ترفندهای پایتون

بوفه ای از ویژگی‌های فوق‌العاده پایتون

دن بیدر

مترجمین: سید محمد بابازاده و سید معین باباپور

خوانندهی محترم، از خرید و حسن اعتماد شما سپاسگزاریم.

لطفا برای حمایت از حفظ حقوق نویسندگان و مصنفان از هرگونه نشر و کپی غیرمجاز این اثر خودداری کنید. حمایت شما موجب تولید محتوای با کیفیت توسط جامعهی نویسندگان خواهد شد.

در صورتی که این کتاب از روشهای دیگری غیر از وبسایت رسمی مترجمین به دست شما رسیده است، لطفا اقدام به خرید و مطالعهی نسخه قانونی خود کنید.

https://boby.cloud

https://devmo.in

امیدواریم از مطالعهی کتاب به همراه یک فنجان قهوه داغ لذت ببرید.

ترفندهای پایتون

مترجمین:

سید محمد بابازاده

سید معین باباپور

سرشناسه: بادر، دن Bader, Dan
عنوان و نام پدیدآور: ترفندهای پایتون/ [دن بادر]؛
مترجمین سیدمحمد بابازاده، سیدمعین باباپور.
مشخصات نشر: اهواز: پژوهندگان راه دانش، ۱۳۹۹.
مشخصات ظاهری: ۲۹۱ ص.
فروست: پژوهندگان راه دانش؛ شماره نشر ۱۴۹۹.
شابک: ۵-۴۹۶-۳۴۵-۶۰۰-۹۷۸
وضعیت فهرست نویسی: فیپا
یادداشت: عنوان اصلی: .Python tricks : the book, [2017]
موضوع: پایتون (زبان برنامه‌نویسی کامپیوتر) -- دستنامه‌ها
موضوع: Python (Computer program language) -- Handbooks, manuals, etc.
شناسه افزوده: بابازاده، سیدمحمد، ۱۳۷۳ -، مترجم
شناسه افزوده: باباپور، سیدمعین، ۱۳۷۲ -، مترجم
رده بندی کنگره: QA۷۶/۷۳
رده بندی دیویی: ۰۰۵/۱۳۳
شماره کتابشناسی ملی: ۷۲۷۶۰۲۰

انتشارات پژوهندگان راه دانش

ناشر: پژوهندگان راه دانش
شماره نشر: ۱۴۹۹
نام کتاب: ترفندهای پایتون
مترجمین: سیدمحمد بابازاده - سید معین باباپور
طراح جلد: امیرمحمد حسینی پور
تیراژ: ۱۰۰۰
نوبت چاپ: اول ۱۳۹۹
چاپ: ترنگ
صفحه‌آرایی: پژوهندگان
سایز: وزیری
شابک: ۵-۴۹۶-۳۴۵-۶۰۰-۹۷۸
قیمت: ۱۵۰۰۰۰ تومان

website
www.db.ketab.ir

instagram
@Pajopress

دفتر تهران: ۶۶۱۲۰۵۵۴ - ۰۹۱۲۸۸۸۴۹۶۹ دفتر اهواز: ۳۲۲۰۲۸۰۲ - ۰۹۱۶۱۱۸۵۱۱۸

پایتونیست‌ها راجع به کتاب «ترفندهای پایتون» چه می‌گویند؟

من عاشق این کتاب هستم. این کتاب همانند یک آموزش طبقه بندی شده ترفندهای جالب پایتون است. من در حال یادگیری پایتون هستم و با پیش زمینه Powershell وارد پایتون شدم. معمولا زمانی که در نوشتن یک قطعه کد پایتون گیر می‌کنم، یک پست در چت روم داخلی پایتون ارسال می‌کنم و سؤالم را می‌پرسم.

من اغلب از پاسخ‌هایی که از همکارانم دریافت می‌کنم، شگفت زده می‌شوم. ترفندهایی در دیکشنری‌ها، لامبداها، ژنراتورها و ... همواره به عنوان چاشنی در پاسخ دوستانم است. من همیشه تحت تاثیر قرار می‌گرفتم و در عین حال شگفت زده می‌شدم که پایتون چقدر قدرتمند خواهد بود اگر این ترفندها را بشناسم و آن‌ها را به درستی پیاده‌سازی کنم.

این کتاب دقیقا همان کتابی است که به دنبال آن می‌گشتم. من با پیش زمینه Powershell وارد پایتون شدم و نیاز به شخصی داشتم تا از او بپرسم چه زمانی باید از ترفندهایی که جامعه پایتون از آن صحبت می‌کند، استفاده کنم.

به عنوان کسی که مدرک مربط با علوم کامپیوتر ندارد، به یک راهنمای متنی نیاز داشتم تا مواردی را توضیح دهد که افراد دیگر در کلاس‌های دانشگاه علوم مرتبط با کامپیوتر مطالعه کرده‌اند. من از خواندن این کتاب لذت زیادی بردم.

Daniel Meyer
مدیر ارشد شرکت Tesla

برای اولین بار از همکارم در مورد این کتاب شنیدم. او می‌خواست من را با مثال‌های نحوه ساخت دیکشنری‌های پایتون در این کتاب فریب دهد. من حدس می‌زدم نتیجه باید یک دیکشنری کوچکتر باشد اما باید اعتراف کنم که انتظار نتیجه را نداشتم.

همان بعد از ظهر یک نسخه از این کتاب خریدم و شروع به مطالعه کتاب کردم. روز بعد برای نوشیدن یک فنجان قهوه یکی از همکارانم را ملاقات کردم و از همان ترفند برای او استفاده کردم!

او شروع به پرسیدن سؤال‌های متفاوتی کرد اما به دلیل اینکه مطالب در این کتاب به خوبی توضیح داده شده است، نه تنها به راحتی جواب را حدس زدم بلکه به درستی نیز به همکارم توضیح دادم. این یعنی کار بزرگی در توضیح ترفندهای کتاب انجام شده است.

من در پایتون تازه کار نیستم و بعضی از ترفندهای گفته شده در این کتاب برای من جدید نبودند اما باید بگویم که از هر فصل مطالب جدیدی را یاد گرفتم. بابت خواندن این کتاب خوب، خوشحالم و سپاسگزارم که پشت صحنه ترفندها به صورت کامل توضیح داده شده است. من یقینا به همکاران و دوستانم این کتاب را معرفی خواهم کرد.

Og Maciel
توسعه دهنده پایتون شرکت Red Hat

من از خواندن کتاب دن لذت زیادی بردم. او مباحث مهم پایتون را با مثال‌هایی واضح توضیح می‌دهد. (برای مثال گربه‌های دوقلو برای توضیح تفاوت بین == و is در پایتون)

این کتاب فقط نمونه کد نیست. بلکه در رابطه با جزئیات پیاده‌سازی به صورت قابل ملاحظه‌ای بحث شده است. هرچند تنها چیز مهم این است که این کتاب باعث می‌شود کدهای پایتون بهتری بنویسید.

ترفندهای پایتون این کتاب، عادت‌های جدیدی در کدنویسی من به وجود آورد. برای مثال استفاده از exception های اختصاصی و AbstractBaseClasses باعث کدنویسی بهتر من شد، چیزهایی که به تنهایی یادگرفتنشان سخت بود.

Bob Belderbos
مهندس ارشد شرکت Oracle

مقدمه مترجمین

اگر درک کدهایتان پس از مدتی دشوار است، باید به این نکته توجه کنید که «*سادگی، نهایت پیچیدگی است*». اغلب اوقات پیچیده انگاشتن یک موضوع، احساس قدرت به افراد می‌دهد. در همین راستا مارتین فاولر، می‌گوید:

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."

این کتاب بوفه‌ای از ترفندهای پایتون است که باعث می‌شود درک بهتری از ویژگی‌های زبان پایتون کسب کنید. در این کتاب روش‌هایی برای پایتونیک نوشتن کدها به همراه مثال‌هایی کاربردی بیان شده است که در عین سادگی، باعث افزایش توانایی شما در کدنویسی می‌شود.

از آنجا که زبان انگلیسی، زبان دنیای کامپیوتر است، در ترجمه‌ی این کتاب سعی شده است تا واژه‌ها و نام‌های مختلف یک عبارت خاص بیان شود. در این کتاب از تکنیکِ ترجمه‌ی ارتباطی استفاده شده است و هیچ کدام از مفاهیم به صورت تحت‌اللفظی ترجمه نشده‌اند. دلیل این کار انتقال راحت مفاهیم به خواننده و صدمه ندیدن واژه‌های انگلیسی و تکنیکال است.

امیدواریم از خواندن این کتاب لذت ببرید و در صورت ارائه‌ی هرگونه پیشنهاد و انتقاد در راستای بهتر شدن این کتاب، صمیمانه پذیرای نظرات شما هستیم.

سید محمد بابازاده[1] - ایمیل: hi@boby.cloud

سید معین باباپور[2] - ایمیل: hi@devmo.in

1 https://boby.cloud
2 https://devmo.in

فهرست مطالب

فصل اول

معـرفـی

۱-۱. ترفندهای پایتون چیست؟

ترفندهای پایتونی یک قطعه کوتاه کد از زبان برنامه نویسی پایتون است که به عنوان ابزاری برای آموزش مفاهیم عمیق استفاده می‌شود. ترفند پایتون جنبه ای از پایتون را با یک مثال ساده آموزش می دهد. ترفند پایتون انگیزه ای در اختیار شما قرار می‌دهد و شما را قادر می سازد تا در زبان برنامه نویسی پایتون عمیق تر حفاری کنید و یک درک شهودی برای خود ایجاد کنید.

ترفندهای پایتونی به عنوان یک سری کوتاه تصاویر از کدها شروع شد که من در یک هفته در توییتر به اشتراک گذاشتم. در کمال تعجب من، افراد بازخوردهای شگفت انگیزی نشان دادند و برای روزهای متوالی تکه کدهای من توسط افراد مختلف ریتوییت می‌شد.

هرروز توسعه دهندگان بیشتری از من سؤال می‌کردند که آیا روشی برای دسترسی به کل سری آموزشی وجود دارد یا نه. درواقع، من فقط چند مورد از این ترفندها را ارائه کرده بودم که موضوعات مختلف مربوط به زبان برنامه نویسی پایتون را شامل می‌شد. پشت سری آموزشی من یک طرح جامع وجود نداشت و فقط یک آزمایش جالب برای تفریح در توییتر بود. اما این سؤالات از تکه کدهای من باعث شد احساس کنم کدهای کوتاه و کاربردی من به عنوان ابزاری برای تدریس عمیق زبان پایتون، ارزش سرمایه‌گذاری و کاوش را دارد. سرانجام تصمیم گرفتم چند ترفند پایتونی دیگر تهیه کنم و آن‌ها را در قالب یک سری ایمیل با افراد به اشتراک بگذارم. در طی چند روز، صدها توسعه دهنده پایتون در اشتراک ایمیل من ثبت نام کردند و این بازخورد برای من بسیار شگفت انگیز بود.

طی روزها و هفته های بعد، تعداد بسیار زیادی از توسعه دهندگان پایتون با من آشنا شدند. آن‌ها از من تشکر می‌کردند زیرا ترفندهای من باعث شده بود توسعه دهندگان،

بخش هایی از زبان برنامه نویسی پایتون که در حال دست و پنجه نرم کردن با مفاهیم آن بودند را به خوبی درک کنند. من فکرمی‌کردم این ترفندها فقط یک سری تصاویر ساده از تکه کدهای پایتون هستند. اما بسیاری از توسعه دهندگان از طریق این کدها مفاهیم عمیقی را آموختند و مشاهده این بازخوردها برای من بسیار خوشحال‌کننده بود.

پس از این من تصمیم گرفتم تجربیاتم را در دیگر زمینه های ترفندهای پایتون افزایش دهم و آن را در یک سری ۳۰ عددی ایمیل به اشتراک بگذارم. هرکدام از این ترفندها دارای یک عنوان و یک تصویر از کد بود و من به زودی متوجه محدودیت‌های اشتراک گذاری در این قالب شدم. در آن زمان یک توسعه دهنده نابینا پایتون به من ایمیل داد و از اینکه نمی‌توانست این تصاویر را برای خود بخواند ناامید شده بود.

واضح است برای اینکه من بتوانم محتوا را در دسترس مخاطبان گسترده تری قرار دهم باید زمان بیشتری را روی آن صرف می‌کردم. بنابراین، من نشستم تا دوباره تمام ایمیل‌های ترفندهای پایتون را با متن ساده و با برجسته سازی مبتنی بر HTML بازنویسی کنم. این بازنویسی باعث شد پاسخ هایی را دریافت کنم و توسعه دهندگان از اینکه می‌توانستند تکه کدها را Copy/Paste کنند و با آن کار کنند، خوشحال بودند.

هرچه توسعه دهندگان بیشتری در این سری ایمیل ثبت نام می‌کردند، از طریق دریافت بازخوردها متوجه الگوی خاصی شدم. برخی از ترفندها بسیار انگیزشی بودند و افراد مفاهیم عمیق را به خوبی متوجه می‌شدند، اما برای ترفندهای پیچیده تر راهنمایی وجود نداشت که منابع بیشتری را در اختیار توسعه دهندگان مشتاق قرار دهد و به درک عمیق‌تر آن‌ها از زبان برنامه نویسی پایتون جامه عمل بپوشاند.

بیایید فقط بگوییم که این یکی دیگر از رسالت‌های من در وب‌سایت realpython.com بود که به کمک آن بتوانم توسعه دهندگان بیشتری را با مفاهیم پیچیده و عمیق پایتون آشنا کنم. بنابراین من تصمیم گرفتم بهترین و ارزشمندترین ترفندهای پایتون را از سری ایمیل‌ها جمع آوری کنم و با توضیحات بیشتر در قالب یک کتاب ارزشمند منتشر کنم.

- کتابی که با نمونه کدهای کوتاه و قابل هضم که جالب ترین جنبه های زبان برنامه نویسی پایتون را آموزش می‌دهد.
- کتابی که مانند بوفه ای از ویژگی های بسیار جذاب زبان برنامه نویسی پایتون عمل می‌کند و می‌تواند سطح انگیزه شما را در هنگام توسعه پایتون بالا ببرد و هربخش به صورت جداگانه قابل مطالعه باشد.
- کتابی که دست شما را می‌گیرد و شما را به اعماق زبان برنامه نویسی پایتون می‌برد.

این کتاب شامل آزمایشات و تجربیات بسیار من در زبان برنامه نویسی پایتون است و از عشق من به این زبان برنامه نویسی به وجود آمده است. امیدوارم که شما هم از خواندن این کتاب لذت ببرید و ترفندهای جدیدی را در رابطه با زبان برنامه نویسی پایتون یاد بگیرید.

۱-۲. این کتاب برای شما چه کاری انجام می‌دهد؟

این کتاب تکنیک هایی را به شما می‌آموزد تا تبدیل به یک توسعه دهنده زیرک و با بصیرت در پایتون شوید. شاید تعجب کنید که چگونه خواندن این کتاب باعث می‌شود تا به این موارد دست یابیم؟

ترفندهای پایتون یک آموزش گام به گام پایتون نیست. یک آموزش مقدماتی برای ورود به زبان برنامه نویسی پایتون نیست. اگر در مراحل ابتدایی یادگیری زبان برنامه نویسی پایتون هستید، این کتاب به تنهایی شما را تبدیل به یک توسعه دهنده حرفه ای پایتون نمی‌کند. خواندن این کتاب برای شما سودمند خواهد بود اما باید اطمینان حاصل کنید که با مطالعه منابع دیگر مهارت های بنیادین زبان برنامه نویسی پایتون را یاد بگیرید.

اگر شما قبلا دانش زبان برنامه نویسی پایتون را یاد گرفته اید و می‌خواهید به سطح بعدی بروید به شما تبریک می‌گویم. زیرا می‌توانید بیشترین بهره را از این کتاب ببرید.

این کتاب برای کسانی که قبلا برای مدتی با زبان پایتون کار کرده‌اند و اکنون می‌خواهند عمیق تر با این زبان برنامه نویسی آشنا شوند تا کدهایشان را سریع تر و بهینه تر بنویسند، بسیار مناسب است و می‌تواند باعث شود کدهای پایتونیک بهتری بنویسند.

۱-۳. این کتاب را چگونه مطالعه کنیم؟

بهترین راه برای خواندن کتاب ترفندهای حرفه ای پایتون آن است که با این کتاب همانند یک بوفه ای شامل از ویژگی های فوق العاده پایتون رفتار کنید. هر ترفند موجود در این کتاب به صورت جداگانه توضیح داده شده است و شما می‌توانید مستقیما به مبحثی که نیاز دارید بروید و آن را مطالعه کنید. من هم شما را به انجام این کار توصیه می‌کنم.

البته شما می‌توانید تمام ترفندهای پایتون را به ترتیبی که در کتاب ذکر شده نیز بیاموزید. به این ترتیب هیچ یک از ترفندها را از دست نخواهید داد و وقتی به صفحه آخر کتاب برسید با تمام ترفندهای حرفه ای زبان برنامه نویسی پایتون آشنا شده اید.

فهمیدن بسیاری از این ترفندها آسان است و هیچ مشکلی نخواهید داشت. شما بلافاصله بعد از خواندن و یادگیری یک ترفند می‌توانید از آن در کار روزانه خود استفاده کنید. اما برای بعضی از ترفندهای پیچیده ممکن است نیاز به کمی زمان برای کاوش و تمرین داشته باشید.

فصل دوم

کدنویسی تمیز با پایتون

۲-۱. ویرگول‌های آشفته

یک نکته مفید برای اضافه یا حذف کردن آیتم‌ها از لیست، دیکشنری یا ست در پایتون این است که حتما تمام خطوط خود را با کاما (ویرگول) تمام کنید.

مطمئن نیستید که راجع به چه چیزی صحبت می‌کنیم؟ بگذارید یک مثال ساده بزنیم. فرض کنید که یک لیست از نام های مختلف در کد دارید.

```
>>> names = ['Alice', 'Bob', 'Dilbert']
```

زمانی که روی نام‌های این لیست تغییر ایجاد کنید، خیلی سخت است با مشاهده خروجی دستور git diff بفهمید نام‌ها چه تغییراتی کردند. زیرا اکثر سیستم های کنترل کد منبع بر اساس شماره خط کد هستند و به سختی می‌توانند تغییرات در یک خط واحد را برجسته کنند.

یک راه سریع حل این مشکل، اتخاذ یک سبک کد نویسی جدید است که در آن آیتم‌های لیست، دیکشنری یا ست در پایتون را در چند خط متوالی بنویسید. همانند مثال زیر.

```
>>> names = [
    'Alice',
    'Bob',
    'Dilbert'
]
```

اگر لیست خود را به صورت بالا تعریف کنید، در هر خط یک مورد وجود دارد و زمانی که در سیستم کنترل کد منبع خود تغییرات را مشاهده کنید، به راحتی می‌توانید متوجه شوید کدام نام ها اضافه، حذف یا اصلاح شده اند. این یک تغییر کوچک در سبک کد نویسی است اما تجربه من نشان می‌دهد از اشتباهات احمقانه جلوگیری می‌کند. اشتباهاتی که ممکن است برطرف شدن آن ساعت ها زمان ببرد. همچنین این تغییر باعث شد هم‌تیمی های من بتوانند راحت تر کدهای من را بررسی کنند.

اکنون دو مورد دیگر وجود دارد که ممکن است باعث سردرگمی شود. هر وقت یک مورد جدید را در انتهای لیست اضافه کنید یا آخرین مورد را حذف کنید باید به صورت مداوم جای ویرگول را به روز رسانی کنید.

۲-۱-۱. بهم ریختن کد با ویرگول‌های نامناسب

فرض کنید می‌خواهیم نام دیگری (Jane) را به لیست اضافه کنیم، اگر Jane را اضافه کنید، مجبور هستید بعد از Dilbert یک کاما قرار دهید و اگر فراموش کنید، با نتیجه‌ای غیر قابل انتظار روبرو می‌شوید.

```
>>> names = [
    'Alice',
    'Bob',
    'Dilbert' # <- Missing comma!
    'Jane'
]
```

اگر قرار دادن علامت ویرگول را فراموش کنید، ممکن است هنگام اجرای کد متعجب شوید.

```
>>> names
['Alice', 'Bob', 'DilbertJane']
```

همانطور که مشاهده می‌کنید، پایتون در زمان اجرا، رشته های Dilbert و Jane را به DilbertJane تبدیل کرده و در واقع آن‌ها را بهم چسباند. به این عمل در پایتون جمع بندی دقیق رشته‌ها[1] می‌گویند که یک رفتار طبیعی در زبان پایتون است. اما می‌تواند گاهی اوقات با ایجاد مشکلاتی در برنامه پایتونی، شما را تا مرز پرتاب کردن مانیتور از پنجره به بیرون پیش ببرد.

با این وجود، جمع بندی دقیق رشته ها در پایتون بسیار مفید است. برای مثال، می‌توانید از این روش برای زمانی استفاده کنید، که می‌خواهید یک رشته متن طولانی

1 String Literal Concatenation

را در برنامه خود وارد کنید اما نمی خواهید که از علامت بک اسلش (یا \) استفاده کنید.

```
my_str = ('This is a super long string constant '
          'spread out across multiple lines. '
          'And look, no backslash characters needed!')
```

از طرف دیگر همین الان دیدیم که این قابلیت مفید، چطور می‌تواند به یک بحران در یک برنامه تجاری تبدیل شده و باعث ایجاد باگ‌هایی عجیب شود و به یک بدهی فنی تبدیل شود. حال چگونه این مشکل را حل کنیم؟

اضافه کردن کامای گم شده، پس از Dilbert باعث می‌شود جلوی این مشکل گرفته شده و دو رشته Dilbert و Jane از یکدیگر جدا شوند.

```
>>> names = [
    'Alice',
    'Bob',
    'Dilbert',
    'Jane'
]
```

اما اکنون دور یک حلقه چرخیدیم و دوباره به نقطه ابتدایی بازگشتیم. برای اضافه کردن یک نام جدید به لیست مجبور شدیم دو خط از کد را تغییر دهیم. این موضوع باعث می‌شود دیدن تغییرات کد منبع در خروجی دستور git diff سخت تر شود. آیا نام جدیدی اضافه شده است یا نام Dilbert تغییر کرده است؟

خوشبختانه راه های پایتونیک باعث می‌شوند تا مشکل قرار دادن کاما را یک بار و برای همیشه حل کنیم. برای حل دائمی این مشکل، باید خود را آموزش دهید تا یک سبک کدنویسی مشخص را اتخاذ کنید تا در وهله ی اول جلوی این ایجاد این مشکل گرفته شود. صبر کنید تا به شما نشان دهیم چطور می‌توانید این کار را انجام دهید.

در پایتون، می‌توانید ویرگول را بعد از هر مورد در لیست، ست یا دیکشنری قرار دهید از جمله آخرین مورد موجود در کالکشن‌های نام برده شده. بنابراین به خود

آموزش دهید که همواره به یاد داشته باشید انتهای خطوط خود را با ویرگول به اتمام برسانید و از اینکه در خط انتهایی ویرگول قرار ندهید، پرهیز کنید.

برای مثال به قطعه کد زیر نگاه کنید.

```
>>> names = [
    'Alice',
    'Bob',
    'Dilbert',
]
```

بعد از اسم Dilbert ویرگول را مشاهده می‌کنید؟ این کار باعث می‌شود بدون نیاز به جایگذاری مجدد ویرگول، بتوانید لیست خود را آپدیت کنید و خطوط کد را ثابت نگه دارید. این روش، کنترل کد منبع شما را ساده‌تر کرده و افراد، کدهای شما را خوشحال‌تر بازبینی می‌کنند. گاهی اوقات جادو در چیزهای کوچک است، درست است؟

۲-۱-۲. نکات کلیدی در هنگام کار با ویرگول‌ها در پایتون

- اصلاح قالب بندی کد و قرار دادن کاما باعث می‌شود لیست، ست یا دیکشنری در برنامه‌ی شما قابلیت مدیریت راحت تری داشته باشد.
- جمع بندی دقیق رشته‌های پایتون ممکن است فواید زیادی برای شما داشته باشد. اما همچنین ممکن است باعث به وجود آمدن باگ هایی شود که به سختی قابل حل هستند.

۲-۲. Assertion را پوششی برای هرچیزی نکنید!

گاهی اوقات یک ویژگی مفید و کاربردی در یک زبان برنامه نویسی مورد توجه کمتری نسبت به سایر ویژگی‌ها قرار می‌گیرد. این اتفاقی است که برای عبارت assert در پایتون افتاده است.

در این بخش می‌خواهیم مقدمه ای برای استفاده از عبارت assert در پایتون را بیان کنیم. شما می‌آموزید که چطور از عبارت assert برای آشکارسازی خودکار خطاها در کد پایتون استفاده کنید. این ویژگی باعث می‌شود برنامه ی شما قابلیت اطمینان بیشتری داشته باشد و به راحتی قابل دیباگ باشد.

در این مرحله ممکن است برای شما سؤال ایجاد شود که عبارت assertion دقیقا چه چیزی هست و برای استفاده در چه جاهایی مناسب است؟ بیایید به این سؤال اساسی پاسخ دهیم.

در هسته زبان برنامه نویسی پایتون، عبارت assert یک ابزار مفید برای دیباگ کدها بوده که یک شرایط خاصی را آزمایش می‌کند. اگر مقدار عبارت assert برابر true شود، اتفاقی رخ نمی‌دهد و برنامه به خوبی به اجرای خود ادامه می‌دهد. اگر مقدار عبارت assert برابر false شود، خطای AssertionError رخ داده و شما به راحتی خطاهای برنامه ی خود را پیدا خواهید کرد.

کلمه assert در لغت به معنی ادعا کردن و دفاع کردن است. بنابراین در کدهای خود ادعا می‌کنید که شرایط خاصی باید برقرار باشد، در غیر این صورت با ایجاد یک خطا توسعه دهنده را مطلع خواهید کرد.

۲-۲-۱. Assert در پایتون با یک مثال ساده

در اینجا یک مثال ساده وجود دارد تا ببینید Assertion در کجا می‌تواند مفید باشد. من سعی کردم جنبه ای از یک مشکل واقعی را که شما واقعاً در یکی از برنامه های خود با آن روبرو می‌شوید، بیان کنم.

فرض کنید شما در حال ساختن یک فروشگاه آنلاین با پایتون هستید. شما در حال توسعه هستید تا یک تابع تخفیف کوپن را به سیستم اضافه کنید و چنین تابعی را به همین منظور می‌نویسید.

```
def apply_discount(product, discount):
    price = int(product['price'] * (1.0 - discount))
    assert 0 <= price <= product['price']
    return price
```

به عبارت assert در کد بالا توجه کردید؟ نرخ تخفیف محاسبه شده توسط این تابع نمی‌تواند کمتر از صفر دلار و بیشتر از قیمت اصلی محصول باشد. اگر از این قابلیت برای اعمال تخفیف معتبر استفاده کردیم اکنون باید اطمینان حاصل کنیم که فرآیند مورد نظر به درستی انجام می‌شود.

برای مثال محصولات فروشگاه به صورت دیکشنری هایی ساده ذخیره می‌شوند. شاید این روشی نباشد که در یک برنامه واقعی به کار رود، اما می‌توان برای اثبات عملکرد Assertion از این روش استفاده کرد. بنابراین یک محصول جدید می‌سازیم که شامل ۲ مقدار نام محصول (یک جفت کفش) و قیمت محصول (۱۴۹۰۰ دلار) است.

```
>>> shoes = {'name': 'Fancy Shoes', 'price': 14900}
```

اکنون بیایید یک تخفیف ۲۵ درصدی به محصول اعمال کنیم و انتظار داریم که قیمت محصول معادل ۱۱۱,۷۵ دلار شود.

```
>>> apply_discount(shoes, 0.25)
11175
```

بسیارعالی، اکنون بیایید یک مقدار تخفیف نامعتبر وارد کنیم. برای مثال، من ۲۰۰٪ تخفیف به محصول اعمال می‌کنم.

```
>>> apply_discount(shoes, 2.0)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    apply_discount(prod, 2.0)
  File "<input>", line 4, in apply_discount
    assert 0 <= price <= product['price']
AssertionError
```

همانطور که مشاهده می‌کنید، زمانی که سعی کردیم از یک تخفیف نامعتبر استفاده کنیم، نرم افزار با Assertion Error متوقف می‌شود. دلیل این امر آن است که در تابع مورد نظر، حاصل عبارت assert نقض شده و مقدار آن برابر با False شده است.

در اطلاعات خروجی خطا، می‌توانید به صورت دقیق ببینید چه مشکلی در برنامه شما رخ داده است. برای مثال اگر شما یا یک نفر از توسعه دهندگان تیم با این خطا رو به رو شود به سرعت می‌تواند دلیل به وجود آمدن مشکل را متوجه شود.

با این کار سرعت شما در رفع مشکلات برنامه بیشتر خواهد شد و می‌توانید برنامه های تجاری خود را در بلند مدت به راحتی مدیریت کنید. دوست من، این قدرت assertion است!

۲-۲-۲. چرا از عبارات منظم[1] استفاده نکنیم؟

اکنون ممکن است از خود سؤال کنید که چرا در مثال قبل تنها یک شرط if برای کنترل فرآیند نگذاشتیم؟ آیا ساده تر نبود؟

همانطور که احتمالا متوجه شده‌اید، هدف استفاده از assertion آگاه سازی توسعه دهندگان برای کشف خطاهایی هست که در برنامه کشف نشده اند. Assertion روشی برای نشان دادن حالت های مختلف خطای برنامه همانند file not found error نیست.

اگر برنامه شما فاقد هیچگونه اشکالی باشد پس این شرایط هرگز در برنامه شما اتفاق نمی‌افتد! امااگر اشکالی در برنامه شما به وجود آمد، برنامه با یک assertion error متوقف خواهد شد و برنامه به شما می‌گوید دقیقا در کدام قسمت یک اتفاق غیر منتظره افتاده است. این کار رفع خطای برنامه را ساده تر می‌کند. من عاشق چیزهایی هستم که زندگی را راحت تر می‌کنند. شما چطور؟

در حال حاضر به یاد داشته باشید که عبارت assert در پایتون روشی برای debug است نه برای مقابله با خطاهای زمان اجرا. هدف اصلی assertion این است که توسعه

1 Regular Expressions

دهندگان به سرعت متوجه ریشه پدید آمدن خطا شوند و خطا را سریع تر رفع کنند. یک assertion error در کد شما رخ نمی‌دهد مگر در حالتی که برنامه‌ی نوشته شده دچار خطا شود.

بیایید یک نگاه نزدیک تر به کارهایی که می‌توانیم با assertion انجام دهیم بیندازیم و سپس دو اشتباه متداول هنگام استفاده از این عبارت در دنیای واقعی را بررسی کنیم.

۲-۲-۳. سینتکس assert در پایتون

همیشه بهتر است قبل از استفاده از یک قابلیت در یک زبان برنامه نویسی، ببینیم این قابلیت چطور پیاده سازی شده است. با توجه به مستندات رسمی پایتون، نگاهی می‌اندازیم به نحوه پیاده سازی عبارت assert در این زبان برنامه نویسی.

```
assert_stmt ::= "assert" expression1 ["," expression2]
```

در عبارت فوق، expression1 شرایطی هست که آن را تست می‌کنیم و مقدار اختیاری expression2 پیامی است که می‌خواهیم زمانی که خطا رخ داد نمایش دهیم. در زمان اجرای برنامه پایتونی، مفسر پایتون هرکدام از expression ها را به صورت زیر پردازش می‌کند.

```
if __debug__:
    if not expression1:
        raise AssertionError(expression2)
```

دو نکته جالب در رابطه با کد فوق

۱. قبل از اینکه عبارت assert چک شود، متغیر سراسری __debug__ چک می‌شود. این متغیر، زمانی که برنامه در حالت توسعه است برابر با true و هنگامی که برنامه برای محیط پروداکشن بهینه سازی می‌شود برابر با false است.

همچنین شما می‌توانید از مقدار expression2 استفاده کنید تا یک پیام خطا اختصاصی نمایش دهید. این کار رفع مشکل را راحت تر می‌کند. برای مثال به قطعه کد زیر دقت کنید.

```
>>> if cond == 'x':
        do_x()
    elif cond == 'y':
        do_y()
    else:
        assert False, (
            'This should never happen, but it does '
            'occasionally. We are currently trying to '
            'figure out why. Email dbader if you '
            'encounter this in the wild. Thanks!')
```

این پیام زشت است؟ موافقم، اما بسیار کاربردی است زمانی که شما با یک خطای هایزنباگ[1] در برنامه خود مواجه می‌شوید.

۲-۲-٤. اشتباهات متداول هنگام استفاده از عبارت assert در پایتون

قبل از ادامه، دو خطای متداول در مورد استفاده از عبارت assert در پایتون وجود دارد که می‌خواهم راجع به آن‌ها توضیح دهم.

مورد اول باعث مخاطرات امنیتی در برنامه شما می‌شود و مورد دوم یک سینتکس عجیب و غریب است که نوشتن asertion ها را بی فایده می‌کند.

رخ دادن این اشتباهات کاملاً وحشتناک است بنابراین به هشدارهای زیر توجه کنید.

۲-۲-۵. هشدار اول: از assert برای اعتبارسنجی داده ها استفاده نکنید

بزرگترین اشتباه رایج استفاده از عبارت assert برای اعتبار سنجی داده ها است. همانطور که می‌دانید در محیط‌های پروداکشن اجرای برنامه با استفاده از سوییچ O- و OO- عبارت assertion به صورت سراسری غیرفعال می‌شود. همانطور که متغیر

1 Heisenbug

محیطی PYTHONOPTIMIZE برای بهینه سازی برنامه به صورت سراسری در CPython فعال می‌شود.

این کار باعث می‌شود تمام عبارت های assertion برابر با null-operation یا فرآیندهای خالی شوند. عبارت های assert به سادگی جمع آوری شده و دیگر اجرا نمی‌شوند. این موضوع به این معنی است که هیچ کدام از عبارات شرطی درون assert دیگر اجرا نخواهند شد.

این تصمیم در زمان طراحی داخلی زبان برنامه نویسی پایتون گرفته شده است، همانطور که در زبان های برنامه نویسی دیگر نیز وجود دارد. بنابراین، استفاده از عبارات شرطی در assert برای اعتبار سنجی داده ها کاری بسیار خطرناک است.

بگذارید کمی بیشتر توضیح دهم. اگر برنامه شما از یک عبارت assert استفاده می‌کند تا مقدار درست یا غلط بودن خروجی یک تابع را بررسی کند، می‌تواند به سرعت تبدیل به یک آتش سوزی در کدهای شما شود و خطاها یا نقص های امنیتی مهلکی را به وجود بیاورد.

یک مثال ساده را برای بررسی این مشکل مطرح می‌کنیم. یک بار دیگر تصور کنید که شما با استفاده از پایتون یک برنامه فروشگاه آنلاین را ایجاد کرده‌اید. در جایی از برنامه تکه کدی وجود دارد که کاربر می‌تواند محصولی را در فروشگاه حذف کند. از آنجا که شما به تازگی عبارات assert را آموخته اید، مشتاق هستید که از آن‌ها در کدهای خود استفاده کنید و پیاده سازی که در زیر می‌بینید را برای حذف محصولات انجام می‌دهید.

```
def delete_product(prod_id, user):
    assert user.is_admin(), 'Must be admin'
    assert store.has_product(prod_id), 'Unknown
product'
    store.get_product(prod_id).delete()
```

به تابع delete_product نگاه دقیق تری بیندازید. اکنون اگر assertion ها در زمان اجرا غیرفعال شوند چه اتفاقی رخ می‌دهد؟ در ۳ خط کد فوق، ۲ مشکل بسیار جدی وجود دارد و به دلیل استفاده اشتباه از عبارات assert پدید آمده است.

۱. چک کردن مجوز مدیرکل با استفاده از assert یک خطر مهلک است. اگر assertion ها غیرفعال شوند این شرط به یک شرط null تبدیل می‌شود. اکنون هرکاربری می‌تواند محصولات را حذف کند! بررسی مجوز هرگز اجرا نمی‌شود و یک در پشتی (backdoor) برای نفوذگران باز شده که بتوانند تمام محصولات برنامه را پاک کنند و به یکپارچگی داده ها آسیب برسانند.

۲. تابع has_product به دلیل غیرفعال شدن assertion ها اجرا نمی‌شود. این موضوع به این معنی است که تابع get_product اکنون بدون مقدارهای معتبر prod_id می‌تواند فراخوانی شود. بنابراین با اینکه محصول وجود ندارد اما تابع get_product به دلیل اشتباه در استفاده از assertion فراخوانی می‌شود! این مخاطره امنیتی می‌تواند باعث حمله DDoS در برنامه شما شود. زیرا یک نفوذگر می‌تواند داده های اشتباه بسیار زیادی را برای حذف کردن محصول به تابع get_product بفرستد و باعث از دسترس خارج شدن برنامه گردد.

اکنون اگر از خود می‌پرسید چطور جلوی این مشکلات را بگیریم، پاسخ بسیار ساده است. از assertion برای اعتبارسنجی داده ها استفاده نکنید. درعوض می‌توانید با استفاده از عبارت شرطی if شروط موردنظر را چک کنید و در صورتی که شرط ها نقض شدند، یک خطای اختصاصی تولید کنید.

```
def delete_product(product_id, user):
    if not user.is_admin():
        raise AuthError('Must be admin to delete')
    if not store.has_product(product_id):
        raise ValueError('Unknown product id')
    store.get_product(product_id).delete()
```

در قطعه کد بالا، در زمان مناسب خطاهای AuthError و ValueError نمایش داده خواهند شد و شروط مورد نظر به خوبی کنترل خواهند شد.

۲-۲-۶. هشدار دوم – assert هایی که همیشه درست هستند

جای تعجب است اما هنگامی که به صورت تصادفی عبارت assert را در پایتون بنویسید همواره به درستی ارزیابی می‌شود. من در گذشته به دلیل این اشتباه زمان زیادی را هدر دادم. به طور خلاصه مشکل به صورت زیر است.

زمانی که یک Tuple را به عنوان اولین پارامتر عبارت assert وارد می‌کنید، این عبارت همواره صحیح درنظر گرفته می‌شود. برای مثال قطعه کد زیر هیچگاه خطا نخواهد داد.

```
assert(1 == 2, 'This should fail')
```

این مشکل به این دلیل است که عبارات tuple که خالی نیستند همیشه به عنوان true در زبان پایتون درنظر گرفته می‌شوند. زمانی که شما یک tuple غیر خالی را به عبارت assert در پایتون می‌دهید، کاری کاملاً بی فایده انجام داده‌اید، زیرا هیچگاه خطایی تولید نخواهد شد و حاصل عبارت همواره صحیح است.

یک مثال دیگر را باهمدیگر می‌بینیم. فرض کنید من با خوشحالی تعدادی تست نوشته ام و حس ایمنی کاذبی را در کدهایم ایجاد کرده ام. تصور کنید از assertion زیر در یکی از تست ها استفاده کرده ام.

```
assert (
    counter == 10,
    'It should have counted all the items'
)
```

در نگاه اول تست کاملاً درست به نظر می‌رسد. با این حال، هرگز نتیجه درستی نخواهد داشت. مقدار assertion همواره برابر با True ارزیابی می‌شود. چرا این اتفاق

می‌افتد؟ زیرا assert ارزیابی می‌کند که آیا tuple دارای مقدار است یا نه و از آن جایی که دارای مقدار است، پس نتیجه ارزیابی همواره True است.

همانطور که گفتم، زمانی که متوجه این حقیقت شدم میخواستم خودکشی کنم!

روشی که می‌توانید با این مشکل مقابله کنید استفاده از code linter است. نسخه های جدید پایتون نیز دارای پیام اخطار برای این روشِ استفاده از assertion هستند.

به هرحال، همواره دقت کنید تست های برنامه باید ابتدا شکست بخورند و سپس کد مربوط به تست، نوشته شود. در غیر این صورت ممکن است دچار اشتباهات مهلکی شوید همانند موردی که بررسی کردیم.

۲-۲-۷. Assertion در پایتون - خلاصه

با وجود هشدارهایی که بررسی کردیم، من فکرمی‌کنم عبارت assertion در پایتون ابزاری قوی برای رفع مشکلات کد است. اما متاسفانه بعضی از توسعه دهندگان از این عبارات استفاده نمی‌کنند.

یادگرفتن طریقه عملکرد assertion و استفاده از آن در کدهایتان، بهترین روش برای مدیریت آسان کد در بلند مدت است. زیرا کدهای یک برنامه تجاری در بلند مدت دچار پیچیدگی های زیادی خواهند شد و مدیریت آن بدون یک ابزار آشکارسازی مشکل، بسیار دشوار و طاقت فرسا است.

با یادگیری و استفاده از assertion می‌توانید مهارت خود را در زبان برنامه نویسی پایتون به مرحله بعد ببرید و تبدیل به یک پایتونیست حرفه‌ای شوید. من مطمئن هستم که استفاده از این ویژگی، ساعت های زیاد از زمان من در هنگام رفع مشکل را نجات خواهد داد.

۲-۲-۸. نکات کلیدی Assertion در پایتون

- عبارت assert در پایتون یک ابزار رفع اشکال قوی است که می‌تواند یک سری شرایط را در برنامه شما چک کند.

- عبارت assert باید تنها برای توسعه دهندگان باشد و استفاده از این عبارت روش مناسبی برای مدیریت خطاهای زمان اجرای برنامه نیست.
- عبارت assert می‌تواند به صورت سراسری در تنظیمات مفسر پایتون غیرفعال شود.

۲-۳. عبارت with و context manager در پایتون

عبارت with در زبان برنامه نویسی پایتون معمولا یک عبارت مرموز درنظر گرفته می‌شود. اما زمانی که به پشت صحنه نگاه می‌کنید، می‌بینید که هیچ جادویی در کار نیست. عبارت with در واقع یک ویژگی جذاب و مفید است که به شما کمک می‌کند تا کد پایتونی تمیزتر و خواناتری داشته باشید.

ممکن است سؤال کنید عبارت with برای چه زمان هایی استفاده می‌شود؟ عبارت with زمانی استفاده می‌شود که بخواهیم با استفاده از الگوی استانداردی، به مدیریت منابع به صورت بهینه بپردازیم.

برای روشن تر شدن موضوع، به سراغ مثالی از کتابخانه open که یکی از کتابخانه های درونی پایتون است می‌رویم. کتابخانه open یکی از بهترین کتابخانه ها برای توضیح یک مثال در رابطه با عبارت with در پایتون است.

```
with open('hello.txt', 'w') as f:
    f.write('hello, world!')
```

به طور کلی توصیه می‌شود فایل ها را با استفاده از عبارت with در زبان برنامه نویسی پایتون باز کنید. این کار باعث می‌شود زمانی که کارِ برنامه با فایل مورد نظر به اتمام می‌رسد، ارتباط برنامه با فایل به صورت خودکار بسته شود. در حقیقت زمانی که از عبارت with استفاده می‌کنید کدهای شما به صورتی که مشاهده می‌کنید، توسط مفسر پایتون ترجمه می‌شود.

```
f = open('hello.txt', 'w')
try:
    f.write('hello, world')
finally:
    f.close()
```

ممکن است بگویید این کد کمی طولانی است و کاملاً درست است. به این نکته توجه کنید که استفاده از عبارت try ... finally روشی موثر برای مدیریت خطا است. برای مثال اگر برای این کار، کدی شبیه به زیر بنویسید، دچار مشکلاتی خواهید شد.

```
f = open('hello.txt', 'w')
f.write('hello, world')
f.close()
```

پیاده سازی فوق تضمین نمی‌کند که هنگام نوشتن در فایل، اگر با خطایی مواجه شدید پس از آن ارتباط با فایل توسط برنامه بسته شود و ممکن است برنامه با خطاهای متفاوتی روبرو شود. به همین دلیل است که در این مواقع از عبارت with در پایتون استفاده می‌کنیم. با استفاده از with پس از اتمام کار برنامه با منابع، منابع به صورت خودکار آزاد خواهند شد.

یک استفاده جالب از عبارت with درپایتون، استفاده از آن هنگام پیاده سازی کلاس درونی threading.lock است.

```
some_lock = threading.Lock()

# Harmful:
some_lock.acquire()
try:
    # Do something...
finally:
    some_lock.release()

# Better:
with some_lock:
    # Do something...
```

همانطور که در کد بالا مشاهده می‌کنید، بجای نوشتن try ... finally می‌توان از عبارت with استفاده کرد تا کد بتواند از منابع استفاده کرده و سپس منابع را آزاد کند. استفاده از عبارت with باعث خوانایی راحت کد هنگام استفاده از منابع سیستمی می‌شود. همچنین عبارت with باعث می‌شود تا از به وجود آمدن حفره امنیتی جلوگیری شود. به دلیل اینکه هرزمان که کار برنامه با منابع تمام شد، منابع را به صورت خودکار آزاد می‌کند.

۲-۳-۱. پشتیبانی از عبارت with در اشیا اختصاصی

اکنون متوجه شدید که هیچ چیز جادویی در رابطه با تابع open یا کلاس threading.Lock وجود ندارد و حقیقت این است که این اشیا را از طریق عبارت with در پایتون فراخوانی می‌کنیم. شما می‌توانید همین عملکرد را در کلاس ها و توابع پایتونی خود با استفاده از context manager ها پیاده سازی کنید.

۲-۳-۲. آشنایی با context manager در پایتون

Context manager چیست؟ یک پروتکل یا رابط است که اشیا برنامه باید از آن پیروی کنند تا اشیا برنامه بتوانند از عبارت with پشتیبانی کنند.

اگر بخواهیم عمیق‌تر بنگریم، برای پیاده سازی این قابلیت باید توابع __enter__ و __exit__ پیاده سازی شوند تا عملکردی مشابه با context manager ایجاد کنیم. زبان پایتون دو تابع فوق را هنگام مواجهه با چرخه مدیریت منابع سیستم فراخوانی می‌کند.

بیایید به این موضوع در عمل نگاه کنیم. در ادامه یک پیاده سازی ساده از لایه انتزاعی تابع open پایتون نوشته شده است.

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

    def __enter__(self):
        self.file = open(self.name, 'w')
        return self.file

    def __exit__(self, exc_type, exc_val, exc_tb):
        if self.file:
            self.file.close()
```

کلاس ManagedFile از پروتکل context manager پیروی می‌کند و به همین دلیل از عبارت with پشتیبانی می‌کند. مشابه با همان کاری که در مثال open انجام دادیم را می‌توانیم با کلاسی که به تازگی توسعه داده ایم، انجام دهیم.

```
>>> with ManagedFile('hello.txt') as f:
        f.write('hello, world!')
        f.write('bye now')
```

زبان برنامه نویسی پایتون زمانی تابع __enter__ را فراخوانی می‌کند که برنامه شروع به اجرای کدهای درون قسمت with می‌کند و برنامه نیاز به در اختیار گرفتن منابع دارد. هنگامی که اجرای برنامه در قسمت with به پایان می‌رسد، پایتون تابع __exit__ را فراخوانی می‌کند تا منابعی که در اختیار گرفته شده بودند، آزاد شوند.

نوشتن یک کلاس از جنس context manager تنها راه برای پشتیبانی از عبارت with در پایتون نیست. همانطور که مشاهده کردید کلاس ManagedFile تعداد خط کدهای زیادی داشت و در زبان برنامه نویسی پایتون همیشه به دنبال ساده ترین و بهترین راه حل هستیم.

با استفاده از کتابخانه contextlib در هسته زبان برنامه نویسی پایتون، می‌توان یک لایه انتزاعی بر روی پروتکل context manager ایجاد کرد. این کار ممکن است زندگی را برای شما راحت تر کند!

برای مثال، در قطعه کد زیر با استفاده از یک decorator پایتون به اسم contextmanager می‌توان امکان پشتیبانی از عبارت with را به وجود آورد. در اینجا مثال قبلی کلاس ManagedFile را به صورت تابع بازنویسی می‌کنیم تا ببینیم این تکنیک چگونه کار می‌کند.

```
from contextlib import contextmanager

@contextmanager
def managed_file(name):
    try:
        f = open(name, 'w')
        yield f
    finally:
        f.close()

>>> with managed_file('hello.txt') as f:
        f.write('hello, world!')
        f.write('bye now')
```

در این مثال، تابع managed_file یک generator است که ابتدا منابع را در اختیار می‌گیرد و پس از آن با استفاده از yield امکان استفاده برای منابع را برای درخواست کننده فعال می‌کند. زمانی که اجرای عبارت with به اتمام برسد، generator شروع به تمیزکاری و آزاد کردن منابع می‌کند تا اختیار استفاده از منابع به سیستم برگردد.

پیاده سازی context manager به دو صورت class-based و generator-based در عمل کاملاً مشابه هستند. بهتر است شما با هر روشی که راحت هستید، از یکی از این دو روش استفاده کنید.

برای درک بهتر پیاده سازی context manager در پایتون بهتر است ابتدا درک عمیقی از decorator ها و generator ها در پایتون به دست آورید. اگر می‌خواهید همین الان این درک عمیق را بدست آورید می‌توانید به فصل های مربوط به این مباحث مراجعه کنید. همانطور که در ابتدا گفتیم این کتاب همانند یک بوفه پر از

ویژگی ها و ترفندهای پایتون است و می‌توانید خیلی سریع به مباحث مختلف گذر کنید.

یک بار دیگر برای تاکید می‌گویم، این که از کدام روش پیاده سازی استفاده کنید کاملاً به اینکه شما و تیمتان با کدام روش راحت تر است، بستگی دارد. همیشه راهی را انتخاب کنید که به خوانایی کدهایتان افزوده شود.

۲-۳-۳. نوشتن API های زیبا توسط context manager در پایتون

Context manager ها در پایتون کاملاً انعطاف پذیر هستند. اگر از عبارت with به صورت خلاقانه‌ای استفاده کنید، می‌توانید API های زیبایی برای ماژول‌ها و کلاس‌های خود طراحی کنید.

برای مثال، اگر منابعی که می‌خواهیم استفاده کنیم مجموعه‌ای از نوشته ها باشند که توسط یک برنامه گزارش گیری تولید شده اند و مجموعه ای از تو رفتگی ها (indent) در آن وجود دارد، چطور این کار را انجام می‌دهیم؟

بیشتر توضیح می‌دهم، اگر بخواهید برنامه ای بنویسید که گزارشی به صورت متن تولید می‌کند و این گزارش در هر سطح شامل یک سری تو رفتگی‌ها است، چطور این کار را انجام می‌دهید؟ چطور می‌توانستیم یک لایه انتزاعی پیاده سازی کنیم تا این کار را به راحتی انجام دهیم؟

```
with Indenter() as indent:
    indent.print('hi!')
    with indent:
        indent.print('hello')
        with indent:
            indent.print('bonjour')
    indent.print('hey')
```

کد بالا شبیه زبان های domain-specific برای ایجاد تورفتگی در متن‌ها است. به کد بالا دقت کنید، متوجه می‌شوید که این کد چندین بار وارد context manager های

یکسان شده و از آن خارج شده است تا تورفتگی ها را اعمال کند. اجرا کد بالا باید نتیجه‌ای مشابه زیر را در کنسول برای شما چاپ کند.

```
hi!
    hello
        bonjour
hey
```

بنابراین، برگردیم به اصل موضوع، چطور یک context manager می‌سازید تا عملکرد بالا را بتوانید ایجاد کنید؟

به هرحال، این می‌تواند تمرین خوبی باشد تا به درک عمیقی از context manager ها در پایتون برسید. قبل از اینکه پیاده سازی من را چک کنید، مقداری زمان بگذارید و سعی کنید یک پیاده سازی انتزاعی به عنوان تمرین برای کد بالا انجام دهید.

اگر آماده هستید که پیاده سازی من را ببینید، این روشی است که من برای پیاده سازی یک context manager به صورت class-based انجام می‌دهم تا بتوانم قابلیت ایجاد تورفتگی را در کدهایم را به صورت یک API ایجاد کنم.

```
class Indenter:
    def __init__(self):
        self.level = 0

    def __enter__(self):
        self.level += 1
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        self.level -= 1

    def print(self, text):
        print('    ' * self.level + text)
```

زیاد بد نیست، درست است؟ امیدوارم با خواندن این قسمت، درک بهتری از context manager ها و عبارت with در برنامه های پایتونی بدست آورید. این یک

ویژگی جذاب در پایتون است که با آن می‌توانید منابع سیستم را پایتونیک تر مدیریت کنید.

۲-۳-٤. نکات کلیدی عبارت with در پایتون

- استفاده از عبارت with باعث می‌شود تا بتوانید راحت تر خطاهای برنامه را مدیریت کنید زیرا با این کار عملیات encapsulation را بر روی عبارت استاندارد try...finally انجام می‌دهید.
- بیشترین استفاده از عبارت with زمانی است که می‌خواهید منابعی را توسط برنامه پایتونی در اختیار بگیرید و پس از انجام کار، منابع را به صورت خودکار آزاد کنید.
- استفاده از عبارت with باعث جلوگیری از نشت منابع خواهد شد و همچنین خوانایی بهتری به کدهای شما اضافه می‌کند.

۲-٤. آندرلاین و داندرلاین در پایتون

استفاده از علامت _[1] و علامت __[2] در نام متغیرها و توابع پایتونی معانی متفاوتی دارد. گاهی اوقات برای راهنمایی به توسعه دهندگان دیگری که با کد سروکار دارند است و گاهی به دلیل اجبار مفسر پایتون این کار را انجام می‌دهیم.

اگر تعجب کرده اید که استفاده از علامت‌های _ و __ در پایتون معانی مختلفی دارند، من در اینجا سعی کرده‌ام به بهترین شکل به این ابهام پاسخ دهم. در این قسمت، در رابطه با پنج حالت مختلف زیر توضیح خواهیم داد و اینکه هرکدام از این موارد در زبان برنامه نویسی پایتون چه مفهومی دارند.

- Single Underscore در ابتدا: _var
- Single Underscore در انتها: var_

1 Single Underscore
2 Double Underscore

- Double Underscore در ابتدا: __var
- Double Underscore در ابتدا و انتها: __var__
- Single Undercorse به تنهایی: _

۲-٤-۱. Single Underscore در ابتدا : _var

زمانی که علامت _ به صورت پیشوند در ابتدای نام متغیرها و توابع پایتون قرار می‌گیرد، فقط به دلیل یک قرارداد بین برنامه نویسان پایتون است. درواقع این یک قرارداد یا رسم، بین برنامه نویسان جامعه پایتون است و تاثیری در نحوه اجرای برنامه شما ندارد.

استفاده از پیشوند _ در ابتدای نام متغیرها و توابع پایتون، تنها برای راهنمایی توسعه دهندگان دیگر است و به این مفهوم است که متغیر یا تابع مورد نظر برای استفاده درونی در کدها در نظر گرفته شده است. این یک قرارداد است که در PEP 8 پایتون تعریف شده است و در استایل کدنویسی پایتون رعایت می‌شود.

هرچند، این قرارداد توسط مفسر پایتون به برنامه نویسان اجبار نمی‌شود. پایتون برای متغیرهای private و public همانند زبان برنامه نویسی جاوا تفاوت قائل نمی‌شود.

با قرار دادن _ در ابتدای نام، دارید به توسعه دهنده دیگری که در حال خواندن کدهای شماست می‌گویید: هِی، من این متغیر را فقط برای استفاده درونی درنظر گرفته ام و بخشی از لایه عمومی کد نیست، بهتره باهاش کاری نداشته باشی.

به مثال زیر نگاه کنید.

