

ولی وقتی از به‌علاوه استفاده می‌کنیم، بایستی حداقل یک کاراکتر بین F و :m وجود داشته باشد و Fm: جور در نخواهد آمد.

به عبارت دیگر این کاراکترهای ساده در رشته‌ی جستجو، صفر یا چند کاراکتر (در حالت ستاره) و یک یا چند کاراکتر (در حالت به‌علاوه) را مطابق به حساب می‌آورند.

حالا با استفاده از این قابلیت، ما جستجوی دقیق‌تری در بین خطوطِ فایل پیشین با به کارگیری وایلد کارت‌های[I] تکرار شونده انجام می‌دهیم:

```
# 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: http://www.py4e.com/code3/re04.py
```

رشته‌ی جستجو، یعنی "^From:.+@" با هر خطی که با From: شروع شود، سپس یک یا چند کاراکتر در جلوی آن باشد و به علامت ات‌ساین برسد، جور در می‌آید.

در مثال بالا به وایلدکارت "+." می‌توانید اینگونه نگاه کنید: مطابق با هر تعداد کاراکتری که بین دو نقطه و علامت @ باشد، به شرطی که حداقل یک کاراکتر بین دونقطه و @ وجود داشته باشد.

با اضافه کردن یک کاراکتر دیگر، می‌توان جلوی طمعِ کاراکترهای به‌علاوه و ستاره را گرفت. چرا طمع؟ زیرا این کاراکترها بسیار حریص هستند و تا جایی که بتوانند گسترش پیدا می‌کنند. به مثال زیر دقت کنید:

---

[I] علامتی که با هر کاراکتری سازگار است. م.

```
>>> bio = 'Hi, my name is Eman and my websites are ilola.ir
and ...'
>>> searchMyBio = re.findall('m.*a', bio)
>>> print(searchMyBio)
['my name is Eman and my websites are ilola.ir a']
>>>
```

اولین زیررشته‌ای که با عبارت باقاعده‌ی ما جور در می‌آید «my na» است که با m در کلمه‌ی my شروع شده، و به a در name می‌رسد. ولی عبارت باقاعده‌ی ما حریص است و تا جایی که بتواند خودش را گسترش می‌دهد، به همین خاطر به a در name قناعت نکرده و خودش را تا آخرین a موجود در رشته، یعنی and می‌کِشاند. همانگونه که نتیجه را مشاهده می‌کنید، تلاش ما برای جستجو در رشته یک زیررشته‌ی تقریبا دراز به خاطر طمع و حرص کاراکتر * است. شرایط برای کاراکتر + هم به همین صورت است.

در حقیقت عنان و افسار این کاراکترها در دستان شماست. برای اطلاعات بیشتر از جزئیات می‌توانید اسناد مرتبط با خاموش کردن رفتار حریصانه را مطالعه کنید.

https://docs.python.org/3/howto/regex.html#greedy-versus-non-greedy

## استخراج داده‌ها با استفاده از عبارت‌های باقاعده

اگر بخواهید که داده از یک رشته در پایتون استخراج کنید، می‌توانید از findall() استفاده کنید. این متد تمام زیررشته‌هایی که با یک عبارت باقاعده جور در می‌آیند را استخراج می‌کند. مثلا فرض کنید که می‌خواهید هر چیزی که شبیه به یک آدرس ایمیل است را از هر خط، بدون توجه به قالب آن، استخراج کنید. به عنوان یک نمونه، می‌خواهیم آدرس‌های ایمیل را از خطوط زیر بیرون بکشیم:

```
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
```

نمی‌خواهیم که برای هر نوع خط یک مدل کد بنویسیم که در آن از تکنیک‌های تکه و قاچ کردن بهره ببریم. با استفاده از متد findall() می‌توانیم خطوطی که شامل آدرس ایمیل می‌شود را پیدا کرده و یک یا چند آدرس را از درون آن‌ها بیرون بکشیم:

```
import re
s = 'A message from csev@umich.edu to cwen@iupui.edu about
meeting @2PM'
lst = re.findall('\S+@\S+', s)
print(lst)


# Code: http://www.py4e.com/code3/re05.py
```

در اینجا متد findall() رشته‌ای که دومین آرگیومنتش هست (در اینجا s) را جستجو کرده و لیستی از تمام رشته‌هایی که شبیه به آدرس ایمیل هستند را برمی‌گرداند، ما از توالی دو کاراکتر که با یک کاراکتر غیرفاصله جور در می‌آید استفاده کردیم (\S). به همین خاطر @2PM که در انتهای رشته است، جزو نتایج نیست – به فاصله‌ی قبل از @ نگاه کنید که با کاراکتر غیرفاصله (\S) جور در نمی‌آید. مراقب باشید که \S را با \s اشتباه نگیرید. اولی (با S بزرگ) با کاراکترهای غیرفاصله جور در می‌آید و دومی دقیقا برعکس آن، با کاراکترهای فاصله مثل فاصله‌ی خالی، تب، خط‌جدید و....

خروجی برنامه شبیه به این خواهد بود:

```
['csev@umich.edu', 'cwen@iupui.edu']
```

بیایید عبارت باقاعده در مثال بالا را ترجمه کنیم: شرایطی که عبارت باقاعده‌ی ما تعیین می‌کند بدین قرار است:

۱) یک زیر رشته؛

۲) که حداقل یک کاراکتر غیرفاصله داشته باشد(\S+)؛

۳) که این کاراکتر غیرفاصله به علامت اتساین برسد @ (فاصله‌ی خالی بین علامت اتساین و کاراکتر نباید وجود داشته باشد و کاراکتر غیرفاصله‌ی ما به اتساین چسبیده باشد)؛

۴) که به دنبال علامت اتساین حداقل یک کاراکتر غیرفاصله وجود داشته باشد (\S+) (حداقل یک کاراکتر غیرفاصله به اتساین چسبیده باشد).

عبارت باقاعده‌ی ما، دو زیررشته را در خط بالا پیدا خواهد کرد (csev@umich.edu و cwen@iupui.edu)، اما همانطور که قبلا گفتیم با @2PM جور در نمی‌آید چرا که هیچ کاراکتر غیرفاصله‌ای قبل از @ وجود ندارد. ما می‌توانیم از این عبارت باقاعده در یک برنامه برای خواندن تمام خطوط و چاپ هر چیزی که شبیه به آدرس ایمیل است 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('\S+@\S+', line)
    if len(x) > 0:
        print(x)

# Code: http://www.py4e.com/code3/re06.py
```

در این برنامه هر خط را می‌خوانیم سپس زیررشته‌هایی که با عبارت باقاعده‌ی ما جور در می‌آید را استخراج می‌کنیم. از آن‌جایی که findall() یک لیست را برمی‌گرداند، کافی‌ست که تعداد المنت‌های موجود در لیست را بشماریم. به این صورت اگر لیست ما شامل بیش از صفر المنت شود، می‌توانیم عناصر آن را چاپ کنیم. این زیررشته‌ها دست کم شبیه به آدرس ایمیل هستند.

اگر برنامه را روی فایل mbox.txt اجرا کنیم، خروجی زیر را مشاهده خواهیم کرد:

```
['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)']
['source@collab.sakaiproject.org;']
```

برخی از آدرس‌های ایمیل ما صحیح نیستند و شامل کاراکترهایی مثل > در ابتدا یا انتهای خود می‌شوند. اما برای ما تنها قسمتی که با عدد یا حروف شروع می‌شود و خاتمه می‌یابد مهم است. چطور اضافات را حذف کنیم؟

برای انجام این کار ما یک ویژگی دیگری از عبارت‌های باقاعده را مورد استفاده قرار می‌دهیم. قلاب‌ها برای نشان دادن دسته‌ای از کاراکترهای مورد قبول که ما می‌خواهیم با رشته‌ی ما جور در بیاید مورد استفاده قرار می‌گیرد. عبارت \S به ما کمک می‌کرد که زیررشته با کاراکترغیرفاصله جور شود. حالا می‌خواهیم درخواستمان از برنامه برای پیدا کردنِ دقیق‌ترِ آدرس ایمیل را کمی واضح‌تر بیان کنیم.

در اینجا ما عبارت باقاعده‌ی جدید خودمان را داریم:

```
[a-zA-Z0-9]\S*@\S*[a-zA-Z]
```

کمی پیچیده شد. اکنون می‌بینید که چرا می‌گوییم عبارت‌های باقاعده زبان مخصوص خودشان را دارند. ترجمه این عبارت باقاعده این است که: ما به دنبال یک زیررشته می‌گردیم که

۱) با یک حرف کوچک یا حرف بزرگ یا یک عدد شروع شود ("[a-zA-Z0-9]")؛ و

۲) در ادامه تعدادِ صفر یا بیشتر کاراکتر غیرفاصله داشته باشد ("\S*")؛ و

۳) سپس علامت اتساین (@)؛ و

۴) بعد از آن تعداد صفر یا بیشتر کاراکتر غیرفاصله ("*\S")؛ و

۵) در نهایت به یک حرف بزرگ یا کوچک ختم شود ("[a-zA-Z]").

توجه داشته باشید که برای عبارت باقاعده‌ی "*\S" از ستاره به جای به‌علاوه استفاده کردیم چرا که ممکن بود قبل یا بعد از علامت اتساین تنها یک کاراکتر داشته باشیم که توسط عبارت‌های باقاعده‌ی "[a-zA-Z0-9]" و "[a-zA-Z]" تامین می‌شدند. در حقیقت حداقل یک کاراکتر غیرفاصله قبل یا بعد از @ خواهیم داشت.

اگر از این عبارت باقاعده در برنامه‌ی خود استفاده کنیم، داده‌های ما بسیار تر و تمیزتر خواهد 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('[a-zA-Z0-9]\S+@\S+[a-zA-Z]', line)
    if len(x) > 0:
        print(x)


# Code: http://www.py4e.com/code3/re07.py
```

خروجی:

```
...
['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 کنید که در خط‌های "source@collab.sakaiproject.org" عبارت باقاعده‌ی ما، جستجو را دو نویسه‌ی قبل از انتهای رشته پایان می‌دهد (";>"). این اتفاق به این خاطر افتاد که ما در عبارت باقاعده مشخص کردیم که یکی از حروف الفبا نشان‌دهنده‌ی پایان زیررشته است ("[a-zA-Z]"). به عبارتی ما درخواست داده‌ایم که هر رشته‌ای که عبارت باقاعده‌ی ما پیدا می‌کند بایستی با یک حرف خاتمه یابد. به همین خاطر زمانی‌که عبارت باقاعده به ">" در زیررشته‌ی ";sakaiproject.org>" می‌رسد، تطابق زیررشته را متوقف می‌کند. به عبارت ساده‌تر، "g" در رشته‌ی بالا، آخرین کاراکتری است که با عبارت باقاعده جور در می‌آید و اگر عبارت باقاعده، آن را به هوای یک کاراکتر غیرفاصله ("\S") پشت سر بگذارد، در نهایت بدون اینکه به یک حرف بزرگ یا کوچک برخورد کند، به کاراکتر فاصله می‌رسد و جستجو با چیزی که ما از او می‌خواهیم جور در نمی‌آید. به همین خاطر قبل از ">" و با رسیدن به "g" متوقف می‌شود.

همچنین توجه داشته باشید که خروجی برنامه یک لیست پایتونی است (در اینجا چندین لیست) که هر لیست تنها یک رشته را، به عنوان تنها المنت، در خودش دارد.

## ترکیب جستجو و استخراج

فرض کنید که ما می‌خواهیم عددی را در خطی که با رشته‌ی "X-" شروع می‌شود (شبیه به خطوط پایین) پیدا کنیم:

```
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000
```

قرار نیست که هر عدد اعشاری از هر خطی که دیدیم را استخراج و تحلیل کنیم، بلکه تنها عددهایی به کار ما می‌آید که ابتدای خطوطشان قاعده‌ی بالا را داشته باشند و با "X-" شروع شوند.

ما عبارت باقاعده‌ی زیر را برای انتخاب آن خطوط نوشتیم:

```
^X-.*: [0-9.]+
```

ترجمه‌ای این خط می‌شود: ما خطوطی را می‌خواهیم که با "X-" شروع شود، و با تعداد صفر یا بیشتر کاراکتر ادامه پیدا کند ("*.") سپس به یک دو نقطه ختم شود (":") و بعد از آن یک فاصله‌ی خالی وجود داشته باشد (" ") و بعد از فاصله‌ی خالی به دنبال یک یا بیشتر کاراکتر که یا یک عدد باشد (0-9) یا یک نقطه "+[.9-0]" بگردد. توجه کنید که داخل قلاب، نقطه برابر با نقطه است و وایلدکارتی برای هر حرف (مثل نقطه‌ی قبلی در همین عبارت باقاعده) نیست.

این یک عبارت باقاعده‌ی بسیار فشرده است که تنها خطوطی که ما می‌خواهیم را از دل متن بیرون می‌کشد:

```
# 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('^X\S*: [0-9.]+', line):
        print(line)


# Code: http://www.py4e.com/code3/re10.py
```

زمانی‌که برنامه را اجرا می‌کنیم، می‌بینیم که داده‌ها به زیبایی فیلتر شده‌اند و تنها خطوطی که مد نظر ما بود، چاپ می‌شود:

```
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000
X-DSPAM-Confidence: 0.6178
X-DSPAM-Probability: 0.0000
```

اما حالا بایستی که مساله‌ی جدا کردن عددها را از این خطوط حل کنیم. شاید استفاده از split ساده باشد، ولی می‌خواهیم از عبارت‌های باقاعده و یک ویژگی جدید آن‌ها بهره ببریم؛ یک ویژگی با قابلیت تحلیل خط در حین جستجوی آن.

پرانتزها کاراکترهای ویژه‌ی دیگری در عبارت‌های باقاعده به حساب می‌آیند. زمانی‌که شما پرانتز را به یک عبارت باقاعده اضافه می‌کنید، آن عبارت به دنبال زیررشته‌هایی که با خودش جور در می‌آید می‌گردد، ولی تنها قسمتی از زیررشته که در پرانتز آمده را برمی‌گرداند. بهتر است با یک مثال عملکرد پرانتز را خودتان ببینید. ما تغییرِ زیر را در برنامه اعمال 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('^X\S*: ([0-9.]+)', line)
    if len(x) > 0:
        print(x)

# Code: http://www.py4e.com/code3/re11.py
```

عبارت باقاعده‌ی "+[0-9.]" را درون پرانتز قرار دادیم. به جای فراخوانی search() برای جدا کردن قسمت مورد نظرمان از زیر رشته، آن قسمت جذاب (در مثال بالا عدد

اعشاری) را در درون پرانتز قرار می‌دهیم. به این صورت ()findall ابتدا به دنبال زیررشته‌هایی که با کل عبارت باقاعده جور در می‌آید می‌گردد "'^X\S*: ([0-9.]+)'" ولی تنها قسمتی که در درون پرانتز قرار دارد یعنی "[0-9.]+" را برمی‌گرداند.

حالا خروجی برنامه با تغییری که اعمال کردیم، شبیه به خطوط زیر خواهد بود:

```
['0.8475']
['0.0000']
['0.6178']
['0.0000']
['0.6961']
['0.0000']
..
```

عددها هنوز هم در لیست‌های تک المنتی قرار دارند و لازم است که از رشته به عدد اعشاری تبدیل شوند، ولی با قدرت عبارت‌های باقاعده توانستیم که اطلاعاتی که می‌خواهیم را از دل داده‌ها جدا و استخراج کنیم.

به عنوان یک مثال دیگر از این تکنیک، اگر شما نگاهی به فایل بیندازید، خطوطی با قالب زیر را خواهید دید:

```
Details:
http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
```

اگر بخواهیم تمام عددهای بازنگری (در مثال بالا 39772) را استخراج کنیم، می‌توانیم از تکنیک بالا بهره ببریم. برنامه‌ی زیر به این منظور نوشته شده است:

```
# Search for lines that start with 'Details: rev='
# followed by numbers and '.'
# 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('^Details:.*rev=([0-9.]+)', line)
    if len(x) > 0:
        print(x)


# Code: http://www.py4e.com/code3/re12.py
```

ترجمه این عبارت باقاعده می‌شود: به دنبال خطوطی که با "Details:" شروع می‌شوند و سپس تعداد صفر یا بیشتر کاراکتر دارند و بعد از آن به "rev=" می‌رسند و در نهایت به یک یا چند عدد برخورد می‌کنند، بگرد. ما می‌خواهیم که عبارت باقاعده‌ی ما به دنبال رشته‌ای که تمام این شرایط را دارد بگردد ولی تنها قسمتِ عددی آن را استخراج کرده و برگرداند، به همین خاطر "+[0-9.]" را در داخل پرانتز قرار می‌دهیم.

زمانی‌که برنامه را اجرا می‌کنیم با خروجی شبیه به خطوط زیر روبرو خواهیم شد:

```
['39772']
['39771']
['39770']
['39769']
...
```

عبارت باقاعده‌ی "+[0-9.]" «حریص» است و برایش فرقی نمی‌کند که رشته‌ی عددی که به آن می‌رسد چقدر بزرگ باشد. در اصل آنقدر حریص است که تا جایی که بتواند این رشته را کش می‌دهد. دلیل اینکه ما در مثال بالا هر پنج رقم را به عنوان خروجی دریافت می‌کنیم همین رفتار حریصانه است. در حقیقت کتابخانه‌ی عبارت باقاعده آن را در هر دو طرف کش می‌دهد، به صورتی که یا به یک کاراکتر غیرعددی برخورد کند یا اینکه به آغاز یا پایان یک خط برسد.

حالا می‌توانیم از عبارت‌های باقاعده برای انجام دوباره‌ی یک تمرین که پیش‌تر به آن پرداخته بودیم، استفاده کنیم؛ تمرینی که در آن به دنبال روزِ هر کدام از پیام‌ها بودیم. ما به دنبال خطوطی به این شکل می‌گشتیم:

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

و می‌خواستیم که ساعتِ روز هر کدام از خطوط را استخراج کنیم. پیشتر با فراخوانی split این کار را انجام دادیم. ابتدا خط را به حروف شکستیم و سپس پنجمین کلمه را خارج کردیم و دوباره با شکستن آن کلمه با حائل دو نقطه، دو کاراکتری را که می‌خواستیم خارج کردیم.

درست است که این روش پاسخ ما را به درستی خارج می‌کرد ولی کد ما، خطوط را خیلی باقاعده در نظر گرفته بود. به عبارتی، کافی بود که در این خطوط یکی دو کلمه کم یا زیاد شود تا برنامه چیز اشتباهی را برگرداند. اگر قرار بود کد را از این خطاها مصون کنیم، و مطمئن شویم که مشکلی پیش نمی‌آید بایستی ۱۰، ۱۵ خط دیگر به آن اضافه می‌کردیم؛ به این صورت از خوانایی کد به شدت کاسته می‌شد.

حالا با استفاده از عبارت‌های باقاعده می‌توانیم همان کار را انجام دهیم، با این تفاوت که کد ما منعطف‌تر خواهد بود و نیازی هم به اضافه کردن کدِ دیگر برای اطمینان از عملکرد برنامه نخواهیم داشت:

```
^From .* [0-9][0-9]:
```

ترجمه این عبارت باقاعده می‌شود:

۱) اول به دنبال خطوطی که با " From" شروع می‌شوند بگرد (به فاصله‌ی بعد از From دقت کنید)؛

۲) به دنبال " From" تعداد صفر یا بیشتر کاراکتر باید وجود داشته باشد("*.")؛

۳) همچنین بایستی یک فاصله (Space) بعد از آن کاراکترها باشد (" ") و

۴) در نهایت به دو عددی برسد ("[0-9][0-9]") که به کاراکتر دو نقطه چسبیده‌اند (":").

برای اینکه ()findall تنها ساعت را از این خطوط بیرون بکشد، از پرانتز استفاده کردیم:

```
^From .* ([0-9][0-9]):
```

نتیجه می‌شود برنامه‌ی زیر:

```
# 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 it is greater than zero
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: http://www.py4e.com/code3/re13.py
```

زمانی‌که برنامه اجرا می‌شود، خروجی زیر را ایجاد خواهد کرد:

```
['09']
['18']
['16']
['15']
…
```

## کاراکتر اسکِیپْ یا فرار

همانگونه که دیدید ما از کاراکترهای ویژه‌ای برای نمایش انتها و ابتدا و وایلد کارت‌ها در عبارت‌های باقاعده استفاده کردیم. حالا شاید بخواهید که به دنبال همین کاراکترهای ویژه بگردید. مثلا به دنبال کاراکتر کرت (^) بدون اینکه منظورتان ابتدای خط باشد. به عبارتی در پاره‌ای از مواقع برای اینکه به برنامه بگویید که ^ یا $ یک کاراکتر نرمال است و نشان دهنده‌ی ابتدای خط، یا چیز دیگری نیست بایستی نشانه‌ی ویژه‌ای به برنامه بدهیم.

ما می‌توانیم با یک پیشوند بک‌اسلش به سادگی به برنامه نشان دهیم که این کاراکتر، کاراکتر نرمال است. به عنوان مثال می‌توانیم مقدار پول را با عبارت باقاعده‌ی زیر از دل متن بیرون بکشیم:

```
import re
x = 'We just received $10.00 for cookies.'
y = re.findall('\$[0-9.]+',x)
```

از آنجایی که ما قبل از علامت دلار یک بک‌اسلش گذاشتیم، عبارت باقاعده به دنبال علامت دلار می‌گردد و کاری به پایان خط ندارد. ادامه‌ی عبارت باقاعده‌ی بالا به دنبال یک یا چند عدد، یا کاراکتر نقطه می‌گردد.

**نکته:** کاراکترها در داخل قلاب، دیگر خاص و ویژه نیستید. به همین خاطر در مواجه با "[0-9.]" برنامه به دنبال عدد و یا نقطه می‌گردد. خارج از قلاب‌ها، یک نقطه، یک وایلد کارت برای هر کاراکتری هست و با هر کاراکتری جور می‌شود، ولی داخل قلاب، نقطه فقط و فقط نقطه است.

## جمع‌بندی

این تنها یک چشمه از دریای عبارت‌های باقاعده بود. تا اینجا کمی در خصوص زبان عبارت‌های باقاعده آموختیم. رشته‌های جستجو، با کاراکترهای خاص در آن‌ها، که خواسته‌های شما را به زبان سیستم عبارت باقاعده بیان می‌کند تا عبارت‌هایی که «جور»

در می‌آید را از دل آن‌ها بیرون بکشد. در اینجا شما را با چند کاراکتر ویژه و توالی‌های کاراکتری دیگر آشنا می‌کنیم:

```
^
```

^ با ابتدای خط جور در می‌آید.

```
$
```

$ با انتهای خط جور در می‌آید.

```
.
```

. با هر کاراکتری جور در می‌آید (وایلد کارت).

```
\s
```

\s با فاصله‌ی سفید یا خالی جور در می‌آید (شامل اسپیس، تب و...).

```
\S
```

\S با کاراکترهایی به جز فاصله‌ی سفید جور در می‌آید (برخلاف \s).

```
*
```

* به کاراکتر پیشین خود می‌چسبد و نشان می‌دهد که با تکرار صفر یا بیشتر از نمونه‌ی کاراکتر پیشین جور در می‌آید. مثلا اگر به \s بچسبد و به‌صورت *\s نوشته شود، با صفر یا چند فاصله‌ی خالی جور در می‌آید.

```
*?
```

?* به کاراکتر پیشین خود می‌چسبد و نشان می‌دهد که با تکرار صفر یا بیشتر از نمونه‌ی کاراکتر پیشین در حالت غیرحریصانه جور در می‌آید. غیرحریصانه به این معنی که به اولین زیررشته‌ای که جور در بیاید، قناعت می‌کند. توضیحات بیشتر یادداشت پایانی این فصل آمده است.

```
+
```

+ به کاراکتر پیشین خود می‌چسبد و نشان می‌دهد که با تکرار یک یا بیشتر از نمونه‌ی کاراکتر پیشین جور در می‌آید. مثلا اگر به \s بچسبد و به‌صورت +\s نوشته شود، با یک یا چند فاصله‌ی خالی جور در می‌آید.

```
+?
```

+? به کاراکتر پیشین خود می‌چسبد و نشان می‌دهد که با تکرار یک یا بیشتر از نمونه‌ی کاراکتر پیشین در حالت غیرحریصانه جور در می‌آید. غیرحریصانه به این معنی که به اولین زیررشته‌ای که جور در بیاید، قناعت می‌کند. توضیحات بیشتر یادداشت پایانی این فصل آمده است.

```
[aeiou]
```

[aeiou] با یک تک‌کاراکتر، اگر آن کاراکتر جزئی از دسته‌ای که داخل قلاب است باشد، جور در می‌آید. در این مثال با هر کاراکتری که a یا e یا i یا o یا u باشد. توجه کنید که تنها با یکی از آن کاراکترها جور در می‌آید و اگر قرار باشد با توالی بیش از یک کاراکتر – که شامل کاراکترهای داخل قلاب می‌شود – جور در بیاید بایستی که از ستاره یا به‌علاوه، درست چسبیده به این عبارتِ باقاعده استفاده کنید، مثلا: +[aeiou] با یک یا چند کاراکتر که شامل کاراکترهای داخلِ قلاب شود جور در می‌آید.

```
[a-z0-9]
```

[a-z0-9] شما می‌توانید از علامت منها برای نشان دادن یک بازه‌ای از کاراکترها استفاده کنید. به مانند مثال قبل با یک تک‌کاراکتر، اگر آن کاراکتر جزئی از دسته‌ای که داخل قلاب است باشد، جور در می‌آید. در این مثال با هر تک‌کاراکتری که یکی از حروف کوچک یا یک عدد باشد جور در می‌آید. توجه کنید که تنها با یکی از آن کاراکترها جور در می‌آید و اگر قرار باشد با توالی بیش از یک کاراکتر – که شامل کاراکترهای داخل قلاب می‌شود – جور در بیاید بایستی که از ستاره یا به‌علاوه درست چسبیده به این عبارت باقاعده استفاده کنید.

```
[^A-Za-z]
```

[^A-Za-z] زمانی‌که اولین کاراکتر داخل قلاب برای تعیین دسته‌ای از کاراکترها یک علامت کرت باشد، منطق داخل قلاب برعکس می‌شود. معنیِ این مثال از عبارت باقاعده می‌شود: برعکسِ حروف بزرگ و کوچک؛ یا به عبارت ساده‌تر: هر کاراکتری که حروف کوچک یا بزرگ نباشد.

```
()
```

( ) وقتی یک جفت پرانتز به عبارت باقاعده‌ای اضافه شود، آن عبارت باقاعده نادیده گرفته می‌شود، ولی با استفاده از ()findall به شما اجازه می‌دهد که یک زیرمجموعه از آن را – به جای کلِ مجموعه – استخراج کنید.

```
\b
```

\b با رشته‌ی خالی اگر در ابتدا یا انتهای یک کلمه باشد جور در می‌آید.

```
\B
```

\B با رشته‌ی خالی اگر در ابتدا یا انتهای یک کلمه نباشد جور در می‌آید.

```
\d
```

d\ با هر عدد دهدهی جور در می‌آید؛ با عبارت باقاعده‌ی [۹-۰] برابر است.

```
\D
```

D\ با هر کاراکتر غیرعددی جور در می‌آید؛ با عبارت باقاعده‌ی [9-0^] برابر است.

## شتل برای کاربران لینوکس/یونیکس

پشتیبانی از جستجوی فایل با استفاده از عبارت‌های باقاعده در سیستم‌های یونیکسی از دهه ۱۹۶۰ قرار داده شده و تقریبا در تمام زبان‌های برنامه‌نویسی به طریقی در دسترس است.

در حقیقت یک برنامه‌ی تحت خط فرمان در یونیکس به اسم grep (تلفظ کنید: گرِپ) (Generalized Regular Expression Parser) تقریبا کاری را که search() انجام می‌دهد، می‌کند. در نتیجه اگر دستگاه مکینتاش یا لینوکسی دارید، می‌توانید دستورات زیر را در خط فرمان خود آزمایش 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
```

دستور بالا به grep می‌گوید: خطوطی که با From: شروع می‌شود را در فایل mbox-short.txt نشان بده. اگر کمی با grep دست و پنجه نرم کنید و کمی اسناد مرتبط با آن را بخوانید، متوجه تفاوت‌های زیرپوستی، بین عبارت‌های باقاعده‌ی پایتونی و عبارت‌های باقاعده‌ای که توسط grep پشتیبانی می‌شود، خواهید شد. به عنوان مثال، grep کاراکتر غیرخالی "\S" را پشتیبانی نمی‌کند و لازم است که دسته‌ای از نمادهای پیچیده‌تر را برای مشخص کردن آن به کار ببرید؛ مثلا در اینجا می‌توانید از "[ ^]" استفاده کنید که ترجمه‌اش می‌شود: هر کاراکتری که فاصله‌ی اسپیس نباشد.

## اشکال‌زدایی

پایتون چند سندِ ساده و اصلی در خود دارد که می‌تواند به شما کمک کند. شاید حافظه‌ی شما به یک هولِ کوچک برای روشن شدن نیاز داشته باشد؛ اینجا دقیقا همان‌جایی است که این اسناد پایتونی به کمکتان می‌آید؛ مثلا زمانی‌که در خصوص اسم دقیق یک متد شک دارید. این اسناد در حالت تعاملی از طریق مفسر پایتون در دسترس شما خواهد بود.

برای به کار بردن «کمک» در حالت تعاملی از help() استفاده کنید:

```
>>> help()


help> modules
```

اگر می‌دانید که قرار است از چه ماژولی استفاده کنید، می‌توانید dir() را احضار کنید تا متدهایی را که در ماژول وجود دارد ببینید:

```
>>> import re
>>> dir(re)
[.. 'compile', 'copy_reg', 'error', 'escape', 'findall',
'finditer', 'match', 'purge', 'search', 'split',
'sre_compile',
'sre_parse', 'sub', 'subn', 'sys', 'template']
```

همچنین می‌توانید مقدار کوچکی از اسناد، در خصوص یک متد خاص را با استفاده از فرمان help یا dir دریافت کنید:

```
>>> 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.


