



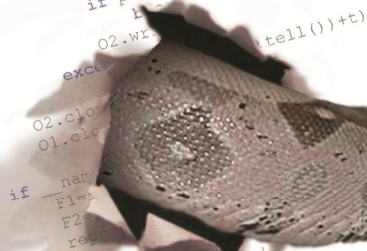
CD اختیاری

از این پس

بایتون

RepCopy.py

```
def repcopy(File1,File2,ReadSize=100000,t=10):
    O1=open(File1,'rb')
    O2=open(File2,'wb')
    while 1:
        try:
            p=O1.read(ReadSize)
            if p=="":
                break
            O2.write(p)
            O1.close()
            O2.close()
        except:
            O1.close()
            O2.close()
            break
```



```
def FastCopy(File1,File2):
    O1=open(File1,'rb')
    O2=open(File2,'wb')
    O1.seek(0)
    while 1:
        try:
            p=O1.read(100000)
            if p=="":
                break
            O2.write(p)
        except:
            break
```

```
"""Shared support for scanning document type declarations
import re
import string

_declname_match = re.compile(r'[a-zA-Z]([-_.a-zA-Z0-9]*\s*)').match
_declstringlit_match = re.compile(r'(\'[^\']*\'|\"[^\"]*\")').match

del re

class ParserBase:
    """Parser base class which provides some common support methods
    by the SGML/HTML and XHTML parsers."""

    def __init__(self):
        if self.__class__ is ParserBase:
            raise RuntimeError(
                "%s.ParserBase must be subclassed" %
                self.__class__.__name__)

    def getpos(self):
        """Return current position and offset.
        return self.lineno, self.offset"""

    def updatepos(self, i, j):
        """Internal -- update line number and offset. This should be
        called for each piece of data exactly once, in order -- in other
        words the concatenation of all the input strings to this
        function should be exactly the entire input.
        if i > j:
            return j
        rawdata = self.rawdata
        nlines = string.count(rawdata, "\n", i, j)
        self.lineno = self.lineno + nlines
        self.offset = string.index(rawdata, "\n", i, j) # Should not fail
        self.offset = j-(pos+1)
        nchars = self.offset + j-i

    def parse_declaration(self, i):
        """Declaration (for use by subclasses).
        This is some sort of declaration; in the
        declaration, this should only be the
        <!DOCTYPE html...>
        rawdata
        """
```

آموزش اصول برنامه‌نویسی

لیست‌ها، چندتایی‌ها، دیکشنری‌ها و ...

انواع داده‌ای کاربر - تعریف

کلاس‌ها و برنامه‌نویسی شیء‌گرا

پیاده‌سازی ساختمان داده‌ها

واژه‌نامه و تمرین

ترجمه و تألیف :

سعید خالقی

علیرضا حق‌نیا

بِسْمِ اللَّهِ الرَّحْمَنِ الرَّحِيمِ

از این پس

پایتون

ترجمه و تألیف:

سعید خالقی

علیرضا حق‌نیا

از این پس پایتون/ ترجمه و تألیف سعید خالقی و علیرضا حق‌نیا- شیراز: کوشامهر، ۱۳۸۲.

۳۳۰ ص.

۱. پایتون (زبان برنامه‌نویسی کامپیوتر) - راهنمای آموزشی

۰۰۵/۱۳۳

Q4۷۶/۷۳

انتشارات کوشامهر

مرکز نشر و پخش کتاب‌های دانشگاهی

شیراز- بلوار کریم‌خان زند- روبروی خیام - پاساژ مسعود - پلاک ۳۳

تلفن: ۲۳۰۳۷۹۸ دورنگار: ۲۳۵۸۴۶۵

ناشر برگزیده سال‌های ۷۴، ۷۶، ۷۹، ۸۰ و ۸۱

خادم نشر ۷۷



انتشارات کوشامهر

عنوان: از این پس پایتون

ترجمه و تألیف: سعید خالقی، علیرضا حق‌نیا

ناشر: کوشامهر شیراز

شمارگان: ۱۰۰۰ نسخه

نوبت چاپ: اول ۱۳۸۲

چاپ: نمونه

لیتوگرافی: واصف

تایپ و صفحه‌آرایی: سعید خالقی، علیرضا حق‌نیا

طرح جلد: سعید خالقی، علیرضا حق‌نیا

تصاویر ابتدای فصل‌ها: روزبه مرادی

شابک: ۹۶۴-۷۹۹۹-۳۰-۵

از اینکه برای فراگیری زبان برنامه‌نویسی پایتون این کتاب را انتخاب کرده‌اید سپاسگزاریم. در صورت تمایل به دریافت نسخه چاپی از طریق این ایمیل from.now.on.python@gmail.com با ما تماس بگیرید.

شما مجازید:

- این کتاب را به صورت رایگان دریافت کنید.
- به صورت رایگان با دوستانتان به اشتراک بگذارید.
- برای دانلود در سایت یا وبلاگ خود قرار دهید.

شما اجازه ندارید:

- مطالب این کتاب را به اسم خود کپی و منتشر کنید.
- از این کتاب استفاده تجاری کنید.
- در محتوای این کتاب دست ببرید.
- در صفحات این کتاب آرم یا لوگو خود را قرار دهید.

با سپاس

سعید خالقی / علیرضا حق نیا

مقدمه

با توسعه علم کامپیوتر و پیشرفت‌هایی که این علم در عصر حاضر پیدا کرده است، زبان‌های برنامه‌نویسی متعددی جهت ساخت برنامه‌های کاربردی مورد نیاز مردم طراحی شده‌اند. این زبان‌ها همه‌روزه ساده‌تر و قدرتمندتر می‌شوند و مردم جهان هم بسته به قابلیت‌هایی که این زبان‌ها دارند از آنها استفاده می‌کنند.

معمولاً هر یک از زبان‌های برنامه‌نویسی برای کاربرد خاصی طراحی شده است و تعداد زبان‌های همه‌منظوره به نسبت کمتر است. از جمله زبان‌هایی که جهت آموزش برنامه‌نویسی به مبتدیان استفاده شده است، «پایتون» می‌باشد. با اینکه این زبان ویژه مبتدیان طراحی شده بود اما طولی نکشید که زبان «پاسکال» در این جهت مورد استفاده قرار گرفت و جایگزین آن شد. پاسکال قدرت خوبی دارد و جهت آموزش کامپیوتر به مبتدیان هم مناسب است، اما تا زمانی که زبانی به نام «پایتون» پا به عرصه نگذارد.

نحوه نگارش مناسب، سادگی، قدرت و انعطاف‌پذیری فوق‌العاده پایتون به حدی است که علیرغم خصوصیات زبان پاسکال و حتی زبان قدرتمندی چون C، روز به روز بر محبوبیت آن افزوده می‌شود.

پایتون علاوه بر تمام خصایصی که زبان‌هایی چون پاسکال و C دارند خصوصیت بسیار مهم دیگری هم دارد و آن شیء‌گرا بودن پایتون است. زبان قدرتمندی چون ++C که یک زبان شیء‌گرا است و از آن برای برنامه‌نویسی‌های پیشرفته استفاده می‌شود، هم‌اکنون با ورود پایتون به عرصه برنامه‌نویسی در رقابت با این زبان قرار گرفته است، به طوری که برنامه‌نویسان پایتون ادعا دارند تقریباً تمام برنامه‌هایی که به زبان ++C نوشته می‌شود را می‌توان در پایتون با سادگی بیشتر و در مدت زمان بسیار کوتاه‌تری پیاده سازی کرد.

پایتون زبانی است که در اوایل سال ۱۹۹۰ در مؤسسه تحقیقات بین‌المللی ریاضیات و کامپیوتر هلند (CWI) توسط Guido van Rossum ساخته شد. با اینکه پایتون نام نوعی مار است، اما Guido این نام را زمانی که در حال مطالعه مطالبی در مورد گروه کمدی Monty Python's Flying Circus بود، برای زبان خود انتخاب کرد. او می‌خواست نامی کوتاه، منحصر به فرد و تا حدودی مرموز بر روی زبان خود بگذارد و اعتقاد داشت که نباید بر خلاف بعضی از مخترعین، نام خود را بر روی آن بگذارد. بنابراین Python را از نام این گروه کمدی که مورد توجه مردم بود، برگزید.

انگیزه اصلی ساختن زبان پایتون، طراحی یک Scripting Language برای سیستم عامل Amoeba بود که Guido در آن زمان درگیر توسعه آن بود، اما طراحی پایتون نشان داد که این زبان به منظور پوشش حوزه گوناگون و وسیعی از مقاصد، به اندازه کافی عمومی است.

پایتون هم‌اکنون توسط هزاران مهندس در سراسر جهان به‌طور فزاینده‌ای در نقش‌های مختلف استفاده می‌شود. کمپانی‌ها امروزه از پایتون در محصولات تجاری برای انجام اعمالی چون آزمایش تراشه‌ها و بُردها، توسعه واسط‌های گرافیکی کاربر (GUIs)، جستجو در وب، کدنویسی بازی‌ها، سازگار کردن کتابخانه‌های کلاس C++ و بسیاری از کارهای دیگر استفاده می‌کنند. این زبان قابل حمل در سیستم عامل‌های مختلفی از جمله Windows، Unix، OS/2 و ... قابل اجرا است.

علیرغم اینکه دانشجویان معمولاً از زبان‌های ایستایی چون پاسکال، C و زیر مجموعه‌ای از C++ یا جاوا برای شروع برنامه‌نویسی استفاده می‌کنند، به اعتقاد ما بهتر است پایتون را به‌عنوان اولین زبان برنامه‌نویسی خود انتخاب کنند، چراکه پایتون نحوه نگارش ساده‌تر، منظم‌تر و باقاعده‌تری دارد و کتابخانه وسیعی نیز با آن همراه است. از همه مهم‌تر اینکه استفاده از پایتون در یک دوره برنامه‌نویسی مقدماتی به آنها اجازه می‌دهد بر روی مهارت‌های مهم برنامه‌نویسی از قبیل تجزیه مسائل و طراحی انواع داده‌ای تمرکز کنند. دانشجویان به‌وسیله پایتون می‌توانند به سرعت با مفاهیم اولیه‌ای چون حلقه‌ها و توابع آشنا شوند. آنها حتی می‌توانند با اشیاء کاربر-تعریف در همان دوره‌های اولیه کار کنند. برای نمونه قادرند یک ساختار درختی را به عنوان لیست‌های تودرتوی پایتون پیاده‌سازی کنند.

برای دانشجویی که قبلاً به هیچ عنوان برنامه‌نویسی نکرده است، استفاده از زبان ایستایی مانند C یا پاسکال، غیرطبیعی به‌نظر می‌رسد. این جریان پیچیدگی‌های اضافه‌ای به‌وجود می‌آورد که دانشجویان مجبورند با آنها دست و پنجه نرم کنند و این روند آموزش را کند می‌کند.

دیگر جنبه‌های پایتون، آن را برای استفاده به عنوان اولین زبان برنامه‌نویسی از هر نظر مناسب می‌سازد. این زبان مانند جاوا کتابخانه استاندارد جامعی دارد، به‌طوری‌که دانشجویان خیلی زود می‌توانند شروع به انجام پروژه‌های برنامه‌نویسی کنند. پروژه‌ها دیگر به طراحی ماشین حساب‌های چهار عمل محدود نمی‌شود، بلکه با استفاده از کتابخانه استاندارد پایتون، دانشجویان می‌توانند ضمن فراگیری مفاهیم بنیادی برنامه‌نویسی از نوشتن برنامه‌های کاربردی واقعی لذت ببرند.

همه این خصوصیات نشان دهنده آن است که پایتون جایگزین مناسبی برای زبان‌های ایستا به منظور آموزش برنامه‌نویسی به مبتدیان است و این ذره‌ای از اهمیت آن در جهت برنامه‌نویسی‌های پیشرفته نمی‌کاهد.

قالب کتاب حاضر را کتابی با عنوان How To Think Like a Computer Scientist تشکیل می‌دهد. این کتاب شامل ۲۰ فصل بوده و در زمینه آموزش زبان برنامه‌نویسی پایتون به مبتدیان کاربرد گسترده‌ای دارد. اما با توجه به کمبودهایی که در کتاب مزبور وجود داشت، از دیگر اسناد موجود در اینترنت در فصل بیست و یکم و دو پیوست به‌منظور جمع‌بندی، تکمیل و ارائه مطالب بیشتر استفاده گردید.

CD همراه با کتاب شامل آخرین نسخه از فایل‌های منبع مفسر پایتون (تا به امروز)، محیط‌های برنامه‌نویسی پایتون، کتاب‌ها و سایت‌های آموزشی گوناگون به زبان انگلیسی و قطعه فیلمی از نظرات برنامه‌نویسان بزرگ دنیا درباره زبان برنامه‌نویسی پایتون است که در صورت تمایل می‌توانید آن را از مراکز فروش درخواست نمایید.

در مدت تهیه این اثر از مساعدت عزیزانی بهره بردیم:

راهنمایی‌های استاد بزرگوار، دکتر احمد توحیدی خصوصاً در ویرایش واژه‌نامه‌ها ؛
دوست، مشوق و یاور همیشگی‌مان مهندس مهدی احمدی در ترجمه و ویراستاری علمی ؛
دوست هنرمندمان، روزبه مرادی در گرافیک و طراحی تصاویر ؛
زحمات جناب آقای نصرالله سیفی مدیر انتشارات کوشامهر در زمینه چاپ و توزیع ؛
و خصوصاً حمایت‌های بی‌دریغ خانواده‌های عزیزمان ؛

از تمامی این عزیزان سپاسگزاریم و از خداوند بزرگ بهروزی آنان را آرزو مندیم.

سعید خالقی

علیرضا حق‌نیا

فهرست

فصل اول

- ۱..... روش برنامه‌نویسی
- ۱-۱- زبان برنامه‌نویسی پایتون..... ۲
- ۲-۱- برنامه چیست؟..... ۴
- ۳-۱- اشکال‌زدایی چیست؟..... ۵
- ۱-۳-۱- خطاهای نحوی..... ۵
- ۲-۳-۱- خطاهای زمان اجرا..... ۶
- ۳-۳-۱- خطاهای معنایی..... ۶
- ۴-۳-۱- اشکال‌زدایی آزمایشی..... ۶
- ۴-۱- زبان‌های طبیعی و رسمی..... ۷
- ۵-۱- اولین برنامه..... ۹
- ۶-۱- واژه‌نامه..... ۹

فصل دوم

- ۱۳..... متغیرها، عبارات و دستورات.....
- ۱-۲- مقادیر و انواع داده‌ها..... ۱۴
- ۲-۲- متغیرها..... ۱۵
- ۳-۲- کلمات کلیدی و اسامی متغیرها..... ۱۶
- ۴-۲- دستورات..... ۱۷
- ۵-۲- ارزیابی عبارات..... ۱۸

۱۹	۶-۲- عملگرها و عملوندها.....
۲۰	۷-۲- ترتیب عملگرها.....
۲۱	۸-۲- عملیات بر روی رشته‌ها.....
۲۲	۹-۲- ترکیب.....
۲۲	۱۰-۲- توضیحات.....
۲۳	۱۱-۲- واژه‌نامه.....

فصل سوم

۲۷	توابع.....
۲۸	۱-۳- فراخوانی تابع.....
۲۹	۲-۳- تبدیل نوع داده.....
۳۰	۳-۳- تبدیل موقت نوع.....
۳۰	۴-۳- توابع ریاضی.....
۳۲	۵-۳- ماژول‌ها.....
۳۳	۶-۳- ترکیب.....
۳۳	۷-۳- اضافه کردن توابع جدید.....
۳۶	۸-۳- تعریف و استفاده از توابع.....
۳۶	۹-۳- روند اجرا.....
۳۷	۱۰-۳- پارامترها و آرگومان‌ها.....
۳۹	۱۱-۳- متغیرها و پارامترها محلی هستند.....
۴۰	۱۲-۳- نمودارهای پشته.....
۴۱	۱۳-۳- توابع نتیجه‌دار.....
۴۲	۱۴-۳- واژه‌نامه.....

فصل چهارم

۴۵	شرطی‌ها و بازگشت.....
۴۶	۱-۴- عملگر باقی‌مانده.....
۴۷	۲-۴- عبارات بولی.....
۴۷	۳-۴- عملگرهای منطقی.....
۴۸	۴-۴- اجرای عبارات شرطی.....

۴۹	۵-۴- اجرای انتخاب‌های دوگانه.....
۵۰	۶-۴- دستورات شرطی زنجیره‌ای.....
۵۱	۷-۴- دستورات شرطی تودرتو.....
۵۲	۸-۴- دستور return
۵۲	۹-۴- توابع بازگشتی.....
۵۴	۱۰-۴- نمودارهای پشت‌پشته برای توابع بازگشتی.....
۵۵	۱۱-۴- بازگشت بی‌انتهای.....
۵۵	۱۲-۴- ورودی صفحه‌کلید.....
۵۷	۱۳-۴- واژه‌نامه.....

فصل پنجم

۵۹	توابع نتیجه‌دار.....
۶۰	۱-۵- مقادیر برگشتی.....
۶۱	۲-۵- توسعه برنامه.....
۶۴	۳-۵- ترکیب.....
۶۵	۴-۵- توابع بولی.....
۶۶	۵-۵- بازگشت نتیجه‌دار.....
۶۹	۶-۵- جهش با اطمینان.....
۶۹	۷-۵- مثالی دیگر.....
۷۰	۸-۵- بررسی انواع داده‌ها.....
۷۲	۹-۵- واژه‌نامه.....

فصل ششم

۷۵	تکرار.....
۷۶	۱-۶- انتساب چندگانه.....
۷۷	۲-۶- دستور while
۷۹	۳-۶- جدول‌ها.....
۸۲	۴-۶- جداول دو بعدی.....
۸۲	۵-۶- بسته‌بندی و تعمیم.....

۸۴	۶-۶- یک بسته‌بندی دیگر.....
۸۴	۶-۷- متغیرهای محلی.....
۸۵	۶-۸- یک تعمیم دیگر.....
۸۷	۶-۹- توابع.....
۸۷	۶-۱۰- واژه‌نامه.....

فصل هفتم

۹۱	رشته‌ها.....
۹۲	۷-۱- نوع داده‌ای مرکب.....
۹۳	۷-۲- طول رشته.....
۹۳	۷-۳- پیمایش و حلقه for
۹۵	۷-۴- برش‌های رشته.....
۹۶	۷-۵- مقایسه رشته‌ها.....
۹۶	۷-۶- رشته‌ها تغییرناپذیرند.....
۹۷	۷-۷- یک تابع find
۹۸	۷-۸- چرخش و شمارش.....
۹۸	۷-۹- ماژول string
۹۹	۷-۱۰- طبقه‌بندی کاراکترها.....
۱۰۱	۷-۱۱- واژه‌نامه.....

فصل هشتم

۱۰۳	لیست‌ها.....
۱۰۴	۸-۱- مقادیر لیست.....
۱۰۵	۸-۲- دستیابی به اعضاء.....
۱۰۷	۸-۳- اندازه لیست.....
۱۰۷	۸-۴- عضویت لیست.....
۱۰۸	۸-۵- لیست‌ها و حلقه‌های for
۱۰۹	۸-۶- عملگرهای لیست.....
۱۰۹	۸-۷- برش‌های لیست.....
۱۱۰	۸-۸- لیست‌ها تغییرپذیرند.....

۱۱۱	۸-۹- حذف لیست.....
۱۱۱	۸-۱۰- اشیاء و مقادیر.....
۱۱۲	۸-۱۱- بدل سازی.....
۱۱۳	۸-۱۲- تکثیر لیست ها.....
۱۱۴	۸-۱۳- لیست ها به عنوان پارامتر.....
۱۱۵	۸-۱۴- لیست های تودرتو.....
۱۱۶	۸-۱۵- ماتریس ها.....
۱۱۶	۸-۱۶- رشته ها و لیست ها.....
۱۱۷	۸-۱۷- واژه نامه.....

فصل نهم

۱۱۹	چندتایی ها.....
۱۲۰	۹-۱- تغییر پذیری و چندتایی ها.....
۱۲۱	۹-۲- نسبت دهی یک چندتایی.....
۱۲۲	۹-۳- چندتایی ها به عنوان مقادیر بازگشتی.....
۱۲۳	۹-۴- اعداد تصادفی.....
۱۲۳	۹-۵- لیستی از اعداد تصادفی.....
۱۲۴	۹-۶- شمارش.....
۱۲۵	۹-۷- طبقات متعدد.....
۱۲۷	۹-۸- یک راه حل تک گذری.....
۱۲۸	۹-۹- واژه نامه.....