```
class Test:
    def __init__(self):
        self.foo = 11
        self._bar = 23
```

اگر از این کلاس یک نمونه جدید بسازید آیا می‌توانید به مقادیر foo و _bar دسترسی پیدا کنید؟ بیایید امتحان کنیم.

```
>>> t = Test()
>>> t.foo
11
>>> t._bar
23
```

همانطور که مشاهده می‌کنید، استفاده از علامت _ در متغیر _bar باعث عدم دسترسی به متغیر مورد نظر نشد و توانستیم به آن دسترسی پیدا کنیم و مقدارش را نمایش دهیم.

این موضوع به این دلیل است که استفاده از علامت _ در ابتدای نام متغیرها و توابع تنها یک توافق در جامعه پایتونی است و یک اجبار نیست.

با این حال، اگر از علامت _ در ابتدای نام توابع استفاده می‌کنید، هنگامی که می‌خواهید یک ماژول را فراخوانی کنید، توابعی که در ابتدای نام آنها علامت _ است فراخوانی نمی‌شوند. همانند مثال زیر که یک ماژول به اسم my_module نوشته‌ایم.

```
# my_module.py:
def external_func():
    return 23
def _internal_func():
    return 42
```

اکنون، به دلیل اینکه در ابتدای نام تابع علامت _ قرار داده‌ایم، اگر تمام توابع را در اسکریپت دیگری import کنیم، پایتون توابعی که در ابتدای نام آن‌ها علامت _ است را در اسکریپت import نمی‌کند.

```
>>> from my_module import *
>>> external_func()
23
>>> _internal_func()
NameError: "name '_internal_func' is not defined"
```

به این طریقه‌ی import، اصلاحا wildcard import گفته می‌شود و باید تا حد امکان از آن اجتناب کرد. زیرا با این کار مشخص نیست چه namespace هایی درون

برنامه پایتون وارد می‌شوند. بهتر است برای خوانایی بیشتر کد، به صورت دقیق موارد import ها مشخص شوند. هنگامی که به صورت عادی یک ماژول را به تنهایی import می‌کنید، قاعده فوق در توابعی که در ابتدای نام آن‌ها علامت _ قرار دارد رعایت نمی‌شود و می‌توانید توابع را فراخوانی کنید. به مثال زیر توجه کنید.

```
>>> import my_module
>>> my_module.external_func()
23
>>> my_module._internal_func()
42
```

می‌دانم که ممکن است در این جا کمی گیج شده باشید. اگر استاندارد توصیه شده استایل کدنویسی PEP 8 را رعایت کنید و از wildcard import ها اجتناب کنید، تنها کافی است یک چیز را به یاد داشته باشید:

استفاده از single underscore در ابتدای نام متغیرها و توابع پایتونی فقط یک قرارداد بین توسعه دهندگان به معنی استفاده داخلی متغیر یا تابع است. عدم فراخوانی این متغیرها و توابع توسط مفسر پایتون اجبار نمی‌شود و تنها برای راهنمایی توسعه دهندگان دیگر به منظور استفاده داخلی متغیر یا تابع است.

۲-٤-۲. Single Underscore در انتها: var_

گاهی اوقات بهترین اسمی که برای یک متغیر می‌توانیم انتخاب کنیم توسط هسته زبان برنامه نویسی پایتون انتخاب شده است و نمی‌توانیم از آن اسم استفاده کنیم. برای مثال اسم هایی نظیر class یا def را نمی‌توانیم برای یک متغیر پایتونی انتخاب کنیم. در این مواقع می‌توانید از یک علامت _ در انتهای نام متغیر استفاده کنید تا جلوی تداخل نام‌ها را بگیرید.

```
>>> def make_object(name, class):
SyntaxError: "invalid syntax"

>>> def make_object(name, class_):
        pass
```

به صورت خلاصه، استفاده از علامت _ در انتهای نام متغیرهای پایتونی، به معنای جلوگیری از تداخل نام متغیر با نام های رزرو شده در زبان پایتون است و این موضوع در راهنمای استایل کدنویسی PEP 8 نوشته شده است.

۲-٤-۳. Double Underscore در ابتدا: __var

الگوهایی از نام گذاری که تاکنون بررسی کردیم، اغلب قراردادی بین توسعه دهندگان بودند. در رابطه با کلاس های پایتونی که ویژگی‌ها و رفتارهای کلاس با علامت __ شروع می‌شود، قضیه کمی متفاوت است.

زمانی که از علامت __ در ابتدای نام متغیرها و توابع یک کلاس استفاده می‌کنید، مفسر پایتون نام این صفات را تغییر می‌دهد تا از تداخل نام‌ها جلوگیری کند. به این کار در پایتون name mangling می‌گویند. مفسر پایتون نام صفت را تغییر می‌دهد تا از ایجاد تداخل نام‌ها در آینده جلوگیری کند. درواقع با این کار هنگام گسترش کدها، احتمال ایجاد تداخل نام‌های یک کلاس کمتر می‌شود.

من می‌دانم که این یک موضوع انتزاعی در پایتون است و به این دلیل قطعه کد زیر را نوشتم تا بهتر بتوانیم راجع به طرز کار مفسر پایتون صحبت کنیم.

```
class Test:
    def __init__(self):
        self.foo = 11
        self._bar = 23
        self.__baz = 42
```

در قطعه کد بالا یک کلاس ساده ی پایتونی نوشتیم. اکنون بیایید با استفاده از تابع dir که از توابع درونی زبان برنامه نویسی پایتون است به صفت های این شی نگاه کنیم.

```
>>> t = Test()
>>> dir(t)
['_Test__baz', '__class__', '__delattr__','__dict__',
 '__dir__', '__doc__', '__eq__', '__format__',
'__ge__',
 '__getattribute__', '__gt__', '__hash__','__init__',
 '__le__', '__lt__', '__module__', '__ne__',
'__new__',
 '__reduce__', '__reduce_ex__', '__repr__',
 '__setattr__', '__sizeof__', '__str__',
 '__subclasshook__', '__weakref__', '_bar', 'foo']
```

با استفاده از این تابع می‌توانیم لیستی از نام ویژگی‌های یک شی را مشاهده کنیم. بیایید به دنبال نام های foo، bar_ و baz__ که به تازگی در شی (یا کلاس) Test ساختیم بگردیم. آیا می‌توانید هر سه صفت را در لیست صفت های شی t پیدا کنید؟ مطمئنم هستم تغییرات عجیبی مشاهده می‌کنید!

اول از همه، صفت self.foo بدون تغییر و با نام foo که برای آن مشخص کردیم در لیست خروجی ظاهر شده است.

بعد از foo، به دنبال صفت self._bar می‌گردیم و مشاهده می‌کنیم که همانند صفت قبلی رفتار کرده است و به صورت نام _bar قابل مشاهده است. همانطور که قبلا گفتیم علامت _ تنها یک راهنما برای برنامه نویسان دیگر است.

به هر حال، به سراغ self.__baz می‌رویم اما ظاهرا نتیجه کمی متفاوت به نظر می‌رسد. زمانی که کلمه __baz را در لیست جست و جو می‌کنیم، مشاهده می‌شود هیچ کلمه ای برابر با این نام وجود ندارد.

پس چه اتفاقی برای __baz افتاد؟

اگر کمی دقیق‌تر نگاه کنید، نام _Test__baz را در خروجی مشاهده می‌کنید. این همان name mangling است که پیش‌تر گفتیم. مفسر پایتون به صورت خودکار نام

کلاس را در ابتدای نام صفت قرار می‌دهد تا از نام این صفت در آینده هنگام گسترش کدها محافظت کند.

اکنون شاید از خود بپرسید مفسر پایتون چرا باید از نام این کلاس در آینده محافظت کند؟ این کار برای محافظت از نام ویژگی‌های یک شی در مقابل override شدن توسط کلاس های فرزندی است که در آینده قرار است ساخته شوند.

برای روشن تر شدن موضوع بیایید یک کلاس دیگر بسازیم و آن را از کلاس Test ارث بری کنیم و سعی کنیم ویژگی‌های کلاس Test را override کنیم. بنابراین کلاس جدید را به شکل زیر می‌نویسم.

```
class ExtendedTest(Test):
    def __init__(self):
        super().__init__()
        self.foo = 'overridden'
        self._bar = 'overridden'
        self.__baz = 'overridden'
```

اکنون فکر می کنید مقادیر صفت‌های foo، _bar و __baz در یک شی از جنس کلاس ExtendedTest چه مقادیری خواهد بود؟

```
>>> t2 = ExtendedTest()
>>> t2.foo
'overridden'
>>> t2._bar 'overridden'
>>> t2.__baz
AttributeError: \
"'ExtendedTest' object has no attribute '__baz'"
```

صبر کنید! چه اتفاقی افتاد؟ چرا هنگامی که خواستیم مقدار صفت t2.__baz را بدست آوریم یک خطای AttributeError گرفتیم؟ ظاهرا name mangling مثل قبل دست به کار شده است! همانطور که مشاهده می کنید همانند قبل خروجی صفحه‌ی بعد شامل نام صفت __baz نیست.

```
>>> dir(t2)
['_ExtendedTest__baz', '_Test__baz', '__class__',
 '__delattr__', '__dict__', '__dir__', '__doc__',
 '__eq__', '__format__', '__ge__',
'__getattribute__',
 '__gt__', '__hash__', '__init__', '__le__',
'__lt__',
'__module__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__',
'__weakref__', '_bar', 'foo', 'get_vars']
```

همانطور که واضح است، نام __baz___ برای جلوگیری از تداخل تصادفی تبدیل به _ExtendedTest__baz شده است اما نام _Test__baz هنوز در لیست فوق وجود دارد و هرکدام دارای مقادیر مختلفی هستند.

```
>>> t2._ExtendedTest__baz
'overridden'
>>> t2._Test__baz
42
```

بسیاری از برنامه نویسان پایتون از نحوه عملکرد name mangling در هنگام استفاده از علامت __ در نام ویژگی‌های یک کلاس اطلاعی ندارند. مثال زیر این موضوع را تایید می‌کند.

```
class ManglingTest:
    def __init__(self):
        self.__mangled = 'hello'

    def get_mangled(self):
        return self.__mangled

>>> ManglingTest().get_mangled()
'hello'
>>> ManglingTest().__mangled
AttributeError: \
"'ManglingTest' object has no attribute '__mangled'"
```

آیا name mangling روی نام توابع هم اعمال می‌شود؟ بله. name mangling درتمام نام های توابع یک کلاس که با علامت __ شروع می‌شوند، اعمال می‌شود.

```
class MangledMethod:
    def __method(self):
        return 42

    def call_it(self):
        return self.__method()

>>> MangledMethod().__method()
AttributeError: \
"'MangledMethod' object has no attribute '__method'"
>>> MangledMethod().call_it()
42
```

یک مثال دیگر را می‌بینیم که احتمالا شما را شگفت زده می کند.

```
_MangledGlobal__mangled = 23

class MangledGlobal:
    def test(self):
        return __mangled
>>> MangledGlobal().test()
23
```

در این مثال، ابتدا یک متغیر سراسری _MangledGlobal__mangled تعریف کرده ام. سپس به این متغیر از طریق کلاس MangledGlobal دسترسی پیدا کرده ام. به دلیل استفاده از قابلیت name mangling در پایتون، توانستم متغیر _MangledGlobal__mangled را از طریق نام __mangled در تابع test فراخوانی کنم.

مفسر پایتون به صورت خودکار نام __mangled را به نام _MangledGlobal__mangled تغییر داد. به این دلیل که این صفت با علامت __ شروع شده است. این موضوع مشخص می کند که name mangling نه تنها در نام

متغیرها و توابع، بلکه در هر نامی که با علامت __ در فضای یک کلاس شروع شود توسط مفسر پایتون اعمال می‌شود.

ظاهرا خیلی عمیق شدیم. جالب است نه؟ حقیقت این است که من این مثال ها و توضیحات را خیلی راحت از ذهن خود در نیاورده ام. زمان و انرژی زیادی برای تحقیق و درک درستی از این ویژگی از دست داده ام. من چندین سال است که از زبان پایتون برای برنامه‌نویسی استفاده می‌کنم. اما درک نحوه عملکرد مفسر پایتون در حالت های خاص نیاز به تحقیقات فراوانی دارد.

گاهی اوقات مهم ترین مهارت یک برنامه نویس، تشخیص الگو و پیدا کردن جایی است که باید مشکل را در آنجا پیدا کند. اگر احساس می‌کنید کمی در درک این مثال ها گیج شده اید، نگران هیچ چیز نباشید. کمی به خود زمان بدهید و با مثال های این قسمت تمرین کنید تا به خوبی متوجه شوید.

۲-٤-٤. نکته جانبی: داندر[1] در پایتون چیست؟

اگر گاهی صحبت پایتونیست های مختلف راجع به زبان پایتون را گوش می‌دهید یا کنفرانس های پایکان را مشاهده می‌کنید ممکن است کلمه ی داندر به گوشتان خورده باشد و تعجب کرده باشید داندر به چه معنایی است.

Double underscore ها (علامت __) گاهی اوقات به دلیل راحتی در صحبت کردن، توسط جامعه پایتون dunder تلفظ می‌شوند. به این دلیل که استفاده از double underscore ها در کدهای پایتونی زیاد اتفاق می‌افتد و برای اینکه توسعه دهندگان پایتون از خسته شدن ماهیچه های فک خود جلوگیری کنند، اغلب double underscore را به حالت کوتاه شده dunder تلفظ می‌کنند.

1 Dunder

برای مثال من نام baz__ را dunder baz می‌خوانم. همچنین نام __init__ را نیز dunder init می‌خوانم. اگرچه ممکن است بعضی افراد فکر کنند باید آن را به صورت dunder init dunder بخوانند.

خب برویم سراغ نحوه نام گذاری بعدی، یک قاعده نام گذاری دیگری وجود دارد که برای پایتونیست ها شبیه به اسم رمز می‌ماند.

۲-٤-۵. double underscore در ابتدا و انتها: __var__

ممکن است باعث شگفت زدگی شما شود اما name mangling در زمانی که یک صفت پایتونی با dunder شروع می‌شود و پایان می‌یابد، اعمال نمی‌شود. متغیرهایی که در ابتدا و انتهای آن‌ها dunder قرار دارد توسط مفسر پایتون دست کاری نمی‌شوند.

```
class PrefixPostfixTest:
    def __init__(self):
        self.__bam__ = 42
>>> PrefixPostfixTest().__bam__
42
```

نام هایی که در پایتون علامت __ در ابتدا و انتهای آن‌ها وجود دارد، برای استفاده خاص در زبان برنامه نویسی پایتون طراحی شده اند. این قانون در توابعی مثل __init__ که تابع سازنده یک کلاس است یا تابع __call__ که برای فراخوانی شی استفاده می‌شود یا توابع دیگر بکار گرفته شده است.

این توابع به عنوان magic method یا متدهای جادویی توسط پایتون معرفی شده اند، اما بسیاری از افراد جامعه پایتون از جمله خودم نام متدهای جادویی را دوست نداریم. این نام باعث می‌شود بسیاری از برنامه نویسان از استفاده از این قابلیت دلسرد شوند. در صورتی که هیچ چیز جادویی در این رابطه وجود ندارد. این یکی از ویژگی های هسته زبان برنامه نویسی پایتون است که هر زمان نیاز باشد، باید بتوانید از آن‌ها استفاده کنید.

به هر حال، بهتر است متغیرهایی که با علامت __ شروع و پایان می‌یابند نسازید تا از تداخل کدهایتان در تغییرات آینده هسته زبان برنامه نویسی پایتون جلوگیری کنید.

۲-٤-٦. Single Undercorse به تنهایی: _

بر اساس قراردادی، یک علامت _ در زبان پایتون هنگامی استفاده می‌شود که می‌خواهیم برای یک متغیر یک نام موقتی در نظر بگیریم.

برای مثال، در حلقه زیر نیازی به شمارنده حلقه نداریم و می‌توانیم از یک علامت "_" برای نام شمارنده حلقه استفاده کنیم.

```
>>> for _ in range(32):
        print('Hello, World.')
```

همچنین شما می‌توانید از علامت _ برای نام گذاری متغیرهایی که برایتان اهمیت ندارند هنگام باز کردن یک کالکشن پایتون استفاده کنید. تاکید می‌کنم، اینکه می‌گویم متغیر اهمیت ندارد تنها یک رسم بین جامعه پایتونی است. مفسر پایتون هنگامی که با "_" روبرو می‌شود هیچ تغییری در نحوه ی اجرای برنامه انجام نمی‌دهد. می‌توانید اینطور در نظر بگیرید که "_" تنها یک متغیر معتبر است که برای مواقعی که نام متغیر برایمان اهمیت ندارد از این نام استفاده می‌کنیم.

برای مثال در کد زیر، من یک tuple را باز می‌کنم که تنها علاقمند به فیلدهای color و mileage کالکشن tuple هستم و بقیه فیلدها برایم مهم نیستند. به هر حال، به دلیل اینکه ترتیب هنگام باز کردن یک کالکشن مهم است متغیرها را به ترتیب قرار می‌دهم و اینجا جایی است که نام "_" به درد من می‌خورد.

```
>>> car = ('red', 'auto', 12, 3812.4)
>>> color, _, _, mileage = car

>>> color
'red'

>>> mileage
```

```
3812.4

>>> _
12
```

در کنار اینکه "_" برای نام گذاری متغیرهای موقتی استفاده می‌شود، "_" یک متغیر مخصوص در Python REPL است و از طریق آن می‌توان به نتیجه محاسبات قبلی در interpreter پایتون دسترسی داشت.

برای مثال هنگامی که با سشن مفسر پایتون در حال کار هستید و نیاز به دسترسی به نتیجه محاسبه قبلی را دارید می‌توانید از "_" استفاده کنید.

```
>>> 20 + 3
23
>>> _
23
>>> print(_)
23
```

همچنین اگر می‌خواهید یک شی موقتی بسازید (اصطلاحا objects on the fly) می‌توانید از "_" استفاده کنید تا نامی برای شی خود انتخاب نکنید.

```
>>> list()
[]
>>> _.append(1)
>>> _.append(2)
>>> _.append(3)
>>> _
[1, 2, 3]
```

۲-٤-۷. نکات کلیدی آندرها و داندرها در پایتون

- Single underscore در ابتدا: یک رسم نام گذاری برای اطلاع برنامه نویسان دیگر از درونی یا محلی بودن متغیر است. مفسر هیچ اجباری برای محلی بودن متغیر انجام نمی‌دهد.

- Single underscore در انتها: یک رسم نام گذاری برای جلوگیری از تداخل نام با کلمات کلیدی زبان برنامه نویسی پایتون است و توسط مفسر پایتون کاری انجام نمی‌شود.
- Double underscore در ابتدا: باعث اعمال name mangling در سطح یک کلاس می‌شود و توسط مفسر پایتون تغییراتی در نام گذاری انجام می‌شود.
- Double underscore در ابتدا و انتها: برای استفاده از توابع مشخص و تعریف شده در پایتون استفاده می‌شود. از این حالت برای نام گذاری صفت های یک کلاس استفاده نکنید تا از تداخل پیشگیری شود.
- Single underscore: گاهی اوقات برای استفاده از نام های موقت استفاده می‌شود یا متغیرهایی که نامشان برایمان مهم نیستند. همچنین در سشن Python REPL برای مشاهده نتیجه محاسبه قبلی استفاده می‌شود.

۲-۵. حقیقتی ناخوشایند در مورد رشته‌های پایتون

ذن پایتون[۱] و اینکه چگونه باید "یک روش واضح برای انجام کار باشد" را به یاد آورید. هنگامی که بفهمید چهار روش برای انجام قالب‌بندی متن در پایتون وجود دارد، ممکن است از تعجب سر خود را بخارانید.

در این فصل، نحوه کار این چهار رویکرد قالب‌بندی متن و نقاط قوت و ضعف مربوط به آن‌ها را نشان خواهیم داد. همچنین "قاعده سرانگشتی ساده" برای نحوه انتخاب بهترین رویکرد قالب‌بندی متن را ارائه خواهیم کرد.

بیایید یک راست برویم سر اصل مطلب، زیرا باید موضوعات زیادی را بیان کنیم. برای اینکه مثال کوچک و ساده‌ای برای آزمایش داشته باشیم، فرض می‌کنیم که متغیرهای صفحه‌ی بعد (یا در حقیقت، ثابت‌ها) را برای بررسی قالب بندی رشته‌های پایتون در نظر گرفته‌ایم.

---

۱ مجموعه ای از ۲۰ اصل نرم‌افزاری تأثیرگذار بر طراحی زبان برنامه‌نویسی پایتون

```
>>> errno = 50159747054
>>> name = 'Bob'
```

و بر پایه این متغیرها، مایلیم رشته خروجی را با پیام خطای زیر ایجاد کنیم.

```
'Hey Bob, there is a 0xbadc0ffee error!'
```

بروز این خطا می‌تواند واقعا صبح شنبه یک توسعه دهنده را خراب کند! اما امروز در اینجا هستیم تا درباره قالب‌بندی متن در پایتون صحبت کنیم. پس بیایید به کار خود برسیم.

۲-۵-۱. قالب بندی متن به سبک قدیمی

رشته‌ها در پایتون دارای عملیات داخلی منحصر بفرد هستند که می‌توان با ٪-اپراتور به آن دست یافت. این میانبری است که به شما امکان قالب‌بندی راحت متن را می‌دهد. اگر تا حالا با تابع printf در زبان برنامه‌نویسی C کار کرده‌اید، فورا نحوه کار با آن ها را متوجه خواهید شد. در زیر مثال ساده‌ای وجود دارد.

```
>>> 'Hello, %s' % name
'Hello, Bob'
```

در اینجا از تعیین‌کننده قالب %s استفاده می‌کنیم تا به پایتون بگوید که مقدار متغیر name را بصورت رشته جایگزین کند. به این قالب‌بندی متن به "سبک قدیمی" گفته می‌شود.

در قالب‌بندی متن به سبک قدیمی، تعیین‌کننده‌های دیگری از قالب نیز وجود دارد که به شما اجازه کنترل رشته خروجی را می‌دهد. برای مثال، تبدیل اعداد به هگزادسیمال یا افزودن فضای خالی در متن برای ایجاد جداول و گزارش‌های قالب‌بندی شده زیبا، امکان‌پذیر است.

در قطعه کد صفحه‌ی بعد، از تعیین‌کننده قالب %x برای تبدیل مقدار از عدد صحیح به رشته و نشان دادن آن بصورت هگزادسیمال استفاده می‌کنیم.

```
>>> '%x' % errno
'badc0ffee'
```

اگر بخواهید جایگزین‌های چندگانه را در رشته واحد ایجاد کنید، سینتکس قالب بندی متن به سبک قدیمی کمی تغییر خواهد کرد. چون ٪-اپراتور فقط یک آرگومان را می‌گیرد. شما باید طرف راست را در یک tuple، مانند دستور زیر قرار دهید.

```
>>> 'Hey %s, there is a 0x%x error!' % (name, errno)
'Hey Bob, there is a 0xbadc0ffee error!'
```

همچنین می‌توانید قطعه کد بالا را به صورت زیر بنویسید تا عملیات جایگزینی انجام شود.

```
>>> 'Hey %(name)s, there is a 0x%(errno)x error!' % {
    "name": name, "errno": errno }
'Hey Bob, there is a 0xbadc0ffee error!'
```

این کار باعث تغییر و اصلاح ساده‌تر رشته‌ها در آینده می‌شود و نیازی نیست نگران ترتیب جایگزینی صحیح ٪-اپراتور ها باشید. البته نقطه ضعف این روش این است که به کمی تایپ بیشتری نیاز دارد.

مطمئن هستم تعجب کرده‌اید که چرا این قالب‌بندی به سبک printf قالب‌بندی متن به "سبک قدیمی" نامیده می‌شود. خب، اجازه دهید به شما بگویم. سبک قدیمی از لحاظ فنی پس از مدتی، به طور مجدد جایگزین قالب‌بندی به "سبک جدید" شد، که در ادامه می‌خواهیم کمی درباره آن صحبت کنیم. با اینکه قالب‌بندی به "سبک قدیمی" به طور کامل ترجیح داده نشده است، اما کاملاً هم کنار گذاشته نشده و هنوز در آخرین نسخه‌های پایتون پشتیبانی می‌شود.

۲-۵-۲. قالب‌بندی متن به سبک جدید

پایتون ۳ روشی جدید برای انجام قالب‌بندی متن ارائه داد که بعدا به پایتون ۲.۷ - انتقال داده شد. این قالب‌بندی متن به "سبک جدید" از سینتکس ویژه ٪-اپراتور خلاص می‌شود و باعث منظم‌ترشدن سینتکس برای قالب‌بندی متن می‌شود. امروزه قالب‌بندی با فراخوانی تابع format() در شیء رشته، مدیریت می‌شود.

می‌توانید از تابع format() برای انجام قالب‌بندی موقعیتی ساده استفاده کنید، دقیقا مانند آنچه که می‌توانستید با قالب‌بندی به "سبک قدیمی" انجام دهید.

```
>>> 'Hello, {}'.format(name)
'Hello, Bob'
```

یا می‌توانید طبق نام به جایگزین‌های متغیرها رجوع کرده و از آنها به هر ترتیب که بخواهید استفاده کنید. این ویژگیِ کاملاً قدرتمندی است، چون اجازه تنظیم مجدد ترتیب نمایش بدون تغییر آرگومان‌های ورودی به تابع format را می‌دهد.

```
>>> 'Hey {name}, there is a 0x{errno:x}
error!'.format(
    name=name, errno=errno)
'Hey Bob, there is a 0xbadc0ffee error!'
```

این روش همچنین نشان می‌دهد که سینتکس برای قالب‌بندی متغیر به صورت رشته هگزادسیمال در برنامه تغییر کرده است. باید با افزودن پسوند ":x" پس از نام متغیر، مشخصات قالب را منتقل کنیم.

در حالت کلی، سینتکس جدید قالب بندی رشته ها، ساده‌تر و قدرتمندتر شده است.

در پایتون ۳، این قالب‌بندی متن به "سبک جدید" بر قالب‌بندی به سبک ٪-اپراتور ترجیح داده می‌شود. با این حال، با انتشار پایتون ۳,۶ حتی روشی بهتر برای قالب‌بندی متن‌های شما وجود دارد. در بخش بعدی، در مورد آن همه چیز را خواهم گفت.

۲-۵-۳. درج یا الحاق رشته های قالب‌بندی شده (پایتون+۳,۶)

پایتون ۳,۶ روشی دیگر برای قالب‌بندی رشته‌ها، تحت عنوان الحاق رشته‌های قالب بندی شده[1] اضافه می‌کند. این روش جدید برای قالب‌بندی رشته‌ها به شما امکان استفاده از شی‌های مقداردهی شده پایتون در داخل رشته را فراهم می‌آورد. در اینجا مثالی ساده برای درک این ویژگی وجود دارد.

```
>>> f'Hello, {name}!' \
'Hello, Bob!'
```

این سینتکس جدید قالب‌بندی، قدرتمند است. چون می‌توانید هر نوع عبارات دلخواه پایتون را در رشته‌ی خود جاسازی کنید، حتی می‌توانید محاسبات درون‌خطی را با آن انجام دهید، مانند مثال زیر.

```
>>> a = 5
>>> b = 10
>>> f'Five plus ten is {a + b} and not {2 * (a +
b)}.'
'Five plus ten is 15 and not 30.'
```

در پشت صحنه، الحاق رشته‌های قالب‌بندی شده، یک ویژگی تجزیه‌گر پایتون است که f-strings را به یک سری از عبارات رشته تبدیل می‌کند. سپس این عبارات برای ایجاد رشته نهایی به هم متصل می‌شوند.

تصور کنید تابع greet زیر که شامل f-strings است را داریم.

```
>>> def greet(name, question):
        return f"Hello, {name}! How's it {question}?"

>>> greet('Bob', 'going')
"Hello, Bob! How's it going?"
```

1 Formatted String Literals

هنگامی که تابع را تجزیه می‌کنیم و آنچه را که در پشت صحنه اتفاق می‌افتد را بررسی می‌کنیم، مشاهده می‌کنیم که f-strings در این تابع به چیزی شبیه به کد زیر تبدیل می‌شود.

```
>>> def greet(name, question):
        return ("Hello, " + name + "! How's it "
+question + "?")
```

پیاده‌سازی واقعی کمی سریعتر از آن است چون از آپکد BUILD_STRING[1] برای بهینه‌سازی استفاده می‌کند. اما به لحاظ کارکردی یکسان هستند.

```
>>> import dis
>>> dis.dis(greet)
  2     0 LOAD_CONST         1 ('Hello, ')
        2 LOAD_FAST          0 (name)
        4 FORMAT_VALUE       0
        6 LOAD_CONST         2 ("! How's it ")
        8 LOAD_FAST          1 (question)
       10 FORMAT_VALUE       0
       12 LOAD_CONST         3 ('?')
       14 BUILD_STRING       5
       16 RETURN_VALUE
```

همچنین الحاق رشته‌های قالب‌بندی شده از سینتکس موجود در روش str.format() پشتیبانی می‌کنند. این کار برای شما امکان حل مسائل قالب‌بندی رشته‌ها، مشابه دو بخش قبلی را فراهم می‌کند.

```
>>> f"Hey {name}, there's a {errno:#x} error!" \
"Hey Bob, there's a 0xbadc0ffee error!"
```

رشته‌های نوشتاری جدید قالب‌بندی شده در پایتون شبیه به الگوهای نوشتاری جاوا اسکریپت در ES2015 است. فکر می‌کنم ویژگی کاملاً خوبی برای زبان هستند و قبلاً

1 Opcode

از آن‌ها در کارهای روزانه پایتون ۳ استفاده کرده‌ام. در مستندات رسمی پایتون می توانید مطالب بیشتری در مورد الحاق رشته‌های قالب بندی شده یاد بگیرید.

۲-۵-٤. الگوهای رشته‌ها

یک روش دیگر برای قالب‌بندی رشته در پایتون، استفاده از الگوی رشته‌ها (یا قالب رشته‌ها)[1] است. این مکانیزمی ساده‌تر و با صرف هزینه کمتر است، اما در برخی موارد ممکن است دقیقا همان چیزی باشد که به دنبال آن هستید.

اجازه دهید تا یک مثال ساده را بررسی کنیم.

```
>>> from string import Template
>>> t = Template('Hey, $name!')
>>> t.substitute(name=name)
'Hey, Bob!'
```

در اینجا می‌بینید که نیاز به import کردن کلاس Template را از ماژول داخلی رشته پایتون داریم. الگوی رشته‌ها، ویژگی اصلی زبان نیستند بلکه با ماژولی در کتابخانه استاندارد پایتون تعبیه شده‌اند.

تفاوت دیگر این است که الگوی رشته‌ها اجازه مشخص کردن فرمت اشیا را نمی دهند. بنابراین به منظور نمایش نمونه رشته ی خطا، باید عدد خطا از جنس عدد صحیح را به رشته هگزادسیمال تبدیل کنیم.

```
>>> templ_string = 'Hey $name, there is a $error
error!'
>>> Template(templ_string).substitute(
    name=name, error=hex(errno))
'Hey Bob, there is a 0xbadc0ffee error!'
```

نحوه کار این روش عالی است، به عقیده من، بهترین مورد استفاده برای الگوی رشته‌ها زمانی است که رشته‌های ورودی تولید شده توسط کاربران را قالب بندی

1 Template Strings

می‌کنید. زیرا به دلیل کاهش مخاطرات ورودی های ارسالی از طرف کاربران، این روش انتخاب امن تری است.

قالب‌بندی بسیار پیچیده رشته‌ها ممکن است آسیب‌پذیری‌های امنیتی را در برنامه شما ایجاد کند. برای مثال، این امکان برای قالب بندی نادرست رشته‌ها وجود دارد که به متغیرهای دلخواه در برنامه شما دسترسی پیدا کنند.

این بدین معناست که اگر کاربر مخرب بتواند رشته قالب را تهیه کند، پس احتمالاً می‌تواند به کلیدهای مخفی و سایر اطلاعات حساس نیز نفوذ کند! در اینجا اثبات ساده ای درباره نحوه استفاده از این حمله وجود دارد.

```
>>> SECRET = 'this-is-a-secret'
>>> class Error:
        def __init__(self):
            pass

>>> err = Error()
>>> user_input = '{error.__init__.__globals__[SECRET]
}'

# Uh-oh...
>>> user_input.format(error=err)
'this-is-a-secret'
```

ببینید چگونه مهاجم فرضی توانست با دسترسی به دیکشنری __globals__ از قالب رشته، رشته مخفی ما را استخراج کند؟ ترسناک است! الگوی رشته‌ها، این نوع حمله را خنثی می‌کنند و در صورت دستکاری قالب رشته‌های تولید شده از ورودی کاربر، پاسخ ایمن‌تری می‌دهند.

```
>>> user_input =
'${error.__init__.__globals__[SECRET]}'
>>> Template(user_input).substitute(error=err)
ValueError:
"Invalid placeholder in string: line 1, col 1"
```

۲-۵-۵. از کدام روش قالب‌بندی رشته‌ها باید استفاده کنیم؟

کاملاً دریافتم که با داشتن انتخاب بسیار زیاد برای نحوه قالب‌بندی رشته‌ها در پایتون ممکن است احساس سردرگمی کنید. این لحظه خوبی خواهد بود تا فلوچارت و اینفوگرافیک های مختلفی را ترسیم کنیم.

اما نمی‌خواهم این کار را انجام دهم، بلکه سعی خواهم کرد تا آن را با استفاده از قاعده سرانگشتی ساده‌ای که هنگام نوشتن پایتون بکار می‌برم، انجام دهم. می‌توانید بسته به شرایط، از این قاعده سرانگشتی در لحظه مواجه با تصمیم‌گیری دشوار در رابطه با استفاده از کدامین روش قالب‌بندی رشته، استفاده کنید.

۲-۵-۶. قاعده سرانگشتی قالب‌بندی متن در پایتون

اگر رشته‌های ورودی برنامه شما توسط کاربر ارسال شده، از الگوی رشته‌ها برای جلوگیری از مشکلات امنیتی استفاده کنید. در غیراینصورت، اگر از پایتون +۶. ۳ استفاده می‌کنید، از الحاق یا درج رشته قالب‌بندی شده استفاده کنید و اگر از پایتون +۶. ۳ استفاده نمی‌کنید از قالب‌بندی متن به "سبک جدید" استفاده کنید.

۲-۵-۷. نکات کلیدی در هنگام کار با رشته‌ها در پایتون

- شاید تعجب‌آور باشد، در پایتون بیش از یک روش برای مدیریت قالب‌بندی متن وجود دارد.
- هر روش دارای جوانب مثبت و منفی جداگانه‌ای است. مورد استفاده شما بر انتخاب روش، اثر خواهد گذاشت.
- اگر در رابطه با اینکه از کدام روش قالب‌بندی رشته استفاده کنید، مشکل تصمیم گیری دارید، قاعده سرانگشتی قالب‌بندی رشته من را امتحان کنید.

۲-۶. ذن پایتون

می‌دانیم آنچه که به شرح زیر است، تا جایی که پایتون ادامه دارد، یک دید مشترک بین برنامه‌نویسان پایتون است. اما واقعا هیچ روشی در مورد اجرای ذن پایتون وجود ندارد. من از بازبینی ذن پایتون در این سال ها سود برده‌ام، و فکر می‌کنم سخنان تیم پیترز [1] مرا به برنامه‌نویس بهتری تبدیل کرد. امیدوارم ذن پایتون بتواند همین کار را برای شما انجام دهد.

برای مشاهده ذن پایتون لازم است وارد مفسر پایتون شوید و چنین دستوری را اجرا کنید.

```
>>> import this
```

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren’t special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one—and preferably only one—obvious way to do it.
Although that way may not be obvious at first unless you’re Dutch. Now is better than never.
Although never is often better than right now.
If the implementation is hard to explain, it’s a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea—let’s do more of those!

1 Tim Peters

ذن پایتون، نوشته شده توسط تیم پیترز

زیبا بهتر از زشت است.

بیان صریح بهتر از ضمنی است.

ساده بهتر از پیچیده است.

پیچیده بهتر از خیلی پیچیده (افتضاح) است.

مستقیم و صاف بهتر از تو در تو است.

پراکنده بهتر از متراکم است.

خوانایی مهم است.

موارد ویژه به اندازه‌ای ویژه نیستند که به خاطر آنها بتوان قوانین را شکست.

گرچه عملی بودن خلوص را از بین می‌برد.

خطاها هرگز نباید با سکوت رد شوند.

مگر اینکه صراحتا خاموش شوند.

در مواجهه با ابهام، از وسوسه‌ی حدس زدن دوری کنید.

برای انجام یک کار باید یک (ترجیحا یک) روش واضح وجود داشته باشد.

هر چند ممکن است این روش در ابتدا واضح و آشکار نباشد مگر اینکه شما هلندی باشید.

حالا، بهتر از هرگز است.

اگر چه هرگز، اغلب راحت تر از حال حاضر است.

اگر توضیح پیاده‌سازی دشوار باشد، پس ایده بدی خواهد بود.

اگر توضیح پیاده‌سازی آسان باشد، پس ایده خوبی خواهد بود.

فضاهای نام یک ایده عالی است، بیایید بیشتر از این کارها کنیم!

به این جملات ذن ۲۰ گانه پایتون می‌گویند. ذن پایتون مجموعه‌ای از ۲۰ اصل بنیادین هنگام طراحی زبان پایتون است که توسط تیم پیترز در سال ۱۹۹۹ نوشته شده است. نکته‌ی جالب در رابطه با ذن پایتون این است که تیم پیترز تنها ۱۹ اصل از ۲۰ اصل را بیان کرده است و اصل شماره ۲۰ را برای خالق زبان پایتون، Guido Van Rossum کنار گذاشته است. اصل ۲۰ ذن پایتون تا لحظه‌ی نگارش این کتاب توسط خالق زبان پایتون بیان نشده است.

فصل سوم

توابع تأثیرگذار

۳-۱. توابع پایتون شهروندان درجه اول[1] هستند

توابع پایتون، اشیاء درجه اول هستند. می‌توانید آن‌ها را به متغیرها اختصاص دهید، آن‌ها را در ساختار داده‌ها ذخیره کنید، آن‌ها را به عنوان آرگومانهایی به سایر توابع منتقل کنید و حتی آن‌ها را بعنوان مقداری از سایر توابع برگردانید.

درک شهودی این مفاهیم باعث درک بسیار راحت ویژگی های پیشرفته در پایتون مانند لامبداها[2] و دکوراتورها[3] خواهد شد. همچنین شما را به سمت روش‌های برنامه نویسی تابعی[4] سوق می‌دهد.

در چند صفحه بعد، شما را از طریق برخی مثال‌ها راهنمایی خواهیم کرد تا به شما در توسعه این درک شهودی کمک کنیم. این مثال‌ها روی یکدیگر قرار خواهند گرفت، بنابراین ممکن است بخواهید آنها را به ترتیب بخوانید و در صورت تداوم ممکن است حتی برخی از آنها را در مفسر پایتون امتحان کنید.

به مفاهیمی که در اینجا مورد بحث قرار خواهیم داد، کمی فکر کنید. ممکن است کمی بیشتر از آنچه انتظار دارید، طول بکشد. نگران نباشید، این شرایط کاملاً طبیعی است. من در این شرایط بوده‌ام. ممکن است حس کنید که بخواهید سرتان را به دیوار بکوبید و بعد هنگامی که آماده هستید، ناگهان همه چیز "کلیک" خواهد خورد و مفاهیم در ذهنتان در جای درست قرار خواهند گرفت.

در سرتاسر این فصل، از تابع yell برای اهداف آموزشی استفاده خواهیم کرد. این مثال ساده‌ای است که خروجی آن را به راحتی می‌توان تشخیص داد.

---

1 First-class
2 lambdas
3 decorators
4 Functional programming

```
>>> def yell(text):
        return text.upper() + '!'

>>> yell('hello')
'HELLO!'
```

۳-۱-۱. توابع اشیا هستند

تمام داده‌های موجود در برنامه پایتون با اشیاء یا روابط بین اشیاء نشان داده می‌شوند. چیزهایی مانند رشته‌ها، لیست‌ها، ماژول‌ها و توابع همگی اشیاء هستند. هیچ چیز خاصی در مورد توابع پایتون وجود ندارد. آنها نیز فقط اشیاء هستند.

چون تابع yell یک شیء در پایتون است، پس می‌توانید درست مانند هر شیء دیگری آن را به متغیر دیگری اختصاص دهید.

```
>>> bark = yell
```

این خط تابع را فراخوانی نمی‌کند. این خط، شیء تابعی به نام yell را می‌گیرد و آن را در شیء bark که به آن اشاره می‌کند، قرار می‌دهد. حال می‌توانید با فراخوانی bark، همان تابع اصلی yell را نیز اجرا کنید.

```
>>> bark('woof')
'WOOF!'
```

اشیای تابع و نام‌های آن‌ها دو مسأله جداگانه هستند. کمی عمیق‌تر بنگریم، شما می‌توانید نام اصلی تابع (yell) را حذف کنید. چون نام دیگر (bark) هنوز به تابع اصلی اشاره می‌کند، سپس می‌توانید این تابع را از طریق دستور زیر فراخوانی کنید.

```
>>> del yell

>>> yell('hello?')
NameError: "name 'yell' is not defined"

>>> bark('hey')
'HEY!'
```

به هر حال، پایتون با اهداف اشکال‌زدایی و عیب‌یابی، شناسه نام تابع را به تابع در زمان اجرا، متصل می‌کند. با تابع جادویی __name__ می‌توانید به این شناسه‌ی نام داخلی دسترسی پیدا کنید.

```
>>> bark.__name__
'yell'
```

حال، با اینکه مقدار __name__ تابع هنوز "yell" است، اما بر نحوه دسترسی شما به شیء تابع اثر نمی‌گذارد. شناسه نام تابع صرفاً برای کمک به اشکال‌زدایی است. بنابراین در این قسمت متوجه شدیم متغیری که به نام تابع اشاره می‌کند و خود تابع، دو نگرانی متفاوت هستند.

۳-۱-۲. توابع می‌توانند در ساختارهای داده ذخیره شوند

چون توابع شهروندان درجه اول هستند، می‌توانید آن‌ها را در ساختارهای داده ذخیره کنید، درست مانند کارهایی که می‌توانید با سایر اشیاء انجام دهید. برای مثال، می‌توانید توابع را به لیست اضافه کنید.

```
>>> funcs = [bark, str.lower, str.capitalize]
>>> funcs
[<function yell at 0x10ff96510>,
<method 'lower' of 'str' objects>,
<method 'capitalize' of 'str' objects>]
```

دسترسی به اشیای تابع ذخیره شده در لیست، مانند دسترسی به هر نوع شیء دیگری در زبان پایتون است.

```
>>> for f in funcs:
        print(f, f('hey there'))
<function yell at 0x10ff96510> 'HEY THERE!'
<method 'lower' of 'str' objects> 'hey there'
<method 'capitalize' of 'str' objects> 'Hey there'
```

حتی می‌توانید شیء تابع ذخیره شده در لیست را به صورت مستقیم فراخوانی کنید.

```
>>> funcs[0]('heyho')
'HEYHO!'
```

۳-۱-۳. توابع را می‌توان به توابع دیگر پاس داد

چون توابع بصورت اشیاء هستند، پس می‌توانید آن‌ها را به عنوان آرگومان های ورودی به توابع دیگر پاس دهید. در اینجا تابع greet را نوشته‌ایم که به عنوان آرگومان ورودی یک تابع را دریافت می‌کند و سپس یک رشته را به عنوان ورودی به این تابع می‌دهد. در نهایت نتیجه را نمایش می‌دهد.

```
def greet(func):
    greeting = func('Hi, I am a Python program')
    print(greeting)
```

می‌توانید با انتقال تابع به توابع مختلف، بر نتیجه اثر بگذارید. برای مثال در قطعه کد زیر تابع bark را به عنوان ورودی تابع greet دادیم تا در نتیجه متغیر greeting تاثیر بگذاریم.

```
>>> greet(bark)
'HI, I AM A PYTHON PROGRAM!'
```

البته، می‌توانید تابع جدیدی را نیز برای ایجاد ویژگی‌های مختلف تعریف کنید. برای مثال، اگر نمی‌خواهید برنامه پایتون شما مانند اپتیموس پرایم[1] به نظر برسد، تابع whisper زیر ممکن است عملکرد بهتری داشته باشد. (زیرا به جای حروف بزرگ، حروف کوچک یک متن را نمایش می‌دهد)

```
>>> def whisper(text):
        return text.lower() + '...'

>>> greet(whisper)
'hi, i am a python program...'
```

1 Optimus Prime – یک شخصیت داستانی

توانایی انتقال اشیاء تابع به صورت آرگومانهایی به توابع دیگر، ویژگی قدرتمندی است. این کار به شما اجازه می‌دهد تا انتزاع را کنار بگذارید و بر رفتار در برنامه‌های خود تمرکز کنید. در این مثال، تابع greet یکسان می‌ماند، اما می‌توانید با انتقال رفتارهای مختلف، بر خروجی آن اثر بگذارید.

توابعی که می‌توانند سایر توابع را بعنوان آرگومان‌ها بپذیرند، توابع مرتبه بالاتر[1] نیز نامیده می‌شوند. آنها الزامی برای سبک برنامه‌نویسی تابعی[2] هستند.

مثال کلاسیک برای توابع مرتبه بالاتر در پایتون، تابع داخلی map در زبان پایتون است. این شیء تابع و تکرار شونده[3] را می‌گیرد و بعد تابع را روی هر عنصری در تکرارشونده فراخوانی می‌کند و همانطور که پیش می‌رود، نتایج را بدست می آورد.

در اینجا نحوه قالب‌بندی توالی احوالپرسی را بصورت یکجا با نگاشتی از تابع bark نوشته ایم که نتیجه در نهایت در یک لیست ذخیره می‌شود.

```
>>> list(map(bark, ['hello', 'hey', 'hi']))
['HELLO!', 'HEY!', 'HI!']
```

همانطور که دیدید نگاشت در کل لیست انجام شده و تابع bark به هر عنصر لیست اعمال شد. در نتیجه، حالا یک شی لیست جدید با رشته‌های اصلاح شده احوالپرسی را داریم.

۳-۱-٤. توابع را می‌توان بصورت تو در تو قرار داد

شاید تعجب‌آور باشد، پایتون به توابع این اجازه را می‌دهد تا در داخل توابع دیگر تعریف شوند. این‌ها اغلب توابع تو در تو[4] یا توابع داخلی[5] نامیده می‌شوند. در اینجا مثالی می‌زنیم.

---

1 Higher-order functions
2 Functional programming style
3 Iterable
4 Nested functions
5 Inner functions

```
>>> def speak(text):
        def whisper(t):
            return t.lower() + '...'
        return whisper(text)

>>> speak('Hello, World')
'hello, world...'
```

هر بار که تابع speak را فراخوانی می‌کنید، پایتون تابع داخلی جدید whisper را تعریف می‌کند و بلافاصله پس از آن فرا می‌خواند. مغز من درست کمی از اینجا به بعد شروع به خارش می‌کند اما، در کل، این هنوز موضوع نسبتا ساده ای است.

کمی با دقت بیشتری به موضوع نگاه کنیم، اگر بخواهیم تابع whisper را فراخوانی کنیم چه اتفاقی خواهد افتاد؟

```
>>> whisper('Yo')
NameError:
"name 'whisper' is not defined"

>>> speak.whisper
AttributeError:
"'function' object has no attribute 'whisper'"
```

اما اگر واقعا بخواهید به تابع تو در توی whisper در خارج از تابع speak دسترسی پیدا کنید، چه اتفاقی می‌افتد؟ خب، توابع اشیاء هستند، پس می‌توانید تابع داخلی را به فراخواننده تابع والد برگردانید.

برای مثال، در ادامه تابعی داریم که دو تابع داخلی را تعریف می‌کند. بسته به آرگومان منتقل شده به تابع سطح بالا، یکی از توابع داخلی را انتخاب می‌کند و آن را به فراخواننده برمی‌گرداند.

```
def get_speak_func(volume):
    def whisper(text):
        return text.lower() + '...'
    def yell(text):
        return text.upper() + '!'
    if volume > 0.5:
        return yell
    else:
        return whisper
```

توجه داشته باشید به اینکه چگونه تابع get_speak_func در واقع هیچ یک از توابع داخلی خود را فراخوانی نمی‌کند. فقط تابع داخلی مناسب را بر پایه آرگومان volume انتخاب می‌کند و بعد شیء تابع را برمی‌گرداند.

```
>>> get_speak_func(0.3)
<function get_speak_func.<locals>.whisper at
0x10ae18>

>>> get_speak_func(0.7)
<function get_speak_func.<locals>.yell at 0x1008c8>
```

البته، می‌توانید ادامه دهید و با استفاده از تابع برگشت یافته، مستقیما یا با تخصیص آن به متغیر، فراخوانی را انجام دهید.

```
>>> speak_func = get_speak_func(0.7)
>>> speak_func('Hello')
'HELLO!'
```

اجازه دهید چند لحظه در این مورد تأمل کنیم. این بدان معناست که توابع نه تنها می‌توانند رفتارها را از طریق ورودی‌ها بپذیرند، بلکه می‌توانند رفتارها را نیز برگردانند. به نظرتان چطور است؟

همانطور که می‌دانید، گاهی درک بعضی از مفاهیم برنامه‌نویسی که تودرتو هستند کمی دشوار است. قصد دارم قبل از ادامه نوشتن کتاب، استراحت کوتاهی کنم و دمی به یک فنجان قهوه بزنم. (پیشنهاد می‌کنم شما هم این کار را انجام دهید)

۳-۱-۵. توابع می‌توانند حالت‌های محلی را ثبت کنند

در قسمت‌های قبل دیدید که چگونه توابع می‌توانند توابع داخلی داشته باشند، توابع دیگری را به عنوان ورودی بگیرند یا به عنوان خروجی برگردانند.

حال کمربند ایمنی خود را محکم ببندید چون می‌خواهیم کمی دیوانه‌بازی در آوریم. قصد داریم به محدوده برنامه‌نویسی تابعی عمیق‌تر وارد شویم. (استراحت و قهوه که یادتان نرفت؟)

توابع نه تنها می‌توانند سایر توابع را پاس کاری کنند، بلکه توابع داخلی می‌توانند از برخی حالات تابع والد باخبر باشند و مقادیری را حمل کنند. خب، این به چه معنی است؟

برای توضیح این مثال، می‌خواهم مثال قبلی get_speak_func را بازنویسی کنم. نسخه جدید این تابع، آرگومان‌های volume و text را می‌گیرد و تابع برگشتی را بلافاصله فراخوانی می‌کند.

```
def get_speak_func(text, volume):
    def whisper():
        return text.lower() + '...'
    def yell():
        return text.upper() + '!'
    if volume > 0.5:
        return yell
    else:
        return whisper


>>> get_speak_func('Hello, World', 0.7)()
'HELLO, WORLD!'
```

حال به توابع داخلی whisper و yell خوب نگاه کنید. دقت کرده‌اید که چگونه آنها دیگر پارامتر text را به عنوان ورودی ندارند؟ اما هنوز می‌توانند به پارامتر text تعریف شده در تابع والد دسترسی پیدا کنند. در حقیقت، اینطور به نظر می‌رسد که آنها مقدار آرگومان‌های والد را ثبت کرده و به یاد می‌آورند.

توابعی که این کار را انجام می‌دهند، بستارهای معنایی[1] نامیده می‌شوند (یا بطور خلاصه، فقط بستارها[2] نامیده می‌شوند). بستار، مقادیر را از تابع والد خود به یاد می‌آورد، حتی اگر جریان اجرای برنامه دیگر در این تابع نباشد. به زبان ساده، این بدان معناست که توابع نه تنها می‌توانند رفتارها را برگردانند بلکه می‌توانند از قبل، آن رفتارها را پیکربندی کنند. برای توضیح این ایده، مثالی می‌زنم.

```
>>> def make_adder(n):
        def add(x):
           return x + n
        return add

>>> plus_3 = make_adder(3)
>>> plus_5 = make_adder(5)

>>> plus_3(4)
7
>>> plus_5(4)
9
```

در این مثال، make_adder به عنوان دیزاین پترن factory[3] برای ایجاد و پیکربندی تابع add عمل می‌کند. دقت کنید که چگونه تابع add می‌تواند به مقدار n از قبل مقداردهی شده، دسترسی پیدا کند. بهتر است زمانی را برای درک عمیق این مثال اختصاص دهید.

۳-۱-۶. اشیاء می‌توانند مانند توابع رفتار کنند

با اینکه همه توابع در پایتون اشیاء هستند، اما عکس آن درست نیست. اشیاء توابع نیستند. اما می‌توان آن‌ها را فراخوانی کرد که به شما اجازه می‌دهد در اغلب موارد با آن ها مانند توابع رفتار کنید.

---

1 Lexical closures

2 Closures

۳ Creational Design Patterns در دسته Factory مراجعه شود به دیزاین پترن

اگر شی قابل فراخوانی باشد به این معنی است که می‌توانید از سینتکس پرانتز باز و بسته برای آن استفاده کنید و در صورت نیاز مقادیری را به شی پاس دهید. این قابلیت با استفاده از متد جادویی __call__ انجام می‌شود.

```
class Adder:
    def __init__(self, n):
        self.n = n
    def __call__(self, x):
        return self.n + x


>>> plus_3 = Adder(3)
>>> plus_3(4)
7
```

در پشت صحنه، زمانی که یک شی را همانند تابع فراخوانی می‌کنید، پایتون تلاش می‌کند تا تابع __call__ درون شی را فراخوانی کند.

البته همه اشیاء قابل فراخوانی نیستند. به همین دلیل یک تابع callable داخلی وجود دارد تا بررسی کند که آیا شیء قابل فراخوانی است یا خیر.