>>> dir(re.search)
['__annotations__', '__call__', '__class__', '__closure__',
'__code__', '__defaults__', '__delattr__', '__dict__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__get__', '__getattribute__', '__globals__', '__gt__',
'__hash__', '__init__', '__kwdefaults__', '__le__',
'__lt__', '__module__', '__name__', '__ne__', '__new__',
'__qualname__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__']
```

اسناد درونی پایتون اطلاعات زیاد و با جزئیات را در اختیار شما قرار نمی‌دهند، ولی در زمان‌هایی که عجله دارید یا دسترسی به مرورگر وب یا موتور جستجو ندارید، می‌تواند کمک بزرگی باشند.

## واژگان فصل

**کدهای شکننده / Brittle Code:**

کدهای شکننده، کدهایی هستند که با قالب خاصی از داده‌های ورودی کار می‌کنند، اما مستعد شکستن و خراب شدن هستند. اگر کمی تحریف از قالبِ درست در ورودی پدید آید، کد از کار خواهد افتاد.

**جور در آمدن حریصانه / Greedy Matching:**
کاراکترهای + و * در یک عبارت باقاعده تا جای ممکن کشیده شده و یک رشته را شامل می‌شوند. این کاراکترها حریص هستند و تا جایی که بتوانند کش می‌آیند.

**گرپ / grep:**
یک دستوری که در اغلب سیستم‌های یونیکسی در دسترس است. این دستور برای جستجو در فایل‌های متنی و پیدا کردن خطوطی که با عبارت‌های باقاعده (که به دستور داده می‌شود) جور در می‌آید، مورد استفاده قرار می‌گیرد. نام grep از حروف ابتدایی عبارت Generalized Regular Expression Parser یا «تجزیه‌کننده‌ی کلی عبارت باقاعده» گرفته شده است.

**عبارت باقاعده / Regular Expression:**
زبانی برای بیان رشته‌های جستجوی پیچیده‌تر از حالت عادی است. یک عبارت باقاعده احتمالا شامل کاراکترهای ویژه‌ای می‌شود که مثلا به برنامه می‌گوید تنها ابتدا یا انتهای خطوط را جستجو کن. از این ویژگی‌ها به وفور در عبارت‌های باقاعده یافت می‌شود.

**وایلد کارت / Wild Card:**
یک کاراکتر ویژه که با هر کاراکتری جور در می‌آید. در عبارت‌های باقاعده یک وایلد کارت با نقطه نمایش داده می‌شود.

## تمرین‌ها

**تمرین ۱:** یک برنامه‌ی ساده بنویسید که کار فرمان grep در یونیکس را شبیه‌سازی کند؛ از کاربر در خصوص وارد کردن یک عبارت باقاعده سوال کند و تعداد خطوطی که با آن جور در می‌آید را چاپ نماید:

```
$ 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 4218 lines that matched java$
```

**تمرین ۲:** برنامه‌ای بنویسید که به دنبال خطوطی با فُرم زیر باشد:

```
`New Revision: 39772`
```

سپس عدد را از هر خط با استفاده از یک عبارت باقاعده و متد findall() استخراج کند. در نهایت میانگین این اعداد را محاسبه و چاپ نماید.

```
Enter file:mbox.txt
38549.7949721
```

```
Enter file:mbox-short.txt
39756.9259259
```

## یادداشت پایانی فصل

### رفتار غیرحریصانه

پیش‌تر مثالی در خصوص رفتار حریصانه دیدیم، اکنون همان مثال را برای نمایش چگونگی عمل‌کرد رفتار غیرحریصانه تکرار می‌کنیم:

```
>>> bio = 'Hi, my name is Eman and my websites are ilola.ir
and ...'
>>> searchMyBio = re.findall('m.+?a', bio)
>>> print(searchMyBio)
['my na', 'me is Ema', 'my websites a']
>>>
```

با اضافه کردن ? به انتهای + یا * به آن‌ها می‌فهمانیم که بایستی قانع بوده و به اولین زیرشته‌ای که با عبارت باقاعده جور در می‌آید بسنده کنند.

در اینجا خروجی برنامه یک لیست با سه المنت است. هر المنت نشان‌دهنده‌ی جور در آمدن عبارت باقاعده با رشته‌ی اصلی است. در رفتار غیرحریصانه، اولین زیررشته که با عبارت باقاعده جور در بیاید برگرداننده می‌شود و سپس برنامه به کاوشِ ادامه‌ی رشته می‌پردازد.

# فصل ۱۲

# برنامه‌های تحت شبکه

با وجود اینکه بسیاری از مثال‌های این کتاب روی خواندن و گشتن در میان داده‌های فایل‌ها متمرکز بوده است، زمانی‌که وارد قلمرو شبکه‌ی جهانی، یعنی اینترنت می‌شویم، سر و کله‌ی منابع بسیارِ دیگری نیز پیدا می‌شود.

در این فصل ما خودمان را یک مرورگر وب فرض می‌کنیم که صفحات وب را با استفاده از پروتکل انتقال اَبَرمتن (HTTP) می‌گیرد. سپس داده‌های صفحات وب را خوانده و تجزیه و تحلیل می‌کنیم.

## پروتکل انتقال اَبَر متن - HTTP

پروتکل شبکه که به وب قدرت داده، در حقیقت بسیار ساده است و توسط پایتون به صورت درونی پشتیبانی می‌شود. این پروتکل sockets/سوکت‌ها خوانده می‌شود. ساخت ارتباطات شبکه و دریافت داده‌ها از طریق این سکوت‌ها در یک برنامه‌ی پایتونی بسیار راحت و آسان است.

یک سوکت، شبیه به یک فایل است، به غیر از اینکه یک سوکت، یک ارتباط دو سویه بین دو برنامه را برقرار می‌کند. شما از طریق یک سوکت می‌توانید هم بخوانید و هم بنویسید. اگر شما چیزی بر سوکت بنویسید، این اطلاعات به برنامه‌ای که در طرف دیگر سوکت قرار دارد می‌رسد. اگر شما از سوکت بخوانید، به شما داده‌ای داده می‌شود که برنامه‌ی دیگر فرستاده است.

اما اگر شما سعی در خواندن یک سوکت، در زمانی‌که برنامه در سمت دیگر هنوز داده‌ای نفرستاده، بکنید چه اتفاقی می‌افتد؟ فقط نشسته‌اید و منتظر مانده‌اید. اگر برنامه‌های دو سر این ارتباط، بدون فرستادن داده‌ای، صرفا منتظر دریافت بمانند، بایستی زمان بسیار طولانی را منتظر بمانند؛ شاید علفی زیر پایشان سبز شد!

یک بخش مهم از برنامه‌ای که ارتباط از طریق اینترنت دارد و به شکلی با شبکه سر و کار دارد، داشتن پروتکل است. یک پروتکل، دسته‌ای از قوانین دقیق است که مشخص می‌کند چه کسی بایستی اول برود، و چه کاری باید انجام دهد، و ببیند که پاسخ چه خواهد بود، و چه کسی پیام بعدی را می‌فرستد و غیره. به عبارتی دو برنامه در طرف‌های این سوکت با یکدیگر می‌رقصند و البته باید مراقب باشند که پاهای همدیگر را لگد نکنند.

اسناد زیادی که پروتکل‌های شبکه را تشریح می‌کند وجود دارد. پروتکل انتقال ابرمتن، در سند زیر تشریح شده است:

http://www.w3.org/Protocols/rfc2616/rfc2616.txt

متن بالا در اصل یک سند بلند و پیچیده‌ی ۱۷۶ صفحه‌ای با جزئیات بسیار فراوان است. اگر به نظرتان جذاب آمد، خواندنش خالی از لطف نخواهد بود. اما اگر شما نگاهی به صفحه‌ی ۳۶ بیندازید، متن درخواست GET را پیدا خواهید کرد. برای درخواست یک سند از وب‌سرور، ما یک ارتباط با www.pr4e.org روی پورت ۸۰ برقرار می‌کنیم و سپس یک خط با فرم زیر ارسال می‌کنیم:

```
GET http://data.pr4e.org/romeo.txt HTTP/1.0
```

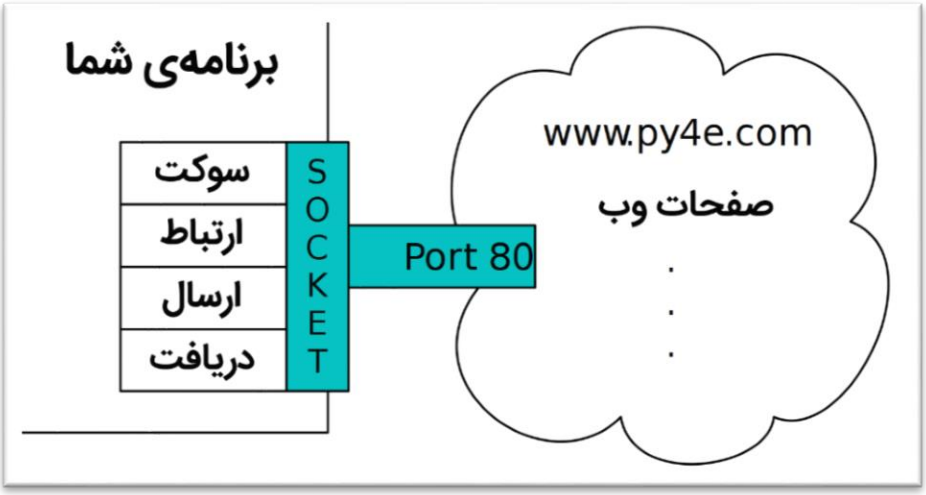
دومین پارامتر، یک صفحه وب است که ما درخواست می‌کنیم. سپس ما یک خط خالی می‌فرستیم. وب‌سرور پاسخی ارسال می‌کند که حاوی اطلاعاتی از هدرِ در خصوص سند و یک خط خالی دنباله‌ی محتویات سند است.

## ساده‌ترین مرورگر وب

شاید راحت‌ترین راه برای نشان دادن چگونگی عملکرد پروتکل HTTP نوشتن برنامه‌ی بسیار ساده‌ی پایتونی است که ارتباطی با یک وب‌سرور برقرار کند. این ارتباط بر اساس قوانین پروتکل HTTP بوده و از آن‌ها تبعیت می‌کند. با توجه به قوانین، درخواست یک سند فرستاده شده و هر چیزی را که سرور پاسخ می‌دهد، برای شما نمایش می‌دهد.

```
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())
mysock.close()


# Code: http://www.py4e.com/code3/socket1.py
```

در ابتدا برنامه یک ارتباط روی پورت ۸۰ با سرور www.py4e.com برقرار می‌کند. از آنجایی که برنامه‌ی ما، نقش مرورگر وب را دارد، پروتکل HTTP می‌گوید که بایستی یک دستور GET که یک خط خالی در ادامه‌اش می‌آید را ارسال کنیم.




بعد از اینکه آن خط خالی را فرستادیم، حلقه‌ای قرار دارد که داده‌ها را در قطعه‌های ۵۱۲ کاراکتری از سوکت دریافت و سپس آن اطلاعات را چاپ می‌کند. این حلقه تا زمانی‌که داده‌ای برای خواندن وجود داشته باشد تکرار می‌شود (تا زمانی‌که recv() یک رشته‌ی خالی برگرداند ).

برنامه، خروجی زیر را ایجاد خواهد کرد:

```
HTTP/1.1 200 OK
Date: Sun, 14 Mar 2010 23:52:41 GMT
Server: Apache
Last-Modified: Tue, 29 Dec 2009 01:31:22 GMT
ETag: "143c1b33-a7-4b395bea"
Accept-Ranges: bytes
Content-Length: 167
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 هِدِر شروع می‌شود. هدر را وب‌سرور برای تشریح سند ارسال می‌کند. به عنوان مثال Content-Type نشان می‌دهد که سند یک سند ساده و بدون فرمت است (text/plain).

بعد از اینکه سرور، هدر را برای ما ارسال کرد، یک خط خالی برای نشان دادن انتهای هدرها اضافه می‌کند و سپس داده‌های اصلی فایل romeo.txt را می‌فرستد.

در این مثال دیدیم که چطور می‌توان یک ارتباط شبکه‌ی سطح پایین با استفاده از سوکت‌ها ساخت. از سوکت‌ها می‌توان برای ارتباط با یک وب‌سرور یا یک مِیْل‌سرور (سرور ایمیل) یا انواع دیگرِ سرورها استفاده کرد. تمام آن چیزی که نیاز دارید، پیدا کردن سندی است که پروتکل را تشریح کرده، و نوشتن کدی که دریافت و ارسال داده‌ها را بر اساس آن پروتکل انجام دهد.

با وجود همه‌ی این‌ها، از آنجایی که پروتکلی که ما به وفور استفاده می‌کنیم، پروتکل وب HTTP است، پایتون یک کتابخانه‌ی اختصاصی برای آن دارد. این کتابخانه برای پشتیبانی پروتکل HTTP طراحی شده است و از آن برای دریافت و ارسال اسناد و داده‌ها در بستر وب استفاده می‌شود.

## دریافت یک تصویر بر بستر HTTP

در مثال بالا، ما یک فایل متنی ساده را گرفتیم که شامل کاراکترهای خط‌جدید در فایلش می‌شد. سپس داده‌ها را کپی کرده و هم‌زمان با اجرا، روی صفحه‌نمایش، چاپ کردیم. ما از همین برنامه برای گرفتن یک تصویر بر بستر HTTP نیز می‌توانیم استفاده کنیم. اینبار به جای کپی کرده داده‌ها و نمایش روی مانیتور، تمام داده‌ها را در یک رشته جمع کرده، هدر را جدا و سپس داده‌های عکس را به روش زیر ذخیره می‌کنیم:

```
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: http://www.py4e.com/code3/urljpeg.py
```

زمانی‌که برنامه اجرا می‌شود، خروجی زیر را خواهید دید:

```
$ python urljpeg.py
2920 2920
1460 4380
1460 5840
1460 7300
...
1460 62780
1460 64240
2920 67160
1460 68620
1681 70301
Header length 240
HTTP/1.1 200 OK
Date: Sat, 02 Nov 2013 02:15:07 GMT
Server: Apache
Last-Modified: Sat, 02 Nov 2013 02:01:26 GMT
ETag: "19c141-111a9-4ea280f8354b8"
Accept-Ranges: bytes
Content-Length: 70057
Connection: close
Content-Type: image/jpeg
```

همانگونه که می‌بینید، برای این آدرس اینترنتی، هدر Content-Type نشان می‌دهد که بدنه‌ی سند یک تصویر است (image/jpeg). زمانی‌که کار برنامه تمام شد، با باز کردن فایل stuff.jpg می‌توانید تصویر را با استفاده از یک برنامه‌ی مخصوص دیدن تصاویر مشاهده کنید.

در حین اجرای برنامه، می‌بینید که در هر بار فراخوانی متدِ recv() ما ۵۲۱۰ کاراکتر را نمی‌گیریم. در حقیقت ما آنقدر کاراکتر دریافت می‌کنیم که برای ما توسط وب‌سرور در زمانی‌که ما recv() را فراخوانی کرده‌ایم، فرستاده می‌شود. در این مثال یا ۱۴۶۰ یا ۲۹۲۰ کاراکتر در هر درخواست دریافت می‌کنیم. البته هر درخواست ما طلب ۵۱۲۰ کاراکتر واحد می‌کند با این حال وب‌سرور تعداد معینی را ارسال می‌کند.

نتایج شما با توجه به سرعت شبکه، می‌تواند متفاوت باشد. همچنین توجه کنید که آخرین فراخوانی recv() تعداد ۱۶۸۱ کاراکتر را از وب‌سرور دریافت می‌کند. دلیل این است که جریان به پایان رسیده و این قطعه از اطلاعات تمام آن چیزی‌ست که باقی مانده بوده است. در فراخوانی بعدی recv() رشته‌ای با طول صفر دریافت می‌کنیم. این بدان معنی است که سرور close() را فراخوانده و پایان سوکت خواهد بود. به عبارتی بعد از آن داده‌ای دریافت نخواهیم کرد.

کم کردنِ سرعتِ دریافت در فراخوانی recv() با فراخوانی time.sleep() میسر می‌شود (بایستی آن را آنکامنت کنید). به این طریق ما یک چهارم ثانیه بعد از هر فراخوانی منتظر می‌شویم و سرور می‌تواند جلو افتاده و داده‌ی بیشتری را قبل از اینکه ما recv() را فراخوانی کنیم، ارسال کند. با اضافه کردن این تاخیر، برنامه به شکل زیر اجرا می‌شود:

```
$ python urljpeg.py
1460 1460
5120 6580
5120 11700
...
5120 62900
5120 68020
2281 70301
Header length 240
HTTP/1.1 200 OK
Date: Sat, 02 Nov 2013 02:22:04 GMT
Server: Apache
Last-Modified: Sat, 02 Nov 2013 02:01:26 GMT
ETag: "19c141-111a9-4ea280f8354b8"
Accept-Ranges: bytes
Content-Length: 70057
Connection: close
Content-Type: image/jpeg
```

اکنون، به جز اولین و آخرین فراخوانی recv() ما هر بار ۵۱۲۰ کاراکتری را که درخواست کرده بودیم، می‌گیریم.

یک بافر بین درخواست‌های ارسال سرور "send()" و درخواست‌های دریافت برنامه‌ی ما "recv()" وجود دارد. زمانی‌که برنامه را با یک تاخیر اجرا می‌کنیم، این بافرِ موجود در سوکت، توسط سرور پر می‌شود و منتظر می‌ماند تا برنامه‌ی ما بافر را خالی کند. توقف در فرستادن یا دریافت توسط برنامه، «کنترل جریان» نامیده می‌شود.

## دریافت صفحات وب با استفاده از urllib

با وجود اینکه می‌توانیم از طریق HTTP دریافت و ارسال داده‌ها را با استفاده از کتابخانه‌ی سوکت داشته باشیم، راه‌های بسیار ساده‌تری برای انجام کارهای رایج در پایتون با استفاده از کتابخانه‌ی urllib تعبیه شده است.

با استفاده از urllib، می‌توانید با یک صفحه‌ی وب مثل یک فایل رفتار کنید. به‌سادگی، صفحه‌ای که می‌خواهید دریافت کنید را مشخص کرده و بقیه را به urllib بسپارید. urllib تمام جزئیات مرتبط با هدر و پروتکل HTTP را خودش سر و سامان می‌دهد.

کدی که برای خواندن roemo.txt نوشتیم را دیدید. حالا کدی که با استفاده از urllib نوشته شده تا همین کار را انجام دهد ببینیم:

```
import urllib.request


fhand =
urllib.request.urlopen('http://data.pr4e.org/romeo.txt')
for line in fhand:
    print(line.decode().strip())


# Code: http://www.py4e.com/code3/urllib1.py
```

همین که صفحه‌ی وب با استفاده از urllib.urlopen باز شد، می‌توانیم با آن، درست مثل یک فایل رفتار کرده و با استفاده از یک حلقه‌ی for در بین محتویات آن پیمایش کنیم.

زمانی‌که برنامه اجرا می‌شود، ما تنها خروجی که شامل محتویات فایل می‌شود را خواهیم دید. هدر فرستاده شده ولی کد urllib آن را می‌بلعد و تنها داده را برای ما برمی‌گرداند.

```
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
```

به عنوان مثال، می‌توانیم یک برنامه جهت دریافت داده‌ها برای فایل romeo.txt بنویسیم، سپس تکرار هر کلمه را در فایل به شکل زیر محاسبه 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: http://www.py4e.com/code3/urlwords.py
```

لازم است دوباره خاطرنشان کنیم که به محضی‌که یک صفحه‌ی وب را با urllib باز کردیم، می‌توانیم مثل یک فایل با او رفتار کنیم.

## تجزیه و تحلیل HTML و زدن به دلِ وب

یکی از قابلیت‌های urllib در پایتون که مورد استفاده‌ی زیادی قرار می‌گیرد، چنگ انداختن (اِسْکرِیْپ Scrape) به وب است. ما معادل خوبی برای اسکریپ پیدا نکردیم به این خاطر بهتر است کمی شما را با این اصطلاح آشنا کنم؛ به زبان ساده چه زمانی می‌گوییم به وب چنگ انداختیم؟ وقتی برنامه‌ای نوشته‌ایم که تظاهر می‌کند یک مرورگر وب است و صفحات را گرفته، داده‌های آن را مورد آزمایش قرار می‌دهد و در آن صفحات به دنبال الگوها می‌گردد اصطلاحاً به وب چنگ انداخته‌ایم.

به عنوان مثال، یک موتور جستجو، مثل گوگل به منبع یک صفحه وب نگاه کرده و لینک‌های داخل آن به صفحات دیگر را بیرون می‌کشد؛ سپس به سراغ آن صفحات رفته و لینک‌های داخل آن‌ها را نیز بیرون می‌کشد. به همین روش در درون صفحات وب می‌خزد. با استفاده از این تکنیک، گوگل راه خود را به تقریبا تمام صفحات وب باز می‌کند.

همچنین گوگل از تکرار لینک‌ها در صفحات مختلف، میزان اهمیت یک صفحه را حدس می‌زند. این صفحات مهم، بایستی که در نتایج جستجوی گوگل ظاهر شوند.

## تجزیهْ تحلیل HTML با استفاده از عبارت‌های باقاعده

یکی از راه‌های ساده‌ی تجزیهْ تحلیل HTML استفاده از عبارت‌های باقاعده برای جستجوی مکرر است؛ در این حالت زیررشته‌هایی که یک الگوی خاص دارند از دل صفحات بیرون کشیده می‌شوند.

یک صفحه‌ی بسیار ساده وب را با هم ببینیم:

```
<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>
```

می‌توانیم یک عبارت ساده‌ی درست و حسابی برای پیدا کردن و استخراج مقدارِ لینک‌ها از متن بالا بنویسیم:

```
href="http://.+?"
```

عبارت باقاعده‌ی ما به دنبال رشته‌ای می‌گردد که با "href="http://" شروع شده و در ادامه یک یا چند کاراکتر دارد ("?+.") و در نهایت به یک نقل‌قول دوگانه می‌رسد. علامت سوال در عبارت باقاعده می‌گوید که به دنبال زیررشته‌ها در حالت غیرحریصانه بگرد. همانطور که در فصل پیش توضیح دادیم، در حالت غیرحریصانه، برنامه به دنبال کوتاه‌ترین زیررشته‌ای می‌گردد که با عبارت باقاعده‌ی ما جور در آید. در حالت حریصانه برعکس این قضیه صادق است و برنامه به دنبال بزرگترین زیررشته‌ی ممکن در رشته‌ی اصلی می‌گردد.

حالا پرانتز را به عبارت باقاعده‌ی بالا اضافه می‌کنیم، تا برنامه درست آن لینکی که ما می‌خواهیم را از دل متن استخراج کرده و تحویل دهد:

```
# Search for lines that start with From and have an at sign
import urllib.request, urllib.parse, urllib.error
import re


url = input('Enter - ')
html = urllib.request.urlopen(url).read()
links = re.findall(b'href="(http://.*?)"', html)
for link in links:
    print(link.decode())


# Code: http://www.py4e.com/code3/urlregex.py
```

متد عبارت باقاعده‌ی findall لیستی از رشته‌هایی که با عبارت باقاعده‌ی ما جور در می‌آیند و بین دو علامت نقل‌قول محصور شده‌اند را برمی‌گرداند. به جایگاه پرانتزها دقت کنید که درست بعد و قبل از علامت نقل‌قول قرار داده شده‌اند تا عبارت داخل آن را برگرداند.

زمانی‌که برنامه را اجرا می‌کنیم، خروجی زیر را می‌گیریم:

```
python urlregex.py
Enter - http://www.dr-chuck.com/page1.htm
http://www.dr-chuck.com/page2.htm
python urlregex.py
Enter - http://www.py4e.com/book.htm
http://www.greenteapress.com/thinkpython/thinkpython.html
http://allendowney.com/
http://www.py4e.com/code
http://www.lib.umich.edu/espresso-book-machine
http://www.py4e.com/py4inf-slides.zip
```

زمانی‌که فایل HTML شما به خوبی قالب‌بندی شده باشد، استفاده از عبارت‌های باقاعده برای پیدا کردن زیررشته‌ها با الگوهای خاص خیلی ساده و روان خواهد بود. اما از آنجایی که صفحات شکسته‌ی HTML زیادی وجود دارد، ممکن است باعث شود داده‌های به‌دردنخوری را جمع‌آوری کنید. به عبارتی پیوندهای بسیاری را دریافت کنید که برخی از آن‌ها به صفحاتی که وجود ندارد ختم شده باشند؛ به زبان ساده‌تر پیوندهای شکسته و خراب!

این موضوع با یک کتابخانه‌ی قدرتمند تحلیل HTML حل شدنی است!

## تحلیل HTML با استفاده از BeautifulSoup

تعدادی کتابخانه پایتونی برای کمک به شما در زمینه‌ی تحلیل HTML و استخراج داده از صفحات وجود دارد. هر کدام از این کتابخانه‌ها نقاط قوت و ضعف مخصوص خود را دارند. شما بایستی براساس نیاز خود، یکی را انتخاب کنید.

به عنوان مثال، ما به سادگی و با استفاده از کتابخانه BeautifulSoup چند ورودی HTML را تجزیه و لینک‌ها را استخراج می‌کنیم. می‌توانید کد BeautifulSoup را از لینک زیر دریافت کرده و آن را نصب کنید:

http://www.crummy.com/software

می‌توانید BeautifulSoup را دانلود و نصب کنید؛ و یا به سادگی فایل BeautifulSoup.py را در پوشه‌ی برنامه‌ی خود قرار دهید.

با وجود اینکه HTML شبیه به XML است، و برخی صفحات با دقت برای XML بودن ساخته شده‌اند، یک تحلیل‌کننده‌ی XML در مواجه با اغلب HTMLهای شکسته و خراب شده، به دلیل مناسب نبودن قالب آن، کل صفحه‌ی HTML را کنار می‌گذارد. در سمت دیگر BeautifulSoup تا جایی که بتواند با HTML مشکل‌دار کنار می‌آید و انعطاف‌پذیرتر است؛ و به شما اجازه‌ی استخراج داده‌های مورد نیازتان را می‌دهد.

ما از urllib برای خواندن صفحه استفاده می‌کنیم؛ سپس با استفاده از BeautifulSoup صفت‌های/attributes مرتبط با href یک تگ a را (که همان لینک‌ها یا پیوندها خواهد بود) بیرون می‌کشیم.