فصل دهم

۱۳۱	دیکشنری ها.....
۱۳۳	۱۰-۱- عملیات بر روی دیکشنری ها.....
۱۳۴	۱۰-۲- متدهای دیکشنری.....
۱۳۵	۱۰-۳- بدل سازی و کپی برداری.....
۱۳۵	۱۰-۴- ماتریس های پراکنده.....
۱۳۷	۱۰-۵- دستاوردها.....

- ۱۰-۶- اعداد صحیح بزرگ ۱۳۹
- ۱۰-۷- شمارش حروف ۱۴۰
- ۱۰-۸- واژه‌نامه ۱۴۰

فصل یازدهم

- فایل‌ها و اعتراض‌ها ۱۴۳
- ۱۱-۱- فایل‌های متنی ۱۴۶
- ۱۱-۲- نوشتن متغیرها ۱۴۸
- ۱۱-۳- دایرکتوری‌ها ۱۵۰
- ۱۱-۴- Pickling ۱۵۰
- ۱۱-۵- اعتراض ۱۵۲
- ۱۱-۶- واژه‌نامه ۱۵۴

فصل دوازدهم

- کلاس و اشیاء ۱۵۷
- ۱۲-۱- انواع ترکیبی کاربر-تعریف ۱۵۸
- ۱۲-۲- مشخصه‌ها ۱۵۹
- ۱۲-۳- وهله‌ها به عنوان پارامترها ۱۶۰
- ۱۲-۴- تشابه و وحدت ۱۶۱
- ۱۲-۵- مستطیل ۱۶۲
- ۱۲-۶- وهله‌ها به عنوان مقادیر برگشتی ۱۶۳
- ۱۲-۷- اشیاء تغییرپذیرند ۱۶۳
- ۱۲-۸- کپی‌برداری ۱۶۴
- ۱۲-۹- واژه‌نامه ۱۶۶

فصل سیزدهم

- کلاس‌ها و توابع ۱۶۹
- ۱۳-۱- زمان ۱۷۰
- ۱۳-۲- توابع محض ۱۷۱
- ۱۳-۳- تغییردهنده‌ها ۱۷۲

- ۱۳-۴- کدام بهتر است؟ ۱۷۳
- ۱۳-۵- توسعهٔ پیش‌نمونه در برابر برنامهٔ طرح‌ریزی شده ۱۷۴
- ۱۳-۶- تعمیم ۱۷۵
- ۱۳-۷- الگوریتم‌ها ۱۷۶
- ۱۳-۸- واژه‌نامه ۱۷۷

فصل چهاردهم

- ۱۷۹- کلاس‌ها و متدها ۱۷۹
- ۱۴-۱- خصوصیات شیء گرا ۱۸۰
- ۱۴-۲- `printTime` ۱۸۱
- ۱۴-۳- مثالی دیگر ۱۸۲
- ۱۴-۴- یک مثال پیچیده‌تر ۱۸۳
- ۱۴-۵- آرگومان‌های اختیاری ۱۸۴
- ۱۴-۶- متد مقداردهی اولیه ۱۸۵
- ۱۴-۷- بازگشتی به `Point` ۱۸۶
- ۱۴-۸- باردهی اضافی عملگر ۱۸۷
- ۱۴-۹- چندریختی ۱۸۹
- ۱۴-۱۰- واژه‌نامه ۱۹۱

فصل پانزدهم

- ۱۹۳- مجموعه‌های اشیاء ۱۹۳
- ۱۵-۱- ترکیب ۱۹۴
- ۱۵-۲- شیء `Card` ۱۹۴
- ۱۵-۳- مشخصه‌های کلاس و متد `__str__` ۱۹۶
- ۱۵-۴- مقایسهٔ کارت‌ها ۱۹۷
- ۱۵-۵- دسته‌های ورق ۱۹۸
- ۱۵-۶- چاپ یک دسته ورق ۱۹۹
- ۱۵-۷- بُر زدن یک دسته ورق ۲۰۱
- ۱۵-۸- حذف و تقسیم کارت‌ها ۲۰۲

۲۰۳	واژه‌نامه	۹-۱۵
-----	----------------	------

فصل شانزدهم

۲۰۵	وراثت	
۲۰۶	۱-۱۶ وراثت	
۲۰۶	۲-۱۶ یک دست کارت	
۲۰۸	۳-۱۶ توزیع کارت‌ها	
۲۰۹	۴-۱۶ چاپ یک دست کارت	
۲۱۰	۵-۱۶ کلاس CardGame	
۲۱۱	۶-۱۶ کلاس OldMaidHand	
۲۱۲	۷-۱۶ کلاس OldMaidGame	
۲۱۶	۸-۱۶ واژه‌نامه	

فصل هفدهم

۲۱۹	لیست‌های پیوندی	
۲۲۰	۱-۱۷ ارجاع‌های توکار	
۲۲۰	۲-۱۷ کلاس Node	
۲۲۲	۳-۱۷ لیست‌ها به عنوان مجموعه	
۲۲۳	۴-۱۷ لیست‌ها و بازگشت	
۲۲۴	۵-۱۷ لیست‌های نامتناهی	
۲۲۵	۶-۱۷ قضیهٔ ابهام بنیادی	
۲۲۶	۷-۱۷ تغییر دادن لیست‌ها	
۲۲۷	۸-۱۷ بسته‌سازها و کمک‌کننده‌ها	
۲۲۸	۹-۱۷ کلاس LinkedList	
۲۲۹	۱۰-۱۷ نامتغیرها	
۲۳۰	۱۱-۱۷ واژه‌نامه	

فصل هجدهم

۲۳۳	پشته‌ها	
۲۳۴	۱-۱۸ نوع داده‌ای انتزاعی	

۲۳۵	پشته	۱۸-۲
۲۳۵	پیااده‌سازی پشته‌ها با لیست‌های پایتون	۱۸-۳
۲۳۶	گذاشتن و برداشتن عناصر در پشته‌ها	۱۸-۴
۲۳۷	 postfix استفاده از یک پشته برای ارزیابی	۱۸-۵
۲۳۸	تجزیه	۱۸-۶
۲۳۸	 postfix ارزیابی روش	۱۸-۷
۲۳۹	فراهم‌گرها و مشتری‌ها	۱۸-۸
۲۴۰	واژه‌نامه	۱۸-۹

فصل نوزدهم

۲۴۳	صف‌ها	
۲۴۴	 ADT صف	۱۹-۱
۲۴۴	صف پیوندی	۱۹-۲
۲۴۶	ویژگی‌های کارکرد	۱۹-۳
۲۴۶	صف پیوندی اصلاح شده	۱۹-۴
۲۴۸	صف اولویت	۱۹-۵
۲۵۰	 Golfer کلاس	۱۹-۶
۲۵۱	واژه‌نامه	۱۹-۷

فصل بیستم

۲۵۳	درخت‌ها	
۲۵۵	ساختن درخت‌ها	۲۰-۱
۲۵۶	پیمایش درخت‌ها	۲۰-۲
۲۵۶	درخت‌های عبارت	۲۰-۳
۲۵۷	پیمایش درختی	۲۰-۴
۲۵۹	ساختن یک درخت عبارت	۲۰-۵
۲۶۳	اداره‌کردن خطاها	۲۰-۶
۲۶۴	درخت جانوران	۲۰-۷
۲۶۶	واژه‌نامه	۲۰-۸

فصل بیست و یکم

از این پس، پایتون	۲۶۹
۱-۲۱- دنباله‌ها	۲۷۰
۲-۲۱- دنباله‌های چندبعدی	۲۷۱
۳-۲۱- وارد کردن اطلاعات	۲۷۴
۴-۲۱- اشتغال لیست‌ها	۲۷۴
۵-۲۱- lambda	۲۷۵
۶-۲۱- یک تابع بازگشتی	۲۷۷
۷-۲۱- آرگومان‌های اختیاری	۲۷۸
۸-۲۱- ماتریس‌ها	۲۸۱
۹-۲۱- متغیرها و توابع اختصاصی	۲۸۵
۱۰-۲۱- ضرب ماتریس‌ها	۲۸۶
۱۱-۲۱- کاربرد else با حلقه‌ها	۲۸۷
۱۲-۲۱- واژه‌نامه	۲۸۹

پیوست الف

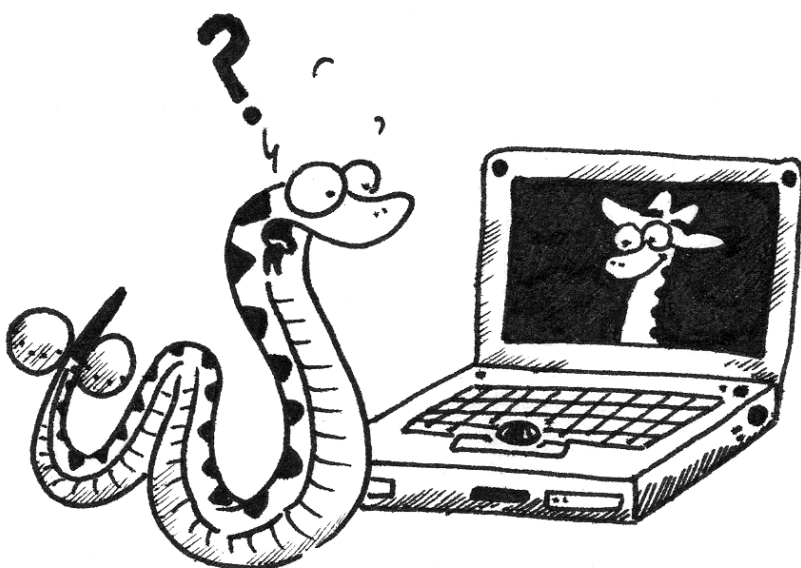
محیط‌های برنامه‌نویسی پایتون	۲۹۱
الف-۱- نحوه استفاده از IDLE	۲۹۲
الف-۱-۱- نصب و شروع به کار IDLE	۲۹۲
الف-۱-۲- استفاده از پنجره Python Shell	۲۹۳
الف-۱-۳- رنگ‌آمیزی	۲۹۴
الف-۱-۴- کنگره‌گذاری	۲۹۴
الف-۱-۵- تکمیل کلمه	۲۹۵
الف-۱-۶- تاریخچه دستورات	۲۹۵
الف-۱-۷- توضیحات در هنگام فراخوانی	۲۹۶
الف-۱-۸- استفاده از ویرایشگر فایل	۲۹۷
الف-۱-۸-۱- ویرایش یک فایل	۲۹۸
الف-۱-۸-۲- ذخیره‌سازی یک فایل	۲۹۹
الف-۱-۸-۳- اجرای یک فایل	۳۰۰

الف-۱-۹- استفاده از پنجره‌های محاوره‌ای Find/Replace	۳۰۰
الف-۱-۱۰- استفاده از مرورگر مسیر (Path Browser)	۳۰۱
الف-۲- PythonWin	۳۰۲
الف-۲-۱- پنجره تعاملی	۳۰۲
الف-۲-۲- منوها و نوار ابزار	۳۰۳

پیوست ب

خطاهای برنامه‌نویسی	۳۰۵
ب-۱- خطاهای نحوی	۳۰۶
ب-۲- اعتراض‌ها	۳۰۶
ب-۳- کنترل اعتراض‌ها	۳۰۷
ب-۴- تولید اعتراض‌ها	۳۱۰
ب-۵- اعتراض‌های کاربر-تعریف	۳۱۰
ب-۶- تعریف اعمال پایانی	۳۱۱
فهرست منابع	۳۱۳

روش برنامه نویسی



هدف کتاب حاضر این است که به شما بیاموزد چگونه مانند یک متخصص کامپیوتر فکر کنید. این روش تفکر برخی از بهترین صور ریاضیات، مهندسی و علوم طبیعی را در هم می آمیزد. مانند ریاضی دانان، متخصصان کامپیوتر از یک زبان رسمی برای مشخص کردن ایده ها (خصوصاً در محاسبات) استفاده می کنند. آنها مانند مهندسين چیزهایی را طراحی می کنند و اجزا را در سیستم ها ترکیب و مقادیر را ارزیابی می کنند. آنها مانند دانشمندان، قواعد سیستم های پیچیده و فرضیه ها را رعایت می کنند و پیشگویی ها را آزمایش می نمایند.

مهمترین مهارت یک متخصص کامپیوتر حل مسئله است. حل مسئله یعنی توانایی فرمول بندی مشکلات، تفکر خلاقانه درباره راه حل ها و بیان یک راه حل واضح و دقیق. بنابراین یادگیری برنامه نویسی فرصتی عالی برای کسب مهارت در حل مسائل است. این دلیل نام گذاری فصل حاضر با عنوان «روش برنامه نویسی» است. در قدم اول نوشتن برنامه را به عنوان یک مهارت مفید می آموزید و در قدم بعد شما از برنامه به منظور دستیابی به یک هدف استفاده می کنید. هرچه پیش تر رویم این هدف واضح تر می شود.

۱-۱- زبان برنامه نویسی پایتون

پایتون نمونه ای از یک **زبان سطح بالا** است. از دیگر زبان های سطح بالا که ممکن است تا به حال شنیده باشید، می توان بیسیک، پاسکال، فورترن و C را نام برد.

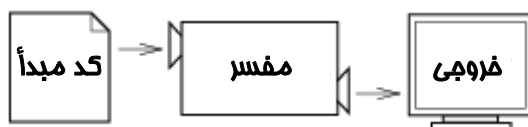
همان طور که ممکن است شما هم انتظار داشته باشید، در مقابل واژه «**زبان سطح بالا**»، «**زبان سطح پایین**» هم وجود دارد که برای نمونه می توان زبان اسمبلی را نام برد.

کامپیوترها تنها برنامه هایی را می توانند اجرا کنند، که به زبان ماشین تبدیل شده باشند. بدین سان برنامه هایی که در یک زبان سطح بالا نوشته شده اند، باید قبل از اجرا پردازش شوند و به زبان ماشین ترجمه شوند. این پردازش اضافه مدتی زمان می برد، که این اشکال کوچک زبان های سطح بالا است.

اما مزایای این نوع زبان ها بسیارند: اول اینکه برنامه نویسی در زبان های سطح بالا به مراتب ساده تر است. در نوشتن برنامه ها به زبان های سطح بالا وقت کمتری مصرف می شود و خواندن آنها ساده تر و سریع تر انجام می گیرد. اشکال زدایی آنها راحت تر است و به زبان محاوره هم نزدیک ترند. دوم اینکه، زبان های سطح بالا **قابل حمل** هستند، به این معنی که آنها می توانند روی انواع کامپیوترها اجرا شوند بدون اینکه نیازی به ویرایش و تغییر داشته باشند. برنامه های سطح پایین تنها بر روی نوع خاصی از ماشین ها قابل اجرا هستند و برای اجرا شدن روی انواع دیگر نیاز به بازنویسی دارند.

با توجه به این مزایا تقریباً همه برنامه‌ها در زبان‌های سطح بالا نوشته می‌شوند. زبان‌های سطح پایین تنها در موارد خاصی کاربرد دارند.

دو دسته از برنامه‌ها که زبان‌های سطح بالا را به زبان‌های سطح پایین پردازش می‌کنند **مفسرها** و **کامپایلرها** هستند. یک مفسر، برنامه سطح بالا را می‌خواند و اجرا می‌کند، بدین معنی که مفسر آنچه را که برنامه می‌گوید انجام می‌دهد. مفسر برنامه را خط به خط می‌خواند و محاسبات را انجام می‌دهد:



یک کامپایلر برنامه را می‌خواند و قبل از اینکه اجرا کند آن را به‌طور کامل به زبان ماشین ترجمه می‌کند. در این مورد برنامه سطح بالا **کد مبدأ** و برنامه ترجمه شده را **کد مقصد** یا **قابل اجرا** می‌نامند. یک بار که برنامه کامپایل شود، می‌توانید آن را بدون ترجمه مجدد به دفعات اجرا کنید:



پایتون یک زبان تفسیری معرفی شده است، زیرا برنامه‌های پایتون به‌وسیله مفسر اجرا می‌شوند. دو راه برای استفاده از مفسر وجود دارد: حالت خط فرمان و حالت اسکریپت. در حالت خط فرمان، شما برنامه‌های پایتون را تایپ می‌کنید و مفسر نتیجه را چاپ می‌کند:

```

Python 2.3 (#46, Jul 29 2003, 18:54:32)
[MSC v.1200 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information
IDLE 1.0
>>> print 1+1
2
  
```

دو خط اول این مثال، فرمانی است که مفسر پایتون را اجرا می‌کند. دو خط بعد پیغام‌هایی از سوی مفسر هستند. خط پنجم با علامت **>>>** شروع می‌شود که مفسر پایتون از این علامت به‌عنوان

اعلان استفاده می‌کند. ما دستور `print 1+1` را تایپ می‌کنیم و پس از فشردن کلید **Enter** مفسر 2 را جواب می‌دهد.

دیگر اینکه شما می‌توانید برنامه را در یک فایل بنویسید و از مفسر برای اجرای محتویات فایل استفاده کنید. چنین فایلی را اسکریپت می‌نامند. برای مثال ما از یک ویرایشگر متنی جهت ساختن فایلی با عنوان `latoya.py` و مضمون زیر استفاده می‌کنیم:

```
print 1+1
```

به‌عنوان قرارداد، فایل‌های پایتون پسوند `.py` دارند. برای اجرای برنامه باید نام اسکریپت را به مفسر اعلان کنیم:

```
$ python latoya.py
2
```

در دیگر محیط‌های پیشرفته جزئیات اجرای برنامه‌ها ممکن است متفاوت باشد. همچنین اغلب برنامه‌ها جذاب‌تر از این هستند. اغلب مثال‌های این کتاب در خط فرمان اجرا شده‌اند. کار کردن در خط فرمان جهت آزمایش و توسعه برنامه راحت‌تر و مناسب‌تر است، زیرا شما می‌توانید برنامه‌ها را تایپ و بلافاصله اجرا کنید. وقتی که یک برنامه صحیح را در حالت خط فرمان به‌دست آوردید، می‌توانید آن را در یک اسکریپت ذخیره کنید و از این پس آن را ویرایش یا اجرا کنید.

۱-۲- برنامه چیست؟

یک **برنامه**، دنباله‌ای از دستورات است که چگونگی انجام محاسبات را مشخص می‌کند. محاسبات می‌توانند از نوع ریاضی مانند حل دستگاه معادلات، به‌دست آوردن ریشه‌های یک چند جمله‌ای یا محاسبات نمادین از قبیل یافتن و جایگزین کردن کلمه در یک متن و ... باشند. جزئیات در زبان‌های مختلف، متفاوت ظاهر می‌شوند، اما تعدادی اصول پایه‌ای در همه زبان‌ها به یک صورت هستند:

ورودی: گرفتن داده‌ها از صفحه کلید، فایل یا دیگر واحدهای ورود اطلاعات.

۱- این دستور در سیستم‌عامل یونیکس کاربرد دارد. برای اجرای اسکریپت‌ها در محیط ویندوز به پیوست الف مراجعه نمایید.

خروجی: نمایش داده‌ها بر روی صفحه نمایش یا ارسال آنها به یک فایل یا دیگر واحدهای خروج اطلاعات.

عملیات ریاضی: انجام دادن اعمال ریاضی بنیادی مانند ضرب و جمع.

تصمیم‌گیری: بررسی شروط خاص و اجرای دنباله‌ای از دستورات بر اساس آن شرایط.

تکرار: انجام برخی اعمال در چندین مرتبه و معمولاً با تغییری در عمل مورد تکرار.

هر برنامه‌ای که تا به حال استفاده نمودید بدون توجه به پیچیدگی آنها از دستوراتی تشکیل شده است که ممکن است شبیه به اینها نباشد، بنابراین می‌توان برنامه‌نویسی را چنین شرح داد: «شکستن یک عمل بزرگ و پیچیده به عملیات کوچک‌تر و ساده‌تر؛ این کار تا آنجا صورت می‌گیرد که این عملیات به صورت دستورات ساده و بنیادی بتوانند مورد استفاده قرار گیرند.» این توضیح ممکن است کمی مبهم باشد، اما پس از بحث دربارهٔ **الگوریتم‌ها**، بیشتر به این موضوع می‌پردازیم.