```
>>> callable(plus_3)
True
>>> callable(yell)
True
>>> callable('hello')
False
```

۳-۱-۷. نکات کلیدی توابع به عنوان شهروندان درجه اول

- همه چیز، از جمله توابع در پایتون، شی است. می‌توانید آن‌ها را به متغیرها اختصاص دهید، آن‌ها را در ساختارهای داده ذخیره کنید و به سایر توابع منتقل کنید یا از سایر توابع برگردانید.

- توابع، به عنوان شهروندان درجه اول به شما اجازه می‌دهند تا انتزاع را کنار بگذارید و بر رفتارهای برنامه خود توجه کنید.
- توابع را می‌توان به صورت تو در تو ایجاد کرد و آن‌ها می‌توانند برخی حالات تابع والد را ثبت کرده و حمل کنند. توابعی که این کار را انجام می‌دهند، بستارها نامیده می‌شوند.
- اشیاء را نیز می‌توان قابل فراخوانی کرد. در اغلب موارد، این کار به شما اجازه می‌دهد تا با آن‌ها مانند توابع رفتار کنید.

۳-۲. لامبداها، توابع تک جمله‌ای

کلیدواژه lambda در پایتون، میانبری برای تعریف توابع ناشناس کوچک است. توابع لامبدا درست مانند توابع عادی تعریف شده با کلیدواژه def رفتار می‌کنند.

برای مثال، اینگونه تابع ساده لامبدا را که عملیات جمع انجام می‌دهد را تعریف می کنیم.

```
>>> add = lambda x, y: x + y
>>> add(5, 3)
8
```

می‌توانید همان تابع جمع را با کلیدواژه def تعریف کنید، اما کمی طولانی‌تر خواهد بود.

```
>>> def add(x, y):
        return x + y
>>> add(5, 3)
8
```

حال ممکن است کنجکاو باشید که "چرا در مورد لامبداها این همه جنجال وجود دارد؟ اگر آنها نسخه کمی مختصر و مفیدتری از تعریف توابع با def هستند، پس چه مشکلی وجود دارد؟"

به مثال زیر نگاهی بیندازید و در حین انجام این کار، عبارات بیان تابع[1] را در ذهن داشته باشید.

```
>>> (lambda x, y: x + y)(5, 3)
8
```

بسیار خب، اینجا چه خبر است؟ فقط از لامبدا برای تعریف تابع جمع درون خطی استفاده کردم و بعد بلافاصله آن را با آرگومانهای ۵ و ۳ فراخوانی کردم.

به لحاظ مفهومی، عبارت لامبدا lambda x, y: x + y همانند اعلام تابع با def است، اما فقط درون خطی نوشته شده است. تفاوت اصلی در اینجا این است که لازم نیست نامی را برای تابع انتخاب کنم. فقط گفتم که می‌خواهم محاسبه سریعی انجام دهم و بعد بلافاصله آن را با فراخوانی عبارت لامبدا مانند تابع های عادی اجرا کردم.

پیش از پرداختن به مورد بعدی، ممکن است بخواهید کمی در مورد مثال قبل فکر کنید تا واقعا مفهوم آن را دریابید. هنوز به یاد دارم که این کار برای من مدتی طول کشید، چون نمی‌توانستم ذهنم را روی آن متمرکز کنم. بنابراین نگران صرف چند دقیقه در مفسر پایتون در این باره نباشید. این کار ارزشش را دارد.

تفاوت نحوی دیگری بین تعاریف لامبدا و تابع عادی وجود دارد. توابع لامبدا به عبارت محاسباتی واحدی محدود می‌شوند. این بدان معنی است که تابع لامبدا نمی‌تواند از جملات[2] یا حاشیه‌نویسی‌ها[3] (حتی بعنوان خروجی تابع) استفاده کند.

پس چگونه می‌توانید مقادیر را از لامبداها برگردانید؟ پس از اجرای تابع لامبدا، عبارت آن ارزیابی می‌شود و بعد به طور خودکار نتیجه عبارت باز می‌گردد، بنابراین همیشه برگشت ضمنی در این توابع وجود دارد. به همین دلیل است که برخی افراد به لامبداها توابع تک جمله‌ای می‌گویند.

---

1 Function expression
2 statements
3 annotations

۳-۲-۱. لامبداهایی که می‌توانید استفاده کنید

چه زمانی باید از توابع لامبدا در کد خود استفاده کنید؟ به لحاظ فنی، هر زمان که بخواهید یک شی سریع از تابع ناشناسی بسازید، می‌توانید از عبارت لامبدا استفاده کنید و چون لامبداها می‌توانند ناشناس باشند، پس لزومی ندارد ابتدا آن‌ها را نامگذاری کنید.

این کار می‌تواند میانبری مفید و "غیر بروکراتیک" برای تعریف تابع در پایتون ارائه کند. بیشترین موردی که اغلب برای لامبداها بکار می‌بریم، نوشتن توابع کوتاه برای مرتب‌سازی موارد قابل تکرار با کلید مشخص است.

```
>>> tuples = [(1, 'd'), (2, 'b'), (4, 'a'), (3, 'c')]
>>> sorted(tuples, key=lambda x: x[1])
[(4, 'a'), (2, 'b'), (3, 'c'), (1, 'd')]
```

در مثال فوق، لیستی از tuple ها را با مقدار دوم (اندیس یک) در هر tuple مرتب می‌کنیم. در این حالت، تابع لامبدا سریعا روشی برای اصلاح و مرتب‌سازی فراهم می کند. در اینجا نمونه دیگری از مرتب‌سازی است که می‌توانید در مورد آن فکر کنید.

```
>>> sorted(range(-5, 6), key=lambda x: x * x)
[0, -1, 1, -2, 2, -3, 3, -4, 4, -5, 5]
```

امیدوارم بتوانید نحوه استفاده از لامبدا برای انعطاف‌پذیری بیشتر در محاسبات را ببینید. آیا می‌خواهید توالی را با کلیدی که بصورت دلخواه محاسبه شده مرتب‌سازی کنید؟ مشکلی نیست. حالا نحوه انجام این کار را می‌دانید.

نکته جالب دیگری درباره لامبداها وجود دارد: دقیقا مانند توابع تو در تو، لامبداها نیز می‌توانند همانند بستارهای معنایی عمل کنند.

بستار معنایی[1] چیست؟ این فقط نامی زیبا برای تابعی است که مقادیر را از حوزه محصور شده فراخوانی می‌کند، حتی وقتی که جریان برنامه دیگر در این حوزه نباشد. در اینجا مثال (نسبتا دانشگاهی) برای توضیح این ایده ارائه شده است.

1 Lexical Closure

```
>>> def make_adder(n):
        return lambda x: x + n
>>> plus_3 = make_adder(3)
>>> plus_5 = make_adder(5)

>>> plus_3(4)
7
>>> plus_5(4)
9
```

در مثال فوق، لامبدای x+n می‌تواند به مقدار n دسترسی پیدا کند اگر در تابع make_adder (حوزه محصور) تعریف شده باشد.

گاهی، استفاده از تابع لامبدا بجای تابع تو در توی اعلام شده با کلیدواژه def می تواند هدف برنامه‌نویس را با وضوح بیشتری بیان کند. صادقانه بگویم، این اتفاق معمولی نیست، حداقل در سبک کدی که مایلم بنویسم، اینطور نیست. پس کمی بیشتر در مورد آن صحبت کنیم.

۳-۲-۲. شاید شما نباید ...

از یک طرف، امیدوارم که این فصل شما را به بررسی توابع لامبدای پایتون علاقمند کند. از طرف دیگر، احساس می‌کنم وقت آن رسیده که این هشدار را بدهم که: توابع لامبدا باید کمتر و با احتیاط بیشتری استفاده شوند.

می‌دانم که قسمت زیادی از کد را با استفاده از لامبداهایی که "جالب" به نظر می رسند، نوشته‌ام، اما در واقع این وظیفه من و همکارانم بوده است. اگر ترغیب شده‌اید تا از لامبدا استفاده کنید، چند ثانیه (یا چند دقیقه) وقت بگذارید تا فکر کنید که آیا این واقعا تمیزترین و پایدارترین روش برای دستیابی به نتیجه مطلوب است.

برای مثال، انجام کاری مانند این برای صرفه‌جویی در دو خط کد احمقانه است. قطعا، به لحاظ فنی کار می‌کند و "ترفند" بسیار مناسبی است، اما افرادی که در آینده قرار است در زمان کمی یک مشکل را در این کد برطرف کنند، بسیار سردرگم خواهند شد.

```
# Harmful:
>>> class Car:
        rev = lambda self: print('Wroom!')
        crash = lambda self: print('Boom!')

>>> my_car = Car()
>>> my_car.crash()
'Boom!'
```

در مورد ساختارهای پیچیده map() یا filter() با استفاده از لامبداها نیز احساس مشابهی دارم. به خوانایی دو قطعه کد زیر دقت کنید و آن‌ها را باهم مقایسه کنید.

```
# Harmful:
>>> list(filter(lambda x: x % 2 == 0, range(16)))
[0, 2, 4, 6, 8, 10, 12, 14]

# Better:
>>> [x for x in range(16) if x % 2 == 0]
[0, 2, 4, 6, 8, 10, 12, 14]
```

اگر فکر می‌کنید با استفاده از lambda ها کار پیچیده‌ای کرده‌اید و از این کار لذت می‌برید، پیشنهاد می‌کنم که این کار پیچیده را به صورت یک تابع عادی با خوانایی کد بیشتری انجام دهید. صرفه‌جویی در فشردن کلیدها در درازمدت اهمیتی ندارد، اما همکاران شما (و خود شما در آینده) کد تمیز و خوانا را بیشتر از یک جادوگر ماهر می‌توانید تغییر دهید.

۳-۲-۳. نکات کلیدی لامبداها

- توابع لامبدا، توابع تک جمله‌ای هستند که لزوما محدود به نام نیستند (ناشناس هستند).
- توابع لامبدا نمی‌توانند از عبارات معمولی پایتون استفاده کنند و همیشه شامل مقدار برگشتی ضمنی هستند.

- همیشه از خود بپرسید: آیا استفاده از توابع عادی با نام گذاری صحیح، خوانایی بیشتری دارد؟ اگر پاسخ بله است، از لامبدا استفاده نکنید.

۳-۳. قدرت اعجاب انگیز دکوریتورها[1]

تصور کنید ۳۰ تابع با منطق های پیچیده کسب و کار در برنامه خود نوشته اید. در یک صبح شنبه بارانی، رئیس کنار میز شما می‌آید و می‌گوید:

"صبح شنبه بخیر! آن گزارشهای TPS را به خاطر داری؟ نیاز دارم تا داده های ورودی و خروجی در هر مرحله تولید گزارش را به صورت کامل ثبت کنید. برای اهداف حسابرسی با شرکت XYZ این اطلاعات باید آماده شوند. به آنها گفتم که می‌توانیم این کار را تا روز دوشنبه ارسال کنیم".

این درخواست فشار خون شما را افزایش خواهد داد یا باعث آرامش شما خواهد شد. بستگی به این دارد که آیا شما از دکوریتورهای پایتون به درک عمیقی دست یافته‌اید یا خیر. بدون دکوریتورها، ممکن است سه روز آینده را صرف اصلاح هر یک از این ۳۰ تابع کنید و با فراخوانی های دستی ثبت وقایع و گزارش‌گیری آنها را درهم بریزید. لحظات سرگرم کننده ای است، اینطور نیست؟ با این حال، اگر دکوریتورهای پایتون را بشناسید، با آرامش به رئیس خود لبخند می‌زنید و می‌گویید:

"جیم نگران نباش، امروز تا ساعت ۲ بعد از ظهر این کار را انجام خواهم داد".

درست پس از آن، کدی را برای دکوریتور عمومی @audit_log تایپ می‌کنید (طول این کد تقریبا ۱۰ خط است) و به سرعت در جلوی تعریف تابع قرار می‌دهید. بعد کد خود را ارسال می‌کنید و یک فنجان قهوه دیگر می‌خورید. لذت بخش است، نه؟

1 Decorators

شاید در اینجا کمی بزرگنمایی می‌کنم. دکوریتورها می‌توانند قدرتمند و اعجاب انگیز باشند. در این مطلب تا جایی پیش می‌رویم که بگوییم درک دکوریتورهای پایتون، نقطه عطفی برای هر برنامه‌نویس سختکوش پایتون است. درک صحیح دکوریتورها نیاز به درک کاملی از چندین مفهوم پیشرفته در زبان پایتون، از جمله ویژگی توابع کلاس اول دارند.

معتقدم که نتایج مثبت درک عمیق نحوه عملکرد دکوریتورها در پایتون، می‌تواند بسیار زیاد باشد.

۳-۳-۱. قدرت دکوریتورها

دکوریتور پایتون به شما امکان گسترش و اصلاح رفتارهای قابل فراخوانی (مانند توابع و کلاس ها) بدون اصلاح دائمی پارامتر قابل فراخوانی را می‌دهد. هر فعالیتی که به اندازه کافی عمومی باشد و بتوانید آن را به کلاس یا رفتارهای موجود اضافه کنید، یک مورد عالی برای استفاده در دکوریتور محسوب می‌شود. کلمه انگلیسی Decorator معادل فارسی تزئین است و در ادامه گاهی از کلمه تزئین به جای دکوریتور استفاده خواهیم کرد.

به زبان ساده دکوریتورهای پایتون امکان اضافه کردن رفتار جدید به اشیا را می‌دهند. این رفتارهای جدید در قالب بسته بندی‌های جدید شی خواهند بود.

کارهای تکراری که با استفاده از Decorator پایتون می‌توانید انجام دهید شامل موارد زیر است:

- ثبت وقایع و گزارش گیری
- اجرای کنترل دسترسی و احراز هویت
- تعیین زمان اجرای توابع
- محدودیت های پراستفاده
- Caching (دخیره سازی موقت اطلاعات) و...

حال، چرا باید به کاربرد دکوریتورها در پایتون تسلط داشته باشید؟ بدون درک این کاربرد چیزهایی که در ابتدا گفتیم کاملاً انتزاعی به نظر می‌رسد و فهمیدن این موضوع که دکوریتورها چقدر می‌توانند در کار روزانه یک توسعه دهنده پایتون سودمند باشند، دشوار خواهد بود.

قطعا، فکر کردن برای اولین بار در مورد دکوریتورها کمی دشوار است، اما ویژگی بسیار سودمندی است که اغلب در فریمورک های شخص ثالث و در کتابخانه های استاندارد پایتون با آن مواجه خواهید شد. در این مطلب نهایت تلاشم را خواهم کرد تا بصورت گام به گام شما را با دکوریتور ها آشنا کنم و تعدادی از ویژگی های اعجاب انگیز دکوریتورها را ببینیم.

قبل از بررسی این موضوع، اکنون لحظه ای عالی برای یادآوری در مورد ویژگی های توابع کلاس اول (یا توابع به عنوان شهروندان درجه اول) در پایتون است. مهمترین نکات اساسی "توابع کلاس اول "برای درک دکوریتورها عبارتند از:

- توابع اشیاء هستند .توابع را می‌توان درون متغیرها ذخیره کرد، یا توابع را به عنوان ورودی و خروجی به توابع دیگر داد.
- توابع را می‌توان در داخل توابع دیگر قرار داد .تابع فرزند می‌تواند از وضعیت محلی تابع والد استفاده کند(lexical closures) .

خب، برای انجام این کار آماده هستید؟ پس شروع کنیم.

۳-۳-۲. مبانی دکوریتورها

حالا، دکوریتورها واقعا چه چیزی هستند؟ دکوریتورها تابع را تزئین یا بسته‌بندی می‌کنند و به شما اجازه می‌دهند تا قبل و بعد از اجرای تابع بسته‌بندی شده، کد دلخواهی را اجرا کنید.

دکوریتور پایتون به شما امکان تعریف بلوک های قابل استفاده مجدد را می‌دهد تا بتوانید رفتار سایر توابع را تغییر یا گسترش دهید. رفتار تابع فقط هنگام تزئین آن تغییر می‌کند.

پیاده سازی دکوریتور ساده چگونه است؟ به بیان ساده، دکوریتور پایتون یک تابع قابل فراخوانی است که یک تابع قابل فراخوانی دیگری را به عنوان ورودی دریافت می‌کند و می‌تواند آن را به عنوان خروجی برگرداند.

تابع زیر دارای این ویژگی است و می‌تواند ساده‌ترین دکوریتوری باشد که می‌توانید به صورت زیر بنویسید.

```
def null_decorator(func):
    return func
```

همانطور که مشاهده می‌کنید null_decorator یک تابع قابل فراخوانی است، و مقدار قابل فراخوانی func که یک تابع است را به عنوان ورودی دریافت می‌کند و آن تابع را بدون ایجاد تغییری در خروجی بر می‌گرداند.

اجازه دهید از آن برای تزئین (بسته‌بندی[1] یا Decorate) تابع دیگری استفاده کنیم.

```
>>> def greet():
        return 'Hello!'

>>> greet = null_decorator(greet)

>>> greet()
'Hello!'
```

در این مثال، تابع greet را تعریف کرده ام و بعد بلافاصله آن را از طریق تابع null_decoratorتزئین کرده ام. می‌دانم که این موضوع چندان مفید به نظر نمی‌رسد. منظورم این است که دکوریتور تهی را به گونه‌ای خاص طراحی کرده‌ایم تا بی‌فایده

1 wrap

باشد، درست است؟ این مثال نحوه کار سینتکس دکوریتور پایتون برای حالت ویژه را توضیح خواهد داد.

به جای فراخوانی صریح null_decorator روی greet و جایگزینی شی greet، می‌توانید از @decorator پایتون برای تزئین مناسب تر و زیباتر تابع استفاده کنید.

```
@null_decorator
def greet():
    return 'Hello!'

>>> greet()
'Hello!'
```

قرار دادن خط @null_decorator در مقابل تعریف تابع، همان چیزی است که به عنوان اولین تعریف تابع ارائه کردیم و بعد از طریق دکوریتور تابع را اجرا کردیم. استفاده از @decorator فقط فرم دگرگون یافته و میانبری برای الگوی پرکاربرد دکوریتور پایتون است.

توجه داشته باشید که استفاده از @decorator تابع را بلافاصله در زمان تعریف تزئین می‌کند. این سبب دسترسی دشوار به نسخه اصلی تزئین نشده می‌شود. بنابراین ممکن است برای حفظ امکان فراخوانی تابع تزئین نشده، گزینه های مختلفی را برای تزئین دستی انتخاب کنید. (در ادامه در رابطه با این ویژگی توضیحات بیشتری خواهیم داد)

۳-۳-۳. دکوریتورها رفتار توابع را اصلاح می‌کنند

حالا که کمی بیشتر با سینتکس دکوریتور آشنا شدید، اجازه دهید دکوریتور دیگری بنویسیم که در واقع کاری را انجام می‌دهد و رفتار تابع تزئین شده را اصلاح می‌کند. در اینجا دکوریتور کمی پیچیده‌تری وجود دارد که نتیجه تابع تزئین شده را به حروف بزرگ تبدیل می‌کند.

```
def uppercase(func):
    def wrapper():
        original_result = func()
        modified_result = original_result.upper()
        return modified_result
    return wrapper
```

این دکوریتور بجای برگرداندن ورودی تابع مانند دکوریتور تهی، تابع جدیدی را on-the-fly تعریف می‌کند و از آن برای بسته‌بندی تابع ورودی استفاده می‌کند تا تابع، رفتار خود را در زمان فراخوانی تغییر دهد. تابع wrapper به تابع ورودی تزئین نشده دسترسی دارد و می‌تواند قطعه کدی را قبل و بعد از فراخوانی تابع ورودی اجرا کند. (به لحاظ فنی، اصلا نیازی به فراخوانی تابع ورودی ندارد)

به این توجه داشته باشید که چرا تابع تزئین شده اجرا نشده است. در واقع، فراخوانی تابع ورودی در این مرحله منطقی نخواهد بود. می‌خواهیم وقتی که دکوریتور در نهایت فراخوانی می‌شود، بتواند رفتار تابع ورودی خود را اصلاح کند.

ممکن است بخواهید یکی دو دقیقه در مورد آن فکر کنید. می‌دانم که این مسئله چقدر می‌تواند دشوار به نظر برسد، اما قول می‌دهم که با کمک یکدیگر این مسئله را حل خواهیم کرد. وقت آن رسیده که در عمل دکوریتور پایتون که حروف را به حروف بزرگ تبدیل می‌کند را ببینیم. اگر با استفاده از آن تابع اصلی greet را تزئین کنید، چه چیزی رخ خواهد داد؟

```
@uppercase
def greet():
    return 'Hello!'

>>> greet()
'HELLO!'
```

امیدوارم این همان نتیجه ای که انتظار آن را داشتید، باشد. اجازه دهید نگاهی دقیق‌تر به آنچه که در اینجا اتفاق افتاده، بیاندازیم.

برخلاف null_decorator، دکوریتور @uppercase هنگامی که تابع را تزئین می کند، شیء متفاوتی از تابع را بر می گرداند.

```
>>> greet
<function greet at 0x10e9f0950>

>>> null_decorator(greet)
<function greet at 0x10e9f0950>

>>> uppercase(greet)
<function uppercase.<locals>.wrapper at 0x76da02f28>
```

در اینجا نکته ی کوچکی وجود دارد، دکوریتور پایتون باید این کار را انجام دهد تا در هنگام فراخوانی نهایی، رفتار تابع تزئین شده را اصلاح کند .دکوریتور @uppercase خود نیز تابع است .تنها روش برای اثرگذاری بر "رفتار آتی " تابع ورودی که آن را تزئین می کند، جایگزینی (یا بسته‌بندی) تابع ورودی است.

به همین دلیل، تابع uppercase تابع دیگری را که بعدا می‌توان فراخوانی کرد، را تعریف کرده و برمی گرداند، تابع ورودی اصلی را اجرا کرده و نتیجه آن را اصلاح می کند.

دکوریتورها رفتار تابع قابل فراخوانی را از طریق بستار بسته ساز (lexical closure) اصلاح می کنند، پس نباید تابع اصلی را به صورت دائم اصلاح کنند .تابع قابل فراخوانی اصلی به صورت دائمی اصلاح نمی‌شود بلکه فقط رفتار آن فقط هنگام فراخوانی تغییر می کند.

این کار به شما امکان اضافه کردن بلوک های ساختاری قابل استفاده مجدد، مانند ثبت وقایع و گزارش گیری و سایر ابزارهای دقیق، به توابع و کلاسهای موجود را می دهد .این کار دکوریتورها را به چنان ویژگی قدرتمندی در پایتون تبدیل می کند که اغلب در کتابخانه استاندارد و کتابخانه‌های شخص ثالث مورد استفاده قرار می گیرد.

۳-۳-٤. یک وقفه کوتاه برای درک بهتر دکوریتورها

اگر احساس می‌کنید در این مرحله به اندازه یک فنجان قهوه یا پیاده روی در اطراف میز کارتان نیاز دارید، این کاملاً طبیعی است .به نظر من، درک مفهوم دکوریتورها یکی از دشوارترین کارها در پایتون است.

لطفا عجله نکنید و نگران فهمیدن جواب مسئله نباشید. تکه کدهای بررسی شده را به صورت جداگانه در یک مفسر پایتون اجرا کنید و نتیجه ی آن را مشاهده کنید تا بهتر درک کنید. می‌دانم که می‌توانید این کار را انجام دهید!

۳-۳-۵. استفاده از چند دکوریتور در یک تابع

شاید تعجب آور نباشد که می‌توانید بیش از یک دکوریتور را روی تابع اعمال کنید. این کار اثرات آن‌ها را انباشته می‌کند و این همان چیزی است که دکوریتورها را به اندازه بلوک های چندبخشی قابل استفاده مجدد سودمند می‌کند.

مثالی در این مورد بررسی خواهیم کرد. دو دکوریتور زیر، رشته خروجی تابع تزئین شده با برچسبهای HTML را بسته بندی می‌کنند. با نگاهی بر نحوه قرارگیری تو در توی تگ ها، می‌توان فهمید که دکوریتور پایتون از چه فرآیندی برای اعمال چندین دکوریتور استفاده می‌کند.

```
def strong(func):
    def wrapper():
        return '<strong>' + func() + '</strong>'
    return wrapper

def emphasis(func):
    def wrapper():
        return '<em>' + func() + '</em>'
    return wrapper
```

حال اجازه دهید این دو دکوریتور را انتخاب کنیم و آنها را در تابع greet خود اعمال کنیم. می‌توانید از syntax@ معمولی برای آن استفاده کنید و چندین دکوریتور را در بالای یک تابع واحد انباشته کنید.

```
@strong
@emphasis
def greet():
    return 'Hello!'
```

در صورت اجرای تابع تزیین شده، انتظار دارید کدام ورودی را ببینید؟ آیا دکوریتور emphasis@ ابتدا تگ خود را اضافه خواهد کرد، یا strong@ اولویت دارد؟ هنگامی که تابع تزیین شده را فراخوانی می‌کنید، چه اتفاقی رخ خواهد داد؟

```
>>> greet()
'<strong><em>Hello!</em></strong>'
```

این نتیجه به وضوح نشان می‌دهد که دکوریتورهای چند گانه چگونه از پایین به بالا عمل کرده‌اند. ابتدا تابع ورودی با دکوریتور emphasis@ بسته بندی شد و بعد تابع تزئین شده حاصل با دکوریتور strong@ دوباره بسته بندی شد.

برای اینکه رفتار پایین به بالا در حافظه تان باقی بماند میخواهم طریقه اجرای دکوریتورهای چندگانه را در هسته پایتون توضیح دهم .دکوریتورهای پایتون با استفاده از پشته پیاده سازی می‌شود و ما این رفتار را decorator stacking می‌نامیم .اولین پشته را در پایین ایجاد می‌کنیم و بعد بلوک های کد جدید را از بالا به آن اضافه می‌کنیم تا اجرای برنامه راه خود را از پایین به سمت بالا پیش ببرد.

اگر مثال بالا را تجزیه کنید و از syntax@ برای اعمال به دکوریتورها خودداری کنید، زنجیره فراخوانی های مربوط به تابع دکوریتور به این شکل خواهد بود.

```
decorated_greet = strong(emphasis(greet))
```

همانطور که می‌بینید ابتدا دکوریتور emphasis اعمال می‌شود و بعد تابع بسته بندی شده حاصل دوباره با دکوریتور strong بسته بندی می‌شود.

این همچنین به این معنی است که در نهایت سطوح عمیق پشته سازی دکوریتورها بر میزان کارایی اثر برنامه خواهند گذاشت، چون مدام فراخوانی های توابع تو در تو انجام می‌دهند. در عمل، معمولا انجام این کار مشکل نخواهد بود، اما اگر کارایی کد نوشته شده برایتان مهم است باید مرتبه زمانی عملیات decoration را در نظر بگیرید.

۳-۳-۶. دکوریت توابعی که آرگومان‌های متعددی می‌پذیرند

تاکنون همه نمونه ها فقط تابع ساده greet که هیچ آرگومانی ندارد را تزئین کرده اند. تا این لحظه، دکوریتورهایی که در این مطلب دیدید لازم نیست که با آرگومان های ارسال به تابع ورودی سر و کار داشته باشند. اگر یکی از این دکوریتورها را به تابعی که آرگومان هایی را می‌گیرد اعمال کنید، کارکرد درستی نخواهد داشت.

چگونه تابعی که آرگومانهای دلخواه می‌گیرد را تزئین می‌کنند؟

اینجاست که به ویژگی *args و **kwargs پایتون برای مقابله با تعداد متغیرهای آرگومان احتیاج خواهیم داشت. دکوریتور پایتون زیر به اسم proxy از این مزیت استفاده می‌کند.

```
def proxy(func):
    def wrapper(*args, **kwargs):
        return func(*args, **kwargs)
    return wrapper
```

دو مورد قابل توجه در مورد این دکوریتور وجود دارد:

- با استفاده از عملگرهای * و ** در ورودی تابع wrapper، همه آرگومان های ورودی را جمع آوری می‌کند و آن‌ها را به صورت شی args و kwargs ذخیره می‌کند.

- سپس تابع wrapper آرگومان های جمع آوری شده توسط عملکرد * و ** را بازگشایی می‌کند[1] و آن را به تابع ورودی ارسال می‌کند.

امیدوارم ایده اصلی مسئله را متوجه شده باشید هرچند می‌دانم کمی نیاز به تمرین با کدها دارید تا ایده اصلی را به خوبی متوجه شوید.

اجازه دهید روش گذاشته شده با دکوریتور پراکسی را در مثالی عملی تر بسط دهیم. در اینجا دکوریتور trace را خواهیم ساخت که آرگومان های ورودی تابع و نتایج حاصل از آن را در طی زمان اجرا ثبت می‌کند.

```
def trace(func):
    def wrapper(*args, **kwargs):
        print(f'TRACE: calling {func.__name__}() '
              f'with {args}, {kwargs}')

        original_result = func(*args, **kwargs)

        print(f'TRACE: {func.__name__}() '
              f'returned {original_result!r}')

        return original_result

    return wrapper
```

تزئین تابع با تابع trace و بعد فراخوانی آن، آرگومان های منتقل شده به تابع و مقدار برگشتی آن را چاپ خواهد کرد.

این کار گاهی اوقات به عنوان یک کمک عالی برای خطایابی محسوب می‌شود.

1 Argument unpacking

```
@trace
def say(name, line):
    return f'{name}: {line}'

>>> say('Jane', 'Hello, World')
'TRACE: calling say() with ("Jane", "Hello, World"),
{}'
'TRACE: say() returned "Jane: Hello, World"'
'Jane: Hello, World'
```

در رابطه با خطایابی صحبت کردیم، مواردی وجود دارند که هنگام خطایابی دکوریتورها باید به خاطر داشته باشید.

۳-۳-۷. نوشتن دکوریتورهای قابل دیباگ

هنگام استفاده از دکوریتور، در واقع تنها کاری که انجام می‌دهید جایگزین کردن یک تابع با تابع دیگر است .یک نکته منفی این فرآیند این است که برخی از metadataهای تابع اصلی تزئین شده مخفی و جایگزین می‌شوند! این یکی از پنهان‌ترین عملکردهای دکوریتورهای پایتون است که ممکن است درنظر گرفته نشود.

برای مثال، نام تابع اصلی، داک‌استرینگ آن و لیست پارامترهای تابع پس از دکوریت تغییر خواهند کرد! به قطعه کد زیر توجه کنید.

```
def greet():
    """Return a friendly greeting."""
    return 'Hello!'

decorated_greet = uppercase(greet)
```

در صورت دسترسی به هریک از metadata های تابع greet پس از decorate شدن، metadata مربوط به دکوریتور را مشاهده خواهید کرد که تغییر کرده‌اند.

```
>>> greet.__name__
'greet'
>>> greet.__doc__
'Return a friendly greeting.'

>>> decorated_greet.__name__
'wrapper'
>>> decorated_greet.__doc__
None
```

این کار خطایابی و کار با مفسر پایتون را چالش برانگیز می‌کند .خوشبختانه، برای این مشکل راهکاری سریع به نام دستور functools.wraps وجود دارد که در کتابخانه استاندارد پایتون گنجانده شده است.

می‌توانید از دستور functools.wraps در دکوریتورهای خود استفاده کنید تا metadataهای مفقود شده را از تابع تزئین نشده به تابع تزئین شده کپی کنید. مثالی را باهم بررسی می‌کنیم.

```
import functools
def uppercase(func):
    @functools.wraps(func)
    def wrapper():
        return func().upper()
    return wrapper
```

استفاده از دستور functools.wraps در دکوریتور باعث می‌شود metadata های مختلف هنگام دکوریت شدن تابع تغییری نکند و مقادیری مانند نام تابع و داک‌استرینگ تابع به صورت دست نخورده باقی بماند. حقه ی جالبیست نه؟

```
@uppercase
def greet():
    """Return a friendly greeting."""
    return 'Hello!'

>>> greet.__name__
'greet'
>>> greet.__doc__
'Return a friendly greeting.'
```

به عنوان بهترین روش، توصیه می‌کنم که در تمام دکوریتورهایی که می‌نویسید، از دستور functools.wraps استفاده کنید. این کار زیاد طول نمی‌کشد ولی باعث می‌شود شما و بقیه از دردسرهای خطایابی در دکورتیورها در امان بمانید.

تبریک می‌گویم، شما به انتهای این مطلب در رابطه با دکوریتورهای پایتون رسیدید و احتمالا درباره دکوریتورهای موجود در پایتون چیزهای زیادی آموخته اید. کارتان عالی بود!

۳-۳-۸. نکات کلیدی دکوریتورهای پایتون

- دکوریتورها، بلوک های کد قابل استفاده مجدد هستند که می‌توانید برای اصلاح رفتار یک تابع از آن استفاده کنید بدون اینکه نیاز به تغییر دائمی تابع باشد.
- دستور decorator@ تنها شکل کوتاه نویسی شده برای فراخوانی دکوریتور روی تابع ورودی است .دکوریتورهای متعدد روی تابع واحد از پایین به بالا اعمال می‌شوند decorator stacking را به خاطر داشته باشید.
- به عنوان بهترین شیوه خطایابی، از دستور کمک کننده functools.wraps روی دکوریتورهای خود استفاده کنید تا metadata های بیشتری را از تابع قابل فراخوانی تزئین نشده (undecorated) به تابع قابل فراخوانی تزئین شده (decorated)منتقل کنید.
- دکوریتورها درست مانند هر ابزار دیگری در جعبه ابزار توسعه نرم افزار، نوشدارو یا علاج قطعی نیستند و نباید از آنها بیش از حد استفاده کرد. متعادل سازی نیازها بر حسب اولویت انجام کارها با هدف گرفتار نشدن در آشفتگی وحشتناک و غیرقابل نگهداری کد منبع بسیار اهمیت دارد.

۳-٤. کاربردهای جالب *args و **kwargs

زمانی همراه با پایتونیستی باهوش که اصطلاح او "argh" و "kwargh" بود، برنامه نویسی دو نفره انجام می‌دادم. او هر بار تعریف تابع را با پارامترهای اختیاری یا کلیدواژه‌ای تایپ می‌کرد. ما در کنار یکدیگر به بینش دیگری از پایتون دست یافتیم.

پارامترهای *args و **kwargs ویژگی بسیار مفیدی در پایتون هستند و درک توانمندی آن‌ها، شما را به توسعه‌دهنده موثرتری مبدل خواهد کرد.

۳-٤-۱. پارامترهای *args و **kwargs برای چه مواردی استفاده می‌شوند؟

این پارامترها به تابع اجازه می‌دهند تا آرگومان‌های اختیاری بپذیرد، بنابراین می توانید رابط های انعطاف‌پذیری را در ماژول‌ها و کلاس‌های پایتونی خود ایجاد کنید.

```
def foo(required, *args, **kwargs):
    print(required)
    if args:
        print(args)
    if kwargs:
        print(kwargs)
```

تابع فوق حداقل به یک آرگومان به نام "required" احتیاج دارد، اما می‌تواند آرگومان‌های موقعیتی و کلیدواژه‌ای اختیاری دیگری را نیز بپذیرد.

اگر تابع را با آرگومان‌های[1] بیشتری فراخوانی کنیم، args آرگومان‌های اضافی را به صورت tuple جمع‌آوری خواهد کرد. پیشوند * باید در ابتدای نام آرگومان باشد تا این کار را انجام دهد.

همچنین، kwargs آرگومان‌های کلیدواژه‌ای[2] اضافه را به صورت dictionary جمع‌آوری خواهد کرد، چون نام پارامتر دارای پیشوند ** است. اگر هیچ‌یک از

1 Arguments
2 Keyword arguments

آرگومان‌های اضافی به تابع منتقل نشوند، هر دوی args و kwargs می‌توانند خالی باشند.

چون این تابع را با ترکیب متفاوتی از آرگومان‌ها فراخوانی می‌کنیم، خواهید دید که چگونه پایتون آن‌ها را مطابق با اینکه آرگومان‌های اختیاری args یا kwargs هستند، درون پارامترهای args و kwargs به صورت tuple و dictionary جمع‌آوری می‌کند.

```
>>> foo()
TypeError:
"foo() missing 1 required positional arg: 'required'"

>>> foo('hello')
hello

>>> foo('hello', 1, 2, 3)
hello
(1, 2, 3)

>>> foo('hello', 1, 2, 3, key1='value', key2=999)
hello
(1, 2, 3)
{'key1': 'value', 'key2': 999}
```

می‌خواهم این نکته را تصریح کنم که فراخوانی پارامترهای args و kwargs صرفا یک قرارداد نام‌گذاری است. در مثال قبلی اگر آن‌ها را به صورت parms* و argv** فراخوانی کنید نیز کار خواهد کرد. سینتکس واقعی به ترتیب بصورت تک‌ستاره (*) یا دوستاره (**) است. مهم نیست چه نامی برای آرگومان‌ها انتخاب می‌کنید.

با این حال، توصیه می‌کنم برای پرهیز از اشتباهات، قرارداد نام‌گذاری پذیرفته شده را رعایت کنید و هر از گاهی برای فراخوانی از نام‌های"argh" و "kwargh" استفاده کنید. شاید جالب باشد!

۳-٤-۲. ارسال آرگومانهای اختیاری یا کلیدواژه‌ای

در پایتون انتقال پارامترهای اختیاری یا کلیدواژه‌ای از یک تابع به تابع دیگر امکان پذیر است. می‌توانید با استفاده از عملگرهای باز کردن آرگومان * و ** در هنگام فراخوانی تابعی که می‌خواهید آرگومان‌ها را به آن ارسال کنید، این کار را انجام دهید.

همچنین این کار به شما امکان اصلاح آرگومان‌ها قبل از انتقال آن‌ها را فراهم می کند. در اینجا مثالی در این رابطه وجود دارد.

```
def foo(x, *args, **kwargs):
    kwargs['name'] = 'Alice'
    new_args = args + ('extra', )
    bar(x, *new_args, **kwargs)
```

این روش جالب می‌تواند در ارث بری، ارتباط جالبی بین کلاس فرزند و کلاس پدر برقرار کند. شما می‌توانید از این ترفند برای اضافه کردن یک رفتار به کلاس پدر استفاده کنید بدون اینکه امضای متد سازنده کلاس پدر را تغییر دهید. این روشی کاربردی برای زمانیست که ممکن است از یک API استفاده کنید که بدون کنترل شما، تغییر خواهد کرد.

```
class Car:
    def __init__(self, color, mileage):
        self.color = color
        self.mileage = mileage

class AlwaysBlueCar(Car):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.color = 'blue'

>>> AlwaysBlueCar('green', 48392).color
'blue'
```

متد سازنده کلاس AlwaysBlueCar به سادگی تمام آرگومان‌ها را بـه کـلاس والـد خود منتقل می‌کند. این بدان معناست که اگر متد سازنده کلاس والد تغییر کند، کلاس AlwaysBlueCar طبق انتظار کار خواهد کرد.

عیب این روش اینجاست که اکنون متد سازنده AlwaysBlueCar امضای تقریبا بـی فایـده‌ای دارد. نمی‌تـوانیم بـدون مشـاهده کـلاس والـد، بفهمـیم آرگومان‌هـای ورودی کلاس AlwaysBlueCar چه چیزی هستند.

معمولا از این روش در سلسله مراتب کلاس‌های کد خود استفاده نخواهید کرد. این کار گیج کننده خواهد بود. سناریوی محتمل این است که می‌خواهید رفتار را در برخی از کلاس‌های خارجی که خارج از کنترل شما هستند، اصلاح کرده یا تحت‌الشعاع قـرار دهید.

اما این کار همواره سناریو خطرناکی است، پس بهتر است مراقب باشید.

یک سناریو دیگر که در آن احتمالا این روش مفید است، نوشتن توابع wrapper از قبیل دکوریتورها است. در آنجا معمولا می‌خواهید آرگومـان‌هـای دلخـواه کـه بـه تـابع wrapper منتقل شده را نیز بپذیرید.

اگر بتوانیم بدون نیاز به تغییر امضای تابع اصلی این کار را انجام دهـیم، کـد شـما در آینده قابلیت نگهداری بهتری خواهد داشت.

```
def trace(f):
    @functools.wraps(f)
    def decorated_function(*args, **kwargs):
        print(f, args, kwargs)
        result = f(*args, **kwargs)
        print(result)
    return decorated_function

@trace
def greet(greeting, name):
    return '{}, {}!'.format(greeting, name)

>>> greet('Hello', 'Bob')
<function greet at 0x1031c9158> ('Hello', 'Bob') {}
'Hello, Bob!'
```

همانطور که مشاهده کردید، با این روش توانستیم پارامترهای اختیاری را به تابع دکوریتور منتقل کنیم. گاهی از این روش به منظور رعایت اصل DRY[1] استفاده خواهد شد.

۳-٤-۳. نکات کلیدی *args و **kwargs

- دستورهای *args و **kwargs به شما امکان نوشتن توابع با تعداد آرگومان‌های متغیر در پایتون را می‌دهد.
- دستور *args آرگومان‌های موقعیتی اضافی را به صورت tuple جمع‌آوری می‌کند و دستور **kwargs آرگومان‌های کلیدواژه‌ای اضافی را به صورت dictionary جمع‌آوری می‌کند.
- سینتکس واقعی * و ** است. فراخوانی آنها با دستورهای args و kwargs فقط یک قرارداد است (و بهتر است آن را رعایت کنید).

1 Don’t Repeat Yourself

۳-۵. بازکردن آرگومان‌های توابع

ویژگی بسیار جالب اما پنهان، قابلیت باز کردن بسته‌بندی آرگومان‌های تابع با استفاده از عملگرهای * و ** است. اجازه دهید تابعی ساده را برای مثال تعریف کنیم.

```
def print_vector(x, y, z):
    print('<%s, %s, %s>' % (x, y, z))
```

همانطور که مشاهده می‌کنید، این تابع سه آرگومان (x، y و z) را می‌گیرد و آنها را به روشی زیبا چاپ می‌کند. ممکن است از این تابع برای چاپ بردارهای ۳ بعدی در برنامه خود استفاده کنیم.

```
>>> print_vector(0, 1, 0)
<0, 1, 0>
```

حال بسته به نوع ساختار داده‌ای که برای نمایش بردارهای سه بعدی انتخاب می کنیم، چاپ آن‌ها با تابع print_vector ممکن است کمی ناخوشایند باشد. برای مثال، اگر بردارهای خود را به صورت تاپل‌ها یا لیست‌ها مقدار دهی کنیم، هنگام چاپ آن‌ها، باید به طور صریح، اندیس هر مولفه را مشخص کنیم.

```
>>> tuple_vec = (1, 0, 1)
>>> list_vec = [1, 0, 1]
>>> print_vector(tuple_vec[0],
                 tuple_vec[1],
                 tuple_vec[2])

<1, 0, 1>
```

فراخوانی معمولی تابع با آرگومان‌های جداگانه، طولانی و غیرضروری به نظر می رسد. اگر بتوانیم شیء وکتور را به سه مولفه آن "تجزیه کنیم" و همه چیز را به یکباره به تابع print_vector منتقل کنیم، ظریف‌تر نخواهد بود؟

البته، به سادگی می‌توانید تابع print_vector را مجددا تعریف کنید تا پارامتر واحدی با نام شیء وکتور را به عنوان ورودی دریافت کند (اما بخاطر رعایت أصول یک مثال ساده، فعلا این گزینه را نادیده می‌گیریم.)

خوشبختانه، روش بهتری برای پرداختن به این وضعیت در پایتون تحت عنوان باز کردن بسته‌بندی آرگومان تابع[1] با استفاده از عملگر * وجود دارد.

```
>>> print_vector(*tuple_vec)
<1, 0, 1>
>>> print_vector(*list_vec)
<1, 0, 1>
```

قرار دادن علامت * قبل از یک شی قابل تکرار[2] در فراخوانی تابع، آن را باز خواهد کرد و عناصر آن را بصورت آرگومان‌های موقعیتی جداگانه به تابع فراخوانی شده منتقل خواهد کرد.

این تکنیک برای هر گونه تابع تکرارپذیر، از جمله عبارات مولد[3] کارساز است. با استفاده از عملگر * روی مولد، تمام عناصر مولد را اجرا می‌کند و آنها را به تابع منتقل می‌کند.

```
>>> genexpr = (x * x for x in range(3))
>>> print_vector(*genexpr)
```

علاوه بر عملگر * برای باز کردن توالی‌هایی مانند تاپل‌ها، لیست‌ها و مولدها در داخل آرگومان‌های موقعیتی، عملگر ** برای باز کردن آرگومان‌های کلیدواژه‌ای از دیکشنری وجود دارد. تصور کنید وکتور ما بصورت شیء دیکشنری زیر تعریف شده است.

```
>>> dict_vec = {'y': 0, 'z': 1, 'x': 1}
```

1 Function Argument Unpacking
2 Iterable
3 Generator expresions

می‌توانیم با استفاده از عملگر ** برای باز کردن، این دیکشنری را به تابع print_vector منتقل کنیم.

```
>>> print_vector(**dict_vec)
< 1, 0, 1 >
```

چون دیکشنری‌ها بدون ترتیب هستند، پس مقادیر دیکشنری و آرگومان‌های تابع بر پایه کلیدهای دیکشنری تطبیق داده می‌شوند. آرگومان x مقدار مرتبط با کلید 'x' در دیکشنری را دریافت می‌کند.

اگر بخواهید از عملگر تک ستاره (*) برای باز کردن بسته‌بندی دیکشنری استفاده کنید، کلیدها به صورت ترتیب تصادفی، به این تابع منتقل خواهند شد.

```
>>> print_vector(*dict_vec)
< y, x, z >
```

ویژگی باز کردن بسته‌بندی آرگومان تابع در پایتون، انعطاف پذیری زیادی را در اختیار شما قرار می‌دهد. اغلب این بدان معنی است که شما نیازی به پیاده‌سازی کلاس برای نوع داده‌های مورد نیاز با برنامه خود ندارید. در نتیجه، استفاده از ساختار داده‌های درونی ساده مانند تاپل‌ها یا لیست‌ها کافی خواهد بود و به کاهش پیچیدگی کد شما کمک خواهد کرد.

۳-۵-۱. نکات اصلی در هنگام بازکردن آرگومان‌های تابع

- از عملگرهای * و ** می‌توان برای "باز کردن" آرگومان‌های تابع از توالی‌ها و دیکشنری‌ها استفاده کرد.
- استفاده موثر از باز کردن بسته‌بندی آرگومان‌ها می‌تواند به شما در نوشتن رابط های بسیار انعطاف‌پذیر برای ماژول‌ها و توابع کمک کند.

۳-۶. هیچ چیز برای Return

پایتون برگشت ضمنی None را به پایان هر تابع اضافه می‌کند. بنابراین، اگر تابع مقدار برگشتی را مشخص نکند، تابع به صورت پیش‌فرض None را برمی‌گرداند.

این بدان معناست که می‌توانید بیانیه‌های برگشتی None را با بیانیه‌های برگشتی خالی جایگزین کنید و یا آنها را کاملاً خالی نگه دارید و همچنان نتیجه مشابه را بدست آورید.

```
def foo1(value):
    if value:
        return value
    else:
        return None

def foo2(value):
    """Bare return statement implies `return None`"""
    if value:
        return value
    else:
        return

def foo3(value):
    """Missing return statement implies `return
None`"""
    if value:
        return value
```

اگر مقداری اشتباه را به عنوان آرگومان‌های تابع ارسال کنید، هر سه تابع به درستی None را بر می‌گردانند.

```
>>> type(foo1(0))
<class 'NoneType'>

>>> type(foo2(0))
<class 'NoneType'>

>>> type(foo3(0))
<class 'NoneType'>
```

حال، چه زمانی استفاده از این ویژگی در کد پایتون شما ایده خوبی است؟

قاعده سرانگشتی من این است که اگر تابع مقدار برگشتی نداشته باشد (سایر زبانها این روش را procedure می‌نامند)، آنگاه با خروجی تابع کاری ندارم. افزودن return خالی فقط کار اضافی و باعث سردرگمی خواهد بود. مثالی برای این روش، تابع print داخلی پایتون است که فقط برای کارهای جانبی (چاپ متن) فراخوانی می‌شود و هرگز با مقدار برگشتی آن کاری نداریم.

اجازه دهید تابعی مانند sum که جزو توابع داخلی پایتون است را انتخاب کنیم. این کار به وضوح دارای مقدار برگشت منطقی است و معمولا تابع sum فقط برای کارهای جانبی آن فراخوانی نخواهد شد. هدف آن جمع کردن اعداد با یکدیگر و بعد ارائه نتیجه است. حال اگر تابع از نقطه نظر منطقی دارای مقدار برگشتی باشد، باید تصمیم بگیرید که آیا از return استفاده می‌کنید یا خیر.

از یک طرف، می‌توانید استدلال کنید که حذف return None کد را دقیق‌تر می کند و در نتیجه خواندن و درک آن راحت تر خواهد بود. به لحاظ ذهنی، ممکن است بگویید که کد را "زیباتر" نیز می‌کند.

از طرف دیگر، ممکن است برخی برنامه‌نویسان از چنین رفتاری در پایتون متعجب شوند. اگر زمان نوشتن کد تمیز و قابل نگهداری برسد، رفتار تعجب‌آور به ندرت نشانه خوبی است.

در نهایت، به طور مستمر ایمیل‌هایی را دریافت کردم که به من" بیانیه برگشت ضمنی" در مثال کد را خاطر نشان می‌کردند. رفتار برگشت ضمنی پایتون به وضوح برای همه مشخص نبود. یادداشتی به آن اضافه کردم تا مشخص شود که چه اتفاقی افتاده و بعد ایمیل‌ها متوقف شدند.

اشتباه برداشت نکنید، بیشتر از هر کسی نوشتن کد تمیز و "زیبا" را دوست دارم و نیز قویا احساس می‌کنم که برنامه‌نویس‌ها باید از جزئیات کامل زبانی که با آن کار می کنند، اطلاعات لازم را داشته باشند.

اما وقتی تاثیر نگهداری چنین سوء تفاهم‌های ساده‌ای را در نظر بگیرید، منطقی است که به سمت نوشتن کد صریح و خوانا متمایل شوید. غیر از این‌ها، کد وسیله برقراری ارتباط است.

۳-۶-۱. نکات کلیدی return None در پایتون

- اگر تابع مقدار برگشتی را مشخص نکند، مقدار None را برمی‌گرداند. اینکه آیا صریحا بنویسید return None بستگی به تصمیم خودتان و سبک کدنویسی شما دارد.
- این ویژگی هسته پایتون است. کد شما ممکن است با نوشتن return None در خروجی توابعی که خروجی ندارند، از ابهام جلوگیری کند. یا ممکن است باعث تعجب برنامه نویسان شود. تصمیم به عهده خودتان و سبک کد نویسی شماست.

فصل چهارم

کلاس‌ها و برنامه‌نویسی شیء گرا

٤-١. مقایسه شیء ها: "is" در مقابل "=="

وقتی بچه بودم، همسایه‌های ما دو گربه دوقلو داشتند. آن‌ها ظاهرا مشابه بودند، خز زغالی و چشم‌های سبز تیزبین مشابهی داشتند. اگر برخی از ویژگی‌های شخصیتی گربه‌ها کنار گذاشته شوند، نمی‌توانستید با نگاه کردن به آن‌ها تفاوتشان را بگویید. البته، آن‌ها دو گربه متفاوت بودند، دو موجود جداگانه، با وجود اینکه دقیقا یکسان به نظر می‌رسیدند.

این کار در من تفاوت در معنایی بین برابر و مشابه را القاء کرد و این تفاوت در درک نحوه رفتار عملگرهای مقایسه is و == در پایتون حائز اهمیت است.

عملگر == با بررسی برابری مقایسه می‌کند: اگر این گربه‌ها اشیاء پایتون بودند و آن ها را با عملگر == مقایسه می‌کردیم، در پاسخ "هر دو گربه برابر هستند"را بدست می-آوردیم.

با این حال، عملگر is، هویت ها را با هم مقایسه می‌کند: اگر گربه‌های خود را با عملگر is مقایسه می‌کردیم، در پاسخ "این دو گربه متفاوت هستند" را بدست می آوردیم.

قبل از اینکه گرفتار این قیاس کلافه کننده گربه شوم، اجازه دهید یک تکه کد پایتون واقعی را بررسی کنیم.

ابتدا شیء جدید لیست را ایجاد می‌کنیم و آن نامگذاری می‌کنیم، و بعد متغیر دیگر (b) که به همان شیء لیست اشاره دارد را تعریف می‌کنیم.

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

اجازه دهید این دو متغیر را بررسی کنیم. می‌توانیم مشاهده کنیم که آن‌ها به لیست های در ظاهر یکسان اشاره می‌کنند.

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

چون دو شیء لیست یکسان به نظر می‌رسند، پس هنگامی که آن‌ها را برای برابری با استفاده از عملگر == مقایسه می‌کنیم، نتیجه صحیح را بدست خواهیم آورد.

```
>>> a == b
True
```

با این حال، این به ما نمی‌گوید که a و b در واقع به شیء یکسان اشاره می‌کنند. البته، می‌دانیم که این اشیا چگونه مقداردهی شده‌اند، اما فرض کنید که این را نمی‌دانیم. چگونه می‌توانیم بفهمیم که این دو اشیا هویت یکسان دارند؟

پاسخ، مقایسه هر دو متغیر با عملگر is است. این عملگر تایید می‌کند که هر دو شی در واقع به یک لیست اشاره می‌کنند.

```
>>> a is b
True
```

اجازه دهید ببینیم هنگام ایجاد کپی مشابهی از شیء لیست، چه اتفاقی می‌افتد. می توانیم این کار با فراخوانی متد list() در لیست موجود، برای ایجاد یک کپی که آن را c می‌نامیم، انجام دهیم.

```
>>> c = list(a)
```

از طرف دیگر، خواهید دید که لیست جدیدی که ایجاد کردیم کاملاً مشابه با شیء لیست اشاره شده با a و b است.

```
>>> c
[1, 2, 3]
```

حالا اینجاست که موضوع جالب می‌شود. اجازه دهید کپی لیست c را با لیست اولیه a با استفاده از عملگر == مقایسه کنیم. انتظار دارید چه پاسخی را مشاهده کنید؟

```
>>> a == c
True
```

خب، امیدوارم این همان چیزی باشد که انتظار آن را داشتید. آنچه این نتیجه به ما می‌گوید این است که c و a محتویات مشابهی دارند. آن‌ها در پایتون برابر در نظر گرفته می‌شوند. اما آیا در واقع آن‌ها به شیء مشابهی اشاره می‌کنند؟ اجازه دهید با عملگر is جواب را پیدا کنیم.