```
# To run this, you can install BeautifulSoup
# https://pypi.python.org/pypi/beautifulsoup4


# Or download the 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: http://www.py4e.com/code3/urllinks.py
```

برنامه درخواست یک آدرس وب می‌کند و سپس با باز کردن آن صفحه، داده‌ها را خوانده و به تحلیل‌گر BeautifulSoup می‌فرستد. سپس تمام تگ‌های انکر (تگ‌های مخصوص لینک‌ها که با a مشخص می‌شود) را بیرون کشیده و صفتِ href هرکدام از این تگ‌ها را چاپ می‌کند.

وقتی برنامه اجرا می‌شود، چیزی شبیه به خروجی زیر ظاهر خواهد شد:

```
python urllinks.py
Enter - http://www.dr-chuck.com/page1.htm
http://www.dr-chuck.com/page2.htm
python urllinks.py
Enter - http://www.py4e.com/book.htm
http://www.greenteapress.com/thinkpython/thinkpython.html
http://allendowney.com/
http://www.si502.com/
http://www.lib.umich.edu/espresso-book-machine
http://www.py4e.com/code
http://www.py4e.com/
```

می‌توانید از BeautifulSoup برای بیرون کشیدن هر کدام از قسمت‌های تگ استفاده کنید:

```
# To run this, you can install BeautifulSoup
# https://pypi.python.org/pypi/beautifulsoup4
# Or download the 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()


# html.parser is the HTML parser included in the standard
Python 3 library.
# information on other HTML parsers is here:
#http://www.crummy.com/software/BeautifulSoup/bs4/doc/#insta
lling-a-parser
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: http://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')]
```

این مثال‌ها تنها به شما قدرت BeautifulSoup را وقتی حرف از تحلیل HTML در میان است نشان می‌دهد.

## خواندن فایل‌های باینری با استفاده از urllib

گاهی لازم است که یک فایل باینری (غیرمتنی) را بگیرید؛ مثلا یک فایل ویدیویی یا یک تصویر را دریافت کنید. داده‌های این فایل‌ها، به درد چاپ با استفاده از print نمی‌خورند. با این حال شما با استفاده از urllib می‌توانید یک کپی از آن‌ها را گرفته و روی هارد دیسک خود به عنوان یک فایل محلی ذخیره کنید.

الگو به این صورت است که شما یک URL را باز می‌کنید و سپس با استفاده از read کل محتوای سند را در یک متغیر رشته (img) دانلود می‌کنید و سپس آن اطلاعات را روی یک فایل محلی به شکل زیر می‌نویسید:

```
import urllib.request, urllib.parse, urllib.error


img =
urllib.request.urlopen('http://data.pr4e.org/cover3.jpg').re
ad()
fhand = open('cover3.jpg', 'wb')
fhand.write(img)
fhand.close()


# Code: http://www.py4e.com/code3/curl1.py
```

این برنامه تمام داده‌ها را یک‌ضرب از طریق شبکه خوانده و در متغیری به اسم img در حافظه‌ی اصلی کامپیوتر نگه‌داری می‌کند؛ سپس فایل cover3.jpg را باز کرده و داده‌ها را روی آن و دیسک شما می‌نویسد. این مورد، اگر اندازه‌ی فایل کمتر از حافظه‌ی اصلی کامپیوتر شما باشد، به کار می‌آید.

با این حال اگر یک فایل بزرگی مثل فایل ویدیویی یا صوتی در کار باشد، احتمالا برنامه یا کرش می‌کند یا حداقل وقتی کامپیوتر با کمبود حافظه مواجه شد، به شدت کُند می‌شود. برای اینکه با مشکل کمبود حافظه مواجه نشویم، ما داده‌ها را در بلاک‌ها (یا بافرها) دریافت می‌کنیم و هر بلاک را قبل از دریافت بلاک بعدی روی دیسک یا حافظه‌ی جانبی می‌نویسیم و سپس به سراغ قطعه‌ی بعدی می‌رویم. به این شکل هرچقدر هم که اندازه‌ی فایل بزرگ باشد، حافظه‌ی ما با مشکل مواجه نخواهد شد چرا که برنامه تنها بلاک‌های در حال دریافت و پردازش را روی حافظه دارد و مابقیِ داده‌های دریافت شده روی دیسک نوشته شده‌اند و جا را برای داده‌های دیگر روی حافظه‌ی اصلی خالی کرده‌اند.

```
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: http://www.py4e.com/code3/curl2.py
```

در این مثال ما تنها ۱۰۰,۰۰۰ کاراکتر را در آنِ واحد خوانده و آن کاراکترها را روی فایل cover.jpg می‌نویسیم؛ سپس بعد از نوشتن این کاراکترها روی فایل، به سراغ دریافت قطعه‌ی بعدی، یعنی ۱۰۰,۰۰۰ کاراکتر بعدی، از وب می‌رود.

برنامه به شکل زیر اجرا می‌شود:

```
python curl2.py
568248 characters copied.
```

اگر شما یک کامپیوتر مکینتاش یا یونیکسی داشته باشید، یک دستور توکاری شده در سیستم‌عامل‌تان وجود دارد که همین کار را به شکل زیر انجام می‌دهد:

```
curl -O http://www.py4e.com/cover.jpg
```

دستور curl کوتاه‌شده‌ی "CopyURL" است و مثال‌های ما هم زیرکانه curl1.py و curl2.py در www.py4e.com/code3 نام‌گذاری شدند؛ چرا که مثال‌های ما هم مثل دستور curl عمل می‌کنند. البته یک برنامه با اسم curl3.py هم در آدرس بالا وجود دارد که کاری که شرحش رفت را به روش موثرتری انجام می‌دهد.

## واژگان فصل

**BeautifulSoup:**
یک کتابخانه پایتونی که اسناد HTML را تحلیل کرده و داده‌هایی از اسناد HTML ناقص بیرون می‌کشد که اکثر مرورگرها آن‌ها را نادیده می‌گیرند. شما می‌توانید کد BeautifulSoup را از www.crummy.com دانلود کنید.

**پورت / Port:**
پورت عبارت است از عددی که مشخص می‌کند شما با کدام برنامه در زمانی‌که از طریق یک ارتباط سوکت به سرور وصل شده‌اید، در ارتباط هستید.

**چنگ‌انداختن / Scrape:**
به تلاش یک برنامه برای جا زدن خودش به عنوان یک مرورگر وب، و دریافت و کاوش یک صفحه‌ی وب و محتویاتش، چنگ‌انداختن یا اسکریپت می‌گوییم.

**سوکت / Socket:**
یک ارتباط شبکه بین دو برنامه که یک برنامه می‌تواند به سمت دیگری داده‌ها را ارسال یا از آن دریافت کند سوکت نام دارد.

**اسپایدر / Spider:**
اسپایدر به عمل یک موتور جستجو برای دریافت یک صفحه و تمام صفحات لینک‌شده از یک صفحه و خزیدن در بین آن‌ها گفته می‌شود. آن‌ها به همین روش تقریبا تمام صفحات اینترنت را برای ساخت یک فهرست کاملِ جستجو، کاوش می‌کنند.

## تمرین‌ها

**تمرین ۱:** برنامه‌ی سوکت socket1.py را تغییر دهید تا از کاربر URL درخواست کند و برای هر صفحه‌ای قابل اجرا باشد. می‌توانید از split('/') برای شکستن URL به بخش‌های تشکیل‌دهنده‌اش استفاده کنید؛ به این ترتیب می‌توانید نام هاست را از فراخوانی سوکت connect جدا کنید.

**تمرین ۲:** برنامه‌ی سوکت‌تان را جوری تغییر دهید که تعداد کاراکترهایی که می‌گیرد را شمرده و بعد از دریافت ۳۰۰۰ کاراکتر، دیگر آن‌ها را نمایش ندهد. برنامه بایستی کل سند را بگیرد و تعداد کل کاراکترها را بشمارد و سپس تعداد کاراکترها را در انتهای سند نمایش دهد.

**تمرین ۳:** با استفاده از urllib تمرین قبل را بازنویسی کنید؛ با این تفاوت که **۱)** سند را از یک URL بگیرد و **۲)** سه هزار کاراکتر را نمایش دهد و **۳)** تعداد کل

کاراکترهای سند را بشمارد. نگران هدر نباشید و تنها اولین ۳۰۰۰ کاراکتر را چاپ کنید.

**تمرین ۴:** برنامه‌ی urllinks.py را به شکلی تغییر دهید که پاراگراف‌ها را از سند HTML شمرده (به تگ p دقت کنید) و تعداد آن‌ها را چاپ کند. برنامه نبایستی که متن پاراگراف‌ها را چاپ کند و تنها آن‌ها را بشمارد. برنامه‌ی خودتان را روی چند صفحه‌ی کوچک و بزرگ وب تست کنید.

**تمرین ۵:** (پیشرفته) برنامه‌ی سوکت را به شکلی تغییر دهید که تنها داده‌هایی که بعد از هدرها و یک خط خالی می‌آیند را دریافت کند. به خاطر داشته باشید که recv تمام کاراکتر (من جمله خط‌جدید) را دریافت می‌کند.

# فصل ۱۳

# پایتون و سرویس‌های وب

## استفاده از سرویس‌های وب

وقتی با دریافت اسناد و تجزیه تحلیل آن‌ها بر بستر HTTP آشنا شدید و کار با آن‌ها برای‌تان ساده شد، زمان زیادی نمی‌گذرد که به سمت نوشتن برنامه‌هایی سوق پیدا می‌کنید که برای استفاده توسط برنامه‌های دیگر طراحی شده‌اند (نه فقط HTML که در یک مرورگر نمایش داده شود).

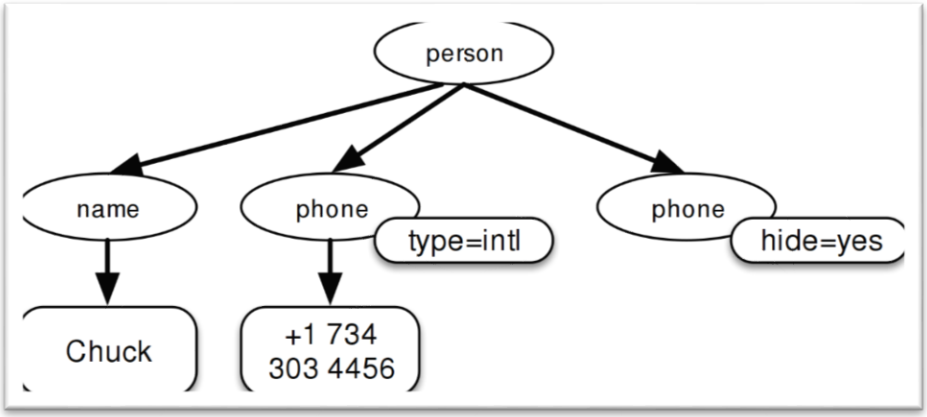
دو قالب رایج برای تبادل داده‌ها بر بستر وب وجود دارد. XML یا «زبان نشانه‌گذاری گسترش‌پذیر» سال‌های زیادی است که برای تبادل داده‌های به شکل سند، مورد استفاده قرار می‌گیرد. زمانیکه برنامه‌ها می‌خواهند دیکشنری، لیست یا اطلاعات داخلی دیگر را تبادل کنند، از نشانه‌گذاری شئ جاوااسکریپت یا JSON استفاده می‌کنند. در ادامه هر دو فرمت را بررسی خواهیم کرد.

## زبان نشانه‌گذاری گسترش‌پذیر - XML

زبان نشانه‌گذاری گسترش‌پذیر بسیار شبیه به زبان نشانه‌گذاری ابرمتن یا HTML است با این تفاوت که XML ساخت‌یافته‌تر از HTML است. به یک نمونه سند XML دقت کنید:

```
<person>
  <name>Chuck</name>
  <phone type="intl">
     +1 734 303 4456
   </phone>
   <email hide="yes"/>
</person>
```

تصور سند XML به صورت یک ساختار درختی در بیشتر مواقع به شما کمک می‌کند تا درک بهتری از آن داشته باشید. به عنوان نمونه در مثال بالا person تگِ رده بالاتری نسبت به بقیه است. همانگونه که در تصویر زیر مشاهده می‌کنید phone به عنوان بچه‌ای از گره‌ی والدِ[I] خود کشیده شده است؛ والد در اینجا تگ person است.




## تجزیه تحلیل XML

در اینجا یک برنامه‌ی ساده را برای تحلیل XML و استخراج برخی از المنت‌ها، می‌بینید:

[I] parent node

```
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: http://www.py4e.com/code3/xml1.py
```

فراخوانی fromstring نمونه‌ی رشته XML را به درختی از گره‌های XML بدل می‌کند. زمانی که XML در داخل یک درخت قرار دارد، به وسیله‌ی متدهایی که در اختیار داریم می‌توانیم بخشی از داده‌ها را استخراج کنیم.

تابع find در درخت XML پیمایش کرده و گره‌ای که با تگ یا برچسب خاصی جور در می‌آید را پیدا می‌کند. هر گره می‌تواند متن، صفت (مانند پنهان بودن) و فرزند داشته باشد. هر گره می‌تواند خودْ گره‌ی بالای چندین گره باشد (به عبارتی والد آن‌ها باشد).

```
Name: Chuck
Attr: yes
```

استفاده از یک تحلیل‌گر XML مانند ElementTree مزایای خاص خود را دارد. ElementTree به شما اجازه می‌دهد که داده‌ها را بدون نگرانی از سینتکس یا متن XML استخراج کنیم.

## حلقه زدن در گره‌ها

اغلبْ XML گره‌های چندگانه دارد و برای پردازش همه‌ی آن‌ها لازم است که یک حلقه بنویسیم. در برنامه‌ی زیر، در بین تمام گره‌های user پیمایش می‌کنیم:

```
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: http://www.py4e.com/code3/xml2.py
```

متد findall یک لیست پایتونی از زیردرخت‌ها را که ساختارهای user را در درخت XML خود دارد، دریافت می‌کند. سپس می‌توانیم با نوشتن یک حلقه‌ی for که به هر

کدام از گره‌های کاربر نگاه می‌اندازد، name و id و صفت x را از گره‌ی user چاپ کنیم:

```
User count: 2
Name Chuck
Id 001
Attribute 2
Name Brent
Id 009
Attribute 7
```

## نشانه‌گذاری شئ جاوااسکریپت - JSON

قالب JSON از فرمت آرایه و شی که در زبان جاوااسکریپت مورد استفاده قرار می‌گرفت، الهام گرفته است. اما از آنجایی که پایتون قبل از جاوااسکریپت ابداع شده بود، متن دیکشنری و لیست‌های پایتون، سینتکسِ JSON را تحت تاثیر قرار داده است. فرمت JSON تقریبا مشابه با ترکیب لیست‌ها و دیکشنری‌های پایتون است.

در اینجا نمونه‌ای از متن JSON را آورده‌ایم. این کد با نمونه‌ی ساده‌ی XML که بالاتر دیدید، تقریبا برابر است:

```
{
  "name" : "Chuck",
  "phone" : {
    "type" : "intl",
    "number" : "+1 734 303 4456"
   },
   "email" : {
     "hide" : "yes"
   }
}
```

چند تفاوت را احتمالا خواهید دید. اول اینکه در XML ما می‌توانیم صفت‌هایی مثل intl را به تگِ phone اضافه کنیم. در JSON ما جفتِ کلید-مقدار را داریم. همچنین تگ person در XML در اینجا وجود ندارد و جایش را به چند آکولاد داده است.

در کل، ساختارهای JSON ساده‌تر از XML است؛ چرا که JSON قابلیت‌های کمتری نسبت به XML دارد. اما JSON این مزیت را دارد که از ترکیب دیکشنری‌ها و لیست‌ها به صورت مستقیم بهره ببرد.

از آنجایی که تقریبا تمام زبان‌های برنامه‌نویسی چیزی شبیه به دیکشنری‌ها و لیست‌های پایتون را دارند، JSON یک قالب طبیعی و خودمانی برای تبادل اطلاعات بین دو برنامه را دارد.

JSON به سرعت در حال تبدیل شدن به قالبِ غالب، برای تبادل داده‌ها بین برنامه‌هاست چرا که نسبت به XML ساده‌تر است.

## تجزیه تحلیل JSON

JSON را با دیکشنری‌های تو در تو و لیست‌ها – در صورت نیاز – ساختاربندی کردیم. در مثال پیشِ رو لیستی از کاربران به صورت جفت‌های کلید-مقدار (به‌مانند دیکشنری) ارائه می‌کنیم. در حقیقت لیستی از دیکشنری‌ها را خواهیم داشت.

در برنامه‌ی پایین، با استفاده از کتابخانه‌ی داخلی json شروع به تحلیل JSON و خواندن داده‌ها می‌کنیم. چیزی که در اینجا می‌بینید، تقریبا شبیه به همان است که در مثال XML دیدید. JSON جزئیات کمتری دارد و بایستی از قبل بدانیم که در حال گرفتن یک لیست هستیم و این لیستِ کاربران برای هر کاربر شامل یک ست از جفت کلید-مقدار می‌شود. JSON مختصرتر (یک ویژگی خوب) ولی کمی گنگ (یک ویژگی بد) است.

```
import json


data = '''
[
  { "id" : "001",
    "x" : "2",
    "name" : "Chuck"
  } ,
  { "id" : "009",
    "x" : "7",
    "name" : "Chuck"
  }
]'''


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: http://www.py4e.com/code3/json2.py
```

اگر کدِ استخراج داده‌ها توسط تحلیل‌گرهای مرتبط با JSON و XML را با هم مقایسه کنید، خواهید دید آن چیزی که از json.loads() دریافت می‌کنید یک لیست پایتونی است. درست مثل داده‌ها با ساختار یک لیست در پایتون پیمایش کرده‌ایم. به عبارتی پیمایش با استفاده از یک حلقه‌ی for درون آیتم‌هایی – که هر کدام یک دیکشنری پایتونی است – صورت پذیرفت. همین که JSON تحلیل شد، از عملگر شاخص در پایتون به منظور استخراج قطعه‌های اطلاعات از هر کاربر می‌توانیم استفاده کنیم. لازم نیست که از کتابخانه‌ی JSON برای کَند و کاو در درون JSON ــ تحلیل‌شده استفاده کنیم؛ چرا که خروجی ما ساختار پایتونی دارد و خود پایتون برای پردازش آن‌ها کافی است. یک بار دیگر به کد نگاه کنید.

خروجی برنامه دقیقا مانند نمونه‌ی XML بالا است:

```
User count: 2
Name Chuck
Id 001
Attribute 2
Name Brent
Id 009
Attribute 7
```

در دنیای سرویس‌های وب، هم‌اکنون حرکتی از XML به سمت JSON وجود دارد؛ به این خاطر که JSON ساده‌تر است و داده‌های آن به ساختاری که در زبان‌های برنامه‌نویسی استفاده می‌کنیم بسیار نزدیکتر است. زمانی که از JSON استفاده می‌کنیم کدِ تحلیل و استخراج داده‌ها ساده و سرراست‌تر است. ولی ساختار XML مفهوم‌تر است و بهتر خودش را تشریح می‌کند. به هر حال بسته به محتوای برنامه، انتخاب بین XML و JSON در رفت و آمد است. به عنوان مثال اغلب واژه‌پردازها برای ذخیره‌ی داده‌های درونی از XML استفاده می‌کنند.

## رابط برنامه‌نویسی نرم‌افزار کاربردی

اکنون قابلیت تبادل اطلاعات بین برنامه‌ها را با استفاده از HTTP و یک راه برای ارائه‌ی داده‌های پیچیده که بین این برنامه‌ها می‌فرستیم یا دریافت می‌کنیم با استفاده از XML یا JSON در چنته داریم.

قدم بعدی تعریف و مستند کردن قراردادهایی است که بین برنامه‌ها برای استفاده از این تکنیک‌ها می‌نویسیم. اسم رایج برای این قراردادهای بین برنامه‌ای «رابط برنامه‌نویسی نرم‌افزار کاربردی» یا APIها هستند.

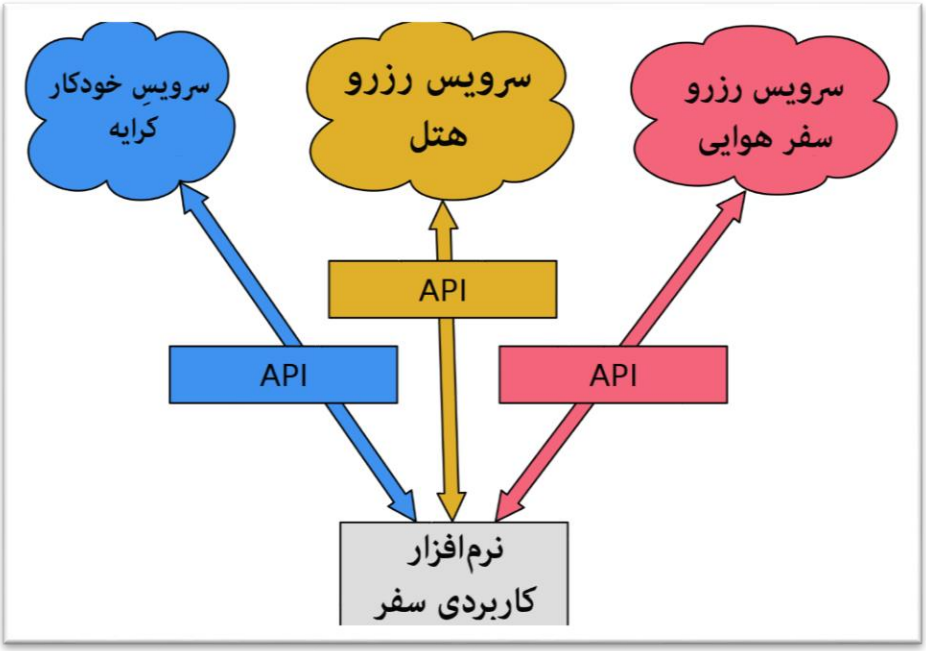
چه زمانی از یک API استفاده می‌کنیم؟ فرض کنید برنامه‌ای دارید که می‌خواهد از سرویس‌های یک برنامه‌ی دیگر استفاده کند. برنامه‌ی دوم در کل یک دسته از سرویس‌ها را برای برنامه‌ی اول آماده می‌کند. این قراردادهایی که ارتباط سرویس اول و دوم را فراهم می‌کند API نامیده می‌شود و ما زمانی که می‌خواهیم این رابطه را بسازیم از آن‌ها

استفاده می‌کنیم. برنامه‌ی دوم، APIها یا همان قوانین مخصوص خود را منتشر می‌کند و برنامه‌ی اول با تبعیت از این قوانین دسترسی به سرویس‌های آن را، طبق چیزی که تعریف شده، پیدا می‌کند.

حالا فرض کنید شروع به ساختن برنامه‌ای کرده‌اید که برای عملکرد نیاز به دسترسی به سرویس‌هایی دارد که توسط برنامه‌های دیگری فراهم می‌شود. ما به این نوع رویکرد «معماری سرویس‌گرا» یا SOA[I] می‌گوییم. رویکرد SOA یکی از جاهایی است که برنامه‌ی ما از سرویس‌های دیگر برنامه‌ها استفاده می‌کند. یک رویکرد غیر SOA زمانیست که برنامه، یک برنامه‌ی خوداتکا بوده و شامل تمام کدهای مورد نیاز برای پیاده‌سازی مفهوم شود.

زمانی که از وب استفاده می‌کنیم مثال‌های زیادی از SOA را خواهیم دید. می‌توانیم به یک سایت رفته و سفرهوایی، هتل و ماشین رزرو کنیم. داده‌های مرتبط با هتل‌ها در کامپیوترهای شرکت هواپیمایی ذخیره نشده‌اند. در عوض کامپیوترهای شرکت هواپیمایی به سرویس‌هایی که روی کامپیوترهای هتل هستند متصل شده و داده‌ها را دریافت و به کاربر ارائه می‌کند. زمانیکه کاربر به رزرو هتل از طریق سایت شرکت هواپیمایی می‌پردازد، شرکت هواپیمایی از یک سرویس دیگر وب روی سیستم‌های هتل برای رزرو کردن جا استفاده خواهد کرد. حالا زمانیکه قرار است پول رزرو را پرداخت کنید، پروسه به سمت دیگری برای انجام تراکنش سوق پیدا می‌کند و سیستم‌های دیگری نیز درگیر این جریان می‌شوند.

[I] Service-oriented Architecture




یک معماری سرویس‌گرا مزایای فراوانی دارد. به عنوان مثال:

۱) ما تنها یک رونوشت از داده‌ها را حفظ و نگه‌داری می‌کنیم (این موضوع برای مواردی مثل رزرو هتل بسیار مهم است چرا که نمی‌خواهیم تعهد بی‌اندازه به مشتری بدهیم)؛ و

۲) صاحبِ داده‌ها می‌تواند قوانین خود برای استفاده از داده‌هایش را تعیین کند.

با این مزایا، یک سیستم SOA بایستی با دقت طراحی شود که هم عملکرد خوبی داشته باشد و هم با نیازهای کاربر به خوبی جور در بیاید.

به سرویس‌هایی که یک برنامه از طریق API در دسترس وب قرار می‌دهد، وب‌سرویس می‌گوییم.

## وب سرویس Google Geocoding

گوگل یک وب‌سرویس بسیار عالی برای استفاده از پایگاه داده‌ی عظیم خود از اطلاعات جغرافیایی در اختیار ما قرار داده است. ما یک رشته‌ی جستجو مثال " Ann Arbor, MI" را به رابط برنامه‌نویسی کاربردی ژئوکدینگ[I] گوگل ارسال می‌کنیم و در پاسخ، گوگل بهترین مکانی که حدس می‌زند به رشته‌ی جستجو شده توسط ما نزدیک است را برمی‌گرداند و به شما در خصوص مکان‌های خاصِ آنجا اطلاعاتی را بیان می‌کند.

سرویس ژئوکدینگْ رایگان، ولی محدود است، در نتیجه نمی‌توانید در برنامه‌ی تجاری خود از این API استفاده‌ی نامحدودی داشته باشید. اما اگر شما برای دریافت مکان کاربر نهایی و سر و سامان دادن به برنامه بخواهید از آن استفاده کنید، به کارتان خواهد آمد.

زمانی که از یک API رایگان مثل ژئوکدینگ استفاده می‌کنید، بایستی که قدر منابعی که در اختیارتان گذاشته شده را بدانید و به سایر کاربران احترام بگذارید. اگر افراد زیادی به استفاده بیش از اندازه (بخوانید سوءاستفاده) از این سرویس‌ها بپردازند احتمال اینکه گوگل کل سرویس را از دسترس خارج کند یا محدودیت‌های ویژه‌ای اعمال کند بالا می‌رود.

شما می‌توانید سند آنلاین این سرویس را بخوانید؛ هرچند کار با این سرویس بسیار ساده است و حتی با تایپ URL زیر در مرورگر می‌توانید اطلاعات بازگردانده شده توسط آن را ببینید:

```
http://maps.googleapis.com/maps/api/geocode/json?address=Ann+Arbor%2C+MI
```

لینک بالا شامل هیچ فاصله‌ی اسپیس نمی‌شود، در نتیجه مطمئن شوید که آدرس را درست وارد می‌کنید.

---

[I] Geocoding

برنامه‌ی زیر، یک برنامه‌ی کاربردی ساده برای درخواست یک رشته از کاربر و ارسال آن به سرویس ژئوکدینگ گوگل است. برنامه در ادامه اطلاعات را از JSON بازگردانده شده استخراج می‌کند.