۳-۱- اشکال‌زدایی چیست؟

برنامه‌نویسی فرایندی پیچیده است که چون به وسیلهٔ انسان ساخته می‌شود، اغلب دارای خطا است. خطاهای برنامه‌نویسی را **Bug** می‌نامند و به عمل جداسازی و تصحیح آن خطاها **Debugging** گفته می‌شود.

در یک برنامه امکان دارد سه نوع خطا رخ دهد:

- ۱- خطاهای نحوی
- ۲- خطاهای زمان اجرا
- ۳- خطاهای معنایی

فراگیری تفاوت‌های این سه نوع خطا بسیار مفید است، زیرا در آینده به راحتی می‌توان آنها را در برنامه یافت و تصحیح کرد.

۳-۱-۱ خطاهای نحوی

پایتون تنها قادر است تنها برنامه‌هایی را اجرا کند که از لحاظ **نحوه نگارش** صحیح باشند و گرنه پردازش متوقف می‌گردد و یک پیغام خطا برگردانده می‌شود. نحوه نگارش، ساختار برنامه و قوانینی در مورد این ساختار را بازگو می‌کند. برای مثال در زبان انگلیسی جملات با حرف بزرگ شروع می‌شوند و با یک نقطه پایان می‌یابند. مثلاً جملهٔ **"this is a book."** دارای یک خطای نحوی می‌باشد، زیرا حرف اول این جمله کوچک نوشته شده است.

برای اغلب خوانندگان، چند خطای کوچک نحوی مشکل چندانی محسوب نمی‌شود اما پایتون از این مطلب به سادگی نمی‌گذرد. حتی اگر تنها یک اشکال نحوی -به هر علت- در برنامه وجود داشته باشد، پایتون یک پیغام خطا چاپ می‌کند و خارج می‌شود و شما قادر به اجرای برنامه نیستید. در هفته‌های اول که برنامه‌نویسی را شروع می‌کنید، ممکن است وقت زیادی را صرف یافتن و تصحیح خطاهای نحوی کنید، اما همین که مهارت کافی در برنامه‌نویسی پیدا کردید، مرتکب اشتباهات کمتری می‌شوید و آنها را سریع‌تر می‌یابید.

۱-۳-۲- خطاهای زمان اجرا

دومین نوع خطا، **خطاهای زمان اجرا** هستند و علت این نام‌گذاری آن است که تا برنامه اجرا نشود، آنها ظاهر نمی‌شوند. این خطاها را **اعتراض** هم می‌نامند زیرا آنها معمولاً نشان می‌دهند که اتفاق اعتراض‌آمیز و بدی رخ داده است. خطاهای زمان‌اجرا در برنامه‌های ساده‌ای که در فصل‌های اول می‌بینید، کم‌تعدادند. بنابراین ممکن است مدتی بگذرد تا با یکی از این خطاها روبرو شوید.

۱-۳-۳- خطاهای معنایی

سومین نوع خطا، **خطاهای معنایی** هستند. اگر یک خطای معنایی در برنامه شما وجود داشته باشد برنامه با موفقیت اجرا خواهد شد؛ به این معنی که کامپیوتر هیچ پیغام خطایی تولید نمی‌کند، اما عمل درست را هم انجام نمی‌دهد. کامپیوتر دقیقاً کاری را انجام می‌دهد که شما به آن گفته‌اید. مشکل آنجا است که برنامه نوشته شده مقصودتان را بیان نکرده و **معنای** برنامه غلط است. تشخیص یک خطای معنایی احتیاج به مهارت دارد، زیرا نیازمند آن است که شما به عقب برگردید و مجدداً با نگاه کردن به خروجی برنامه سعی کنید بفهمید چه اتفاقی افتاده است.

۱-۳-۴- اشکال‌زدایی آزمایشی

یکی از مهمترین مهارت‌هایی که شما به‌دست می‌آورید، **اشکال‌زدایی** است. اگرچه ممکن است ناامیدکننده باشد، اما اشکال‌زدایی یکی از مبارزه‌طلبی‌های فکری باشکوه و جذاب برنامه‌نویسی است.

به بیان دیگر، اشکال‌زدایی شبیه به عمل یک کارآگاه است. شما با سرنخ‌ها مواجه‌اید و مجبورید فرایندها و وقایع را بر اساس نتایجی که می‌بینید استنتاج کنید.

اشکال‌زدایی، همانند یک علم تجربی است. به محض اینکه شما ایده‌ای در مورد اشکال کار به‌دست می‌آورید، برنامه را تصحیح کرده و دوباره تلاش می‌کنید. اگر فرضیه شما درست باشد، آنگاه می‌توانید نتیجه تغییر و تحول را پیش‌گویی کنید و یک قدم به برنامه قابل اجرا و صحیح نزدیک‌تر شوید، اما اگر فرضیه شما غلط باشد مجبورید ایده جدیدی ارائه دهید.

همان‌طور که «شرلوک هلمز» اشاره کرده است: «وقتی غیرممکن را از معادله حذف می‌کنید، آنچه باقی می‌ماند هر چقدر هم که غیرمنتظره باشد، حقیقت است.»

برای برخی از مردم، برنامه‌نویسی و اشکال‌زدایی یک چیز هستند، به این‌صورت که برنامه‌نویسی به عمل اشکال‌زدایی گام به گام گفته می‌شود. این عمل تا آنجا ادامه می‌یابد که برنامه آنچه را که می‌خواهیم انجام دهد.

منظور این است که شما برنامه‌ای را که عمل خاصی انجام می‌دهد شروع کنید، اصطلاحات جزئی و اشکال‌زدایی را انجام دهید تا به یک برنامه کامل و عملی دست یابید.

برای نمونه «لینوکس»، سیستم عاملی است که شامل هزاران خط کد است اما همه چیز از یک برنامه ساده «لینوس توروالدز» آغاز شد که تراشه **Intel 80386** را بررسی می‌کرد. بر طبق گفته «لری گرینفیلد»: «یکی از پروژه‌های ابتدایی لینوس برنامه‌ای بود که به صورت متناوب **AAAA** و **BBBB** را چاپ می‌کرد. تکمیل این پروژه در آینده منجر به ساخت لینوکس شد.»

فصل‌های بعد پیشنهادات بیشتری در مورد اشکال‌زدایی و دیگر تمرین‌های برنامه‌نویسی به شما ارائه می‌دهد.

۱-۴- زبان‌های طبیعی و رسمی

زبان‌های طبیعی، زبان‌هایی هستند که مردم به‌وسیله آنها صحبت می‌کنند، از قبیل انگلیسی، فارسی، اسپانیولی و... آنها توسط اشخاص بخصوصی طراحی نشده‌اند بلکه در بستر فکر مردمان مختلف شکل گرفته‌اند.

زبان‌های رسمی، زبان‌هایی هستند که توسط اشخاص ویژه‌ای طراحی شده‌اند. برای نمونه، آن نمادگذاری که ریاضی‌دانان استفاده می‌کنند یک زبان رسمی است که ارتباطات بین نمادها و اعداد را به‌طور خاصی مشخص می‌کند. شیمی‌دان‌ها از یک زبان رسمی برای نشان دادن ساختار شیمیایی مولکول‌ها استفاده می‌نمایند و از همه مهم‌تر اینکه زبان‌های برنامه‌نویسی، زبان‌هایی رسمی هستند که برای بیان محاسبات طراحی شده‌اند.

زبان‌های رسمی دارای قوانین دشواری درباره نحوه نگارش هستند. برای مثال، $3+3=6$ دارای یک نحوه نگارش صحیح در دستورات ریاضی است، اما $3+=6\$$ این طور نیست. **H20** یک نحوه نگارش صحیح در شیمی است، اما **2zz** اینطور نیست.

قوانین نحوی دو نوع هستند، که یکی مربوط به **توکن‌ها** و دیگری مربوط به ساختار عبارات است. توکن‌ها، عناصر اولیه زبان هستند، مانند کلمات، اعداد و عناصر شیمیایی. یکی از مشکلات $3+=6\$$ این است که **\$** (تا آنجا که ما می‌دانیم) یک عنصر قانونی در ریاضیات نیست. همین‌طور **2zz** مجاز نیست، زیرا هیچ عنصری با نام اختصاری **zz** وجود ندارد.

دومین نوع خطاهای نحوی مربوط به ساختار یک عبارت است که نحوه آرایش توکن‌ها است. عبارت $3+=6\$$ دارای ساختاری غیرمجاز است، زیرا شما نمی‌توانید علامت **=** را بلافاصله بعد از علامت **+** استفاده کنید. به‌طور مشابه فرمول‌های مولکولی باید دارای اندیسی بعد از نام عنصر باشند نه قبل از نام آن.

وقتی جمله‌ای را به زبان انگلیسی و یا عبارتی را به یک زبان رسمی می‌خوانید، باید ساختار جمله را تحلیل کنید؛ اگرچه این کار را به‌طور ناخودآگاه در زبان طبیعی انجام می‌دهید. این پردازش را تجزیه می‌نامند.

برای مثال وقتی که شما جمله «گل پژمرد» را می‌شنوید، در می‌یابید که گل فاعل است و پژمرد، فعل. یکبار که جمله را تجزیه کنید منظور آن را متوجه می‌شوید. فرض کنید شما گل را می‌شناسید و معنی پژمردن را نیز می‌دانید، بنابراین شما معنی جمله را متوجه خواهید شد. اگرچه زبان‌های رسمی و طبیعی خصوصیات مشترک زیادی دارند (توکن‌ها، ساختار، نحوه و معنا)، اما تفاوت‌های بسیاری میان آنها است:

ابهام: زبان‌های طبیعی پر از ابهاماتی است که مردم به‌وسیله نشانه‌های مفهومی یا اطلاعات دیگر آنها را درک و استفاده می‌کنند. زبان‌های رسمی برای این طراحی شده‌اند که دوپهلو نباشند و هر عبارت صرف‌نظر از **مفهوم** دقیقاً دارای یک معنی باشند.

افزونگی (اطناب): در زبان‌های طبیعی جهت جبران کردن ابهام و کاهش نامفهومی‌ها، از افزونگی در جملات استفاده می‌شود؛ در عوض زبان‌های رسمی از ایجاز بیشتری برخوردارند.

لفظ: زبان‌های طبیعی پر از اصطلاح و کنایه‌اند. اگر بگوییم «گل پژمرد» شاید منظور ما نه «گل» باشد و نه «پژمردن». اما عبارات در زبان‌های رسمی دقیقاً همان معنی را می‌دهند که گفته می‌شوند.

برنامه: معنی یک برنامه کامپیوتری بدون ابهام و دقیق است و می‌تواند به‌طور کامل و دست‌نخورده توسط تجربه و تحلیل نشانه‌ها و ساختار درک شود.

در اینجا به چند پیشنهاد برای خواندن برنامه‌ها (و دیگر زبان‌های رسمی) توجه کنید. اول اینکه به‌خاطر داشته باشید که زبان‌های رسمی خیلی متراکم‌تر و فشرده‌تر از زبان‌های طبیعی هستند و لذا برای خواندن بیشتر وقت می‌برند. همچنین ساختار عبارت‌ها خیلی مهم است، بنابراین خواندن بالا به پایین یا چپ به راست آنها ایده خوبی نیست. در عوض یاد بگیرید که برنامه را در ذهنتان تجزیه کنید، نشانه‌ها را تشخیص دهید و ساختار را تفسیر نمایید.

همچنین چیزهای کوچکی مانند اشتباهات لفظی و نشانه‌گذاری غلط که شما می‌توانید در زبان‌های طبیعی آنها را نادیده بگیرید، ممکن است در زبان‌های رسمی خطاهای بزرگی محسوب شوند.

۵-۱- اولین برنامه

به‌طور معمول اولین برنامه‌ای که در یک زبان جدید نوشته می‌شود، "Hello, World!" نام دارد، زیرا تمام کاری که انجام می‌دهد نمایش کلمات Hello, World! است. در پایتون این برنامه به‌صورت زیر است:

```
print "Hello, World!"
```

این مثالی برای **دستور چاپ** است که در حقیقت چیزی را بر روی کاغذ چاپ نمی‌کند بلکه تنها مقداری را بر روی صفحه تصویر نشان می‌دهد. در این مثال نتیجه چاپ کلمات زیر است:

```
Hello, World!
```

علامت کوتیشن در برنامه شروع و پایان مقدار را مشخص می‌کند و در نتیجه برنامه (خروجی) ظاهر نمی‌شوند.

بعضی از مردم در مورد کیفیت یک زبان برنامه‌نویسی بر اساس سادگی برنامه "Hello, World!" قضاوت می‌کنند. با این استاندارد، پایتون زبان خوبی است.

۶-۱- واژه‌نامه

problem solving (حل مسئله)

فرایند فرمول‌بندی یک مسئله، یافتن و ارائه راه‌حل آن.

high-level language (زبان سطح بالا)

یک زبان برنامه‌نویسی مانند پایتون که برای راحتی انسان‌ها در خواندن و نوشتن برنامه‌ها طراحی شده است.

low-level language (زبان سطح پایین)

یک زبان برنامه‌نویسی که جهت سادگی اجرا برای کامپیوتر طراحی شده است و همچنین "زبان ماشین" یا "اسمبلی" هم نامیده می‌شود.

portable (قابل حمل)

خاصیت برنامه‌ای که می‌تواند روی بیش از یک نوع ماشین اجرا شود.

interpret (تفسیر)

اجرا کردن برنامه در یک زبان سطح بالا به وسیله ترجمه خط به خط آن.

compile (کامپایل، ترجمه)

ترجمه یکباره برنامه‌ای نوشته شده در زبان سطح بالا به زبان سطح پایین، جهت آماده‌سازی برای اجراهای بعدی.

source code (کد مبدأ)

یک برنامه نوشته شده به زبان سطح بالا قبل از کامپایل شدن.

object code (کد مقصد)

خروجی یک کامپایلر بعد از ترجمه برنامه.

executable (قابل اجرا)

نام دیگری برای کد مقصدی که آماده اجرا است.

script (اسکریپت)

برنامه ذخیره شده در یک فایل (معمولاً برنامه‌ای که می‌خواهد تفسیر شود).

program (برنامه)

یک سری از دستورات که محاسباتی را شرح می‌دهند.



algorithm (الگوریتم)

یک فرایند کلی برای حل کردن دسته‌ای از مسائل.

bug (اشکال)

خطایی در برنامه.

debugging (اشکال‌زدایی)

فرایند یافتن و برطرف کردن هر سه نوع خطاهای برنامه‌نویسی.

syntax (نحوه نگارش)

ساختار دستوری یک برنامه.

syntax error (خطای نحوی)

خطایی در برنامه که آن را برای تجزیه (و بنابراین برای تفسیر) غیرممکن می‌سازد.

runtime error (خطای زمان اجرا)

خطایی که تا وقتی برنامه شروع به اجرا نشود اتفاق نمی‌افتد، اما از ادامه اجرای آن جلوگیری می‌کند.

exception (اعتراض)

نامی دیگر برای خطاهای زمان اجرا.

semantic error (خطای معنایی)

خطایی که باعث می‌شود برنامه، عملی برخلاف خواسته برنامه‌نویس انجام دهد.

semantics (مفهوم)

مقصد و هدف یک برنامه.

natural language (زبان طبیعی)

هریک از زبان‌هایی که مردم با آن صحبت می‌کنند و به طور طبیعی نمو پیدا کرده‌اند.

formal language (زبان رسمی)

هریک از زبان‌هایی که مردم برای اهداف خاصی طراحی کرده‌اند، از قبیل نمایش تئوری‌های ریاضی و یا برنامه‌های کامپیوتری. همهٔ زبان‌های برنامه‌نویسی زبان رسمی هستند.

token (توکن، نشانه، نماد)

یکی از عناصر ابتدایی ساختار نحوی یک برنامه، مانند کلمه در زبان طبیعی.

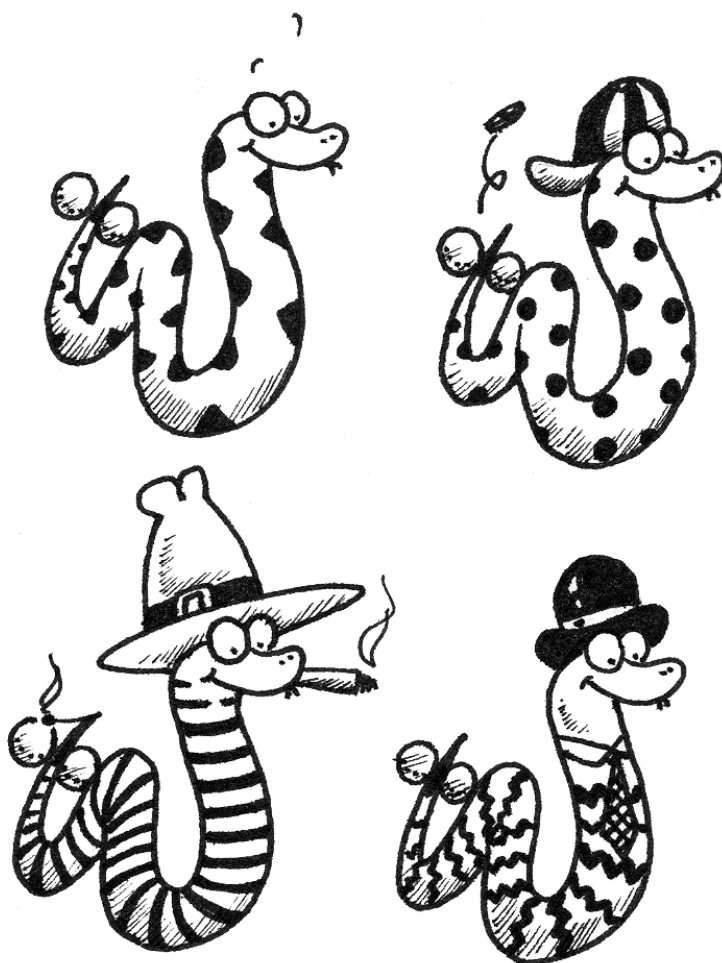
parse (تجزیه)

بررسی یک برنامه و تحلیل ساختار نحوی آن.

print statement (دستور چاپ)

دستوری که باعث می‌شود مفسر پایتون مقداری را بر روی صفحهٔ نمایش نشان دهد.

متغیرها، عبارات و دستورات



در این فصل با بررسی عناصر پایه‌ای زبان و آشنایی با خصیصه‌هایی چون متغیرها، عبارات و دستورات، برنامه‌نویسی را آغاز می‌کنیم.

هر زبان انواع داده‌ای مشخصی دارد که در افزایش قدرت برنامه‌نویسی با آن زبان بسیار مؤثرند. آشنایی با این انواع به منظور بهره‌گیری مطلوب از توانایی‌های آن زبان امری ضروری است. در این فصل پس از معرفی موارد مذکور به شرح قوانین اولویت در عملگرها، اعمال بر روی انواع داده‌ای و چگونگی ترکیب دستورات در زبان پایتون می‌پردازیم.

۲-۱- مقادیر و انواع داده‌ها

یک مقدار، یکی از موجودیت‌های بنیادی - مانند یک حرف یا یک عدد - است که برنامه با آن کار می‌کند. مقداری که تا اینجا دیدیم 2 (یعنی نتیجه جمع 1+1) و "Hello, World!" هستند.

این مقادیر مربوط به دو نوع مختلف داده‌ها هستند. 2 یک عدد صحیح و "Hello, world!" یک رشته است. منظور از رشته تعدادی از حروف است که پشت سر هم (به صورت یک زنجیر) قرار گرفته‌اند. شما (و همچنین مفسر) می‌توانید رشته‌ها را به راحتی تشخیص دهید زیرا آنها در بین دو علامت جفت کوتیشن (") و یا کوتیشن (') قرار می‌گیرند.

دستور چاپ بر روی اعداد صحیح نیز کار می‌کند:

```
>>> print 4
4
```

اگر شما مطمئن نیستید که چه نوع داده‌ای دارید، مفسر می‌تواند به شما نوع داده را بگوید:

```
>>> type("Hello, World!")
<type 'string'>
>>> type(17)
<type 'int'>
```

رشته‌ها به نوع `str` و اعداد صحیح به نوع `int` تعلق دارند. اعدادی با یک ممیز (که در زبان انگلیسی با یک نقطه میان اعداد نشان داده می‌شوند) به نوعی تحت عنوان `float` تعلق دارند، که این اعداد قالبی برای نمایش اعداد اعشاری هستند.

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

مقادیری شبیه به "17" و "3.2" از کدام نوع داده‌ای هستند؟ آنها شبیه اعداد هستند اما اگر دقت کنید مانند رشته‌ها میان جفت کوتیشن قرار گرفته‌اند.