```
>>> a is c
False
```

بوم! اینجاست که نتیجه متفاوتی را بدست می‌آوریم. پایتون می‌گوید که متغیرهای c و a به دو شیء متفاوت اشاره می‌کنند، حتی اگر محتویات آن‌ها یکسان باشند.

پس، برای یادآوری، اجازه دهید تفاوت بین is و == را در دو تعریف کوتاه خلاصه کنیم.

- عبارت is زمانی برابر با True خواهد شد که دو شیء هویت یکسانی داشته باشند. (به یک مکان از حافظه اشاره کنند.)
- عبارت == زمانی برابر با True خواهد شد که دو شیء محتویات یکسانی داشته باشند.

فقط به یاد داشته باشید که هر زمان بخواهید بین استفاده از is و == در پایتون تصمیم بگیرید، گربه‌های دوقلو را تصور کنید (در مورد سگ‌ها هم عمل می‌کند). اگر این کار را انجام دهید، مشکلی نخواهید داشت.

٤-۲. تبدیل رشته (هر کلاس به __repr__ نیاز دارد)

وقتی کلاس سفارشی را در پایتون تعریف کرده و بعد یکی از نمونه‌های آن را روی کنسول چاپ می‌کنید (یا آن را در مفسر بررسی می‌کنید)، نتیجه‌ای نسبتا ناخوشایند بدست می‌آورید. رفتار تبدیل پیش‌فرض to_string در کلاس‌ها ساده است اما توضیحات زیادی در رابطه با آن وجود ندارد.

```
class Car:
    def __init__(self, color, mileage):
        self.color = color
        self.mileage = mileage

>>> my_car = Car('red', 37281)
>>> print(my_car)
<__console__.Car object at 0x109b73da0>
>>> my_car
<__console__.Car object at 0x109b73da0>
```

بطور پیش‌فرض، تمام آن چیزی که بدست می‌آورید، رشته حاوی نام کلاس و شناسه نمونه شیء (که آدرس حافظه شیء در CPython است) می‌باشد. این بهتر از هیچ است، اما چندان مفید نیست.

ممکن است کمی روی این شیء کار کرده باشید و با استفاده از پرینت یک خروجی برای شیء خود درنظر بگیرید یا حتی یک متد اختصاصی to_string() نوشته باشید که این کار را می‌کند.

```
>>> print(my_car.color, my_car.mileage)
red 37281
```

به جای نوشتن یک متد to_string() سفارشی، بهتر است متدهای __str__ و __repr__ را به کلاس خود اضافه کنید. آن‌ها روش پایتونی برای کنترل نحوه تبدیل اشیاء به رشته‌ها در موقعیت‌های مختلف هستند.

اجازه دهید نحوه کار این روش‌ها را در عمل بررسی کنیم. برای شروع، می‌خواهیم متد __str__ به کلاس Car که قبلا تعریف کرده‌ایم را اضافه کنیم.

```
class Car:
    def __init__(self, color, mileage):
        self.color = color
        self.mileage = mileage

    def __str__(self):
        return f'a {self.color} car'
```

وقتی نمونه کلاس Car را چاپ می‌کنید، نتیجه‌ای متفاوت و کمی بهبودیافته بدست خواهید آورد.

```
>>> my_car = Car('red', 37281)
>>> print(my_car)
'a red car'
>>> my_car
<__console__.Car object at 0x109ca24e0>
```

بررسی شیء Car در کنسول هنوز نتیجه قبلی را که حاوی شناسه شیء است را به ما می‌دهد. اما چاپ شیء منجر به بازگشت رشته با متد __str__ که اضافه کردیم، می شود.

متد __str__ یکی از متدهای داندر پایتون است و وقتی می‌خواهید یک شیء را به رشته تبدیل کنید، به صورت خودکار فراخوانی می‌شود.

```
>>> print(my_car)
a red car

>>> str(my_car)
'a red car'

>>> '{}'.format(my_car)
'a red car'
```

با پیاده‌سازی مناسب __str__، دیگر نیازی نیست نگران چاپ مستقیم ویژگی‌های شیء یا نوشتن تابع جداگانه ()to_string باشید. این روش پایتونی برای کنترل تبدیل شیء به رشته است.

به هر حال، برخی افراد متدهای "داندر" پایتون را "متدهای جادویی" می‌نامند. اما این متدها به هیچ وجه جادویی نیستند. واقعیت این است که این متدها به صورت زیرخط‌های دوتایی شروع شده و به پایان می‌رسند، که صرفا قوانین نامگذاری است تا متوجه شویم این متدها، متدهای هسته پایتون هستند. همچنین این کار جلوی تداخل نام این متدها، با متدهای اختصاصی برنامه شما را می‌گیرد. شیء سازنده __init__ نیز از قوانین مشابهی پیروی می‌کند و هیچ چیز جادویی یا سری در مورد آن وجود ندارد.

از استفاده از متدهای داندر پایتون نترسید، این‌ها به شما کمک می‌کنند.

٤-۲-۱. متد __str__ در مقایسه با __repr__

داستان تبدیل رشته ما به اینجا ختم نمی‌شود. آیا نحوه بررسی my_car در مفسر که هنوز نتیجه عجیب <Car object at 0x109ca24e0> را می‌دهد را می‌بینید؟

سبب وقوع این امر این بود که در واقع دو روش داندر وجود دارد که نحوه تبدیل اشیاء به رشته‌ها در پایتون ۳ را کنترل می‌کند. روش اول __str__است و اطلاعات کمی در مورد آن دارید. روش دوم __repr__ است و نحوه کار با آن شبیه به __str__ است، اما در موقعیت های مختلف استفاده می شود. (پایتون 2.x نیز دارای روش __unicode__ است که بعدا کمی به این موضوع می‌پردازم.)

در اینجا آزمایش ساده‌ای وجود دارد که می‌توانید در هنگام استفاده از __str__ یا __repr__ برای درک این ایده بکار گیرید. اجازه دهید دوباره کلاس Car خود را تعریف کنیم، پس این کلاس حاوی هر دو متد داندر برای تبدیل رشته خواهد بود.

```
class Car:
    def __init__(self, color, mileage):
        self.color = color
        self.mileage = mileage

    def __repr__(self):
        return '__repr__ for Car'

    def __str__(self):
        return '__str__ for Car'
```

حال، وقتی از طریق نمونه‌های قبلی کد را اجرا می‌کنید، می‌فهمید که کدام روش نتیجه تبدیل رشته را در هر مورد کنترل می‌کند.

```
>>> my_car = Car('red', 37281)
>>> print(my_car)
__str__ for Car
>>> '{}'.format(my_car)
'__str__ for Car'
>>> my_car
__repr__ for Car
```

این آزمایش تایید می‌کند که بررسی یک شیء در مفسر پایتون صرفا نتیجه __repr__ شیء را چاپ می‌کند.

جالب اینکه مجموعه‌هایی مانند لیست‌ها و دیکشنری‌ها همواره از نتیجه __repr__ برای نمایش اشیاء حاوی آن استفاده می‌کنند، حتی اگر str را روی یک مجموعه‌ای همانند لیست فراخوانی کنید.

```
str([my_car])
'[__repr__ for Car]'
```

برای انتخاب دستی بین دو روش تبدیل رشته، برای بیان دقیق‌تر هدف کد خود، بهتر است از توابع داخلی پایتون str() و repr() استفاده کنید. استفاده از آن‌ها بر فراخوانی مستقیم __str__ یا __repr__ ترجیح داده می‌شود، چون بهتر به نظر می‌رسد و نتیجه مشابهی را می‌دهد.

```
>>> str(my_car)
'__str__ for Car'
>>> repr(my_car)
'__repr__ for Car'
```

حتی با این تحقیق کامل، ممکن است متعجب شوید که تفاوت واقعی بین __str__ و __repr__ در چیست. اینگونه به نظر می‌رسد که هر دو هدف مشابهی را دنبال می کنند، بنابراین ممکن است مشخص نباشد که چه زمانی از کدام باید استفاده کنید.

با پرسیدن سؤالاتی مانند این، معمولا بهترین ایده، بررسی این مورد است که کتابخانه استاندارد پایتون چه کاری انجام می‌دهد. وقت این است که آزمایش دیگری را انجام دهیم. یک شیء datetime.date را ایجاد کرده و خواهیم فهمید که چگونه از __repr__ و __str__ برای کنترل تبدیل رشته استفاده می کند.

```
>>> import datetime
>>> today = datetime.date.today()
```

نتیجه تابع __str__ شیء تاریخ در پایتون باید خوانا باشد. این به معنای برگشت نمایش متنی دقیق برای برنامه‌نویسان است (چیزی که شما با آن احساس راحتی می کنید را نشان می‌دهد.) بنابراین، هنگام فراخوانی با متد str() در مورد شیء تاریخ، چیزی شبیه به فرمت تاریخ استاندارد را بدست می‌آوریم.

```
>>> str(today)
'2017-02-02'
```

با متد داندر __repr__، معتقدیم که مهمتر از همه، نتیجه باید بدون ابهام باشد. رشته حاصل بیشتر به عنوان کمک به اشکال‌زدایی برای توسعه‌دهندگان در نظر گرفته شده است و به همین خاطر باید در مورد محتویات این شیء، تا حد امکان صریح باشد. به همین دلیل نتیجه دقیق‌تری بدست خواهید آورد اگر متد repr() را برای شیء فراخوانی کنید. این نتیجه حتی شامل نام کامل ماژول و نام کلاس نیز می‌باشد.

```
>>> repr(today)
'datetime.date(2017, 2, 2)'
```

می‌توانیم رشته برگشت داده شده را با __repr__ کپی و پیست کنیم و آن را به عنوان کد معتبر پایتون اجرا کنیم تا شیء اصلی تاریخ را دوباره ایجاد کنیم. این رویکردی عالی برای نوشتن repr های اختصاصی است.

از طرف دیگر، درمی‌یابیم که نوشتن __repr__ های اختصاصی کمی دشوار است. معمولا فکر می‌کنید ارزش این همه زحمت را نخواهد داشت و فقط کار اضافی به شما تحمیل می‌کند. قاعده سرانگشتی من این است که رشته‌های __repr__ را بدون ابهام و مفید برای توسعه‌دهندگان تولید کنم، اما انتظار ندارم آن‌ها بتوانند شیء کاملی را براساس خروجی من تولید کنند.

٤-٢-٢. چرا هر کلاس به __repr__ نیاز دارد

اگر متد __str__ را اضافه نکنید، پایتون هنگام جستجوی __str__ به نتیجه __repr__ متوسل می‌شود. بنابراین، توصیه من این است که همیشه حداقل یک متد __repr__ را به کلاس‌های خود اضافه کنید. این کار تقریبا در همه موارد، با حداقل کار اجرایی، نتیجه تبدیل رشته‌ای مفید را تضمین خواهد کرد.

در اینجا نحوه افزودن سریع و کارآمد تبدیل رشته به کلاس‌های شما ارائه شده است. برای کلاس Car ممکن است با متد __repr__ زیر شروع کنیم.

```
def __repr__(self):
    return f'Car({self.color!r}, {self.mileage!r})'
```

لطفا توجه داشته باشید که من از پرچم[1] تبدیل !r استفاده می‌کنم تا مطمئن شوم که رشته خروجی بجای str(self.color) و str(self.mileage) از rep(self.color) و rep(self.mileage) استفاده می‌کند.

1 Flag

این کار خوب جواب می‌دهد، اما تنها عیب آن این است که نام کلاس داخل رشته را تکرار می‌کنیم. ترفندی که می‌توانید در اینجا برای جلوگیری از این تکرار استفاده کنید، استفاده از ویژگی __name__ از __class__ شیء است که همیشه نام کلاس را بصورت رشته بیان خواهد کرد.

مزیت این روش این است که هنگام تغییر نام کلاس، پیاده‌سازی __repr__ را اصلاح نخواهید کرد. این کار سبب پایبندی راحت به اصل خودت را تکرار نکن[1] می شود.

```
def __repr__(self):
    return (f'{self.__class__.__name__}('
        f'{self.color!r}, {self.mileage!r})')
```

نکته منفی این اجراء این است که رشته فرمت کاملاً طولانی و سنگین است. اما با فرمت‌دهی دقیق، می‌توانید کد را مطابق با پیشنهاد PEP 8 بهبود دهید.

با اجرای متد __repr__ فوق، هنگام بررسی شیء یا فراخوانی مستقیم repr() روی آن نتیجه مفیدی را بدست می‌آوریم.

```
>>> repr(my_car)
'Car(red, 37281)'
```

چاپ شیء یا فراخوانی str() روی آن همان رشته را برمی‌گرداند چون هنگام اجرای متد __str__ به صورت خودکار متد __repr__ را فراخوانی می‌کند.

```
>>> print(my_car)
'Car(red, 37281)'
>>> str(my_car)
'Car(red, 37281)'
```

بر این باورم که این رویکرد با کمترین میزان کار اجرایی، با ارزش‌ترین چیز را فراهم می‌کند. به همین دلیل، همیشه پیاده سازی اولیه __repr__ را به کلاس‌های خود

1 DRY – Don’t Repeat Yourself

اضافه می‌کنم. در اینجا مثال کاملی برای پایتون ۳، از جمله پیاده سازی اختیاری __str__ وجود دارد.

```
class Car:
    def __init__(self, color, mileage):
        self.color = color
        self.mileage = mileage

    def __repr__(self):
        return (f'{self.__class__.__name__}('
                f'{self.color!r}, {self.mileage!r})')

    def __str__(self):
        return f'a {self.color} car'
```

٤-٢-٣. تفاوت‌های پایتون 2.x: __unicode__

پایتون 2.x از مدل متفاوتی برای تبدیل رشته‌ها استفاده می‌کند. دو راه برای نمایش متن str که محدود به مجموعه کاراکترهای ASCII است و unicode که معادل str پایتون ۳ است، وجود دارد.

به دلیل این تفاوت، روش داندر دیگری برای کنترل تبدیل رشته در پایتون ۲: __unicode__ وجود دارد. در پایتون ۲، __str__ بایت‌ها را برمی‌گرداند و __unicode__ کاراکترها را برمی‌گرداند.

برای بیشتر اهداف، __unicode__ روش جدیدتر و ترجیحی برای کنترل تبدیل رشته است. تابع داخلی unicode() نیز وجود دارد که می‌توانید از آن استفاده کنید. این کارشبیه به نحوه کار str() و rep() خواهد بود.

تا اینجا بسیار خوب بود. حال، زمانی که به قواعد فراخوانی در پایتون ۲ برای __str__ و __unicode__ نگاه می‌کنید، کمی غیرعادی به نظر می‌رسد.

عبارت print و str() متد داندر __str__ را در صورت وجود فراخوانی می‌کند. متد داخلی unicode() نیز متد داندر __unicode__ را در صورت وجود فراخوانی می‌کند، در غیر اینصورت متد داندر __str__ فراخوانی خواهد شد.

در مقایسه با پایتون ۳، این موارد خاص، قواعد تبدیل متن را تا حدودی پیچیده می کند. اما راهی برای ساده کردن چیزها برای اهداف کاربردی وجود دارد. یونیکد، شیوه ترجیحی و آینده‌آزمایی برای دستکاری متن در برنامه‌های پایتون شما است.

بنابراین، آنچه که توصیه می‌کنم در پایتون 2.x انجام دهید، این است که تمام کد فرمت‌دهی رشته خود را داخل متد داندر __unicode__ قرار دهید و بعد پیاده سازی متد داندر __str__ که نمایش یونیکد رمزگذاری شده با عنوان UTF-8 برمی‌گرداند را ایجاد کنید.

```
def __str__(self):
    return unicode(self).encode('utf-8')
```

متد داندر __str__ برای بیشتر کلاس‌هایی که می‌نویسید یکسان خواهد بود، بنابراین می‌توانید در صورت لزوم آن‌را کپی و پیست کنید. تمام کدهای تبدیل رشته که برای استفاده غیر توسعه‌دهنده در نظر گرفته شده، در متد داندر __unicode__ قرار دارند.

در اینجا مثال کاملی برای استفاده از این روش در پایتون 2.x ارائه شده است.

```
class Car(object):
    def __init__(self, color, mileage):
        self.color = color
        self.mileage = mileage

    def __repr__(self):
        return '{}({!r}, {!r})'.format(
            self.__class__.__name__,
            self.color, self.mileage)

    def __unicode__(self):
        return u'a {self.color} car'.format(
            self=self)

    def __str__(self):
        return unicode(self).encode('utf-8')
```

٤-۲-٤. نکات کلیدی مقایسه __str__ و __repr__

- با استفاده از متدهای داندر __str__ و __repr__ می‌توانید تبدیل به رشته‌ها را در کلاس‌های خود کنترل کنید.
- نتیجه متد داندر __str__ باید خوانا باشد. نتیجه متد داندر__repr__ باید بدون ابهام باشد.
- همواره متد داندر __repr__ را به کلاس‌های خود اضافه کنید. پیاده‌سازی پیش‌فرض __str__ فقط متد داندر __repr__ را فراخوانی می‌کند.
- بجای استفاده از متد __str__ در پایتون ۲ از متد __unicode__ استفاده کنید.

٤-۳. کلاس‌های Exception سفارشی

هنگامی که از پایتون استفاده می‌کردم، به نوشتن کلاس‌های استثناء سفارشی در کد خود تردید داشتم. اما تعریف انواع خطاها می‌تواند بسیار ارزشمند باشد. می‌توانید موارد خطای احتمالی را به وضوح مشخص کنید و در نتیجه توابع و ماژول‌های شما بهتر قابل مدیریت خواهد بود. همچنین می‌توانید از انواع خطاهای سفارشی برای ارائه اطلاعات بیشتری در مورد اشکال‌زدایی استفاده کنید.

کلیه این موارد کد پایتون شما را بهبود بخشیده و درک، اشکال‌زدایی و نگهداری از آن را راحت‌تر خواهد کرد. تعریف کلاس‌های استثناء در هنگام تجزیه آن به چند مثال ساده چندان سخت نیست. در این فصل، نکات اصلی که باید به خاطر بسپارید، را بیان خواهیم کرد.

فرض کنید بخواهید رشته ورودی یک تابع را که نشانگر نام یک شخص در برنامه شما است، را اعتبار سنجی کنید. نمونه تابعی برای اعتبارسنجی نام ممکن است بدین صورت باشد.

```
def validate(name):
    if len(name) < 10:
        raise ValueError
```

در صورت عدم انجام اعتبارسنجی، استثناء ValueError ایجاد می‌شود. اکنون این روش به نظر مناسب می‌باشد و نوعی سینتکس پایتونیک است. تا اینجا بسیار خوب بود.

با این حال، در استفاده از کلاس استثنایی عمومی سطح بالا مانند ValueError نقطه ضعفی وجود دارد. تصور کنید یکی از هم تیمی‌های شما این تابع را بصورت بخشی از کتابخانه فراخوانی می‌کند و اطلاعات زیادی درباره بخش‌های داخلی آن ندارد. وقتی در اعتبارسنجی یک نام موفق نمی‌شود، در کنسول پیام زیر نمایش داده می‌شود.

```
>>> validate('joe')
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    validate('joe')
  File "<input>", line 3, in validate
    raise ValueError
ValueError
```

در واقع، این اطلاعات چندان سودمند نیست. قطعا، می‌دانیم که یه چیزی اشتباه شده و مسئله با "مقدار ناصحیح" روبه‌رو شده است، اما برای اینکه بتوانید مسئله را برای هم تیمی خود حل کنید، مسلما باید به دنبال بررسی تابع validate() باشید. با این حال، خواندن کدها وقت‌گیر است اما این کار می‌تواند به سرعت انجام شود.

خوشبختانه، می‌توانیم عملکرد بهتری داشته باشیم. اجازه دهید نوعی استثناء سفارشی را پیاده‌سازی کنیم تا اعتبارسنجی ناموفق نام را نشان دهیم. کلاس استثناء جدید خود را از ValueError داخلی پایتون ارث بری می‌کنیم، اما با دادن نامی صریح‌تر به آن، باعث می‌شویم خودش صحبت کند.

```
class NameTooShortError(ValueError):
    pass

def validate(name):
    if len(name) < 10:
        raise NameTooShortError(name)
```

حال، یک نوع استثناء "خود استنادی" تحت عنوان NameTooShortError داریم که کلاس ValueError داخلی را گسترش می‌دهد. در حالت کلی، می‌خواهید استثناءهای سفارشی خود را از کلاس استثناء ریشه یا سایر استثناءهای داخلی پایتون مانند ValueError یا TypeError، هرکدام که مناسب باشد را ارث بری کنید.

همچنین، ببینید که چگونه وقتی آن را به عنوان نمونه‌ای معتبر معرفی می‌کنیم، متغیر نام را به سازنده کلاس استثناء سفارشی خود منتقل می‌کنیم؟ اجرای جدید منجر به نمایش خطای بسیار بهتری برای همکارانتان می‌شود.

```
>>> validate('jane')
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    validate('jane')
  File "<input>", line 3, in validate
    raise NameTooShortError(name)
NameTooShortError: jane
```

یکبار دیگر، خودتان را بجای هم‌تیمی خود قرار دهید. این کار، درک کلاس‌های استثناء سفارشی را بسیار آسانتر می‌کند. مخصوصا هنگامی که با مشکلی روبرو می‌شویم (که در نهایت با آن روبرو خواهیم شد) چه اتفاقی باید رخ دهد.

این موضوع باید در نظر گرفته شود حتی اگر به تنهایی روی کد منبع کار می‌کنید. چند هفته یا چند ماه بعد، در صورت داشتن ساختار مناسب، زمان نگهداری بسیار کمتری برای کدتان صرف خواهید کرد.

با صرف فقط ۳۰ ثانیه برای تعریف کلاس استثناء ساده، این تکه کد حالا به کد بسیار مفیدتری تبدیل می‌شود. اما اجازه دهید تا ادامه دهیم. موارد بیشتری برای مطرح کردن وجود دارند.

زمانی که پکیج پایتونی را به صورت عمومی منتشر می‌کنید، یا حتی ماژول قابل استفاده مجدد برای شرکت خود ایجاد می‌کنید، شیوه خوبی برای ایجاد کلاس پایه استثناء سفارشی برای ماژول (پکیج) است و بعد سایر استثناءهای خود را بر پایه آن می‌سازید.

در اینجا نحوه ایجاد سلسله مراتب استثناء سفارشی برای همه استثناءها در ماژول یا پکیج‌های پایتون ارائه شده است. اولین گام، اعلان کلاس پایه است که همه خطاهای برنامه‌ی ما از آن ارث بری خواهند کرد.

```
class BaseValidationError(ValueError):
    pass
```

حال، کلیه کلاس‌های خطای "واقعی" خود را می‌توانید از کلاس خطای پایه بدست آورید. این کار با کمی تلاش بیشتر، سلسله مراتب استثنای خوب و تمیزی را ارائه می دهد.

```
class NameTooShortError(BaseValidationError):
    pass


class NameTooLongError(BaseValidationError):
    pass


class NameTooCuteError(BaseValidationError):
    pass
```

برای مثال، این به کاربران پکیج شما امکان نوشتن عبارت try...except را می‌دهد تا بتوانند تمام خطاهای ایجاد شده توسط پکیج پایتونی شما را کنترل کنند.

```
try:
    validate(name)
except BaseValidationError as err:
    handle_validation_error(err)
```

کاربرانتان می‌توانند بدین شیوه استثناءهای خاص‌تری را بدست آورند، اما اگر نخواهند، حداقل مجبور نخواهند بود برای گرفتن استثناءها با قلاب در کد به دنبال خطا بگردند. البته این کار ممکن است گاهی یک ضدالگو در نظر گرفته شود زیرا می‌تواند خطاهای نامربوط برنامه را در بلند مدت از بین ببرد و پنهان کند و اشکال‌زدایی برنامه های شما را بسیار سخت‌تر کند.

البته، می‌توانید این ایده را بیشتر گسترش دهید و به لحاظ منطقی استثنائات خود را در سلسله مراتب فرعی گروه‌بندی کنید. اما مراقب باشید، زیاده‌روی کردن در این کار، ورود پیچیدگی غیرضروری به کدهای شما را آسان می‌کند.

٤-٣-١. نکات کلیدی استثناهای سفارشی

- با تعریف انواع استثناءهای اختصاصی، ماهیت کد شما با وضوح بیشتری بیان خواهد شد و اشکال‌زدایی کدها راحت‌تر خواهد بود.
- استثناءهای سفارشی خود را از کلاس استثنای داخلی پایتون Exception یا کلاس‌های استثنای اختصاصی مانند ValueError یا KeyError تولید کنید.
- می‌توانید از وراثت برای تعریف سلسله مراتب استثناءهای دارای گروه‌بندی منطقی استفاده کنید.

٤-٤. Clone اشیاء برای تفریح و سود

هنگام ساخت یک نمونه از یک شیء در پایتون، کپی‌هایی از اشیاء ایجاد نمی‌شود. بلکه تنها نام جدیدی به یک شیء اختصاص داده می‌شود. در مورد اشیاء تغییرناپذیر، معمولا تغییری ایجاد نمی‌شود.

اما برای کار با اشیاء تغییرپذیر یا مجموعه‌هایی از اشیاء تغییرپذیر، ممکن است به دنبال روشی برای ایجاد "کپی‌های واقعی" یا "کلون‌هایی" از این اشیاء باشید.

اساسا، گاهی کپی‌هایی از اشیاء را می‌خواهید تا بتوانید بدون انجام اصلاحات خودکار در شیء اصلی، اصلاحات لازم را انجام دهید. در این فصل، می‌خواهم خلاصه‌ای در مورد نحوه کپی یا "clone" اشیاء در پایتون و برخی هشدارهای مربوط به آن صحبت کنم.

اجازه دهید با بررسی نحوه کپی کردن مجموعه‌های داخلی[1] پایتون شروع کنیم. مجموعه‌های تغییرپذیر داخلی پایتون مانند لیست‌ها، دیکشنری‌ها و set ها را می‌توان با فراخوانی توابع آن‌ها روی مجموعه موجود کپی کرد.

```
new_list = list(original_list)
new_dict = dict(original_dict)
new_set = set(original_set)
```

با این حال، این روش برای اشیاء سفارشی مناسب نیست و از همه مهم‌تر اینکه، این روش فقط کپی‌های سطحی[2] را ایجاد می‌کند. برای مجموعه‌های تغییرپذیر مانند لیست ها، دیکشنری‌ها و setها، یک تفاوت مهم بین کپی سطحی و کپی عمیق[3] وجود دارد.

کپی سطحی به معنای ایجاد شیء جدید مجموعه و بعد مقداردهی آن با مراجعه به اشیاء موجود در لیست اصلی است. در حقیقت، کپی سطحی فقط یک عمق دارد. روند

1 Built-in collections
2 Shallow copy
3 Deep copy

کپی‌برداری تکرار نمی‌شود و از این رو، کپی جدیدی از اشیاء درون مجموعه کپی شده، ساخته نخواهد شد.

کپی عمیق، روند کپی را بصورت برگشتی[1] انجام می‌دهد. این بدین معنی است که ابتدا شیء مجموعه جدید ساخته می‌شود و بعد بصورت برگشتی با کپی اشیاء فرزند موجود در مجموعه اصلی مقداردهی می‌شود. کپی‌برداری از یک شیء با این روش، کل درخت شی[2] را طی می‌کند تا کلون کاملاً مستقل از شیء اصلی و همه فرزندان آن ایجاد کند.

می‌دانم، توضیحات کمی طولانی بود. پس اجازه دهید چند مثال را بررسی کنیم تا این تفاوت بین کپی‌های سطحی و عمیق را مشخص کنیم.

٤-٤-١. ایجاد کپی‌های سطحی

در مثال زیر، لیست تو در توی جدیدی ایجاد خواهیم کرد و بعد به صورت سطحی آن را با تابع list() کپی می‌کنیم.

```
>>> xs = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> ys = list(xs) # Make a shallow copy
```

این بدان معناست که ys حالا یک شیء جدید و مستقل با محتویات مشابه با xs خواهد بود. با بررسی هر دو شیء می‌توانید این موضوع را تایید کنید.

```
>>> xs
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> ys
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

برای تایید اینکه ys واقعا مستقل از لیست اصلی است، اجازه دهید آزمایش کوچکی را انجام دهیم. شما می‌توانید زیرلیست جدیدی را امتحان کرده و آن را به لیست اصلی

1 recursive
2 Object tree

(xs) اضافه کنید و سپس بررسی کنید تا مطمئن شوید این اصلاحات بر کپی (ys) اثر نمی‌گذارد.

```
>>> xs.append(['new sublist'])
>>> xs
[[1, 2, 3], [4, 5, 6], [7, 8, 9], ['new sublist']]
>>> ys
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

همانطور که می‌بینید، این کار اثر مورد انتظار را داشت. اصلاح لیست کپی شده در سطح "ظاهری" اصلا مشکل نبود.

با این حال، چون فقط کپی سطحی لیست اصلی را ایجاد کردیم، ys هنوز حاوی ارجاع‌هایی به اشیاء فرزند لیست اصلی ذخیره شده در xs است. این فرزندان به صورت کامل کپی نشده بودند.

بنابراین، هنگامی که یکی از اشیاء فرزند را در xs اصلاح می‌کنید، بازتاب این اصلاحات در ys نیز خواهد بود! این اتفاق به این دلیل است که هر دو لیست، اشیاء فرزند مشابهی را به اشتراک می‌گذارند. این کپی فقط یک کپی سطحی است، درواقع در مرحله کپی کردن، تنها یک عمق کپی اعمال می‌شود.

```
>>> xs[1][0] = 'X'
>>> xs
[[1, 2, 3], ['X', 5, 6], [7, 8, 9], ['new sublist']]
>>> ys
[[1, 2, 3], ['X', 5, 6], [7, 8, 9]]
```

در مثال فوق (به ظاهر) فقط xs را تغییر داده‌ایم. اما معلوم می‌شود که هر دو زیرلیست با اندیس ۱ در xs و ys اصلاح شدند. از طرف دیگر، سبب وقوع این امر این بود که فقط کپی سطحی لیست اصلی را ایجاد کرده بودیم.

اگر در وهله اول کپی عمیق از xs ایجاد کرده بودیم، هر دو شیء کاملاً مستقل بودند. این تفاوت کاربردی بین کپی‌های سطحی و عمیق اشیاء است.

حال می‌دانید که چگونه کپی‌های سطحی برخی کلاس‌های مجموعه داخلی پایتون را ایجاد کنید و تفاوت بین کپی‌برداری سطحی و عمیق را نیز می‌دانید. سؤالاتی که حالا می‌خواهیم به آن‌ها پاسخ دهیم عبارتند از:

- چگونه می‌توانید کپی‌های عمیق را در مجموعه‌های داخلی پایتون ایجاد کنید؟
- چگونه می‌توانید کپی‌های سطحی و عمیق اشیاء دلخواه، از جمله کلاس‌های سفارشی را ایجاد کنید؟

پاسخ این سؤالات در ماژول کپی در کتابخانه استاندارد پایتون قرار دارد. این ماژول رابط ساده‌ای را برای ایجاد کپی‌های سطحی و عمیق اشیاء دلخواه پایتون فراهم می‌کند.

٤-٤-٢. ایجاد کپی‌های عمیق

اجازه دهید مثال قبلی در مورد کپی‌برداری از لیست را، با یک تفاوت مهم تکرار کنیم. این بار می‌خواهیم از تابع deepcopy() تعریف شده در ماژول copy پایتون استفاده کنیم و یک نسخه کپی عمیق ایجاد کنیم.

```
>>> import copy
>>> xs = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> zs = copy.deepcopy(xs)
```

وقتی xs و کلون zs آن که با copy.deepcopy() ایجاد شد را بررسی کنید، مشاهده خواهید کرد که هر دوی آن‌ها دوباره یکسان به نظر می‌رسند.

```
>>> xs
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> zs
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

با این حال، اگر در یکی از اشیاء فرزند در شی اصلی (xs) اصلاحاتی انجام دهید، خواهید دید که این اصلاحات روی کپی عمیق (zs) اثری نخواهد داشت.

این بار هر دو شیء، اصل و کپی، کاملاً مستقل از یکدیگر هستند. کپی xs به صورت برگشتی کلون‌سازی شد، که شامل همه اشیاء فرزند آن می‌باشد.

```
>>> xs[1][0] = 'X'
>>> xs
[[1, 2, 3], ['X', 5, 6], [7, 8, 9]]
>>> zs
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

ممکن است بخواهید مدتی با مفسر پایتون بنشینید و این مثال‌ها را اجرا کنید. وقتی مستقیما مثال‌ها را اجرا می‌کنید، فکر کردن در مورد اشیاء کپی‌برداری شده راحت تر است.

در هر حال، می‌توانید با استفاده از یک تابع دیگر در ماژول کپی، کپی‌های سطحی نیز ایجاد کنید. تابع copy.copy() کپی‌های سطحی اشیاء را ایجاد می‌کند.

اگر لازم است ارتباط واضحی بین اشیاء برقرار کنید و در جایی از کد خود یک کپی سطحی ایجاد کنید، این کار مفید خواهد بود. استفاده از تابع copy.copy() این امکان را به شما می‌دهد تا این واقعیت را درک کنید. برای مجموعه‌های داخلی، این کار بیشتر پایتونیک به نظر می‌رسد تا صرفا از توابع list، dict و set پایتون برای ایجاد کپی‌های سطحی استفاده کنید.

٤-٤-٣. کپی‌برداری از اشیاء دلخواه

سؤالی که باید به آن پاسخ دهیم این است که چگونه می‌توانیم کپی‌های سطحی و عمیق اشیاء دلخواه، از جمله کلاس‌های سفارشی را ایجاد کنیم.

حال به بررسی آن می‌پردازیم.

درحقیقت دوباره ماژول کپی به کمک ما می‌آید. توابع copy.copy() و copy.deepcopy() را می‌توان برای تکثیر هر شی بکار برد.

یکبار دیگر، بهترین روش برای درک نحوه استفاده از این‌ها، انجام آزمایش است. می‌خواهم این آزمایش را بر پایه مثال قبلی کپی‌برداری از لیست انجام دهم. اجازه دهید با تعریف کلاس نقطه دوبعدی ساده شروع کنیم.

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f'Point({self.x!r}, {self.y!r})'
```

امیدوارم موافق باشید که این موضوع نسبتا ساده بود. متد داندر ()__repr__ را اضافه کردم تا بتوانیم به راحتی اشیاء ایجاد شده از این کلاس را در مفسر پایتون بررسی کنیم.

در مرحله بعدی، با استفاده از ماژول کپی نمونه ای از شیء Point ایجاد می‌کنیم و بعد (بصورت سطحی) آن را کپی می‌کنیم.

```
>>> a = Point(23, 42)
>>> b = copy.copy(a)
```

اگر محتویات شیء Point اصلی و کپی سطحی آن را بررسی کنیم، نتیجه زیر را خواهیم دید.

```
>>> a
Point(23, 42)
>>> b
Point(23, 42)
>>> a is b
False
```

این مورد دیگری است که باید آن را در ذهن نگه دارید. چون شیء Point برای مختصات خود از نوع تغییرناپذیر داده (int) استفاده می‌کند، پس هیچ تفاوتی بین کپی سطحی و عمیق در این مورد وجود ندارد.

اجازه دهید مثال پیچیده‌تری را در نظر بگیریم. می‌خواهم کلاس دیگری را برای نمایش مستطیل‌های ۲ بعدی تعریف کنم. این کار را به روشی انجام خواهم داد که به ما امکان ایجاد سلسله مراتب شیء پیچیده را بدهد. مستطیل‌های من از اشیاء نقطه‌ای برای نمایش مختصات خود استفاده خواهند کرد.

```
class Rectangle:
    def __init__(self, topleft, bottomright):
        self.topleft = topleft
        self.bottomright = bottomright

    def __repr__(self):
        return (f'Rectangle({self.topleft!r}, '
                f'{self.bottomright!r})')
```

از طرف دیگر، ابتدا می‌خواهیم کپی سطحی نمونه مستطیل را ایجاد کنیم.

```
rect = Rectangle(Point(0, 1), Point(5, 6))
srect = copy.copy(rect)
```

در صورت بررسی مستطیل اصلی و کپی آن، خواهید دید که نادیده گرفتن تابع () __repr__ چقدر موثر است و اینکه روند کپی سطحی طبق انتظار ما کار می‌کند.

```
>>> rect
Rectangle(Point(0, 1), Point(5, 6))
>>> srect
Rectangle(Point(0, 1), Point(5, 6))
>>> rect is srect
False
```

به یاد داشته باشید که چگونه نمونه قبلی لیست تفاوت بین کپی‌های عمیق و سطحی را نشان داد؟ در اینجا می‌خواهم از رویکردی مشابه استفاده کنم. یک شیء را از قسمت

پایین در سلسله‌مراتب شیء اصلاح خواهم کرد و بعد بازتاب این تغییر را در کپی (سطحی) نیز مشاهده خواهید کرد.

```
>>> rect.topleft.x = 999
>>> rect
Rectangle(Point(999, 1), Point(5, 6))
>>> srect
Rectangle(Point(999, 1), Point(5, 6))
```

امیدوارم این توابع طوری که انتظار آن را داشتید، رفتار کنند. در مرحله بعد، کپی عمیق مستطیل اصلی را ایجاد خواهیم کرد. پس اصلاحات دیگری را اعمال کرده و خواهید دید که کدام اشیاء تحت تأثیر قرار می‌گیرند.

```
>>> drect = copy.deepcopy(srect)
>>> drect.topleft.x = 222
>>> drect
Rectangle(Point(222, 1), Point(5, 6))
>>> rect
Rectangle(Point(999, 1), Point(5, 6))
>>> srect
Rectangle(Point(999, 1), Point(5, 6))
```

بفرمائید! این بار کپی عمیق (drect) کاملاً مستقل از کپی اصلی (rect) و کپی سطحی (srect) است.

در اینجا قسمت زیادی از مبحث را پوشش داده‌ایم، اما هنوز نکته‌های ریزتری برای کپی‌برداری از اشیاء وجود دارند.

برای پرداخت بیشتر این موضوع، ممکن است بخواهید در مورد مستندات ماژول کپی و احتمالا کد منبع ماژول بیشتر مطالعه کنید. برای مثال، اشیاء می‌توانند با تعریف توابع خاص __copy__() و __deepcopy__() روی آن‌ها، نحوه کپی‌برداری را کنترل کنند.

٤-٤-٤. نکات کلیدی Clone اشیاء در پایتون

- ایجاد یک کپی سطحی از یک شیء، اشیاء فرزند را کلونی یا تکثیر نمی‌کند. بنابراین، این کپی کاملاً مستقل از کپی اصلی نیست.
- کپی عمیق شیء بصورت برگشتی اشیاء فرزند را کلونی خواهد کرد. کلون کاملاً مستقل از کپی اصلی است، اما ایجاد کپی عمیق آهسته‌تر است.
- می‌توانید اشیاء دلخواه (از جمله کلاس‌های سفارشی) را با ماژول copy، کپی برداری کنید.

٤-٥. بررسی وراثت در کلاس‌های انتزاعی پایه[1]

کلاس‌های انتزاعی پایه (ABCها) برای اطمینان حاصل کردن از پیاده سازی متدهای خاصی از کلاس پایه هستند. در این فصل در مورد مزایای کلاس‌های انتزاعی پایه و نحوه تعریف آن‌ها با ماژول داخلی abc پایتون آموزش داده خواهد شد.

کلاس‌های انتزاعی پایه برای چه مواردی مناسب هستند؟ چند وقت پیش بحثی درباره اینکه از کدام الگو برای پیاده سازی سلسله مراتب کلاس پایدار[2] در پایتون استفاده کنم، داشتم. هدف، تعریف سلسله مراتبی از یک کلاس ساده برای یک اند سرویس، یک روش پایدار و مطلوب برای برنامه‌نویسان بود.

ما یک کلاس BaseService داشتیم که یک interface مشترک و چندین پیاده سازی معمول (غیرانتزاعی) را تعریف می‌کرد. پیاده‌سازی های معمول کارهای مختلفی را انجام می‌دادند اما همه آن‌ها رابط کاربری یکسانی داشتند (MockService، RealService، و غیره). برای ایجاد یک رابطه صریح بین کلاس‌ها، همه کلاس‌ها از کلاس BaseService ارث برده شدند.

---

1 Abstract Base Classes
2 maintainable

برای اینکه این کد تا حد امکان پایدار و مورد پسند برنامه‌نویسان باشد، باید مطمئن شویم که:

- نمونه‌سازی کلاس پایه ناممکن باشد.
- فراموش کردن پیاده‌سازی متدهای interface در یکی از زیرکلاس‌ها در سریع‌ترین زمان ممکن خطایی را آشکار کند.

حال، چرا می‌خواهیم از ماژول abc پایتون برای حل این مسئله استفاده کنیم؟ طراحی فوق در سیستم‌های پیچیده نسبتا شایع است. برای اجبار به پیاده‌سازی متدهای کلاس پایه در زیرکلاس‌ها، معمولا از روش پایتونیکی بدین صورت استفاده می‌شود.

```
class Base:
    def foo(self):
        raise NotImplementedError()

    def bar(self):
        raise NotImplementedError()


class Concrete(Base):
    def foo(self):
        return 'foo() called'

    # Oh no, we forgot to override bar()...
    # def bar(self):
    #   return "bar() called"
```

بنابراین، از اقدامات اولیه خود برای حل این مسئله به چه نتیجه‌ای می‌رسیم؟ فراخوانی متدهای کلاس پایه به درستی خطایی را تولید می‌کند.

```
>>> b = Base()
>>> b.foo()
NotImplementedError
```

افزون بر این، نمونه‌سازی از کلاس Concrete طبق انتظار پیش می‌رود و اگر متدهای لازم پیاده‌سازی نشده باشند مانند bar() هنگام فراخوانی یک استثنا (خطا) تولید می‌شود.

```
>>> c = Concrete()
>>> c.foo()
'foo() called'
>>> c.bar()
NotImplementedError
```

این پیاده سازی اولیه مناسب است، اما هنوز کامل نیست. در آن نقاط ضعفی وجود دارند ولی می‌توانیم:

- کلاس پایه را بدون ایجاد خطا بدرستی نمونه‌سازی کنیم.
- زیرکلاس‌های غیرکاملی را پیاده‌سازی کنیم و تا زمانی که متد bar() را فراخوانی نکنیم، نمونه‌سازی معمول هیچ خطایی ایجاد نخواهد کرد.

با ماژول abc پایتون افزوده شده در پایتون ۲,۶، می‌توانیم عملکرد بهتری داشته باشیم و بقیه مسائل را حل کنیم. در پیاده سازی زیر، روش پیاده سازی را ارتقا می‌دهیم و از کلاس‌های انتزاعی پایه استفاده خواهیم کرد.

```
from abc import ABCMeta, abstractmethod


class Base(metaclass=ABCMeta):
    @abstractmethod
    def foo(self):
        pass

    @abstractmethod
    def bar(self):
        pass


class Concrete(Base):
    def foo(self):
        pass
```

```
    # We forget to declare bar() again...
```

این کار هنوز طبق پیش‌بینی رفتار می‌کند و سلسله‌مراتب کلاس را به صورت صحیح ایجاد می‌کند.

```
assert issubclass(Concrete, Base)
```

با این حال، در این کد مزیت دیگری را مشاهده می‌کنیم. هر وقت پیاده‌سازی متدهای انتزاعی را فراموش کنیم، زیرکلاس‌های پایه استثنا TypeError را در زمان نمونه‌سازی ایجاد می‌کند. استثناء مطرح شده به ما می‌گوید کدام متد یا متدها را از دست داده‌ایم.

```
>>> c = Concrete()
TypeError:
"Can't instantiate abstract class Concrete " \
"with abstract methods bar"
```

بدون abc، در صورت فراخوانی متد از دست رفته، فقط خطای NotImplementedError را دریافت می‌کنیم. کسب اطلاعات در مورد متدهای از دست رفته در زمان نمونه‌سازی مزیت بزرگی است. این کار نوشتن زیرکلاس‌های نامعتبر را دشوارتر می‌کند. اگر در حال نوشتن کد جدید هستید، ممکن است این کار چندان مهمی نباشد، اما قول می‌دهم چند هفته یا چند ماه بعد، به نفعتان خواهد بود.

دریافتیم که این کار اغلب اوقات باعث می‌شود سلسله مراتب کلاس‌ها پایدارتر و با اطمینان بالاتری قابل تغییر باشند. استفاده از ABC به وضوح هدف برنامه‌نویس را بیان می‌کند و کد ارتباطی را ایجاد می‌کند. به شما توصیه می‌کنم تا مستندات ماژول abc را بخوانید و به وضعیت‌هایی توجه کنید که اعمال این الگو در آن‌ها منطقی است.

٤-٥-١. نکات اصلی کلاس‌های انتزاعی پایه

- کلاس‌های انتزاعی پایه (ABC) تصریح می‌کنند که کلاس‌های بدست آمده، متدهای خاصی را از کلاس پایه در زمان نمونه‌سازی پیاده‌سازی می‌کنند.
- استفاده از ABC به جلوگیری از اشکالات کمک می‌کند و حفظ سلسله مراتب کلاس را آسان‌تر می‌کند.

٤-٦. آشنایی با namedtupleها

پایتون با نوع اختصاصی کانتینر "namedtuple" همراه است که به نظر نمی‌رسد چندان سزاوار تامل باشد. این یکی از آن ویژگی‌های شگفت‌انگیز در پایتون است که با دید باز نیز قابل مشاهده نیست.

استفاده از namedtupleها می‌تواند جایگزین بسیار خوبی برای تعریف دستی کلاس باشد و ویژگی‌های جالب دیگری نیز دارند که در این قسمت قصد دارم شما را با آن‌ها آشنا کنم.

حال، namedtuple چیست و چه چیزی آن را خاص‌تر می‌کند؟ روشی مناسب برای فکر کردن در مورد namedtupleها، مشاهده آن‌ها بصورت افزونه‌ای از نوع داده‌های tuple است.

Tupleها در پایتون، ساختار داده ساده‌ای برای گروه‌بندی اشیاء دلخواه هستند. tupleها تغییرناپذیر هستند. یعنی پس از ایجاد، آن‌ها را نمی‌توان اصلاح کرد. در ادامه مثالی مختصر در این زمینه وجود دارد.

```
>>> tup = ('hello', object(), 42)
>>> tup
('hello', <object object at 0x105e76b70>, 42)
>>> tup[2]
42
>>> tup[2] = 23
TypeError:
"'tuple' object does not support item assignment"
```

یک نکته مهم در مورد tupleهای ساده این است که داده‌هایی که در آن‌ها ذخیره می‌کنید را فقط با استفاده از اندیس‌های عدد صحیح استخراج کرد. نمی‌توانید به مقادیر ذخیره شده در tuple نام‌هایی اختصاص دهید. این می‌تواند بر خوانایی کد اثر بگذارد.

همچنین، tuple همواره از ساختار موقتی برخوردار است. کسب اطمینان از اینکه دو تاپل از تعداد فیلدهای یکسان و خصوصیات ذخیره شده مشابه برخوردارند، دشوار است. این کار ممکن است باعث مشکلاتی در هنگام کار با tuple ها شود.

٤-٦-١. استفاده از namedtuple ها برای نجات

هدف namedtupleها حل این مشکل است.

اول از همه، namedtupleها دقیقا مانند تاپل‌های معمولی کانتینرهای تغییرناپذیر هستند. هنگام ذخیره‌سازی داده‌های سطح بالا روی namedtupleها ، نمی‌توانید آن را به روزرسانی یا اصلاح کنید. تمام ویژگی‌ها موجود روی شیء namedtuple از اصل "یکبار بنویس، چند بار بخوان" پیروی می‌کنند.

علاوه بر این، namedtupleها، تاپل‌های خوب نامگذاری شده هستند. به هر شیء ذخیره شده در آن‌ها می‌توان از طریق شناسه منحصر به فرد (و قابل فهم برای انسان[1]) دسترسی پیدا کرد. این کار شما را از یادآوری اندیس‌های عدد صحیح یا استفاده از راه‌حل‌هایی مانند تعیین ثابت‌های عدد صحیح بصورت کلمات یادآور برای اندیس‌ها رها می‌کند.

حال، namedtuple به چه شکلی است؟

```
>>> from collections import namedtuple
>>> Car = namedtuple('Car' , 'color mileage')
```

namedtupleها در پایتون ۲.۶ به کتابخانه استاندارد پایتون افزوده شدند. برای استفاده از آن‌ها، باید ماژول collections را import کنید. در مثال فوق، نوع ساده‌ای از

1 Human readable

شیء Car را با دو فیلد color (رنگ) و mileage (مسافت پیموده شده برحسب مایل) تعریف کردم.

شاید تعجب‌آور باشد که چرا رشته 'car' را به عنوان اولین آرگومان تابع namedtuple در این مثال انتخاب می‌کنم.

در مستندات پایتون به این پارامتر "typename" گفته می‌شود. این نام کلاس جدیدی است که با فراخوانی تابع namedtuple ایجاد می‌شود.

چون namedtuple روشی برای دانستن نام متغیر اختصاص داده شده به کلاس حاصل از آن ندارد، پس باید صریحا به آن بگوییم که قصد داریم از چه نامی برای کلاس استفاده کنیم. نام کلاس در داک‌استرینگ و پیاده‌سازی __repr__ که namedtuple بطور خودکار برای ما ایجاد می‌کند، استفاده شده است.

در این مثال، نابهنجاری دستوری دیگری وجود دارد. چرا فیلدها را به صورت رشته ای انتخاب می‌کنیم که نام‌های آن‌ها را بصورت 'car mileage' کدگذاری کند؟

پاسخ این است که تابع namedtuple تابع split() را روی نام‌های رشته فراخوانی می‌کند تا رشته آن را به لیستی از نام‌ها، تجزیه کند. پس این در واقع کوتاه‌نویسی شده قطعه کد زیر است.

```
>>> 'color mileage'.split()
['color', 'mileage']
>>> Car = namedtuple('Car', ['color', 'mileage'])
```

البته، می‌توانید لیستی را مستقیما با نام‌های رشته‌ها وارد کنید. مزیت استفاده از لیست این است که اگر بخواهید آن را به چندین خط تجزیه کنید، این کار ساده‌تر خواهد بود.

```
>>> Car = namedtuple('Car', [
        'color',
        'mileage',
])
```

حال می‌توانید به راحتی شیء جدید Car را ایجاد کنید. شیء car طوری رفتار می کند که گویی کلاس Car را به صورت دستی تعریف کرده‌اید و به آن سازنده‌ای داده-اید که "رنگ" و مقدار "مسافت پیموده شده بر حسب مایل" را می‌پذیرد.

```
>>> my_car = Car('red', 3812.4)
>>> my_car.color
'red'
>>> my_car.mileage
3812.4
```

علاوه بر دسترسی به مقادیر ذخیره شده در شناسه namedtuple، می‌توانید از طریق اندیس به آن‌ها نیز دسترسی پیدا کنید. بدین طریق، namedtuple را می‌توان بصورت جایگزینی برای tuple های معمولی استفاده کرد.

```
>>> my_car[0]
'red'
>>> tuple(my_car)
('red', 3812.4)
```

باز کردن tuple و عملگر * برای بازکردن آرگومان‌های تابع نیز طبق انتظار کار می کند.

```
>>> color, mileage = my_car
>>> print(color, mileage)
red 3812.4
>>> print(*my_car)
red 3812.4
```

حتی یکی از فواید استفاده از namedtuple ها این است که می‌توانید نمایش رشته‌ای زیبایی برای اشیاء اختصاصی بدون هیچ زحمتی داشته باشید.

```
>>> my_car
Car(color='red' , mileage=3812.4)
```

مانند تاپل‌ها، namedtuple ها تغییرناپذیر هستند. اگر یکی از فیلدهای آن‌ها را بازنویسی کنید، استثنا AttributeError را دریافت خواهید کرد.

```
>>> my_car.color = 'blue'
AttributeError: "can't set attribute"
```

اشیاء namedtuple بصورت کلاس‌های معمولی داخلی در پایتون پیاده‌سازی می شوند. همچنین از نظر بهینگی حافظه، بهتر از کلاس‌های معمولی پایتون هستند و به اندازه tupleهای معمولی از حافظه کارآمدی برخوردار هستند.

یک روش خوب برای مشاهده آن‌ها این است که تصور کنید namedtuple ها میانبری حافظه کارآمد، برای تعریف کلاس‌های تغییرناپذیر در پایتون هستند.

٤-٦-٢. ایجاد زیرکلاس برای namedtuple ها

چون namedtuple ها روی کلاس‌های معمولی پایتون ساخته می‌شوند، حتی می توانید متدهایی را به شیء namedtuple اضافه کنید.

برای مثال، می‌توانید مانند هر کلاس دیگری، کلاس namedtuple را گسترش دهید و متدها و ویژگی‌های جدیدی را به آن اضافه کنید. در ادامه مثالی ارائه شده است.

```
Car = namedtuple('Car', 'color mileage')


class MyCarWithMethods(Car):
    def hexcolor(self):
        if self.color == 'red':
            return '#ff0000'
        else:
            return '#000000'
```

حال می‌توانیم شیء MyCarWithMethods را نمونه‌سازی کنیم و متد hexcolor() آن را فراخوانی کنیم.

```
>>> c = MyCarWithMethods('red', 1234)
>>> c.hexcolor()
'#ff0000'
```

با این حال، این ممکن است کمی ناخوشایند باشد. شاید اگر بخواهید کلاس با خصوصیات تغییرناپذیر داشته باشید، ارزش انجام این کار را دارد، همچنین در این شرایط ممکن است خود را در وضعیت بدتری قرار دهید.

برای مثال، اضافه کردن فیلد تغییرناپذیر جدید به دلیل نحوه ساخت namedtuple ها مستلزم دقت زیادی است. ساده‌ترین روش برای ایجاد سلسله مرتبه ای از namedtuple ها استفاده از ویژگی _fields در تاپل های پایه پایتون است.

```
>>> Car = namedtuple('Car', 'color mileage')
>>> ElectricCar = namedtuple(
        'ElectricCar', Car._fields + ('charge',))
```

از این کار نتیجه مطلوب زیر بدست می‌آید.

```
>>> ElectricCar('red', 1234, 45.0)
ElectricCar(color='red', mileage=1234, charge=45.0)
```

٤-٦-٣. متدهای کمکی درونی[1] در پایتون

علاوه بر ویژگی _fields، هر نمونه namedtuple چند متد کمکی دیگری نیز ارائه می‌کند که ممکن است مفید باشند. نام‌های آن‌ها همگی با یک کاراکتر زیرخط (_) شروع می‌شوند که معمولا آن متد یا ویژگی که "خصوصی" است و بخشی از رابط عمومی کلاس یا ماژول نیست را علامت‌گذاری می‌کند.

با namedtupleها، قرارداد نامگذاری زیرخط معنای متفاوتی دارد. این متدها و ویژگی‌های کمکی، بخشی از رابط عمومی namedtuple ها هستند. متدها و ویژگی‌های کمکی برای جلوگیری از تداخل نامگذاری‌ها با فیلدهای تاپل تعریف شده

1 Built-in Helper Methods

توسط کاربر، بدین صورت نامگذاری شدند. بنابراین ادامه دهید و در صورت نیاز از آن ها استفاده کنید! نگران خصوصی بودنشان نباشید!