```
import urllib.request, urllib.parse, urllib.error
import json


# Note that Google is increasingly requiring keys
# for this API
serviceurl =
'http://maps.googleapis.com/maps/api/geocode/json?'


while True:
    address = input('Enter location: ')
    if len(address) < 1: break


    url = serviceurl + urllib.parse.urlencode(
        {'address': address})


    print('Retrieving', url)
    uh = urllib.request.urlopen(url)
    data = uh.read().decode()
    print('Retrieved', len(data), 'characters')


    try:
        js = json.loads(data)
    except:
        js = None


    if not js or 'status' not in js or js['status'] != 'OK':
        print('==== Failure To Retrieve ====')
        print(data)
        continue


    print(json.dumps(js, indent=4))


    lat = js["results"][0]["geometry"]["location"]["lat"]
    lng = js["results"][0]["geometry"]["location"]["lng"]
```

```
    print('lat', lat, 'lng', lng)
    location = js['results'][0]['formatted_address']
    print(location)


# Code: http://www.py4e.com/code3/geojson.py
```

برنامه، رشته‌ی جستجو را دریافت کرده و یک URL با استفاده از آن می‌سازد. سپس با استفاده از urllib متن مورد نظر را از API ژئوکدینگ گوگل دریافت می‌کند. برخلاف یک صفحه‌ی وب ثابت، داده‌ها بر اساس پارامترهایی که ارسال می‌کنیم و داده‌های جغرافیایی موجود در سرورهای گوگل متفاوت خواهد بود.

زمانی که داده JSON را دریافت کردیم آن‌ها را با استفاده از کتابخانه json تحلیل کرده و بررسی کوتاهی را برای اطمینان از گرفتن داده‌های خوب انجام می‌دهیم. در نهایت اطلاعاتی را که به دنبالش بودیم، استخراج می‌کنیم.

خروجی برنامه شبیه به این خواهد بود (برخی از JSON برگرداننده شده پاک شده‌اند):

```
$ python3 geojson.py
Enter location: Ann Arbor, MI
Retrieving http://maps.googleapis.com/maps/api/
  geocode/json?address=Ann+Arbor%2C+MI
Retrieved 1669 characters
{
    "status": "OK",
    "results": [
        {
            "geometry": {
                "location_type": "APPROXIMATE",
                "location": {
                    "lat": 42.2808256,
                    "lng": -83.7430378
                }
            },
            "address_components": [
                {
                    "long_name": "Ann Arbor",
                    "types": [
                        "locality",
                        "political"
                    ],
                    "short_name": "Ann Arbor"
                }
            ],
            "formatted_address": "Ann Arbor, MI, USA",
            "types": [
                "locality",
                "political"
            ]
        }
    ]
}
lat 42.2808256 lng -83.7430378
Ann Arbor, MI, USA
Enter location:
```

می‌توانید این یکی برنامه را از www.py4e.com/code3/geoxml.py دانلود کنید و کمی به کاوش در این سرویس، اینبار با XML بپردازید.

## امنیت و استفاده از API

برای استفاده از API یک شرکت معمولا شما نیاز به نوعی کلید / API key خواهید داشت. ایده‌ی اصلی این است که مشخص شود چه شخصی و به چه میزان از سرویس ایشان در حالِ استفاده است. احتمال دارد که سرویس‌های رایگان یا پولی داشته باشند و یا قوانین خاصی که تعداد درخواست‌ها در بازه‌ی محدودی از زمان از طرف یک شخص را محدود می‌کند در خصوص سرویس‌ها برقرار باشد.

گاهی وقتی کلید API خود را دریافت می‌کنید، این کلید جزئی از داده‌ی POST خواهد بود و یا به عنوان یک پارامتر در URL در زمان احضار API مورد استفاده قرار خواهد گرفت.

در زمان‌های دیگر، ارائه‌کننده برای اطمینان بیشتر از منبعِ درخواست‌ها، از شما درخواست امضای رمزشده می‌کند. این امضا شامل کلیدها و رمزینه‌های مخصوص به خود است. یک تکنولوژی بسیار رایج برای امضای درخواست در اینترنت OAuth خوانده می‌شود. در پیوند زیر می‌توانید در خصوص پروتکل OAuth اطلاعات بیشتری را کسب کنید:

www.oauth.net

با توجه به اینکه API توییتر بسیار ارزشمند شده است، این شرکت از یک API باز و عمومی به سمت تدارک یک API رفت که از امضاهای OAuth برای هر درخواست استفاده می‌کند. خوشبختانه هنوز تعدادی کتابخانه رایگان و سهل‌الوصول OAuth وجود دارد تا شما را از پیاده‌سازی OAuth از پایه نجات دهد. این کتابخانه‌ها از نظر پیچیدگی و غنی بودن در درجات مختلفی وجود دارند. وب‌سایت OAuth اطلاعاتی را در خصوص تعداد مختلفی از این کتابخانه‌ها در خود دارد.

برای برنامه‌ی بعدی، ما فایل‌های twurl.py و hidden.py و oauth.py و twitter1.py را از www.py4e.com/code دانلود کرده و همه را در یک پوشه قرار می‌دهیم.

برای استفاده از این برنامه‌ها لازم است که یک اکانت توییتر داشته باشید و اجازه برای کدِ پایتون خود را به عنوان یک برنامه‌ی کاربردی بگیرید. سپس یک کلید، رمزینه، نشانه[I] و رمزینه نشانه[II] تنظیم کنید. شما باید فایل hidden.py را ویرایش کرده و این چهار رشته را در متغیرهای خود در فایل قرار دهید:

```
# Keep this file separate


# https://apps.twitter.com/
# Create new App and get the four strings


def oauth():
    return {"consumer_key": "h7Lu...Ng",
            "consumer_secret": "dNKenAC3New...mmn7Q",
            "token_key": "10185562-eibxCp9n2...P4GEQQOSGI",
            "token_secret": "H0ycCFemmC4wyf1...qoIpBo"}


# Code: http://www.py4e.com/code3/hidden.py
```

وب‌سرویس توییتر از یک URL شبیه به این در دسترس خواهد بود:

https://api.twitter.com/1.1/statuses/user_timeline.json

ولی زمانی که تمام اطلاعات امنیتی اضافه شد، URL بیشتر شبیه به این یکی خواهد شد:

---

[I] token

[II] token secret

```
https://api.twitter.com/1.1/statuses/user_timeline.json?coun
t=2
&oauth_version=1.0&oauth_token=101...SGI&screen_name=drchuck
&oauth_nonce=09239679&oauth_timestamp=1380395644
&oauth_signature=rLK...BoD&oauth_consumer_key=h7Lu...GNg
&oauth_signature_method=HMAC-SHA1
```

می‌توانید مشخصات OAuth را برای درک معنای پارامترهای مختلف بخوانید. این پارامترهای اضافه شده برای پیش‌نیازهای امنیتی الزامی است.

برای برنامه‌ای که با توییتر اجرا کردیم، همه این پیچیدگی‌ها را در فایل‌های oauth.py و twurl.py قرار دادیم. رمزینه‌ها را در hidden.py گذاشتیم و سپس URL مورد نظر را به تابع twurl.augment() فرستادیم و در نهایت کد کتابخانه، تمام پارامترهای لازم را به URL اضافه کرد.

این برنامه گاه‌شمار[I] یک کاربر خاص توییتر را گرفته و آن را به فرمت JSON برای ما باز می‌گرداند. ما هم تنها اولین ۲۵۰ کاراکتر رشته را چاپ می‌کنیم:

[I] تایم‌لاین، Timeline عبارت است از فهرستی از وقایع با ترتیب بر اساس زمان. در توییتر به صفحه‌ی نوشته‌های کاربرها که بر اساس ترتیب زمانی مشخص شده گاه‌شمار یا تایم‌لاین می‌گویند.

```
import urllib.request, urllib.parse, urllib.error
import twurl
import ssl


# https://apps.twitter.com/
# Create App and get the four strings, put them in hidden.py


TWITTER_URL =
'https://api.twitter.com/1.1/statuses/user_timeline.json'


# Ignore SSL certificate errors
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE


while True:
    print('')
    acct = input('Enter Twitter Account:')
    if (len(acct) < 1): break
    url = twurl.augment(TWITTER_URL,
                        {'screen_name': acct, 'count': '2'})
    print('Retrieving', url)
    connection = urllib.request.urlopen(url, context=ctx)
    data = connection.read().decode()
    print(data[:250])
    headers = dict(connection.getheaders())
    # print headers
    print('Remaining', headers['x-rate-limit-remaining'])


# Code: http://www.py4e.com/code3/twitter1.py
```

زمانی که برنامه اجرا می‌شود، خروجی زیر را ایجاد خواهد کرد:

```
Enter Twitter Account:drchuck
Retrieving https://api.twitter.com/1.1/ ...
[{"created_at":"Sat Sep 28 17:30:25 +0000 2013","
id":384007200990982144,"id_str":"384007200990982144",
"text":"RT @fixpert: See how the Dutch handle traffic
intersections: http:\/\/t.co\/tIiVWtEhj4\n#brilliant",
"source":"web","truncated":false,"in_rep
Remaining 178


Enter Twitter Account:fixpert
Retrieving https://api.twitter.com/1.1/ ...
[{"created_at":"Sat Sep 28 18:03:56 +0000 2013",
"id":384015634108919808,"id_str":"384015634108919808",
"text":"3 months after my freak bocce ball accident,
my wedding ring fits again!
:)\n\nhttps:\/\/t.co\/2XmHPx7kgX",
"source":"web","truncated":false,
Remaining 177


Enter Twitter Account:
```

در کنار داده برگرداننده شده‌ی گاه‌شمار، توییتر فراداده‌ی درخواستی را نیز در پاسخ هدرهای HTTP برمی‌گرداند. یک هدرِ به‌خصوص، x-rate-limit-remaining، ما را از تعداد درخواست‌هایی که می‌توانیم قبل از بسته شدن، در یک بازه‌ی زمانی کوتاه بفرستیم مطلع می‌سازد. می‌توانید ببینید که هر بار که درخواستی به API می‌فرستیم یک عدد از درخواست‌های باقیمانده کم می‌شود.

در مثال زیر، اطلاعات دوستانِ یک کاربر توییتر را گرفته و JSON برگرداننده شده را تحلیل می‌کنیم و در نهایت مقداری از اطلاعات در خصوص دوستانش را استخراج می‌کند. همچنین از JSON، بعد از تحلیل نمونه‌برداری کرده و خروجی را با چهار

کاراکتر به سمت داخل پرِتی-پرینت[I] می‌کنیم. به این صورت خروجی ما جلوه‌ی بهتری را خواهد داشت.

[I] Prettyprint عبارت است از تغییراتی – مانند اضافه کردن تورفتگی یا رنگ کردن متن – که در ظاهر کد اعمال می‌کنیم تا خواناتر، زیباتر و قابل فهم‌تر شود.

```
import urllib.request, urllib.parse, urllib.error
import twurl
import json
import ssl


# https://apps.twitter.com/
# Create App and get the four strings, put them in hidden.py


TWITTER_URL =
'https://api.twitter.com/1.1/friends/list.json'


# Ignore SSL certificate errors
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE


while True:
    print('')
    acct = input('Enter Twitter Account:')
    if (len(acct) < 1): break
    url = twurl.augment(TWITTER_URL,
                        {'screen_name': acct, 'count': '5'})
    print('Retrieving', url)
    connection = urllib.request.urlopen(url, context=ctx)
    data = connection.read().decode()


    js = json.loads(data)
    print(json.dumps(js, indent=2))


    headers = dict(connection.getheaders())
    print('Remaining', headers['x-rate-limit-remaining'])


    for u in js['users']:
        print(u['screen_name'])
        if 'status' not in u:
```

```
            print('   * No status found')
            continue
        s = u['status']['text']
        print('  ', s[:50])


# Code: http://www.py4e.com/code3/twitter2.py
```

از آنجایی که JSON تبدیل به دیشکنری‌ها و لیست‌های پایتونیِ تو در تو می‌شود، می‌توانیم از ترکیب عملیات شاخص و حلقه‌های for برای گشت و گذار بین داده‌های برگردانده شده با مقدار کمی از کدهای پایتونی استفاده کنیم.

خروجی برنامه شبیه فیلد زیر خواهد بود (کمی از آیتم‌ها برای جا شدن در صفحه کوتاه شده‌اند):

```
Enter Twitter Account:drchuck
Retrieving https://api.twitter.com/1.1/friends ...
Remaining 14
{
    "next_cursor": 1444171224491980205,
    "users": [
        {
            "id": 662433,
            "followers_count": 28725,
            "status": {
                "text": "@jazzychad I just bought one .__.",
                "created_at": "Fri Sep 20 08:36:34 +0000
2013",
                "retweeted": false,
            },
            "location": "San Francisco, California",
            "screen_name": "leahculver",
            "name": "Leah Culver",
        },
        {
            "id": 40426722,
            "followers_count": 2635,
            "status": {
                "text": "RT @WSJ: Big employers like Google
...",
                "created_at": "Sat Sep 28 19:36:37 +0000
2013",
            },
            "location": "Victoria Canada",
            "screen_name": "_valeriei",
            "name": "Valerie Irvine",
    ],
    "next_cursor_str": "1444171224491980205"
}
leahculver
   @jazzychad I just bought one .__.
_valeriei
   RT @WSJ: Big employers like Google, AT&amp;amp;T are h
ericbollens
   RT @lukew: sneak peek: my LONG take on the good &a
```

```
halherzog
   Learning Objects is 10. We had a cake with the LO,
scweeker
   @DeviceLabDC love it! Now where so I get that "etc


Enter Twitter Account:
```

آخرین قسمت خروجی، پنج دوست اخیر اکانت توییتر دکتر چاک به همراه آخرین وضعیت‌شان است. داده‌های زیاد دیگری در JSON بازگردانده شده وجود دارد. اگر شما به خروجی برنامه نگاهی بیندازید خواهید دید که "find the friends" هر اکانت خاص، نرخِ محدودیت خاص خود را نسبت به استعلام گاه‌شماری که ما مجاز به اجرا در یک زمانِ خاص هستیم، دارد.

کلیدهای امن API به توییتر اجازه می‌دهد تا با اطمینان از اینکه چه کسی از API این کمپانی و داده‌ها، در چه سطحی استفاده می‌کند، اطلاع داشته باشد. رویکرد تحدیدِ نرخ به ما اجازه می‌دهد تا داده‌های شخصی و ساده را دریافت کنیم ولی به ما اجازه‌ی ساخت یک محصول برای کشیدن اطلاعات از API با نرخ میلیون بر روز نمی‌دهد.

## واژگان

**رابط برنامه‌نویسی کاربردی / API:**
رابط برنامه‌نویسی کاربردی، یک قرارداد بین برنامه‌های کاربردی است که الگوهایی را برای تعامل بین اجزاء برنامه‌ها تعیین می‌کند.

**ElementTree:**
یک کتابخانه‌ی توکاری شده‌ی پایتونی برای تحلیل داده‌های XML است.

**نشانه‌گذاری شئ جاوااسکریپت / JSON:**
نشانه‌گذاری شئ جاوااسکریپت، یک استاندارد باز متنیِ سبک برای انتقال داده‌ها است به گونه‌ای که برای انسان نیز خوانا باشد.

**معماری سرویس‌گرا / SOA:**
معماری سرویس‌گرا به حالتی گفته می‌شود که یک برنامه از اجزائی تشکیل شده که بر بستر شبکه به یکدیگر متصل شده‌اند.

**زبان نشانه‌گذاری گسترش‌پذیر / XML:**
زبان نشانه‌گذاری گسترش‌پذیر قالبی برای نمایش داده‌ها در حالت ساختاریافته است.

## تمرین‌ها

تمرین ۱: برنامه www.py4e.com/code3/geojson.py یا www.py4e.com/code3/geoxml.py را به نحوی تغییر دهید که دو کاراکتر کد کشور را از داده‌های دریافت شده بیرون بکشد و چاپ کند. یک سیستم بررسی خطا به برنامه اضافه کنید که اگر کد کشور در دسترس نبود، با تریس‌بک مواجه نشویم. اکنون Atlantic Ocean را جستجو کرده و مطمئن شوید که در خصوص مکان‌هایی که به هیچ کشوری تعلق ندارد نیز درست عمل می‌کند.

# فصل ۱۴

# برنامه‌نویسی شئ‌گرا

## مدیریت برنامه‌های بزرگتر

در ابتدای این کتاب در خصوص چهار الگوی اساسیِ برنامه‌نویسی برای ساخت برنامه‌هایمان صحبت کردیم:

I. کدهای ترتیبی
II. کدهای شرطی (گزاره‌های if)
III. کدهای تکرارشونده (حلقه‌ها)
IV. ذخیره و باز استفاده (توابع)

در فصول بعدی متغیرهای ساده و ساختارهای جمع‌آوری اطلاعات مانند لیست‌ها، تاپل‌ها و دیکشنری‌ها را بررسی کردیم.

با نوشتن برنامه، ما ساختارهای داده را طراحی و کد را برای دستکاری این ساختارها می‌نویسیم. راه‌های زیادی برای نوشتن برنامه وجود دارد و تا اینجای کار شما هم چند برنامه نوشته‌اید، حالا یا قشنگ و مجلسی یا نه‌چندان قشنگ. با وجود اینکه برنامه‌ی شما احتمالا کوچک است، هنر و زیبایی نوشتن کد را تا حدودی به چشم دیده‌اید.

وقتی که برنامه‌ها تبدیل به برنامه‌های میلیون خطی می‌شوند، نوشتنِ کدی که به آسانی قابل فهم باشد پراهمیت‌تر می‌شود. اگر روی یک برنامه‌ی میلیون خطی کار

می‌کنید، نمی‌توانید کل برنامه را در آنِ واحد در ذهن خود داشته باشید. به همین خاطر لازم است که آن را به قطعه‌های کوچک‌تری بشکنیم تا اضافه‌کردن یک ویژگی، برطرف کردن یک باگ یا حل یک مشکل، ساده‌تر شود.

در طرف دیگر، برنامه‌نویسی شئ‌گرا راهی برای سامان دادن به کدهای شما برای تمرکز روی قطعه‌های کوچک‌تر کد و درک آن، بدون توجه به سایر بخش‌هاست. مثلا فرض کنید که یک برنامه‌ی ۱,۰۰۰,۰۰۰ خطی دارید؛ برنامه‌نویسی شئ‌گرا به شما کمک می‌کند روی ۵۰۰ خط کدی که مشکل آفریده تمرکز کنید و ۹۹۹,۵۰۰ خط دیگر را نادیده بگیرید.

## بزن بریم

درست به‌مانند بسیاری از جنبه‌های دیگر برنامه‌نویسی، برای به‌کارگیری این سبک از برنامه‌نویسی لازم است مفهوم برنامه‌نویسی شئ‌گرا را یاد بگیرید و درک کنید. در این فصل ما به شما چند اصطلاح و مفهوم مرتبط را آموزش می‌دهیم و با مثال‌های اندک، یک زیرساخت برای درک بهتر شما از برنامه‌نویسی شئ‌گرا بنا می‌کنیم.

در ادامه‌ی این کتاب از آبجکت‌ها در بسیاری از برنامه‌ها استفاده خواهیم کرد، ولی آبجکت‌های جدید را در داخل برنامه‌هایمان نمی‌سازیم.

سرانجام در این فصل به درک پایه‌ای از چگونگی ساخت آبجکت‌ها و عملکرد آن‌ها خواهید رسید. همچنین یاد خواهید گرفت که چطور از قابلیت‌های آبجکت‌هایی که توسط پایتون و کتابخانه‌های پایتونی برای شما فراهم شده استفاده کنید.

## استفاده از آبجکت‌ها

به نظر می‌آید که ما در طول این کتاب از آبجکت‌ها استفاده کرده‌ایم. پایتون آبجکت‌های توکاری شده‌ی فراوانی را فراهم کرده است. در اینجا کد ساده‌ای را می‌بینید. چند خط اول بایستی برای شما خیلی ساده و قابل درک باشد.

```
stuff = list()
stuff.append('python')
stuff.append('chuck')
stuff.sort()
print (stuff[0])


print (stuff.__getitem__(0))
print (list.__getitem__(stuff,0))


# Code: http://www.py4e.com/code3/party1.py
```

ولی به جای تمرکز روی کارهایی که این خطوط انجام می‌دهند، بیایید ببینیم که از منظر برنامه‌نویسی شئ‌گرا، زیر پوست این برنامه چه خبر است. اگر منظور من از پاراگراف‌های بعدی را نمی‌فهمید، نگران نباشید چرا که هنوز که هنوز است بسیاری از اصطلاحات مرتبط با آن برای‌تان قابل هضم نشده است.

خط اول یک آبجکت با نوع «لیست» را «می‌سازد». خط دوم و سوم متد append() را فراخوانی کرده و در خط چهارم متد sort() احضار می‌شود. خط پنجم مقدارِ آیتم را در مکان 0 دریافت و چاپ می‌کند.

در خط ششم متد __getitem__() در لیست stuff با یک پارامتر «صفر» فراخوانی می‌شود.

```
print (stuff.__getitem__(0))
```

خط هفتم گویاتر است و صفریم آیتم موجود در لیست دریافت و چاپ می‌شود.

```
print (list.__getitem__(stuff,0))
```

در این کد ما متد __getitem__ را در کلاس list فراخوانی کرده و به لیست (stuff) فرستادیم. سپس آیتمی که می‌خواستیم را به عنوان پارامتر از لیست دریافت کردیم.

سه خطِ آخر برنامه کاملا با هم یکسانند ولی استفاده از قلاب برای دریافت آیتم راحت‌تر است.

با استفاده از خروجیِ تابع dir() می‌توانیم قابلیت‌های یک آبجکت را نظاره 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']
>>>
```

در اصل dir() لیستی از متدها و صفات یک آبجکت پایتونی است.

در ادامه‌ی این فصل به موشکافی این اصطلاحات می‌پردازیم. بد نیست در انتهای فصل به مرورِ دوباره‌ی آنچه تا اینجای فصل گفته شد بپردازید.

## شروع با برنامه‌ها

در پایه‌ای‌ترین حالت، یک برنامه ورودی می‌گیرد، پردازش انجام می‌دهد و خروجی تولید می‌کند. برنامه‌ی «تبدیل طبقه» یک برنامه‌ی کوتاه ولی کامل را در سه مرحله به ما نشان می‌دهد:

```
usf = input('Enter the US Floor Number: ')
wf = int(usf) - 1
print('Non-US Floor Number is',wf)


# Code: http://www.py4e.com/code3/elev.py
```

اگر کمی بیشتر در خصوص این برنامه فکر کنید، یک «دنیای خارج» و یک «برنامه» داریم. ورودی و خروجی جایی است که برنامه با دنیای خارج در تماس و تعامل است. در درون برنامه، کد و داده‌هایی خواهیم داشت که برنامه به کمک آن‌ها وظیفه‌ای را انجام می‌دهد که به آن محول شده است.


زمانی‌که «داخل» یک برنامه‌ایم تعاملات تعیین‌شده‌ای با دنیای «خارج» خواهیم داشت، اما این تعاملات به خوبی برنامه‌ریزی شده‌اند و نیازی به تمرکز روی آن‌ها نیست. در طرف مقابل، زمانی‌که در حال نوشتن کد هستیم، عکس این مطلب صدق می‌کند و ما به جزئیات «درون برنامه» دقت می‌کنیم.

برای نگاهِ بهتر به برنامه‌نویسی شیءگرا آن را به صورت «نقاطی» در نظر بگیرید که برنامه‌ی ما را تجزیه و جدا می‌کند. هر «نقطه» حاوی مقداری کد و داده می‌شود (درست شبیه به یک برنامه) و برای تعامل با دنیای بیرون و سایر «نقاط» به خوبی آماده شده است.

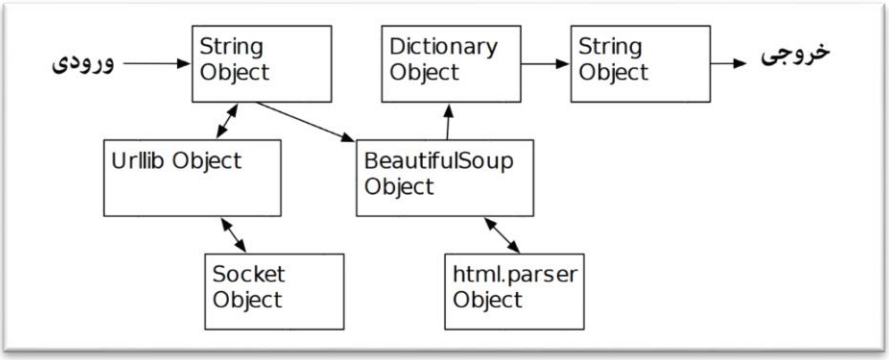
اگر نگاهی به برنامه‌ای که پیوندها را خارج می‌کرد بیندازید ما از کتابخانه‌ی BeautifulSoup استفاده کردیم. به عبارتی برنامه‌ای که ما ساختیم با اتصالِ آبجکت‌های مختلف برای انجام یک وظیفه بنا شده بود.

```
# To run this, you can install BeautifulSoup
# https://pypi.python.org/pypi/beautifulsoup4


# Or download the 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: http://www.py4e.com/code3/urllinks.py
```

ما URL را از یک رشته خوانده و به urllib می‌فرستیم تا داده‌ها را از وب دریافت کند. کتابخانه urllib با استفاده از کتابخانه socket یک ارتباط واقعی برای دریافت داده‌ها

برقرار می‌کند. ما رشته‌ای که urllib تحویل می‌دهد را می‌گیریم و به BeautifulSoup برای تحلیل می‌فرستیم.

BeautifulSoup از آبجکتی به اسم html.parser استفاده کرده و یک آبجکت را برمی‌گرداند. سپس متد tags() را، در آبجکتِ برگردانده شده، فراخوانی کرده و یک دیشکنری از آبجکت‌های تگ تحویل می‌دهد. در ادامه‌ی برنامه ما با پیمایش در بین تگ‌ها و احضار متد get() برای هر تگ، صفت href را چاپ می‌کنیم.



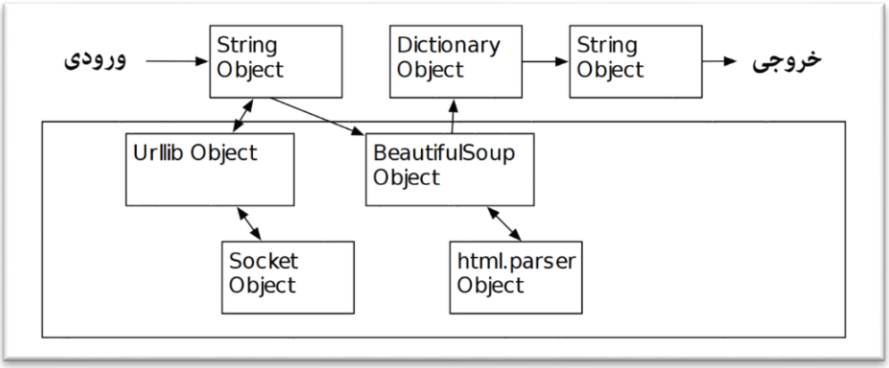

می‌توانیم طرحی از این برنامه بکشیم تا مشخص شود آبجکت‌ها چگونه با یکدیگر کار می‌کنند.

کلید اصلی، فهمیدن کامل این برنامه‌ها و نحوه‌ی کارکردشان نیست، بلکه ساخت یک شبکه از تعاملات بین آبجکت‌ها و تنظیم جریان اطلاعات بین آن‌ها برای ساخت یک برنامه است.

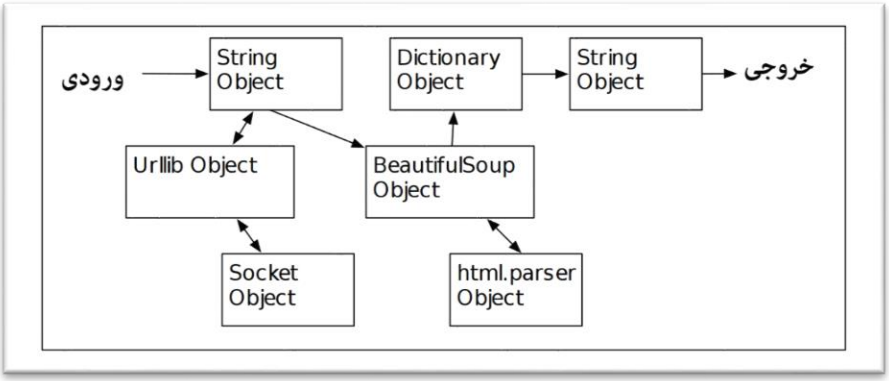
شما با نمونه‌ی این برنامه در چند فصل پیش هم آشنا شدید. در آنجا می‌فهمیدید که برنامه چه کاری را انجام می‌دهد؛ بدون اینکه نیازی به درک هماهنگی حرکات داده‌ها بین آبجکت‌ها داشته باشید. در آنجا خطوط کد را می‌دیدید که کارشان را انجام می‌دهند.

## تقسیم یک مشکل - کپسول کردن

یک مزیت بزرگ برنامه‌نویسی شئ‌گرا توانایی مخفی کردن پیچیدگی‌هاست. به عنوان مثال تنها لازم است کار با urllib و BeautifulSoup را بدانید، بدون اینکه از درون این کتابخانه‌ها و کدهایشان سر در بیاورید. این کار به ما اجازه می‌دهد تا با تمرکز روی قسمتی از مشکل که برای عملکرد صحیح برنامه لازم است، حواسمان پرتِ سایر قسمت‌ها نشود.




قابلیت تمرکز روی یک قسمت از برنامه که ما به آن (در آن زمان) اهمیت می‌دهیم و نادیده گرفتن سایر قسمت‌ها به توسعه‌دهندگان آبجکت‌ها نیز کمک می‌کند. به عنوان نمونه، برنامه‌نویسانی که روی توسعه BeautifulSoup کار می‌کنند نه می‌دانند و نه اهمیت می‌دهند که ما چطور صفحه HTML را می‌گیریم، چه قسمت‌هایی را می‌خواهیم بخوانیم و قرار است با اطلاعاتی که از صفحه‌ی وب استخراج شده چه دردی را درمان کنیم.




عبارت دیگری که به این مفهوم اشاره دارد «کپسول کردن» است. کپسول کردن به این معنی است که ما می‌دانیم چطور از یک آبجکت استفاده کنیم بدون اینکه بدانیم وظایف در درون آن چطور کامل می‌شوند.

## اولین آبجکت پایتونی ما

تعریف آبجکت در ساده‌ترین حالت عبارت است از: مقداری کد به‌علاوه‌ی ساختارهای داده که کوچکتر از یک برنامه‌ی کامل است. تعریف کردن یک تابع به ما اجازه می‌داد تا بسته‌ای با نام مشخص درست کنیم، سپس مقداری کد درون آن قرار دهیم و هر زمان که لازم بود با فراخوانی نام آن بسته، تابع را احضار کنیم.

یک آبجکت می‌تواند شامل داده و تعدادی تابع شود (که به آن‌ها متد می‌گوییم). داده‌ها توسط تابع‌های داخل آبجکت مورد استفاده قرار می‌گیرند. ما به آیتم‌های مرتبط با داده‌ها در آبجکت صفات یا attributes می‌گوییم. صفات، در برنامه‌نویسی به مشخصه‌هایی که به تعریف خواص اشیاء می‌پردازد گفته می‌شود.