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

لذا آنها رشته هستند.

وقتی که شما یک عدد صحیح بزرگ را تایپ می‌کنید ممکن است به ازای هر سه رقم از یک علامت ' , ' (کاما) استفاده کنید تا برای خواندن آسان‌تر شوند مانند 1,000,000. برای پایتون این به عنوان یک عدد قانونی نیست اما در حالت کلی، عبارتی معتبر است:

```
>>> print 1,000,000
1 0 0
```

خوب، این به هیچ وجه پاسخی نیست که ما انتظار داشتیم. مفسر پایتون 1,000,000 را به عنوان لیستی شامل سه عضو چاپ می‌کند. بنابراین به خاطر داشته باشید که در بین این گونه اعداد نباید از کاما استفاده کنید.

۲-۲- متغیرها

یکی از توانمندترین خصوصیات یک زبان برنامه‌نویسی توانایی کار با **متغیرها** است. متغیر نامی است که به یک مقدار اشاره می‌کند.

دستور نسبت‌دهی یک متغیر جدید می‌سازد و مقداری را به آن نسبت می‌دهد:

```
>>> message = "What's up, Doc?"
>>> n = 17
>>> pi = 3.14159
```

این مثال منجر به سه مورد انتساب می‌شود. مورد اول رشته "What's up Doc?" را به یک متغیر جدید به نام **message** نسبت می‌دهد. دومین عبارت عدد صحیح 17 را به **n** می‌دهد و مورد سوم عدد اعشاری 3.14159 را به **pi** می‌دهد.

یک روش معمول برای نشان دادن متغیرها بر روی کاغذ، نوشتن نام آنها و رسم یک پیکان است که به مقدار اختصاص داده شده اشاره می‌کند. این گونه نمودارها را **نمودار وضعیت** می‌نامند زیرا نشان می‌دهد که هر متغیر چه وضعیتی دارد. نمودار ۱-۲ نتیجه دستور نسبت‌دهی را نشان می‌دهد.

message	→	"What's up, Doc?"
n	→	17
pi	→	3.14159

شکل ۱-۲

دستور چاپ برای متغیرها نیز کار می‌کند.

```
>>> print message
What's up, Doc?
>>> print n
17
>>> print pi
3.14159
```

در هر مورد، نتیجه مقدار متغیر است. متغیرها نیز انواعی دارند و ما باز هم می‌توانیم از مفسر بپرسیم که نوع متغیر چیست.

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

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

۲-۳- کلمات کلیدی و اسامی متغیرها

برنامه‌نویسان معمولاً اسامی معنی‌داری برای متغیرها انتخاب می‌کنند. آنها با این کار نشان می‌دهند که متغیر به چه منظور استفاده شده است.

اسامی متغیرها می‌توانند تا هر اندازه طولانی باشند. آنها می‌توانند شامل اعداد و هم حروف باشند اما باید با یک حرف شروع شوند. اگر چه استفاده از حروف بزرگ هم قانونی است اما بنا به قرارداد ما چنین کاری نمی‌کنیم. اگر شما چنین عملی انجام دادید دقیقاً به خاطر داشته باشید که از چه نوع حروفی (کوچک یا بزرگ) استفاده کرده‌اید. **Bruce** و **bruce** دو متغیر متفاوتند.

کاراکتر خط‌زیر (_) نیز می‌تواند در نام متغیرها استفاده شود. این کاراکتر در اسم‌هایی که از چند کلمه تشکیل شده‌اند به کار می‌رود از قبیل **my_name** یا **price_of_tea_in_china**.

اگر شما به متغیری نامی غیرمجاز بدهید یک خطای نحوی دریافت خواهید کرد:

```
>>> 76trombones = "big parade"
SyntaxError: invalid syntax
>>> more$ = 1000000
SyntaxError: invalid syntax
>>> class = "Computer Science 101"
SyntaxError: invalid syntax
```

76trombones نامی غیرمجاز است زیرا با یک حرف شروع نشده است. **more\$** غیرمجاز است زیرا شامل یک کاراکتر غیرقانونی (\$) است. اما چرا **class** غلط است؟ این خطا نشان می‌دهد که **class** یکی از کلمات کلیدی پایتون است. کلمات کلیدی ساختار و قوانین زبان را تعریف می‌کنند و نمی‌توانند به عنوان اسامی متغیرها استفاده شوند.

پایتون ۲۸ کلمهٔ کلیدی دارد:

and	continue	else	for	import	not	raise
assert	def	except	from	in	or	return
break	del	exec	global	is	pass	try
class	elif	finally	if	lambda	print	while

ممکن است بخواهید این لیست را در دسترس نگه دارید. اگر مفسر در مورد نام یکی از متغیرها ایراد گرفت و شما علت آن را نمی‌دانستید، به این لیست نگاه کنید تا ببینید نام مورد نظر در این لیست وجود دارد یا خیر.

۲-۴- دستورات

یک دستور عبارتی است که مفسر پایتون قادر است آن را اجرا کند. تا به حال ما دو نوع از دستورات را دیده‌ایم: چاپ و نسبت‌دهی.

وقتی دستوری را در خط فرمان تایپ می‌کنید، پایتون آن را اجرا می‌کند و اگر نتیجه‌ای وجود داشته باشد آن را نمایش می‌دهد. نتیجه دستور چاپ یک مقدار است اما دستورات نسبت‌دهی نتیجه‌ای تولید نمی‌کنند.

یک اسکریپت معمولاً شامل دنباله‌ای از دستورات است. اگر بیش از یک دستور وجود داشته باشد، نتیجه پس از اجرا شدن همه دستورات یکباره نمایش داده می‌شود.
برای مثال اسکریپت صفحه بعد:

```
print 1
x = 2
print x
```

خروجی زیر را تولید می‌کند:

```
1
2
```

می‌بینیم که دستور نسبت‌دهی هیچ خروجی تولید نمی‌کند.

۲-۵- ارزیابی عبارات

یک عبارت ترکیبی از مقادیر، متغیرها و عملگرها است. اگر شما عبارتی را در خط فرمان تایپ کنید، مفسر آن را ارزیابی می‌کند و نتیجه را نمایش می‌دهد.

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

یک مقدار خود به تنهایی یک عبارت محسوب می‌شود و برای یک متغیر نیز چنین است.

```
>>> 17
17
>>> x
2
```

ارزیابی یک عبارت به‌طور کامل شبیه به چاپ یک مقدار نیست.

```
>>> message = "What's up, Doc?"
>>> message
"What's up, Doc?"

>>> print message
What's up, Doc?
```

وقتی پایتون مقدار یک عبارت را نمایش می‌دهد از قالبی مشابه آنچه برای مقداردهی به کار برده‌اید، استفاده می‌کند. مثلاً برای رشته‌ها از علامت کوتیشن یا جفت کوتیشن استفاده می‌شود اما دستور **print** مقدار عبارت را چاپ می‌کند که در این مثال شامل محتوای رشته است. در یک اسکریپت، یک عبارت خود به تنهایی دستوری مجاز است، اما هیچ کاری انجام نمی‌دهد.

مثلاً اسکریپت زیر:

```
17
3.2
"Hello, World!"
1 + 1
```

به هیچ وجه خروجی تولید نمی‌کند. چگونه می‌توانید اسکریپت را طوری تغییر دهید که مقادیر این چهار عبارت را نمایش دهد؟

۲-۶- عملگرها و عملوندها

عملگرها نمادهای ویژه‌ای هستند که محاسباتی همچون جمع و ضرب را نمایش می‌دهند. مقادیری که عملگرها استفاده می‌کنند **عملوند** نامیده می‌شوند. عبارات زیر که کمابیش معنی آنها واضح است، همگی در پایتون مجازند:

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

نمادهای +، - و / و نحوه استفاده پرانتزها برای دسته‌بندی، همان معنی را می‌دهند که در ریاضیات از آنها استفاده می‌شود. ستاره (*) نمادی است برای ضرب کردن و دو ستاره (**) نمادی است برای توان‌رسانی.

وقتی که نام یک متغیر به جای یک عملوند ظاهر می‌شود قبل از اینکه عملگر اجرا شود، این نام با مقدار متغیر جایگزین می‌شود.

جمع، تفریق، ضرب و توان‌رسانی همگی به صورت آنچه شما انتظار دارید انجام می‌شود، اما ممکن است از نحوه عمل تقسیم متعجب شوید. این عملگر یک نتیجه غیرمنتظره دارد:

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

مقدار **minute** عدد 59 است. نتیجه تقسیم 59 بر 60 برابر با 0.98333 است و نه عدد صفر. دلیل اختلاف این است که پایتون تقسیم صحیح را انجام می‌دهد.

وقتی هر دو عملوند صحیح هستند، نتیجه هم باید یک عدد صحیح باشد و بنا به قرارداد تقسیم صحیح همیشه به سمت پایین گرد می‌شود، حتی در مثال‌هایی شبیه به مثال اخیر که پاسخ به عدد صحیح بعدی (در این مثال عدد 1) بسیار نزدیک است.

یک راه حل ممکن برای رفع این مشکل، محاسبه درصدی از کسر است:

```
>>> minute*100/60
98
```

نتیجه باز هم به سمت پایین گرد شده است، اما این بار حداقل جواب تقریباً درست است. چاره دیگر استفاده از تقسیم اعشاری است که ما آن را در فصل سوم یاد می‌گیریم.

۷-۲- ترتیب عملگرها

وقتی که بیشتر از یک عملگر در عبارتی استفاده شود، ترتیب ارزیابی آنها به قوانین اولویت بستگی دارد. پایتون از قوانین اولویتی مشابه عملگرهایی که در ریاضیات استفاده می‌شوند، پیروی می‌کند.

برای حفظ نحوه آرایش عملگرها می‌توانید کلمه **PEMDAS** را به خاطر بسپارید. این کلمه از حروف اول شش کلمه انگلیسی زیر ساخته شده است و ترتیب اولویت عملگرها را نشان می‌دهد:

Parentheses	Exponentiation	Multiplication	Division	Addition	Subtraction
پرانتزها	توان‌رسانی	ضرب	تقسیم	جمع	تفریق

- پرانتزها بالاترین اولویت را دارند و می‌توانند باعث شوند که یک عبارت به طریقی که ما می‌خواهیم ارزیابی شود. از آنجایی که نخست عبارات داخل پرانتز بررسی می‌شوند، $2 * (3 - 1)$ برابر با 4 و $(5 - 2) * (1 + 1)$ برابر با 8 است. شما می‌توانید از پرانتزها برای خواناتر کردن عبارات هم استفاده کنید. همانطور که در $(minute * 100) / 60$ وجود یا عدم وجود پرانتزها در نتیجه عبارت تأثیری ندارد.
- بعد از پرانتزها، توان‌رسانی بالاترین اولویت را داراست، بنابراین نتیجه $2 * 1 + 1$ برابر با 3 می‌باشد نه 4 همچنین $3 * 1 ** 3$ برابر با 3 می‌باشد و نه 27.

- ضرب و تقسیم اولویتی برابر دارند. همچنین است برای جمع و تفریق که البته اولویت ضرب و تقسیم از جمع و تفریق بیشتر است، بنابراین $1-3*2$ عدد 5 را نتیجه می‌دهد، نه 4 و $1-2/3$ برابر با -1 است، نه 1 (به خاطر داشته باشید که در تقسیم صحیح، $2/3=0$).
- عملگرهایی که اولویت مشابه دارند از چپ به راست ارزیابی می‌شوند، بنابراین در عبارت $minute*100/60$ ابتدا عمل ضرب انجام می‌شود و $5900/60$ را نتیجه می‌دهد و سپس تقسیم انجام می‌گیرد و 98 حاصل می‌شود. اگر عملگرها از راست به چپ ارزیابی می‌شدند، نتیجه $59*1$ می‌شد که برابر بود با 59 و این اشتباه است.

۲-۸- عملیات بر روی رشته‌ها

به طور کلی شما نمی‌توانید عملیات ریاضی را بر روی رشته‌ها انجام دهید، حتی اگر شبیه به اعداد باشند. عبارات زیر غیرمجازند (فرض کنید متغیر message از نوع رشته است):

message-1 "Hello"/123 message*"Hello" "15"+2

اگرچه ممکن است بر خلاف انتظار شما باشد، اما عملگر + به طور جالبی بر روی رشته‌ها کار می‌کند. عملگر + باعث الحاق رشته‌ها می‌شود. الحاق به معنی پیوستن دو عملوند بوسیله چسبانند آنها به یکدیگر است. برای نمونه:

```
fruit = "banana"
bakedGood = " nut bread"
print fruit + bakedGood
```

خروجی این برنامه banana nut bread است. فاصله قبل از کلمه nut قسمتی از رشته می‌باشد و برای تولید فاصله میان دو رشته الحاقی لازم است.

عملگر * نیز بر روی رشته‌ها کار می‌کند و عمل تکرار را انجام می‌دهد. برای مثال $3*'Fun'$ برابر با $'FunFunFun'$ است. یکی از عملگرها باید از نوع رشته و دیگری حتماً باید از نوع صحیح باشد.

از یک طرف، تفسیر + * بوسیله قیاس با عمل جمع و ضرب، معنا می‌یابد. درست همان‌طور که $3*4$ برابر با $4+4+4$ است، انتظار داریم $3*'Fun'$ هم $'Fun'+ 'Fun'+ 'Fun'$ باشد، که اینگونه هم است. از طرف دیگر راه قابل توجهی وجود که در آن الحاق و تکرار رشته‌ها از جمع و ضرب اعداد صحیح متمایزند. آیا می‌توانید به خصوصیتی فکر کنید که جمع و ضرب اعداد صحیح دارند اما الحاق و تکرار رشته‌ها ندارند؟

۲-۹- ترکیب

تا به حال ما به عناصر یک برنامه به طور جداگانه نگریسته‌ایم - متغیرها، عبارات و دستورات- ولی دربارهٔ اینکه چگونه با هم ترکیب می‌شوند صحبتی نکرده‌ایم. یکی از مفیدترین خصوصیات زبان‌های برنامه‌نویسی توانایی آنها در گرفتن اجزای کوچک برنامه و ترکیب آنها است. برای نمونه، ما جمع کردن اعداد را می‌دانیم و نیز آموخته‌ایم که چگونه چیزی را چاپ کنیم، پس ما می‌توانیم این دو عمل را در یک زمان انجام دهیم:

```
>>> print 17 + 3
20
```

در حقیقت جمع کردن باید قبل از عمل چاپ صورت گیرد، لذا عملیات در واقع در یک زمان انجام نمی‌شود. نکته این است که هر عبارتی که شامل اعداد، رشته‌ها و متغیرها است، می‌تواند در یک دستور چاپ استفاده شود. شما قبلاً نمونه‌ای از این مورد را دیده‌اید.

```
print "Number of minutes since midnight: ", hour*60+minute
```

همچنین شما می‌توانید یک عبارت دلخواه را در سمت راست یک گزارهٔ نسبت‌دهی قرار دهید:

```
percentage = (minute * 100) / 60
```

این توانایی شاید در حال حاضر برای شما زیاد جالب نباشد، اما شما مثال‌هایی خواهید دید که در آنها ترکیب به شما امکان می‌دهد که محاسبات پیچیده را مختصر و بدون نقص ارائه دهید.

اخطار: محدودیت‌هایی در نحوهٔ استفاده از برخی عبارات وجود دارد. برای نمونه سمت چپ یک گزارهٔ نسبت‌دهی باید یک نام متغیر باشد و نه یک عبارت، بنابراین این دستور غیرمجاز است:

```
minute+1 = hour
```

۲-۱۰- توضیحات

همچنان‌که برنامه‌ها پیچیدگی بیشتری می‌یابند، خواندن آنها مشکل‌تر می‌شود. زبان‌های رسمی متراکمند و نگاه کردن به تکه‌ای از کد برنامه و فهمیدن اینکه چه کاری انجام می‌دهند و به چه علت، اغلب مشکل است. به همین دلیل بسیار جالب است که یادداشت‌هایی به برنامه‌تان اضافه کنید و با زبان طبیعی توضیح دهید که برنامه چه کاری انجام می‌دهد. این یادداشت‌ها، **توضیحات** نامیده می‌شوند و با نماد # علامت‌گذاری می‌شوند:

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

در این مورد، توضیح در یک خط اختصاصی نشان داده شده است. شما می‌توانید توضیحات را در آخر خطوط کد برنامه بگذارید:

```
percentage = (minute * 100) / 60    # caution: integer
division
```

هر چیز که بعد از علامت `#` قرار گیرد تا آخر خط نادیده گرفته می‌شود، یعنی هیچ تأثیری در روند برنامه ندارد. این پیغام برای برنامه‌نویس یا برنامه‌نویسان آینده که ممکن است از این کد استفاده کنند در نظر گرفته شده است. در این مثال، توضیح، رفتار غیرعادی تقسیم صحیح را به خواننده یادآوری می‌کند.

۲-۱۱- واژه‌نامه

value (مقدار)

یک عدد یا رشته (یا چیزهای دیگری که در آینده نام می‌بریم) که بتواند در یک متغیر ذخیره یا در یک عبارت محاسبه شود.

type (نوع)

جنس مقادیر. نوع یک مقدار معین می‌کند که چگونه می‌تواند در یک عبارت استفاده شود. انواع داده‌ای که تاکنون دیده‌اید عبارتند از اعداد صحیح (نوع `int`)، اعداد اعشاری (نوع `float`) و رشته‌ها (نوع `str`).

floating-point (اعداد اعشاری)

قابلی برای نشان دادن اعدادی که دارای بخش اعشاری هستند.

variable (متغیر)

نامی که به یک مقدار اشاره می‌کند.

statement (دستور، گزاره)

بخشی از کد برنامه که فرمان یا عملی را نشان می‌دهد. دستوراتی که تاکنون دیده‌اید گزاره نسبت‌دهی و چاپ هستند.

assignment (نسبت‌دهی)

گزاره‌ای که یک مقدار را به متغیری نسبت می‌دهد.

state diagram (نمودار وضعیت)

نمایش گرافیکی مجموعه متغیرها و مقادیری که به آنها اشاره می‌کنند.

keyword (کلمه کلیدی)

یک کلمه رزرو شده توسط زبان برنامه‌نویسی که کامپایلر یا مفسر برای تجزیه برنامه از آن استفاده می‌کند؛ شما نمی‌توانید کلمه‌های کلیدی همچون `while`، `def`، `if` و ... را به عنوان نام متغیر استفاده کنید.

expression (عبارت)

ترکیب متغیرها، عملگرها و مقادیری که یک نتیجه واحد را نشان می‌دهند.

evaluate (ارزیابی)

ساده کردن یک عبارت به وسیله انجام عملیاتی به منظور بدست آوردن مقداری واحد.

operator (عملگر)

نماد ویژه‌ای که یک محاسبه ساده مانند جمع، ضرب یا الحاق رشته را نشان می‌دهد.

operand (عملوند)

یکی از مقادیری که عملگر بر روی آن عمل می‌کند.

integer division (تقسیم صحیح)

عملیاتی که یک عدد صحیح را بر عدد صحیح دیگری تقسیم می‌کند و یک عدد صحیح را هم نتیجه می‌دهد. تقسیم صحیح تنها تعداد دفعاتی که مقسوم به مقسوم‌علیه بخش می‌شود را نتیجه می‌دهد و هر چیزی که باقی ماند را دور می‌ریزد.

rules of precedence (قوانین اولویت)

مجموعه‌ای از قواعد حاکم که معین می‌کند در یک عبارت با چند عملگر و عملوند، کدامیک و با چه ترتیبی ارزیابی شوند.

concatenate (الحاق)

اتصال دو عملوند پشت سر هم.

composition (ترکیب)

توانایی ترکیب عبارات و دستورات ساده در عبارات و دستورات مرکب برای نشان دادن محاسبات پیچیده به صورت مختصر.

comment (توضیح)

اطلاعاتی در برنامه که برای دیگر برنامه‌نویسان (و یا دیگر کسانی که کد مبداء برنامه را می‌خوانند) در نظر گرفته می‌شود و هیچ تأثیری در روند اجرای برنامه ندارند.