می‌خواهم چند سناریو به شما نشان دهم که در آن متدهای کمکی namedtuple ممکن است سودمند باشد. اجازه دهید با متد کمکی _asdict() شروع کنیم. این کار محتویات namedtuple را بصورت دیکشنری برمی‌گرداند.

```
>>> my_car._asdict()
OrderedDict([('color', 'red'), ('mileage', 3812.4)])
```

این برای جلوگیری از اشتباه تایپی نام فیلدها در هنگام ایجاد خروجی JSON بسیار عالی است، برای مثال به قطعه کد زیر دقت کنید.

```
>>> json.dumps(my_car._asdict())
'{"color": "red", "mileage": 3812.4}'
```

دیگر متد کمکی مفید تابع _replace() است. این یک کپی (سطحی) از تاپل ایجاد می‌کند و به شما امکان جایگزینی انتخابی برخی فیلدهای آن را می‌دهد.

```
>>> my_car._replace(color='blue')
Car(color='blue', mileage=3812.4)
```

و در پایان، می‌توان از متد _make() برای ایجاد نمونه‌های جدید namedtuple در توالی یا تکرار استفاده کرد.

```
>>> Car._make(['red', 999])
Car(color='red', mileage=999)
```

٤-٦-٤. چه زمانی باید از namedtuple ها استفاده کنیم؟

عبارت namedtuple می‌تواند روشی ساده برای پاکسازی کد شما و خوانا کردن آن با اجرای ساختار بهتری برای داده‌های شما باشد.

برای مثال، فهمیدم که استفاده از انواع داده‌های موقت مانند دیکشنری‌ها با فرمت ثابت به namedtuple ها به من کمک می‌کند تا اهداف خود را با وضوح بیشتری بیان کنم. اغلب وقتی این سازماندهی مجدد را انجام می‌دهم، برای مسئله‌ای که با آن مواجه هستم، راه‌حل جادویی بهتری پیدا می‌کنم.

همچنین استفاده از namedtuple ها روی تاپل‌ها و دیکشنری‌های ساختاریافته می تواند زندگی همکاران من را راحت‌تر کند، چون باعث می‌شود داده‌ها پیرامون "خوداستنادی[1]" کدهای نوشته شده عمل کنند.

از طرف دیگر، اگر آن‌ها به من در نوشتن کد تمیزتر و نگهداشت‌پذیرتر کمک نکنند، از namedtuple ها به نفع خود استفاده نخواهم کرد. مانند سایر تکنیک‌های نشان داده شده در این کتاب، گاهی می‌توانند نکات بسیار خوبی در این زمینه باشند. با این حال، اگر با احتیاط مناسبی از آن‌ها استفاده کنید، بی‌شک namedtuple ها می توانند کد پایتون بهتر و گویاتری ایجاد کنند.

٤-٦-٥. نکات کلیدی namedtuple ها در پایتون

- نوع داده collections.namedtuple میانبر حافظه کارآمدی برای تعریف دستی کلاس تغییرناپذیر در پایتون است.
- عبارت namedtuple با اعمال ساختار قابل درک روی داده ها، می‌تواند به پاکسازی کد شما کمک کند.
- عبارت namedtuple چند متد کمکی مفید که همگی با یک زیرخط شروع می‌شوند ارائه می‌دهد که بخشی از رابط عمومی هستند. استفاده از آن‌ها اشکالی ندارد.

1 Self-documenting

٤-۷. مشکلات شیء کلاس در مقابل نمونه‌ای از کلاس

علاوه بر ایجاد تمایز بین متدهای کلاس و متدهای نمونه‌ای از کلاس، همچنین مدل، اشیاء پایتون تفاوت بین شیء کلاس و نمونه‌ها را تشخیص می‌دهند.

این نه تنها تمایز مهمی است، بلکه سبب ایجاد مشکل در برنامه‌نویس‌های تازه کار پایتون نیز می‌شود. برای مدت زمانی طولانی، به منظور درک این مفاهیم از مفاهیم پایه، زمان موردنیاز را سرمایه گذاری نکردم. بنابراین آزمایش‌های اولیه OOP من با رفتارهای غافلگیرانه و اشکالات اتفاقی همراه شدند. در این فصل، هرگونه سردرگمی ادامه‌دار درباره این موضوع را با برخی نمونه‌های عملی پاکسازی خواهیم کرد.

همانطور که گفتم، دو نوع ویژگی داده روی اشیاء پایتون وجود دارد. متغیرهای کلاس [1] و متغیرهای نمونه [2].

متغیرهای کلاس در داخل تعریف کلاس اعلام می‌شوند. آن‌ها با هیچ نمونه خاصی از کلاس ارتباط ندارند. از طرفی، متغیرهای کلاس محتویات خود را روی خود کلاس ذخیره می‌کنند و همه اشیاء ایجاد شده از کلاس خاص، دسترسی به مجموعه یکسانی از متغیرهای کلاس را دارند. برای مثال، این بدان معناست که اصلاح متغیرهای کلاس همزمان بر تمام نمونه‌های آن شیء اثر می‌گذارد.

متغیرهای نمونه همواره با نمونه خاصی از شیء مرتبط هستند. محتویات آن‌ها روی کلاس اصلی ذخیره نمی‌شوند. بلکه روی هر شیء تکی ایجاد شده از کلاس ذخیره می‌شوند. بنابراین، محتویات متغیر نمونه کاملاً مستقل از نمونه شیء بعدی است. پس، تغییر متغیرهای یک نمونه فقط روی یک نمونه از شیء اثر می‌گذارد.

خب، این کاملاً انتزاعی بود. اکنون بهتر است مقداری کد بررسی کنیم! اجازه دهید مثال قدیمی "سگ" را بررسی کنیم. به چند دلیل، برنامه‌های آموزشی OOP همیشه

1 Class variables
2 Instance variables

برای نشان دادن نکته خود از اتومبیل‌ها یا حیوانات خانگی استفاده می‌کنند و تجزیه آن ها با روش‌های سنتی مشکل است.

یک سگ شاد به چه چیزی نیاز دارد؟ چهار پا و یک نام.

```
class Dog:
    num_legs = 4 # <- Class variable

    def __init__(self, name):
        self.name = name # <- Instance variable
```

بسیار خب، این نمایش شی‌گرا نمونه‌ای از وضعیت سگی که به تازگی توضیح داده ام، است. ایجاد نمونه‌های جدید سگ طبق انتظار کار می‌کند و هر یک از آن‌ها، متغیر نمونه‌ای تحت عنوان نام دریافت می‌کنند.

```
>>> jack = Dog('Jack')
>>> jill = Dog('Jill')
>>> jack.name, jill.name
('Jack', 'Jill')
```

در صورت استفاده از متغیرهای کلاس، انعطاف‌پذیری بیشتری وجود دارد. برای مثال می‌توانید به متغیر کلاس num_legs روی هر نمونه شیء Dog یا به صورت مستقیم در خود کلاس Dog دسترسی داشته باشید.

```
>>> jack.num_legs, jill.num_legs
(4, 4)
>>> Dog.num_legs
4
```

با این حال، در صورت دسترسی به متغیر نمونه از طریق کلاس، با خطای AttributeError مواجه خواهید شد. متغیرهای نمونه برای هر نمونه شیء خاص هستند و وقتی متد سازنده __init__ را اجراء می‌کند مقداردهی می‌شوند. آن‌ها حتی در خود کلاس نیز وجود ندارند.

این تمایز اصلی بین متغیرهای کلاس و متغیرهای نمونه است.

```
>>> Dog.name
AttributeError:
"type object 'Dog' has no attribute 'name'"
```

خب، تا اینجا خوب بوده است.

فرض می‌کنیم که جک، سگی است که هنگام غذا خوردن خود را به مایکروویو نزدیک می‌کند و در او یک جفت پا اضافی رشد می‌کند. چگونه این مسئله را در جعبه شنی کوچک کد که تا بحال بدست آورده‌ایم نمایش می‌دهید؟

اولین ایده برای حل مسئله، ممکن است صرفا تغییر متغیر num_legs در کلاس سگ باشد.

```
>>> Dog.num_legs = 6
```

اما به یاد داشته باشید، نمی‌خواهیم همه سگ‌ها دویدن روی شش پا را آغاز کنند. پس فعلا فقط هر نمونه سگ موجود در جهان کوچک خود به سوپر سگ تبدیل کرده‌ایم چون متغیر کلاس را تغییر داده‌ایم. و این کار بر روی همه سگ‌ها، حتی آنهایی که قبلا ایجاد شده‌اند، اثر می‌گذارد.

```
>>> jack.num_legs, jill.num_legs
(6, 6)
```

بنابراین این کار عملکرد چندانی نداشت. دلیل کار نکردن این است که تغییر متغیر کلاس در فضای نام‌های کلاس، بر همه نمونه‌های کلاس اثر می‌گذارد. اجازه دهید به تغییر متغیر کلاس برگردیم و بجای دادن یک جفت پاهای اضافی به همه، فقط به جک اختصاص دهیم.

```
>>> Dog.num_legs = 4
>>> jack.num_legs = 6
```

حال، این کار چه هیولاهایی خلق کرد؟ اجازه دهید بررسی کنیم.

```
>>> jack.num_legs, jill.num_legs, Dog.num_legs
(6, 4, 4)
```

خب، این نسبتا خوب به نظر می‌رسد (گذشته از این واقعیت که به جک بیچاره چند پای اضافه دادیم). اما چگونه این تغییر واقعا روی اشیاء سگ اثر گذاشته است؟

همانطور که می‌بینید، در اینجا مشکل این است که ضمن گرفتن نتیجه مدنظر خود (پاهای اضافی برای جک)، متغیر نمونه num_legs را برای نمونه جک معرفی کردیم و حالا متغیر جدید نمونه num_legs روی متغیرکلاسی به همین نام سایه می‌اندازد و هنگام دسترسی به نمونه شیء، آن را تحت‌الشعاع قرار داده و پنهان می‌کند.

```
>>> jack.num_legs, jack.__class__.num_legs
(6, 4)
```

همانطور که مشاهده می‌کنید، متغیرهای کلاس ظاهرا از همگام‌سازی خارج شدند. سبب وقوع این امر این بود که تغییر دادن متغیر jack.num_legs یک متغیرنمونه با همان نام متغیرکلاس ایجاد کرد.

آگاه شدن از آنچه در اینجا رخ داده، در پشت صحنه اهمیت بسیاری دارد. در نهایت قبل از اینکه محدوده سطح کلاس و سطح نمونه را در پایتون بفهمم، این عملکرد باعث شد تا مشکلات زیادی وارد کدهای من شوند.

اگر بخواهم واقعیت را بگویم، تلاش برای تغییر متغیرکلاس از طریق نمونه شیء، بر متغیر کلاس اصلی ما اثر می‌گذارد، که این از مشکلات OOP در پایتون است.

٤-٧-١. مثال بدون سگ

با اینکه هیچ سگی در ایجاد این فصل صدمه ندید، می‌خواهم مثال عملی دیگر از چیزهای مفیدی که می‌توانید با متغیرهای کلاس انجام دهید را برای شما بیان کنم. چیزی که کمی نزدیکتر به کاربردهای واقعی برای متغیرهای کلاس است.

کلاس CountedObject زیر تعداد دفعاتی که در طول عمر برنامه نمونه‌سازی شده را پیگیری می‌کند. (که درواقع معیار جالبی برای دانستن عملکرد برنامه است.)

```
class CountedObject:
    num_instances = 0

    def __init__(self):
        self.__class__.num_instances += 1
```

کلاس CountedObject متغیر کلاسیِ num_instances را حفظ می‌کند که به عنوان شمارنده مشترک عمل می‌کند. در هنگام تعریف کلاس، شمارنده را به صفر مقداردهی می‌کند و بعد آن را خالی می‌گذارد.

هر بار که نمونه جدیدی از این کلاس ایجاد می‌کنید، وقتی پایتون به صورت خودکار متد سازنده __init__ را اجراء می‌کند، شمارنده مشترک را با عدد یک جمع می‌کند تا یک واحد افزایش یابد.

```
>>> CountedObject.num_instances
0
>>> CountedObject().num_instances
1
>>> CountedObject().num_instances
2
>>> CountedObject().num_instances
3
>>> CountedObject.num_instances
3
```

به اینکه باید این کد را با یک حلقه اجرا کنید تا مطمئن شوید متغیر شمارنده در کلاس افزایش می‌یابد، توجه داشته باشید. اگر سازنده را به شرح زیر نوشته بودم، متحمل اشتباهاتی در محاسبات می‌شدم.

```
# WARNING: This implementation contains a bug


class BuggyCountedObject:
```

```
    num_instances = 0

    def __init__(self):
        self.num_instances += 1 # !!!
```

همانطور که مشاهده کردید، این پیاده‌سازی بد، هرگز متغیر شمارنده مشترک را افزایش نمی‌دهد.

```
>>> BuggyCountedObject.num_instances
0
>>> BuggyCountedObject().num_instances
1
>>> BuggyCountedObject().num_instances
1
>>> BuggyCountedObject().num_instances
1
>>> BuggyCountedObject.num_instances
0
```

قطعا حالا می‌توانید ببینید در کجا اشتباه کرده‌ام. این پیاده‌سازی (دارای اشکال) هرگز شمارنده مشترک را افزایش نمی‌دهد چون در مورد مثال "جک" که قبلا توضیح دادم مرتکب اشتباه شدم.

این پیاده‌سازی کارایی نخواهد داشت. چون بطور تصادفی متغیر کلاس num_instances را با ایجاد متغیر نمونه با همین نام در سازنده سایه زدم. (سایه انداختن یک متغیر روی متغیر دیگر باعث تغییر آن خواهد شد.)

این کار به درستی مقدار جدید شمارنده را محاسبه می‌کند (از ۰ تا ۱)، اما سپس نتیجه را در متغیر نمونه ذخیره می‌کند. این بدان معنی است که سایر نمونه‌های کلاس هرگز مقدار به روز رسانی شده شمارنده را نمی‌بینند.

همانطور که می‌بینید، انجام این کار اشتباه محض است. پس در هنگام مواجه با وضعیتی مشترک در یک کلاس، بهتر است مراقب باشید و انجام بررسی در محدوده کد را دو برابر کنید. تست‌های خودکار و بازبینی دقیق کد به این کار بسیار کمک می کند.

با این وجود، امیدوارم اینکه چرا و چگونه متغیرهای کلاس، به رغم وجود مشکلات آن‌ها می‌توانند در عمل ابزارهای مفیدی باشند را بتوانید مشاهده کنید. موفق باشید!

٤-٧-٢. نکات کلیدی متغیرهای شی در مقابل نمونه شی

- متغیرهای کلاسی برای اشتراک گذاری داده‌ها با همه نمونه‌های موجود از شیء هستند. آن‌ها متعلق به کلاس هستند و متعلق به نمونه خاصی نیستند و در بین همه نمونه‌های کلاس به اشتراک گذاشته می‌شوند.
- تمام داده‌های منحصر به فرد متعلق به متغیرهای نمونه، فقط متعلق به همان نمونه هستند. آن‌ها به نمونه‌های تکی شیء تعلق دارند و در سایر نمونه‌های کلاس به اشتراک گذاشته نمی‌شوند. هر متغیر نمونه، یک واحد حافظه پشتیبان مختص به خود را بدست می‌آورد.
- چون متغیرهای کلاس را می‌توان با متغیرهای نمونه با همان نام "سایه" انداخت، بازنویسی تصادفی متغیرهای کلاس به روشی که اشکالات و رفتارهای غیرعادی را در برنامه ایجاد کند، ساده است.

٤-٨. نمونه، کلاس و متدهای استاتیک به زبان ساده

در این فصل خواهید دید که در پس متدهای کلاس[1]، متدهای استاتیک[2] و متدهای معمول یک نمونه[3] در پایتون چه اتفاقی می‌افتد.

اگر درک بصری از تفاوت‌های این متدها داشته باشید، می‌توانید شی گرایی زیباتری در کدهای پایتون طراحی کنید و ارتباط بین بخش های کد، گویا و قابل فهم باشد.

اجازه دهید با نوشتن یک کلاس پایتونی حاوی نمونه‌های ساده برای هر سه نوع متد شروع کنیم.

---

1 Class methods
2 Static methods
3 Regular instance methods

```
class MyClass:
    def method(self):
        return 'instance method called', self

    @classmethod
    def classmethod(cls):
        return 'class method called', cls

    @staticmethod
    def staticmethod():
        return 'static method called'
```

نکته قابل توجه برای کاربران پایتون ۲: دکوریتورهای @staticmethod و @classmethod از پایتون ۲,۴ به بعد در دسترس هستند. پس این مثال در پایتون ۲ نیز کار خواهد کرد. در پایتون ۲ به جای اعلان ساده کلاس MyClass، لازم است کلاس را به سبک دیگری با سینتکس MyClass(object) تعریف کنید.

٤-٨-١. متدهای یک نمونه یا Instance method

متد اول در MyClass، به نام method، یک متد معمول در یک نمونه کلاس است. این نوع متد، متدی ساده است که معمولا از آن استفاده خواهید کرد. مشاهده می‌کنید که این متد از یک پارامتر یعنی self استفاده می‌کند که در هنگام فراخوانی این متد به نمونه MyClass اشاره می‌کند. البته، متدهای نمونه می‌توانند بیش از یک پارامتر را قبول کنند.

از طریق پارامتر self، متدهای نمونه می‌توانند آزادانه به ویژگی‌ها و سایر متدهای روی همان شی دسترسی پیدا کنند. این کار در هنگام تغییر حالت [1] یک شیء قدرت زیادی برای کنترل برنامه می‌دهد.

آن‌ها نه تنها می‌توانند حالت شی را تغییر دهند، بلکه متدهای نمونه می‌توانند از طریق ویژگی __class__.self به خود کلاس دسترسی پیدا کنند. این بدان معنی است که متدهای نمونه می‌توانند حالت کلاس را تغییر دهند.

٤-٨-٢. متدهای کلاس یا Class methods

اجازه دهید با روش دوم، MyClass.classmethod آشنا شویم. این متد را با دکوریتور @classmethod برای نشان دادن متد کلاس علامت‌گذاری کردیم.

به جای آرگومان ورودی self، متدهای کلاس آرگومان cls را دریافت می‌کنند که وقتی این متد فراخوانی می‌شود به کلاس اشاره می‌کند و به یک نمونه شیء اشاره نمی‌کند.

چون متدهای کلاس فقط به این آرگومان cls دسترسی دارند، پس نمی‌توانند وضعیت نمونه شیء را تغییر دهند. این کار مستلزم دسترسی به self است. با این حال، متدهای کلاس می‌توانند وضعیت کلاس را که به تمام نمونه‌های کلاس اعمال می‌شود، اصلاح کنند.

٤-٨-٣. متدهای استاتیک یا Static methods

متد سوم، MyClass.staticmethod است که با دکوریتور @staticmethod برای مشخص کردن این متد استاتیک، علامت‌گذاری کردم.

این نوع متد از آرگومان self یا cls استفاده نمی‌کند. البته می‌توان کاری انجام داد تا این متد پارامترهای دلخواهی را بپذیرد. در نتیجه، متد استاتیک نمی‌تواند حالت شی یا حالت کلاس را اصلاح کند. متدهای استاتیک محدود به آنچه که داده‌ها می‌توانند به

1 state

آن دسترسی داشته باشند، هستند. آن‌ها عمدتا روشی برای فضای نام متدهای شما می‌باشند.

٤-٨-٤. تفاوت انواع متدها در عمل

می‌دانم این بحث تا این لحظه کاملاً نظری بوده است. همچنین بر این باور هستم که کدنویسی و تمرین باعث درک بصری بهتری برای میزان تفاوت انواع متدها در عمل است. به همین دلیل حالا دست به کار خواهیم شد.

ابتدا به بررسی نحوه رفتار این متدها هنگام فراخوانی آن‌ها می‌پردازیم. با ایجاد نمونه ای از کلاس شروع می‌کنیم و سپس سه متد مختلف را روی آن فراخوانی می‌کنیم.

شیء MyClass به گونه‌ای پیاده‌سازی شده است که در آن هر متد، حاوی tuple برای بازگرداندن اطلاعات است، به صورتی که بتوانیم نحوه اجرای برنامه را ردیابی کنیم.

در متد نمونه[1] زیر مشخص شده است که در هنگام فراخوانی چه اتفاقی می‌افتد.

```
>>> obj = MyClass()
>>> obj.method()
('instance method called', <MyClass instance at
0x11a2>)
```

این کد نشان می‌دهد که متد نمونه از طریق آرگومان self به نمونه شیء (چاپ شده بصورت <MyClass instance>) دسترسی دارد.

هنگام فراخوانی متد، پایتون آرگومان self را با شیء نمونه، obj جایگزین می‌کند. می‌توانیم فرمت تغییر یافته با سینتکس فراخوانی نقطه‌ای obj.method()[2] را نادیده بگیریم و آن را به صورت دستی فراخوانی کنیم تا نتیجه مشابه را بدست آوریم.

1 Instance Method
2 Dot-call

```
>>> MyClass.method(obj)
('instance method called', <MyClass instance at
0x11a2>)
```

بدین ترتیب، متدهای نمونه می‌توانند از طریق ویژگی self.__class__ به خود کلاس دسترسی پیدا کنند. این کار متد نمونه را از لحاظ محدودیت های دسترسی قدرتمند می کند. آن‌ها می‌توانند آزادانه حالت روی نمونه شیء و خود کلاس را اصلاح کنند.

اجازه دهید در ادامه متدهای کلاس را بررسی کنیم.

```
>>> obj.classmethod()
('class method called', <class MyClass at 0x11a2>)
```

فراخوانی classmethod() به ما نشان می‌دهد که به شیء <MyClass instance> دسترسی ندارد. بلکه فقط به شیء <class MyClass> دسترسی دارد، که بیانگر خود کلاس می‌باشد (همه چیز حتی خود کلاس‌ها در پایتون شیء هستند).

به اینکه چگونه پایتون هنگام فراخوانی MyClass.classmethod()، به صورت خودکار کلاس را بعنوان اولین آرگومان به تابع مرتبط می کند، توجه داشته باشید. فراخوانی یک متد در پایتون از طریق سینتکس نقطه‌ای، سبب این رفتار می‌شود. آرگومان self در متد نمونه به همین طریق عمل می کند.

لطفا توجه داشته باشید که نام گذاری آرگومان‌های self و cls قراردادی است. می توانید به راحتی آن‌ها را به صورت the_object و the_class نام گذاری کنید و نتیجه مشابه را بدست آورید. تنها چیزی که اهمیت دارد این است که آن‌ها ابتدا در لیست پارامتر برای آن متد خاص قرار می گیرند.

حال زمان فراخوانی از طریق متد استاتیک است.

```
>>> obj.staticmethod()
'static method called'
```

دیدید که چگونه staticmethod() را روی شیء فراخوانی کردیم و توانستیم با موفقیت این کار را انجام دهیم. برخی برنامه‌نویس ها وقتی می‌فهمند فراخوانی با متد استاتیک روی نمونه شیء امکان‌پذیر است، شگفت زده می‌شوند.

در پشت صحنه، هنگام فراخوانی متد استاتیک با استفاده از سینتکس نقطه‌ای، پایتون با عبور دادن self یا آرگومان cls صرفا محدودیت های دسترسی را اعمال می کند.

این امر نشانگر این است که متدهای استاتیک نه می‌توانند به حالت نمونه شیء و نه حالت کلاس دسترسی داشته باشند. آن‌ها مانند توابع معمولی کار می کنند، اما متعلق به فضای نام کلاس (و هر نمونه) هستند.

حال، به بررسی مواردی خواهیم پرداخت که هنگام فراخوانی این متدها روی شیء خود کلاس رخ می‌دهد، بدون اینکه از قبل یک نمونه شیء ایجاد کنیم.

```
>>> MyClass.classmethod()
('class method called', <class MyClass at 0x11a2>)

>>> MyClass.staticmethod()
'static method called'

>>> MyClass.method()
TypeError: """unbound method method() must
    be called with MyClass instance as first
    argument (got nothing instead)"""
```

می‌توانیم classmethod() و staticmethod() را به راحتی فراخوانی کنیم، اما فراخوانی متد method() با خطای TypeError ناموفق است.

این چیزی است که انتظار آن می‌رفت. این بار نمونه شیء را ایجاد نکردیم و تابع نمونه را مستقیما روی خود کلاس فراخوانی کردیم. این بدان معنی است که هیچ راهی در پایتون برای ایجاد آرگومان self وجود ندارد، پس فراخوانی با استثناء TypeError انجام نمی‌شود.

این کار باعث تمایز آشکار بین این سه نوع روش می‌شود. اما نگران نباشید، قصد ندارم این موضوع را در اینجا پایان دهم. در دو بخش بعدی، به بررسی دو مثال کمی واقع گرایانه‌تر از زمان استفاده از این نوع متد خاص خواهم پرداخت.

مبنای مثال های خود را پیرامون یک کلاس پیتزا قرار خواهم داد.

```
class Pizza:
    def __init__(self, ingredients):
        self.ingredients = ingredients

    def __repr__(self):
        return f'Pizza({self.ingredients!r})'

>>> Pizza(['cheese', 'tomatoes'])
Pizza(['cheese', 'tomatoes'])
```

٤-٨-٥. تولید پیتزاهای خوشمزه با @classmethod

همانطور که در واقعیت با پیتزا روبرو شده‌اید، می‌دانید که متغیرهای بسیار خوشمزه ای وجود دارند.

```
Pizza(['mozzarella', 'tomatoes'])
Pizza(['mozzarella', 'tomatoes', 'ham', 'mushrooms'])
Pizza(['mozzarella'] * 4)
```

ایتالیایی‌ها قرن ها پیش پیتزا را کشف کردند، پس انواع خوشمزه پیتزا با نام‌های خاص آن‌ها است. در صورت بهره‌مندی از نام آن‌ها، عملکرد بهتری خواهیم داشت و به کاربران کلاس پیتزا خود، رابط بهتر برای ایجاد اشیاء پیتزایی که طلب می‌کنند ارائه می‌دهیم.

روشی خوب و تمیز برای انجام این کار با استفاده از متدهای کلاس به صورت توابع کارخانه[1] برای انواع مختلف پیتزایی که می‌توانیم ایجاد کنیم است.

1 Factory design pattern (object-oriented programming)

```
class Pizza:
    def __init__(self, ingredients):
        self.ingredients = ingredients

    def __repr__(self):
        return f'Pizza({self.ingredients!r})'

    @classmethod
    def margherita(cls):
        return cls(['mozzarella', 'tomatoes'])

    @classmethod
    def prosciutto(cls):
        return cls(['mozzarella', 'tomatoes', 'ham'])
```

به اینکه چگونه به جای فراخوانی مستقیم سازنده پیتزا، از آرگومان cls در متدهای کارخانه margherita و prosciutto استفاده می‌کنم، توجه داشته باشید.

این ترفندی است که می‌توانید از آن برای دنبال کردن اصل "خودت را تکرار نکن (DRY)" استفاده کنید. اگر تصمیم بگیریم این کلاس را در برخی از نقاط تغییر نام دهیم، لازم نیست به روز رسانی نام سازنده در تمام توابع کارخانه را به خاطر داشته باشیم.

حال، با این متدهای کارخانه چه کاری می‌توانیم انجام دهیم؟ اجازه دهید آنها را بررسی کنیم.

```
>>> Pizza.margherita()
Pizza(['mozzarella', 'tomatoes'])

>>> Pizza.prosciutto()
Pizza(['mozzarella', 'tomatoes', 'ham'])
```

همانطور که مشاهده می‌کنید، می‌توانیم از توابع کارخانه‌ای برای ایجاد اشیاء جدید پیتزا که دقیقا به روشی درست مانند روشی که ما می‌خواهیم پیکربندی شوند، استفاده کنیم. همه آن‌ها در داخل از سازنده مشابه __init__ استفاده می‌کنند و فقط میانبری برای به خاطر آوردن همه مواد تشکیل دهنده ارائه می‌دهند.

روشی دیگر برای بررسی استفاده از متدهای کلاس، فهمیدن این مطلب است که این کار به شما امکان تعریف سازنده‌های جایگزین برای کلاس‌های شما را فراهم می‌کند.

پایتون فقط از یک متد __init__ در هر کلاس استفاده می‌کند. استفاده از متدهای کلاس، افزودن سازنده‌های جایگزین بیشتر را در صورت لزوم امکان‌پذیر می‌کند. این کار می‌تواند رابطی برای خوداستنادی کلاس‌های شما (تا اندازه معینی) و ساده‌سازی کاربرد آن‌ها ایجاد کند.

٤-٨-٦. چه موقع باید از متدهای استاتیک استفاده کنیم

در اینجا، ارزیابی مثال مناسب کمی دشوارتر است، اما باید به شما بگویم که فقط مقایسه پیتزای هر چه نازکتر را ادامه خواهم داد. (یامی[1]!)

این چیزی است که باید در اینجا ارزیابی کنیم.

```
import math


class Pizza:
    def __init__(self, radius, ingredients):
        self.radius = radius
        self.ingredients = ingredients

    def __repr__(self):
        return (f'Pizza({self.radius!r}, '
                f'{self.ingredients!r})')

    def area(self):
        return self.circle_area(self.radius)

    @staticmethod
    def circle_area(r):
        return r ** 2 * math.pi
```

1 Yummy

حال در این رابطه چه تغییری انجام دادم؟ ابتدا، سازنده و متد __repr__ برای پذیرفتن آرگومان شعاع اضافی را اضافه کردم.

همچنین متد نمونه area() را اضافه کردم که مساحت پیتزا را محاسبه کرده و بازمی گرداند. این کار همچنین جایگزین مناسبی برای @property خواهد بود. اما این فقط مثالی سرگرم‌کننده است.

بجای محاسبه مستقیم مساحت در داخل area()، با استفاده از فرمول شناخته شده مساحت دایره، این روش را به متد استاتیک جداگانه circle_area() تبدیل کردم.

اجازه دهید این روش را بررسی کنیم.

```
>>> p = Pizza(4, ['mozzarella', 'tomatoes'])
>>> p
Pizza(4, {self.ingredients})
>>> p.area()
50.26548245743669
>>> Pizza.circle_area(4)
50.26548245743669
```

قطعا، این هنوز یک مثال ساده است، اما به توضیح برخی از مزایایی که متدهای استاتیک ارائه می‌دهند کمک می‌کند.

طبق آموخته‌های ما، متدهای استاتیک نمی‌توانند به حالت کلاس یا حالت نمونه دسترسی پیدا کنند، چون آرگومان cls یا self را دریافت نمی‌کنند. این محدودیت بزرگی است. همچنین سیگنالی عالی برای نشان دادن مستقل بودن این متدهای خاص از هر چیز پیرامون آن است.

در مثال فوق، مشخص است که متد circle_area() به هیچ وجه نمی‌تواند کلاس یا نمونه کلاس را اصلاح کند. (قطعا، همواره می‌توانید پیرامون آن با متغیر کلی کار کنید، اما در اینجا موضوع مهمی نیست.)

حالا، چرا این کار حائز اهمیت است؟

نام گذاری یک متد به عنوان متد استاتیک، فقط نشانه این نیست که آن متد نمی‌تواند حالت کلاس یا حالت نمونه را اصلاح کند. همانطور که مشاهده کردید، این محدودیت در زمان اجرای پایتون نیز اعمال می‌شود.

تکنیک هایی مانند این، امکان برقراری ارتباط مستقیم با بخش هایی از معماری کلاس را فراهم می‌کنند تا کار توسعه جدید به صورت طبیعی و با وجود این محدودیت‌ها انجام شود. البته، حذف این محدودیت ها کار بسیار آسانی خواهد بود. اما در عمل، آن‌ها به جلوگیری از اصلاحات تصادفی که برخلاف طرح اصلی هستند، کمک می‌کنند.

به بیانی متفاوت، استفاده از متدهای استاتیک و متدهای کلاس، روش هایی برای برقراری ارتباط با هدف برنامه‌نویس هستند و هدف را به اندازه مناسب مشخص می‌کنند تا از اشتباهات و اشکالات "فراموشی" که طرح را خراب می‌کند، اجتناب شود.

با کاربرد به میزان کم و منطقی، نوشتن برخی متدها به این صورت می‌توانند مزایای نگهداری را فراهم کنند و احتمال استفاده ناصحیح سایر برنامه‌نویس‌ها از کلاس‌های شما را کاهش دهند.

همچنین متدهای استاتیک هنگام نوشتن یونیت تست‌ها مزایایی دارند. چون متد circle_area() کاملاً مستقل از بقیه کلاس است، پس آزمایش آن بسیار ساده‌تر خواهد بود.

از طرف دیگر، این کار فرآیندهای آتی نگهداری را آسان‌تر می‌کند و پیوندی بین سبک برنامه‌نویسی شی‌گرا و رویه‌ای[1] ایجاد می‌کند.

٤-٨-٧. نکات کلیدی در هنگام کار با انواع متدها

• متدهای نمونه به نمونه کلاسی نیاز دارند و می‌توانند از طریق self به نمونه دسترسی پیدا کنند.

1 Procedural programming style

• متدهای کلاس به نمونه کلاس نیازی ندارند. آن‌ها نمی‌توانند به نمونه (self) دسترسی پیدا کنند اما از طریق cls به خود کلاس دسترسی دارند.

• متدهای استاتیک به cls یا self دسترسی ندارند. آن‌ها مانند توابع معمولی کار می کنند، اما متعلق به فضای نام کلاس هستند.

• متدهای استاتیک و کلاس با یکدیگر ارتباط برقرار می‌کنند و (تا اندازه معینی) هدف برنامه‌نویس را درباره طراحی کلاس اجراء می‌کنند. این امر می‌تواند مزایای قطعی در زمینه نگهداری از سیستم را ایجاد کند.

فصل پنجم

ساختارهای داده در پایتون

یک برنامه نویس پایتون باید بر ساختارهای داده[1] کاملاً مسلط باشد. ساختارهای داده یکی از پایه‌ای ترین اجزای برنامه‌ی شما بوده و هرکدام از آن‌ها روش خاصی را برای سازماندهی داده‌های شما فراهم می‌کنند؛ بنابراین بسته به نیاز، می‌توان از هرکدام از آن‌ها در موقعیتی مناسب استفاده نمود.

مرور برخی از مباحث پایه‌ای در پایتون، شاید در گام اول کمی خسته کننده به نظر برسد؛ اما به یاد داشته باشید که با این کار قدرت درک خود را از این زبان بالا برده‌اید و بهتر می‌توانید مفاهیم پیچیده را یاد بگیرید.

ما در پایتون ساختارهای داده‌ی مختلفی مانند لیست، دیکشنری، مجموعه (set) و ... داریم. اما پشته[2] چطور؟ آیا در پایتون پشته هم داریم؟

معمولا برخی از ساختارهای داده‌ای انتزاعی مانند پشته در پایتون مشخص نیست که به چه صورت هستند. برای مثال زبانی مانند جاوا دارای انواع مختلفی از لیست است: LinkedList یا ArrayList.

این کار باعث فهم بهتر نوع داده‌ها در جاوا می‌شود. پایتون به یک نامگذاری ساده بسنده می‌کند و همین باعث جذابیت و سادگی این زبان برنامه نویسی می‌شود. اما نقطه ضعف زمانی مشخص می‌شود که صحبت از لیست در پایتون است، منظور ما LinkedList است یا ArrayList.

ما در این فصل به بحث در مورد ساختارهای داده‌ی پایه‌ای و پیاده سازی انواع داده‌های انتزاعی[3] در پایتون می‌پردازیم. هدف ما این است تا دریابیم معادل انواع داده‌های انتزاعی در پایتون به چه صورت است و در مورد هرکدام از آن‌ها توضیحات مختصری ارایه می‌دهیم.

---

1 Data Structures
2 Stack
3 Abstract Data Types (ADTs)

۵-۱. Dictionaryها، Mapها و Hashtableها

دیکشنری‌ها در پایتون می‌توانند تعدادی آبجکت[1] در خود ذخیره کنند که هرکدام از آن‌ها با یک کلید[2] قابل دسترسی هستند.

دیکشنری‌ها معمولاً با نام‌هایی مانند maps یا hashmaps یا lookup tables نیز شناخته می‌شوند که به ما اجازه می‌دهند تا آبجکتی جدید را براساس یک کلید حذف کرده، اضافه کنیم و یا بیابیم.

بیایید یک دفترچه‌ی تلفن را در نظر بگیریم. با استفاده از یک دفترچه تلفن می‌توانیم اطلاعات مربوط به یک فرد را به سادگی براساس یک کلید مشخص (نام فرد مورد نظر) به دست آوریم.

دیکشنری‌ها به ما این اجازه را می‌دهند تا به سرعت اطلاعات مرتبط با یک کلید خاص را بدست آوریم. به طور کلی، دیکشنری یکی از پایه‌ای ترین و اساسی ترین ساختارهای داده در علوم کامپیوتر است.

۵-۱-۱. dict – ساده‌ترین نوع دیکشنری

نوع داده‌ی dict به صورت پیش فرض در پایتون پیاده سازی شده است. برای ساخت یک دیکشنری در پایتون از آکولاد استفاده می‌شود.

1 object
2 key

```
phonebook = {
    'bob': 7387,
    'alice': 3719,
    'jack': 7052,
}

squares = {x: x * x for x in range(6)}

>>> phonebook['alice']
3719

>>> squares
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

اما محدودیت‌هایی نیز در هنگام ساخت یک دیکشنری وجود دارد که تنها می‌توان از برخی از نوع داده‌ها به عنوان کلید استفاده کرد.

دیکشنری‌ها در پایتون توسط کلیدهایی ساخته می‌شوند که hashable باشند: یک آبجکت hashable یک مقدار هش دارد که در طول زمان تغییر نمی‌کند (__hash__) و قابل قیاس با سایر آبجکت‌ها می‌باشد. علاوه بر آن، آبجکت‌های hashable که با هم برابرند (__eq__) دارای مقدار هش یکسانی هستند.

داده‌های تغییر ناپذیر[1] مانند رشته‌ها یا اعداد hashable هستند و می‌توان از آن‌ها به عنوان کلید در دیکشنری استفاده نمود. همچنین از تاپل‌ها[2] نیز (هنگامی که محتویات آن‌ها از نوع hashable باشند) می‌توان به عنوان کلید در دیکشنری استفاده کرد.

در اکثر موارد نوع داده‌ی dict که پایتون آن را برای ما فراهم کرده است، پاسخگوی تمام نیازهای ما است. دیکشنری‌ها بسیار بهینه سازی شده‌اند و در قسمت‌های مختلف زبان پایتون نیز وجود دارند. به عنوان مثال هنگامی که شما در پایتون یک کلاس ایجاد می‌کنید، attributeهای آن کلاس به صورت دیکشنری در داخل شیء پایتون ذخیره می‌شوند.

1 immutable
2 tuples

هزینه‌ی استفاده از دیکشنری برای اعمالی مانند جستجو، حذف، افزودن یا بروز رسانی یک آیتم بسیار پایین است و به صورت بهینه، پیاده سازی شده است. که این هزینه برابر با O(1) است.

علاوه بر نوع داده‌ی dict، در پایتون دیکشنری‌های دیگری مانند دیکشنری مبتنی بر B-tree نیز وجود دارد که تمام آن‌ها بر پایه‌ی همین کلاس دیکشنری (dict) در پایتون هستند که در کنار تمام ویژگی‌های این کلاس، یکسری ویژگی‌های اضافه تر را نیز به همراه خود دارند. در ادامه به برخی از آن‌ها نگاهی می‌اندازیم.

۵-۱-۲. Collections.OrderedDict – دیکشنری به همراه کلیدهای دارای اولویت

این نوع از دیکشنری‌ها در پایتون، به سادگی و با نوشتن نام کلیدها و مقدار آن‌ها در پایتون ساخته می‌شوند. هر زمان کلید جدیدی به یک نوع از این دیکشنری اضافه شود، آن کلید به عنوان آیتم آخر دیکشنری در نظر گرفته می‌شود.

بنابراین درصورتی که در الگوریتم برنامه‌ی شما ترتیب کلیدهای موجود در یک دیکشنری اهمیت دارد، می‌توانید از کلاس OrderedDict استفاده کنید.

البته توجه داشته باشید که این کلاس جزء هسته‌ی اصلی زبان پایتون نیست و برای استفاده از آن باید آن را از کتابخانه‌ی collections ایمپورت کنید.

```
>>> import collections
>>> d = collections.OrderedDict(one=1, two=2,
three=3)

>>> d
OrderedDict([('one', 1), ('two', 2), ('three', 3)])

>>> d['four'] = 4
>>> d
OrderedDict([('one', 1), ('two', 2),
             ('three', 3), ('four', 4)])

>>> d.keys()
odict_keys(['one', 'two', 'three', 'four'])
```

۵-۱-۳. collections.defaultdict – بازگرداندن مقداری پیش فرض برای کلیدهای ناموجود

نوعی دیگری از دیکشنری در پایتون وجود دارد که در هنگام ساخت آن، می‌توان یک نوع داده را به آن ارسال کرد. در هنگام دستیابی به کلیدی که در این دیکشنری وجود ندارد، مقداری خالی از آن نوع داده برگشت داده می‌شود.

این نوع دیکشنری، به شما کمک می کند تا بجای استفاده از یک دیکشنری معمولی و دریافت خطاهایی مانند KeyError یا استفاده از متد get، به سادگی و بدون نگرانی با دیکشنری‌های خود کار کنید.

```
>>> from collections import defaultdict
>>> dd = defaultdict(list)

# Accessing a missing key creates it and
# initializes it using the default factory,
# i.e. list() in this example:
>>> dd['dogs'].append('Rufus')
>>> dd['dogs'].append('Kathrin')
>>> dd['dogs'].append('Mr Sniffles')

>>> dd['dogs']
['Rufus', 'Kathrin', 'Mr Sniffles']
```

۵-۱-٤. collections.ChainMap – جستجوی چندین دیکشنری در یک mapping

این نوع داده می‌تواند چندین دیکشنری را در یک نگاشت ذخیره کند. بدین صورت برای یافتن یک کلید در هرکدام از دیکشنری‌ها، می‌توانید به سادگی از این نگاشت استفاده کنید.

```
>>> from collections import ChainMap
>>> dict1 = {'one': 1, 'two': 2}
>>> dict2 = {'three': 3, 'four': 4}
>>> chain = ChainMap(dict1, dict2)

>>> chain
ChainMap({'one': 1, 'two': 2}, {'three': 3, 'four':
4})

# ChainMap searches each collection in the chain
# from left to right until it finds the key (or
fails):
>>> chain['three']
3
>>> chain['one']
1
>>> chain['missing']
KeyError: 'missing'
```

۵-۱-۵. types.MappingProxyType – ساخت دیکشنری‌های فقط خواندنی

از این نوع داده برای ایجاد دیکشنری‌هایی به صورت فقط خواندنی استفاده می‌شود. به بیانی دیگر می‌توان دیکشنری‌هایی تغییر ناپذیر (immutable) ایجاد نمود.

برای مثال، در صورتی که قصد دارید در یک کلاس یا متدی از برنامه‌ی خود یک دیکشنری را برگردانید و از تغییر آن در سایر قسمت‌های برنامه جلوگیری کنید، می‌توانید با استفاده از MappingProxyType این محدودیت را روی دیکشنری اعمال نمایید.

```
>>> from types import MappingProxyType
>>> writable = {'one': 1, 'two': 2}
>>> read_only = MappingProxyType(writable)

# The proxy is read-only:
>>> read_only['one']
1
>>> read_only['one'] = 23
TypeError:
"'mappingproxy' object does not support item
assignment"

# Updates to the original are reflected in the proxy:
>>> writable['one'] = 42
>>> read_only
mappingproxy({'one': 42, 'two': 2})
```

۵-۱-٦. نکات کلیدی در هنگام کار با دیکشنری‌ها

تمام انواع دیکشنری‌هایی که در این بخش به توضیح درباره‌ی آن‌ها پرداختیم، جزء کتابخانه‌های استاندارد در زبان پایتون هستند.

اما با تمام این توضیحات، ما به شما پیشنهاد می‌کنیم که در برنامه‌ی خود از نوع داده‌ی dict که همان دیکشنری اصلی در پایتون است استفاده کنید. این دیکشنری برای همه‌ی افرادی که در آینده کد شما را می‌خوانند قابل فهم بوده و همچنین اکثر نیازهای شما را برطرف می‌کند.

بهتر است تنها در صورتی که در برنامه‌ی خود نیاز به کاری خاص یا پیچیده دارید از سایر دیکشنری‌هایی که در این بخش نام برده شده است استفاده کنید.

۵-۲. انواع آرایه‌های پایتونی

آرایه‌ها[1] یکی از پایه‌ای ترین نوع داده در تمام زبان‌های برنامه نویسی هستند و از آن‌ها به صورت گسترده در بسیاری از برنامه‌ها و الگوریتم‌ها استفاده می‌شود.

1 Array

در این بخش نگاهی به پیاده سازی آرایه‌ها در پایتون می‌اندازیم و کتابخانه‌هایی را در زبان پایتون بررسی می‌کنیم که آرایه‌ها را به نحوی دیگر در این زبان پیاده سازی نموده‌اند. بنابراین شما با نقاط قوت و ضعف هرکدام از آن‌ها آشنا شده و می‌توانید تصمیم بگیرید که از کدام یک در برنامه‌های خود استفاده کنید. اما بهتر است ابتدا به توضیحی درباره‌ی ماهیت آرایه‌ها بپردازیم.

آرایه‌ها از تعداد ثابتی از مقادیر تشکیل شده‌اند که به هر کدام از این مقادیر می‌توان با استفاده از اندیس آن دسترسی داشت. از آنجا که آرایه‌ها اطلاعات را در بلوک‌هایی همجوار در حافظه (memory) ذخیره می‌کنند، آن‌ها را به عنوان ساختار داده‌ی همجوار در نظر می‌گیرند.

یک مثال واقعی برای آرایه‌ها یک پارکینگ است: می‌توان به پارکینگ به عنوان یک آبجکت نگاه کرد. در پارکینگ نقاطی وجود دارد که با شماره‌ای منحصر به فرد مشخص شده‌اند. این نقاط، مخصوص وسایل نقلیه بوده که ممکن است خالی یا دارای یک ماشین یا موتور باشند. اما تمام پارکینگ‌ها به یک صورت نیستند. برخی از پارکینگ‌ها تنها مخصوص نوع خاصی از وسایل نقلیه هستند. این نوع از پارکینگ‌ها را می‌توان با آرایه‌هایی مقایسه کرد که تنها یک نوع داده[1] را در خود جای داده‌اند.

از لحاظ عملکرد، دسترسی به عناصر یک آرایه با استفاده از اندیس آن بسیار سریع بوده و هزینه‌ای برابر با O(1) دارد.

آرایه‌ها در پایتون و کتابخانه‌های استاندارد آن به گونه‌های مختلفی پیاده سازی شده‌اند که در ادامه نگاهی به انواع آن‌ها خواهیم انداخت.

۵-۲-۱. list – آرایه‌های تغییرپذیر[2]

لیست در پایتون جزئی از هسته‌ی اصلی این زبان است. علی‌رغم نام آن، از لیست در پایتون به عنوان آرایه‌هایی تغییر پذیر استفاده می‌شود.

1 typed array
2 mutable

این بدان معناست که می‌توان به یک لیست در پایتون، عناصری را اضافه یا از آن حذف نمود و این لیست در پشت صحنه به طور خودکار فضای حافظه‌ای را برای این عناصر جدید اختصاص می‌دهد یا فضای حافظه‌ای را در هنگام حذف یک عنصر آزاد می‌کند.

یک لیست در پایتون می‌تواند مقدار دلخواهی از عناصر را در خود نگه دارد. در پایتون همه چیز به عنوان آبجکت در نظر گرفته می‌شود. بنابراین شما می‌توانید یک لیست بسازید که شامل انواع مختلفی از داده‌ها است. شاید بتوان به این قابلیت به عنوان نکته‌ی قوت لیست‌ها نگاه کرد؛ اما در واقع با این کار، یک لیست مجبور است فضای بیشتری از حافظه را اشغال کند.

```
>>> arr = ['one', 'two', 'three']
>>> arr[0]
'one'

# Lists have a nice repr:
>>> arr
['one', 'two', 'three']

# Lists are mutable:
>>> arr[1] = 'hello'
>>> arr
['one', 'hello', 'three']

>>> del arr[1]
>>> arr
['one', 'three']

# Lists can hold arbitrary data types:
>>> arr.append(23)
>>> arr
['one', 'three', 23]
```

۵-۲-۲. tuple – آرایه‌های تغییر ناپذیر[1]

همانند لیست‌ها، تاپل‌ها نیز جزء هسته‌ی اصلی زبان پایتون هستند. اما برخلاف لیست، تاپل‌ها تغییر ناپذیرند. بدین معنی که نمی‌توان عنصری را از یک تاپل حذف یا به آن اضافه نمود. تمام عناصر موجود در تاپل باید در هنگام ساخت آن تعریف شوند.

تاپل‌ها نیز می‌توانند انواع مختلفی از داده‌ها را درون خود نگه دارند. اما همانند لیست‌ها، این عمل باعث می‌شود تا یک تاپل فضای حافظه‌ی بیشتری اشغال کند.

```
>>> arr = 'one', 'two', 'three'
>>> arr[0]
'one'

# Tuples have a nice repr:
>>> arr
('one', 'two', 'three')

# Tuples are immutable:
>>> arr[1] = 'hello'
TypeError:
"'tuple' object does not support item assignment"

>>> del arr[1]
TypeError:
"'tuple' object doesn't support item deletion"

# Tuples can hold arbitrary data types:
# (Adding elements creates a copy of the tuple)
>>> arr + (23,)
('one', 'two', 'three', 23)
```

۵-۲-۳. array.array – آرایه‌هایی با ساختار پایه‌ای

آرایه‌هایی که از کلاس array.array در پایتون ساخته می‌شوند، تغییر پذیر بوده و رفتاری مشابه با لیست در پایتون دارند. اما تفاوت اصلی این نوع از آرایه‌ها با لیست‌ها در

1 immutable

پایتون در ذخیره سازی نوع داده‌ها است. بدین معنا که در این نوع از آرایه‌ها تنها می‌توان عناصری از یک نوع داده ذخیره نمود.

به دلیل این محدودیت موجود در array.array، این نوع از آرایه‌ها فضای کمتری را در مقایسه با لیست یا تاپل در حافظه اشغال می‌کنند.

همچنین آرایه‌ها، از متدهایی مشابه با لیست نیز پشتیبانی می‌کنند. بنابراین شما می‌توانید از این آرایه‌ها به عنوان جایگزینی برای لیست‌ها در برنامه‌های خود استفاده کنید؛ بدون اینکه نیازی به تغییر اساسی در سایر قسمت‌های برنامه‌ی خود داشته باشید.

```
>>> import array
>>> arr = array.array('f', (1.0, 1.5, 2.0, 2.5))
>>> arr[1]
1.5

# Arrays have a nice repr:
>>> arr
array('f', [1.0, 1.5, 2.0, 2.5])

# Arrays are mutable:
>>> arr[1] = 23.0
>>> arr
array('f', [1.0, 23.0, 2.0, 2.5])

>>> del arr[1]
>>> arr
array('f', [1.0, 2.0, 2.5])

>>> arr.append(42.0)
>>> arr
array('f', [1.0, 2.0, 2.5, 42.0])

# Arrays are "typed":
>>> arr[1] = 'hello'
TypeError: "must be real number, not str"
```

۵-۲-٤. str – آرایه‌هایی تغییر ناپذیر از کاراکترها

پایتون از آبجکت‌های str به منظور ذخیره سازی دنباله‌ای از کاراکترهای یونیکدِ تغییر ناپذیر استفاده می‌کند. یعنی str را می‌توان به عنوان آرایه‌ای از کاراکترها در نظر گرفت. به بیان دیگر هر کاراکتر در رشته نیز خود یک آبجکت str به طول ۱ است.

از آنجا که رشته‌ها در پایتون از نوع داده‌ی تغییر ناپذیر هستند، به منظور تغییر یک رشته باید یک کپی از آن ایجاد کرده و آن را به عنوان متغیری جدید در نظر گرفت. نزدیک ترین معادل برای عنوان «رشته‌ی قابل تغییر»، ذخیره‌ی هر کاراکتر از یک رشته به عنوان عناصری در یک لیست است.

```
>>> arr = 'abcd'
>>> arr[1]
'b'

>>> arr
'abcd'

# Strings are immutable:
>>> arr[1] = 'e'
TypeError:
"'str' object does not support item assignment"

>>> del arr[1]
TypeError:
"'str' object doesn't support item deletion"

# Strings can be unpacked into a list to
# get a mutable representation:
>>> list('abcd')
['a', 'b', 'c', 'd']
>>> ''.join(list('abcd'))
'abcd'

# Strings are recursive data structures:
>>> type('abc')
"<class 'str'>"
>>> type('abc'[0])
"<class 'str'>"
```

۵-۲-۵. bytes – آرایه‌هایی تغییر ناپذیر از بایت‌ها

آبجکت‌هایی از بایت، دنباله‌ای تغییر ناپذیر متشکل از تعداد زیادی بایت (اعداد صحیح در بازه‌ی ۰ تا ۲۵۵) هستند. از لحاظ مفهومی، آن‌ها همانند آبجکت str هستند که تنها بجای کاراکتر شامل بایت می‌باشند.

برای ساخت آبجکت‌هایی از نوع بایت، باید از سینتکسی مخصوص به خود آن استفاده نمود. البته نوعی دیگر از این آرایه‌ها با نام bytearray نیز وجود دارد که برخلاف bytes و str تغییر پذیر هستند. در بخش آینده بیشتر با این نمونه از آرایه‌ها آشنا خواهیم شد.

```
>>> arr = bytes((0, 1, 2, 3))
>>> arr[1]
1

# Bytes literals have their own syntax:
>>> arr
b'\x00\x01\x02\x03'
>>> arr = b'\x00\x01\x02\x03'

# Only valid "bytes" are allowed:
>>> bytes((0, 300))
ValueError: "bytes must be in range(0, 256)"

# Bytes are immutable:
>>> arr[1] = 23
TypeError:
"'bytes' object does not support item assignment"

>>> del arr[1]
TypeError:
"'bytes' object doesn't support item deletion"
```

۵-۲-٦. bytearray – آرایه‌هایی تغییرپذیر از بایت‌ها

این نوع از آرایه‌ها متشکل از بایت‌ها (اعداد صحیح در بازه‌ی ۰ تا ۲۵۵) بوده که تغییر پذیر هستند. bytearrayها بسیار شبیه به bytes در بخش قبل هستند با این تفاوت که شما می‌توانید این نمونه از آرایه‌ها را به راحتی تغییر دهید (عنصری را حذف یا اضافه نمایید).

همچنین این نوع از آرایه‌ها را می‌توان به آرایه‌های تغییر ناپذیر bytes تبدیل نمود. اما توجه داشته باشید که این عمل به صورت کپی کردن تمام محتویات bytearray انجام می‌گیرد که عملی بسیار کند با هزینه‌ی O(n) است.