با استفاده از کلیدواژه class داده‌ها و کدی را، که هر کدام آبجکت‌ها را می‌سازند، تعریف می‌کنیم. این کلیدواژه تشکیل شده است از نام کلاس/class و شروع یک قطعه یا بلاک از کد که تورفته است و شامل صفات (داده‌ها) و متدها (کد) می‌شود.

```
class PartyAnimal:
   x = 0


   def party(self) :
     self.x = self.x + 1
     print("So far",self.x)


an = PartyAnimal()
an.party()
an.party()
an.party()
PartyAnimal.party(an)


# Code: http://www.py4e.com/code3/party2.py
```

هر متد شبیه به یک تابع است و با کلیدواژه def آغاز می‌شود و در نهایت شامل قطعه‌ای تورفته از کدهاست. در این مثال ما یک صفت (x) و یک متد (party) داشتیم. متدها یک پارامتر اولیه‌ی ویژه دارند که طبق عرف آن را self می‌نامیم.

شبیه به کلیدواژه def که باعث نمی‌شد کد موجود در تابع اجرا شود، کلیدواژه class یک آبجکت را نمی‌سازد. در عوض کلیدواژه class یک قالبی درست می‌کند تا نشان دهد چه داده‌ها و کدی در هر آبجکت با نوع PartyAnimal قرار خواهد گرفت. class شبیه به یک دستگاه بُرش کلوچه است و آبجکت‌هایی که توسط این دستگاه ساخته می‌شوند، خود کلوچه‌ها خواهند بود. شما روی دستگاه برش کلوچه، خامه نمی‌گذارید، بلکه خامه‌ها را روی خود کلوچه‌ها قرار می‌دهید – همینطور می‌توانید خامه‌های متفاوتی روی کلوچه‌های متفاوت قرار دهید. برای درک یک کلاس و دو آبجکت، تصویر زیر را برانداز کنید.

خب به ادامه‌ی کد برگردیم. اولین خط قابل اجرا در کد ما، خط زیر بود:

```
an = PartyAnimal()
```

اینجا دقیقا همان قسمت است که ما به پایتون می‌گوییم یک آبجکت یا نمونه‌ای از کلاس را با نام PartyAnimal بسازد. شبیه به فراخوانی یک تابع، کلاس اجرا شده و با داده‌ها و متدهای مناسب، یک آبجکت را برمی‌گرداند. این آبجکت سپس به متغیر an تخصیص داده می‌شود. این کار شبیه به خط زیر است که در طول کتاب زیاد مورد استفاده قرار دادیم:

```
counts = dict()
```

در خط بالا ما به پایتون اعلام کردیم که با استفاده از قالب dict که در پایتون از پیش موجود است، یک آبجکت بسازد و نمونه‌ی دیشکنری را برگرداننده و به متغیر counts نسبت دهد (گماشتن).

در اینجا از کلاس PartyAnimal برای ساخت یک آبجکت و از متغیر an برای ارجاع به آن آبجکت استفاده می‌کنیم. با استفاده از an به کد و داده‌های آن نمونه‌ی خاص از آبجکت PartyAnimal دسترسی خواهیم داشت.

هر آبجکت/نمونه از PartyAnimal در دل خود یک متغیر x و یک متد/تابع با نام party دارد. متد party را با خط زیر می‌توان احضار کرد:

```
an.party()
```

زمانی که متد بالا فراخوانی شد، اولین پارامتر (که ما بنا به رسم و رسوم self نامیدیم) به آن نمونه‌ی خاص از آبجکت PartyAnimal که متد party از آن فراخوانی شده ارجاع پیدا می‌کند (در مثال بالا آبجکت ما an نام دارد). در داخل متد party ما این خط را می‌بینیم:

```
self.x = self.x + 1
```

با استفاده از عملگر نقطه (دات) ما می‌گوییم: ایکسی که در داخل self قرار دارد را می‌خواهیم. به این صورت، هر بار که party() احضار شود، مقدار x داخلی یک واحد افزایش یافته و چاپ می‌شود.

برای درک بهتر تفاوت یک تابع کلی و یک متد در داخل یک کلاس/آبجکت، خط زیر را ببینید. این خط نحوه‌ی دیگر از فراخوانی متد party در داخل آبجکت an را نشان می‌دهد:

```
PartyAnimal.party(an)
```

در این نمونه، ما به کد از داخل class دسترسی پیدا کرده و اشاره‌گرِ (pointer) آبجکتِ an را به عنوان اولین پارامتر (که برابر با self در داخل متد است) می‌فرستیم. می‌توانید به an.party() به عنوان نمونه‌ی کوچک‌شده‌ی خط بالا نگاه کنید.

زمانی که برنامه اجرا می‌شود، خطوط زیر را تولید خواهد کرد:

```
So far 1
So far 2
So far 3
So far 4
```

آبجکتْ ساخته شده و متد party چهار مرتبه فراخوانی می‌شود؛ هر بار مقدار x داخل آبجکتِ an یک واحد افزایش پیدا می‌کند.

## کلاس‌ها(Classes) به عنوان انواع (Types)

همانگونه که در پایتون دیدیم، تمام متغیرها نوع دارند؛ و ما با استفاده از تابع توکاری شده‌ی dir می‌توانیم قابلیت‌های هر متغیر را مشاهده کنیم. به صورت مشابه از type و dir روی کلاس‌هایی که خود ساخته‌ایم می‌توانیم بهره ببریم.

```
class PartyAnimal:
   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: http://www.py4e.com/code3/party3.py
```

برنامه که اجرا شود، خروجی زیر را تولید خواهد کرد:

```
Type <class '__main__.PartyAnimal'>
Dir  ['__class__', '__delattr__', ...
'__sizeof__', '__str__', '__subclasshook__',
'__weakref__', 'party', 'x']
Type <class 'int'>
Type <class 'method'>
```

با استفاده از کلیدواژه class ما یک نوع جدیدی ساختیم. از طریق خروجی dir می‌توان فهمید که هم صفت عدد صحیح x و هم متد party در آبجکت در دسترس است.

## چرخه‌ی حیات آبجکت

در مثال قبلی ما یک کلاس (قالب یا تِمْپِلِیْتْ) تعریف کردیم و با استفاده از آن کلاس یک نمونه از کلاس (آبجکت) ساختیم؛ سپس از آن نمونه استفاده کردیم. زمانی که کارِ برنامه تمام می‌شود تمام متغیرها مرخص می‌شوند. معمولا لازم نیست که نگران ساخت و نابود کردن متغیرها باشید اما همینکه آبجکت‌های ما پیچیده‌تر می‌شود لازم است کارهایی را در درون آن‌ها انجام دهیم؛ به این ترتیب که شرایط را برای ساخت مهیا کنیم و از طرف دیگر در حین نابود کردن آن‌ها تمیزکاری انجام دهیم.

متدها با نام‌های خاصی وجود دارند که می‌توانید با اضافه‌کردنشان، آبجکت را در جریانِ ساخت و نابود کردن آن‌ها قرار می‌دهید.

```
class PartyAnimal:
   x = 0


   def __init__(self):
     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: http://www.py4e.com/code3/party4.py
```

زمانی که 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
```

همینطور که پایتون آبجکت ما را می‌سازد متد __initi__ را فراخوانی کرده و به ما اجازه می‌دهد تا مقدار پیش‌فرض و اولیه را برای آبجکت تعیین کنیم. زمانی که پایتون با خط زیر مواجه می‌شود:

```
an = 42
```

در این زمان پایتون آبجکت را بیرون انداخته و با استفاده مجدد از متغیر an مقدار 42 را در آن ذخیره می‌کند. درست در زمانی که آبجکت an در حال نابود شدن است، کد نابودکننده (__del__) فراخوانی می‌شود. ما نمی‌توانیم از نابود شدن متغیرمان جلوگیری کنیم ولی می‌توانیم پاک‌سازی‌های لازم را درست قبل از نابود شدن آبجکت انجام دهیم.

زمانی‌که یک آبجکت را می‌سازید و توسعه می‌دهید، اضافه کردن یک سازنده (constructor) برای تعیین مقادیر اولیه در آبجکت بسیار معمول است ولی اضافه کردن یک نابودگر (destructor) یه مورد نادر است که کمتر با آن مواجه می‌شوید.

## چندین و چند نمونه

تا اینجا یک کلاس را تعریف کردیم و تنها یک آبجکت بر اساس آن ساخته‌ایم. سپس از آن آبجکت استفاده کرده و در نهایت آن را دور انداخته‌ایم؛ اما قدرت واقعی شئ‌گرایی زمانی مشخص می‌شود که نمونه‌های زیادی از کلاسمان بسازیم.

زمانی که در حال ساخت چندین آبجکت هستیم، ممکن است مقادیر اولیه‌ی خاصی برای هر کدام در نظر داشته باشیم. برای این منظور می‌توانیم داده‌ها را به «سازنده» بفرستیم تا به هر آبجکت مقدار اولیه‌ی متفاوتی دهد.

```
class PartyAnimal:
   x = 0
   name = ''
   def __init__(self, nam):
     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: http://www.py4e.com/code3/party5.py
```

سازنده هم پارامتر self را در خود دارد که به نمونه‌ی آبجکت اشاره دارد، هم یک پارامتر اضافی که به محض ساخته شدن آبجکت، داده‌هایش به سمت سازنده گسیل می‌شود:

```
s = PartyAnimal('Sally')
```

در داخل سازنده، این خط کد را داریم:

```
self.name = nam
```

رونوشتی از پارامتر nam به صفت name در نمونه‌ی آبجکت فرستاده می‌شود.

خروجی برنامه نشان می‌دهد که هر آبجکت (s و j) کپی‌های جداگانه‌ای از x و nam را در خود دارند:

```
Sally constructed
Sally party count 1
Jim constructed
Jim party count 1
Sally party count 2
```

## ارث‌بری

یکی دیگر از قابلیت‌های قدرمتند برنامه‌نویسی شئ‌گرا توانایی ساخت یک کلاس با گسترش یک کلاسِ موجود است. زمانی که یک کلاس را گسترش می‌دهیم، کلاس اصلی، «کلاس والد» و کلاس جدید، «کلاس فرزند» خوانده می‌شود.

برای این مثال ما کلاس PartyAnimal را در فایل خودش ذخیره می‌کنیم:

```
class PartyAnimal:
   x = 0
   name = ''
   def __init__(self, nam):
     self.name = nam
     print(self.name,'constructed')


   def party(self) :
     self.x = self.x + 1
     print(self.name,'party count',self.x)


# Code: http://www.py4e.com/code3/party.py
```

سپس می‌توانیم در مسیرِ فایلِ ذخیره شده، آن را «درون‌ریزی» یا ایمپورت کرده و سپس گسترش دهیم (فایل را با نام party.py ذخیره کنید تا خط اول کدِ بعدی برای برنامه معنی‌دار باشد و بتواند فایل را برای درون‌ریزی کلاسِ موجود پیدا کند):

```
from party import PartyAnimal


class CricketFan(PartyAnimal):
   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: http://www.py4e.com/code3/party6.py
```

در این مثال درست زمانی که آبجکت CricketFan را می‌سازیم، مشخص می‌کنیم که در حال گسترش کلاس PartyAnimal هستیم. این بدان معناست که تمام متغیرها (x) و متدها (party) از کلاس PartyAnimal به کلاس CricketFan به ارث خواهند رسید.

همانگونه که می‌بینید قادر خواهیم بود که در داخل متد six در کلاس CricketFan متد party را از کلاس PartyAnimal احضار کنیم. متغیرها و متدهای کلاسِ والد در کلاس فرزند ادغام شده است.

با اجرای برنامه می‌توانید ببینید که s و j نمونه‌های مستقل از کلاس‌های PartyAnimal و CricketFan هستند. آبجکت j قابلیت‌های اضافه‌تری نسبت به آبجکت s در خود 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 dir برای آبجکت j، که نمونه‌ای از کلاس CricketFan است، می‌بینید که متدها و صفات کلاس والد در کنار متدها و صفاتی که در کلاس CricketFan وجود دارد، قرار گرفته‌اند. به عبارتی کلاس والد گسترش یافته تا CricketFan را تشکیل دهد.

## جمع‌بندی

این فصل درآمدی کوتاه از برنامه‌نویسی شئ‌گرا بود که بیشتر روی اصطلاحات و متن یا سینتکس تعریف و استفاده از آبجکت‌ها تمرکز داشت. اکنون با هم نگاهی گذرا به کدی که در ابتدای فصل دیدیم، بیندازیم. بایستی متوجه اینکه در پس پرده‌ی این کد چه می‌گذرد شوید.

```
stuff = list()
stuff.append('python')
stuff.append('chuck')
stuff.sort()
print (stuff[0])


print (stuff.__getitem__(0))
print (list.__getitem__(stuff,0))


# Code: http://www.py4e.com/code3/party1.py
```

خط اول، یک آبجکت list را می‌سازد. زمانی که پایتون یک آبجکت list را ساخت متد سازنده (با نام __init__) را برای تنظیم صفات داخلی که به منظور ذخیره‌ی لیستی از داده‌ها مورد استفاده قرار می‌گیرد، احضار می‌کند. با توجه به کپسول کردن، نیازی به دانستن چگونگی آن وجود ندارد. در حقیقت لازم نیست اهمیتی بدهید که در صفات داخلی آن چه اتفاقی می‌افتد و چگونه تنظیم می‌شوند.

ما هیچ پارامتری را به سازنده ارسال نمی‌کنیم. با استفاده از متغیر stuff ما به چیزی که سازنده برمی‌گرداند اشاره می‌کنیم. در حقیقت stuff یک اشاره‌گر به آبجکتی است که سازنده، می‌سازد و برمی‌گرداند. به عبارت دیگر متغیر stuff نمونه‌ای از کلاس list است.

خط دوم و سوم، متد append را با یک پارامتر احضار می‌کند. این متد برای اضافه کردن یک آیتمِ جدید به انتهای لیست با به‌روز رسانی صفات موجود در stuff به کار گرفته می‌شود. در خط چهارم ما متد sort را بدون هیچ پارامتری احضار می‌کنیم. این متد داده‌های موجود در آبجکت stuff را سر و سامان می‌دهد.

سپس اولین آیتم موجود در لیست را با استفاده از قلاب چاپ می‌کنیم. این کار میانبری از متد __getitem__ در آبجکت stuff به حساب می‌آید؛ و برابر با خط بعدی است که در آن دو پارامتر را به متد __getitem__ ارسال می‌کنیم؛ ارسال آبجکت stuff به آن به عنوان اولین پارامتر و نقطه‌ی قرارگیری آن به عنوان دومین پارامتر به این متد ارسال می‌شود.

در انتهای برنامه، آبجکتْ stuff را مرخص می‌کند. این کار قبل از احضار نابودکننده (با نام __del__) صورت نمی‌پذیرد. به همین خاطر آبجکت فرصت دارد تا کمی کثیف‌کاری‌ها را تمیز کند.

این‌ها پایه‌ی اصطلاحات برنامه‌نویسی شئ‌گرا است. جزئیات زیاد دیگری برای استفاده‌ی بهتر از روش شئ‌گرایی در برنامه‌ها و کتابخانه‌های بزرگ وجود دارد که از حوصله‌ی این فصل خارج است.

## واژگان فصل

**صفت / Attribute:**
متغیری که جزئی از یک کلاس است.

**کلاس / class:**
قالبی که برای ساخت یک آبجکت مورد استفاده قرار می‌گیرد و صفات و متدهایی که آبجکت را می‌سازد تعریف کرده و در دل خود دارد.

**کلاس فرزند / child class:**
کلاس جدیدی که با گسترش کلاس والد ساخته می‌شود. کلاس فرزند تمام صفات و متغیرهای کلاس والد را با خود دارد (ارث‌بری).

**سازنده / Constructor:**
یک متد اختیاری با نام __init__ که درست در لحظه‌ی ابتدایی استفاده از کلاس، احضار می‌شود و یک آبجکت را می‌سازد. معمولا برای تعیین مقادیر اولیه‌ی آبجکت مورد استفاده قرار می‌گیرد.

**نابود کننده / Destructor:**
یک متد اختیاری با نام __del__ که درست در لحظه‌ی پایان و نابودی آبجکت احضار می‌شود. نابود کننده‌ها به ندرت مورد استفاده قرار می‌گیرند.

**ارث‌بری / Inheritance:**
زمانی که یک کلاس جدید (فرزند) با گسترش یک کلاس موجود (والد) می‌سازیم، تمام صفات و متدهای کلاس والد به فرزند به ارث می‌رسد. در حقیقت فرزند تمام صفات و متدهای کلاس والد به علاوه‌ی داده‌های خود را در دل دارد.

**متد / Method:**
یک تابع که در دل یک کلاس و آبجکت‌هایی که از آن کلاس ساخته می‌شوند، قرار دارد. برخی الگوهای شئ‌گرایی از «پیام» یا «message» به جای «متد» برای تعریف این مفهوم استفاده می‌کنند.

**آبجکت / Object:**
آبجکت، یک نمونه‌ی ساخته شده از یک کلاس است. یک آبجکت شامل تمام صفات و متدهایی که توسط کلاس تعریف شده، می‌شود. برخی از اسناد شئ‌گرایی از «نمونه / instance» به جای آبجکت استفاده می‌کنند.

**کلاس والد / parent class:**
کلاسی که یک کلاس دیگر (کلاس فرزند) با گسترش آن ساخته شده است. کلاس والد تمام متدها و صفات را به کلاس فرزند به ارث می‌دهد.

# فصل ۱۵

# استفاده از پایگاه‌های داده و SQL

## پایگاه داده[I] چیست؟

یک پایگاه داده (database) فایلی برای ذخیره‌سازی مرتب داده‌ها است. اکثر پایگاه‌های داده شبیه به یک دیکشنری و شیوه‌ی کلیدها و مقادیر آن ساخته می‌شوند با این تفاوت که اطلاعات آن روی دیسک سخت (یا هر محل ذخیره‌سازی دائمی دیگر) ذخیره می‌شود؛ به همین خاطر، با خاتمه‌ی برنامه، پایگاه داده باز هم در جای خود باقی می‌ماند. با توجه به محدودیت کمتر فضای دیسک سخت نسبت به حافظه‌ی اصلی، داده‌های بیشتری در مقایسه با دیکشنری در پایگاه داده جا می‌گیرد.

به‌مانند یک دیشکنری، نرم‌افزار پایگاه داده برای وارد کردن و دسترسی بسیار سریع به داده‌ها طراحی شده است؛ حتی زمانی‌که با داده‌های بسیار زیادی سر و کار داریم. نرم‌افزار پایگاه داده بازدهی خود را با ساخت شاخص‌هایی، به ازای داده‌های اضافه شده به پایگاه داده، حفظ می‌کند؛ به این صورت به کامپیوتر اجازه می‌دهد که به سرعت به سراغ یک مدخل خاص رفته و آن را بیرون بکشد.

سیستم‌های پایگاه داده فراوانی که برای هدف‌های گوناگونی استفاده می‌شوند وجود دارد که در این بین می‌توان به: Oracle و MySQL و Microsoft SQL Server و PostgreSQL و SQLite اشاره کرد. در این کتاب، ما روی SQLite تمرکز می‌کنیم

---

[I] کلمه‌ی data جمع است با این حال در فارسی هم «داده» و هم «داده‌ها» ترجمه می‌شود.

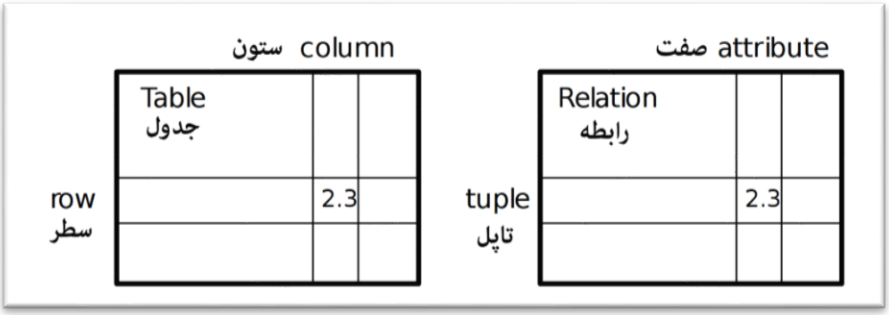
چرا که یک پایگاه داده‌ی بسیار رایج است و در پایتون وجود دارد. SQLite طوری طراحی شده که در داخل برنامه‌های کاربردی دیگر قرار داده شود و پشتیبانی از پایگاه داده را از درون برنامه فراهم کند. به عنوان مثال مرورگر فایرفاکس، به‌مانند بسیاری دیگر از برنامه‌ها، از پایگاه داده SQLite به صورت درونی استفاده می‌کند.

http://sqlite.org

سیستم پایگاه داده SQLite برای مشکلات مرتبط با دستکاریِ داده‌ها بسیار مناسب است؛ مواردی مثل برنامه‌ی عنکبوت توییتر[I] که در این فصل به تشریح آن می‌پردازیم.

## مفاهیم پایگاه داده

زمانی‌که برای اولین بار به یک پایگاه داده نگاه می‌کنید، آن را شبیه یک صفحه‌گسترده با صفحات متعدد خواهید دید. ساختار اصلی داده‌ها در یک پایگاه داده: جدول‌ها، سطرها و ستون‌ها هستند.




اگر بخواهیم عناصر پایگاه‌های داده نسبت‌مند / relational را تشریح کنیم، مفاهیم جدول، سطر و ستون به‌ترتیب به نسبت/relation و تاپل و صفت/attribute اشاره دارد. در این فصل ما از اصطلاح‌های ساده‌ترِ جدول، سطر و ستون استفاده می‌کنیم.

[I] Twitter spidering application

## مرورگر پایگاه داده برای SQLite

با وجود اینکه در این فصل روی پایتون برای کارکردن با فایل‌های پایگاه داده SQLite تمرکز می‌کنیم، شما می‌توانید با استفاده از نرم‌افزار مرورگر پایگاه داده برای SQLite که از طریق پیوند زیر در دسترس است، به انجام عملیات‌های مختلف در دیتابیس بپردازید.

http://sqlitebrowser.org

با استفاده از مرورگر، به راحتی می‌توانید جدول‌ها را بسازید، داده‌ها را وارد یا ویرایش کنید و جستارهای ساده‌ی SQL روی داده‌های دیتابیس انجام دهید. ما به جستارها، کوئری نیز می‌گوییم.

به عبارتی، مرورگر پایگاه داده به مانند یک ویرایش‌گر متن در زمان کار کردن با فایل‌های متنی است. زمانی که شما قرار است عملیات‌های کوچکی روی یک فایل متنی انجام دهید، آن را توسط یک ویرایش‌گر متن باز کرده و تغییرات را اعمال می‌کنید. زمانی که قرار است تغییرات زیادی را انجام دهید، یک برنامه‌ی ساده‌ی پایتونی می‌نویسید و آن را اعمال می‌کنید. الگوهای مشابه در دیتابیس‌ها به چشم می‌خورند و ما در زمان کار کردن با آن‌ها، این الگوها را مشاهده خواهیم کرد. عملیات‌های ساده را در مدیر دیتابیس و عملیات‌های پیچیده را با پایتون انجام خواهید داد.

## ساخت یک جدول دیتابیس

دیتابیس‌ها ساختارهای تعریف‌شده‌ی بیشتری نسبت به لیست‌ها و دیکشنری‌ها نیاز دارند.

زمانی‌که ما یک جدول دیتابیس را می‌سازیم، بایستی قبل از آن چیزهایی را برای دیتابیس مشخص کنیم. این موارد عبارتند از نام هر کدام از ستون‌های جدول و نوع داده‌ای را که قرار است در هر ستون نگه‌داری کنیم. با علم به نوع داده در هر ستون، پایگاه داده بهترین و پربازده‌ترین راه را برای ذخیره و جستجوی داده استفاده می‌کند.

شما می‌توانید به انواع داده‌های پشتیبانی‌شده توسط SQLite در پیوند زیر نگاهی بیندازید:

http://www.sqlite.org/datatypes.html

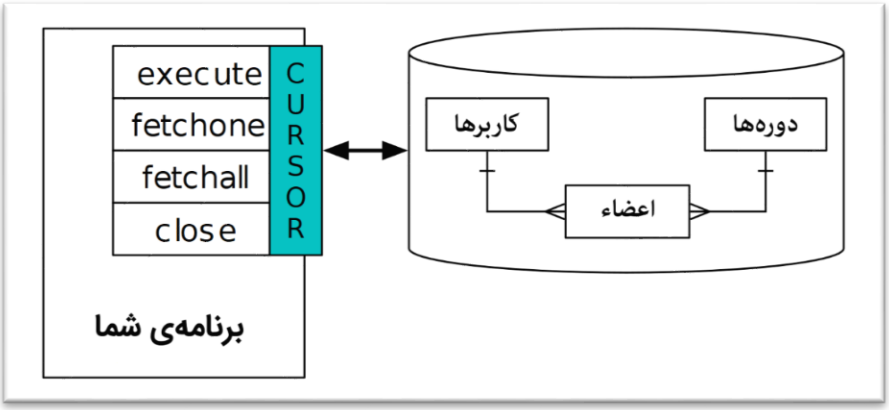
تعریف ساختار برای داده‌هاتان شاید در ابتدا ناخوشایند و سخت به نظر بیاید ولی پاداش آن را با دسترسی سریع به داده‌هایتان خواهید گرفت؛ حتی زمانی‌که دیتابیس حاوی مقادیر زیادی از اطلاعات می‌شود.

کدی که عملیات ساخت یک فایل دیتابیس و یک جدول با نام Tracks با دو ستون را اجرا می‌کند به شرح زیر است:

```
import sqlite3

conn = sqlite3.connect('music.sqlite')
cur = conn.cursor()

cur.execute('DROP TABLE IF EXISTS Tracks')
cur.execute('CREATE TABLE Tracks (title TEXT, plays
INTEGER)')

conn.close()

# Code: http://www.py4e.com/code3/db1.py
```

عمل connect یک ارتباط با پایگاه داده‌ی ذخیره شده برقرار می‌کند. این پایگاه داده در فایلی به اسم music.sqlite3 در مسیرِ جاری قرار گرفته است. اگر فایل وجود خارجی نداشته باشد، آن را می‌سازد. دلیلی که آن را یک «ارتباط» می‌خوانیم این است که گاهی دیتابیس روی «سرور دیتابیس» جداگانه‌ای ذخیره شده است؛ و روی سروری که ما برنامه را اجرا می‌کنیم وجود ندارد. در مثالِ ساده‌ی ما، دیتابیس یک فایل محلی در مسیرِ جاری است؛ همان مسیری که کد پایتون قرار دارد و اجرایش می‌کنیم.

نشانه‌گر یا کِرسِر یا cursor شبیه به یک دستگیره‌ی فایل است که با استفاده از آن می‌توانیم عملیات‌ها را روی داده‌های ذخیره شده در دیتابیس انجام دهیم. فراخوانی cursor() بسیار شبیه به فراخوانی open() است.




زمانی که cursor فراخوانی شد، می‌توانیم دستورات را، روی محتوای پایگاه داده با استفاده از متد execute() اجرا کنیم.

دستورات دیتابیس به زبانِ خاص خودشان نوشته می‌شوند. این زبان در بین ارائه‌کنندگان دیتابیس‌های مختلف، استاندارد شده است؛ به این معنی که اغلب آن‌ها از دستورات یکسانی برای اجرای عملیات استفاده می‌کنند. به این صورت با یادگیری یک زبان دیتابیس، بتوانید با سایر سیستم‌های پایگاه داده کار کنید. زبان دیتابیس Structured Query Language یا SQL نامیده می‌شود.

http://en.wikipedia.org/wiki/SQL

در مثال بالا، دو دستور اس‌کیوال را روی دیتابیس خود اجرا کردیم. همانگونه که رسم است، کلمه‌های کلیدی اس‌کیوال با حروف بزرگ نمایش داده شده و قسمت‌هایی از دستور که ما اضافه می‌کنیم (مانند نام‌های جدول و ستون) با حروف کوچک نشان داده می‌شوند.

اولین دستور اس‌کیوال، جدول Tracks را از دیتابیس، در صورت وجود، حذف می‌کند. این الگو به ما اجازه می‌دهد که برنامه‌ی مشابه را برای ساخت جدول Tracks برای دفعات مکرر اجرا کنیم، بدون اینکه باعث بروز خطایی شویم. توجه داشته باشید که دستور DROP TABLE جدول و تمام محتوایتش را از دیتابیس حذف می‌کند؛ به عبارتی راه برگشت یا undo وجود نخواهد داشت.