توابع



در فصل قبل با برخی از انواع داده‌ای پایتون از جمله رشته‌ها، اعداد صحیح و اعداد اعشاری آشنا گشته‌اید و احتمالاً در هنگام کار با آنها با محدودیت‌هایی مواجه شده‌اید. در این فصل، ضمن آشنایی با روش‌های تبدیل انواع داده‌ای به یکدیگر، نحوه تعریف و کار با توابع را فرا می‌گیرید. توابع به عنوان یکی از مفاهیم اساسی زبان‌های برنامه‌نویسی، دارای مزایای بسیاری هستند. با استفاده از توابع، برنامه‌نویسان می‌توانند مجموعه دستوراتی که مکرراً اجرا می‌شوند را بسته‌بندی کرده و در مکان‌های مختلف استفاده کنند. همچنین تقسیم برنامه‌های بزرگ به توابع کوتاه‌تر، برنامه‌نویسان را قادر می‌سازد که در طراحی برنامه‌ها همکاری داشته باشند. توانایی طراحی، فراخوانی و اجرای توابع از مهم‌ترین مسائل برنامه‌نویسی است که در این فصل و فصل‌های آینده با آنها آشنا می‌شوید.

۳-۱- فراخوانی تابع

شما قبلاً مثالی از یک فراخوانی تابع دیده‌اید:

```
>>> type("32")
<type 'str'>
```

در این مثال، نام تابع **type** است و نوع یک مقدار یا متغیر را نمایش می‌دهد. مقدار یا متغیری که **آرگومان** تابع نامیده می‌شود باید بین دو پرانتز قرار گرفته باشد. معمول است که گفته شود تابع آرگومانی را "می‌گیرد" و نتیجه‌ای را "برمی‌گرداند". نتیجه مقدار برگشتی نامیده می‌شود. می‌توانیم به جای چاپ کردن مقدار برگشتی، آن را به یک متغیر نسبت دهیم:

```
>>> betty = type("32")
>>> print betty
<type 'str'>
```

به عنوان یک مثال دیگر، تابع **id** مقدار یا متغیری را می‌گیرد و عدد صحیحی را برمی‌گرداند. این عدد صحیح به عنوان شناسه منحصر بفرد مقدار عمل می‌کند:

```
>>> id(3)
134882108
>>> betty = 3
>>> id(betty)
134882108
```

هر مقدار یک **id** دارد. این **id** عدد منحصر بفردی است که به محل ذخیره شدن مقدار در حافظه کامپیوتر بستگی دارد. **id** یک متغیر، **id** مقداری است که به آن اشاره می‌کند.

۲-۳- تبدیل نوع داده

پایتون مجموعه‌ای از توابع پیش‌ساخته را فراهم می‌کند که می‌توانند مقادیر را از یک نوع به نوع دیگر تبدیل کنند. تابع **int** هر مقداری را می‌گیرد و در صورت امکان به یک عدد صحیح تبدیل می‌کند و در غیر این صورت پیغام خطایی را نمایش می‌دهد:

```
>>> int("32")
32
>>> int("Hello")
ValueError: invalid literal for int(): Hello
```

تابع **int** همچنین می‌تواند مقادیر اعشاری را به صحیح تبدیل کند، اما به خاطر داشته باشید این تابع قسمت اعشاری را دور می‌ریزد:

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

تابع **float** رشته‌ها و اعداد صحیح را به اعداد اعشاری تبدیل می‌کند:

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

سرانجام اینکه تابع **str** مقادیر را به نوع رشته تبدیل می‌کند:

```
>>> str(32)
'32'
>>> str(3.14149)
'3.14149'
```

ممکن است عجیب به نظر برسد که پایتون میان مقدار صحیح **1** و مقدار اعشاری **1.0** تمایز قائل شود. آنها ممکن است هر دو یک عدد را نمایش دهند اما به انواع داده‌ای متفاوتی تعلق دارند. علت این است که این دو عدد درون کامپیوتر به دو شکل متفاوت نشان داده می‌شوند.

۳-۳- تبدیل موقت نوع

اکنون که ما می‌توانیم انواع داده‌ها را به هم تبدیل کنیم، یک بار دیگر به تقسیم صحیح بازمی‌گردیم. با رجوع به مثالی از فصل قبل فرض می‌کنیم که می‌خواهیم کسری از زمان سپری شده را محاسبه کنیم. عبارت `minute/60` یک تقسیم صحیح انجام می‌دهد، بنابراین نتیجه همیشه صفر است؛ حتی زمانی که ۵۹ دقیقه از ساعت گذشته باشد!

یک راه حل این است که متغیر `minute` را به نوع اعشاری تبدیل کنیم و یک تقسیم اعشاری انجام دهیم:

```
>>> minute = 59
>>> float(minute) / 60.0
0.983333333333
```

راه حل دیگر این است که از قواعد تبدیل خودکار نوع داده سود ببریم. قواعدی که از آن به عنوان **تبدیل موقت نوع** نام برده می‌شود. برای عملگرهای حسابی، اگر یکی از عملوندها اعشاری باشد، عملوند دیگر خودبه‌خود به نوع اعشاری تبدیل می‌شود:

```
>>> minute = 59
>>> minute / 60.0
0.983333333333
```

با قرار دادن مخرج به عنوان یک عدد اعشاری ما پایتون را مجبور می‌کنیم که یک تقسیم اعشاری انجام دهد.

۳-۴- توابع ریاضی

در ریاضیات شما احتمالاً توابعی همچون `sin` و `log` را دیده‌اید و نحوه ارزیابی عباراتی همچون `sin(pi/2)` و `log(1/x)` را آموخته‌اید. شما ابتدا عبارت داخل پرانتز (آرگومان) را ارزیابی می‌کنید. برای نمونه `pi/2` تقریباً برابر `1.571` و `1/x` در صورتی که `x` را `10.0` فرض کنیم، برابر `0.1` است.

آنگاه شما خود تابع را با جستجو در یک جدول و یا انجام محاسبات گوناگون ارزیابی می‌کنید. سینوس عدد `1.571` برابر با `1` است و لگاریتم عدد `0.1` (در مبنای ۱۰) برابر با `-1` است.

این فرایند می‌تواند جهت ارزیابی عبارات پیچیده‌تر همچون $\log(1/\sin(\pi/2))$ به طور مکرر انجام شود. ابتدا شما آرگومان درونی‌ترین تابع را ارزیابی می‌کنید، سپس تابع را مورد بررسی قرار می‌دهید و به همین ترتیب.

پایتون ماژولی به نام **math** دارد که اغلب توابع رایج ریاضیات در آن وجود دارد. یک ماژول فایلی است که مجموعه‌ای از توابع مرتبط و مجتمع را شامل می‌شود. قبل از اینکه بتوانیم از توابع یک ماژول استفاده کنیم، باید آن ماژول را با استفاده از دستور **import** وارد فضای کاری کنیم:

```
>>> import math
```

به منظور فراخوانی یک تابع ما باید نام یک ماژول و نام تابع مورد نظر را مشخص کرده و آن را با یک نقطه جدا کنیم. این نوع قالب‌بندی را **نمادگذاری نقطه‌ای** می‌نامند.

```
>>> decibel = math.log10 (17.0)
>>> angle = 1.5
>>> height = math.sin(angle)
```

دستور اول لگاریتم **17** در مبنای **10** را در متغیر **decibel** قرار می‌دهد. در پایتون همچنین تابعی با نام **log** موجود است که لگاریتم را در پایه **e** (عدد نپر) می‌گیرد. سومین دستور، سینوس مقدار متغیر **angle** را محاسبه می‌کند. **sin** و دیگر توابع مثلثاتی (**cot**، **tan** و غیره) آرگومان‌ها را در واحد رادیان دریافت می‌کنند. برای تبدیل درجه به رادیان، درجه را بر **۱۸۰** تقسیم و در عدد **pi** ضرب می‌کنیم. برای مثال جهت پیدا کردن سینوس **۴۵** درجه ابتدا زاویه را در واحد رادیان محاسبه کرده و سپس از آن سینوس می‌گیریم:

```
>>> degrees = 45
>>> angle = degrees * math.pi / 180
>>> math.sin(angle)
```

ثابت **pi** هم قسمتی از ماژول **math** می‌باشد. اگر شما اطلاعاتی درباره هندسه داشته باشید، می‌توانید نتیجه را به وسیله مقایسه با جذر **۲** بخش **۲** بسنجید:

```
>>> math.sqrt(2) / 2.0
0.707106781187
```

۳-۵- مازول‌ها

همان‌طور که گفته شد، مازول فایل‌ی است که مجموعه‌ای از توابع مرتبط و مجتمع را شامل می‌شود. ارتباط میان توابع می‌تواند از طریق دستورات برنامه‌نویسی در زبان پایتون صورت گیرد. بنابراین در این میان می‌توانیم متغیرها و ثابت‌هایی هم تعریف کرده و از آنها استفاده نماییم. ما با نمادگذاری نقطه‌ای قادریم به تمام این توابع، متغیرها و ثوابت دسترسی پیدا کنیم.

در فصل اول توضیح دادیم که برنامه‌های پایتون را می‌توان به دو روش نوشت؛ یکی روش خط فرمان که در آن با نوشتن هر خط برنامه و فشردن کلید **Enter** دستور (یا دستورات) اجرا می‌شود و دیگری روش اسکریپت که کل برنامه را در فایل‌ی می‌نویسیم و آن را اجرا می‌کنیم. مفسر پایتون خطوط برنامه را یک‌به‌یک و به ترتیب می‌خواند و اجرا می‌کند. این فایل که ما از آن به عنوان اسکریپت یاد کردیم، در حقیقت یک مازول است. وقتی شما برنامه‌ای را در فایل‌ی با پسوند **.py** بنویسید، این فایل می‌تواند توسط مفسر پایتون اجرا شود و همچنین تمام توابع، متغیرها و ثوابتی که در آن نوشته‌اید، توسط دیگر مازول‌ها و نیز در برنامه‌های نوشته شده در خط فرمان استفاده شود.

در بخش قبل چگونگی وارد کردن یک مازول را آموختید و همچنین فرا گرفتید که چگونه به‌وسیلهٔ نمادگذاری نقطه‌ای از ثابت‌ها و توابع یک مازول استفاده کنید. چارهٔ دیگری وجود دارد که در آن دیگر لازم نیست از نمادگذاری نقطه‌ای استفاده کنیم.

دستور **from...import** می‌تواند تمامی توابع، متغیرها و ثوابت یک مازول و یا بخش دلخواهی از آنها را وارد کند. الگوی استفاده از این دستور به این صورت است:

```
from MODULE import NAME1, NAME2, NAME3,...
```

دستور زیر تابع **cos** و **exp** را وارد محیط کاری می‌کند:

```
from math import cos, exp
```

شما همچنین می‌توانید برای دسترسی به تمام محتویات مازول (توابع، متغیرها و ثوابت) به جای ذکر نام تک‌تک آنها از علامت ***** استفاده کنید:

```
>>> from math import *
```

این دستور همهٔ توابع و متغیرها و ثابت‌های تعریف شده در مازول **math** را (به‌جز آن دسته که با علامت زیر خط **(_)** شروع شده‌اند)، وارد محیط کاری می‌کند.

۳-۶- ترکیب

درست مانند توابع ریاضی، توابع پایتون هم می‌توانند با هم ترکیب شوند. یعنی اینکه شما می‌توانید از عبارتی به عنوان قسمتی از یک عبارت دیگر استفاده کنید. برای نمونه شما می‌توانید از هر عبارتی به عنوان آرگومان تابع استفاده کنید:

```
>>> x = cos(angle + pi/2)
```

این دستور مقدار π را می‌گیرد و آن را بر ۲ تقسیم کرده و نتیجه را با مقدار `angle` جمع می‌کند و آنگاه مجموع را به عنوان یک آرگومان به تابع `cos` می‌فرستد. شما همچنین می‌توانید نتیجه یک تابع را گرفته و به عنوان یک آرگومان به تابعی دیگر بفرستید:

```
>>> x = exp(log(10.0))
```

این دستور نتیجه لگاریتم ۱۰ در مبنای `e` را محاسبه کرده و سپس `e` را به توان آن می‌رساند. نتیجه پایانی به متغیر `x` اختصاص می‌یابد.

۳-۷- اضافه کردن توابع جدید

تاکنون ما تنها از توابعی استفاده کرده‌ایم که همراه پایتون بوده‌اند اما این امکان هم وجود دارد که توابع جدیدی اضافه کنیم. ساختن توابع جدید برای حل مسائل خاص یکی از مفیدترین امکانات یک زبان برنامه‌نویسی چندمنظوره است.

در مبحث برنامه‌نویسی، یک تابع دنباله نام‌داری از دستورات است که عملیات خاصی را انجام می‌دهد. این عملکرد در قسمت **تعریف تابع** مشخص می‌شود. توابعی که تاکنون از آنها استفاده کرده‌ایم، از قبل تعریف شده و این تعاریف از دید ما پنهان بودند. این ویژگی خوبی است زیرا به شما اجازه می‌دهد بدون اینکه نگران جزئیات تعاریف باشید از آن استفاده کنید. الگوی کلی تعریف تابع به صورت زیر است:

```
def NAME( LIST OF PARAMETERS ) :  
    STATEMENTS
```

شما می‌توانید هر نامی را که بخواهید برای تابعی که ساخته‌اید بکار ببرید، به جز اسامی کلیدی زبان پایتون. لیست پارامترها (`LIST OF PARAMETERS`) مشخص می‌کند که برای استفاده از تابع جدید چه اطلاعاتی را (در صورت وجود) باید به تابع بدهیم.

می‌توانیم هر تعداد دستوری را درون تابع بکار ببریم. اما این دستورات باید از حاشیه سمت چپ کنگره‌گذاری^۲ شوند. اولین دسته از توابعی که ما قصد داریم بنویسیم، هیچ پارامتری ندارند. بنابراین نحوه نگارش آنها به این صورت است:

```
def newLine():
    print
```

این تابع **newLine** نام دارد. پرانتزهای خالی مشخص می‌کند که این تابع هیچ پارامتری ندارد و تنها یک دستور دارد که خروجی آن، کاراکتر خط جدید است. (این اتفاق، یعنی چاپ کاراکتر خط جدید، هنگامی رخ می‌دهد که شما از یک دستور **print** بدون هیچ آرگومانی استفاده کنید) نحوه فراخوانی توابع جدید، درست مانند نحوه فراخوانی توابع پیش‌ساخته پایتون است:

```
print "First Line."
newLine()
print "Second Line."
```

خروجی این برنامه به این صورت است:

```
First line.
Second line.
```

به سطر خالی میان این دو خط دقت کنید. اگر سطرهای خالی بیشتری نیاز داشتیم چطور؟ می‌توانیم همین تابع را چندین مرتبه فراخوانی کنیم:

```
print "First Line."
newLine()
newLine()
newLine()
print "Second Line."
```

یا اینکه تابع جدیدی به نام **threeLines** بنویسیم که سه خط جدید چاپ کند:

```
def threeLines():  
    newLine()  
    newLine()  
    newLine()  
  
print "First Line."  
threeLines()  
print "Second Line."
```

این تابع شامل سه دستور است که همگی آنها بوسیله دو کاراکتر "فاصله" کنگره‌گذاری شده‌اند. از زمانی که به اولین دستور بدون تورفتگی برسیم، پایتون می‌فهمد که این دستور جزء تابع نیست. توجه به نکاتی چند درباره این برنامه ضروری است:

۱. شما می‌توانید یک روال را مکرراً فراخوانی کنید. در واقع انجام این کار بسیار رایج و مفید است.

۲. شما می‌توانید تابعی داشته باشید که خود تابع دیگری را فراخوانی کند. در این مثال، تابع **threeLines** تابع **newLine** را فراخوانی می‌کند.

شاید تابه‌حال علت استفاده از توابع جدید به خوبی روشن نشده باشد. در حقیقت دلایل زیادی برای استفاده از توابع وجود دارد که در زیر دو نمونه از موارد مشهود در مثال اخیر را می‌بینید:

- ساختن یک تابع جدید به شما این امکان را می‌دهد که گروهی از دستورات را نامگذاری کنید. توابع می‌توانند با پنهان کردن یک سری محاسبات پیچیده در پشت یک دستور ساده، آن هم با کلمات انگلیسی (به جای یک کد محرمانه) برنامه را ساده‌تر نمایند.
- ساختن یک تابع جدید، برنامه را با حذف کدهای تکراری کوتاه‌تر می‌کند. برای مثال یک راه کوتاه‌تر برای چاپ نُه خط جدید متوالی این است که تابع **threeLines** را سه مرتبه فراخوانی کنیم.

تمرین ۳-۱: تابعی با نام **nineLines** بنویسید که خود تابع **threeLines** را به‌منظور چاپ نُه خط جدید، سه مرتبه به‌کار بندد. چطور می‌توان ۲۷ خط جدید چاپ کرد؟

۳-۸- تعریف و استفاده از توابع

از اتصال کدهای جدا از هم بخش ۳-۶ برنامه‌ای شبیه به آنچه در زیر می‌بینید نتیجه می‌شود:

```
def newLine():
    print

def threeLines():
    newLine()
    newLine()
    newLine()

print "First Line."
threeLines()
print "Second Line."
```

این برنامه شامل دو تعریف تابع است: **newLine** و **threeLines**. قسمت معرفی تابع همچون دیگر دستورات برنامه اجرا می‌شود، با این تفاوت که نتیجه حاصل، ساخته شدن تابع جدید است. دستورات درون یک تابع تا زمانی که تابع فراخوانی نشود اجرا نمی‌شود و تعریف تابع هیچ خروجی‌ای تولید نمی‌کند.

همان‌طور که انتظار دارید ساختن یک تابع باید پیش از اجرای آن صورت گیرد. به بیان دیگر معرفی تابع باید قبل از اینکه تابع برای اولین بار فراخوانده شود، انجام گرفته باشد.

تمرین ۳-۲: سه خط پایانی برنامه بالا را به ابتدای برنامه انتقال دهید. بدین ترتیب فراخوانی تابع قبل از معرفی آن رخ می‌دهد. حال برنامه را اجرا کنید تا ببینید چه پیغام خطایی دریافت می‌کنید.

تمرین ۳-۳: کار را با نسخه صحیح برنامه آغاز کنید و این بار تعریف تابع **newLine** را به جایی پس از تعریف تابع **threeLines** انتقال دهید. پس از اجرای برنامه چه اتفاقی رخ می‌دهد؟

۳-۹- روند اجرا

برای اطمینان از اینکه تابع قبل از اولین استفاده، تعریف شده است باید از ترتیب اجرای دستورات آگاه باشید، که به این مطلب اصطلاحاً **روند اجرا** گفته می‌شود.

عمل اجرا همواره از اولین دستور برنامه شروع می‌شود و دستورات یکی یکی و به ترتیب از بالا به پایین اجرا می‌گردند. تعریف تابع روند اجرای برنامه را تغییر نمی‌دهد، اما به خاطر داشته باشید که

دستورات درون تابع تا زمانی که تابع فراخوانی نشود، اجرا نمی‌شوند. اگرچه این کار معمول نیست اما شما می‌توانید یک تابع را درون تابع دیگری تعریف کنید. در این مورد تعریف تابع درونی تا زمانی که تابع بیرونی فراخوانده نشود، اجرا نمی‌شود.

فراخوانی تابع شبیه یک راه فرعی در روند اجرا است. هنگامی که برنامه به دستور فراخوانی تابع می‌رسد، به جای رفتن به دستور بعد، روند برنامه به اولین خط تابع فراخوانی شده می‌پردازد. تمام دستورات آنجا را اجرا می‌کند و سپس به جایی که روند اجرا را رها کرده بود برمی‌گردد و اجرای دستورات را ادامه می‌دهد.

قبل از اینکه به یاد بیاورید یک تابع می‌تواند تابع دیگری را هم صدا بزند، موضوع ساده به نظر می‌رسد. حال آنکه ممکن است در وسط یک تابع، برنامه مجبور شود که دستورات یک تابع دیگر را اجرا کند. اما در حین اجرای آن تابع برنامه مجبور شود باز هم تابع دیگری را اجرا کند و به همین ترتیب. خوشبختانه، پایتون در حفظ روند اجرای برنامه ماهر است. هر زمان که تابعی کامل می‌شود، برنامه مجدداً به تابعی که آن را فراخوانده باز می‌گردد و ادامه کار را از سر می‌گیرد. وقتی که به انتهای برنامه می‌رسد کار پایان می‌یابد.

اما نتیجه این داستان بی‌مزه چیست؟ وقتی که شما برنامه‌ای را می‌خوانید، آن را از بالا به پایین دنبال نکنید، بلکه در عوض روند اجرا را پیگیری نمایید.