```
>>> arr = bytearray((0, 1, 2, 3))
>>> arr[1]
1

# The bytearray repr:
>>> arr
bytearray(b'\x00\x01\x02\x03')

# Bytearrays are mutable:
>>> arr[1] = 23
>>> arr
bytearray(b'\x00\x17\x02\x03')

>>> arr[1]
23

# Bytearrays can grow and shrink in size:
>>> del arr[1]
>>> arr
bytearray(b'\x00\x02\x03')

>>> arr.append(42)
>>> arr
bytearray(b'\x00\x02\x03*')

# Bytearrays can only hold "bytes"
# (integers in the range 0 <= x <= 255)
>>> arr[1] = 'hello'
TypeError: "an integer is required"

>>> arr[1] = 300
ValueError: "byte must be in range(0, 256)"

# Bytearrays can be converted back into bytes
objects:
# (This will copy the data)
>>> bytes(arr)
b'\x00\x02\x03*'
```

۵-۲-۷. نکات کلیدی در هنگام کار با آرایه‌ها در پایتون

شما می‌توانید از انواع مختلفی از آرایه‌هایی که در این بخش نام برده شده‌اند بسته به نیاز خود در برنامه‌هایتان استفاده کنید. کتابخانه‌های دیگری به غیر از کتابخانه‌های استاندارد پایتون به منظور کار با آرایه‌ها نیز وجود دارد. برای مثال NumPy، کتابخانه‌ای بسیار قدرتمند برای کار با داده‌ها در زمینه‌ی داده کاوی و علوم کامپیوتر است که می‌توانید از آن نیز استفاده نمایید.

پیشنهاد می‌شود تا قبل از شروع به کار با آرایه‌ها در برنامه‌ی خود به نکات زیر توجه کنید:

- در صورتی که قصد دارید داده‌هایی با نوع مختلف را در یک آرایه ذخیره کنید، بهتر است از list یا tuple در پایتون استفاده کنید.
- در صورتی که تنها داده‌هایی به صورت اعداد صحیح یا اعشاری دارید و سرعت عملکرد کار با آرایه‌ها برای شما نیز مهم است، بهتر است تا از array.array در برنامه خود استفاده کنید. همچنین استفاده از کتابخانه‌هایی مانند NumPy و Pandas نیز برای کار با این نوع از داده‌ها پیشنهاد می‌شود.
- در صورتی که قصد دارید با داده‌هایی به صورت متن کار کنید، بهتر است تا آن‌ها را در متغیری به صورت str ذخیره کنید و یا اگر قصد دارید تا محتویات متن خود را تغییر دهید، در برنامه‌ی خود می‌توانید این متغیر را به صورت کاراکترهایی جداگانه در یک لیست ذخیره نمایید.
- و اگر قصد دارید داده‌هایی را به صورت بایت ذخیره کنید، بهتر است تا از bytes یا bytearray استفاده نمایید.

به طور کلی در بسیاری از موارد پیشنهاد می‌شود تا از list در برنامه‌ی خود استفاده کنید و در صورتی که در آینده نیاز به بهینه سازی کد خود داشتید، می‌توانید آن را تغییر دهید. چرا که استفاده از listها ساده تر بوده و همچنین سایر برنامه نویسان نیز با این نوع داده آشنایی کامل دارند.

۵-۳. structها و اشیاء انتقال داده[1]

در مقایسه با آرایه‌ها، ساختارهای داده‌ی رکورد دارای مقداری فیلد ثابت هستند که هر فیلد می‌تواند یک نام داشته و نوع داده‌های آن ها با یکدیگر تفاوت داشته باشد.

در این بخش، با نحوه‌ی پیاده سازی رکوردها و structها در پایتون آشنا می‌شویم. برای این کار تنها از کتابخانه‌های استاندارد پایتون استفاده خواهیم نمود.

پایتون انواع داده‌های مختلفی را برای ما فراهم آورده است که به ما این قابلیت را می‌دهد تا بتوانیم رکوردها، structها و آبجکت‌های انتقال داده را پیاده سازی نماییم. در این بخش یک نگاه کلی به تمام آن‌ها و ویژگی‌های هریک خواهیم انداخت.

۵-۳-۱. dict - آبجکت‌های داده ساده

دیکشنری‌ها در پایتون می‌توانند تعداد دلخواهی از آبجکت‌ها را در خود ذخیره نمایند که هر کدام با یک کلید مشخص می‌شوند. دیکشنری‌ها را با نام‌هایی چون map و آرایه‌های انجمنی نیز می‌شناسند. از قابلیت‌های خوب دیکشنری می‌توان به جست و جو در آن با استفاده از یک کلید با سرعت بالا اشاره نمود.

از دیکشنری‌ها در پایتون می‌توان به عنوان نوع داده‌ی رکورد یا آبجکت استفاده نمود. همچنین نحوه‌ی ساخت دیکشنری‌ها نیز در پایتون بسیار ساده و مختصر است. آبجکت‌هایی که به صورت دیکشنری ساخته می‌شوند قابل تغییر هستند و به راحتی می‌توان فیلدهایی را به یک آبجکت اضافه نمود و یا از آن حذف کرد.

1 Data Transfer Objects

```
car1 = {
    'color': 'red',
    'mileage': 3812.4,
    'automatic': True,
}
car2 = {
    'color': 'blue',
    'mileage': 40231,
    'automatic': False,
}

# Dicts have a nice repr:
>>> car2
{'color': 'blue', 'automatic': False, 'mileage':
40231}

# Get mileage:
>>> car2['mileage']
40231

# Dicts are mutable:
>>> car2['mileage'] = 12
>>> car2['windshield'] = 'broken'
>>> car2
{'windshield': 'broken', 'color': 'blue',
'automatic': False, 'mileage': 12}

# No protection against wrong field names,
# or missing/extra fields:
car3 = {
    'colr': 'green',
    'automatic': False,
    'windshield': 'broken',
}
```

۵-۳-۲. tuple – گروهی از آبجکت‌های تغییر ناپذیر

از تاپل‌ها در پایتون می‌توان برای گروه بندی تعداد دلخواهی آبجکت استفاده نمود. توجه داشته باشید که این نوع داده تغییر ناپذیر است و پس از ساخت یک تاپل نمی‌توان آن را تغییر داد.

از لحاظ عملکرد، تاپل‌ها از فضای حافظه‌ی کمتری نسبت به listها در پایتون استفاده کرده و همچنین سریع تر نیز هستند.

همانطور که در کد زیر مشاهده می‌کنید، برای ساخت یک تاپل از تنها یک آپکد[1] LOAD_CONST استفاده شده است. در صورتی که برای ساخت یک لیست با همان محتویات به عملیات بیشتری نیاز است.

```
>>> import dis
>>> dis.dis(compile("(23, 'a', 'b', 'c')", '',
'eval'))
        0 LOAD_CONST               4 ((23, 'a', 'b',
'c'))
        3 RETURN_VALUE

>>> dis.dis(compile("[23, 'a', 'b', 'c']", '',
'eval'))
        0 LOAD_CONST               0 (23)
        3 LOAD_CONST               1 ('a')
        6 LOAD_CONST               2 ('b')
        9 LOAD_CONST               3 ('c')
       12 BUILD_LIST               4
       15 RETURN_VALUE
```

البته شما نباید برای تفاوت بین این دو اهمیت زیادی قائل شوید. در برخی از پروژه‌ها تفاوت عملکرد بین این دو بسیار ناچیز است.

شاید بتوان گفت یکی از نقاط ضعف در تاپل‌ها، توانایی دسترسی به فیلدهای آن تنها با استفاده از شماره‌ی اندیس آن فیلد است. شما نمی‌توانید همانند دیکشنری به فیلدهایتان در تاپل یک نام اختصاص دهید. شاید بتوان گفت این عمل خوانایی کد شما را کمی تحت تاثیر قرار خواهد داد.

همچنین گاهی اوقات کمی سخت است که بتوان اطمینان حاصل نمود که دو تاپل دارای تعداد فیلد برابر بوده و ویژگی‌های ذخیره شده در آن دو به یک صورت است.

---

1 opcode

بنابراین پیشنهاد می‌شود تا در صورت امکان فیلدهای کمتری در یک تاپل ذخیره نمایید.

```
# Fields: color, mileage, automatic
>>> car1 = ('red', 3812.4, True)
>>> car2 = ('blue', 40231.0, False)

# Tuple instances have a nice repr:
>>> car1
('red', 3812.4, True)
>>> car2
('blue', 40231.0, False)

# Get mileage:
>>> car2[1]
40231.0

# Tuples are immutable:
>>> car2[1] = 12
TypeError:
"'tuple' object does not support item assignment"

# No protection against missing/extra fields
# or a wrong order:
>>> car3 = (3431.5, 'green', True, 'silver')
```

۵-۳-۳. نوشتن یک کلاس - کنترل بهتر و دقیق‌تر

با ساخت یک کلاس در پایتون نیز می‌توانیم اشیا داده‌ای بسازیم و مطمئن باشیم که هر آبجکت ایجاد شده از این کلاس، فیلدهای یکسانی دارد.

ساخت چنین کلاس‌هایی کار ساده‌ای است. اما به منظور پیاده سازی سایر ویژگی‌های مربوط به یک آبجکت نیازمند یک سری اعمال اضافه است. برای مثال برای ایجاد فیلدهایی جدید در یک کلاس در هنگام ساخت یک شیء، باید از متد __init__ استفاده نمود. همچنین می‌توان از متد __repr__ نیز در یک کلاس به منظور شخصی سازی نمایش آبجکت‌های کلاس استفاده کرد.

فیلدهای ذخیره شده در یک کلاس قابل تغییر هستند و همچنین به راحتی می‌توان فیلدهای جدیدی به یک کلاس اضافه نمود. البته شما می‌توانید با استفاده از دکوریتور[1] property فیلدهایی فقط خواندنی نیز ایجاد کنید.

نوشتن یک کلاس این قابلیت را به ما می‌دهد تا بتوانیم متدهای مختلفی برای آبجکت‌های یک کلاس تعریف کرده و رفتار داده‌های خود را شخصی سازی کنیم.

```
class Car:
    def __init__(self, color, mileage, automatic):
        self.color = color
        self.mileage = mileage
        self.automatic = automatic

>>> car1 = Car('red', 3812.4, True)
>>> car2 = Car('blue', 40231.0, False)

# Get the mileage:
>>> car2.mileage
40231.0

# Classes are mutable:
>>> car2.mileage = 12
>>> car2.windshield = 'broken'

# String representation is not very useful
# (must add a manually written __repr__ method):
>>> car1
<Car object at 0x1081e69e8>
```

۵-۳-٤. collections.namedtuple - اشیا داده‌ای ساده

این کلاس که در نسخه‌ی پایتون 2.6 به بعد قابل استفاده است، زیرساخت آن همان نوع داده‌ی tuple است. همانند زمانی که یک کلاس تعریف می‌کنیم، از namedtuple نیز می‌توان به منظور ایجاد فیلدهایی یکسان برای آبجکت‌های خود استفاده کنیم.

1 Decorator

namedtupleها همانند tupleها تغییر ناپذیر هستند. این بدان معناست که پس از ساخت یک نمونه از namedtuple، شما نمی‌توانید فیلدهایی را به آن اضافه کرده، تغییر دهید و یا از آن حذف نمایید.

همانطور که از اسم namedtuple مشخص است، به هر عنصر از یک namedtuple می‌توان به وسیله‌ی نام آن دسترسی پیدا کرد. طبیعتا دسترسی به یک عنصر با استفاده از نام آن، ساده تر از دسترسی به آن با کمک عدد اندیس مربوط به آن عنصر است.

namedtuple به همان اندازه‌ی tupleها حافظه اشغال می‌کند که مقدار آن بسیار کمتر از مقدار حافظه‌ی اشغال شده توسط یک کلاس در پایتون است.

```
>>> from collections import namedtuple
>>> from sys import getsizeof

>>> p1 = namedtuple('Point', 'x y z')(1, 2, 3)
>>> p2 = (1, 2, 3)

>>> getsizeof(p1)
72
>>> getsizeof(p2)
72
```

استفاده از namedtupleها به خوانایی بیشتر کد شما کمک کرده و ساختار بهتری به برنامه‌ی شما می‌دهد. چراکه می‌توان با استفاده از این نوع داده، منظور خود را در برنامه بهتر به برنامه‌نویسان دیگر منتقل کنیم.

```
>>> from collections import namedtuple

>>> Car = namedtuple('Car' , 'color mileage
automatic')
>>> car1 = Car('red', 3812.4, True)

# Instances have a nice repr:
>>> car1
Car(color='red', mileage=3812.4, automatic=True)

# Accessing fields:
>>> car1.mileage
3812.4

# Fields are immtuable:
>>> car1.mileage = 12
AttributeError: "can't set attribute"
>>> car1.windshield = 'broken'
AttributeError:
"'Car' object has no attribute 'windshield'"
```

۵-۳-۵. typing.NamedTuple – نوع داده‌ی NamedTuple بهبود یافته

از NamedTuple در پایتون نسخه ۳/۶ به بعد می‌توان استفاده کرد. این نوع داده بسیار شبیه به namedtuple است که در بخش قبل معرفی شد. اما تفاوت آن در نحوه‌ی تعریف یک تایپ و قابلیت افزودن راهنما برای نوع فیلدهای درون تایپ است.

بدون وجود mypy[1] که عملیات type checking را انجام می‌دهد، نوشتن type annotaionها اجباری نمی‌باشد. اما بدون ابزارهایی نظیر mypy نوشتن typeها می‌تواند راهنمایی برای سایر توسعه دهندگان باشد.

1 mypy-lang.org

```
>>> from typing import NamedTuple

class Car(NamedTuple):
    color: str
    mileage: float
    automatic: bool

>>> car1 = Car('red', 3812.4, True)

# Instances have a nice repr:
>>> car1
Car(color='red', mileage=3812.4, automatic=True)

# Accessing fields:
>>> car1.mileage
3812.4

# Fields are immutable:
>>> car1.mileage = 12
AttributeError: "can't set attribute"
>>> car1.windshield = 'broken'
AttributeError:
"'Car' object has no attribute 'windshield'"

# Type annotations are not enforced without
# a separate type checking tool like mypy:
>>> Car('red', 'NOT_A_FLOAT', 99)
Car(color='red', mileage='NOT_A_FLOAT', automatic=99)
```

۵-۳-۶. struct.Struct – ساختارهای مرتب شده‌ی زبان C

کلاس struct.Struct مقدار ورودی دریافت شده از طرف برنامه نویس را به اشیاء بایت در پایتون تبدیل می‌کند. با استفاده از این کلاس می‌توان داده‌های باینری در یک فایل را مدیریت نمود.

Structها از سینتکسی خاص برای تعریف نوع فیلدهای درون خود استفاده می‌کنند. برای مثال i نشان دهنده‌ی integer، ? نشان دهنده‌ی داده‌ی bool و f نشان دهنده‌ی float است.

از Structها در پایتون به ندرت به منظور نمایش اشیا داده استفاده می‌شود. این نوع داده‌ها بیشتر به منظور روشی برای تبادل فرمت داده‌ها بین زبان پایتون و C در نظر گرفته می‌شوند.

```
>>> from struct import Struct
>>> MyStruct = Struct('i?f')
>>> data = MyStruct.pack(23, False, 42.0)

# All you get is a blob of data:
>>> data
b'\x17\x00\x00\x00\x00\x00\x00\x00\x00\x00(B'

# Data blobs can be unpacked again:
>>> MyStruct.unpack(data)
(23, False, 42.0)
```

۵-۳-۷. types.SimpleNamespace – دسترسی آسان به فیلدهای یک آبجکت

این کلاس که در پایتون نسخه‌ی ۳/۳ به بعد می‌توان استفاده نمود، این قابلیت را به ما می‌دهد که با استفاده از نام فیلدها به عناصر یک آبجکت دسترسی پیدا کنیم.

SimpleNamespace ها تمام کلیدهای دسترسی به عناصر درون خود را به عنوان attributeهای خود در نظر می‌گیرند. این بدین معناست که شما می‌توانید برای دسترسی به یک فیلد بجای استفاده از obj['key'] از obj.key استفاده کنید. همچنین این نوع داده همانند دیکشنری قابل تغییر است و شما می‌توانید عناصر درون آن را به راحتی تغییر دهید.

```
>>> from types import SimpleNamespace
>>> car1 = SimpleNamespace(color='red',
                           mileage=3812.4,
                           automatic=True)

# The default repr:
>>> car1
namespace(automatic=True, color='red',
mileage=3812.4)

# Instances support attribute access and are mutable:
>>> car1.mileage = 12
>>> car1.windshield = 'broken'
>>> del car1.automatic
>>> car1
namespace(color='red', mileage=12,
windshield='broken')
```

۵-۳-۸. نکات کلیدی در هنگام کار با رکوردها در پایتون

حال باید تصمیم بگیریم که از کدام یک از موارد ذکر شده در این بخش برای ذخیره سازی اشیاء داده استفاده نماییم. همانطور که مشاهده نمودید، گزینه‌های زیادی در زبان پایتون بدین منظور پیاده سازی شده‌اند.

- در صورتی که فیلدهای شما برای ذخیره سازی کم است (۲ یا ۳ فیلد)، پیشنهاد می‌شود تا از تاپل‌ها در پایتون استفاده کنید.
- در صورتی که قصد دارید تا فیلدهای شما تغییر ناپذیر باشند، پیشنهاد می‌شود تا از تاپل‌ها یا collections.namedtuple یا typing.NamedTuple استفاده کنید.
- در صورتی که قصد دارید فیلدهای شما هرکدام یک نام مشخص داشته باشند، پیشنهاد می‌شود از collections.namedtuple یا typing.NamedTuple استفاده نمایید.

- در صورتی که قصد دارید تا همه چیز را به صورت ساده و خوانا برای تمام برنامه نویسان دیگر پیاده سازی نمایید، بهتر است تا از دیکشنری‌ها در پایتون استفاده کنید.
- در صورتی که قصد دارید متدهایی را برای اشیاء داده‌ی خود در نظر بگیرید، بهتر است یک کلاس بسازید یا از collections.namedtuple یا typing.NamedTuple برای ساخت کلاس خود ارث بری کنید.
- در صورتی که قصد دارید اشیاء داده‌ی خود را در سطح یک شبکه انتقال دهید یا آن‌ها را در دیسک ذخیره نمایید، پیشنهاد می‌شود از struct.Struct برای این کار استفاده نمایید.

به طور کلی در صورتی که به دنبال یک راه همیشگی و ساده برای ذخیره سازی اشیاء داده هستید، بهتر است از collections.namedtuple در پایتون ۲ و از typing.NamedTuple در پایتون ۳ استفاده کنید.

٥-٤. مجموعه‌ها در پایتون

در این بخش در مورد نحوه‌ی پیاده سازی مجموعه‌های تغییر پذیر و تغییر ناپذیر در پایتون صحبت خواهیم نمود. بهتر است ابتدا در مورد مجموعه‌ها در پایتون بیشتر بدانیم:

یک مجموعه (set)، اجتماعی از اشیاء بدون ترتیب در یک ساختار داده است که نمی‌توان عناصری تکراری درون آن قرار داد. معمولا از مجموعه‌ها در پایتون به منظور تست وجود یا عدم وجود یک عنصر در یک ساختار داده استفاده می‌شود. البته لازم به ذکر است که می‌توان عناصری نیز به یک مجموعه اضافه یا از آن حذف نمود. همچنین می‌توان اجتماع یا اشتراک دو مجموعه را با استفاده از این ساختار داده محاسبه نمود.

هزینه‌ی چک کردن عضویت یک عنصر در مجموعه‌ای در پایتون برابر با O(1) است. عملیات اتحاد یا اجتماع دو مجموعه و همچنین تفاوت دو مجموعه و بدست آوردن زیر مجموعه‌ای خاص در پایتون به طور میانگین هزینه‌ای برابر با O(n) دارد.

ساخت یک مجموعه در پایتون بسیار ساده است.

```
vowels = {'a', 'e', 'i', 'o', 'u'}
squares = {x * x for x in range(10)}
```

توجه داشته باشید که به منظور ساخت یک مجموعه‌ی خالی در پایتون باید از دستور set() استفاده کنید. از {} برای ایجاد یک دیکشنری خالی در پایتون استفاده می‌شود.

کتابخانه‌های استاندارد موجود در پایتون انواع مختلفی از این نوع داده را پیاده سازی کرده‌اند که در ادامه نگاهی به آن‌ها می‌اندازیم.

۵-٤-۱. set – مجموعه‌های ساده در پایتون

این نوع از مجموعه‌ها، به صورت پیش فرض در پایتون وجود داشته و می‌توان از آن‌ها استفاده نمود. این نوع داده قابل تغییر بوده و به راحتی می‌توان عناصری به آن اضافه یا از آن حذف نمود. مجموعه‌ها در پایتون در واقع همان ویژگی‌های عملکردی دیکشنری‌ها را به همراه دارند.

```
>>> vowels = {'a', 'e', 'i', 'o', 'u'}
>>> 'e' in vowels
True

>>> letters = set('alice')
>>> letters.intersection(vowels)
{'a', 'e', 'i'}

>>> vowels.add('x')
>>> vowels
{'i', 'a', 'u', 'o', 'x', 'e'}

>>> len(vowels)
6
```

۵-٤-۲. frozenset – مجموعه‌های تغییر ناپذیر

frozensetها در واقع نوعی از مجموعه‌ها در پایتون هستند که پس از ساخت، نمی‌توانیم آن را تغییر دهیم. یک از قابلیت‌های خوب frozensetها این است که از آن‌ها می‌توان به عنوان کلید یک دیکشنری یا به عنوان عنصری از یک set دیگر استفاده نمود.

```
>>> vowels = frozenset({'a', 'e', 'i', 'o', 'u'})
>>> vowels.add('p')
AttributeError:
"'frozenset' object has no attribute 'add'"

# Frozensets are hashable and can
# be used as dictionary keys:
>>> d = { frozenset({1, 2, 3}): 'hello' }
>>> d[frozenset({1, 2, 3})]
'hello'
```

۵-٤-۳. collections.Counter – مجموعه‌های چندتایی[1]

این کلاس در پایتون یک مجموعه‌ی چندتایی را پیاده سازی می‌کند که عناصر موجود در یک مجموعه می‌توانند بیش از یک عنصر داشته باشند.

با استفاده از این نوع مجموعه‌ها می‌توان وجود یا عدم وجود یک عنصر در مجموعه را بررسی کرده و همچنین می‌توان تعداد دفعات رخداد آن عنصر را در مجموعه نیز بدست آورد.

1 multisets

```
>>> from collections import Counter
>>> inventory = Counter()

>>> loot = {'sword': 1, 'bread': 3}
>>> inventory.update(loot)
>>> inventory
Counter({'bread': 3, 'sword': 1})

>>> more_loot = {'sword': 1, 'apple': 1}
>>> inventory.update(more_loot)
>>> inventory
Counter({'bread': 3, 'sword': 2, 'apple': 1})
```

توجه داشته باشید که تابع len روی این کلاس، تعداد عناصر منحصر به فرد را به شما نمایش می‌دهد. برای بدست آوردن مجموع تمام عناصر موجود در این multiset می‌توانید از تابع sum استفاده نمایید.

```
>>> len(inventory)
3 # Unique elements

>>> sum(inventory.values())
6 # Total no. of elements
```

٥-٤-٤. نکات کلیدی در هنگام کار با مجموعه‌ها در پایتون

شما می‌توانید از انوع این مجموعه‌ها در برنامه‌های خود بسته به نیاز استفاده کنید:

- در صورتی که قصد دارید تا مجموعه‌ای با قابلیت تغییر عناصر ایجاد کنید، پیشنهاد می‌شود از نوع ساختار داده‌ی پیش فرض set در پایتون استفاده کنید.
- اشیاء frozenset را می‌توانید به عنوان کلید یک دیکشنری یا عنصری از یک set بسته به نیاز خود استفاده کنید.
- از collections.Counter می‌توانید به عنوان یک ساختار داده با قابلیت بدست آوردن دفعات تکرار هر عنصر در یک مجموعه استفاده کنید.

۵-۵. پشته‌ها[1] در پایتون (LIFOs)

پشته‌ها شامل مجموعه‌ای از اشیاء هستند که به صورت last-in، first-out عمل می‌کنند. بدین معنا که آخرین عنصری که به آن افزوده می‌شود، به عنوان اولین عنصر در هنگام حذف از پشته خارج می‌شود. برخلاف آرایه و لیست، در پشته‌ها نمی‌توان به تمام عناصر آن به صورت دلخواه دسترسی داشت. به عمل افزودن و حذف در پشته push و pop می‌گویند.

برای درک بهتر این نوع ساختار داده بیاید به یک مثال در دنیای واقعی بپردازیم. فرض کنید تعداد زیادی بشقاب روی هم قرار دارند. در این حالت تنها می‌توان به بالاترین بشقاب دسترسی داشت و آن را از روی پشته‌ای از بشقاب‌ها برداشت یا بشقاب دیگری بر روی آن افزود. همچنین برای دسترسی به یکی از بشقاب‌های پایینی باید تمام بشقاب‌های روی آن را برداشت.

پشته‌ها و صف‌ها[2] بسیار شبیه به یکدیگر هستند. هر دو مجموعه به عنوان عناصری از یک ساختار داده‌ی خطی در نظر گرفته می‌شوند. در صف برخلاف پشته، مجموعه‌ای از عناصر به صورت first-in، first-out عمل می‌کنند. یعنی اولین عنصری که افزوده می‌شود، در هنگام حذف نیز به عنوان اولین عنصر از مجموعه داده خارج می‌شود.

از لحاظ عملکرد، هزینه افزودن و حذف یک عنصر در پشته هزینه‌ای برابر با O(1) دارد.

پشته‌ها در بسیاری از الگوریتم‌های مختلف در دنیای کامپیوتر نیز استفاده می‌شوند. یکی از نمونه‌های کاربرد پشته‌ها در الگوریتم DFS[3] است.

در پایتون انواع مختلفی از پشته‌ها پیاده سازی شده است که بسته به نیاز خود می توانید از ویژگی‌های هرکدام از آن‌ها در برنامه‌ی خودتان استفاده نمایید. در ادامه نگاهی به برخی از انواع پشته‌ها در پایتون می‌اندازیم.

---

1 stack
2 queue
3 depth-first search

۵-۵-۱. list – یک پشته‌ی ساده

از آنجا که لیست‌ها در پایتون از عملیات push و pop پشتیبانی می‌کنند، از آن‌ها می‌توان به عنوان ساختار داده‌ی پشته در پایتون نیز استفاده نمود.

در واقع لیست‌ها در پایتون به عنوان نوعی از ساختار داده‌ی آرایه شناخته می‌شوند. این بدان معناست که فضای حافظه‌ی اشغال شده توسط یک لیست در هنگام حذف یا افزودن عناصر به آن تغییر می‌کند.

به منظور اینکه در هنگام حذف یا افزودن یک عنصر به لیست از هزینه‌ای برابر با O(1) بهره ببریم، باید از متد append برای افزودن یک عنصر به انتهای لیست و از متد pop برای حذف یک عنصر از انتهای لیست استفاده کنیم.

```
>>> s = []
>>> s.append('eat')
>>> s.append('sleep')
>>> s.append('code')

>>> s
['eat', 'sleep', 'code']

>>> s.pop()
'code'
>>> s.pop()
'sleep'
>>> s.pop()
'eat'

>>> s.pop()
IndexError: "pop from empty list"
```

۵-۵-۲. collections.deque – پشته‌هایی سریع و قدرتمند

کلاس deque در پایتون به صورت یک پشته با دو انتها (double-ended) عمل می‌کند که می‌توان عملیات حذف و افزودن یک عنصر به این نوع پشته را از هر دو

انتهای آن با هزینه‌ی O(1) انجام داد. این قابلیت باعث می‌شود که بتوان از این کلاس هم به عنوان پشته و هم به عنوان صف در پایتون استفاده کرد.

نقطه‌ی ضعف این کلاس در دسترسی به عنصری در وسط آن است که هزینه‌ی برابر با O(n) برای ما دارد.

به طور کلی deque می‌تواند بهترین انتخاب شما برای کار با پشته‌ها در برنامه‌هایتان باشد.

```
>>> from collections import deque
>>> s = deque()
>>> s.append('eat')
>>> s.append('sleep')
>>> s.append('code')

>>> s
deque(['eat', 'sleep', 'code'])

>>> s.pop()
'code'
>>> s.pop()
'sleep'
>>> s.pop()
'eat'
>>> s.pop()
IndexError: "pop from an empty deque"
```

۵-۵-۳. queue.LifoQueue – پشته‌ای مناسب برای محاسبات موازی

این نوع از پشته در پایتون برای پردازش‌هایی که به صورت همزمان اجرا می‌شوند پیاده سازی شده است و از قابلیت قفل کردن برنامه‌ی در حال اجرا استفاده می‌کند. بدین معنا که این قابلیت را دارد تا از تعداد زیادی پردازش به منظور کار با یک پشته برای افزودن یک عنصر یا حذف یک عنصر از پشته‌ی مورد نظر استفاده کند.

بسته به نیاز شما در برنامه‌هایتان، می‌توانید از این پشته استفاده کنید.

```
>>> from queue import LifoQueue
>>> s = LifoQueue()
>>> s.put('eat')
>>> s.put('sleep')
>>> s.put('code')

>>> s
<queue.LifoQueue object at 0x108298dd8>

>>> s.get()
'code'
>>> s.get()
'sleep'
>>> s.get()
'eat'

>>> s.get_nowait()
queue.Empty

>>> s.get()
# Blocks / waits forever...
```

٥-٥-٤. نکات کلیدی در هنگام کار با پشته‌ها در پایتون

همانطور که مشاهده نمودید انواع مختلفی از پشته‌ها در پایتون پیاده سازی شده است که می‌توان از آن‌ها بسته به نیاز خود استفاده نمود.

در صورتی که قصد ندارید در برنامه‌ی خود از پردازش‌هایی به صورت همزمان و موازی استفاده کنید، باید بین list و collections.deque در پایتون یک کدام را انتخاب کنید.

لیست ها در واقع نوعی از آرایه‌ها تلقی می‌شوند که به شما این قابلیت را می‌دهد تا به عناصر مختلف در لیست بسیار سریع دسترسی داشته باشید؛ اما در هنگام افزودن یا حذف عنصری از آن نیاز به تغییر فضای حافظه‌ی آن است. البته لیست‌ها مقدار فضای حافظه‌ی بیشتری نسبت به عناصرشان اشغال می‌کنند تا برای عملیات push و pop در پشته نیاز به تغییر سایز در هر بار استفاده از این متدها نباشد. اما باید توجه داشته باشید که

برای اینکار باید از متدهای append و pop استفاده کنید تا هزینه‌ای مناسب برابر با O(1) برای عملیات خود داشته باشید.

کلاس collections.deque در پایتون به صورت یک پشته با دو جهت پیاده سازی شده است که عملیات حذف یا افزودن عنصری به آن را می‌توان از هر سمت انجام داده و هزینه‌ای برابر با O(1) خواهد داشت.

به طور کلی بهتر است برای پیاده سازی پشته‌ها در پایتون از collections.deque استفاده کنیم.

۵-٦. صف‌ها[1] در پایتون (FIFOs)

در این بخش با نحوه‌ی کار با صف‌ها در پایتون آشنا می‌شویم. اما قبل از آن بهتر است نگاهی به تعریف صف بیندازیم:

یک صف شامل مجموعه‌ای از اشیاء است که به صورت first-in، first-out عمل می‌کند. یعنی اولین شیء وارد شده به یک صف به عنوان اولین شیء در هنگام عملیات حذف از آن خارج می‌شود. برخلاف لیست‌ها یا آرایه‌ها، در صف‌ها امکان دسترسی به عناصری که در وسط صف هستند وجود ندارد.

فرض کنید صفی از برنامه‌نویسان برای ورود به کنفرانس PyCon در حال ثبت نام هستند. هر فرد جدیدی به انتهای صف اضافه می‌شود و هر فردی که ثبت نام آن به اتمام برسد از جلوی صف خارج می‌شود.

صف‌ها همانند پشته‌ها هستند و تنها تفاوت آن‌ها در نحوه‌ی حذف یک عنصر از آن‌ها است. در صف اولین عنصری که به صف اضافه شده است در هنگام حذف خارج می‌شود. اما در پشته آخرین عنصری که به آن اضافه شده است در هنگام حذف از آن خارج می‌شود.

1 Queue

از لحاظ عملکرد، عملیات حذف و افزودن عنصری به یک صف هزینه‌ای برابر با O(1) دارد و بنابراین این عملیات سریع انجام می‌شود.

از صف‌ها در بسیاری از برنامه‌ها و الگوریتم‌ها در علوم کامپیوتر استفاده می‌شود. یکی از الگوریتم‌های رایجی که از این ساختار داده استفاده می‌کند، الگوریتم BFS[1] است.

الگوریتم‌های زمان بندی در کامپیوتر معمولا از صف‌های با اولویت استفاده می‌کنند. برخلاف صف‌های عادی که عناصر را بر اساس ترتیب ورود از صف خارج می‌کند، صف‌های با اولویت در هنگام خارج کردن یک عنصر از صف به اولویت عناصر توجه می‌کند. اولویت عناصر در یک صف بر اساس کلیدی مخصوص به هر عنصر مشخص می‌شود. در بخش آینده به توضیحات بیشتری در مورد صف‌های اولویت دار می‌پردازیم.

صف‌های عادی در پایتون به صورت‌های مختلفی پیاده سازی شده‌اند. در ادامه به بررسی هرکدام خواهیم پرداخت.

۵-٦-۱. list – صف‌هایی بسیار کند

می‌توانیم از لیست‌ها در پایتون به عنوان صف در برنامه‌های خود استفاده کنیم. اما این عمل در لیست‌ها به دلیل اینکه افزودن و حذف یک عنصر از ابتدای صف نیاز به جابجایی تمام عناصر موجود در لیست دارد، بسیار کند بوده و هزینه‌ای برابر با O(n) دارد.

بنابراین، به هیچ عنوان پیشنهاد نمی‌شود که از لیست‌ها به عنوان ساختار داده‌ی صف در برنامه‌های خود استفاده کنید.

1 breadth-first search

```
>>> q = []
>>> q.append('eat')
>>> q.append('sleep')
>>> q.append('code')

>>> q
['eat', 'sleep', 'code']

# Careful: This is slow!
>>> q.pop(0)
'eat'
```

۵-۶-۲. collections.deque – صف‌هایی سریع و قدرتمند

کلاس deque صف‌هایی با دو انتها (double-ended) در پایتون هستند که به ما قابلیت افزودن و حذف عناصر را از هر دو سمت آن داده و هزینه‌ای برابر با O(1) دارد. از آن جهت که deque از هر دو انتها می‌تواند عناصری را حذف یا اضافه نماید، از این کلاس در پایتون می‌توان هم به عنوان صف و هم به عنوان پشته استفاده نمود.

نقطه‌ی ضعف این کلاس در دسترسی به عنصری در وسط آن است که هزینه‌ی برابر با O(n) برای ما دارد.

به طور کلی deque می‌تواند بهترین انتخاب شما برای کار با صف‌ها در برنامه‌هایتان باشد.

```
>>> from collections import deque
>>> q = deque()
>>> q.append('eat')
>>> q.append('sleep')
>>> q.append('code')

>>> q
deque(['eat', 'sleep', 'code'])

>>> q.popleft()
'eat'
>>> q.popleft()
'sleep'
>>> q.popleft()
'code'
>>> q.popleft()
IndexError: "pop from an empty deque"
```

۵-۶-۳. queue.Queue – صف‌هایی مناسب برای محاسبات موازی

این نوع از صف‌ها در پایتون برای پردازش‌هایی که به صورت همزمان اجرا می‌شوند پیاده سازی شده است و از قابلیت قفل کردن برنامه‌ی در حال اجرا استفاده می‌کند. بدین معنا که این قابلیت را دارد تا از تعداد زیادی پردازش به منظور کار با یک صف برای افزودن یک عنصر یا حذف یک عنصر از صف مورد نظر، استفاده کند.

این نوع از صف ها قابلیت multi producer/ multi consumer را پیاده سازی می‌کنند که برای پردازش‌های موازی بسیار مفید است. بسته به نیاز شما در برنامه‌هایتان، می‌توانید از این صف استفاده کنید.

```
>>> from queue import Queue
>>> q = Queue()
>>> q.put('eat')
>>> q.put('sleep')
>>> q.put('code')

>>> q
<queue.Queue object at 0x1070f5b38>

>>> q.get()
'eat'
>>> q.get()
'sleep'
>>> q.get()
'code'

>>> q.get_nowait()
queue.Empty

>>> q.get()
# Blocks / waits forever...
```

۵-٦-٤. multiprocessing.Queue – صف‌های مشترک

این نوع از صف‌ها به ما این قابلیت را می‌دهد تا از آن‌ها در یک برنامه به صورت موازی در workerهای مختلف استفاده کنیم. از این صف‌ها به منظور به اشتراک گذاری داده‌ها بین فرایندهای مختلف در یک برنامه نیز استفاده می‌شود. این صف می‌تواند هر نوع شیء در پایتون را ذخیره کرده و به فرایندهای دیگر منتقل نماید.

به دلیل GIL[1]، پردازش‌های موازی در CPython محبوب است. با استفاده از این صف می‌توان پردازشی را بین چندین پردازشگر در سیستم توزیع نموده تا بتوان تا حدی از محدودیت‌های GIL فارغ شد.

1 Global Intrepreter Lock

```
>>> from multiprocessing import Queue
>>> q = Queue()
>>> q.put('eat')
>>> q.put('sleep')
>>> q.put('code')

>>> q
<multiprocessing.queues.Queue object at 0x1081c12b0>

>>> q.get()
'eat'
>>> q.get()
'sleep'
>>> q.get()
'code'

>>> q.get()
# Blocks / waits forever...
```

۵-۶-۵. نکات کلیدی در هنگام کار با صف‌ها در پایتون

در پایتون انواع مختلفی از صف‌ها با ویژگی‌های متفاوت پیاده سازی شده است. به طور کلی پیشنهاد نمی‌شود از لیست به دلیل عملکرد کند آن به عنوان نوع داده‌ی صف استفاده شود. در صورتی که به دنبال صفی نیستید که در پردازش‌های موازی خود از آن استفاده کنید، بهتر است تا از collections.deque به عنوان نوع داده‌ی FIFO استفاده نمایید.

۵-۷. صف‌های با اولویت

صف‌های دارای اولویت، نوعی از صف‌ها هستند که در آن‌ها هر عنصر دارای یک کلید بوده که این کلید مشخص کننده‌ی اولویت عنصر بوده و معمولا به عنوان یک ارزش وزنی به عناصر اختصاص می‌یابد.

در واقع در این نوع از صف‌ها بجای آن‌که اولین عنصر وارد شده در هنگام فراخوانی از صف خارج گردد، عنصری با بیشترین اولویت در هنگام فراخوانی از صف خارج می‌گردد.

از این صف‌ها برای مسائل مختلف زمانبندی استفاده می‌شود که مشخص می‌کند کدام برنامه اولویت بیشتری داشته و باید زودتر مورد پردازش قرار گیرد. به طور معمول برنامه‌هایی با اولویت بالا روی سیستم، باید نسبت به برنامه‌هایی با اولویت کمتر سریعتر پردازش شوند. با ساماندهی این برنامه‌ها در صف‌های با اولویت، سیستم عامل می‌تواند برنامه‌هایی با اولویت بالا را انتخاب کرده و آن‌ها را زودتر از سایر برنامه‌ها مورد پردازش قرار دهد.

در این بخش با انواع مختلفی از صف‌های با اولویت در پایتون و چگونگی نحوه‌ی استفاده از آن‌ها در برنامه‌های خود آشنا می‌شویم.

۵-۷-۱. list - نگهداری یک صف به صورت دستی

از لیست‌ها می‌توان به عنوان یک صف به منظور کار با صف‌های دارای اولویت استفاده نمود. نقطه ضعف استفاده از این روش در هزینه‌ی افزودن یک عنصر به آن بوده که برابر با O(n) است.

همچنین عملیات مرتب سازی در این لیست نیز که باید به صورت دستی انجام شود برابر با O(n) است. شما به عنوان یک برنامه نویس حتما باید پس از افزودن عنصری جدید به این صف، عمل مرتب سازی را انجام دهید تا در روند اجرای برنامه‌ی خود به مشکل نخورید.

بنابراین بهتر است تنها در صورتی از list به عنوان یک صف اولویت دار استفاده کنید که تعداد داده‌های مورد نیاز برای ذخیره سازی درون آن کم است.

```
q = []
q.append((2, 'code'))
q.append((1, 'eat'))
q.append((3, 'sleep'))

# NOTE: Remember to re-sort every time
# a new element is inserted, or use
# bisect.insort().
q.sort(reverse=True)

while q:
    next_item = q.pop()
    print(next_item)

# Result:
# (1, 'eat')
# (2, 'code')
# (3, 'sleep')
```

۵-۷-۲. heapq – پشته‌های دودویی

این نوع از ساختار داده که پایه‌ی آن همان list در پایتون است، عملیات افزودن یا حذف کردن عناصر از درون آن با هزینه‌ی O(log n) انجام می‌شود.

استفاده از این ماژول یکی از بهترین روش ها به منظور کار با صف‌های با اولویت است. heapq عناصر موجود در خود را بطور خودکار به صورت صعودی مرتب می‌کند.

```
import heapq

q = []

heapq.heappush(q, (2, 'code'))
heapq.heappush(q, (1, 'eat'))
heapq.heappush(q, (3, 'sleep'))

while q:
    next_item = heapq.heappop(q)
    print(next_item)

# Result:
# (1, 'eat')
# (2, 'code')
# (3, 'sleep')
```

۵-۷-۳. queue.PriorityQueue – مناسب ترین نوع از صف‌های با اولویت

این نوع از صف‌ها در واقع از همان ساختار درونی heapq استفاده می‌کنند. تنها تفاوت آن با heapq در قابلیت پشتیبانی آن از وظایف همزمان به منظور کار با چند روند در برنامه است.

شما بر اساس نیاز خود می‌توانید از heapq (به صورت function-based) یا queue.PriorityQueue (به صورت class-based) در برنامه‌های خود استفاده کنید.

```
from queue import PriorityQueue

q = PriorityQueue()

q.put((2, 'code'))
q.put((1, 'eat'))
q.put((3, 'sleep'))

while not q.empty():
    next_item = q.get()
    print(next_item)

# Result:
# (1, 'eat')
# (2, 'code')
# (3, 'sleep')
```

٥-٧-٤. نکات کلیدی در هنگام کار با صف‌های با اولویت در پایتون

در پایتون انواع مختلفی از صف‌های با اولویت وجود دارد که می‌توانیم در برنامه‌های خود از آن‌ها استفاده کنیم.

پیشنهاد می‌شود در اکثر مواقع از queue.PriorityQueue در برنامه‌هایتان استفاده کرده و در صورتی که قصد ندارید برنامه‌هایی با پردازش موازی و چند نخی بنویسید از heapq استفاده نمایید.

فصل ششم

حلقه‌ها در پایتون

٦-١. نوشتن حلقه‌های پایتونی

دانستن چگونگی نوشتن حلقه‌ها یکی از اساسی ترین موارد برای برنامه نویسی در هر زبانی است. اما نحوه‌ی نوشتن حلقه‌ها در پایتون کمی با سایر زبان‌ها مانند C یا Java متفاوت است.

برای مثال در قطعه کد زیر، برنامه نویس سعی کرده است یک حلقه را همانند زبان C یا Java بنویسد.

```
my_items = ['a', 'b', 'c']

i = 0

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

حال سؤال پیش می‌آید که چرا کد بالا به صورت پایتونی نوشته نشده است؟

نکته‌ی اول اینکه برنامه نویس اندیس i را به صورت دستی ایجاد کرده و هربار در انتهای حلقه آن را افزایش می‌دهد؛ و نکته دوم: برنامه نویس برای بدست آوردن طول لیست از تابع len استفاده کرده است.

ما در پایتون می‌توانیم حلقه‌ها را به صورتی بسیار ساده تر پیاده سازی کنیم که دو عمل بالا را به صورت خودکار برای ما انجام می‌دهد. بدین ترتیب لازم نیست اندیس یک لیست را به صورت دستی ایجاد کرده و آن را مدیریت کنید. بنابراین حلقه‌ی پایتونی ما بسیار ساده تر و خواناتر می‌شود.

حال حلقه‌ی بالا را بهبود می‌بخشیم و آن را به صورت پایتونی پیاده سازی خواهیم نمود. برای این کار از تابع درونی range استفاده می‌کنیم که بصورت خودکار اندیس را برای ما مدیریت می‌کند.

```
>>> range(len(my_items))
range(0, 3)

>>> list(range(0, 3))
[0, 1, 2]
```

نتیجه‌ی حاصل شده از range یک نوع داده‌ی تغییر ناپذیر است. مزیت استفاده از range بجای list در اشغال فضای حافظه‌ی کمتر است. range مقادیر یک لیست را به صورت جداگانه ذخیره نمی‌کند و تنها به صورت یک شیء تکرار شونده کار می‌کند.

```
for i in range(len(my_items)):
    print(my_items[i])
```

حتی کد بالا را باز هم می‌توان بهبود بخشید. چرا که استفاده از range و len به منظور ایجاد یک حلقه در پایتون مناسب نیست. بنابراین بیایید بازهم کد بالا را تغییر دهیم.

همانطور که قبلا هم به آن اشاره کردیم، هر حلقه در پایتون به صورت خودکار اندیس یک لیست را مدیریت می‌کند و به صورت یک حلقه‌ی for-each عمل می‌کند. شاید بهتر باشد کد بالا را به صورت زیر بنویسیم.

```
for item in my_items:
    print(item)
```

قطعه کد بالا، مناسب ترین روش برای ایجاد یک حلقه در پایتون است. همانطور که مشاهده می‌کنید در این حلقه بدون توجه به اندیس یک لیست، می‌توان روی تمام مقادیر موجود در لیست پیمایش انجام داد. همچنین توجه داشته باشید که مقادیر موجود در لیست به همان ترتیبی که قرار دارند در حلقه مورد پیمایش قرار می‌گیرند.

شاید در یک برنامه ما نیاز به اندیس هر عنصر موجود در لیست نیز داشته باشیم. با قطعه کد بالا نمی‌توانیم به اندیس موجود دسترسی داشته باشیم.

در پایتون بدون استفاده از range و len هم می‌توان حلقه‌هایی ایجاد کرد که اندیس عناصر را نیز مدیریت می‌کند. با استفاده از تابع درونی enumerate می‌توانیم روی مقادیر و اندیس آن‌ها در یک لیست پیمایش انجام دهیم.

```
>>> for i, item in enumerate(my_items):
        print(f'{i}: {item}')

0: a
1: b
2: c
```

همانطور که مشاهده می‌کنید، در قطعه کد بالا می‌توانیم در یک حلقه به عناصر و اندیس آن‌ها دسترسی داشته باشیم. همچنین در پایتون می‌توانیم روی کلیدها و مقادیر موجود در دیکشنری نیز پیمایش انجام داده و برای آن یک حلقه ایجاد نماییم.

```
>>> emails = {
    'Bob': 'bob@example.com',
    'Alice': 'alice@example.com',
}

>>> for name, email in emails.items():
        print(f'{name} -> {email}')

'Bob -> bob@example.com'
'Alice -> alice@example.com'
```

در انتهای این بخش لازم است تا در مورد یک موضوع مهم دیگر در حلقه‌ها نیز صحبت کنیم. شاید شما قصد داشته باشید تا یک حلقه ایجاد کرده و اندازه‌ی گام حلقه (step size) را نیز بخواهید در آن مدیریت کنید. برای نمونه حلقه‌ی جاوایی زیر را در نظر بگیرید.

```
for (int i = a; i < n; i += s) {
    // ...
}
```

قطعه کد بالا به چه صورت باید در پایتون نوشته شود؟ در این مواقع باید دوباره از تابع range استفاده کنیم. این تابع می‌تواند مقادیری به عنوان نقطه‌ی شروع (start) برای حلقه، نقطه‌ی پایان (stop) و اندازه‌ی گام (step) را از برنامه نویس دریافت کند. بنابراین قطعه کد جاوایی بالا به صورت زیر به یک قطعه کد پایتون تبدیل می‌شود.

```
for i in range(start, stop, step):
    # ...
```

۶-۱-۱. نکات کلیدی در هنگام کار با حلقه‌ها در پایتون

به صورت کلی بهتر است در پایتون حلقه‌ها را به صورت قطعه کد جاوا یا C پیاده سازی نکنیم و آن‌ها را به صورت پایتونی بنویسیم. حلقه‌ها در پایتون در واقع به صورت حلقه‌های for-each عمل کرده که پردازش آن روی عناصر یک کالکشن انجام می‌شود.

۶-۲. ایجاد لیست در یک خط

یکی از ویژگی‌های خوب زبان پایتون، توانایی ایجاد یک لیست به صورت خودکار با حلقه‌ی for در یک خط است. شاید در ابتدا درک آن کمی دشوار به نظر بیاید، اما پس از مدتی متوجه خواهید شد که کار با آن‌ها بسیار ساده است.

توجه داشته باشید که آن‌ها دقیقا همان حلقه‌های for در پایتون بوده که با سینتکسی خاص در یک خط نوشته می‌شوند. برای مثال به ساخت یک لیست به صورت زیر توجه کنید.

```
>>> squares = [x * x for x in range(10)]
```

این لیست نشان دهنده‌ی مربع اعداد ۰ تا ۹ است.

```
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

در صورتی که بخواهید این لیست را با استفاده از حلقه‌ی for در پایتون بنویسید، قطعه کد آن به صورت زیر خواهد بود.

```
>>> squares = []
>>> for x in range(10):
        squares.append(x * x)
```

این قطعه کد نیز بسیار ساده است. با یک بررسی ساده و مقایسه‌ی این دو روش با یکدیگر می‌توانیم به ساختار ایجاد لیست در یک خط پی ببریم. به صورت کلی به منظور ایجاد یک لیست طبق یک قاعده‌ی مشخص، باید به صورت زیر عمل کنیم.

```
values = [expression for item in collection]
```

خط بالا، می‌تواند به صورت یک حلقه‌ی for همانند قطعه کد زیر نیز نوشته شود.

```
values = []
for item in collection:
    values.append(expression)
```

در قطعه کد بالا، ابتدا یک لیست خالی ایجاد نمودیم تا در ادامه مقادیر را به آن اضافه نماییم. سپس روی تمام اعضای مجموعه (collections) پیمایش کرده و حاصل را به لیست اضافه می‌کنیم. این یک قاعده‌ی کلی به منظور ایجاد یک لیست طبق یک الگوی مشخص است که می‌توان از آن در نوشتن برنامه‌های خود استفاده کنیم.

حال بهتر است یک قدم جلوتر رفته و شرط ها را نیز به این حلقه اضافه کنیم.

ما می‌توانیم در هنگام ایجاد لیست خود در یک خط دستوری پایتون، به آن شرط نیز اضافه نماییم تا تصمیم گیری شود کدام مقادیر به لیست افزوده شوند.

```
>>> even_squares = [x * x for x in range(10)
                    if x % 2 == 0]
```

در لیست بالا، تنها مربع اعدادی (۰ تا ۹) قرار می‌گیرند که زوج باشند. در اینجا از عملگر ٪ به منظور محاسبه‌ی باقیمانده استفاده شده است. لیست حاصل به صورت زیر خواهد بود.

```
>>> even_squares
[0, 4, 16, 36, 64]
```

دستور بالا را می‌توان به صورت یک حلقه‌ی for به همراه شرط if نیز همانند زیر نوشت.

```
even_squares = []
for x in range(10):
    if x % 2 == 0:
        even_squares.append(x * x)
```

همانند حالت قبل، برای این قطعه کد نیز می‌توان یک قاعده‌ی کلی بدست آورد. با این تفاوت که در این حالت یک بخش با عنوان شرط نیز باید به قاعده اضافه شده تا نتایج را فیلتر کند. قاعده‌ی کلی به صورت زیر خواهد بود.

```
values = [expression
          for item in collection
          if condition]
```

این دستور را می‌توان به صورت قطعه کد زیر نیز ترجمه نمود.

```
values = []
for item in collection:
    if condition:
        values.append(expression)
```

به طور کلی بهتر است لیست‌هایی که در برنامه‌های خود نیاز داریم را در یک خط ایجاد کنیم. این کار توانایی کد زنی ما را در برنامه نویسی افزایش می‌دهد.

اما علاوه بر ایجاد لیست، در پایتون به همین صورت می‌توانیم مجموعه و دیکشنری نیز ایجاد کنیم. برای ایجاد یک مجموعه به صورت زیر عمل می‌کنیم.

```
>>> { x * x for x in range(-9, 10) }
set([64, 1, 36, 0, 49, 9, 16, 81, 25, 4])
```

بر خلاف لیست‌ها در پایتون که ترتیب افزوده شدن عناصر را نیز نگه می‌دارد، در مجموعه، عناصر به صورت نامرتب وجود دارند.

همچنین برای ایجاد یک دیکشنری نیز به صورت زیر عمل می‌کنیم.

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

بهتر است در هنگام کد نویسی به این صورت، به یک نکته توجه داشته باشید. هرچه بیشتر در این روش مهارت داشته باشید و بتوانید لیست‌ها یا دیکشنری‌های پیچیده ایجاد کنید، ممکن است خواندن کد شما برای دیگران کمی سخت باشد. پس بهتر است همیشه و همه جا از این روش استفاده نکنید.

به طور کلی پیشنهاد می‌شود بیشتر از همان روش قدیمی (نوشتن حلقه‌های for) استفاده شود. چرا که نگهداری برنامه و خوانایی آن بیشتر است.

۶-۲-۱. نکات کلیدی در هنگام ساخت لیست‌هایی در یک خط

استفاده از این روش (ایجاد لیست در یک خط) بسیار مناسب بوده و کد نویسی شما را بهبود می‌بخشد. این روش در واقع راه ساده تری نسبت به نوشتن حلقه‌ی for است؛ اما در پس زمینه به همان صورت عمل می‌کند. توجه داشته باشید که استفاده‌ی زیاد از این روش، خوانایی برنامه‌ی شما را مشکل می‌کند.

۶-۳. ترفندهایی برای برش لیست‌ها و استفاده از عملگر سوشی[1]

در پایتون می‌توان یک لیست را تکه تکه نمود. به این عمل در پایتون slicing گفته می‌شود. با استفاده از این روش می‌توان تنها به بخشی از یک لیست که مورد نیاز است

1 Sushi

دسترسی پیدا نمود. برای مثال می‌توان یک لیست بزرگ را با این روش به چندین لیست کوچکتر تقسیم کرد.

برای اینکار از علامت براکت [] استفاده می‌شود که قاعده‌ی کلی آن به صورت [start:stop:step] است.