```
cur.execute('DROP TABLE IF EXISTS Tracks ')
```

دستور دوم یک جدول با نام Tracks با ستونِ متن با نام title و یک ستونِ عدد صحیح با نام plays می‌سازد.

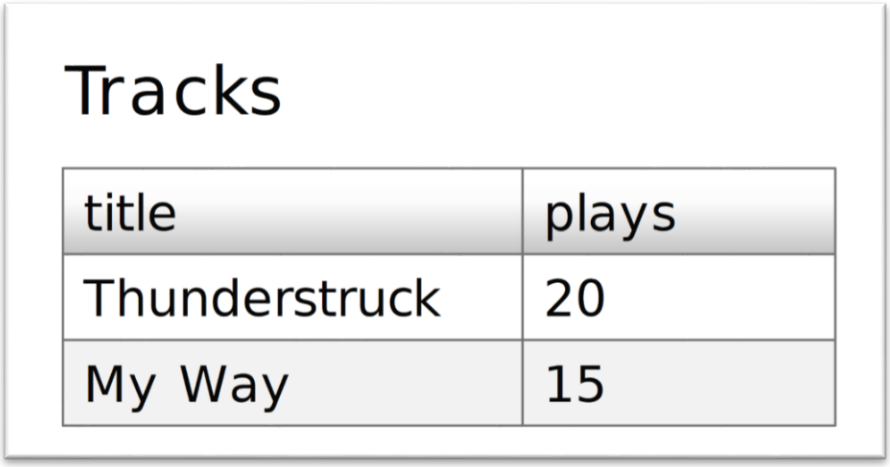
```
cur.execute('CREATE TABLE Tracks (title TEXT, plays
INTEGER)')
```

حالا که جدولی با نام Tracks ساختیم، می‌توانیم داده‌هایی را درون جدول، با استفاده از عملیات INSERT اس‌کیوال، قرار دهیم. دو مرتبه با ساخت یک ارتباط به دیتابیس و دریافت cursor این کار را انجام می‌دهیم. ما می‌توانیم با استفاده از cursor دستورات اس‌کیوال را اجرا کنیم.

دستور INSERT در اس‌کیوال مشخص می‌کند که در حال استفاده از کدام جدول هستیم؛ و سپس سطر جدید را با لیستِ فیلدهایی که ما می‌خواهیم، تعریف می‌کند، در اینجا (title, plays)؛ در ادامه VALUES یا مقادیری که ما می‌خواهیم در سطر جدید قرار بگیرند. مقادیر را با علامت سوال مشخص می‌کنیم (?, ?)، به این صورت نشان می‌دهیم که مقادیر به عنوان یک تاپل و پارامتر دوم به فراخوانی ()execute فرستاده شده‌اند ( 'My Way', 15 ).

```
import sqlite3


conn = sqlite3.connect('music.sqlite')
cur = conn.cursor()


cur.execute('INSERT INTO Tracks (title, plays) VALUES (?,
?)',
    ('Thunderstruck', 20))
cur.execute('INSERT INTO Tracks (title, plays) VALUES (?,
?)',
    ('My Way', 15))
conn.commit()


print('Tracks:')
cur.execute('SELECT title, plays FROM Tracks')
for row in cur:
     print(row)


cur.execute('DELETE FROM Tracks WHERE plays < 100')


cur.close()


# Code: http://www.py4e.com/code3/db2.py
```

ابتدا دو سطر در جدول خود وارد می‌کنیم (INSERT) و با استفاده از ()commit داده‌ها را برای نوشته‌شدن در فایلِ دیتابیس می‌فرستیم.


Tracks

| title | plays |
|---|---|
| Thunderstruck | 20 |
| My Way | 15 |

سپس با استفاده از فرمان SELECT سطرهایی را می‌گیریم که به جدول وارد کرده‌ایم. در فرمان SELECT مشخص می‌کنیم که اطلاعات را از کدام ستونِ (title, plays) از کدام جدول می‌خواهیم. بعد از اجرای گزاره‌ی SELECT، به نشانه‌گر/cursor می‌رسیم. با استفاده از نشانه‌گر، و به کارگیری گزاره‌ی for می‌توانیم در بین داده‌ها پیمایش کرده و حلقه بزنیم. البته برای بازدهی بالاتر، cursor، در حین اجرای SELECT، تمام داده‌ها را از پایگاه داده‌ها نمی‌خواند؛ در عوض داده‌ها در حین حلقه زدن در بین سطور، توسط گزاره‌ی for در وقت نیاز خوانده می‌شوند.

خروجی برنامه، شبیه به این خواهد بود:

```
Tracks:
('Thunderstruck', 20)
('My Way', 15)
```

حلقه‌ی for دو سطر پیدا کرد. هر سطر یک تاپلِ پایتونی است که مقدار اول آن title و مقدار دوم تعداد plays است.

**نکته:** شاید شما، در کتاب‌های دیگر یا روی اینترنت، رشته‌هایی را دیده باشید که با u' شروع می‌شوند. در پایتون ۲، این حالت، نشانه‌ی یونیکد بودن رشته است؛ به

عبارتی رشته‌ی مورد نظر قادر به ذخیره‌ی کاراکترهای غیر لاتین است. در پایتون ۳ به صورت پیش‌فرض تمام رشته‌ها، رشته‌های یونیکد هستند.

در انتهای برنامه، یک دستور اس‌کیوال صادر کردیم تا سطرهایی که ساخته بودیم، حذف شوند؛ در نتیجه می‌توانیم برنامه را بارها و بارها اجرا کنیم. دستور حذف (DELETE) نحوه‌ی استفاده از عبارت WHERE را نشان می‌دهد. عبارت WHERE به ما این امکان را می‌دهد یک معیار برای انتخاب تعیین کنیم؛ به عبارتی با استفاده از WHERE می‌توانیم از دیتابیس بخواهیم که دستور را تنها بر سطوری اعمال کند که با آن معیار هم‌خوانی دارند. در این مثال، معیار، تمام سطرها را شامل می‌شود، در نتیجه ما کلِ جدول را خالی می‌کنیم و می‌توانیم برنامه را دو مرتبه اجرا نماییم. بعد از اینکه DELETE اجرا شد، ما ()commit را فرا می‌خوانیم تا داده‌ها را از جدول حذف کند.

## نگاهی اجمالی بر زبان جستار ساختارمند - SQL

تا اینجا از زبان جستارِ ساختارمند در مثال‌های پایتونی خود استفاده کردیم و دستورات اصولی SQL را فرا گرفتیم. در این بخش به زبان SQL و متن آن خواهیم پرداخت.

از آنجایی که کمپانی‌های مختلفی در رابطه با دیتابیس فعالیت می‌کنند، SQL مطابق با معیارهایی، استاندارد شده است، در نتیجه برقراری ارتباط با سیستم‌های دیتابیس از ارائه‌کنندگان مختلف، روند مشابهی را دارد. پایگاه داده‌ی رابطه‌ای از جدول‌ها، سطور، و ستون‌ها ساخته شده است. ستون‌ها معمولا نوع خود را دارند و می‌توانند متن، عدد، تاریخ و داده‌هایی از این قبیل باشند. زمانی‌که یک جدول می‌سازیم، نام‌ها و نوع ستون‌ها را مشخص می‌کنیم:

```
CREATE TABLE Tracks (title TEXT, plays INTEGER)
```

برای وارد کردن یک سطر به جدولی از دستور INSERT استفاده می‌کنیم:

```
INSERT INTO Tracks (title, plays) VALUES ('My Way', 15)
```

گزاره‌ی INSERT نامِ جدول، لیستی از فیلدها/ستون‌هایی که قرار است در سطر جدید جا بگیرد، کلمه‌ی کلیدی VALUES، و لیستی از مقادیر نظیر به نظیر را مشخص می‌کند.

دستور SELECT برای دریافت سطرها و ستون‌ها از یک دیتابیس است. گزاره‌ی SELECT به شما اجازه‌ی تعیین ستون‌هایی را می‌دهد، که می‌خواهید دریافت کنید. همچنین با بکارگیری یک عبارت WHERE می‌توانید سطرهای مورد نظر را برای این جستار، مشخص کنید. عبارتِ اختیاری ORDER BY می‌تواند خروجی را برایتان با ترتیب خاصی که مد نظر دارید مرتب کند.

```
SELECT * FROM Tracks WHERE title = 'My Way'
```

ستاره (*) در مثال بالا مشخص می‌کند که شما می‌خواهید تمام ستون‌ها از هر سطری که با عبارت WHERE جور در می‌آید برگردانده شود.

توجه کنید که برخلاف پایتون، عبارت WHERE از یک علامت مساوی برای بررسی تساوی، به جای دو علامت مساوی، استفاده می‌کنیم. سایر عملیات‌های مجاز در عبارت WHERE شامل < و > و <= و >= و != همچنین AND و OR و پرانتزها برای ساخت عبارت منطقی می‌شود.

شما می‌توانید از برنامه بخواهید تا سطرهای برگرداننده شده را بر اساس یکی از فیلدها، مرتب کند:

```
SELECT title,plays FROM Tracks ORDER BY title
```

برای حذف یک سطر شما به یک عبارت WHERE در گزاره‌ی DELETE نیاز خواهید داشت. عبارت WHERE تعیین‌کننده‌ی سطوری است که حذف خواهند شد.

```
DELETE FROM Tracks WHERE title = 'My Way'
```

امکان به‌روز رسانی (UPDATE) ستون، یا ستون‌هایی در یک یا چند سطر از یک جدول با استفاده از دستور UPDATE به شکل زیر وجود دارد:

```
UPDATE Tracks SET plays = 16 WHERE title = 'My Way'
```

گزاره‌ی UPDATE شامل این جزئیات است: یک جدول و سپس لیستی از فیلدها و مقادیری که بعد از کلمه‌ی کلیدی SET برای تغییر مشخص می‌شوند، به علاوه‌ی گزینه‌ی اختیاریِ عبارت WHERE برای انتخاب سطرهایی که بایستی به‌روز شوند. یک گزاره‌ی UPDATE به‌تنهایی تمام سطوری که با عبارت WHERE جور در می‌آید را تغییر می‌دهد. اگر عبارت WHERE در کار نباشد، UPDATE روی کلیه‌ی سطرهای جدول اعمال خواهد شد.

این چهار دستور اصلی زبان جستار ساختارمند (INSERT و SELECT و UPDATE و DELETE) عملیات‌های پایه‌ای برای ساخت و نگه‌داری داده‌ها را فراهم می‌کند.

## اسپایدر کردن توییتر با استفاده از یک دیتابیس

در این بخش، یک برنامه‌ی اسپایدرینگ می‌نویسم تا از آن برای پیمایش در اکانت‌های توییتر و ساختن یک دیتابیس از آن‌ها استفاده کنیم. زمان اجرای این برنامه مراقب باشید؛ چرا که بیرون کشیدن اطلاعاتِ بیش از حد برای زمان طولانی می‌تواند منجر به بسته شدن اکانت توییتر شما شود.

یکی از مشکلات برنامه‌های اسپایدرینگ، نیاز به شروع‌مجددِ پیاپی و ذخیره‌ی اطلاعات به صورت مکرر است؛ البته مزیتش این است که احتمال از دست رفتن اطلاعات به حداقل می‌رسد.

قرار نیست که با هر بار شروع مجدد، دریافت اطلاعات از نو آغاز شود، در نتیجه بایستی برنامه به سبکی نوشته شود، که اطلاعات را بعد از شروع مجدد، از جایی که رها کرده بود، دریافت کرده و به جلو برود.

ما با دریافت استاتوس‌های رفقای یک اکانت توییتر شروع می‌کنیم و سپس در بین لیست دوستانْ حلقه می‌زنیم و او را به دیتابیس برای دریافت در آینده اضافه می‌کنیم. وقتی دوستانِ اکانت توییترِ مورد نظر را بررسی و پردازش کردیم، به سراغ دیتابیس رفته و اینبار اطلاعات یکی از دوستانِ دوست را دریافت می‌کنیم. این کار را بارها تکرار می‌کنیم تا دوستانِ نادیده را نیز به لیست آینده‌ی خود اضافه کرده باشیم.

همچنین تعداد دفعاتی که یک دوست در دیتابیس ما قرارگرفته را محاسبه و میزان محبوبیت را بر اساس آن تعیین می‌کنیم.

با ذخیره‌ی لیستی از اکانت‌های آشنا، چه آن اکانت را دریافت کرده باشیم چه نه، و میزان محبوبت اکانت در یک دیتابیس روی دیسک سختِ کامپیوتر، برنامه را متوقف و مجددا استارت می‌زنیم. این کار را به تعداد دفعاتی که مایلیم انجام خواهیم داد.

این برنامه کمی پیچیده است و بر اساس کدی نوشته شده است که در تمرین قبلیِ این کتاب در رابطه با API توییتر دیده‌ایم.

کد برنامه‌ی اسپایدرکردن توییتر به قرار زیر است:

```
from urllib.request import urlopen
import urllib.error
import twurl
import json
import sqlite3
import ssl


TWITTER_URL =
'https://api.twitter.com/1.1/friends/list.json'


conn = sqlite3.connect('spider.sqlite')
cur = conn.cursor()


cur.execute('''
            CREATE TABLE IF NOT EXISTS Twitter
            (name TEXT, retrieved INTEGER, friends
INTEGER)''')


# Ignore SSL certificate errors
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE


while True:
    acct = input('Enter a Twitter account, or quit: ')
    if (acct == 'quit'): break
    if (len(acct) < 1):
        cur.execute('SELECT name FROM Twitter WHERE
retrieved = 0 LIMIT 1')
        try:
            acct = cur.fetchone()[0]
        except:
            print('No unretrieved Twitter accounts found')
            continue
```

```
    url = twurl.augment(TWITTER_URL, {'screen_name': acct,
'count': '5'})
    print('Retrieving', url)
    connection = urlopen(url, context=ctx)
    data = connection.read().decode()
    headers = dict(connection.getheaders())


    print('Remaining', headers['x-rate-limit-remaining'])
    js = json.loads(data)
    # Debugging
    # print json.dumps(js, indent=4)


    cur.execute('UPDATE Twitter SET retrieved=1 WHERE name =
?', (acct, ))


    countnew = 0
    countold = 0
    for u in js['users']:
        friend = u['screen_name']
        print(friend)
        cur.execute('SELECT friends FROM Twitter WHERE name
= ? LIMIT 1',
                    (friend, ))
        try:
            count = cur.fetchone()[0]
            cur.execute('UPDATE Twitter SET friends = ?
WHERE name = ?',
                        (count+1, friend))
            countold = countold + 1
        except:
            cur.execute('''INSERT INTO Twitter (name,
retrieved, friends)
                        VALUES (?, 0, 1)''', (friend, ))
            countnew = countnew + 1
    print('New accounts=', countnew, ' revisited=',
countold)
    conn.commit()
```

```
cur.close()

# Code: http://www.py4e.com/code3/twspider.py
```

دیتابیسِ ما در فایلی با نام spider.sqlite3 ذخیره شده و یک جدول به اسم Twitter دارد. هر سطر از جدول Twitter یک ستون برای نام حسابِ کاربری دارد؛ اعم از اینکه این نام کاربری از دوستان حسابِ کاربری مورد نظر باشد و همچنین تعداد انتخاب شدنش به عنوان دوست.

در حلقه‌ی اصلی این برنامه ما از کاربرْ نامِ یک حساب کاربری توییتر را می‌خواهیم. البته کاربر می‌تواند با وارد کردن "quit" از برنامه خارج شود. اگر کاربر یک حساب کاربری توییتر را وارد کند، ما لیست دوستان را دریافت می‌کنیم، وضعیت هر کدام از کاربران را بررسی می‌کنیم، و هر دوست را، در صورتی‌که قبلا موجود نباشد، به دیتابیس‌مان اضافه می‌کنیم. اگر دوست، در لیست موجود باشد، ما ۱ را به فیلد friends در سطر دیتابیس‌مان اضافه می‌کنیم.

اگر کاربر Enter را فشار دهد، در دیتابیس به دنبال کاربر بعدی توییتر، که هنوز اطلاعاتش را دریافت نکرده‌ایم، می‌گردیم. سپس به جمع‌آوری وضعیت و دوستان آن اکانت می‌پردازیم و آن را به دیتابیس اضافه و به‌روزسانی می‌کنیم و در نهایت شمارنده‌ی friends را افزایش می‌دهیم.

وقتی ما لیست دوستان و وضعیت‌هایشان را دریافت کردیم در بین تمام آیتم‌های user nv JSON برگردانده‌شده حلقه می‌زنیم و screen_name هر کاربر را دریافت می‌کنیم. سپس با استفاده از گزاره SELECT وجود یک screen_name خاص را در دیتابیس بررسی کرده و در صورت وجود، شمار دوستان ثبت شده را دریافت می‌کنیم.

```
countnew = 0
countold = 0
for u in js['users'] :
    friend = u['screen_name']
    print(friend)
    cur.execute('SELECT friends FROM Twitter WHERE name = ?
LIMIT 1',
        (friend, ) )
    try:
        count = cur.fetchone()[0]
        cur.execute('UPDATE Twitter SET friends = ? WHERE
name = ?',
            (count+1, friend) )
        countold = countold + 1
    except:
        cur.execute('''INSERT INTO Twitter (name, retrieved,
friends)
            VALUES ( ?, 0, 1 )''', ( friend, ) )
        countnew = countnew + 1
print('New accounts=',countnew,' revisited=',countold)
conn.commit()
```

به محضی که cursor، گزاره‌ی SELECT را اجرا کند، بایستی سطرها را دریافت کنیم. می‌توانیم با استفاده از گزاره‌ی for این کار را انجام دهیم ولی از آنجایی که قرار است تنها یک سطر را دریافت کنیم (LIMIT 1) می‌توانیم از متد ()fetchone برای دریافت اولین (و تنها) سطری – که نتیجه‌ی عملیات SELECT است – بهره ببریم. از آنجایی که ()fetchone سطر را به عنوان یک تاپل برمی‌گرداند (با وجود اینکه تنها یک فیلد است) ما مقدار اول از تاپل را برای دریافت شمارش دوستِ فعلی و انتقال به متغیر count مورد استفاده قرار می‌دهیم.

اگر دریافت موفقیت‌آمیز بود، با استفاده از گزاره‌ی UPDATE در SQL و با بهره‌گیری از عبارت WHERE، مقدار ۱ را به ستون friends مربوط به سطر مطابق با حساب کاربری دوست اضافه می‌کنیم. توجه کنید که در اینجا دو جای خالی برای ورود (همان علامت‌های سؤال) در SQL داریم و دومین پارامترِ ()execute یک تاپل

با دو المنت است که مقدارهایی را برای جایگزینی در SQL با علامت‌های سوال، در خود دارد.

اگر کد در بلاکِ try شکست بخورد، احتمالا رکوردی که با عبارت WHERE name = ? در عبارتِ SELECT جور در بیاید، وجود ندارد. به همین خاطر است که در بلاک except از گزاره‌ی INSERT برای اضافه کردن screen_name ــ دوست به جدول استفاده کردیم؛ شمارنده‌ی این screen_name در جدول مقدار صفر را به خود می‌گیرد؛ به این معنی که هنوز این screen_name دریافت نشده است.

بنابراین اولین باری که این برنامه اجرا شود و یک اکانت توییتر را وارد کنیم، برنامه به شکل زیر اجرا خواهد شد:

```
Enter a Twitter account, or quit: drchuck
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 20  revisited= 0
Enter a Twitter account, or quit: quit
```

از آنجایی که این نخستین بار است که برنامه اجرا می‌شود، دیتابیس ما خالی است و ما در فایل spider.sqlite3 با اضافه کردن یک جدول به اسم Twitter دیتابیس‌مان را می‌سازیم. سپس برخی از دوستان را گرفته و تمام آن‌ها را به دیتابیس اضافه می‌کنیم (چرا که فعلا دیتابیس خالی است).

در این مرحله شاید بخواهید یک دامپر ساده‌ی دیتابیس برای نگاه در فایل spider.sqlite3 بنویسید:

```
import sqlite3


conn = sqlite3.connect('spider.sqlite')
cur = conn.cursor()
cur.execute('SELECT * FROM Twitter')
count = 0
for row in cur:
    print(row)
    count = count + 1
print(count, 'rows.')
cur.close()


# Code: http://www.py4e.com/code3/twdump.py
```

برنامه به سادگی دیتابیس را باز، و تمام ستون‌ها از تمام سطور در جدول Twitter را انتخاب کرده سپس در بین سطرها حلقه می‌زند و یکی یکی چاپشان می‌کند.

اگر ما این برنامه را بعد از اولین اجرای اسپایدر توییتری که بالاتر نوشتیم، اجرا کنیم، خروجی چیزی شبیه به این خواهد بود:

```
('opencontent', 0, 1)
('lhawthorn', 0, 1)
('steve_coppin', 0, 1)
('davidkocher', 0, 1)
('hrheingold', 0, 1)
...
20 rows.
```

یک سطر برای هر screen_name داریم؛ هنوز داده‌ای برای screen_name دریافت نشده است؛ و هر کس در این دیتابیس فقط یک دوست خواهد داشت.

اکنون دیتابیس ما، دوستانِ اولین اکانت توییتر (drchuck) را نمایش می‌دهد. با اجرای دوباره، برنامه به سراغ دوستان اکانت بعدی که هنوز پردازش نشده می‌رود. تنها کافی‌ست به جای وارد کردن اکانت توییتر، Enter را فشار دهید.

```
Enter a Twitter account, or quit:
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 18  revisited= 2
Enter a Twitter account, or quit:
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 17  revisited= 3
Enter a Twitter account, or quit: quit
```

از آنجایی که ما Enter را فشار دادیم (به عبارتی اکانت توییتری را مشخص نکردیم) کد زیر اجرا خواهد شد:

```
if ( len(acct) < 1 ) :
    cur.execute('SELECT name FROM Twitter WHERE retrieved =
0 LIMIT 1')
    try:
        acct = cur.fetchone()[0]
    except:
        print('No unretrieved twitter accounts found')
        continue
```

از گزاره‌ی SELECT برای دریافت نام اولین کاربری استفاده می‌کنیم که مقدار " have we retrieved this user" آن برابر با صفر است. همچنین با استفاده از الگوی fetchone()[0] در یک بلاک try/except یا یک screen_name از داده‌ها دریافت می‌کنیم یا یک پیغام خطایی گرفته و حلقه را دوباره اجرا خواهیم کرد.

اگر یک screen_name پردازش‌نشده را دریافت کنید، شروع به دریافت داده‌هایش به روش زیر خواهیم کرد:

```
url=twurl.augment(TWITTER_URL,{'screen_name': acct,'count':
'20'})
print('Retrieving', url)
connection = urllib.urlopen(url)
data = connection.read()
js = json.loads(data)


cur.execute('UPDATE Twitter SET retrieved=1 WHERE name =
?',(acct, ))
```

بعد از اینکه داده‌ها را با موفقیت دریافت کردیم، با استفاده از گزاره‌ی UPDATE مقدار موجود در ستون retrieved را به 1 تغییر می‌دهیم تا نشان دهیم دریافت لیست دوستان آن اکانت انجام شده است. این کار از دریافت مجدد داده‌های یکسان برای چندین مرتبه جلوگیری می‌کند و اجازه می‌دهد که در شبکه‌ی دوستان توییتری به جلو حرکت کنیم.

اگر برنامه دوستان را اجرا کنیم و دو مرتبه enter را برای دریافت دوستانِ دوستانی که هنوز مشاهده نشده‌اند فشار دهیم و سپس برنامه‌ی دامپر را اجرا کنیم خروجی زیر را مشاهده خواهیم کرد:

```
('opencontent', 1, 1)
('lhawthorn', 1, 1)
('steve_coppin', 0, 1)
('davidkocher', 0, 1)
('hrheingold', 0, 1)
...
('cnxorg', 0, 2)
('knoop', 0, 1)
('kthanos', 0, 2)
('LectureTools', 0, 1)
...
55 rows.
```

می‌بینید که lhawthorn و opencontent را دیده و ثبت کرده‌ایم. همچنین حساب‌های کاربری cnxorg و kthanos تا اینجا دو دنبال‌کننده‌ی حتمی دارند. از آنجایی که ما دوستان سه شخص (drchuck و opencontent و lhawthorn) را دریافت کردیم، جدول ما ۵۵ سطر، از دوستانی که دریافت شده‌اند، دارد.

هر بار که برنامه را اجرا می‌کنیم و Enter را فشار می‌دهیم، برنامه، اکانت مشاهده‌نشده‌ی بعدی را انتخاب کرده (در اینجا steve_coppin) و دوستانش را دریافت می‌کند و آن‌ها را به عنوان دریافت‌شده نشانه‌گذاری می‌کند و در نهایت هر کدام از دوستانِ steve_coppin را به انتهای دیتابیس اضافه کرده و یا شمارنده‌ی آن‌ها را، اگر در دیتابیس موجود باشند، به‌روز می‌کند.

از آنجایی که داده‌های برنامه روی دیسک سخت و در یک دیتابیس ذخیره می‌شود، فعالیت اسپایدر می‌تواند، به هر تعداد که شما بخواهید، متوقف و دوباره از سر گرفته شود بدون اینکه داده‌ای از دست رود.

## مباحث پایه‌ای در خصوص مدل‌سازی داده

قدرت واقعی یک دیتابیس رابطه‌ای زمانی مشخص می‌شود که ما چندین جدول ساخته و بین آن‌ها پیوندهایی برقرار می‌کنیم. چگونگیِ شکستن داده‌های برنامه‌ی شما به چندین جدول و ساخت رابطه بین جدول‌ها، مدل‌سازی داده (data modeling) خوانده می‌شود. طراحی سندی که جدول‌ها و رابطه‌هایشان را نشان دهد یک «مدل داده» خوانده می‌شود.

مدل‌سازی داده یک مهارت تقریبا پیچیده و پیشرفته است و ما در اینجا مفاهیم پایه‌ای مرتبط با مدل‌سازی داده رابطه‌ای را مورد بحث قرار می‌دهیم. برای جزئیات بیشتر می‌توانید از اینجا شروع کنید:

http://en.wikipedia.org/wiki/Relational_model

مثلا فرض کنید در برنامه‌ی اسپایدر توییترِ ما، به جای شمردن دوستانِ یک شخص، می‌خواهیم یک لیست از تمام رابطه‌های بین افراد داشته باشیم تا بدانیم هرکس توسط چه کسانی دنبال می‌شود.

از آنجایی که هرکسی، تعداد زیادی حساب کاربریِ دنبال‌کننده دارد، با اضافه کردن یک ستون به جدول Twitter کارها پیش نمی‌رود. به همین خاطر یک جدول جدید ساخته و جفتِ دوستان را در آن پِی می‌گیریم. مثال زیر یک راه ساده برای ساختن آن است:

```
CREATE TABLE Pals (from_friend TEXT, to_friend TEXT)
```

هربار که با شخصی که drchuck دنبال می‌کند روبرو می‌شویم، یک سطر با قالب زیر در جدول جا می‌دهیم:

```
INSERT INTO Pals (from_friend,to_friend) VALUES ('drchuck',
'lhawthorn')
```

همین‌طور که بیست دوست از فید توییتر drchuck را بررسی می‌کنیم، بیست رکورد که پارامتر اولشان "drchuck" باشد را در جدول جا می‌دهیم. این موضوع باعث می‌شود نمونه‌های زیادی از رشته را در دیتابیس خود داشته باشیم.

نمونه‌های زیاد و تکراری، از رویه‌ی نرمال‌سازی دیتابیس، تخطی می‌کند و آن را زیر پا می‌گذارد. نرمال‌سازی دیتابیس می‌گوید نبایستی رشته‌ی مشابهی از داده را بیش از یک بار در دیتابیس قرار دهیم. ولی اگر به داده‌ای بیش از یکبار احتیاج پیدا کردیم چه؟ در این صورت بایستی با استفاده از یک کلیدِ عددی برای داده و ارجاع به وسیله‌ی آن کلید، این مساله را حل کنیم.

در حقیقت یک رشته، فضای بسیار بیشتری را روی دیسک و حافظه‌ی کامپیوتر نسبت به یک عدد صحیح اشغال می‌کند و در نهایت زمانِ پردازش بیشتری را خواهد طلبید. اگر تنها چند صد مدخل داشته باشیم، فضا و قدرت پردازشی مساله‌ی مهمی نخواهد بود ولی فرض کنید قرار است یک میلیون نفر و صدها میلیون پیوند بین

دوستان آن‌ها را مورد بررسی قرار دهید. در اینجا مساله فضا و پردازش اهمیت ویژه‌ای پیدا می‌کند.

حساب‌های کاربری توییترمان را در جدولی به اسم People به جای جدول Twitter – که در مثال قبل استفاده کردیم – قرار می‌دهیم. جدول People یک ستون اضافی برای ذخیره کلیدهای عددی مرتبط با سطرِ کاربر توییتر را در خود دارد. SQLite ویژگی‌ای برای اضافه کردن خودکار مقدار کلید برای هر سطری با نوع خاصی از ستون داده‌ها که وارد جدول می‌کنیم دارد (INTEGER PRIMARY KEY).

می‌توانیم جدول People را با یک ستون اضافه به اسم id به شکل زیر درست کنیم:

```
CREATE TABLE People
    (id INTEGER PRIMARY KEY, name TEXT UNIQUE, retrieved
INTEGER)
```

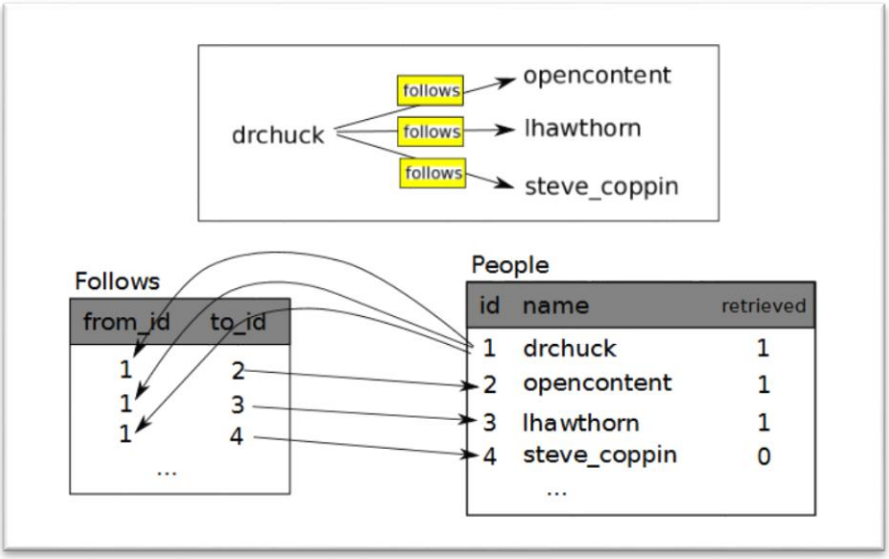
توجه کنید که در هر سطر از جدول People دیگر با شمار دوستان کاری نداریم. زمانی که INTEGER PRIMARY KEY را به عنوان نوع ستون id انتخاب می‌کنیم، نشان می‌دهیم که می‌خواهیم از SQLite برای مدیریت این ستون و گمارش یک کلیدِ عددی منحصر به فرد برای هر سطری که به صورت خودکار وارد می‌شود استفاده کنیم. همچنین کلمه‌ی کلیدی UNIQUE را به کار بردیم تا نشان دهیم به SQLite اجازه‌ی وارد کردن دو سطر را با یک مقدار برای name نمی‌دهیم.

حالا به جای ساخت جدول Pals، جدولی با اسم Follows با دو ستون عددی from_id و to_id می‌سازیم. همچنین آن را مقید به داشتن ترکیب منحصر به فرد برای from_id و to_id می‌کنیم؛ به عبارتی نمی‌توانیم دو سطر مشابه را وارد دیتابیس کنیم.