۳-۱- پارامترها و آرگومان‌ها

برخی از توابع پیش‌ساخته‌ای که تا به حال استفاده کرده‌اید، به آرگومان‌هایی نیاز دارند؛ مقادیری که نحوه کارکرد توابع را کنترل می‌کنند. برای مثال هنگامی که می‌خواهید سینوس عددی را بدست آورید، باید عدد مورد نظر را مشخص کنید. بنابراین تابع **sin** یک مقدار عددی به عنوان آرگومان می‌گیرد.

بعضی از توابع بیش از یک آرگومان می‌گیرند. برای نمونه تابع **pow** دو آرگومان می‌گیرد، پایه و نما. مقادیری که به تابع فرستاده می‌شوند، در درون تابع به متغیرهایی به نام پارامتر اختصاص می‌یابند.

در اینجا مثالی از یک تابع کاربر-تعریف را می‌بینید که یک پارامتر می‌گیرد:

```
def printTwice(bruce):
    print bruce, bruce
```

این تابع یک آرگومان واحد را می‌گیرد و آن را به پارامتری به نام **bruce** نسبت می‌دهد. مقدار پارامتر (صرف نظر از اینکه چه باشد) دو مرتبه و در یک سطر چاپ می‌شود. نام **bruce** به این دلیل انتخاب شده که نشان دهیم انتخاب نام پارامترها به شما بستگی دارد. در حالت کلی شاید بخواهید اسامی گویاتری برای پارامترهای خود انتخاب کنید.

تابع **printTwice** با نوع داده‌ای که قابل چاپ باشد، کار می‌کند:

```
>>> printTwice('Spam')
Spam Spam
>>> printTwice(5)
5 5
>>> printTwice(3.14159)
3.14159 3.14159
```

در اولین فراخوانی تابع، آرگومان یک رشته، در دومین فراخوانی، یک عدد صحیح و در سومین فراخوانی یک عدد اعشاری می‌باشد.

همان قواعد ترکیب که در توابع پیش‌ساخته پایتون برقرار بودند، در توابع کاربر-تعریف هم عمل می‌کنند. بنابراین ما می‌توانیم از هر عبارتی به عنوان آرگومان **printTwice** استفاده کنیم:

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

به طور معمول، عبارت قبل از اجرای تابع ارزیابی می‌شود. بنابراین تابع **printTwice** مقدار **'Spam'*4-SpamSpamSpamSpam-SpamSpamSpamSpam** را به جای **'Spam'*4** چاپ می‌کند. لازم به ذکر است که **printTwice** مقدار **'Spam'*4** را به **SpamSpamSpamSpam** تبدیل نمی‌کند بلکه این کار توسط مفسر پایتون صورت می‌گیرد.

printTwice تنها عبارت ارزیابی شده را چاپ می‌کند.

رشته‌ها می‌توانند در میان تک کوتیشن یا زوج کوتیشن قرار گیرند و آن نوع کوتیشنی که برای محصور کردن رشته استفاده نشده، می‌تواند به عنوان قسمتی از رشته درون آن استفاده شود:

```
>>> state1 = "Don't worry!"
>>> print state1
Don't worry!
>>> state2 = 'Mr. Rossum', has created Python.'
"Mr. Rossum", has created Python.
```

تمرین ۳-۴: `printTwice` را طوری فراخوانی کنید که مقدار `'Spam' * 4 - 'Spam' * 4` را بازگرداند.

ما همچنین می‌توانیم از یک متغیر به عنوان آرگومان استفاده کنیم:

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

در اینجا به نکته بسیار مهمی توجه کنید. نام متغیری که به عنوان آرگومان فرستاده می‌شود (`michael`) هیچ ارتباطی با نام پارامتر (`bruce`) ندارد و تغییری در آن به وجود نمی‌آورد. مهم نیست که تابع با چه مقداری فراخوانده شود؛ اینجا در تابع `printTwice` ما همه چیز را `bruce` خطاب می‌کنیم.

۳-۱۱- متغیرها و پارامترها محلی هستند

هنگامی که شما یک متغیر محلی را درون تابعی می‌سازید، این متغیر تنها درون تابع موجودیت دارد و شما نمی‌توانید بیرون تابع از آن استفاده کنید. برای نمونه:

```
def catTwice(part1, part2):
    cat = part1 + part2
    printTwice(cat)
```

این تابع دو آرگومان می‌گیرد، آنها را با هم جمع (در صورتی که عدد باشند) یا الحاق (در صورتی که رشته باشند) کرده و سپس نتیجه را دو مرتبه چاپ می‌کند. ما تابع را با دو رشته فراخوانی می‌کنیم:

```
>>> chant1 = "Pie Jesu domine, "
>>> chant2 = "Dona eis requiem."
>>> catTwice(chant1, chant2)
Pie Jesu domine, Dona eis requiem. Pie Jesu domine, Dona eis requiem.
```

وقتی عملیات تابع `catTwice` پایان می‌یابد، متغیر `cat` از بین می‌رود. اگر سعی کنیم آن را چاپ کنیم، پیغام خطا دریافت می‌کنیم:

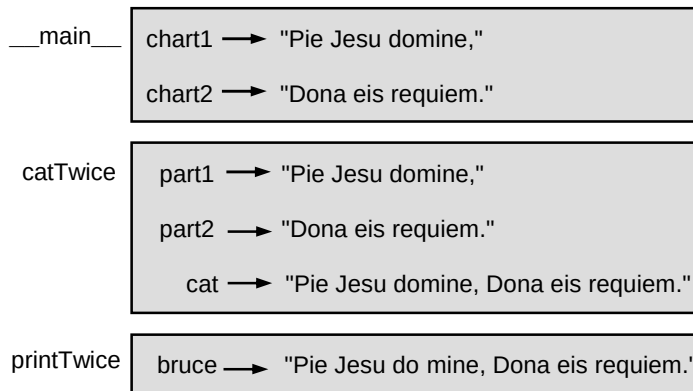
```
>>> print cat
Traceback (most recent call last):
  File "<interactive input>", line 1, in ?
NameError: name 'cat' is not defined
```

پارامترها نیز محلی هستند. برای مثال، خارج از تابع **printTwice** چیزی به عنوان **bruce** وجود ندارد و اگر سعی کنید از آن استفاده نمایید، پایتون اعتراض می‌کند.

۱۲-۳- نمودارهای پشته

جهت ثبت ردّ متغیرها و به‌خاطر سپردن این موضوع که هر یک در چه قسمتی استفاده می‌شوند، رسم یک **نمودار پشته** گاهی مفید است. نمودارهای پشته نیز همچون نمودارهای حالت، مقدار یک متغیر را نمایش می‌دهند، با این تفاوت که آنها علاوه بر این مشخص می‌کنند که هر متغیر به چه تابعی تعلق دارد.

هر تابع با یک **قاب** نمایش داده می‌شود. هر قاب از یک کادر مستطیل شکل تشکیل شده است که نام تابع در کنار آن و نام متغیرها و پارامترها درون آن نوشته می‌شوند. نمودار پشته مثال قبل را در زیر می‌بینید:



شکل ۱-۳

ترتیب پشته، روند اجرای برنامه را نمایش می‌دهد. تابع **printTwice** به‌وسیلهٔ تابع **catTwice** فراخوانی می‌شود و **catTwice** خود توسط تابع **__main__** که نامی برای بالاترین تابع است، فراخوانده می‌شود. هرگاه شما متغیری خارج از محیط توابع می‌سازید، این متغیر به **__main__** تعلق دارد.

هر پارامتر به مقدار آرگومان نظیرش اشاره می‌کند. بنابراین **part1** مقداری برابر **chant1** و **part2** مقداری برابر با **chant2** دارد. همچنین **bruce** برابر با **cat** است.

اگر در طول فراخوانی تابع خطایی اتفاق افتد، پایتون نام آن تابع و نام تابعی که آن را فراخوانده و به همین ترتیب تا جایی که به **__main__** برگردد را چاپ می‌کند.

برای مثال، اگر سعی کنید از درون تابع **printTwice** به **cat** دسترسی پیدا کنید، یک خطای **NameError** دریافت می‌کنید:

```
Traceback (innermost last):
  File "test.py", line 13, in __main__
    catTwice(chant1, chant2)
  File "test.py", line 5, in catTwice
    printTwice(cat)
  File "test.py", line 9, in printTwice
    print cat
NameError: cat
```

این لیست از توابع را پس‌یابی می‌نامند. پس‌یابی به شما می‌گوید خطا در کدام فایل برنامه و در کدام خط اتفاق افتاده است و چه تابعی در آن لحظه در حال اجرا بوده‌اند. همچنین آن خطا از برنامه که موجب خطا شده است را نشان می‌دهد.

به شباهت میان پس‌یابی و نمودار پشته توجه کنید. این شباهت تصادفی نیست.

۳-۱۳- توابع نتیجه‌دار

ممکن است متوجه شده باشید برخی از توابعی که تا به حال به کار برده‌اید، مانند توابع ریاضی نتیجه‌ای را برمی‌گردانند. توابع دیگری مانند **newLine** عملی انجام می‌دهند اما مقداری را برنمی‌گردانند. این بحث سؤالاتی را به‌وجود می‌آورد:

- اگر تابعی را فراخوانی کنید و روی نتیجه آن هیچ عملی انجام ندهید، چه روی می‌دهد؟ (مثلاً آن را به متغیری نسبت ندهید و یا در یک عبارت بزرگتر استفاده نکنید).
- ۳. اگر تابعی را که نتیجه‌ای برنمی‌گرداند به عنوان قسمتی از یک عبارت استفاده کنید چه روی می‌دهد؟ مثلاً **newLine() + 7**
- ۴. آیا می‌توان توابعی نوشت که نتیجه‌ای را بازگردانند یا همچنان باید درگیر توابع ساده‌ای نظیر **newLine** و **printTwice** بود؟

پاسخ سومین سؤال مثبت است و ما این کار را در فصل پنجم انجام خواهیم داد.

تمرین ۳-۵: دو سؤال دیگر را پاسخ دهید. وقتی که شما دربارهٔ مجاز یا عدم مجاز بودن عملی در پایتون پرسشی دارید، یک راه مناسب برای یافتن پاسخ، پرسیدن آن از مفسر است.

۳-۱۴- واژه‌نامه

function call (فراخوانی تابع)

دستوری که یک تابع را اجرا می‌کند و عبارت است از نام تابع و لیستی از آرگومان‌های آن که درون یک پرانتز قرار دارد.

argument (آرگومان)

مقداری که هنگام فراخوانی تابع به آن ارسال می‌شود. این مقدار به پارامتر نظیر خود در تابع، اختصاص می‌یابد.

return value (مقدار برگشتی)

نتیجهٔ یک تابع. اگر فراخوانی تابع به عنوان یک عبارت استفاده شود، مقدار برگشتی مقدار آن عبارت خواهد بود.

type conversion (تبدیل نوع)

یک دستور واضح و صریح که مقداری را از یک نوع می‌گیرد و به مقدار نظیرش از نوع دیگر تبدیل می‌کند.

type coercion (تبدیل موقت نوع)

یک تبدیل نوع که بر اساس قوانین تبدیل موقت پایتون به صورت خودکار روی می‌دهد.

module (ماژول)

فایلی شامل مجموعه‌ای از کلاس‌ها و توابع وابسته به هم.

dot notation (نمادگذاری نقطه‌ای)

نحوهٔ نگارش برای فراخوانی یک تابع از درون یک ماژول دیگر که با مشخص کردن نام ماژول، یک نقطه و نام ماژول انجام می‌گیرد.

function (تابع)

دنباله نام‌داری از دستورات که عملیات مفیدی انجام می‌دهد. توابع ممکن است پارامتری داشته باشند و یا نداشته باشند. همین‌طور ممکن است نتیجه‌ای را تولید کنند و یا هیچ نتیجه‌ای نداشته باشند.

function definition (تعریف تابع)

دستوری که تابع جدیدی می‌سازد و نام، پارامترها و دستورات اجرایی آن را مشخص می‌کند.

flow of execution (روند اجرا)

ترتیب اجرای دستورات در طول اجرای برنامه

parameter (پارامتر)

نامی که درون تابع استفاده می‌شود و مقدار ارسال شده به عنوان آرگومان به آن نسبت داده می‌شود.

local variable (متغیر محلی)

متغیر تعریف شده درون یک تابع. یک متغیر محلی فقط درون آن تابع استفاده می‌شود.

stack diagram (نمودار پشته)

نمایش گرافیکی پشته‌ای از توابع، متغیرهایشان و مقادیری که به آنها رجوع می‌کنند.

frame (قاب)

کادر مستطیل شکلی که فراخوانی را در یک نمودار پشته نمایش می‌دهد. قاب شامل متغیرهای محلی و پارامترهای تابع می‌باشد.

traceback (پس‌یابی)

لیستی از توابع در حال اجرا که در هنگام وقوع یک خطای زمان اجرا چاپ می‌شوند.

شرطی‌ها و بازگشت



در فصل گذشته آموختید که چگونه یک تابع را طراحی و با آن کار کنید، اما هنوز هم به اطلاعات بیشتری در مورد نحوه ساخت و طرز استفاده از آنها نیاز دارید.

در این فصل علاوه بر بررسی توابع، با یکی دیگر از مفاهیم بنیادی زبان‌های برنامه‌نویسی به‌نام عبارات شرطی و منطقی آشنا می‌شوید. تصمیم‌گیری در زبان‌های برنامه‌نویسی از جمله خصیصه‌هایی است که تقریباً در تمامی برنامه‌های کاربردی به چشم می‌خورد. یک برنامه کاربردی، به همان میزان انعطاف‌پذیری، از تنوع تصمیم‌گیری برای بررسی حالت‌های مختلف برخوردار است.

علاوه بر بررسی این خصیصه در زبان پایتون، در این فصل با ساختار توابع بازگشتی آشنا می‌شوید. با استفاده از این ساختار قادر خواهید بود بسیاری از برنامه‌های پیچیده را با روندی ساده‌تر پیاده‌سازی کنید.

۴-۱- عملگر باقی‌مانده

عملگر باقی‌مانده بر روی اعداد صحیح (و عبارات صحیح) کار می‌کند و باقی‌مانده را پس از تقسیم عدد اول بر عدد دوم، نتیجه می‌دهد. در پایتون عملگر باقی‌مانده با علامت درصد (%) نمایش داده می‌شود. الگوی استفاده آن درست شبیه دیگر عملگرها است:

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

بنابراین نتیجه تقسیم 7 بر 3 برابر با 2 است و باقی‌مانده برابر با 1 می‌باشد.

ممکن است این عملگر به ظاهر کاربردی نداشته باشد، اما چنان‌که خواهید دید بسیار مفید است. برای مثال شما می‌توانید بخش‌پذیری یک عدد بر عددی دیگر را بررسی کنید. اگر نتیجه $x\%y$ صفر باشد، پس x بر y بخش‌پذیر است.

همچنین شما می‌توانید رقم یا رقم‌های سمت راست یک عدد را به‌دست آورید. برای نمونه $x\%10$ رقم سمت راست عدد x (در مبنای ۱۰) را نتیجه می‌دهد و به‌طور مشابه $x\%100$ دو رقم سمت راست x را منتج می‌شود.

۴-۲- عبارات بولی

یک عبارت بولی عبارتی است که درست (true) یا نادرست (false) است. در پایتون عبارتی که صحیح است مقدار 1 و عبارتی که نادرست است مقدار 0 را دارا است. عملگر == دو مقدار را مقایسه و یک مقدار بولی را تولید می‌کند:

```
>>> 5 == 5
1
>>> 5 == 6
0
```

در دستور اول دو عملوند با هم برابرند بنابراین عبارت، 1 (true) ارزیابی می‌شود و در دومین دستور 5 با 6 برابر نیست و لذا نتیجه 0 (false) است. عملگر == یکی از عملگرهای مقایسه‌ای است. دیگر عملگرهای مقایسه‌ای عبارتند از:

$x \neq y$	x با y برابر نیست یا x مخالف y است
$x > y$	x از y بزرگتر است
$x < y$	x از y کوچکتر است
$x \geq y$	x بزرگتر یا مساوی y است
$x \leq y$	x کوچکتر یا مساوی y است

اگرچه ممکن است این عملگرها برایتان آشنا باشند، اما نمادهای پایتون با نمادهای ریاضی متفاوتند. یکی از خطاهای معمول، استفاده از علامت = به جای علامت == است. به‌خاطر داشته باشید که = یک عملگر نسبت‌دهی و == یک عملگر مقایسه‌ای است. لازم به ذکر است که نمادهای >= (بزرگ‌تر یا مساوی) و <= (کوچک‌تر یا مساوی) در پایتون، تنها به همین صورت قابل استفاده‌اند و صورت‌های دیگری نظیر > یا < که ممکن است در ریاضی استفاده شود، غیرمجاز می‌باشند.

۴-۳- عملگرهای منطقی

در پایتون سه عملگر منطقی وجود دارد: and، or و not. مفهوم این عملگرها شبیه به معانی‌شان در زبان انگلیسی است. برای مثال، $0 < x$ and $x < 10$ تنها زمانی درست است که x بزرگ‌تر از صفر و کوچک‌تر از ده باشد.

$n \% 2 == 0$ or $n \% 3 == 0$ درست است مگر اینکه هر دو عبارت نادرست باشد. پس اگر یکی از اعداد بر 2 یا بر 3 بخش پذیر باشد عبارت صحیح است.

و در آخر، عملگر **not** یک عبارت بولی را نقیض می کند، بنابراین اگر $x > y$ غلط باشد و یا به عبارت دیگر x کوچک تر یا مساوی y باشد، $\text{not}(x > y)$ درست است.

در حالت کلی و دقیق، عملوندهای یک عملگر منطقی باید عبارتی بولی باشند اما پایتون زیاد سختگیر نیست. هر عدد غیر صفر به عنوان **true** تفسیر می شود.

```
>>> x = 5
>>> x and 1
1
>>> y = 0
>>> y and 1
0
```

در کل، این قسم از عبارات به روش خوبی مورد توجه قرار نگرفته است. اگر شما بخواهید مقداری را با 0 مقایسه کنید، باید این کار را با صراحت انجام دهید.

۴-۸- اجرای عبارات شرطی

برای نوشتن یک برنامه مفید، تقریباً همیشه به یک توانایی برای بررسی شروط و تغییر رفتار برنامه بر اساس آنها نیازمندیم. **دستورات شرطی** این توانایی را به ما می دهند. ساده ترین دستور شرطی **if** است:

```
if x > 0:
    print "x is positive"
```

عبارت بولی که بعد از دستور **if** قرار می گیرد، شرط نامیده می شود. اگر شرط درست باشد، دستور کنگره گذاری شده اجرا می شود و در غیر این صورت هیچ اتفاقی نمی افتد.

همچون دیگر **دستورات مرکب**، دستور شرطی **if** نیز از یک عنوان و بلوکی از دستورات تشکیل شده است:

```
HEADER:
FIRST STATEMENT
...
LAST STATEMENT
```


عنوان (**Header**)، از یک خط جدید آغاز می‌شود و با یک کولن (:) پایان می‌یابد. دستورات کنگره‌گذاری شده که در ادامه می‌آیند یک **بلوک** نامیده می‌شوند. اولین دستوری که از کنگره‌گذاری خارج شود، پایان بلوک را مشخص می‌کند. گزاره‌ای (یا گزاره‌هایی) که درون یک دستور مرکب محصور شده است، **بدنه** نامیده می‌شود.

هیچ محدودیتی در تعداد دستوراتی که در بدنه یک دستور **if** قرار می‌گیرد وجود ندارد، اما وجود حداقل یک دستور الزامی است. بعضی اوقات داشتن بدنه‌ای بدون دستور، مفید است (معمولاً جهت نگه داشتن جا برای کدی که هنوز ننوشته‌اید). در آن صورت می‌توانید در قسمت بدنه از دستور **pass** که هیچ عملی انجام نمی‌دهد استفاده کنید.