```
>>> lst = [1, 2, 3, 4, 5]
>>> lst
[1, 2, 3, 4, 5]

# lst[start:end:step]
>>> lst[1:3:1]
[2, 3]
```

در مثال بالا با نوشتن دستور [1:3:1] قصد داریم به اندیس یکم تا اندیس دوم با گام ۱ دسترسی داشته باشیم. توجه داشته باشید که عدد ۳ که در بالا استفاده شده است، شامل اندیس سوم نمی‌شود و به یک عنصر قبل از آن اشاره می‌کند.

در صورتی که گام را در قاعده‌ی بالا لحاظ نکنید، پایتون به صورت پیش فرض آن را برابر با ۱ قرار می‌دهد.

```
>>> lst[1:3]
[2, 3]
```

می‌توان تنها با استفاده از گام نیز به بخشی از لیست دسترسی داشت. به این ترفند stide نیز گفته می‌شود. برای مثال با استفاده از دستور زیر یکی در میان به عناصر لیست دسترسی پیدا می‌کنیم.

```
>>> lst[::2]
[1, 3, 5]
```

به عملگر : در پایتون که در مثال بالا استفاده شده است، سوشی (sushi) گفته می‌شود.

یکی دیگر از ترفندهایی که می‌توان روی یک لیست اجرا کرد، برعکس نمودن ترتیب عناصر در لیست به صورت زیر است.

```
>>> numbers[::-1]
[5, 4, 3, 2, 1]
```

در مثال بالا با استفاده از دو علامت :: قصد داریم به کل محتویات لیست دسترسی داشته باشیم؛ و در ادامه گام را برابر با -1 قرار دادیم تا ترتیب نمایش عناصر از آخر به اول شود. این یک ترفند جالب برای برعکس کردن ترتیب عناصر در لیست است. اما پیشنهاد می‌شود تا به منظور برعکس کردن ترتیب عناصر در یک لیست، از list.reverse یا تابع reversed استفاده کنیم.

یکی دیگر از ترفندها، حذف تمام مقادیر موجود در لیست است؛ بدون آنکه متغیری که به عنوان لیست تعریف شده است حذف شود. ممکن است در برنامه‌ای بخواهید تمام محتویات یک لیست را حذف کرده و در ادامه با محتویاتی جدید با آن کار کنید. این ترفند به شما این اجازه را می‌دهد تا این کار را به سادگی انجام دهید.

```
>>> lst = [1, 2, 3, 4, 5]
>>> del lst[:]
>>> lst
[]
```

شما همچنین می‌توانید با استفاده از متد lst.clear نیز این کار را انجام دهید. البته توجه داشته باشید که این متد تنها در پایتون ۳ قابل استفاده است.

علاوه بر این، می‌توانیم تمام عناصر یک لیست را با این روش، یا مقادیری جدید جایگزین کنیم. به مثال صفحه بعد توجه کنید.

```
>>> original_lst = lst
>>> lst[:] = [7, 8, 9]
>>> lst
[7, 8, 9]
>>> original_lst
[7, 8, 9]
>>> original_lst is lst
True
```

در قطعه کد بالا، مقادیری جدید در lst با مقادیر قبلی جایگزین شد؛ اما متغیر lst حذف نشد. به همین دلیل متغیر original_lst نیز پس از آن دستخوش تغییر شده و همان مقادیر lst را در خود نگه داشته است.

یکی دیگر از کاربردهای عملگر سوشی این است که با استفاده از آن می‌توان یک کپی از تمام محتویات (نه متغیر) یک لیست ایجاد نمود. به مثال زیر توجه کنید.

```
>>> copied_lst = lst[:]
>>> copied_lst
[7, 8, 9]
>>> copied_lst is lst
False
```

در قطعه کد بالا، تنها محتویات موجود در لیست را در متغیری جدید کپی نمودیم. به همین دلیل است که ماهیت دو متغیر lst و copied_lst با یکدیگر برابر نیستند.

۶-۳-۱. نکات کلیدی در هنگام کار با عملگر سوشی

از عملگر سوشی (:) علاوه بر انتخاب زیر مجموعه‌ای از یک لیست، می‌توان به عنوان پاک سازی، برعکس نمودن ترتیب عناصر و کپی نمودن محتویات یک لیست نیز استفاده نمود. البته توجه داشته باشید که استفاده بیش از حد از این ترفند ممکن است خوانایی کد شما را مشکل کرده و نگهداری برنامه‌ی نوشته شده سخت شود.

٦-٤. آشنایی بیشتر با اشیاء قابل پیمایش

نوشتن یک حلقه در پایتون در مقایسه با سایر زبان‌ها بسیار ساده است. این کار در پایتون همانند نوشتن یک جمله‌ی انگلیسی است.

```
numbers = [1, 2, 3]
for n in numbers:
    print(n)
```

اما این حلقه در پشت صحنه چگونه کار می‌کند؟ چگونه پایتون تشخیص می‌دهد که یک شیء قابل پیمایش با حلقه است یا خیر؟

در واقع اشیایی که دو متد __iter__ و __next__ را پشتیبانی کنند، جزیی از پروتکل iterator در پایتون حساب شده و می‌توان از آن‌ها به عنوان اشیایی قابل پیمایش با یک حلقه استفاده نمود.

در این بخش در مورد نحوه‌ی نوشتن یک کلاس با قابلیت پشتیبانی از پروتکل iterator در پایتون صحبت می‌کنیم. بدین صورت می‌توانیم درک عمیق تری نسبت به حلقه‌ها در پایتون داشته باشیم تا در آینده بتوانیم قدرت برنامه نویسی خود را افزایش دهیم.

٦-٤-١. حلقه‌ی بی نهایت

در این قسمت ابتدا یک کلاس ایجاد کرده تا با استفاده از آن پایه و اساس پروتکل iterator را درک کنیم. شاید در ابتدا درک این مفهوم کمی سخت باشد؛ اما این کار به شناخت شما در مورد پایه‌ای ترین جزییات زبان پایتون کمک خواهد نمود.

در ادامه، یک کلاس با نام Repeater ایجاد خواهیم کرد که با استفاده از این کلاس می‌توانیم روی یک رشته از متن به صورت بی‌نهایت همانند صفحه‌ی بعد یک حلقه‌ی for را اجرا کنیم.

```
repeater = Repeater('Hello')
for item in repeater:
    print(item)
```

همانطور که از نام این کلاس مشخص است، نمونه‌ی ساخته شده‌ی این کلاس به تکرار مقدار دریافتی به صورت بی‌نهایت می‌پردازد.

برای پیاده سازی این قابلیت، ابتدا کلاس Repeater را به صورت زیر تعریف می‌کنیم.

```
class Repeater:
    def __init__(self, value):
        self.value = value

    def __iter__(self):
        return RepeaterIterator(self)
```

نکته‌ی قابل توجه در پیاده سازی این کلاس، نحوه‌ی تعریف متد __iter__ در آن است. کلاس RepeaterIterator در واقع یک کلاس کمکی است که به ما کمک می‌کند تا حلقه‌ی for که در بالا نوشتیم، به درستی اجرا شود.

```
class RepeaterIterator:
    def __init__(self, source):
        self.source = source

    def __next__(self):
        return self.source.value
```

کلاس RepeaterIterator نیز یک کلاس ساده است که به صورت بالا نوشته شده است. در این کلاس نیز دو متد تعریف شده است:

۱- در متد __init__ هر نمونه از کلاس RepeaterIterator را به شیء ساخته شده از کلاس Repeater متصل می‌کنیم. بدین صورت می‌توانیم به آن شی که از کلاس Repeater ساخته شده است (source) دسترسی داشته باشیم.

۲- در متد __next__، با استفاده از source به مقدار value که در Repeater وجود دارد دسترسی پیدا می‌کنیم.

در نمونه کد بالا، دو کلاس Repeater و RepeaterIterator با همکاری یکدیگر می‌توانند از پروتکل iterator در پایتون پشتیبانی کرده و این عمل با کمک دو متد اصلی در آن‌ها، یعنی __iter__ و __next__ انجام می‌شود. در ادامه نگاهی دقیق‌تر به نحوه‌ی کار این دو متد خواهیم انداخت.

حال از این کلاس استفاده کرده و آن را در یک حلقه‌ی for مورد استفاده قرار می‌دهیم. بدین منظور یک نمونه از کلاس Repeater ساخته و رشته‌ی Hello را به عنوان مقدار ورودی این کلاس در نظر می‌گیریم.

```
>>> repeater = Repeater('Hello')
```

حال از این شیء به صورت زیر در یک حلقه‌ی for استفاده می‌کنیم.

```
>>> for item in repeater:
        print(item)
```

همانطور که مشخص است، عبارت Hello به مقدار بی‌نهایت در صفحه‌ی نمایش چاپ می‌شود.

```
Hello
Hello
Hello
Hello
Hello
...
```

بدین ترتیب ما توانستیم یک iterator در پایتون بسازیم و از آن در یک حلقه‌ی for استفاده کنیم. به منظور توقف نمایش این کلمه در مفسر، از کلیدهای ترکیبی Ctrl+C استفاده کنید.

در ادامه به بررسی دقیق تر این مثال خواهیم پرداخت تا بهتر با نحوه‌ی کارکرد دو متد __iter__ و __next__ آشنا شویم.

٦-٤-٢. حلقه‌ی for در پایتون به چه صورت کار می‌کند؟

بیایید تا درک کنیم یک حلقه‌ی for در پشت صحنه به چه صورت کار می‌کند. در مثال قبل، این حلقه به چه صورت مقدار ورودی را پشت هم چاپ می‌نمود؟

بدین منظور، بهتر است کد نوشته‌ی شده‌ی بالا را کمی با جزییات بیشتری بررسی کنیم و آن را به صورت زیر دوباره نویسی کنیم.

```
repeater = Repeater('Hello')
iterator = repeater.__iter__()
while True:
    item = iterator.__next__()
    print(item)
```

همانطور که مشاهده می‌کنید، حلقه‌ی for را به while تبدیل نمودیم. در ابتدا شیء ایجاد شده از کلاس Repeater را با فراخوانی متد __iter__ از آن، برای پروتکل iterator آماده نمودیم. پس از آن، در حلقه‌ی while متد __next__ را فراخوانی کرده و سپس مقدار حاصل را چاپ کردیم.

کلاس Repeater یک توالی از مقادیر را در اختیار ما قرار می‌دهد. این کار را نمی‌توانیم با استفاده از یک لیست در پایتون انجام دهیم. چرا که در لیست نمی‌توان تعداد نامتنهایی از مقادیر قرار داد. بدین جهت iteratorها در پایتون بسیار کارآمد هستند.

شما چه با لیست یا دیکشنری کار کنید و چه با دنباله‌ای از مقادیر به صورت بی‌نهایت (همانند مثال بالا) کار کنید، تمام آن‌ها را می‌توان به صورتی که گفته شد در یک حلقه‌ی while نیز پردازش نمایید. همانطور که مشخص است، تنها با فراخوانی دو متد __iter__ و __next__ و کار با آن‌ها می‌توان عملیات iterate را انجام داد.

شما حتی می‌توانید به صورت دستی نیز مقادیر بعدی موجود در یک لیست یا دیکشنری را به صورت زیر بدست آورید. در کد زیر ما این کار را با کلاسی که در بالا پیاده سازی نمودیم انجام دادیم و از تابع next در پایتون استفاده کردیم.

```
>>> repeater = Repeater('Hello')
>>> iterator = iter(repeater)
>>> next(iterator)
'Hello'
>>> next(iterator)
'Hello'
>>> next(iterator)
'Hello'
...
```

بدین صورت در هر بار استفاده از تابع next همان کلمه‌ی Hello برای ما نمایش داده می‌شود. همانطور که مشاهده می‌کنید، در این قطعه کد بجای استفاده از متدهای __iter__ و __next__ در کلاس Repeater، از توابع next و iter در پایتون استفاده شده است. در واقع این توابع همان کار را برای ما انجام می‌دهند و از لحاظ عملکرد تفاوتی با حالت قبل ندارند؛ فقط در این حالت، خوانایی کد ما بهبود یافته است.

این قابلیت برای برخی دیگر از توابع داخلی در پایتون نیز وجود دارد. برای مثال، استفاده از len(x) دقیقا برابر با فراخوانی متد x.__len__ است. اصولا استفاده از این توابع در پایتون (بجای فراخوانی داندر متدها به صورت مستقیم) در هنگام کد نویسی بیشتر کاربرد داشته و به خوانایی کد شما کمک می‌کند.

۶-۴-۳. پیاده سازی یک کلاس Iterator ساده تر

تا بدین جا دیدیم که مثال پیاده سازی شده برای iterator شامل دو کلاس Repeater و RepeaterIterator بود. این دو کلاس به یکدیگر مرتبط بوده و با هم

کار می‌کنند. اکثر اوقات می‌توان کارایی این دو کلاس را در یک کلاس ادغام کرد. به این صورت، پیاده سازی کلاس ما ساده تر شده و نیاز به کد نویسی کمتری دارد.

کلاس RepeaterIterator نیاز ما به متد __next__ را برای دستیابی به مقادیر جدید از iterator رفع می‌نمود. اما در واقع مکان تعریف این متد در برنامه، اهمیت چندانی ندارد. پروتکل iterator تنها به متد __iter__ نیاز دارد که در آن مقادیر را از متد __next__ دریافت کند.

در این بخش قصد داریم تا متد __next__ را مستقیما در داخل همان کلاس Repeater تعریف کنیم. بدین صورت، نیاز ما به کلاس RepeaterIterator از بین خواهد رفت و تمام اعمال را در یک کلاس پیاده سازی می‌نماییم.

کلاس ساده شده‌ی ما به صورت زیر خواهد بود.

```
class Repeater:
    def __init__(self, value):
        self.value = value

    def __iter__(self):
        return self

    def __next__(self):
        return self.value
```

این کلاس Repeater همچنان می‌تواند از پروتکل iterator در پایتون پشتیبانی کند.

```
>>> repeater = Repeater('Hello')
>>> for item in repeater:
        print(item)
Hello
Hello
Hello
...
```

در واقع در بخش‌های ابتدایی این فصل ما قصد داشتیم این عمل را با پیاده سازی دو کلاس انجام دهیم تا بهتر و دقیق تر مفاهیم را درک کنیم. اما در ادامه از این کلاس استفاده خواهیم نمود.

٦-٤-٤. چرا این حلقه به صورت بی نهایت کار می‌کند؟

تا بدین جا کلاس Repeater به صورت بی نهایت یک مقدار دریافتی را چاپ می‌کند. در واقع این عمل کارایی چندانی در پایتون ندارد. اکثر حلقه‌ها در پایتون به صورت زیر کار می‌کنند.

```
numbers = [1, 2, 3]
for n in numbers:
    print(n)
```

این کد اعداد ۱، ۲ و ۳ موجود در لیست را چاپ کرده و سپس متوقف می‌شود. یعنی دیگر نیازی به فشردن کلید ترکیبی Ctrl+C برای متوقف نمودن برنامه و قطع حلقه نیست.

در این بخش قصد داریم تا یک کلاس iterator پیاده سازی نماییم تا به صورت خودکار متوقف شود و به صورت بی نهایت یک مقدار را در خروجی نمایش ندهد. بدین منظور کلاسی با نام BoundedRepeater پیاده سازی می‌کنیم. این کلاس بسیار شبیه به همان کلاس Repeater در بخش قبل است.

اما اینکار باید به چه صورت انجام شود؟ تشخیص انتهای لیستی از مقادیر توسط پایتون به چه صورت انجام می‌شود؟

بیاید به همان قطعه کد قبل نگاهی دوباره بیندازیم. از تابع iter در پایتون برای اینکار کمک می‌گیریم تا ببینیم پایتون iterate روی یک لیست را به چه صورت مدیریت می‌کند.

```
>>> my_list = [1, 2, 3]
>>> iterator = iter(my_list)
>>> next(iterator)
1
>>> next(iterator)
2
>>> next(iterator)
3
```

در قطعه کد بالا سه مقدار حاصل در لیست را بدست آوردیم. حال اگر دوباره از تابع next استفاده کنیم، چه اتفاقی خواهد افتاد؟

```
>>> next(iterator)
StopIteration
```

همانطور که مشاهده می‌کنید، با اینکار خطای StopIteration به ما نمایش داده می‌شود. بنابراین پروتکل iterator از این خطا به منظور تشخیص انتهای یک لیست و مدیریت حلقه استفاده می‌کند. اگر بازهم از این تابع استفاده کنیم، دوباره همین خطا به ما نمایش داده می‌شود.

```
>>> next(iterator)
StopIteration
>>> next(iterator)
StopIteration
...
```

برای بازگرداندن متغیر iterator باید دوباره از لیست خود در تابع iter استفاده کنیم تا اینکه بتوانیم تابع next را فراخوانی نماییم.

حال بیایید کلاس BoundedRepeater خود را بنویسیم. این کلاس علاوه بر دریافت مقدار از ورودی، تعداد دفعات تکرار را نیز به عنوان ورودی دریافت می‌کند.

```
class BoundedRepeater:
    def __init__(self, value, max_repeats):
        self.value = value
        self.max_repeats = max_repeats
        self.count = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.count >= self.max_repeats:
            raise StopIteration
        self.count += 1
        return self.value
```

در کلاس بالا دفعات تکرار با پارامتر max_requests مدیریت می‌شود. برای استفاده از این کلاس به صورت زیر می‌توانیم عمل کنیم.

```
>>> repeater = BoundedRepeater('Hello', 3)
>>> for item in repeater:
        print(item)

Hello
Hello
Hello
```

اگر بخواهیم کد بالا را با استفاده while و توابع درونی iter و next در پایتون پیاده سازی نماییم، می‌توانیم این قطعه کد را به صورت زیر بازنویسی کنیم.

```
repeater = BoundedRepeater('Hello', 3)
iterator = iter(repeater)
while True:
    try:
        item = next(iterator)
    except StopIteration:
        break
    print(item)
```

در قطعه کد بالا، در هر بار خطای StopIteration به منظور تشخیص انتهای حلقه بررسی می‌شود.

البته پیشنهاد می‌شود بیشتر از حلقه‌های for بجای while استفاده کنید. چرا که اینکار به خوانایی کد شما کمک می‌کند.

٦-٤-٥. نکات کلیدی در هنگام کار با Iteratorها

از پروتکل Iterator به منظور کنترل حلقه‌های for در پایتون استفاده می‌شود. برای این‌که کلاسی از این پروتکل پشتیبانی کند، باید در کلاس مورد نظر دو متد __iter__ و __next__ را تعریف کنیم. با استفاده از این دو متد می‌توانیم رفتار کلاس مورد نظر خود را برای پروتکل iterator به صورت دلخواه پیاده سازی کنیم.

البته توجه داشته باشید که این روش، تنها روش موجود برای ساخت اشیاء قابل پیمایش در پایتون نیست. شما می‌توانید از مولدها نیز برای این‌کار استفاده کنید.

٦-٥. مولدها[1] در پایتون

تا بدین جای فصل در مورد نحوه‌ی ساخت کلاس‌هایی با قابلیت پشتیبانی از پروتکل iterator صحبت کردیم. اگر توجه کرده باشید، ایجاد چنین کلاس‌هایی نیاز به کمی کدنویسی تکراری و خسته کننده دارد. اکثر برنامه نویسان علاقه‌مند به نوشتن کدهای تکراری نیستند.

همانطور که می‌دانید iteratorها در پایتون بسیار کاربردی بوده که با استفاده از آن‌ها می‌توانید با حلقه‌های for کار کنید. در این بخش یک روش ساده تر برای ایجاد یک iterator را بررسی می‌کنیم. این روش نسبت به روش قبل بسیار ساده تر و سریع تر بوده و نیاز به کدنویسی کمتری دارد. در این بخش از generatorها و yield در پایتون استفاده خواهیم کرد.

1 Generators

۶-۵-۱. ایجاد مولدهای بی‌نهایت

در این بخش قصد داریم نمونه کد کلاس Repeater در بخش قبل را به نحوی دیگر پیاده سازی کنیم. اگر به یاد داشته باشید، کلاس Repeater به صورت زیر پیاده سازی شده بود.

```
class Repeater:
    def __init__(self, value):
        self.value = value

    def __iter__(self):
        return self

    def __next__(self):
        return self.value
```

بدیهی است که برای ایجاد یک Iterator نوشتن این حجم از کد کمی بیش از اندازه است. در این جا به مولدها در پایتون نیاز پیدا می‌کنیم. پیاده سازی کلاس بالا با استفاده از generator به صورت زیر خواهد بود.

```
def repeater(value):
    while True:
        yield value
```

همانطور که مشاهده می‌کنید از یک کلاس ۷خطی به یک قطعه کد ۳خطی رسیدیم. در واقع مولدها همانند تابع در پایتون عمل می‌کنند. با این تفاوت که بجای استفاده از return برای بازگرداندن یک مقدار، از yield استفاده شده است.

حال بیاید از این مولد در حلقه‌ی for استفاده کنیم.

```
>>> for x in repeater('Hi'):
        print(x)
'Hi'
'Hi'
'Hi'
'Hi'
'Hi'
...
```

این مولد همانند کلاس Repeater عمل می‌کند و به همان صورت نتایج را در خروجی چاپ می‌کند. (توجه داشته باشید که برای توقف باید از Ctrl+C استفاده کنید.)

اما یک مولد در پایتون به چه صورت کار می‌کند؟ مولدها از لحاظ ظاهری همانند یک تابع نوشته می‌شوند اما عملکرد آن‌ها متفاوت است. هنگامی که یک مولد صدا زده می‌شود، مقداری به سمت ما برگردانده نمی‌شود. بلکه یک شیء از مولد مورد نظر ساخته می‌شود.

```
>>> repeater('Hey')
<generator object repeater at 0x107bcdbf8>
```

برای بدست آوردن مقداری که yield می‌شود باید از تابع next در پایتون (همانند آنچه در بخش قبل دیدیم) استفاده کنیم.

```
>>> generator_obj = repeater('Hey')
>>> next(generator_obj)
'Hey'
```

در واقع با استفاده از yield، مولد ما متوقف شده و مقداری را برگردانده و دوباره به کار خود ادامه می‌دهد.

```
def repeater(value):
    while True:
        yield value
```

هنگامی که در یک تابع از return استفاده شده و یک مقدار برگشت داده می‌شود، کنترل برنامه بطور دائم به صدا زننده‌ی تابع منتقل می‌شود. اما هنگامی که از yield استفاده می‌شود، کنترل برنامه بطور موقت به صدا زننده‌ی مولد منتقل می‌گردد. اما این به چه معناست؟

درحالی که return وضعیت یک تابع را پس از برگشت دادن یک مقدار از بین می‌برد، yield یک مولد را به حالت تعلیق در آورده و وضعیت آن را در خود نگه می‌دارد. به عبارت دیگر متغیرهای محلی در مولد و وضعیت اجرای برنامه در آن به طور موقت از بین رفته و به بیرون درز پیدا نمی‌کند. روند اجرا می‌تواند با فراخوانی دوباره‌ی تابع next روی مولد ادامه یابد.

```
>>> iterator = repeater('Hi')
>>> next(iterator)
'Hi'
>>> next(iterator)
'Hi'
>>> next(iterator)
'Hi'
```

این قابلیت باعث می‌شود تا مولدها بطور کامل بتوانند از پروتکل iterator در پایتون پشتیبانی کنند. به همین دلیل پیشنهاد می‌شود تا بجای نوشتن کلاس‌هایی طولانی، از مولد ها بدین منظور استفاده کنید.

٦-٥-٢. پیاده سازی مولدهایی که بطور خودکار متوقف می‌شوند

در بخش قبل یک مولد پیاده سازی نمودیم که به صورت بی نهایت یک مقدار را در خروجی چاپ می‌نمود. در ادامه قصد داریم مولدی بنویسیم که پس از مدتی چاپ در خروجی را متوقف کند.

اگر به یاد داشته باشید، زمانی که این کار را با کمک نوشتن یک کلاس پیاده سازی نمودیم، از خطای StopIteration به منظور متوقف کردن روند اجرای حلقه استفاده

کردیم. در واقع در مولدها نیز همین عمل در پشت صحنه رخ می‌دهد، اما نیازی نیست تا ما آن را در کد خود به صورت دستی بنویسیم. هنگامی که مقداری با استفاده از yield از یک مولد برگردانده نشود، بطور خودکار روند آن متوقف می‌شود.

```
def repeat_three_times(value):
    yield value
    yield value
    yield value
```

در این مولد از while استفاده نشده و به صورت بسیار ساده تنها سه بار از yield استفاده شده است. پس از فراخوانی مولد، هر کدام از yieldها طبق روال مقداری را به صدا زننده برمی‌گردانند. هنگامی که به انتهای مولد برسیم، روند اجرای آن بطور خودکار متوقف خواهد شد.

```
>>> for x in repeat_three_times('Hey there'):
        print(x)
'Hey there'
'Hey there'
'Hey there'
```

این مولد پس از ۳بار نمایش مقدار دریافتی متوقف خواهد شد. این عمل در پشت صحنه با همان خطای StopIteration کنترل می‌شود. برای اطمینان به قطعه کد زیر نگاهی می‌اندازیم.

```
>>> iterator = repeat_three_times('Hey there')
>>> next(iterator)
'Hey there'
>>> next(iterator)
'Hey there'
>>> next(iterator)
'Hey there'
>>> next(iterator)
StopIteration
>>> next(iterator)
StopIteration
```

پس از ۳ بار فراخوانی تابع next، دیگر مقداری از مولد yield نخواهد شد و پس از آن خطای StopIteration به ما نمایش داده می‌شود.

بیایید دوباره نگاهی به کلاس BoundRepeater در بخش قبل بیندازیم. این کلاس علاوه بر دریافت یک مقدار به منظور نمایش، تعداد دفعات تکرار آن را نیز دریافت می‌نمود.

```
class BoundedRepeater:
    def __init__(self, value, max_repeats):
        self.value = value
        self.max_repeats = max_repeats
        self.count = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.count >= self.max_repeats:
            raise StopIteration
        self.count += 1
        return self.value
```

اگر بخواهیم این کلاس را با استفاده از یک مولد پیاده سازی کنیم، می‌توانیم کد آن را به صورت زیر بنویسیم.

```
def bounded_repeater(value, max_repeats):
    count = 0
    while True:
        if count >= max_repeats:
            return
        count += 1
        yield value
```

در قطعه کد بالا به صورت عمدی از return در حلقه‌ی while خود استفاده کردیم تا پس از تعداد max_repeats خطای StopIteration را دریافت کنیم. در ادامه این

مولد را به صورت ساده تر پیاده سازی می‌کنیم. اما ابتدا بیایید نگاهی به عملکرد این برنامه بیندازیم.

```
>>> for x in bounded_repeater('Hi', 4):
        print(x)
'Hi'
'Hi'
'Hi'
'Hi'
```

بدین صورت یک مولد ایجاد نمودیم که با دریافت تعداد تکرار، یک مقدار را نمایش می‌دهد. برای نمایش مقدار دریافتی از yield استفاده شده است و پس از تعداد max_request از return استفاده شده است که دیگر چیزی نمایش داده نشده و حلقه متوقف خواهد شد.

همانطور که گفته بودیم، قصد داریم تا این مولد را به صورتی ساده تر نیز پیاده سازی کنیم. می‌دانید که پایتون به صورت ضمنی عبارت return None را در انتهای هر تابع قرار می‌دهد. ما از این قابلیت استفاده کرده و مولد خود را به صورت زیر می‌نویسیم.

```
def bounded_repeater(value, max_repeats):
    for i in range(max_repeats):
        yield value
```

قطعه کد بالا همانند مولد قبلی عمل می‌کند. ما از کلاس ۱۲خطی BoundedRepeater به یک مولد ۳خطی رسیدیم که دقیقا همان قابلیت را دارد.

استفاده از مولدها در برنامه نویسی، به کد نویسی شما کمک زیادی می‌کند و با استفاده از آن‌ها می‌توانید کدهایی ساده تر و تمیز تر پیاده سازی کنید.

٦-٥-٣. نکات کلیدی در هنگام کار با مولدها در پایتون

از مولدها می‌توان به عنوان Iterator در پایتون استفاده نمود. آن‌ها به ما کمک می‌کنند که دیگر نیازی به نوشتن کلاس‌هایی طولانی برای پشتیبانی از پروتکل Iterator نداشته باشیم.

با استفاده از yield می‌توان روند اجرای برنامه را به صورت موقت در مولد متوقف کرده و مقداری را به صدا زننده بازگرداند. پس از آن که دیگر عبارت yield در یک مولد وجود نداشته باشد، خطای StopIteration به صورت خودکار رخ می‌دهد که این به معنای انتهای کار حلقه خواهد بود.

٦-٦. آشنایی بیشتر با مولدها

همانطور که در بخش قبل مشاهده کردید، دریافتیم که کلاس‌ها و مولدهایی که از پروتکل iterator پشتیبانی می‌کنند، از یک الگوی طراحی[1] پیروی می‌کنند. استفاده از مولدها باعث می‌شود تا نیاز به کدنویسی کاهش یابد و سریع تر به هدف خود برسیم.

از آنجا که بسیاری از برنامه نویسان از این الگوی طراحی پیروی می‌کنند، خالق زبان‌های برنامه نویسی به این فکر افتادند تا راه‌هایی ساده تر و سریع تر برای پیاده سازی همچین قابلیتی برای برنامه نویسان به وجود آورند. به همین دلیل است که زبان‌های برنامه نویسی در طول زمان بهتر می‌شوند و برنامه نویسان نیز باید همیشه بروز باشند تا بتوانند از امکانات مفید زبان برنامه نویسی خود استفاده کنند.

در بخش قبل با نحوه‌ی پیاده سازی مولدها آشنا شدیم. در این بخش یک گام بیشتر رو به جلو برداشته و پیاده سازی قبل را ساده تر می‌نماییم. این عبارات دقیقا همانند پیاده سازی یک لیست در یک خط است (که در بخش‌های قبل با آن آشنا شدیم)؛ تنها با این تفاوت که بجای استفاده از براکت [] از پرانتز () استفاده می‌کنیم. در ادامه نحوه‌ی پیاده سازی مولد در یک خط نشان داده شده است.

1 Design Pattern

```
iterator = ('Hello' for i in range(3))
```

این عبارت تک خطی دقیقا همان کاری را برای ما انجام می‌دهد که در مولد bounded_repeater پیاده سازی کرده بودیم. در زیر برای یادآوری این متد را دوباره می‌توانید ببینید.

```
def bounded_repeater(value, max_repeats):
    for i in range(max_repeats):
        yield value

iterator = bounded_repeater('Hello', 3)
```

همانطور که مشاهده می‌کنید توانستیم همین کار را تنها با نوشتن یک خط کد انجام دهیم. در قطعه کد زیر از مولدی که با یک خط ایجاد کرده بودیم استفاده می‌نماییم تا مطمئن شویم به درستی کار می‌کند.

```
>>> iterator = ('Hello' for i in range(3))
>>> for x in iterator:
        print(x)
'Hello'
'Hello'
'Hello'
```

البته توجه داشته باشید، زمانی که یک مولد بدین صورت ساخته می‌شود، پس از یکبار استفاده از آن، دیگر نمی‌توان از آن استفاده کرد. بنابراین در برخی موارد بهتر است تا مولدها را به همان صورت قبل بطور تابعی یا به صورت کلاس پیاده سازی نماییم.

۶-۶-۱. تفاوت ساخت مولد و ساخت لیست در یک خط

مطمئنا تا بدین جا توجه کرده‌اید که ساخت مولد در یک خط بسیار شبیه به ایجاد یک لیست است. در قطعه کد صفحه‌ی بعد به این دو در کنار یکدیگر نگاهی می‌اندازیم.

```
>>> listcomp = ['Hello' for i in range(3)]
>>> genexpr = ('Hello' for i in range(3))
```

برخلاف لیست، ساخت یک مولد بدین صورت اشیایی از عناصر به صورت لیست نمی‌سازد و تنها مقادیری را به صورت لحظه‌ای و یکبار مصرف ایجاد می‌کند (همانند مولدهایی که به صورت تابع یا کلاس در بخش‌های قبل می‌ساختیم).

در صورتی که بخواهیم محتویات این متغیرها را مشاهده کنیم، خروجی زیر به ما نمایش داده می‌شود.

```
>>> listcomp
['Hello', 'Hello', 'Hello']
>>> genexpr
<generator object <genexpr> at 0x1036c3200>
```

همانند بخش قبل، برای دسترسی به مقادیر این مولد باید از تابع next در پایتون استفاده کنیم.

```
>>> next(genexpr)
'Hello'
>>> next(genexpr)
'Hello'
>>> next(genexpr)
'Hello'
>>> next(genexpr)
StopIteration
```

همچنین می‌توانید از تابع list روی یک مولد استفاده نمود تا آن را به یک لیست تبدیل کند. به مثال زیر توجه کنید.

```
>>> genexpr = ('Hello' for i in range(3))
>>> list(genexpr)
['Hello', 'Hello', 'Hello']
```

به صورت بالا می‌توان به راحتی یک مولد را به لیست تبدیل نمود. البته پیشنهاد می‌شود برای ساخت یک لیست در یک خط از همان روش قبل استفاده کنید. اگر

بخواهیم به طور کلی به الگوی چگونگی ساخت یک مولد در یک خط دست یابیم، با کمی بررسی کدهای بالا می‌توان الگویی همانند زیر نوشت.

```
genexpr = (expression for item in collection)
```

خط بالا را می‌توان همانند یک تابع نیز برای ایجاد مولد به صورت زیر نوشت که این دو دقیقا به یک صورت عمل می‌کنند.

```
def generator():
    for item in collection:
        yield expression
```

بدین ترتیب می‌توان هر مولدی که به صورت تابع نوشته شده است را با کمک الگوی بالا، در یک خط پیاده سازی نمود.

۶-۶-۲. فیلتر کردن مقادیر در هنگام ساخت مولد

ما می‌توانیم با استفاده از الگوی بالا، خاصیت شرط ها را نیز در هنگام ساخت مولد نیز اعمال کنیم. به مثال زیر توجه کنید.

```
>>> even_squares = (x * x for x in range(10)
                    if x % 2 == 0)
```

مولد بالا مربع تمام اعداد زوج بین ۰ تا ۹ را شامل می‌شود. همانطور که مشاهده می‌کنید، در شرط این مولد خاصیت زوج بودن اعداد مورد نظر بررسی شده و تنها از آن‌ها استفاده می‌شود.

```
>>> for x in even_squares:
        print(x)
0 4
16
36
64
```

حال می‌توانیم الگوی قبل را بروز رسانی کرده و خاصیت بررسی شرط و فیلتر کردن مقادیر را نیز در آن لحاظ کنیم.

```
genexpr = (expression for item in collection
           if condition)
```

الگوی بالا دقیقا همانند الگوی ساخت یک مولد به صورت تابع است. همانند حالت قبل، این دو از لحاظ عملکرد به یک صورت هستند.

```
def generator():
    for item in collection:
        if condition:
            yield expression
```

۶-۶-۳. عبارات مولد بر خط[1]

از چنین مولدهایی می‌توان به همراه سایر عبارات نیز در یک خط استفاده نمود. برای مثال می‌توانیم یک مولد را در یک خط به همراه حلقه‌ی for نوشته تا مقادیر آن را بلافاصله در خروجی نمایش دهیم.

```
for x in ('Bom dia' for i in range(3)):
    print(x)
```

همچنین اگر بخواهیم مقادیر یک مولد را به عنوان آرگومان به یک تابع دهیم، می‌توانیم از نوشتن پرانتز برای آن صرف نظر کنیم تا کد ما زیبا تر و ساده تر شود. به مثال زیر توجه کنید:

```
>>> sum((x * 2 for x in range(10)))
90

# Versus:

>>> sum(x * 2 for x in range(10))
90
```

1 In-line Generator Experssions

بدین صورت کد نویسی ما ساده تر می‌شود. توجه داشته باشید که مولد ها فضای حافظه‌ی بسیار کمی اشغال می‌کنند. چرا که مقادیر درون آن پس از یکبار استفاده بلافاصله از بین می‌روند. بدین ترتیب برای بسیاری از برنامه‌های که می‌خواهیم بنویسیم، استفاده از مولدها می‌تواند مفید واقع شود.

٦-٦-٤. عدم استفاده‌ی بیش از حد برای ساخت مولد در یک خط

مولدهایی که در یک خط نوشته می‌شوند، ممکن است گاهی بسیار پیچیده شوند. برای مثال می‌توان چندین حلقه‌ی تو در تو و چندین شرط را به صورتی که در ادامه می‌بینید برای ساخت یک مولد پیاده سازی نمود.

```
(expr for x in xs if cond1
      for y in ys if cond2
      ...
      for z in zs if condN)
```

عبارت بالا را می‌توان به صورت یک تابع مولدی همانند زیر نیز پیاده سازی نمود.

```
for x in xs:
    if cond1:
        for y in ys:
            if cond2:
                ...
                for z in zs:
                    if condN:
                        yield expr
```

پیشنهاد می‌شود به هیچ وجه چنین حلقه‌هایی را به صورت یک عبارت تک خطی ننویسید. چرا که درک برنامه‌ی شما برای سایر برنامه نویسان بسیار سخت بوده و نگهداری چنین کدهایی بسیار دشوار است.

همیشه سعی کنید تا از عبارات تک خطی برای موارد ساده استفاده کنید. بهتر است در صورتی که نیاز است تا دو حلقه‌ی تو در تو در یک مولد ایجاد کنید، از نوشتن عبارت تک خطی برای ساخت مولد پرهیز کرده و بجای آن از روش تابعی استفاده

نمایید. بنابراین بهتر است تا برای ساخت مولدهای پیچیده از روش تابعی یا ساخت یک کلاس بدین منظور کمک بگیرید.

در صورتی که قصد دارید تا از شرط‌های پیچیده در کنار حلقه‌ها برای ساخت یک مولد استفاده کنید، بهتر است تا آن را به چندین زیر مولد کوچکتر تقسیم کرده و سپس آن‌ها را به یکدیگر پیوند دهید. در بخش آینده با این روش بیشتر آشنا خواهیم شد.

٦-٦-٥. نکات کلیدی در هنگام ایجاد مولدها در یک خط

مولدها برخلاف لیست‌ها مقادیری را تنها برای یکبار تولید می‌کنند و پس از فراخوانی آن‌ها، دیگر نمی‌توان به آن مقادیر دسترسی داشت. بدین جهت است که از حافظه‌ی کمی در سیستم اشغال می‌کنند.

هنگامی که از یک مولد در برنامه استفاده شود، دیگر نمی‌توان به مقادیر آن دسترسی داشت و نیاز است تا دوباره آن مولد تولید شود.

مولدهایی که در یک خط نوشته می‌شوند، یک روش ساده‌تر برای ایجاد یک مولد است. این مولدها از لحاظ عملکرد دقیقا برابر با معادلشان بوده که به صورت یک کلاس یا به صورت تابعی نوشته می‌شوند.

٦-٧. ساخت مولدها به صورت زنجیره‌ای[1]

در این بخش قصد داریم تا یکی از دیگر از ویژگی‌های iteratorها را بررسی کنیم. با به هم پیوستن چندین iterator ساده در پایتون به صورت زنجیره‌ای، می‌توان یک مولد پیچیده تر و طولانی تر ایجاد نمود.

در این بخش ابتدا این روش را با نوشتن مولدهای تابعی پیاده سازی می‌نماییم تا روش کار این ویژگی را بهتر درک کنیم و در ادامه این مولدها را به صورت تک خطی می‌نویسیم.

1 Chain

در قطعه کد زیر دنباله‌ای از اعداد را از ۱ تا ۸ در یک مولد تابعی ایجاد نمودیم.

```
def integers():
    for i in range(1, 9):
        yield i
```

حال از این مولد استفاده کرده و مقادیر خروجی آن را با استفاده از تابع list در پایتون، به لیستی از مقادیر تبدیل می‌کنیم.

```
>>> chain = integers()
>>> list(chain)
[1, 2, 3, 4, 5, 6, 7, 8]
```

در ادامه قصد داریم تا مولدی دیگر بسازیم که این مولد دنباله‌ای از اعداد را گرفته و مربع آن‌ها را به ما برمی‌گرداند.

```
def squared(seq):
    for i in seq:
        yield i * i
```

این دو مولد می‌توانند به یکدیگر به صورت زنجیره‌ای وصل شده و دنباله‌ای از پردازش‌ها را انجام دهند و یک خروجی واحد را به ما نمایش دهند. تنها کافی است تا مولد integers را به squared پاس دهیم و خروجی آن‌ها را به صورت یک لیست مشاهده نماییم. در قطعه کد زیر برای صحت عملکرد این دو مولد به صورت زنجیره‌ای این کار را انجام داده‌ایم.

```
>>> chain = squared(integers())
>>> list(chain)
[1, 4, 9, 16, 25, 36, 49, 64]
```

همانطور که مشاهده می‌کنید، لیست نمایش داده شده برابر با مربع اعداد ۱ تا ۸ است. در واقع ما یک خط لوله[1] برای پردازش پشت هم دو مولد ایجاد کرده‌ایم. مفهوم خط

1 pipeline

لوله در اینجا، در سیستم‌های Unix نیز بکار برده می‌شود که چندین پردازش را به یکدیگر مرتبط کرده و خروجی هرکدام را به عنوان ورودی دستور بعد در نظر می‌گیرند.

در ادامه قصد داریم تا یک مولد دیگر به خط لوله‌ی پردازش‌های خود اضافه کنیم. این مولد تمام مقادیر موجود در یک دنباله را منفی می‌نماید.

```
def negated(seq):
    for i in seq:
        yield -i
```

حال اگر این سه مولد را به صورت زنجیره‌ای به یکدیگر متصل کنیم، خروجی زیر را خواهیم داشت.

```
>>> chain = negated(squared(integers()))
>>> list(chain)
[-1, -4, -9, -16, -25, -36, -49, -64]
```

توجه داشته باشید که تنها یک مقدار در هر لحظه برای پردازش می‌تواند در این خط لوله قرار گیرد و هیچ داده‌ای بافر نمی‌شود. در مرحله‌ی اول مولد integers یک مقدار را تولید می‌کند (برای مثال عدد ۳)؛ سپس مولد squared فعال شده و این عدد را پردازش کرده و مربع آن یعنی ۹ را آماده می‌نماید؛ در مرحله‌ی آخر عدد ۹ حاصل به مولد negated برای پردازش فرستاده شده و عدد ۹- بازگردانده می‌شود.

شما می‌توانید با استفاده از این روش مولدهای پیچیده تری را با استفاده از چندین مولد کوچک و ساده پیاده سازی کنید. استفاده از این روش بسیار مناسب است. چرا که در آینده برای تغییر بخشی از برنامه‌ی خود تنها کافی است همان مولد کوچک مورد نظر را تغییر دهید و از پیچیدگی‌های بیشتر دوری نمایید.

ما می‌توانیم تمام این مولدها را به صورت تک خطی نیز پیاده سازی کنیم که دقیقا همان خروجی را برای ما نمایش دهد. در قطعه کد زیر نحوه‌ی پیاده سازی این سه مولد را می‌توانید مشاهده کنید.

```
integers = range(8)
squared = (i * i for i in integers)
negated = (-i for i in squared)
```

توجه داشته باشید که تمام کدهای بالا تنها در همین سه خط خلاصه شده‌اند. اگر بخواهیم خروجی مولد negated را مشاهده کنیم، می‌توانیم از تابع list استفاده نماییم.

```
>>> negated
<generator object <genexpr> at 0x1098bcb48>
>>> list(negated)
[0, -1, -4, -9, -16, -25, -36, -49]
```

تنها نقطه ضعف استفاده از این روش این است که آن‌ها را نمی‌توان با آرگومان‌هایی دیگر استفاده نمود. پس از یکبار استفاده از این مولدها، آن‌ها کاملاً از بین رفته و برای استفاده‌ی مجدد از آن‌ها باید دوباره این مولدها را ایجاد نماییم. بنابراین توجه داشته باشید که بسته به نیاز خود در جای مناسب از چنین مولدهایی استفاده کنید.

۶-۷-۱. نکات کلیدی در هنگام ساخت مولدهای زنجیره‌ای

مولدها را می‌توان همانند خط لوله به صورت زنجیره‌ای به منظور پردازش داده‌ها به یکدیگر متصل کرده و یک مولد بزرگتر ایجاد نمود. برای تغییر هر بخش از این مولد تنها کافی است تا زیر مولد آن را تغییر دهیم.

همچنین برای راحتی در کد نویسی می‌توان تمام آن‌ها را به صورت تک خطی پیاده سازی کرد. البته توجه داشته باشید که استفاده بیش از حد از این روش باعث می‌شود تا از خوانایی برنامه‌ی شما کاسته شود و کار را برای سایر برنامه نویسان سخت تر نماید.

فصل هفتم

ترفندهایی برای کار با دیکشنری‌ها

۷-۱. مقادیر پیش فرض در دیکشنری‌ها

یکی از متدهای پر کاربرد در پایتون برای دیکشنری‌ها، متد get است. این متد به دنبال یک کلید در دیکشنری گشته و در صورت نبود آن، یک مقدار پیش فرض را برای ما برمی‌گرداند. استفاده از این متد به منظور جلوگیری از برخی خطاها در برنامه نویسی بسیار مناسب است. در ادامه مثالی را در این مورد بررسی خواهیم نمود. فرض کنید که یک دیکشنری به صورت زیر داریم که شماره‌ی هر کاربر به عنوان کلید و نام کاربر به عنوان مقدار آن کلید در نظر گرفته شده است.

```
name_for_userid = {
    382: 'Alice',
    950: 'Bob',
    590: 'Dilbert',
}
```

حال قصد داریم تا از این دیکشنری استفاده کرده و یک تابع با نام greeting بنویسیم. این تابع شماره‌ی هر کاربر را دریافت کرده و پیام خوش آمدی را به آن کاربر نمایش می‌دهد. پیاده سازی اولیه این تابع به صورت زیر است.

```
def greeting(userid):
    return 'Hi %s!' % name_for_userid[userid]
```

تابع بالا بسیار ساده نوشته شده است. اما اگر شماره‌ای که به کاربری اختصاص ندارد را به این تابع بدهیم، برنامه‌ی ما دچار خطا می‌شود.

```
>>> greeting(382)
'Hi Alice!'

>>> greeting(33333333)
KeyError: 33333333
```

خطای KeyError چیزی نیست که ما بخواهیم در برنامه‌ی خود مشاهده کنیم. بهتر است تابع خود را طوری پیاده سازی نماییم تا در صورت عدم وجود یک شماره، دچار خطا نشده و یک پیام پیش فرض را بجای خطا به ما نمایش دهد.

بدین منظور از شرط if در برنامه‌ی خود استفاده می‌کنیم تا اگر شماره‌ی ارسال شده به تابع در دیکشنری مورد نظر ما وجود نداشت، یک پیغام خطای ثابت برگشت داده شود.

```
def greeting(userid):
    if userid in name_for_userid:
        return 'Hi %s!' % name_for_userid[userid]
    else:
        return 'Hi there!'
```

حال اگر از این تابع استفاده کنیم، خروجی‌هایی بصورت زیر به ما نمایش داده می‌شود.

```
>>> greeting(382)
'Hi Alice!'

>>> greeting(33333333)
'Hi there!'
```

بدین ترتیب کارایی برنامه‌ی ما بهتر شده است. از این پس اگر عددی را به این تابع بدهیم که متعلق به هیچ کاربری نباشد، پیغامی پیش فرض برای ما نمایش داده می‌شود. اما هنوز هم می‌توان کارایی این برنامه را بهبود بخشید. قطعه کد بالا به دلایل زیر از لحاظ کارایی ضعیف می‌باشد:

- عمل جست و جو در دیکشنری در این تابع دو بار انجام می‌شود (یکبار در شرط if و بار دیگر در هنگام جست و جو برای شماره‌ی کاربر در هنگام نمایش پیام)
- پیام Hi یکبار در if و بار دیگر در else نوشته شده است. بهتر است تا این دو با هم ادغام شوند.

- همچنین نوشتن چنین کدی از لحاظ پایتونی صحیح نیست. چرا که در صورت نبود کلیدی برای یک دیکشنری، بهتر است تا این خطا با try...except مدیریت شود.

در ادامه، کد بالا را به نوعی دیگر پیاده سازی می‌کنیم. این بار با استفاده از try...except خطای احتمالی را مدیریت خواهیم نمود.

```
def greeting(userid):
    try:
        return 'Hi %s!' % name_for_userid[userid]
    except KeyError:
        return 'Hi there'
```

در قطعه کد بالا دیگر عمل جست و جو در دیکشنری ۲بار انجام نمی‌شود.

اما باز هم می‌توانیم این کد را بهبود ببخشیم و آن را تمیز تر پیاده سازی نماییم. دیکشنری‌ها در پایتون دارای متدی با نام get هستند. با استفاده از این متد می‌توانیم مقداری پیش فرض را برای یک دیکشنری در نظر بگیریم تا در صورت عدم وجود یک کلید در دیکشنری، مقدار پیش فرض برای کاربر نمایش داده شود.

```
def greeting(userid):
    return 'Hi %s!' % name_for_userid.get(
        userid, 'there')
```

زمانی که از متد get استفاده می‌شود، کلید داده شده به آن در دیکشنری بررسی شده و در صورت وجود آن کلید، مقدار مورد نظر برگشت داده می‌شود؛ و اگر کلید در دیکشنری وجود نداشته باشد، مقدار پیش فرض (در مثال بالا there) برگشت داده خواهد شد.

```
>>> greeting(950)
'Hi Bob!'

>>> greeting(333333)
'Hi there!'
```

آخرین روش پیاده سازی تابع greeting بسیار تمیز و بهینه بوده و همچنین نیاز به کدنویسی کمتری دارد. بنابراین بهتر است در هنگام کار با دیکشنری‌ها از متد get برای آن‌ها استفاده شود تا از خطاهای احتمالی نیز جلوگیری شود.

۷-۱-۱. نکات کلیدی در هنگام کار با مقادیر پیش فرض در دیکشنری‌ها

بهتر است تا بجای استفاده از روش قدیمی برای یافتن یک کلید در دیکشنری، از متد get استفاده شود. بدین ترتیب کدهای برنامه بهینه تر و ساده تر خواهند شد. گاهی اوقات پیشنهاد می‌شود تا از کلاس collections.defaultdict در پایتون استفاده شود تا به برنامه نویسی بهتر شما کمک نماید.

۷-۲. مرتب سازی دیکشنری‌ها

عناصر موجود در یک دیکشنری دارای ترتیب خاصی نیستند. بنابراین در هنگام پیمایش روی یک دیکشنری، هیچ تضمینی وجود ندارد که عناصر آن بر اساس ترتیب مشخصی مورد پردازش قرار گیرند.

اما گاهی اوقات نیاز است تا دیکشنری خود را طبق روال خاصی بر اساس کلیدهای موجود در دیکشنری (یا مقادیر کلیدها) مرتب نماییم. برای مثال فرض کنید یک دیکشنری به صورت زیر در اختیار داریم.

```
>>> xs = {'a': 4, 'c': 2, 'b': 3, 'd': 1}
```

برای اینکه بتوانیم کلید و مقادیر این دیکشنری را مرتب کنیم، می‌توانیم از متد items روی این دیکشنری استفاده کرده و سپس از تابع sorted برای مرتب سازی آن استفاده نماییم.

```
>>> sorted(xs.items())
[('a', 4), ('b', 3), ('c', 2), ('d', 1)]
```

همانطور که مشاهده می‌کنید، کلیدها و مقادیر آن‌ها به صورت یک تاپل نمایش داده شده‌اند و سپس تابع sort تمام آن‌ها را در یک لیست قرار داده است.

به منظور مقایسه‌ی دو تاپل، تابع sorted اندیس صفرم هر کدام از آن‌ها را مقایسه می‌کند (که در مثال بالا همان کلید دیکشنری است). در صورتی که هر دو اندیس صفرم برابر بودند، به سراغ اندیس یکم می‌رود. در مثال بالا، تمام اندیس‌های صفرم منحصر به فرد بوده و عملیات مرتب سازی به سادگی انجام می‌شود.

گاهی اوقات ممکن است قصد داشته باشیم تا یک دیکشنری را براساس مقادیر کلیدهای آن‌ها مرتب نماییم. این عمل را با استفاده از آرگومان key در تابع sorted می‌توانیم انجام دهیم. Key در واقع یک تابع پایتون را از ما دریافت می‌کند که با توجه به آن تابع، مقادیر یک تاپل را مرتب می‌نماید. توجه داشته باشید که اینجا منظور از key، کلید موجود در دیکشنری‌ها نیست؛ بلکه کلیدی است که تابع sorted با استفاده از آن عملیات مرتب سازی را انجام می‌دهد.

فرض کنید که همان دیکشنری مثال قبل را می‌خواهیم بر اساس مقادیر کلیدهای موجود در آن مرتب نماییم. بدین منظور از تابعی به صورت lambda استفاده می‌کنیم که برای اینکار از اندیس یکم هر تاپل برای مرتب سازی استفاده می‌نماید.

```
>>> sorted(xs.items(), key=lambda x: x[1])
[('d', 1), ('c', 2), ('b', 3), ('a', 4)]
```

همانطور که مشاهده می‌کنید لیستی از تاپل‌ها که بر اساس مقادیر کلیدهایشان مرتب شده‌اند به ما نمایش داده شده است.

یکی دیگر از روش‌های موجود برای اینکار استفاده از کتابخانه‌ی operator در پایتون است. در این کتابخانه بسیاری از توابع کلیدی مرسوم در پایتون پیاده سازی شده است. مانند تابع operator.itemgetter یا operator.attrgetter. در مثال صفحه بعد همان عملیات مرتب سازی را با استفاده از این کتابخانه انجام داده‌ایم.

```
>>> import operator
>>> sorted(xs.items(), key=operator.itemgetter(1))
[('d', 1), ('c', 2), ('b', 3), ('a', 4)]
```

گاهی اوقات برای استفاده از برخی از توابع پیچیده، استفاده از کتابخانه‌ی operator به خوانایی بیشتر کد شما کمک می‌کند. اما برای توابع ساده‌ای مانند مثال گفته شده، استفاده از lambda پیشنهاد می‌شود.

همچنین یکی دیگر از مزیت‌های استفاده از lambda در این است که می‌توانید به طور کامل خروجی آن را به صورت دلخواه مدیریت کنید. برای مثال می‌توانید مرتب سازی یک دیکشنری را با محاسبه‌ی قدر مطلق مقادیر موجود در آن انجام دهید.