```
CREATE TABLE Follows
    (from_id INTEGER, to_id INTEGER, UNIQUE(from_id, to_id)
)
```

وقتی عبارت UNIQUE را به جدول‌ها اضافه می‌کنیم، در حقیقت قوانینی را تعیین می‌کنیم، و از دیتابیس می‌خواهیم که در حین وارد کردن رکوردها، آن‌ها را اجرایی کند. این قوانین هم جلوی خطاهای احتمالی را خواهد گرفت و هم اینکه نوشتن کد را ساده‌تر خواهد کرد.

به طور خلاصه، در ساخت جدول Follows ما در حال مدل‌سازی یک رابطه / relationship هستیم. مدلی که نشان می‌دهد شخصی، فرد دیگری را دنبال می‌کند؛ و نشان‌دهنده‌ی یک جفت عدد برای نمایان کردن **(الف)** افرادی است که با هم در ارتباط هستند و **(ب)** سمت و سوی این ارتباط.




## برنامه‌نویسی با چندین جدول

اکنون برنامه‌ی اسپایدر توییتر را با استفاده از دو جدول، بازنویسی می‌کنیم. جدول‌ها شامل کلیدهای اصلی و کلیدهای مرجع می‌شوند. کد نسخه‌ی جدید برنامه به شرح زیر است:

```
import urllib.request, urllib.parse, urllib.error
import twurl
import json
import sqlite3
import ssl


TWITTER_URL =
'https://api.twitter.com/1.1/friends/list.json'


conn = sqlite3.connect('friends.sqlite')
cur = conn.cursor()


cur.execute('''CREATE TABLE IF NOT EXISTS People
            (id INTEGER PRIMARY KEY, name TEXT UNIQUE,
retrieved INTEGER)''')
cur.execute('''CREATE TABLE IF NOT EXISTS Follows
            (from_id INTEGER, to_id INTEGER, UNIQUE(from_id,
to_id))''')


# Ignore SSL certificate errors
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE


while True:
    acct = input('Enter a Twitter account, or quit: ')
    if (acct == 'quit'): break
    if (len(acct) < 1):
        cur.execute('SELECT id, name FROM People WHERE
retrieved = 0 LIMIT 1')
        try:
            (id, acct) = cur.fetchone()
        except:
            print('No unretrieved Twitter accounts found')
            continue
    else:
```

```
        cur.execute('SELECT id FROM People WHERE name = ?
LIMIT 1',
                    (acct, ))
        try:
            id = cur.fetchone()[0]
        except:
            cur.execute('''INSERT OR IGNORE INTO People
                        (name, retrieved) VALUES (?, 0)''',
(acct, ))
            conn.commit()
            if cur.rowcount != 1:
                print('Error inserting account:', acct)
                continue
            id = cur.lastrowid


    url = twurl.augment(TWITTER_URL, {'screen_name': acct,
'count': '100'})
    print('Retrieving account', acct)
    try:
        connection = urllib.request.urlopen(url,
context=ctx)
    except Exception as err:
        print('Failed to Retrieve', err)
        break


    data = connection.read().decode()
    headers = dict(connection.getheaders())


    print('Remaining', headers['x-rate-limit-remaining'])


    try:
        js = json.loads(data)
    except:
        print('Unable to parse json')
        print(data)
        break
```

```
    # Debugging
    # print(json.dumps(js, indent=4))


    if 'users' not in js:
        print('Incorrect JSON received')
        print(json.dumps(js, indent=4))
        continue


    cur.execute('UPDATE People SET retrieved=1 WHERE name =
?', (acct, ))


    countnew = 0
    countold = 0
    for u in js['users']:
        friend = u['screen_name']
        print(friend)
        cur.execute('SELECT id FROM People WHERE name = ?
LIMIT 1',
                    (friend, ))
        try:
            friend_id = cur.fetchone()[0]
            countold = countold + 1
        except:
            cur.execute('''INSERT OR IGNORE INTO People
(name, retrieved)
                        VALUES (?, 0)''', (friend, ))
            conn.commit()
            if cur.rowcount != 1:
                print('Error inserting account:', friend)
                continue
            friend_id = cur.lastrowid
            countnew = countnew + 1
        cur.execute('''INSERT OR IGNORE INTO Follows
(from_id, to_id)
                    VALUES (?, ?)''', (id, friend_id))
    print('New accounts=', countnew, ' revisited=',
countold)
    print('Remaining', headers['x-rate-limit-remaining'])
```

```
    conn.commit()
cur.close()


# Code: http://www.py4e.com/code3/twfriends.py
```

برنامه کمی پیچیده‌تر شد ولی الگویی که برای پیوند یک کلید عددی به جدول‌ها نیاز داشتیم را به خوبی نشان می‌دهد. الگوهای پایه‌ای عبارتند از:

۱) ساخت جدول‌ها با کلیدهای اصلی و محدودیت‌هایشان؛

۲) وقتی یک کلید منطقی برای یک شخص (مانند نامِ حساب کاربری) داریم و احتیاج به مقدار id برای شخص داریم، چه شخصِ در جدول People وجود داشته باشد چه نداشته باشد ما نیاز داریم که: (۱) به دنبال شخص در جدول Peopole بگردیم و مقدار id برای شخص را دریافت کنیم یا (۲) شخص را به جدول People اضافه کرده و مقدارِ id برای سطر اضافه شده را دریافت کنیم.

۳) سطری را وارد کنیم که رابطه‌ی «دنبال شدن‌ها» را ثبت کند.

هر کدام از این موارد را به صورت جداگانه توضیح می‌دهیم.

## محدودیت‌ها در جدول‌های دیتابیس

با شکل دادن به ساختارِ جدول، به سیستم دیتابیس می‌گوییم که تعدادی از قوانین را برای ما اجرایی کند. این قوانین به ما کمک می‌کند تا از اشتباهات اجتناب کرده و داده‌های مشکل‌دار در جدول مشخص شوند. زمانی که جدول‌هایمان را می‌سازیم:

```
cur.execute('''CREATE TABLE IF NOT EXISTS People
    (id INTEGER PRIMARY KEY, name TEXT UNIQUE, retrieved
INTEGER)''')
cur.execute('''CREATE TABLE IF NOT EXISTS Follows
    (from_id INTEGER, to_id INTEGER, UNIQUE(from_id,
to_id))''')
```

ابتدا تعیین می‌کنیم که ستون name در جدول People بایستی UNIQUE و منحصربه‌فرد باشد. همچنین شماره‌ی هر سطر از جدول Follows نیز بایستی همین خاصیت منحصربه‌فرد بودن را داشته باشد. این محدودیت‌ها باعث می‌شود تا از اشتباهاتی نظیر وارد کردنِ رابطه‌ی یکسان برای بیش از یک بار اجتناب شود.

به‌کارگیری این محدودیت‌ها در کد زیر مشخص می‌شود:

```
cur.execute('''INSERT OR IGNORE INTO People (name,
retrieved)
    VALUES ( ?, 0)''', ( friend, ) )
```

با اضافه کردن عبارت OR IGNORE به گزاره‌ی INSERT مشخص کردیم که اگر INSERT از قوانین name، که بایستی منحصربه‌فرد باشد، تخطی کرد، سیستم دیتابیس اجازه دارد که آن INSERT را نادیده بگیرد. در اینجا از محدودیت‌های دیتابیس به عنوان شبکه‌ای امن برای اطمینان از صحت انجام کارها استفاده کرده‌ایم.

به طریق مشابه، کد زیر از ورود رابطه‌ی مشابه در Follows جلوگیری به عمل می‌آورد.

```
cur.execute('''INSERT OR IGNORE INTO Follows
    (from_id, to_id) VALUES (?, ?)''', (id, friend_id) )
```

در یک کلام: ما به دیتابیس می‌گوییم که تلاش برای وارد کردن (INSERT) اگر با قوانین محدودیت (در اینجا منحصربه‌فرد بودن) مغایرت داشت، نادیده گرفته شود.

## دریافت یا وارد کردن یک رکورد

وقتی از کاربر، برای وارد کردن حساب کاربری توییتر، درخواست صادر می‌کنیم، اگر حساب کاربری از قبل موجود باشد، بایستی به دنبال مقدار id آن بگردیم. اگر حساب کاربری در جدول People موجود نبود، بایستی یک رکورد وارد کرده و مقدار id را از سطر وارد شده دریافت کنیم.

این الگوی بسیار رایج، دوبار در برنامه‌ی بالا مورد استفاده قرار گرفت. این کد به ما نشان می‌دهد در زمانی‌که یک screen_name از گره‌ی یک user در JSON توییترِ برگرداننده شده است، چطور به دنبال id حسابِ کاربری دوستی بگردیم.

از آنجایی که با گذشت زمان، احتمال آن می‌رود که حساب کاربری مورد نظر در دیتابیس ما موجود باشد، ابتدا رکوردِ موجود People را با استفاده از یک گزاره‌ی SELECT بررسی می‌کنیم.

اگر همه چیز به خوبی پیش رفت، در قسمت try ما رکورد را با استفاده از ()fetchone دریافت کرده و سپس اولین (و تنها) عنصر از تاپلِ برگردانده شده را گرفته و در friend_id ذخیره می‌کنیم.

اگر SELECT با مشکل مواجه شود، کد ()fetchone[0] با خطا روبرو شده و جریان به سمت بخش except هدایت می‌شود.

```
    friend = u['screen_name']
    cur.execute('SELECT id FROM People WHERE name = ? LIMIT
1',
        (friend, ) )
    try:
        friend_id = cur.fetchone()[0]
        countold = countold + 1
    except:
        cur.execute('''INSERT OR IGNORE INTO People (name,
retrieved)
            VALUES ( ?, 0)''', ( friend, ) )
        conn.commit()
        if cur.rowcount != 1 :
            print('Error inserting account:',friend)
            continue
        friend_id = cur.lastrowid
        countnew = countnew + 1
```

وقتی جریان به سمت کد except برود، معنای آن وجود نداشتن سطر مورد نظر است. در نتیجه بایستی که آن سطر را اضافه کنیم. با استفاده از INSERT OR IGNORE

برای اجتناب از خطاهای احتمالی، و سپس فراخوانی ()commit، دیتابیس را [حتی به زور هم که شده] به‌روزرسانی می‌کنیم. بعد از اینکه نوشتن به پایان رسید، با استفاده از cur.rowcount بررسی می‌کنیم که چه تعدادی از سطرها تحت تاثیر قرار گرفته‌اند. از آنجایی که ما تنها یک سطر را وارد کرده‌ایم، اگر سطرهای تاثیرگرفته بیش از ۱ باشد، در حقیقت با یک خطا مواجه شده‌ایم.

اگر عملیات وارد کردن با INSERT به خوبی انجام شده باشد، با نگاه به cur.lastrowid مقدارِ اختصاص داده شده به ستون id در سطر جدیدمان را به دست خواهیم آورد.

## ذخیره‌ی ارتباطات دوستان

زمانیکه مقدار کلید برای کاربر توییتر و دوستش را در JSON به دست آوردیم، وارد کردن دو عدد در جدول Follows با استفاده از کد زیر ساده خواهد بود:

```
cur.execute('INSERT OR IGNORE INTO Follows (from_id, to_id)
VALUES (?, ?)',
    (id, friend_id) )
```

با اضافه کردن OR IGNORE به گزاره‌ی INSERT افسار را به دست دیتابیس دادیم تا از «ورود دومرتبه» یک رابطه، با ساختن جدولی، که قید و شرط‌های منحصربه‌فردی دارد، جلوگیری کند.

نمونه‌ای از اجرای برنامه را با هم ببینیم:

```
Enter a Twitter account, or quit:
No unretrieved Twitter accounts found
Enter a Twitter account, or quit: drchuck
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 20  revisited= 0
Enter a Twitter account, or quit:
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 17  revisited= 3
Enter a Twitter account, or quit:
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 17  revisited= 3
Enter a Twitter account, or quit: quit
```

با حساب کاربری drchuck شروع کردیم و سپس به برنامه اجازه دادیم تا دو حساب کاربری را به صورت خودکار انتخاب و به دیتابیس اضافه کند.

چند سطر اول در جدول‌های People و Follows بعد از اجرا کامل شده‌اند:

```
People:
(1, 'drchuck', 1)
(2, 'opencontent', 1)
(3, 'lhawthorn', 1)
(4, 'steve_coppin', 0)
(5, 'davidkocher', 0)
55 rows.
Follows:
(1, 2)
(1, 3)
(1, 4)
(1, 5)
(1, 6)
60 rows.
```

فیلدهای name، id و visited در جدول People را به همراه شماره‌ی هر دو طرف رابطه در جدول Follows می‌بینید. در جدول People سه شخص بازدید شده‌اند و اطلاعاتشان دریافت شده است. اطلاعات موجود در جدول Follows مشخص می‌کند

که drchuck (user 1) دوست تمام افرادیست که در پنج سطر اول نشان داده شده‌اند. با عقل جور در می‌آید چرا که داده‌های اولیه‌ای که دریافت و ذخیره کردیم، در اصل دوستان drchuck بودند. اگر سطرهای بیشتری از جدول Follows را چاپ می‌کردیم، دوستانِ user 2 و user 3 را نیز نمایش داده می‌شدند.

## سه نوع از کلیدها

حالا که شروع به ساخت یک «مدل داده» کرده و داده‌هایمان را در جدول‌های متصل به هم قرار دادیم و سطرهای آن‌ها را به وسیله‌ی کلیدها پیوند دادیم بایستی واژگان خاصِ کلیدها را یاد بگیریم. درکل سه نوع کلید در یک مدل دیتابیس وجود دارد.

I. **کلید منطقی** که کلیدی در «جهان واقع» است و ممکن است برای پیدا کردن یک سطر مورد استفاده قرار بگیرد. در مثالِ مدل داده‌ی ما، فیلد name یک کلید منطقی بود. این کلید screen name یا همان نام کاربریِ کاربر بود و چندین و چند مرتبه با استفاده از فیلد NAME به دنبال سطرِ کاربر در برنامه می‌گشتیم. معمولا معقول است که یک قیدِ منحصربه‌فرد برای یک کلید منطقی تعیین کنیم. از آنجایی که کلید منطقی چگونگی گشتن به دنبال یک سطر را از «دنیای بیرون» مشخص می‌کند منطقی نیست که چندین سطر با مقدار یکسانی در جدول داشته باشیم.

II. **کلید اصلی** معمولا یک عدد است که به صورت خودکار توسط دیتابیس تعیین می‌شود. این عدد در خارج از برنامه معنای خاصی ندارد و تنها برای پیوند دادن سطرهای جدول‌های مختلف به یکدیگر به کار می‌رود. زمانی که می‌خواهیم نگاهی به یک سطر در جدولی کنیم، جستجو با استفاده از کلید منطقی معمولا سریع‌ترین راه خواهد بود. این کلیدها مقدار خیلی کمی فضا اشغال می‌کنند و می‌توان آن‌ها

را با سرعت هرچه تمام‌تر مقایسه کرد یا نظم داد. در مدل داده‌ی ما، فیلد id مثالی از یک کلید اصلی بود.

III. **کلید خارجی** معمولا یک عدد است که به کلید اصلی از سطرِ مرتبط با آن در یک جدول دیگر ارجاع دارد و اشاره می‌کند. یک مثال از کلید خارجی در مدل داده ما from_id بود.

بر اساس رسم رایج، فیلدِ کلید اصلی را id می‌نامیم و پیشوند id_ را به تمام فیلدهای نام‌هایی که یک کلید خارجی‌اند اضافه می‌کنیم.

## استفاده از JOIN برای دریافت داده‌ها

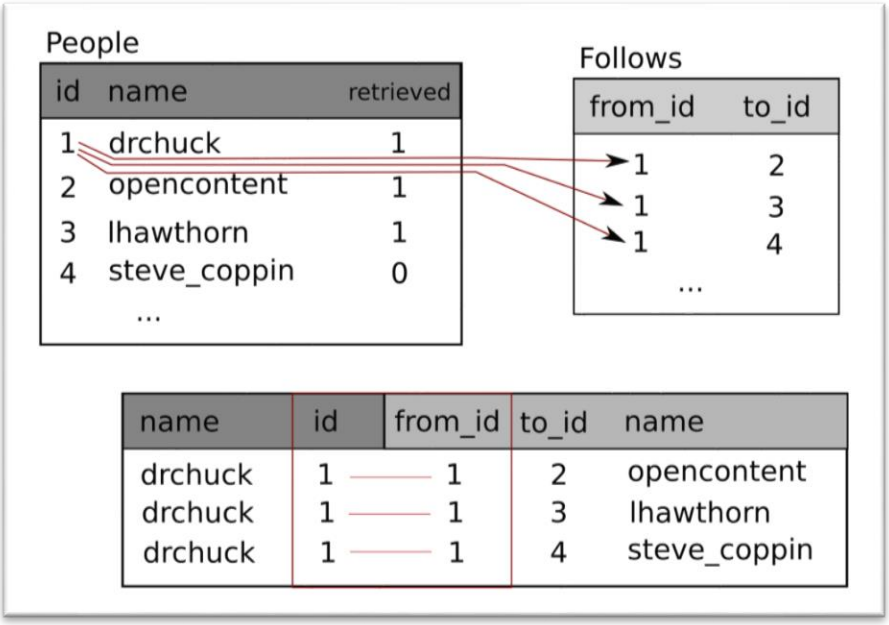
اکنون با تعیین قوانینی برای بهنجار بودن دیتابیس، تقسیم داده‌ها بین دو جدول و پیوند دادن آن‌ها با استفاده از کلیدهای اصلی و خارجی، می‌توانیم از SELECT برای جمع‌آوری دوباره‌ی اطلاعات از جدول‌ها استفاده کنیم.

SQL از عبارت JOIN برای اتصال دوباره‌ی این جدول‌ها بهره می‌برد. در عبارت JOIN شما فیلدهایی را مشخص می‌کنید که برای اتصال سطرها، بین جدول‌ها، مورد استفاده قرار گرفته‌اند.

مثال زیر استفاده از SELECT به همراه عبارت JOIN را نشان می‌دهد:

```
SELECT * FROM Follows JOIN People
    ON Follows.from_id = People.id WHERE People.id = 1
```

عبارت JOIN فیلدهایی را مشخص می‌کند که در بین جدول‌های Follows و People انتخاب می‌کنیم. عبارت ON چگونگی الحاق دو جدول را نشان می‌دهد: سطر را از جدول Follows بگیر و به سطری از جدول People الصاق کن. مقدار id سطر مورد نظر در جدول People بایستی مطابق با from_id در جدول Follows باشد.




نتیجه‌ی JOIN ساخت یک metarows که شامل هر دو فیلد است: فیلد People و فیلد مطابق آن از Follows. اگر بیش از یک موردِ منطبق بین فیلد id از People و from_id از Follow وجود داشته باشد، JOIN یک metarow برای هر جفت از سطرها که جور در آمده می‌سازد و داده‌ها را بنا به نیاز تکثیر می‌کند.

کد زیر داده‌هایی را به نمایش می‌گذارد که ما بعد از چند بار اجرای برنامه‌ی چند جدوله‌ی اسپایدر توییتر به دست می‌آوریم.

```
import sqlite3


conn = sqlite3.connect('friends.sqlite')
cur = conn.cursor()


cur.execute('SELECT * FROM People')
count = 0
print('People:')
for row in cur:
    if count < 5: print(row)
    count = count + 1
print(count, 'rows.')


cur.execute('SELECT * FROM Follows')
count = 0
print('Follows:')
for row in cur:
    if count < 5: print(row)
    count = count + 1
print(count, 'rows.')


cur.execute('''SELECT * FROM Follows JOIN People
            ON Follows.to_id = People.id
            WHERE Follows.from_id = 2''')
count = 0
print('Connections for id=2:')
for row in cur:
    if count < 5: print(row)
    count = count + 1
print(count, 'rows.')


cur.close()


# Code: http://www.py4e.com/code3/twjoin.py
```

در این برنامه، ابتدا People و Follows را استخراج و سپس زیرمجموعه‌ای از جدول‌هایی که به یکدیگر پیوند خورده‌اند را به دست می‌آوریم.

خروجی برنامه به این شکل خواهد بود:

```
python twjoin.py
People:
(1, 'drchuck', 1)
(2, 'opencontent', 1)
(3, 'lhawthorn', 1)
(4, 'steve_coppin', 0)
(5, 'davidkocher', 0)
55 rows.
Follows:
(1, 2)
(1, 3)
(1, 4)
(1, 5)
(1, 6)
60 rows.
Connections for id=2:
(2, 1, 1, 'drchuck', 1)
(2, 28, 28, 'cnxorg', 0)
(2, 30, 30, 'kthanos', 0)
(2, 102, 102, 'SomethingGirl', 0)
(2, 103, 103, 'ja_Pac', 0)
20 rows.
```

ستون‌هایی از جدول‌های People و Follows را می‌بینید. آخرین دسته از سطرها نتیجه‌ی SELECT به همراه عبارت JOIN است.

در انتخابِ آخر، به دنبال حساب‌های کاربری دوستان از opencontent می‌گردیم (به عبارتی People.id=2).

در هر یک از metarowها در انتخابِ آخر، دو ستون ابتدایی از جدول Follows و ستون‌های بعدی، که بین سه تا پنج عددند، از جدول People هستند. همچنین

می‌بینید که دومین ستون (Follows.to_id) با سومین ستون (People.id) در هر metarow پیوند خورده، مطابق است.

## جمع‌بندی

در این فصل بخش‌های زیادی را از مبحث پایگاه داده‌ها بررسی کردیم و دید کلی از اصول استفاده از یک پایگاه داده را در پایتون آموختیم. نوشتن کدی که با استفاده از یک دیتابیس اقدام به ذخیره‌سازی داده‌ها کند، بسیار پیچیده‌تر از استفاده از دیکشنری‌ها یا فایل‌های ساده است در نتیجه تا زمانی که برنامه‌ی شما به دیتابیس نیازی نداشته باشد، لازم نیست که خودتان را با دردسرهای آن گلاویز کنید. موقعیت‌هایی که دیتابیس به کارتان خواهد آمد عبارتند از:

I. زمانی‌که نیاز به به‌روزرسانی‌های کوچک و تصادفی زیادی در یک دسته داده‌ی بزرگ دارید؛
II. زمانی‌که داده‌های شما آنقدر بزرگ است که در یک دیکشنری جا نمی‌شود و نیاز به جستجو در بین آن‌ها به کرات احساس می‌شود؛
III. زمانی‌که یک پروسه‌ی بلندمدت دارید و نیاز است که داده‌های شما بعد از توقف و شروع مجدد برنامه همچنان در دسترس برنامه، بعد از اجرای دوباره، باشند.

می‌توانید یک دیتابیس ساده را با یک جدول تکی بسازید و نیاز بسیاری از برنامه‌ها را با آن رفع کنید، ولی اکثر مسائل با چند جدولی حل می‌شوند که پیوند خورده‌اند و رابطه‌ای بین سطرهای آن‌ها برقرار است. زمانی‌که شروع به ساختن پیوند بین جدول‌ها کردید، داشتن یک طرح هوشمندانه و رعایت قوانین نرمال‌سازیِ دیتابیس ضروری به نظر می‌رسد. به این صورت شما بیشترین بهره را از قابلیت‌های دیتابیس خواهید برد. از آن‌جایی که دلیل اصلی برای استفاده از دیتابیس، سر و کله زدن با مقدار زیادی داده است، پربازده و اثربخش بودن مدل داده برای سرعت بخشیدن به برنامه‌تان مهم است.

## اشکال‌زدایی

روشی رایج برای بررسی نتایج اجرای برنامه‌ی پایتونی که به یک دیتابیس SQLite متصل شده است، استفاده از مرورگر دیتابیس برای SQLite است. مرورگر اجازه‌ی بررسی سریع را، برای اطمینان از صحت عملکرد برنامه، به شما می‌دهد.

بایستی مراقب باشید چرا که SQLite اجازه‌ی تغییرِ داده‌های مشابه را در آن واحد به دو برنامه نمی‌دهد. به عنوان مثال، اگر دیتابیسی را در مرورگر باز کنید و تغییری در آن صورت دهید، تا قبل از فشردن دکمه «save» برنامه دیتابیس را قفل خواهد کرد و دیگر برنامه‌ها دسترسی به آن فایل نخواهند داشت. علی الخصوص اگر برنامه‌ی پایتونی نیاز به دسترسی به آن فایل داشته باشد، دستش کوتاه خواهد بود.

راه‌حل ساده است: یا مطمئن شوید که مرورگرِ دیتابیس را بسته‌اید یا از منوی File قبل از اینکه با پایتون به سراغ دیتابیس بروید آن را ببندید.

## واژگان فصل

**صفت / Attribute:**
یکی از مقدارهای داخل تاپل که معمولا «ستون» یا «فیلد» خوانده می‌شود.

**محدودیت / Constraint:**
اگر دیتابیس را مجبور به رعایت قوانینی برای یک فیلد یا یک سطر در جدولی کنیم در اصل قید و شرطی برای آن تعیین کرده‌ایم. به این کار محدود کردن می‌گوییم. یک محدودیت رایج برای دیتابیس اطمینان از نبودِ مقدار همسان در یک فیلد خاص است. به عبارتی بایستی تمام مقادیر منحصربه‌فرد باشند.

**نشانه‌گر، کرسِر / Cursor:**
نشانگر به شما اجازه می‌دهد که دستورات SQL را در یک دیتابیس اجرا و داده‌ها را دریافت کنیم. نشانگر برابر با یک سوکت یا دستگیره‌ی فایل برای، به ترتیب، ارتباط‌های شبکه و فایل‌هاست.

**مرورگر دیتابیس / Database Browser:**
نرم‌افزاری که به شما اجازه‌ی ارتباط مستقیم با یک دیتابیس را می‌دهد. مروگر دیتابیس همچنین امکان دستکاریِ مستقیم دیتابیس بدون نوشتن برنامه را میسر می‌سازد.

**کلید خارجی / Foreign Key:**
یک کلیدِ عددی که به کلید اصلی از سطری در جدول دیگری اشاره دارد. کلیدهای خارجی رابطه‌هایی را، بین سطرهای ذخیره شده در جدول‌های مختلف، می‌سازد.

**شاخص / Index:**
داده‌های اضافه‌تر که نرم‌افزار دیتابیس به عنوان سطرها نگه‌داری کرده و به جدول تزریق می‌کند؛ به این صورت جستجو سریع‌تر می‌شود.

**کلید منطقی / Logical Key:**
کلیدی که «دنیای خارج» برای پیدا کردن یک سطر خاص به کار می‌گیرید. به عنوان مثال در یک جدول از حساب‌های کاربری، ایمیل شخصی، یک کاندید خوب به عنوان، کلید منطقی برای داده‌های آن کاربر است.

**نرمال‌سازی / Normalization:**
به طراحی یک مدل داده به صورتی که هیچ داده‌ای تکرار نشود، نرمال‌سازی می‌گویند. هر آیتم از داده‌ها در یک مکان در دیتابیس ذخیره و با استفاده از یک کلید خارجی ارجاع داده می‌شوند.

**کلید اصلی / Primary Key:**

یک کلید عددی که به هر سطر الصاق شده است و برای ارجاع به یک سطر در جدول از طریق جدول دیگر مورد استفاده قرار می‌گیرد. معمولا دیتابیس به‌صورتی پیکربندی می‌شود که کلیدهای اصلی به صورت خودکار به سطرهای وارد شده الصاق شوند.

**رابطه / Relation:**

مکانی در یک دیتابیس که شامل تاپل‌ها و صفات می‌شود. معمولا یک «جدول» خوانده می‌شود.

**تاپل Tuple:**

یک ورودی تکی در جدول دیتابیس را تاپل می‌گوییم. معمولا یک «سطر» خوانده می‌شود.

# فصل ۱۶

# مصورسازی داده

## مصور کردن داده‌ها

تا اینجا زبان برنامه‌نویسی پایتون را یاد گرفتیم و فهمیدم چطور از پایتون، شبکه، و دیتابیس‌ها برای کار با داده‌ها استفاده کنیم.

در این فصل، نگاهی به سه برنامه، که موارد فوق‌الذکر را در کنار هم می‌آورد، خواهیم انداخت، سپس داده‌ها را مدیریت و تصویرسازی خواهیم کرد. ممکن است بخواهید از این برنامه‌ها به عنوان کدِ نمونه استفاده کنید تا در آینده از کد منبع آن برای حل مشکلات واقعی بهره ببرید.

هر کدام از این برنامه‌های کاربردی یک فایل زیپ است که می‌توانید فایل‌ها را از درون آن استخراج و اجرا کنید.

## ساخت یک نقشه‌ی گوگل داده‌های کدبندی‌های جغرافیایی

در این پروژه از Google geocoding API به منظور بررسی داده‌های وارد شده توسط کاربر و قرار دادن نامِ دانشگاه‌ها روی نقشه‌ی گوگل استفاده می‌کنیم.


برای شروع، برنامه را از پیوند زیر دانلود کنید:

http://www.py4e.com/code3/geodata.zip

اولین مساله محدودیتِ روزانه‌ی Google geocoding API در صورت استفاده رایگان از آن است. اگر داده‌های زیادی دارید، شاید لازم باشد که پروسه را چندین مرتبه متوقف و دوباره اجرا کنید. به همین خاطر مساله را به دو بخش می‌شکنیم.

در مرحله‌ی اول ما داده‌های ورودی را در where.data قرار می‌دهیم و سپس در آنِ واحد یک خط از آن را می‌خوانیم. بعد از آن، داده‌های کدبندی جغرافیایی را از گوگل گرفته و در دیتابیسی با نام geodata.sqlite ذخیره می‌کنیم. قبل از استفاده از geocoding API برای هر مکانِ وارد شده توسط کاربر، بررسی می‌کنیم که داده‌ی وارد شد آیا در جدول از قبل وجود داشته یا خیر. دیتابیس به عنوان یک کَش یا انبارِ محلی برای داده‌های کدبندی جغرافیایی عمل می‌کند تا مطمئن شویم یک سوال را دو مرتبه از گوگل نمی‌پرسیم.

با حذف فایل geodata.sqlite پروسه مجددا راه‌اندازی خواهد شد.

برنامه‌ی geoload.py را اجرا کنید. این برنامه خطوط ورودی در where.data را می‌خواند و سپس بررسی می‌کند که آیا هر خط در دیتابیس وجود دارد یا خیر. اگر داده‌ای برای آن مکان در دیتابیس موجود نباشد geocoding API فراخوانی شده و داده‌ها دریافت و در دیتابیس ذخیره می‌شود.

یک نمونه از اجرای برنامه را در اینجا می‌بینیم. همانگونه که مشاهده می‌کنید بعضی از داده‌ها تکراری هستند:

```
Found in database  Northeastern University
Found in database  University of Hong Kong, ...
Found in database  Technion
Found in database  Viswakarma Institute, Pune, India
Found in database  UMD
Found in database  Tufts University


Resolving Monash University
Retrieving http://maps.googleapis.com/maps/api/
    geocode/json?address=Monash+University
Retrieved 2063 characters {    "results" : [
{'status': 'OK', 'results': ... }


Resolving Kokshetau Institute of Economics and Management
Retrieving http://maps.googleapis.com/maps/api/
    geocode/json?address=Kokshetau+Inst ...
Retrieved 1749 characters {    "results" : [
{'status': 'OK', 'results': ... }
...
```

پنج مکان ابتدایی در دیتابیس از پیش موجودند و ما از آن‌ها پرس‌وجو می‌کنیم. برنامه نقاطی که به عنوان مکان جدید، پیدا می‌کند را بررسی و شروع به دریافت آن‌ها می‌کند.

برنامه geoload.py در هر زمانی که بخواهید می‌تواند متوقف شود. همچنین می‌توانید محدودیتی برای فراخوانی geocoding API در هر اجرا تعیین کنید. با وجود

اینکه where.data تنها چند صد آیتم در خود دارد، احتمالا به مشکلی از نظر محدودیت برخورد نخواهید کرد. اگر داده‌های بیشتری دارید ممکن است نیاز به اجرای چندین باره در چندین روز متوالی برای تا تمام داده‌های کدبندی‌شده جغرافیایی داشته باشید.

وقتی که داده‌ها در geodata.sqlite شروع به ذخیره شدن کردند، می‌توانید داده‌ها را با استفاده از برنامه geodump.py مصورسازی کنید. این برنامه دیتابیس را می‌خواند و فایل where.js را با مکان، طول و عرض جغرافیایی در یک فرم قابل اجرا به زبان جاوااسکریپت می‌نویسد.

برنامه‌ی geodump.py به شکل زیر اجرا می‌شود:

```
Northeastern University, ... Boston, MA 02115, USA
42.3396998 -71.08975
Bradley University, 1501 ... Peoria, IL 61625, USA
40.6963857 -89.6160811
...
Technion, Viazman 87, Kesalsaba, 32000, Israel 32.7775
35.0216667
Monash University Clayton ... VIC 3800, Australia -
37.9152113 145.134682
Kokshetau, Kazakhstan 53.2833333 69.3833333
...
12 records written to where.js
Open where.html to view the data in a browser
```

فایلِ where.html شامل HTML و JavaScript برای مصورسازی یک نقشه از گوگلِ می‌شود. این فایل، داده‌های موجود در where.js را می‌خواند. where.js شروع به دریافت داده‌ها برای مصورسازی آن‌ها می‌کند. قالب فایل where.js به این صورت است:

```
myData = [
[42.3396998,-71.08975, 'Northeastern Uni ... Boston, MA
02115'],
[40.6963857,-89.6160811, 'Bradley University, ... Peoria, IL
61625, USA'],
[32.7775,35.0216667, 'Technion, Viazman 87, Kesalsaba,
32000, Israel'],
   ...
];
```

در اینجا یک متغیر جاوااسکریپت داریم که شامل لیستی از لیست‌هاست. سینتکس لیست‌های جاوااسکریپت بسیار شبیه به پایتون است، به همین خاطر بایستی برایتان آشنا باشد.

فایل where.html را در مرورگر خود باز کنید تا موقعیت‌های مکانی را ببینید. می‌تواند روی قسمت‌های سنجاق‌شده برای پیدا کردن مکانی، که geocoding API براساس داده‌های ورودی کاربر برگردانده، نشانگر را تکان دهید. اگر داده‌ای را مشاهده نمی‌کنید بایستی جاوااسکریپت یا کنسول توسعه‌دهنده‌ی مرورگر خود را بررسی کنید.

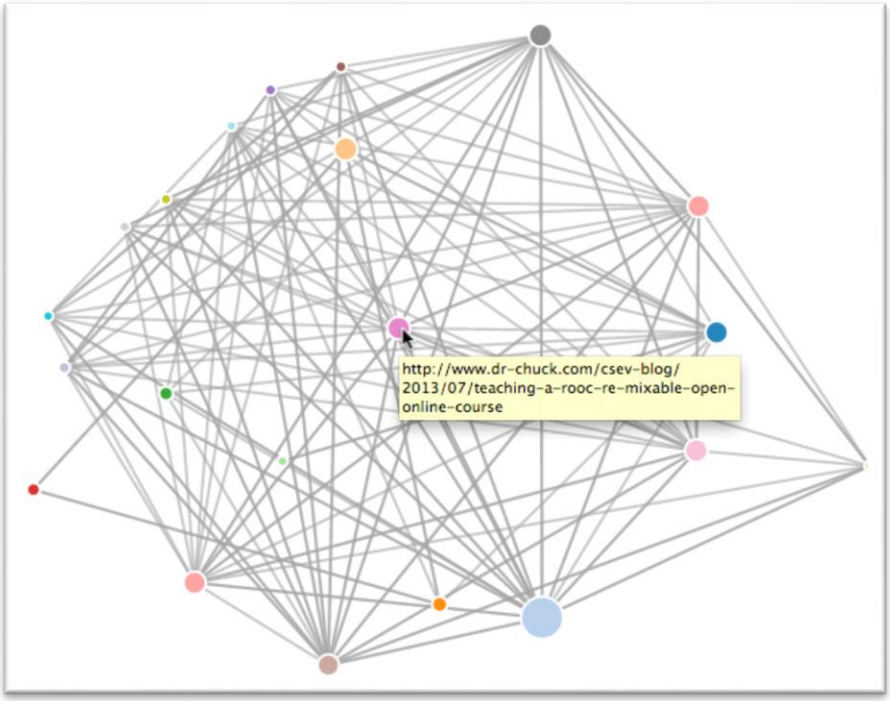
## مصورسازی شبکه‌ها و ارتباطات بینابین

در این برنامه، ما تعدادی از پردازه‌های یک موتور جستجو را اجرا می‌کنیم. ابتدا بخش کوچکی از وب را اسپایدر می‌کنیم و سپس نسخه‌ای ساده‌شده از الگوریتم Google page rank را استفاده می‌کنیم تا تعیین کنیم کدام صفحات با هم ارتباط تنگاتنی دارند. سپس رتبه‌ی صفحه و ارتباط صفحات محدود ما از وب را مصورسازی می‌کنیم. از کتابخانه‌ی مصورسازی D3 JavaScript برای تولید خروجی مصورسازی‌شده استفاده خواهیم کرد:

http://d3js.org

می‌توانید برنامه را از اینجا گرفته و استخراج کنید:

http://www.py4e.com/code3/pagerank.zip




برنامه‌ی اول (spider.py) در یک وب‌سایت می‌خزد و یک سری از صفحات را در دیتابیس spider.sqlite می‌گنجاند و پیوند بین صفحات را ثبت و ضبط می‌کند. با حذف فایل spider.sqlite و اجرای دوباره‌ی spider.py می‌توانید پروسه را از نو شروع 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 ما در یک صفحه خزیدیم و دو پیج را دریافت کردیم. اگر برنامه را مجددا اجرا کنیم و به آن بگوییم در بین صفحات بیشتری بخزد، دیگر در بین صفحاتی

که در دیتابیس موجودند نخواهد خزید. بعد از شروع مجدد، به‌صورت تصادفی به سمت صفحاتی که درونشان نخزیده رفته و از آنجا کار را آغاز می‌کند. در نتیجه هر اجرای spider.py بر مجموعه‌ی داده‌های دیتابیس می‌افزاید.

```
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 نقطه‌ی آغاز را در یک دیتابیس – در یک برنامه – داشته باشید. به آن‌ها «وب‌ها» یا «همباف‌ها» می‌گوییم. اسپایدر به صورت تصادفی، لینک‌هایی که بررسی نشده را از سطح همباف‌ها انتخاب می‌کند و به سمت صفحه‌ی بعدی برای خزیدن می‌رود.

اگر می‌خواهید که محتویات spider.sqlite را خالی کنید، اجرای spdump.py به شکل زیر به کارتان خواهد آمد:

```
(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 کار، تعداد پیوندهای ورودی، رتبه‌ی پیشین صفحه، رتبه‌ی جدید صفحه‌ی، شناسه‌ی صفحه و آدرس url صفحه را نشان می‌دهد. برنامه spdump.py تنها صفحاتی را نشان می‌دهد که حداقل یک پیوند ورودی به آن‌ها وجود داشته باشد.

به محضی که صفحاتی در دیتابیس پدیدار شد، می‌توانید page rank را از طریق برنامه‌ی sprank.py اجرا کنید. کافیست تعیین کنید که چند تکرار از page rank اجرا شود:

```
How many iterations:2
1 0.546848992536
2 0.226714939664
[(1, 0.559), (2, 0.659), (3, 0.985), (4, 2.135), (5, 0.659)]
```

برای مشاهده‌ی به‌روز شدن page rank می‌توانید دیتابیس را خالی کنید.

```
(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.
```

می‌توانید sprank.py را به تعداد دفعاتی که دوست دارید اجرا کنید. با هر اجرا، رتبه‌ی صفحه سامان‌مندتر می‌شود. می‌توانید sprank.py را چندین بار اجرا کنید و سپس با استفاده از spider.py و اجرای دوباره‌ی sprank.py مقدارهای مرتبط با رتبه‌ی صفحه را مجددا همگرا کنید. یک موتور جستجو معمولا برنامه‌های خزیدن و رتبه‌بندی را مدام اجرا می‌کند.

اگر می‌خواهید که محاسبات رتبه‌بندی صفحات را بدون خزیدنِ مجدد در صفحات وب بازنشانی کنید، می‌توانید ابتدا spreset.py را اجرا و سپس sprank.py را شروع‌مجدد کنید.

```
How many iterations:50
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 تکرارِ الگوریتمِ رتبه‌بندی صفحه، برنامه، متوسطِ تغییر در رتبه‌ی صفحه را چاپ می‌کند. در ابتدا شبکهٔ نامتعادل است و رتبه‌ی هر کدام از صفحات به شدت، با هر تکرار، تغییر می‌کند. ولی بعد از چند تکرار، همگراتر می‌شود. شما بایستی sprank.py را آنقدر اجرا کنید که مقدار رتبه‌ی صفحه کاملا همگرا شود.

اگر می‌خواهید که صفحات سطح بالا از نظر رتبه‌ی صفحه را مصورسازی کنید، برنامه spjson.py را، برای خواندن دیتابیس و نوشتن داده‌های مرتبط با صفحات دارای پیوندهای زیاد به فرمت JSON، اجرا کنید. نتیجه را می‌توانید توسط مرورگر وب بخوانید.

```
Creating JSON output on spider.json...
How many nodes? 30
Open force.html in a browser to view the visualization
```

با باز کردن فایل force.html در مرورگر خود، داده‌ها قابل مشاهده خواهند بود. یک طرح خودکار از گره‌ها و پیوندها به شما نشان داده خواهد شد. روی گره‌ها کلیک کنید و آن‌ها را بکشید. با دوبار کلیک روی یک گره URL آن گره را مشاهده خواهید کرد.

اگر سایر ابزارها را دوباره اجرا کردید، spjson.py را مجددا اجرا کرده و سپس دکمه‌ی refresh را برای مشاهده‌ی داده‌های جدید از spider.json روی مرورگر خود فشار دهید.

## مصورسازی داده‌های مراسلات (mail)

تا اینجای کتاب، با فایل‌های mbox-short.txt و mbox.txt به خوبی آشنا شده‌اید. حالا نوبت آن است که آنالیز داده‌های ایمیل را انجام دهیم.

در دنیای واقعی گاهی لازم است داده‌های مراسلات را از سرورها بیرون بکشیم. کار زمان‌بری است و ممکن است داده‌ها ناهماهنگ و پُرخطا بوده و نیاز به تمیزکاری و تنظیم داشته باشند. در این بخش با پیچیده‌ترین برنامه‌ای که تا به حال داشته‌ایم کار می‌کنیم و نزدیک به یک گیگابایت داده را برای مصورسازی آن‌ها بیرون می‌کشیم.


برنامه از طریق لینک زیر در دسترس است:

http://www.py4e.com/code3/gmane.zip

در اینجا از داده‌های یک لیست رایگان ایمیل به نامه www.gmane.org استفاده می‌کنیم. پروژه‌های منبع‌باز علاقه‌ی زیادی به این سرویس دارند چرا که آرشیوی از ایمیل‌ها را، با قابلیت جستجو، برای توسعه‌دهندگان فراهم می‌کند. همچنین خط‌مشی آزادمنشانه‌ای دارد و در استفاده از API آن دستتان باز خواهد بود. محدودیت نرخِ

استفاده وجود ندارد ولی از شما می‌خواهد که سرویس را زیاد تحت فشار قرار ندهید و تنها به بیرون کشیدن اطلاعاتی که به آن نیاز دارید بسنده کنید. شرایط استفاده از gmane را از طریق پیوند زیر بخوانید:

http://gmane.org/export.php

در استفاده از gmane.org مسئولانه رفتار کنید و تاخیر را در بین دسترسی‌هایتان به سرویس‌های مورد نظر قرار دهید و کارتان را در بازه‌ی زمانی طولانی پخش کنید تا فشار قابل توجهی به سرویس‌های ارائه شده توسط gmane.org وارد نشود. سوءاستفاده از این سرویس باعث نابودیِ آن برای همگان می‌شود.

وقتی داده‌های ایمیل Sakai توسط این نرم‌افزار اسپایدر شد، حدود یک گیگابایت داده تولید شده که چندین و چند بار اجرا در طول چند روز را می‌طلبد. فایل README.txt در فایل زیپ بالا شما را برای دریافت یک رونوشت از پیش آماده‌شده‌ی فایلِ content.sqlite راهنمایی خواهد کرد. این دیتابیس شامل اکثر ایمیل‌های موجود در مجموعه‌ی Sakai است و شما نیازی به پنج روز اسپایدر کردن برای جمع‌آوری آن‌ها نخواهد داشت. اگر محتوای اسپایدر شده را دانلود کردید، برای دریافت آخرین پیام‌ها لازم است که پروسه‌ی اسپایدار کردن را خودتان نیز اجرا کنید.

اولین مرحله، اسپایدر کردنِ مخزنِ gmane است. URL اصلی در gmane.py هارْدْکُد شده است؛ به این معنی که خود URL مربوط به لیست توسعه‌دهندگان Sakai در برنامه موجود است. برای تغییر آن و اسپایدر کردن یک URL دیگر می‌توانید آن را عوض کنید. البته از حذف فایل content.sqlite اطمینان حاصل کنید.

فایل gmane.py به صورت یک اسپایدر کش‌کننده‌ی مسئول عمل می‌کند و در هر ثانیه تنها یک ایمیل را به آرامی دریافت می‌کند. همین موضوع باعث می‌شود که راه دریافت‌ها توسط gmane بسته نشود. سپس تمام داده‌ها را در یک دیتابیس ذخیره می‌کند. برنامه می‌تواند به دفعاتی که نیاز دارید بسته یا شروع‌مجدد شود. احتمالا ساعت‌های زیادی طول خواهد کشید که داده‌ها دریافت شوند و نیاز به شروع مجدد آن به دفعات احساس خواهد شد.

نمونه‌ای از اجرا gmane.py و دریافت پنج پیام آخر از لیست توسعه‌دهندگان Sakai را با هم می‌بینیم:

```
How many messages:10
http://download.gmane.org/gmane.comp.cms.sakai.devel/51410/5
1411 9460
    nealcaidin@sakaifoundation.org 2013-04-05 re: [building
...
http://download.gmane.org/gmane.comp.cms.sakai.devel/51411/5
1412 3379
    samuelgutierrezjimenez@gmail.com 2013-04-06 re:
[building ...
http://download.gmane.org/gmane.comp.cms.sakai.devel/51412/5
1413 9903
    da1@vt.edu 2013-04-05 [building sakai] melete 2.9 oracle
...
http://download.gmane.org/gmane.comp.cms.sakai.devel/51413/5
1414 349265
    m.shedid@elraed-it.com 2013-04-07 [building sakai] ...
http://download.gmane.org/gmane.comp.cms.sakai.devel/51414/5
1415 3481
    samuelgutierrezjimenez@gmail.com 2013-04-07 re: ...
http://download.gmane.org/gmane.comp.cms.sakai.devel/51415/5
1416 0


Does not start with From
```

برنامه، content.sqlite را از یک تا اولین پیامی که اسپایدار نشده بررسی می‌کند و سپس از همان پیام شروع به اسپایدر کردن می‌کند. این کار تا زمانی ادامه دارد که به عدد دلخواه برسد یا با صفحه‌ای که فرمت درستی از پیام‌ها را ندارد برخورد کند.

گاهی gmane.org پیامی را جا می‌اندازد. احتمالا مدیران آن را پاک کرده‌اند یا از دست رفته است. اگر اسپایدر شما متوقف شود، به صورتی‌که به نظر می‌آید به یک پیامِ گم‌شده برخورده است، به برنامه‌ی مدیریت SQLite بروید و یک سطر با شناسه‌ی پیامِ گم‌شده ایجاد و بقیه فیلدها را خالی رها کنید. سپس gmane.py را شروع‌مجدد کنید.

این کار باعث می‌شود که اسپایدار بتواند بدون گیر افتادن، به کارش ادامه دهد. پیام‌های خالی در پردازش بعدی نادیده گرفته می‌شوند.

وقتی که شما تمام پیام‌ها را اسپایدر کردید و در content.sqlite قرار دادید، می‌توانید با اجرای دوباره‌ی gmane.py پیام‌های جدیدی که به لیست فرستاده شده را دریافت کنید.

داده‌های content.sqlite تقریبا خام هستند و مدل داده‌های آن‌ها ناکارا و غیرفشرده خواهد بود. مزیت این داده‌ها توانایی بررسی content.sqlite با استفاده از برنامه‌ی مدیریت SQLite و اشکال‌زدایی مشکلات مرتبط با پروسه‌ی اسپایدر کردن است. جستجو در این دیتابیس با توجه به کُند بودنش، کار معقولی نخواهد بود.

دومین پروسه اجرای gmodel.py است. این برنامه داده‌های خام را از content.sqlite خوانده و نسخه‌ی تر و تمیز از آن در فایل index.sqlite ایجاد می‌کند. این فایل خیلی کوچک‌تر است (اغلب یک دهم اندازه‌ی content.sqlite) و در آن، متنِ بدنه و هدر نیز فشرده شده‌اند.

هر بار که gmodel.py را اجرا می‌کنید، برنامهْ فایل index.sqlite را حذف و دوباره می‌سازد. برنامه به شما اجازه‌ی تغییر در پارامترها و ویرایش جدول‌های مسیردهی را در content.sqlite می‌دهد؛ به این صورت، تمیزکاری دیتابیس می‌تواند بهبود پیدا کند. در اینجا نمونه‌ای از اجرای gmodel.py را مشاهده می‌کنید. برنامهْ بعد از پردازش هر ۲۵۰ پیامِ پستی، یک خط را چاپ می‌کند تا شما مطمئن شوید که پروسه در حال اجراست. اگر به برنامه مهلت کافی بدهید، بعد از مدتی، تقریبا یک گیگابایت داده‌ی مرتبط با مرسولات خواهید داشت.

```
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
...
```

برنامه‌ی gmodel.py تعدادی از امور مرتبط با پاک‌سازی داده‌ها را انجام می‌دهد.

نام دامنه‌ها به دو سطح com.، org.، edu.، و net. تقسیم می‌شوند. بقیه‌ی دامنه‌ها به سه سطح تقسیم می‌شوند. در نتیجه si.umich.edu به umich.edu و caret.cam.ac.uk به cam.ac.uk تبدیل می‌شود. آدرس‌های ایمیل با حروف کوچک نوشته می‌شوند و برخی از آدرس‌های gmane.org@ مانند آدرسِ زیر:

arwhyte-63aXycvo3TyHXe+LvDLADg@public.gmane.org

به یک ایمیل واقعیِ مطابق، اگر در بین اطلاعات موجود باشد، تبدیل می‌شود.

در دیتابیس content.sqlite دو جدول وجود دارد که به شما اجازه می‌دهد نام‌های دامنه و آدرس‌های ایمیل را بازنمایی کنید. به عنوان مثال، Steve Githens آدرس‌های ایمیل زیر را با توجه به تغییر شغل، در لیست توسعه‌دهندگان Sakai داشته است:

```
s-githens@northwestern.edu
sgithens@cam.ac.uk
swgithen@mtu.edu
```

ما می‌توانیم دو مدخل را به جدول مسیردهی در دیتابیس content.sqlite وارد کنیم تا gmodel.py هر سه آدرس را به یک آدرس مسیردهی کند:

```
s-githens@northwestern.edu ->  swgithen@mtu.edu
sgithens@cam.ac.uk -> swgithen@mtu.edu
```

اگر بخواهید چند نام DNS را به یک DNS مسیردهی کنید، می‌توانید همین مدخل‌ها را در جدول DNSMapping بسازید. مسیردهیِ زیر به داده‌های Sakai اضافه شده بود:

```
iupui.edu -> indiana.edu
```

در نتیجه همه‌ی حساب‌های کاربری از دانشکده‌های مختلف ایندیانا با همدیگر رهگیری می‌شوند.

می‌توانید برنامه‌ی gmodel.py را هرچند بار که خواستید اجرا کنید و با اضافه کردن مسیردهی‌ها، داده‌ها را بیشتر و بیشتر تر و تمیز کنید. وقتی کارتان تمام شد یک نسخه‌ی سر و سامان‌دار از ایمیل‌ها در index.sqlite خواهید داشت. این فایل برای آنالیز داده می‌تواند مورد استفاده قرار بگیرد. با استفاده از این فایل، آنالیز داده بسیار سریع انجام خواهد شد.

در ابتدا ساده‌ترین آنالیز داده مشخص کردن «چه کسی بیشترین ایمیل را فرستاده؟» و «چه سازمانی بیشترین ایمیل را فرستاده؟» است. با استفاده از gbasic.py می‌توانید این کار را انجام دهید:

```
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
```

به سرعت اجرای بالاترِ برنامه‌ی gbasic.py نسبت به gmane.py یا حتی gmodel.py دقت کنید. همه روی داده‌های یکسانی کار می‌کنند با این تفاوت که gbasic.py از داده‌های فشرده و نرمال‌شده‌ی index.sqlite استفاده می‌کند. اگر داده‌های زیادی برای توسعه و مدیریت دارید، یک پروسه‌ی چندمرحله‌ای شبیه به چیزی که در این برنامه است، شاید زمان توسعه‌ی بیشتری را برای ساخت نرم‌افزار طلب کند،

ولی در آینده در وقت شما، بابت اکتشاف و مصورسازیِ داده‌هایتان، صرفه‌جویی فراوانی خواهد کرد.

با استفاده فایل gword.py می‌توانید یک تصویرسازیِ ساده، از میزان تکرار کلمه‌ها در عنوانِ ایمیل‌ها، داشته باشید:

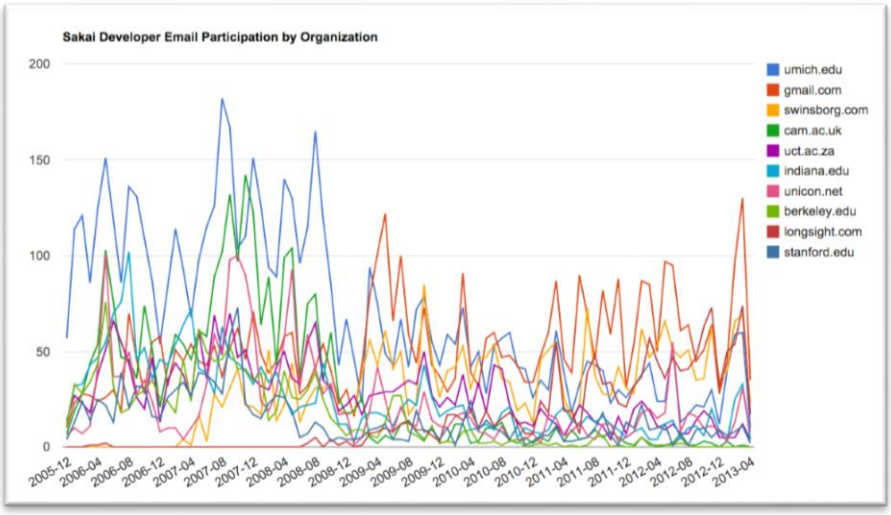
```
Range of counts: 33229 129
Output written to gword.js
```

به این صورت فایل gword.js ساخته می‌شود و با استفاده از gword.htm می‌توانید یک ابرِ کلمه، شبیه به چیزی که در ابتدا این فصل دیدید بسازید.

دومین مصورسازی با استفاده از gline.py است. این برنامه ایمیل‌های مشارکت‌کنندگان را در طول زمان بررسی و محاسبه می‌کند.

```
Loaded messages= 51330 subjects= 25033 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
```

خروجی در فایل gline.js نوشته شده و با استفاده از gline.htm مصورسازی می‌شود.




برنامه‌ی ما، تقریبا پیشرفته و پیچیده است و قابلیت‌هایی برای دریافت، پاک‌سازی و مصورسازی داده‌های واقعی را در خود دارد.