۴-۵- اجرای انتخاب‌های دوگانه

دومین فرم دستور **if** اجرای انتخاب‌های دوگانه است که در آن دو احتمال وجود دارد و شرط معین می‌کند کدامیک اجرا شود. الگوی این حالت به صورت زیر است:

```
if x%2 == 0:
    print x, "is even"
else:
    print x, "is odd"
```

می‌دانیم که وقتی باقی‌مانده تقسیم **x** بر 2، صفر است، **x** زوج می‌باشد و برنامه پیغامی را مبنی بر این نتیجه نمایش می‌دهد. اگر شرط غلط باشد، دومین دسته از دستورات اجرا می‌شوند. از آنجایی که شرط باید **true** یا **false** باشد، دقیقاً یکی از انتخاب‌ها اجرا خواهد شد. انتخاب‌ها **شاخه** هم نامیده می‌شوند، زیرا آنها همچون شاخه‌هایی در روند اجرا هستند. به‌علاوه اگر شما به بررسی زوج یا فرد بودن یک عدد نیاز دارید، می‌توانید آن را در یک بسته‌بندی قرار دهید. این بسته‌بندی یک تابع است:

```
def printParity(x):
    if x%2 == 0:
        print x, "is even"
    else:
        print x, "is odd"
```

برای هر مقدار **x**، **printParity** یک پیغام مناسب نمایش می‌دهد. وقتی که شما آن را فراخوانی می‌کنید، می‌توانید هر عبارت صحیحی را به عنوان آرگومان به آن بدهید:

```
>>> printParity(17)
17 is odd
>>> y=21
>>> printParity(y+1)
22 is even
```

۴-۶- دستورات شرطی زنجیره‌ای

در بعضی مواقع بیش از دو احتمال وجود دارد و ما به بیش از دو شاخه نیاز داریم. یکی از راه‌های بیان چنین محاسباتی دستورات شرطی زنجیره‌ای است:

```
if x < y:
    print x, "is less than", y
elif x > y:
    print x, "is greater than", y
else:
    print x, "and", y, "are equal"
```

elif مخفف دو کلمه **else** و **if** است. باز هم تنها یکی از شاخه‌ها اجرا می‌شود. هیچ محدودیتی در تعداد دستورات **elif** وجود ندارد اما **else** (در صورت وجود) باید آخرین شاخه باشد:

```
if choice == 'A':
    functionA()
elif choice == 'B':
    functionB()
elif choice == 'C':
    functionC()
else:
    print "Invalid choice."
```

همه شرط‌ها به ترتیب بررسی می‌شود. اگر اولین شرط **false** باشد، شرط بعدی بررسی می‌شود و به همین ترتیب.

اگر یکی از شرط‌ها درست باشد، شاخه متناظر با آن اجرا می‌شود و دستور پایان می‌یابد. حتی اگر بیش از یک شرط درست داشته باشیم تنها اولین شاخه درست اجرا می‌شود.

تمرین ۴-۱: این مثال‌ها را در توابعی به نام **compare(x,y)** و **dispatch(choice)** بسته‌بندی نمایید.

۴-۷- دستورات شرطی تودرتو

یک گزاره شرطی می‌تواند به صورت **تودرتو** خود در گزاره شرطی دیگری قرار گیرد. ما توانستیم یک مثال سه قسمتی بنویسیم که اعمال زیر را انجام دهد:

```
if x == y:
    print x, "and", y, "are equal"
else:
    if x < y:
        print x, "is less than", y
    else:
        print x, "is greater than", y
```

گزاره شرطی بیرونی شامل دو شاخه است. اولین شاخه یک دستور خروجی ساده دارد، اما شاخه دوم شامل یک دستور **if** است که خود دارای دو شاخه است. آن دو شاخه هر کدام یک دستور خروجی هستند، اگرچه می‌توانند باز هم یک گزاره شرطی باشند. با اینکه کنگره‌گذاری دستورات ساختار برنامه را آشکار می‌سازد، تودرتو کردن دستورات شرطی خواندن برنامه را دشوار می‌کند. در مجموع بهتر است تا آنجا که می‌توانید، از دستورات شرطی تودرتو پرهیز نمایید. عملگرهای منطقی اغلب راهی را برای ساده کردن دستورات شرطی تودرتو فراهم می‌کنند. برای نمونه ما می‌توانیم کد زیر را در یک دستور شرطی واحد بنویسیم:

```
if 0 < x:
    if x < 10:
        print "x is a positive single digit."
```

دستور **print** تنها زمانی اجرا می‌شود که ما هر دو شرط را پشت سر گذاشته باشیم. لذا می‌توانیم از عملگر **and** استفاده کنیم:

```
if 0 < x and x < 10:
    print "x is a positive single digit."
```

این نوع دستورات شرطی بسیار رایجند. بنابراین پایتون نحوه نگارشی شبیه به نمادگذاری ریاضی فراهم می‌کند:

```
if 0 < x < 10:
    print "x is a positive single digit."
```

این شرط از نظر معنایی با عبارت بولی مرکب و شرطهای تودرتو شباهت دارد.

۸-۴- دستور return

دستور **return** به شما این امکان را می‌دهد که قبل از رسیدن به آخر تابع اجرای آن را متوقف کنید. یک دلیل استفاده از این امکان آن است که شما یک وضعیت خطا را نمایش دهید:

```
import math

def printLogarithm(x):
    if x <= 0:
        print "Positive numbers only, please."
        return

    result = math.log(x)
    print "The log of x is", result
```

تابع **printLogarithm** پارامتری با نام **x** می‌گیرد. اولین چیزی که بررسی می‌شود این است که آیا **x** کوچک‌تر یا مساوی ۰ است یا نه؛ اگر چنین بود یک پیغام خطا نمایش داده می‌شود و سپس دستور **return** برای خروج از تابع اجرا می‌شود. روند اجرا فوراً به فراخوان تابع بر می‌گردد و خطوط باقی‌مانده تابع اجرا نمی‌شود. به خاطر داشته باشید که برای استفاده از توابع ماژول **math** باید این ماژول را **import** کنید.

۹-۴- توابع بازگشتی

ما اشاره کردیم که یک تابع می‌تواند توسط تابع دیگری فراخوانی شود و شما چندین مثال را در این مورد ملاحظه کردید.

جالب‌تر اینکه یک تابع می‌تواند خودش را هم فراخوانی کند. ممکن است مزایای چنین امکانی به خوبی معلوم نباشد، اما این یکی از جذاب‌ترین و جالب‌ترین کارهایی است که یک برنامه می‌تواند انجام دهد. برای نمونه به تابع زیر بنگرید:

```
def countdown(n):
    if n == 0:
        print "Blastoff!"
    else:
        print n
        countdown(n-1)
```

تابع **countdown** انتظار دارد که پارامتر **n** مثبت باشد. اگر **n** صفر باشد خروجی، کلمه **Blastoff!** است، در غیر این صورت خروجی **n** است و سپس تابعی به نام **countdown**

(خودش) فراخوانی می‌شود و **n-1** به عنوان آرگومانی به آن فرستاده می‌شود. اگر تابعی مانند تابع زیر را فراخوانی کنید چه اتفاقی می‌افتد؟

```
>>> countdown(3)
```

اجرای **countdown** با **n=3** شروع می‌شود و چون **n** صفر نیست، خروجی مقدار **3** می‌باشد و سپس تابع خودش را صدا می‌زند...

اجرای **countdown** با **n=2** شروع می‌شود و چون **n** صفر نیست، خروجی مقدار **2** می‌باشد و سپس تابع خودش را صدا می‌زند...

اجرای **countdown** با **n=1** شروع می‌شود و چون **n** صفر نیست، خروجی مقدار **1** می‌باشد و سپس تابع خودش را صدا می‌زند...

اجرای **countdown** با **n=0** شروع می‌شود و چون **n** برابر با **0** است، خروجی

کلمه **Blastoff!** است و سپس باز می‌گردد.

آن تابع **countdown** که پارامتر **n=1** را گرفته بود باز می‌گردد.

آن تابع **countdown** که پارامتر **n=2** را گرفته بود باز می‌گردد.

آن تابع **countdown** که پارامتر **n=3** را گرفته بود هم باز می‌گردد.

و سپس شما به **__main__** بر می‌گردید (چه سفری!!). بنابراین کل خروجی که دیده می‌شود به این صورت است:

```
3
2
1
Blastoff!
```

به عنوان دومین مثال دوباره به تابع **newLine** و **threeLines** نگاه کنید:

```
def newLine():
    print

def threeLines():
    newLine()
    newLine()
    newLine()
```

اگر چه این توابع کار می‌کنند اما اگر بخواهیم خروجی ۲ یا ۱۰۶ خط جدید باشد، کمکی به ما نمی‌کنند. چاره بهتر آن است که تابعی به این صورت بنویسیم:

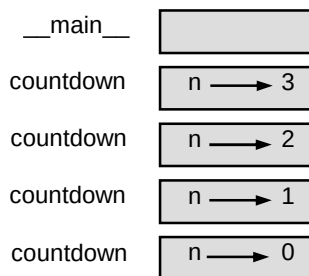
```
def nLines(n):
    if n > 0:
        print
        nLines(n-1)
```

این برنامه شبیه به **countdown** است؛ تا زمانی که **n** بزرگ‌تر از ۰ است، خروجی یک خط جدید است و سپس تابع خودش را جهت چاپ **n-1** خط جدید دیگر فراخوانی می‌کند. بدین‌سان تعداد کل خطوط جدید **1+(n-1)** است که اگر محاسبات را درست انجام دهید به نتیجه **n** خواهید رسید!

فرایند فراخوانی یک تابع توسط خود آن تابع را **بازگشت** و این‌گونه توابع را بازگشتی می‌نامند.

۱-۴-۱- نمودارهای پشته برای توابع بازگشتی

در بخش ۳-۱۱، ما از یک نمودار پشته برای نمایش دادن حالت یک برنامه در هنگام فراخوانی تابع استفاده کردیم. همان نوع نمودار می‌تواند به تفسیر یک تابع بازگشتی کمک کند. هر وقت که تابع فراخوانی می‌شود پایتون یک قاب برای تابع جدید می‌سازد که شامل متغیرها و پارامترهای محلی تابع است. برای یک تابع بازگشتی ممکن است در یک زمان بیش از یک قاب بر روی پشته داشته باشیم. شکل ۴-۱، نمودار پشته‌ای را برای **countdown** نمایش می‌دهد که با **n=3** فراخوانی شده است:



شکل ۴-۱

همیشه بالای پشته قابی برای **__main__** قرار دارد. این قاب خالی است زیرا ما هیچ متغیری در **__main__** نساخته‌ایم و یا هیچ پارامتری به آن نفرستاده‌ایم. چهار قاب **countdown** مقادیر متفاوتی برای **n** دارند. پایین پشته در جایی که **n=0** است، **حالت مبنا** نامیده می‌شود. این قاب به صورت بازگشتی فراخوانی نمی‌شود بنابراین قاب‌های بیشتری وجود ندارد.

تمرین ۴-۲: یک نمودار پشته برای تابع `nLines` با مقدار `n=4` رسم کنید.

۱۱-۴- بازگشت بی‌انتها

اگر یک بازگشت هیچ‌گاه به حالت مبنا نرسد، تابع بازگشتی برای همیشه فراخوانی می‌شود و لذا برنامه هیچ‌گاه پایان نمی‌یابد. از این مطلب به عنوان **بازگشت بی‌انتها** یاد می‌شود و به‌طور کلی ایده خوبی نیست.

در اینجا کوچک‌ترین برنامه با یک بازگشت بی‌انتها را مشاهده می‌کنید:

```
def recurse():
    recurse()
```

در اغلب محیط‌های برنامه‌نویسی برنامه‌ای با یک بازگشت بی‌انتها واقعاً برای همیشه اجرا نمی‌شوند. پایتون وقتی که به حداکثر عمق بازگشت رسید یک پیغام خطا گزارش می‌دهد:

```
File "<stdin>", line 2, in recurse
(۹۸ تکرار حذف شده)
File "<stdin>", line 2, in recurse
RuntimeError: Maximum recursion depth exceeded
```

این پس‌پایی کمی از آنچه در فصل قبل دیدیم بزرگ‌تر است. وقتی خطا رخ می‌دهد، صد قاب بازگشتی در پشته وجود دارد. بنابراین در عمل خط اول پیغام فوق صد مرتبه تکرار می‌گردد که در اینجا ۹۸ تکرار آن حذف شده است.

تمرین ۴-۳: تابعی با بازگشت بی‌کران بنویسید و آن را در مفسر پایتون اجرا کنید.

۱۲-۴- ورودی صفحه‌کلید

برنامه‌هایی که تاکنون نوشته‌ایم، از این بابت که هیچ ورودی از کاربر دریافت نمی‌کردند کمی خشک و غیرقابل انعطاف هستند. آنها هر بار یک کار مشابه انجام می‌دهند. پایتون توابع پیش‌ساخته‌ای برای گرفتن ورودی از صفحه‌کلید تدارک دیده است. ساده‌ترین تابع `raw_input` نام دارد. وقتی این تابع فراخوانده می‌شود برنامه متوقف می‌شود و منتظر می‌ماند تا کاربر چیزی را وارد کند. وقتی که کاربر کلید **Enter** را فشار داد، اجرای برنامه ادامه می‌یابد و `raw_input` کاراکترهای وارد شده توسط کاربر را به‌عنوان یک رشته برمی‌گرداند:

```
>>> input = raw_input ()
What are you waiting for?
>>> print input
What are you waiting for?
```

قبل از فراخوانی **raw_input** بد نیست با چاپ پیغامی به کاربر گزارش دهیم که چه چیزی را وارد کند. این پیغام **اعلان** نام دارد. ما می‌توانیم اعلانی را به‌عنوان یک آرگومان به **raw_input** بدهیم:

```
>>> name = raw_input ("What...is your name? ")
What...is your name? Cyrus, King of the World!
>>> print name
Cyrus, King of the World!
```

اگر بخواهیم جواب، یک عدد صحیح باشد، می‌توانیم از تابع **input** استفاده کنیم:

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

علامت “\” که در پایان خط اول بکار رفته به ما اجازه می‌دهد ادامهٔ این رشته را در خط بعد بیاوریم. بنابراین دو خط اول این کد یک دستور به‌شمار می‌روند.

اگر کاربر دنباله‌ای از ارقام را تایپ کند، آنها به یک عدد صحیح تبدیل می‌شوند و به متغیر **speed** اختصاص می‌یابد. متأسفانه اگر کاربر کاراکتری را تایپ کند که رقم نباشد برنامه یک پیغام خطا نمایش می‌دهد:

```
>>> speed = input (prompt)
What...is the airspeed velocity of an unladen swallow?
What do you mean, an African or a European swallow?
SyntaxError: invalid syntax
```

برای جلوگیری از این‌گونه خطاها بهتر است از تابع **raw_input** استفاده کنیم و مقداری به عنوان یک رشته بگیریم و سپس توسط توابع مبدل، آن را به انواع دیگر داده تبدیل کنیم.

modulus operator (عملگر باقی‌مانده)

عملگری که با علامت درصد (%) نشان داده می‌شود و بر روی اعداد صحیح کار می‌کند. این عملگر، باقی‌مانده تقسیم یک عدد صحیح بر یک عدد صحیح دیگر را نتیجه می‌دهد.

boolean expression (عبارت بولی)

عبارتی که یا درست (true) و یا نادرست (false) است.

comparison operator (عملگر مقایسه‌ای)

یکی از عملگرهایی که دو مقدار را با هم مقایسه می‌کند: >, <, <=, >=, !=, ==

logical operator (عملگر منطقی)

دستوری مرکب از عنوان و بدنه. عنوان با یک کولن (:) پایان می‌یابد. بدنه نسبت به عنوان تورفتگی دارد.

conditional statement (گزاره شرطی)

دستوری که روند اجرا را بر اساس برخی شروط کنترل می‌کند.

condition (شرط)

عبارتی بولی که در یک گزاره شرطی مشخص می‌کند کدام شاخه اجرا شود.

compound statement (دستور مرکب)

گزاره‌ای شرطی که شامل عنوان و بدنه است. عنوان با یک علامت (:) خاتمه می‌یابد و بدنه نسبت به عنوان دارای تورفتگی است.

block (بلوک)

مجموعه‌ای از دستورات پیاپی با تورفتگی مشابه.

body (بدنه)

بلوکی در یک دستور مرکب که به دنبال عنوان می‌آید.

branches (شاخه‌ها)

انتخاب‌هایی که در روند اجرای یک دستور شرطی به طور مستقل اجرا می‌شوند.

nested (تودرتو)

یک ساختار برنامه در ساختاری دیگر، مانند یک دستور شرطی درون شاخه‌ای از یک دستور شرطی دیگر.

recursion (بازگشت)

فراخوانی تابعی که در حال اجرا است (معمولاً توسط خودش).

base case (حالت مبنا)

شاخه‌ای از دستورات شرطی در یک تابع بازگشتی که منجر به یک فراخوانی بازگشتی نمی‌شود.

infinite recursion (بازگشت بی‌انتهای)

تابعی که به صورت بازگشتی خود را فراخوانی می‌کند اما هیچگاه به حالت مبنا نمی‌رسد. یک بازگشت بی‌انتهای سرانجام منجر به یک خطای زمان اجرا می‌شود.

prompt (اعلان)

یک اشاره بصری که به کاربر می‌گوید چه داده را وارد کند.

توابع نتیجه‌دار



همان طور که در فصل های گذشته نیز گفته شد، توابع از مهم ترین مفاهیم زبان های برنامه نویسی هستند. توابعی که تا اینجا آموخته اید، تنها عمل مشخصی را انجام داده و پایان می یابند. در این فصل قصد داریم توابعی را بررسی کنیم که علاوه بر انجام مجموعه ای از دستورات، مقدار یا مقادیری را تولید کنند و به عنوان نتیجه برگردانند. در پایان این گونه توابع را به صورت بازگشتی استفاده می کنیم.

۵-۱- مقادیر برگشتی

برخی از توابع پیش ساخته را که تاکنون استفاده کرده ایم، همچون توابع ریاضی نتایجی را تولید کرده اند. فراخوانی تابع، مقدار جدیدی تولید می کند که ما معمولاً به یک متغیر نسبت می دهیم و یا به عنوان قسمتی از یک عبارت به کار می بریم.

```
e = math.exp(1.0)
height = radius * math.sin(angle)
```

اما هیچ کدام از توابعی که نوشته ایم، تاکنون مقداری را باز نگردانده اند.

در این فصل، ما قصد داریم توابعی بنویسیم که مقادیری را بازگردانند و آنها را **توابع نتیجه دار** می نامیم. اولین مثال، تابع مساحت است که با گرفتن شعاع یک دایره، مساحت آن را برمی گرداند:

```
import math

def area(radius):
    temp = math.pi * radius**2
    return temp
```

ما قبلاً با دستور **return** برخورد کرده ایم، اما در یک تابع نتیجه دار دستور **return** شامل یک مقدار برگشتی است. این دستور یعنی: «فوراً از این تابع برگرد و از عبارتی که در ادامه آمده است به عنوان مقدار برگشتی استفاده کن.» عبارتی که ارائه می شود می تواند تا هر مقدار دلخواه پیچیده باشد. بنابراین می توانیم این تابع را خلاصه تر بنویسیم:

```
def area(radius):
    return math.pi * radius**2
```

از سوی دیگر، متغیرهای موقتی همچون **temp** عیب یابی را ساده تر می سازد. استفاده از چند دستور **return** به طوری که هر کدام در یک شاخه قرار گیرد، در بعضی مواقع مفید است:

```
def absoluteValue(x):
    if x < 0:
        return -x
    else:
        return x
```

از آنجا که این دستورات **return** در یک گزاره شرطی دوگانه قرار دارند، تنها یکی از آنها اجرا می‌شود. به محض اینکه یکی اجرا شود، تابع بدون اجرای دستورات زیرین پایان می‌یابد. کدی که بعد از دستور **return** (یا هر مکانی که روند اجرا نمی‌تواند هیچ‌گاه به آن برسد) قرار گیرد، **کد مرده** نامیده می‌شود. در یک تابع نتیجه‌دار بهتر است مطمئن شویم که تمام مسیرهای ممکن برنامه به یک دستور **return** ختم شده‌اند. برای مثال:

```
def absoluteValue(x):
    if x < 0:
        return -x
    elif x > 0:
        return x
```

این برنامه صحیح نیست، زیرا اگر x صفر باشد هیچ‌کدام از شرطها درست نیست و تابع بدون رسیدن به دستور **return** پایان می‌یابد. در این صورت، مقدار برگشتی، مقدار ویژه‌ای به نام **None** می‌باشد:

```
>>> print absoluteValue(0)
None
```

تمرین ۵-۱: یک تابع مقایسه‌ای به نام **compare** بنویسید که اگر $x > y$ ، ۱، اگر $x == y$ ، ۰ و اگر $x < y$ باشد، -۱ را برگرداند.