```
>>> sorted(xs.items(), key=lambda x: abs(x[1]))
```

در صورتی که بخواهید خروجی شما برعکس نمایش داده شود (از بزرگتر به کوچکتر)، تنها کافی است آرگومان reverse را در تابع sorted برابر با True قرار دهید.

```
>>> sorted(xs.items(),
           key=lambda x: x[1],
           reverse=True)
[('a', 4), ('b', 3), ('c', 2), ('d', 1)]
```

همانطور که پیش تر نیز اشاره شد، بهتر است تا با برخی از توابع کلیدی در پایتون آشنا شوید. این توابع به کد نویسی شما کمک شایانی می‌کنند و همچنین با استفاده از آن‌ها می‌توانید یک نوع داده را به نوعی دیگر نیز تبدیل کنید.

۷-۲-۱. نکات کلیدی در هنگام مرتب سازی دیکشنری‌ها

هنگامی که از تابع sorted در پایتون استفاده می‌کنیم، می‌توانیم مشخص نماییم که مرتب سازی بر اساس کدام اندیس موجود در تاپل انجام شود. برخی از توابع پر استفاده برای اینکار در کتابخانه‌ی operator نیز وجود دارد. البته پیشنهاد می‌شود تا برای

مدیریت بیشتر به منظور مرتب سازی عناصر، از توابع به صورت lambda استفاده نمایید.

۷-۳. پیاده سازی عبارت switch/case با استفاده از دیکشنری

همانطور که می‌دانید در پایتون دستورات switch/case وجود ندارد. بنابراین گاهی اوقات ممکن است نیاز باشد تا عبارات طولانی شرطی به صورت if..elif..else نوشته شود. ما در این بخش قصد داریم با روشی آشنا شویم که با استفاده از دیکشنری، عبارات switch/case را شبیه سازی نماییم.

فرض کنید که قطعه کدی به صورت زیر در اختیار داریم.

```
>>> if cond == 'cond_a':
        handle_a()
    elif cond == 'cond_b':
        handle_b()
    else:
        handle_default()
```

مسلما برای بررسی ۳ حالت مختلف، این عبارت به اندازه کافی مناسب است. اما اگر تعداد حالات مورد نیاز برای بررسی بیشتر باشد باید چه کار کرد؟ اگر بخواهیم ۱۰ حالت مختف را بررسی کنیم، کد ما بسیار پیچیده و طولانی خواهد شد.

یکی از راه‌ها برای رفع این مشکل استفاده از دیکشنری‌ها است. پایتون دارای توابعی به صورت درجه یک[1] است؛ یعنی آن‌ها می‌توانند به عنوان آرگومان به توابع دیگر منتقل شده، به عنوان مقدار از توابع دیگر بازگردانده شوند، به متغیرها اختصاص داده شده و حتی در ساختارهای داده‌ای ذخیره شوند.

برای شفاف سازی بیشتر مثال صفحه بعد را در نظر بگیرید. در این قطعه کد یک تابع تعریف نمودیم و سپس برای دسترسی به آن در آینده، آن را در یک لیست ذخیره می‌کنیم.

1 first-class functions

```
>>> def myfunc(a, b):
        return a + b

>>> funcs = [myfunc]
>>> funcs[0]
<function myfunc at 0x107012230>
```

برای فراخوانی این تابع، باید به اندیس لیست مورد نظر دسترسی پیدا کنیم؛ و در ادامه مقادیری را به عنوان آرگومان به تابع خود انتقال دهیم.

```
>>> funcs[0](2, 3)
5
```

حال با بکارگیری این ایده قصد داریم تا بلاک‌های if..else طولانی خود را بهینه کنیم. در ادامه یک دیکشنری تعریف کرده و سپس برای کلیدهای آن، توابعی را به عنوان مقدار در نظر می‌گیریم. به مثال زیر توجه کنید.

```
>>> func_dict = {
    'cond_a': handle_a,
    'cond_b': handle_b
}
```

در مثال بالا، handle_a و handle_b دو تابع هستند که هر کدام برای یک کلید در نظر گرفته شده‌اند. حال بجای استفاده از if/else، مقدار کلیدی را در این دیکشنری جست و جو کرده و سپس تابع مربوط به آن را فراخوانی می‌کنیم تا عملیات مورد نظر برای حالتی که قصد داریم بررسی شود، انجام گیرد.

```
>>> cond = 'cond_a'
>>> func_dict[cond]()
```

در صورتی که cond، در دیکشنری ما وجود داشته باشد، تابع مربوط به آن صدا زده خواهد شد. اما اگر این cond در دیکشنری وجود نداشته باشد، خطای KeyError به ما برگشت داده می‌شود. این خطای KeyError در واقع همان بلاک else ما خواهد بود.

برای رفع این مشکل از متد get روی دیکشنری خود استفاده می‌کنیم (برای اطلاعات بیشتر به بخش ۷,۱ مراجعه کنید). با استفاده از این متد، تابعی را به عنوان مقدار پیش فرض (در صورت عدم وجود کلید در دیکشنری) نیز به عنوان آرگومان به آن پاس می‌دهیم. در صورتی که کلید موجود پیدا نشود، این تابع فراخوانی می‌شود.

```
>>> func_dict.get(cond, handle_default)()
```

قطعه کد بالا نیز همانند مثال قبل عمل می‌کند. تنها با این تفاوت که تابعی نیز به عنوان مقدار پیش فرض برای معادل سازی بلاک else در نظر گرفته شده است و همچنین از خطای KeyError نیز جلوگیری شده است.

بیایید کمی این حالت را پیچیده تر کرده و مثالی کامل تر را بررسی نماییم. پس از خواندن مثال زیر، دیگر نوشتن هرگونه عبارت if..elif..else به صورت دیکشنری برای شما ساده خواهد شد.

ابتدا تابعی را به صورت if..elif..else پیاده سازی می‌کنیم. این تابع مقداری را به عنوان آپکد دریافت می‌نماید که مشخص کننده‌ی نوع عملیات مورد نظر است (جمع، تفریق، ضرب و تقسیم)؛ و دو مقدار دریافتی دیگر با توجه به عملیات مورد نظر جمع یا ضرب یا ... خواهند شد.

```
>>> def dispatch_if(operator, x, y):
        if operator == 'add':
            return x + y
        elif operator == 'sub':
            return x - y
        elif operator == 'mul':
            return x * y
        elif operator == 'div':
            return x / y
```

حال از تابع dispatch_if که در بالا نوشتیم می‌توانیم به صورت زیر استفاده کنیم.

```
>>> dispatch_if('mul', 2, 8)
16
>>> dispatch_if('unknown', 2, 8)
None
```

توجه داشته باشید زمانی که از unknown به عنوان آپکد برای تابع مورد نظر استفاده می‌کنیم، مقدار None به ما نمایش داده می‌شود. چرا که پایتون به صورت ضمنی در انتهای تمام توابع عبارت return None را می‌نویسد.

حال قصد داریم تابع dispatch_if را تغییر داده تا با استفاده از دیکشنری، شرایط و حالات مختلف را بررسی نماییم. این تابع را dispatch_dict نامگذاری می‌کنیم.

```
>>> def dispatch_dict(operator, x, y):
        return {
            'add': lambda: x + y,
            'sub': lambda: x - y,
            'mul': lambda: x * y,
            'div': lambda: x / y,
        }.get(operator, lambda: None)()
```

این تابع همانند تابع قبل عمل کرده و دقیقا همان خروجی را به ما برمی‌گرداند.

```
>>> dispatch_dict('mul', 2, 8)
16
>>> dispatch_dict('unknown', 2, 8)
None
```

راه‌های بسیاری وجود دارد تا بتوان تابع dispatch_dict را باز هم بهبود بخشید.

هر زمان که ما این تابع را فراخوانی می‌کنیم، یک دیکشنری به همراه تعدادی تابع lambda ایجاد می‌شود که از لحاظ عملکرد بسیار بد است. برای بهبود عملکرد این تابع، بهتر است تا دیکشنری مورد نظر تنها یکبار به صورت ثابت (constant) تعریف

شود و زمانی که تابع فراخوانی می‌شود، به آن دیکشنری اشاره گردد. بدین ترتیب دیگر نیازی به ساخت یک دیکشنری پس از هربار فراخوانی تابع نمی‌باشد.

همچنین برای استفاده از عملگرهای ساده‌ای مانند جمع یا ضرب، بهتر است تا از توابع داخلی پایتون و کتابخانه‌ی operator (بجای تابع lambda) استفاده شود. این کتابخانه اکثر عملگرهای مورد نیاز را برای ما فراهم کرده است. مانند operator.add، operator.mul و ... . اما در مثال بالا، دلیل استفاده از lambda تنها نشان دادن قابلیت پیاده سازی یک دیکشنری برای حالات مختلف و انجام یکسری عملیات بوده است.

بدین ترتیب می‌توانیم بسیاری از قطعه کدهای خود را که با عبارات طولانی if..elif..else نوشته شده‌اند، بدین صورت بازنویسی نماییم. البته توجه داشته باشید که از این تکنیک در همه‌ی شرایط استفاده نکنید. عبارات کوتاه if..else بهتر است تا به همان صورت نوشته شوند.

۷-۳-۱. نکات کلیدی در هنگام معادل سازی عبارات switch/case در پایتون

در پایتون عبارات switch/case وجود ندارد؛ اما با استفاده از دیکشنری‌ها می‌توان عبارات طولانی if..elif..else را بهینه نموده و کد خود را ساده تر نماییم.

۷-٤. نکاتی که هنگام کار با دیکشنری‌ها باید به آن ها توجه کنید

گاهی اوقات هنگام برنامه نویسی با قطعه کدهایی روبرو می‌شویم که دید ما را نسبت به آن زبان باز تر می‌کند و ممکن است حتی یک خط کد باعث شود که یک زبان برنامه نویسی را خیلی بیشتر از پیش درک کنیم.

قطعه کدی که در این بخش می‌خواهیم در مورد آن صحبت کنیم، یک دیکشنری بوده که ممکن است در نگاه اول بسیار ساده به نظر بیاید؛ اما با توجهی بیشتر به آن،

باعث می‌شود تا نحوه‌ی کارکرد مفسر[1] پایتون را بهتر درک نماییم. به قطعه کد زیر توجه کنید.

```
>>> {True: 'yes', 1: 'no', 1.0: 'maybe'}
```

ابتدا نگاهی دقیق تر به کد بالا بیندازید. به نظرتان با نوشتن کد بالا، دیکشنری مورد نظر به همین صورت ساخته خواهد شد؟

مفسر پایتون قطعه کد بالا را به صورت زیر تفسیر کرده و خروجی زیر را به ما نمایش می‌دهد.

```
>>> {True: 'yes', 1: 'no', 1.0: 'maybe'}
{True: 'maybe'}
```

من هم برای اولین بار با مشاهده‌ی این خروجی متعجب شدم. اما در ادامه با نگاهی دقیق تر به آن، گام به گام پیش خواهیم رفت تا دریابیم این خروجی به چه صورت حاصل شده است.

هنگامی که مفسر پایتون دیکشنری نوشته شده‌ی ما را پردازش می‌نماید، ابتدا یک شیء از دیکشنری ساخته می‌شود و سپس کلیدها و مقادیر آن‌ها به ترتیب وارد شده و درون آن قرار می‌گیرد. بنابراین اگر بخواهیم دقیق تر به نحوه‌ی ساخت این دیکشنری نگاه کنیم، مراحل ساخت آن به صورت زیر خواهد بود.

```
>>> xs = dict()
>>> xs[True] = 'yes'
>>> xs[1] = 'no'
>>> xs[1.0] = 'maybe'
```

در واقع مفسر پایتون تمام کلیدهایی که به دیکشنری خود اختصاص داده‌ایم را برابر با یکدیگر می‌داند.

---

1 interpreter

```
>>> True == 1 == 1.0
True
```

طبیعتا عدد ۱ با ۱٫۰ برابر است. اما چرا پایتون True را نیز برابر با ۱ می‌داند؟ پس از بررسی داکیومنت زبان پایتون، متوجه خواهیم شد که bool در واقع زیر کلاسی از int است. طبق آنچه که در داکیومنت رسمی پایتون آمده است:

نوع bool در واقع زیرمجموعه ای از نوع intاست و مقادیر bool همانند ۰ و ۱ عمل می‌کنند.

این بدان معناست که شما حتی می‌توانید از boolها به عنوان اندیس یک لیست یا تاپل نیز برای فراخوانی عنصری از آن‌ها نیز استفاده نمایید.

```
>>> ['no', 'yes'][True]
'yes'
```

اما به هر حال اینکار به هیچ وجه پیشنهاد نمی‌شود. چرا که خوانایی کد شما را سخت می کند.

حال از آنجا که پایتون مقدار True را با ۱ و ۱٫۰ برابر می‌داند، در هنگام ساخت دیکشنری، هر بار مقدار این کلید با مقدار جدید جایگزین می‌شود. به همین دلیل است که در انتها، تنها یک کلید با نام True وجود داشته و مقدار آن آخرین مقداری است که به ۱٫۰ داده شده است؛ یعنی maybe .

```
>>> {True: 'yes', 1: 'no', 1.0: 'maybe'}
{True: 'maybe'}
```

حال سؤالی که پیش می‌آید این است که چرا تنها مقدار کلید تغییر کرده و خود کلید بروز نمی‌شود و برابر با ۱٫۰ نمی‌باشد؟

دلیل این امر آن است که در واقع مفسر پایتون، خود کلید را هیچگاه بروز رسانی نمی کند.

```
>>> ys = {1.0: 'no'}
>>> ys[True] = 'yes'
>>> ys
{1.0: 'yes'}
```

همانطور که در مثال بالا مشاهده می‌کنید، کلید True جایگزین کلید ۱٫۰ نشده است. اما مقدار کلید ۱٫۰ بروز رسانی شده و رشته‌ای جدید به آن اختصاص داده شده است.

دیکشنری‌ها در پایتون بر اساس یک ساختار داده‌ی جدول هش[1] ساخته شده‌اند. یک جدول هش، کلیدهای موجود را در باکت‌های[2] مختلف بر اساس مقادیر هش هر کلید ذخیره می‌کند. مقدار هش در واقع به عنوان مقداری عددی با طول ثابت از کلید گرفته می‌شود. این عمل باعث جست و جوی سریعتر در یک دیکشنری می‌شود. جست و جوی مقدار هش عددیِ یک کلید، بسیار سریعتر از مقایسه‌ی یک شیء از کلید با تمام کلیدهای موجود در دیکشنری است. اما نحوه‌ی محاسبه‌ی هش به اندازه‌ی کافی خوب نیست و گاهی دو کلید که با یکدیگر متفاوت هستند، ممکن است مقادیر هش یکسانی داشته و در یک باکت قرار گیرند.

دو کلیدی که دارای مقدار هش یکسان هستند، تصادم هش‌ها[3] نامیده می‌شوند. این یک مورد خاص بوده که در این حالت الگوریتم‌های جدول هش به منظور یافتن یا قرار دادن عناصر، باید مدیریت شوند. بر این اساس، ممکن است نتایج غیر قابل انتظاری از دیکشنری‌های خود دریافت کنیم. برای شفاف سازی بیشتر، در ادامه کلاسی به منظور آشنایی بیشتر با این مفهوم تعریف نمودیم.

1 hash table data structure
2 buckets
3 hash collision

```
class AlwaysEquals:
    def __eq__(self, other):
        return True

    def __hash__(self):
        return id(self)
```

متد __eq__ در این کلاس همیشه مقدار True را برای ما برمی‌گرداند. یعنی در صورت مقایسه‌ی این کلاس با هر شیء دیگری، مقدار True به ما نمایش داده می‌شود.

```
>>> AlwaysEquals() == AlwaysEquals()
True
>>> AlwaysEquals() == 42
True
>>> AlwaysEquals() == 'waaat?'
True
```

همچنین هر نمونه از کلاس AlwaysEquals یک هش منحصر به فرد را با استفاده از تابع id به ما برمی‌گرداند.

```
>>> objects = [AlwaysEquals(),
               AlwaysEquals(),
               AlwaysEquals()]
>>> [hash(obj) for obj in objects]
[4574298968, 4574287912, 4574287072]
```

در مفسر پایتون، تابع id آدرس شیء موجود در حافظه را به نمایش می‌دهد که همیشه منحصر به فرد است.

با استفاده از این کلاس می‌توانیم اشیایی ایجاد نماییم که وانمود می‌کنند با هر شیء دیگری برابر بوده اما مقدار هش متفاوتی دارند. حال این کلاس به ما این اجازه را می‌دهد تا ببینیم که آیا کلیدهای موجود در یک دیکشنری، تنها در صورت برابر بودنشان با یکدیگر روی یکدیگر نوشته می‌شوند (overwrite) یا خیر.

همانطور که در مثال زیر مشاهده می‌کنید، دیکشنری این دو کلید را با یکدیگر برابر نمی‌داند.

```
>>> {AlwaysEquals(): 'yes', AlwaysEquals(): 'no'}
{ <AlwaysEquals object at 0x110a3c588>: 'yes',
  <AlwaysEquals object at 0x110a3cf98>: 'no' }
```

حال قصد داریم تا متد __hash__ را تغییر دهیم و همیشه یک مقدار را از آن برگردانیم تا ببینیم آیا بدین صورت می‌شود یک کلید را روی کلیدی دیگر نوشت یا خیر.

```
class SameHash:
    def __hash__(self):
        return 1
```

در ادامه مشاهده می‌کنیم که نمونه‌های کلاس SameHash با یکدیگر برابر نیستند اما هر دوی آن‌ها یک مقدار را به عنوان مقدار هش به ما بر می‌گردانند.

```
>>> a = SameHash()
>>> b = SameHash()
>>> a == b
False
>>> hash(a), hash(b)
(1, 1)
```

حال اگر از نمونه‌های این کلاس به عنوان کلیدهای یک دیکشنری استفاده نماییم چه اتفاقی رخ خواهد داد؟

```
>>> {a: 'a', b: 'b'}
{ <SameHash instance at 0x7f7159020cb0>: 'a',
  <SameHash instance at 0x7f7159020cf8>: 'b' }
```

همانطور که مشاهده می کنید، در مثال بالا نیز کلیدها روی یکدیگر نوشته نشده‌اند.

دیکشنری‌ها برابری را بررسی کرده و مقادیر هش‌ها را با یکدیگر مقایسه می‌کنند تا تشخیص دهند که دو کلید با یکدیگر برابر هستند یا خیر. بیایید تا به جمع بندی آنچه که تا بدین جا فهمیده‌ایم بپردازیم:

به طور کلی دیکشنری {True: 'yes', 1: "no", 1.0: "maybe"} تبدیل به دیکشنری {True: "maybe"} خواهد شد. چراکه کلیدهای True، ۱ و ۱٫۰ هر سه با یکدیگر برابر بوده و دارای مقدار هش یکسانی هستند.

```
>>> True == 1 == 1.0
True
>>> (hash(True), hash(1), hash(1.0))
(1, 1, 1)
```

حال دیگر مشاهده‌ی خروجی زیر به هنگام ساخت این دیکشنری برای شما غیر قابل انتظار نخواهد بود.

```
>>> {True: 'yes', 1: 'no', 1.0: 'maybe'}
{True: 'maybe'}
```

اگر درک موضوعات مطرح شده در این بخش برای شما کمی دشوار است، سعی کنید نمونه کدهای موجود را خودتان در مفسر پایتون بنویسید تا بتوانید این مفاهیم را بهتر فرا گیرید. بدین صورت دانش شما از نحوه‌ی کارکرد زبان پایتون بیشتر خواهد شد.

۷-٤-۱. نکات کلیدی که در هنگام کار با دیکشنری‌ها باید به آن‌ها توجه نمود

کلیدهای موجود در دیکشنری در صورتی که متد __eq__ در آن‌ها برابر باشند و همچنین مقادیر هش آن‌ها یکسان باشند، روی یکدیگر نوشته شده و با هم برابر در نظر گرفته می‌شوند. به طور کلی تصادم کلیدهای دیکشنری ممکن است باعث ایجاد نتایج غیر منتظره‌ای شود.

۷-۵. ادغام دیکشنری‌ها با یکدیگر

آیا تا بحال پیش آمده که بخواهید یک پیکربندی یا تنظیماتی برای برنامه‌های پایتونی خود ایجاد کنید؟ یکی از موارد رایج برای ایجاد چنین پیکربندی‌هایی، استفاده از یک ساختار داده‌ی مناسب بوده که دارای تنظیمات و مقادیری پیش فرض است و کاربر می‌تواند در ادامه تنظیمات شخصی خود را بجای آن‌ها اعمال کند.

در اکثر برنامه‌های پایتونی، از دیکشنری به عنوان ساختار داده‌ی مربوط به پیکربندی برنامه‌ها استفاده می‌شود. پیکربندی‌ها در دیکشنری به صورت یک کلید و مقدار (key/value) تعریف شده و کاربر می‌تواند مقدار هر کلید را در آینده با مقادیر مناسب خود (تنظیمات شخصی) جایگزین نماید. بنابراین نیاز است روشی را فراگیریم تا با استفاده از آن به ترکیب یا ادغام تنظیمات پیش فرض در دیکشنری‌ها با مقادیر دلخواه بپردازیم. همچنین گاهی اوقات ممکن است بخواهید دو یا چند دیکشنری را با هم ادغام نمایید و تمام آن‌ها را در یک دیکشنری داشته باشید.

در این بخش برای انجام این موارد، با راه‌های مختلفی آشنا خواهیم شد.

برای مثال فرض کنید دو دیکشنری به صورت زیر در اختیار داریم.

```
>>> xs = {'a': 1, 'b': 2}
>>> ys = {'b': 3, 'c': 4}
```

حال قصد داریم تا یک دیکشنری با نام zs بسازیم که تمام کلیدها و مقادیر دو دیکشنری xs و ys درون آن وجود داشته باشد. اگر به مثال بالا بیشتر توجه کنید، پی خواهید برد که رشته‌ی b به عنوان کلید در هر دو دیکشنری وجود دارد. بنابراین نیاز است تا این مغایرت[1] در دو دیکشنری را نیز به نوعی رفع نماییم.

یکی از روش‌های مرسوم برای ادغام چنین دیکشنری‌هایی در پایتون، استفاده از متد update موجود در دیکشنری است.

1 conflict

```
>>> zs = {}
>>> zs.update(xs)
>>> zs.update(ys)
```

در واقع متد update تمام عناصر موجود (کلیدها و مقادیرشان) در دیکشنری سمت راست را در دیکشنری سمت چپ می‌نویسد و در صورتی که کلیدی در دیکشنری سمت چپ وجود داشته باشد، با مقدار جدید از کلید سمت راست جایگزین می‌شود.

```
def update(dict1, dict2):
    for key, value in dict2.items():
        dict1[key] = value
```

در نهایت دیکشنری حاصل zs شامل عناصر زیر خواهد بود.

```
>>> zs
>>> {'c': 4, 'a': 1, 'b': 3}
```

همانطور که مشاهده می‌کنید، از آنجا که آخرین متد update روی دیکشنری ys انجام شده است، کلید b با مقدار موجود در ys برای دیکشنری zs قرار داده شده است. شما می‌توانید در ادامه نیز باز هم از این متد استفاده کرده تا تعداد دیکشنری‌های بیشتری را درون zs ادغام نمایید.

یکی دیگر از ترفندهای موجود برای ایجاد یک دیکشنری با ادغام دو دیکشنری دیگر، استفاده از تابع dict به همراه عملگر ** در پایتون است.

```
>>> zs = dict(xs, **ys)
>>> zs
{'a': 1, 'c': 4, 'b': 3}
```

البته توجه داشته باشید که این روش برای ادغام تنها دو دیکشنری بکار برده می‌شود و نمی‌توان در یک خط بیش از دو دیکشنری را با یکدیگر ادغام نمود.[1]

---

۱ برای ادغام بیش از دو دیکشنری باید دوباره این دستور را برای دیکشنری‌های جدید بنویسید.

همچنین در نسخه‌ی پایتون ۳٫۵ به بعد، شما می‌توانید از عملگر ** به نوعی دیگر نیز برای ادغام دیکشنری‌ها کمک بگیرید. این روش یکی از روش‌های بهینه برای اینکار است و قدرت برنامه نویسی شما را افزایش می‌دهد. از این عملگر می‌توان به صورت زیر برای ادغام تعداد دلخواهی از دیکشنری‌ها استفاده نمود.

```
>>> zs = {**xs, **ys}
```

عملکرد این روش دقیقا برابر با روش‌های قبل است. در قطعه کد بالا نیز دیکشنری سمت راست اولویت را بدست گرفته و در صورتی که کلیدی تکراری در این دو دیکشنری وجود داشته باشد، مقدار این کلید برابر با مقدار موجود در دیکشنری ys خواهد بود. خروجی حاصل از دستور بالا در دیکشنری zs قرار داشته که به صورت زیر است.

```
>>> zs
>>> {'c': 4, 'a': 1, 'b': 3}
```

استفاده از این سینتکس جدید، کار را برای ما بسیار آسان کرده است. با استفاده از این روش دیگر نیازی به نوشتن چندین متد update برای ادغام بیش از دو دیکشنری نمی‌باشد. بنابراین عملگر ** در پایتون ۳ بسیار کاربردی بوده و کار را برای ما ساده تر می‌کند.

۷-۵-۱. نکات کلیدی در هنگام ادغام دیکشنری‌ها

در نسخه‌ی پایتون ۳٫۵ به بعد، شما می‌توانید با کمک عملگر ** تعداد دلخواهی از دیکشنری‌ها را در یک دیکشنری ادغام نمایید. توجه داشته باشید که در صورت وجود کلیدی تکراری در چندین دیکشنری، اولویت با سمت راست ترین دیکشنری است.

اما اگر قصد داشته باشید تا برنامه‌ی شما با نسخه‌های قبل از پایتون ۳٫۵ نیز سازگاری داشته باشد، بهتر است تا از متد update برای ادغام دیکشنری‌های خود استفاده کنید.

۷-۶. چاپ و نمایش بهتر دیکشنری‌ها

مطمئنا شما هم در هنگام کد نویسی از print برای اشکال زدایی و دیباگ برنامه‌هایتان استفاده کرده‌اید. یا شاید ممکن است از log برای اینکار استفاده نمایید. اما برخی از داده‌ها هنگامی که با print نمایش داده می‌شوند، خواندن آن‌ها کمی دشوار است. یکی از انواع این داده‌ها دیکشنری است.

در مثال زیر، یک دیکشنری تعریف شده است. در هنگام نمایش این دیکشنری، ترتیب کلیدها لحاظ نشده و همچنین هیچ گونه تو رفتگی[1] نیز در هنگام نمایش نتایج وجود ندارد.

```
>>> mapping = {'a': 23, 'b': 42, 'c': 0xc0ffee}
>>> str(mapping)
{'b': 42, 'c': 12648430, 'a': 23}
```

خوشبختانه روش‌های دیگری نیز بجای استفاده از تابع str وجود دارد که برای نمایش زیباتر یک دیکشنری بکار برده می‌شود. یکی از این روش‌ها، استفاده از ماژول json در پایتون است. با کمک json.dumps می‌توان دیکشنری‌ها را به صورتی زیباتر نمایش داد.

```
>>> import json
>>> json.dumps(mapping, indent=4, sort_keys=True)
{
    "a": 23,
    "b": 42,
    "c": 12648430
}
```

خروجی بالا بسیار زیبا تر از حالت قبل با تو رفتگی‌های مناسب بوده و همچنین کلیدهای دیکشنری نیز در آن به ترتیب مرتب شده‌اند.

1 indention

درحالی که دیکشنری‌ها بدین صورت بسیار زیبا و قابل خوانا نمایش داده می‌شوند، اما این روشی کامل و بدون نقص نیست. نمایش دیکشنری‌ها با استفاده از ماژول json، تنها برای دیکشنری‌هایی موثر است که می‌توانند با استفاده از آن سریال[1] شوند. انواع داده‌های قابل پشتیبانی در آن عبارتند از:

dict, list, tuple, str, int, float, bool, None

بنابراین هنگامی که بخواهید یک دیکشنری را چاپ کنید که شامل یکی از انواع داده بجز موارد بالا باشد(برای مثال یک تابع)، با خطا مواجه خواهید شد.

```
>>> json.dumps({all: 'yup'})
TypeError: "keys must be a string"
```

همچنین در صورتی که از نوع داده‌ی set نیز استفاده نماییم، به خطا برمی‌خوریم.

```
>>> mapping['d'] = {1, 2, 3}
>>> json.dumps(mapping)
TypeError: "set([1, 2, 3]) is not JSON serializable"
```

حال بهتر است روشی دیگر را مورد بررسی قرار دهیم تا دیگر با چنین مشکلاتی مواجه نشویم. یکی از روش‌های قدیمی و مرسوم برای نمایش زیبای اشیا در پایتون، استفاده از ماژول pprint است. به مثال زیر توجه کنید.

```
>>> import pprint
>>> pprint.pprint(mapping)
{'a': 23, 'b': 42, 'c': 12648430, 'd': set([1, 2,
3])}
```

همانطور که مشاهده می‌کنید، این ماژول توانایی نمایش داده‌ی set را داشته و همچنین کلیدها را نیز به ترتیب نمایش می‌دهد. در مقایسه با روش قدیمی و سنتی، خروجی حاصل از این روش زیباتر بوده و قابلیت خوانایی بالاتری دارد.

1 serialized

اگرچه در مقایسه با روش json.dumps، با استفاده از این روش نمی‌توان تورفتگی‌هایی برای خروجی یک دیکشنری در نظر گرفت. به طور کلی، پیشنهاد می‌شود اگر در دیکشنری تنها داده‌هایی وجود دارد که قابل سریال شدن هستند، از json.dumps برای نمایش زیبا تر آن‌ها استفاده کنید.

۷-٦-۱. نکات کلیدی در هنگام نمایش دیکشنری‌ها

به صورت پیش فرض، خروجی‌های حاصل از نمایش دیکشنری‌ها در پایتون خوانایی بالایی ندارند. می‌توان برای اینکه نمایش زیباتری از دیکشنری‌های خود داشته باشیم، از ماژول‌هایی مانند pprint یا json در پایتون استفاده نماییم. البته توجه داشته باشید که در هنگام استفاده از ماژول json، تنها از داده‌هایی با قابلیت سریال شدن در دیکشنری خود استفاده کنید.

فصل هشتم

تکنیک‌هایی برای بهره وری بیشتر از زبان پایتون

۸-۱. آشنایی بیشتر با ماژول‌ها در پایتون

در زبان برنامه نویسی پایتون، شما به راحتی می‌توانید با نوشتن یک خط کد در ابتدای برنامه، از ماژول‌های مختلفی در هنگام کد نویسی استفاده کنید. این کار در سایر زبان‌های برنامه نویسی ممکن است کمی دشوار تر باشد. در بسیاری از زبان‌های برنامه نویسی استفاده از یک پکیج، بدون خواندن داکیومنت‌های آن و آشنایی کامل با نحوه‌ی کارکرد آن‌ها، بسیار سخت است. در پایتون، شما می‌توانید با ورود به مفسر خط فرمان آن (REPL) و استفاده از دستورات خاصی، اکثر قابلیت‌ها و ویژگی‌های یک ماژول را دریابید و با آن آشنا شوید.

در این بخش قصد داریم تا با نحوه‌ی کار با ماژول‌ها در خط فرمان پایتون آشنا شویم تا بتوانیم به سادگی از آن‌ها در برنامه‌های خود استفاده کنیم. تنها کافی است دستور python را در خط فرمان خود اجرا کرده و وارد مفسر پایتون شوید. این قابلیت به شما این امکان را می‌دهد تا بدون اجرای کامل برنامه‌های خود، به بررسی و اشکال زدایی بخش‌هایی از کدتان بپردازید و به سادگی قطعه کدهایی که احساس می‌کنید مشکل دارند را دیباگ کنید.

فرض کنید در برنامه‌ای از ماژول datetime استفاده کرده‌ایم. حال قصد داریم تا بفهمیم در این ماژول چه کلاس‌ها و توابعی وجود دارد و چگونه می‌توانیم بسته به نیاز از توابع و کلاس‌های موجود در این ماژول استفاده کنیم.

یکی از راه‌های آن جست و جو در اینترنت و خواندن مستندات کامل این کتابخانه است. اما راه ساده‌تر، استفاده از دستور dir در پایتون است که به ما این قابلیت را می‌دهد که به تمام اطلاعات مورد نیاز از یک ماژول دسترسی داشته باشیم.

```
>>> import datetime
>>> dir(datetime)
['MAXYEAR', 'MINYEAR', '__builtins__', '__cached__',
'__doc__', '__file__', '__loader__', '__name__',
'__package__', '__spec__', '_divide_and_round',
'date', 'datetime', 'datetime_CAPI', 'time',
'timedelta', 'timezone', 'tzinfo']
```

در مثال بالا، ابتدا کتابخانه‌ی datetime را ایمپورت کرده و سپس با استفاده از دستور dir، جزییات مربوط به آن را مشاهده نمودیم. فراخوانی تابع dir روی یک ماژول باعث می‌شود تا لیستی از تمام ویژگی‌های موجود در یک کتابخانه را به ما نمایش دهد.

از آنجا که در پایتون همه چیز به عنوان یک شیء (object) در نظر گرفته می‌شود، با استفاده از این تابع می‌توانیم مشخصات تمام کلاس‌های موجود در یک کتابخانه را نیز مشاهده کنیم (برای مثال datetime.date). بنابراین هنگامی که از این تابع برای کلاس datetime.date استفاده کنیم، خروجی زیر به ما نمایش داده می‌شود.

```
>>> dir(datetime.date)
['__add__', '__class__', ..., 'day', 'fromordinal',
'isocalendar', 'isoformat', 'isoweekday', 'max',
'min', 'month', 'replace', 'resolution', 'strftime',
'timetuple', 'today', 'toordinal', 'weekday', 'year']
```

همانطور که مشاهده می‌کنید، این تابع یک دید کلی نسبت به کلاس‌ها و توابع موجود در یک ماژول را به ما می‌دهد.

گاهی اوقات ممکن است اطلاعات نمایش داده شده، بسیار طولانی بوده و نیازی به بسیاری از آن‌ها نباشد. بدین منظور از ترفندی استفاده می‌کنیم تا تنها اطلاعات مربوط به مواردی که نیاز داریم به ما نشان داده شود.

```
>>> [_ for _ in dir(datetime) if 'date' in _.lower()]
['date', 'datetime', 'datetime_CAPI']
```

در مثال بالا با نوشتن یک لیست با حلقه‌ی for و شرط if در یک خط، تنها مواردی را از خروجی دستور dir(datetime) استخراج کردیم که در آن‌ها نام date وجود دارد. توجه داشته باشید که استفاده از متد lower بدین منظور است که جست و جوی ما حساس به حروف کوچک و بزرگ نباشد.

گاهی اوقات دریافت تنها لیستی از ویژگی‌های موجود در یک ماژول ممکن است کمکی به ما نکند. بنابراین نیاز است تا اطلاعات بیشتری در مورد هرکدام از آن‌ها بدست آورده و با نحوه‌ی کار با آن‌ها آشنا شویم.

با استفاده از تابع help در پایتون می‌توانیم اطلاعات جامعی در مورد تمام اشیاء موجود در پایتون بدست آوریم.

```
>>> help(datetime)
```

با اجرای دستور بالا در مفسر پایتون، اطلاعاتی در مورد ماژول datetime به شما نمایش داده می‌شود.

```
Help on module datetime:

NAME
    datetime - Fast implementation of the datetime
type.
CLASSES
    builtins.object
        date
            datetime
        time
```

بدین ترتیب شما می‌توانید مستندات کاملی در مورد ماژول datetime بدست آورید. برای خروج از این محیط کافی است تا از کلید q استفاده کنید.

شما می‌توانید تابع help را برای سایر اشیا موجود در پایتون نیز استفاده کنید. بدین صورت مفسر پایتون به طور خودکار مستنداتی برای شما آماده کرده و به شما نشان می‌دهد. شما برای کلاس‌هایی که خود ایجاد می‌کنید نیز می‌توانید از تابع help استفاده

نمایید. این تابع در صورت وجود docstring در کلاس شما، آن را نمایش خواهد داد. در ادامه سایر موارد استفاده از این تابع را می‌توانید مشاهده کنید.

```
>>> help(datetime.date)
>>> help(datetime.date.fromtimestamp)
>>> help(dir)
```

طبیعتا با کمک موتورهای جست و جویی مانند Google و سایت‌هایی مانند Stackoverflow، اطلاعات جامع تری نسبت به استفاده از توابعی مانند dir و help می‌توانید بدست آورید. اما این توابع نیز گاهی اوقات برای بدست آوردن اطلاعاتی مختصر و زمانی که دسترسی به اینترنت ندارید، می‌توانند مفید واقع شوند.

۸-۱-۱. نکات کلیدی در مورد ماژول‌ها در پایتون

با استفاده تابع dir در پایتون می‌توانیم ویژگی‌ها، توابع و کلاس‌های موجود در یک ماژول را مشاهده کنیم. همچنین با کمک تابع help نیز می‌توان مستندات مختصری در مورد هر کدام از ماژول‌ها و حتی سایر اشیاء موجود در پایتون بدست آورد.

۸-۲. ایزوله سازی محیط پروژه با کمک Virtualenv

زبان برنامه نویسی پایتون دارای یک سیستم یکپارچه سازی بسته‌ها است که با استفاده از آن می‌توانید بسته‌های مورد نیاز خود را که به کمک ابزار pip نصب و مدیریت کنید. یکی از مشکلات موجود در هنگام نصب یک بسته[1]، اثر گذاری آن روی محیط global پایتون است. بدین معنا که بسته‌ی مورد نظر روی کل سیستم عامل نصب می‌شود.

مشکل زمانی رخ می‌دهد که چندین پروژه در سیستم خود داشته باشیم که هر کدام از آن‌ها از نسخه‌ی متفاوتی از یک پکیج استفاده می‌کنند. برای مثال یک پروژه با Django 2.2 کار می‌کند؛ در صورتی که پروژه‌ی دیگر نیاز به Django 3.0 دارد.

1 package

هنگامی که یک بسته را به صورت global روی سیستم نصب می‌کنید، تنها می‌توانید یک نسخه از هر بسته روی سیستم خود داشته باشید که می‌تواند مشکلاتی برای شما در آینده به وجود آورد.

حتی ممکن است پروژه‌هایی روی سیستم خود داشته باشید که هرکدام با نسخه‌ی مختلفی از پایتون کار می‌کنند. برای مثال برنامه‌ای قدیمی روی سیستم شما وجود دارد که هنوز به توسعه‌ی آن مشغولید و نیاز به استفاده از پایتون ۲ دارید. در حالی که سایر پروژه‌های شما از پایتون ۳ استفاده می‌کنند. یا حتی شاید پروژه‌ای نیاز به پایتون ۳,۵ داشته و پروژه‌ای دیگر به پایتون ۳,۷ نیاز داشته باشد.

علاوه بر این موارد، نصب یک بسته به صورت global ممکن است مشکلاتی امنیتی روی سیستم عامل ایجاد کند. چرا که برای نصب یک بسته به این صورت، نیاز است تا شما دسترسی سطح بالا (root یا admin) در سیستم داشته باشید؛ و زمانی که دستور pip install را اجرا می‌کنید، پکیج شما از اینترنت دانلود شده و نصب می‌شود.

۸-۲-۱. استفاده از محیط مجازی[1] در پروژه‌ها

راه حل مشکل بالا، جدا سازی محیط پروژه‌ی خود با کمک یک محیط مجازی (Virtual Environment) است. بدین ترتیب شما می‌توانید بسته‌های مورد نیاز خود را در هر پروژه جدا کرده و حتی از نسخه‌های مختلفی از پایتون برای برنامه‌هایتان استفاده کنید.

در واقع یک Virtual Environment، یک محیط کاملاً ایزوله است و در همان مسیر پروژه‌ی ما قرار می‌گیرد. برای شفاف سازی بیشتر با این موضوع، در ادامه یک محیط مجازی (یا به اختصار virtualenv) ساخته و سپس بسته‌هایی را برای آن نصب می‌کنیم.

1 Virtual Environment

ابتدا بیایید تا ببینیم محیط پایتون global ما در چه مسیری قرار دارد. برای اینکار می‌توانیم از دستور which در سیستم عامل‌های Linux یا macOS استفاده کرده تا مسیر مدیر بسته‌ی[1] pip را پیدا کنیم.

```
$ which pip3
/usr/local/bin/pip3
```

در اکثر مواقع پیشنهاد می‌شود تا محیط ایزوله‌ی خود را در همان مسیر پروژه‌ی خود بسازید. برای ساخت یک virtualenv از دستور زیر استفاده می‌شود.

```
$ python3 –m venv ./venv
```

با اجرای این دستور، یک مسیر با نام venv برای شما ساخته می‌شود که این محیط ایزوله از پایتون ۳ استفاده می‌کند.

```
$ ls venv/
bin     include     lib     pyvenv.cfg
```

حال اگر دوباره از دستور which برای پیدا کردن نسخه‌ی فعال pip استفاده کنیم، دوباره همان خروجی global مانند حالت قبل به ما نمایش داده می‌شود.

```
$ which pip3
/usr/local/bin/pip3
```

این خروجی بدان معناست که هنوز هم اگر پکیجی را با pip نصب کنیم، آن بسته به صورت global روی سیستم نصب می‌شود. ما یک گام دیگر برای استفاده از محیط ایزوله‌ی خود فاصله داریم و آن هم فعال سازی آن است. برای فعال سازی محیط virtualenv که دقایقی پیش ایجاد کردیم نیاز است تا دستور زیر را وارد کنیم.

```
$ source ./venv/bin/activate
(venv) $
```

1 Package Manager

با اجرای فایل activate در virtualenv، محیط مجازی ما فعال شده و از این پس تمام دستورات پایتونیِ ما تنها روی همین محیط تاثیر می‌گذارند. برای تصدیق این موضوع می‌توانیم دوباره از دستور which برای pip استفاده کنیم.

```
(venv) $ which pip3
/Users/dan/my-project/venv/bin/pip3
```

همانطور که مشاهده می‌کنید، از این پس با اجرای دستور pip، نسخه‌ی موجود در virtualenv فراخوانی می‌شود و دیگر بسته‌ای به صورت global در سیستم نصب نخواهد شد. همین حالت برای اجرای دستور python نیز صدق می‌کند. بار دیگر با کمک دستور which قصد داریم بفهمیم که آیا در هنگام اجرای مفسر python، از محیط مجازی ما استفاده می‌شود یا خیر.

```
(venv) $ which python
/Users/dan/my-project/venv/bin/python
```

البته توجه داشته باشید که محیط ما در حال حاضر کاملاً خالی بوده و بسته‌ای برای محیط پایتونی ما نصب نشده است. با اجرای دستور زیر می‌توانیم مشاهده کنیم که تنها بسته‌های پایه‌ای و پیش فرض در محیط ایزوله شده‌ی ما وجود دارد.

```
(venv) $ pip list
pip (9.0.1)
setuptools (28.8.0)
```

به منظور نصب یک بسته در محیط virtualenv، می‌توانیم از دستور pip install <package> استفاده کنیم. در قطعه کد زیر بسته‌ای با نام schedule را نصب می‌کنیم.

```
(venv) $ pip install schedule
Collecting schedule
    Downloading schedule-0.4.2-py2.py3-none-any.whl
Installing collected packages: schedule
Successfully installed schedule-0.4.2
```

همانطور که مشاهده می‌کنید، دیگر نیازی به دسترسی admin/root برای اجرای این دستور نمی‌باشد. همچنین زمانی که بخواهید این بسته را بروز رسانی کنید، تنها همین بسته‌ی موجود در این محیط مجازی بروز می‌شود. به معنای دیگر نیازمندی‌های پروژه‌های شما در هر محیط مجازی از یکدیگر جدا بوده (حتی با محیط global) و هیچ ارتباطی با یکدیگر ندارند.

حال اگر دوباره دستور pip list را اجرا کنیم، می‌بینیم که بسته‌ی schedule نیز به ما نمایش داده می‌شود.

```
(venv) $ pip list
pip (9.0.1)
schedule (0.4.2)
setuptools (28.8.0)
```

اگر در این حالت یک فایل پایتونی (فایلی با پسوند py) را اجرا کنیم یا برای هر کار دیگری از دستور python استفاده کنیم، از نسخه‌ی پایتون موجود در همین محیط مجازی و بسته‌های نصب شده برای آن استفاده می‌شود.

اما چگونه می‌توان از این محیط خارج شد؟ همانند دستور activate، دستوری با نام deactivate نیز وجود دارد که با اجرای آن می‌توانیم از محیط venv خارج شویم.

```
(venv) $ deactivate
$ which pip3
/usr/local/bin
```

به طور کلی استفاده از virtual environment باعث می‌شود تا بهتر بتوانید بسته‌های پایتونی خود را با توجه به نیازتان در هر پروژه به صورت مجزا مدیریت کنید. بنابراین همیشه پیشنهاد می‌شود تا برای تمام پروژه‌هایتان از یک محیط مجازی جدا استفاده کنید.

همچنین توجه داشته باشید که برای پروژه‌های خود یک فایل requirements.txt ایجاد کرده و تمام بسته‌های نصب شده برای پروژه‌ی خود را در آن ذکر کنید. برای این کار می‌توانید از دستور زیر استفاده کنید.

```
(venv) $ pip freeze > requirements.txt
```

از این پس هر برنامه نویس دیگری می‌تواند با ایجاد یک virtualenv، تنها با اجرای دستور زیر بسته‌های مورد نیاز پروژه را برای خود نصب کند.

```
(venv) $ pip install –r requirements.txt
```

۸-۲-۲. نکات کلیدی در هنگام ایزوله سازی پروژه‌های پایتونی

استفاده از virtuale environment باعث می‌شود تا نیازمندی‌های پروژه‌های ما از یکدیگر جدا شوند. به منظور جلوگیری از تداخل در نسخه‌های مختلفی از زبان پایتون یا نسخه‌های مختلف یک بسته، ساخت یک محیط مجازی مجزا برای هر پروژه پیشنهاد می‌شود. این عمل به مدیریت بهتر بسته‌های شما در آینده بسیار کمک می‌کند.

۸-۳. نگاهی به پشت پرده‌ی بایت کدها[1] در پایتون

هنگامی که مفسر CPython برنامه‌ی شما را اجرا می‌کند، ابتدا تمام برنامه به دنباله‌ای از بایت کدها ترجمه می‌شود. در واقع بایت کد، یک زبان واسطه برای پایتون بوده که به بهینه سازی کد شما کمک می‌کند.

این کار باعث صرفه جویی در زمان و حافظه در هنگام اجرای دوباره‌ی کد می‌شود. برای مثال، بایت کدهای حاصل از مرحله‌ی کامپایل، روی دیسک با نام‌هایی مانند pyc یا pyo کش شده تا اجرای دوباره‌ی کدها سریعتر شود.

تمام این مراحل از چشم برنامه نویس پنهان است. چراکه برنامه نویس لازم نیست تا از مراحل و چگونگی تفسیر کدهای خود اطلاع داشته باشد. اما با این حال ما قصد

1 bytecodes

داریم تا دریابیم که مفسر CPython چگونه این کار را انجام می‌دهد. گاهی اوقات درک چگونگی کارکرد اجزاء درونی پایتون، به ما کمک می‌کند تا کدهای خود را به صورتی بهینه تر بنویسیم.

در ادامه یک تابع با نام greet پیاده سازی خواهیم کرد و با استفاده از آن در ادامه‌ی این بخش، قصد داریم تا چگونگی کارکرد بایت کدها در پایتون را دریابیم.

```
def greet(name):
    return 'Hello, ' + name + '!'

>>> greet('Guido')
'Hello, Guido!'
```

همانطور که اشاره کرده بودیم، CPython کد نوشته شده‌ی ما را قبل از اجرا، به یک زبان میانی ترجمه می‌کند. خب اگر این ادعا درست باشد، ما باید بتوانیم نتیجه‌ی حاصل از مرحله‌ی کامپایل را مشاهده کنیم.

هر تابع در پایتون دارای ویژگی __code__ است که با استفاده از آن می‌توانیم دستور العمل‌های مربوط به ماشین مجازی پایتون[1]، ثابت‌ها[2] و متغیرهایی[3] که در تابع ما استفاده می‌شود را مشاهده کنیم.

```
>>> greet.__code__.co_code
b'd\x01|\x00\x17\x00d\x02\x17\x00S\x00'
>>> greet.__code__.co_consts
(None, 'Hello, ', '!')
>>> greet.__code__.co_varnames
('name',)
```

همانطور که مشاهده می‌کنید، co_consts شامل بخشی از رشته‌ی برگشت داده شده در تابع است. همانطور که از نام constants مشخص است، آن‌ها مقادیر ثابتی هستند و

1 Virtual Machine Instructions
2 Constants
3 Variables

در برنامه تغییر نمی‌کنند. این مقادیر به منظور صرفه جویی در مصرف حافظه، جدا از code و varnames نگهداری شده‌اند.

بنابراین زبان پایتون بجای تکرار مقادیر ثابت در co_code، آن‌ها را در مکانی مجزا نگهداری می‌کند؛ و از این پس co_code می‌تواند با جست و جو در فهرست ثابت‌ها، آن‌ها را بیابد و در برنامه جای دهد. همین عمل برای مقادیر co_varnames نیز صدق می‌کند.

هنوز هم با مشاهده‌ی خروجی co_code نکته‌ی خاصی دستگیرمان نمی‌شود. این خروجی در واقع زبانی برای ارتباط با ماشین مجازی پایتون بوده و قابل خواندن نیست. توسعه دهندگان CPython این زبان را درک می‌کنند. بنابراین آن‌ها ابزاری با نام disassembler برای ما فراهم کرده‌اند که با کمک آن می‌توانیم این بایت کدها را بهتر درک کنیم.

دیس اسمبلر (disassembler) در پایتون در ماژولی با نام dis قرار دارد. بنابراین به راحتی می‌توانیم از این ماژول و تابع dis.dis موجود در آن به منظور درک بهتر بایت کدهای تابعی که نوشته‌ایم استفاده کنیم.

```
>>> import dis
>>> dis.dis(greet)
2           0 LOAD_CONST               1 ('Hello, ')
            2 LOAD_FAST                0 (name)
            4 BINARY_ADD
            6 LOAD_CONST               2 ('!')
            8 BINARY_ADD
            10 RETURN_VALUE
```

در واقع این کتابخانه دستورالعمل‌های موجود را به بخش‌های مختلفی تقسیم کرده و به هرکدام یک آپکد مشخص اختصاص داده است. همچنین می‌توانید مشاهده کنید که مراجع مربوط به ثابت‌ها و متغیرها با بایت کدها ترکیب شده و co_consts و co_varnames به صورتی واضح به ما نشان داده شده است.

با نگاهی دقیق تر به آپکدهای ارایه شده، می‌توانیم دریابیم که CPython چگونه دستورات نوشته شده در تابع greet را اجرا می‌کند. در واقع CPython ابتدا مقدار ثابت ('Hello',) در اندیس ۱ را دریافت کرده و آن را در یک پشته قرار می‌دهد. سپس متغیر name را دریافت کرده و آن را نیز به پشته اضافه می‌کند.

ساختار داده‌ی پشته (stack) برای فضای ذخیره سازی درونی در ماشین مجازی پایتون بسیار مورد استفاده قرار می‌گیرد. کلاس‌های متنوعی از ماشین مجازی در پایتون وجود دارد که یکی از آن‌ها ماشین پشته است. ماشین مجازی CPython نمونه‌ای از پیاده سازی یک ماشین مجازی است. ما در این بخش وارد جزییات مربوط به این مسائل نخواهیم شد. خواندن تئوری‌های مربوط به ماشین مجازی بسیار مفصل بوده که در این کتاب نمی‌گنجد.

آنچه که در مورد پشته به عنوان یک ساختار داده جالب است، پشتیبانی آن تنها از دو عمل push و pop است. عمل push مقداری را به بالای یک پشته اضافه کرده و عمل pop آخرین مقدار اضافه شده به بالای پشته را برمی‌گرداند. برخلاف ساختار داده‌ی آرایه، امکان دسترسی به سایر عناصر موجود پایین تر از عنصر روی پشته وجود ندارد.

فرض می‌کنیم که در ابتدا پشته‌ی موجود خالی است و سپس دو آپکد ابتدایی اجرا می‌شود. پس از اجرای آن‌ها، پشته‌ی ما همانند زیر می‌شود.

```
0: 'Guido' (contents of "name")
1: 'Hello, '
```

سپس دستور BINARY_ADD دو مقدار بالای پشته را برداشته و به یکدیگر می‌چسباند. حال دوباره مقدار حاصل را روی پشته قرار می‌گیرد.

```
0: 'Hello, Guido'
```

پس از آن دوباره دستور LOAD_CONST اجرا شده و مقدار ثابت بعدی به روی پشته اضافه می‌شود.

```
0: '!'
1: 'Hello, Guido'
```

حال دوباره دستور آپکد BINARY_ADD اجرا شده و دو مقدار روی پشته به یکدیگر متصل می‌شود که حاصل آن رشته‌ی نهایی ما خواهد بود.

```
0: 'Hello, Guido!'
```

آخرین دستور بایت کد RETURN_VALUE است که از ماشین مجازی مقدار روی پشته را درخواست کرده و آن را به عنوان مقدار بازگشتی تابع، به برنامه برمی‌گرداند.

همانطور که در مثال بالا مشاهده کردید، نحوه‌ی اجرای تابع greet در ماشین مجازی CPython را به سادگی شبیه سازی نمودیم.

مبحث ماشین‌های مجازی در پایتون بسیار طولانی و مفصل است که در چارچوب این کتاب نمی‌گنجد. پیشنهاد می‌کنیم تا در آینده در مورد این مبحث مطالعه کنید تا بیشتر با زبان شیرین پایتون و لایه‌های پایین آن آشنا شوید.

۸-۳-۱. نکات کلیدی در هنگام کار با بایت کدها در پایتون

مفسر پایتون ابتدا برنامه‌های نوشته شده‌ی ما را به زبانی واسطه ترجمه کرده و سپس به اجرای دستورات بایت کد روی یک ماشین مجازی پشته‌ای می‌پردازد.

با استفاده از ماژول dis در پایتون می‌توان چگونگی اجرای دستورات بایت کد در پشت صحنه‌ی تمام برنامه‌ها را به سادگی مشاهده نمود.