۵-۲- توسعه برنامه

در اینجا، شما باید بتوانید به یک تابع کامل بنگرید و بگویید که چه کاری انجام می‌دهد. همچنین اگر تمرین‌ها را انجام داده باشید، توابع کوچکی هم نوشته‌اید. همچنان که نوشتن توابع بزرگ‌تر را شروع کنید، ممکن است با دشواری بیشتری روبرو شوید و مشکلات بیشتری با خطاهای زمان اجرا و خطاهای معنایی داشته باشید.

به منظور بحث در مورد توسعه برنامه‌های پیچیده، ما تکنیکی با عنوان **توسعه افزایشی** ارائه می‌دهیم. هدف از توسعه افزایشی برنامه، اجتناب از اختصاص زمانی طولانی برای اشکال‌یابی از طریق اضافه و آزمایش کردن قسمت‌های کوچک کد در هر زمان است.

به عنوان مثالی، فرض کنید می‌خواهید فاصله میان دو نقطه با مختصات (x_1, y_1) و (x_2, y_2) را پیدا کنید. با توجه به رابطه فیثاغورث این فاصله برابر است با:

$$\text{distance} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

گام اول آن است که تشخیص دهیم در پایتون یک تابع **distance** باید چه شکلی داشته باشد. به بیان دیگر، باید مشخص کنیم ورودی‌ها (پارامترها) و خروجی‌های (مقادیر برگشتی) تابع چه هستند.

در این مثال، ورودی‌ها دو نقطه هستند که ما می‌توانیم آنها را با چهار پارامتر نمایش دهیم و مقدار برگشتی، فاصله است که مقداری اعشاری دارد. تا اینجا می‌توانیم خطوط اصلی تابع را بنویسیم:

```
def distance(x1, y1, x2, y2):
    return 0.0
```

واضح است که این نسخه از تابع، فاصله را محاسبه نمی‌کند و همواره صفر را بر می‌گرداند اما این تابع از لحاظ نحوی درست است و کار می‌کند. یعنی ما می‌توانیم قبل از اینکه تابع را پیچیده‌تر کنیم آن را آزمایش نماییم. جهت آزمایش تابع جدید، آن را با مقادیر نمونه فراخوانی می‌کنیم:

```
>>> distance(1, 2, 4, 6)
0.0
```

ما این مقادیر را انتخاب کرده‌ایم، تا فاصله افقی برابر 3 و فاصله عمودی برابر 4 شود؛ با این روش، نتیجه 5 می‌باشد (مثلثی با اضلاع 3-4-5). هنگام آزمایش یک تابع، خوب است که جواب درست را بدانیم.

تا اینجا مطمئن شدیم که تابع از لحاظ نحوی صحیح است و حال می‌توانیم شروع به اضافه کردن خطوط کد کنیم. بعد از هر مرحله تغییر، تابع را دوباره آزمایش می‌کنیم. اگر خطایی در هر لحظه رخ دهد، ما می‌دانیم که مشکل در چه قسمتی است - در خطی که اخیراً اضافه شده -.

اولین گام منطقی در محاسبه، پیدا کردن تفاضل‌های $x_2 - x_1$ و $y_2 - y_1$ است. ما این مقادیر را در متغیرهای موقتی به نام dx و dy ذخیره می‌کنیم و آنها را چاپ می‌نماییم:

```
def distance(x1, y1, x2, y2):  
    dx = x2 - x1  
    dy = y2 - y1  
    print "dx is", dx  
    print "dy is", dy  
    return 0.0
```

اگر تابع کار می‌کند، خروجی باید اعداد 3 و 4 (و البته مقدار برگشتی 0.0) باشد. در این صورت ما می‌فهمیم که تابع پارامترها را به درستی دریافت کرده و اولین محاسبه را به‌طور صحیح انجام داده است. در غیر این صورت تنها چند خط برای بررسی وجود دارد.
سپس مجموع مجذورات dx و dy را محاسبه می‌کنیم:

```
def distance(x1, y1, x2, y2):  
    dx = x2 - x1  
    dy = y2 - y1  
    dsquared = dx**2 + dy**2  
    print "dsquared is: ", dsquared  
    return 0.0
```

دقت کنید که ما دستور **print** را که در مرحله قبل نوشته بودیم، حذف کرده‌ایم. کدهایی شبیه به این را **داربست** می‌نامند زیرا در ساختن برنامه مفیدند اما در نتیجه نهایی وجود ندارند. در این مرحله مجدداً برنامه را اجرا می‌کنیم و خروجی را (که باید 25 باشد) بررسی می‌کنیم. در نهایت اگر ماژول **math** را وارد کرده باشیم، می‌توانیم از تابع **sqrt** جهت محاسبه و برگرداندن نتیجه استفاده کنیم.

```
def distance(x1, y1, x2, y2):  
    dx = x2 - x1  
    dy = y2 - y1  
    dsquared = dx**2 + dy**2  
    result = math.sqrt(dsquared)  
    return result
```

اگر برنامه کار کند، کار شما تمام است. در غیر این صورت می‌توانید مقدار **result** را قبل از دستور **return** چاپ کنید.

وقتی که شروع به نوشتن برنامه می‌کنید، در هر مرحله باید تنها یک یا دو خط کد را اضافه نمایید. هنگامی که تجربه بیشتری به دست آوردید، می‌توانید قطعات کد بزرگتری را بنویسید و اشکال‌زدایی کنید. از این طریق فرایند توسعه افزایشی می‌تواند تا حد زیادی از اتلاف وقت در اشکال‌زدایی جلوگیری کند.

نکات کلیدی این فرایند در زیر آمده است:

- با یک برنامه مقدماتی کار را آغاز کنید و گام به گام تغییرات کوچکی را بوجود آورید. در هر لحظه اگر خطایی وجود داشته باشد، شما از محل وقوع آن خطا مطلع خواهید شد.
- ۵. جهت نگهداری مقادیر واسطه از متغیرهای موقتی استفاده کنید. بدین صورت شما می‌توانید آنها را در خروجی چاپ و بررسی کنید.
- ۶. همین که دیدید برنامه کار می‌کند، می‌توانید داربست‌ها را حذف کنید یا دستورات چندگانه را در عبارات مرکب ادغام نمایید، اما تنها در صورتی که خواندن برنامه دشوارتر نشود.

تمرین ۵-۲: از روش توسعه افزایشی به منظور نوشتن تابعی به نام **hypotenuse** استفاده کنید. این تابع باید طول وتر یک مثلث قائم‌الزاویه را با گرفتن طول دو ساقش (به عنوان پارامتر) برگرداند. تمام مراحل تدریجی نوشتن تابع را ذخیره کنید.

۵-۳- ترکیب

همان‌طور که اکنون باید انتظار داشته باشید، شما می‌توانید یک تابع را از درون تابع دیگر فراخوانی کنید. این توانایی ترکیب نامیده می‌شود. برای مثال، ما تابعی خواهیم نوشت که مرکز یک دایره و نقطه‌ای روی محیط آن را بگیرد و با استفاده از این دو نقطه مساحت دایره را محاسبه کند. فرض کنید مختصات نقطه مرکزی در دو متغیر **xc** و **yc** و مختصات نقطه محیطی در متغیرهای **xp** و **yp** ذخیره شده است. گام اول پیدا کردن شعاع دایره است که برابر با فاصله دو نقطه می‌باشد. خوشبختانه تابعی به نام **distance** داریم که این کار را انجام می‌دهد:

radius = distance(xc, yc, xp, yp)

گام دوم یافتن مساحت با استفاده از این شعاع و برگرداندن آن است:


```
result = area(radius)
return result
```

از ادغام این دستورات درون این تابع داریم:

```
def area2(xc, yc, xp, yp):
    radius = distance(xc, yc, xp, yp)
    result = area(radius)
    return result
```

ما این تابع را **area2** نامیدیم تا بتوان آن را از تابع **area** (که قبلاً آن را تعریف کرده‌ایم) تمیز داد. در یک ماژول مشخص تنها یک تابع با یک نام معین می‌تواند وجود داشته باشد. متغیرهای موقتی **radius** و **result** جهت توسعه و اشکال‌زدایی برنامه مفیدند، اما همین‌که برنامه کار کرد می‌توانیم آن را با ترکیب فراخوانی‌های تابع خلاصه‌تر کنیم:

```
def area2(xc, yc, xp, yp):
    return area(distance(xc, yc, xp, yp))
```

تمرین ۳-۵: تابع **slope(x1, y1, x2, y2)** را طوری بنویسید که شیب خط حاصل از دو نقطه **(x1, y1)** و **(x2, y2)** را برگرداند. سپس از این تابع در ساخت تابع دیگری با عنوان **intercept(x1, y1, x2, y2)** طوری استفاده کنید که عرض از مبدا خط فوق برگردانده شود.

۵-۴- توابع بولی

توابع می‌توانند مقادیر بولی را بازگردانند که اغلب برای مخفی کردن محاسبات آزمایشی پیچیده درون توابع مناسب است. برای مثال:

```
def isDivisible(x, y):
    if x % y == 0:
        return 1      # it's true
    else:
        return 0      # it's false
```

نام این تابع `isDivisible` می‌باشد. معمول است که نام توابع بولی را چیزی شبیه به سؤالات بلی/خیر انتخاب کنند. تابع `isDivisible` یکی از دو مقدار 1 یا 0 را برمی‌گرداند تا نشان دهد `x` بر `y` بخش پذیر است یا خیر.

ما می‌توانیم با استفاده از این واقعیت که دستور شرطی `if` خود یک عبارت بولی است، تابع را فشرده‌تر سازیم و بدون استفاده از دستور `if` مستقیماً آن را برگردانیم:

```
def isDivisible(x, y):
    return x % y == 0
```

این بخش رفتار تابع جدید را نشان می‌دهد:

```
>>> isDivisible(6, 4)
0
>>> isDivisible(6, 3)
1
```

توابع بولی اغلب در دستورات شرطی استفاده می‌شوند:

```
if isDivisible(x, y):
    print "x is divisible by y"
else:
    print "x is not divisible by y"
```

`if` ممکن است شما را به نوشتن کدی شبیه به این وسوسه کند:

```
if isDivisible(x, y) == 1:
```

اما مقایسهٔ زائد لازم نیست.

تمرین ۵-۴: تابع `isBetween(x, y, z)` را طوری بنویسید که اگر `x ≤ y ≤ z` برقرار بود، 1 و در غیر این صورت 0 را برگرداند.

۵-۵- بازگشت نتیجه‌دار

تاکنون، تنها زیرمجموعهٔ کوچکی از پایتون را آموخته‌اید اما شاید جالب باشد که بدانید این زیرمجموعهٔ کوچک یک زبان برنامه‌نویسی کامل است. یعنی به وسیلهٔ این زبان می‌توانید هر چیزی را که قابل محاسبه است، بیان کنید. هر برنامه‌ای که تاکنون نوشته شده می‌تواند تنها با استفاده از ویژگی‌هایی از زبان که تا این زمان آموخته‌اید بازنویسی شود. (در واقع، شما به تعداد محدودی از

دستورات نیاز دارید تا بتوانید واحدهایی همچون صفحه کلید، ماوس، دیسک‌ها و ... را کنترل کنید. اما این آخر ماجرا است!

اثبات این ادعا که اولین بار توسط «آلن تورینگ» انجام شده، تمرین قابل توجهی است (او یکی از اولین دانشمندان علم کامپیوتر است. علی‌رغم اینکه بعضی عقیده دارند «آلن تورینگ» یک ریاضی‌دان است اما بسیاری از دانشمندان اولیه علم کامپیوتر با ریاضیات شروع کردند). بنابراین از آن به عنوان فرضیه تورینگ یاد می‌شود. اگر در جریان فرضیه محاسبات هستید، شانس دیدن اثبات آن را خواهید داشت.

برای اینکه به شما ایده‌ای بدهیم که با ابزارهایی که تاکنون آموخته‌اید چه کارهایی می‌توانید انجام دهید، تعدادی از توابع ریاضی تعریف شده به حالت بازگشتی را ارزیابی خواهیم کرد. تعریف بازگشتی، شبیه یک تعریف حلقوی است. به این مفهوم که تعریف، شامل ارجاعی است به چیزهایی که تعریف شده‌اند. یک تعریف حلقوی واقعی چندان مفید نیست:

frabjuous: صفتی است برای شرح هر چیزی که **frabjuous** باشد!

اگر تعریف فوق را در لغت‌نامه جستجو کنید ممکن است آزرده شوید. از طرف دیگر اگر به تعریف فاکتوریل ریاضی نگاه کنید، چیزی شبیه به عبارت زیر را خواهید دید:

$$\begin{aligned} 0! &= 1 \\ n! &= n * (n-1)! \end{aligned}$$

این تعریف می‌گوید فاکتوریل عدد **0**، **1** است و فاکتوریل هر مقدار **n** برابر است با ضرب **n** در فاکتوریل **n-1**. بنابراین **3!** برابر است با **3!** تا **2!** که خود برابر **2!** تا **1!** و آن هم برابر **1!** در **0!** است. لذا نتیجه برابر **6** می‌باشد.

اگر بتوانید تابع بازگشتی چیزی را بنویسید، معمولاً می‌توانید یک برنامه پایتون هم برای ارزیابی آن بنویسید. اولین گام، مشخص کردن پارامترهای تابع است. با تلاشی جزئی متوجه خواهید شد که **factorial** یک پارامتر می‌گیرد:

```
def factorial(n):
```

اگر مقدار آرگومان تابع **0** باشد، کافی است **1** را بازگردانیم:

```
def factorial(n):
    if n == 0:
        return 1
```

در غیر این صورت (این قسمت جالب برنامه است) ما باید یک فراخوانی بازگشتی برای پیدا کردن فاکتوریل $n-1$ بسازیم و سپس آن را در n ضرب کنیم:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        recurse = factorial(n-1)
        result = n * recurse
        return result
```

روند اجرای این برنامه شبیه به روند تابع **countdown** در بخش ۴-۹ است. اگر ما تابع فاکتوریل را با مقدار ۳ صدا بزنیم:

از آنجا که ۳ مخالف ۰ است، شاخه دوم را دنبال می‌کنیم که محاسبه مقدار فاکتوریل $n-1$ است. از آنجا که ۲ مخالف ۰ است، شاخه دوم را دنبال می‌کنیم که محاسبه مقدار فاکتوریل $n-1$ است.

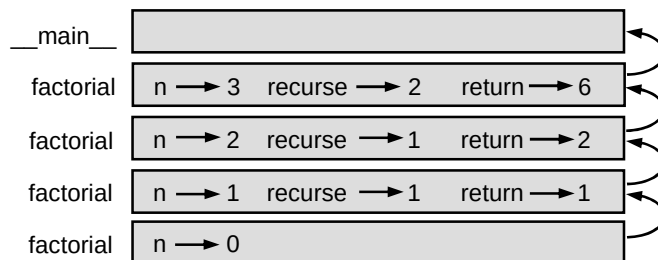
از آنجا که ۱ مخالف ۰ است، شاخه دوم را دنبال می‌کنیم که محاسبه مقدار فاکتوریل $n-1$ است.

از آنجا که ۰ با ۰ مساوی است، شاخه اول را دنبال می‌کنیم که برگرداندن مقدار ۱ بدون هیچ فراخوانی بازگشتی دیگر است.

مقدار برگشتی (۱) در n ، که برابر با ۱ می‌باشد، ضرب شده و نتیجه بازگردانده می‌شود.

مقدار برگشتی (۱) در n ، که برابر با ۲ می‌باشد، ضرب شده و نتیجه بازگردانده می‌شود. مقدار برگشتی (۲) در n ، که برابر با ۳ می‌باشد، ضرب شده و ۶ را نتیجه می‌دهد. مقدار بازگشتی به اولین تابعی که فراخوانی شده باز می‌گردد.

در شکل ۵-۱ آنچه را که نمودار پشته برای دنباله‌ای از فراخوانی‌های تابع نشان می‌دهد، ملاحظه می‌کنیم:



شکل ۵-۱

تحویل مقادیر بازگشتی به پشتۀ بالایی در نمودار توسط پیکان‌هایی نمایش داده شده‌اند. در هر قاب مقدار بازگشتی **result** است که به وسیلۀ **n** و **recurse** تولید شده است. توجه کنید که در قاب آخر متغیرهای محلی **recurse** و **result** وجود ندارند، زیرا شاخۀ سازندۀ آنها اجرا نمی‌شود.

۵-۶- جهش با اطمینان

دنبال کردن روند اجرا، یک راه برای خواندن برنامه‌ها است اما می‌تواند به سرعت پر پیچ و خم شود. روشی وجود دارد که ما آن را «جهش با اطمینان» می‌نامیم. وقتی شما به فراخوانی یک تابع رسیدید، فرض کنید تابع درست کار می‌کند و مقادیر مناسب را هم باز می‌گرداند. بنابراین به‌جای دنبال کردن روند اجرای تابع، از آن عبور کنید.

در حقیقت شما قبلاً این روش را زمانی که با توابع پیش‌ساختۀ پایتون کار می‌کردید، تمرین نموده‌اید. وقتی که شما **math.cos** یا **math.exp** را فرا می‌خواندید، عملکرد آن توابع را امتحان نمی‌کردید. شما فقط فرض می‌کردید که آنها کار می‌کنند، زیرا کسانی که کتابخانۀ پیش‌ساختۀ پایتون را نوشته‌اند، برنامه‌نویسان خوبی بوده‌اند.

همین‌طور است برای وقتی که شما یکی از توابع خودتان را فرا می‌خوانید. برای نمونه در بخش ۴-۵ ما تابعی به نام **isDivisible** نوشتیم که بخش‌پذیری یک عدد بر عدد دیگر را مشخص می‌کرد. یک بار که ما مطمئن شدیم این تابع درست است (با آزمایش و اشکال‌زدایی کد)، می‌توانیم آن را بدون نگاه کردن به کد آن باز هم استفاده کنیم.

برای توابع بازگشتی نیز به همین صورت است. وقتی که به فراخوانی بازگشتی رسیدید، به جای دنبال کردن روند اجرا فرض کنید که فراخوانی بازگشتی کار می‌کند (نتیجۀ صحیح می‌دهد) و سپس از خود بپرسید: «به فرض اینکه من فاکتوریل **n-1** را به‌دست آورم، آیا می‌توانم فاکتوریل **n** را به‌دست آورم؟» در این مورد واضح است که با ضرب کردن نتیجه در عدد **n**، می‌توانید این کار را انجام دهید. البته این روش که بدون به پایان رساندن یک تابع آن را صحیح فرض کنیم کمی غیرعادی است، اما دلیل اینکه ما آن را «جهش با اطمینان» نامیده‌ایم نیز همین است!

۵-۷- مثالی دیگر

در مثال قبل ما از متغیرهای موقتی برای توضیح دادن گام‌های برنامه و آسان ساختن کد، جهت خطایابی استفاده کردیم، اما اکنون می‌توانیم خطوط برنامه را کمتر کنیم:

```
def factorial(n):  
    if n == 0:  
        return 1  
    else:  
        return n * factorial(n-1)
```

از این پس آمادگی بیشتری برای فشرده‌سازی کد برنامه داریم، اما سفارش می‌کنیم که شما تا وقتی کد را توسعه می‌دهید از نسخهٔ صریح برنامه استفاده کنید. هرگاه دیدید برنامه‌تان کار می‌کند اما احساس کردید حجیم است، می‌توانید آن را فشرده‌تر سازید.
بعد از فاکتوریل معمول‌ترین مثال بازگشتی، تابع ریاضی **fibonacci** است که تعریف زیر را دارا است:

```
fibonacci(0) = 1  
fibonacci(1) = 1  
fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)
```

با ترجمهٔ آن به پایتون این تابع به‌صورت زیر مبدل می‌شود:

```
def fibonacci (n):  
    if n == 0 or n == 1:  
        return 1  
    else:  
        return fibonacci(n-1) + fibonacci(n-2)
```

در اینجا اگر بخواهید سعی کنید روند اجرا را دنبال کنید، حتی برای مقادیر کوچک n ممکن است کاملاً گیج شوید، اما بر اساس روش جهش با اطمینان اگر فرض کنید که هر دو فراخوانی بازگشتی درست کار می‌کنند، آنگاه واضح است که با جمع کردن آن دو با هم جواب صحیح را به‌دست خواهید آورد.

۵-۸- بررسی انواع داده‌ها

اگر ما تابع **factoial** را با مقدار **1.5** بررسی کنیم، چه اتفاقی می‌افتد؟

```
>>> factorial (1.5)  
RuntimeError: Maximum recursion depth exceeded
```

این شبیه یک بازگشت بی‌انتهای به‌نظر می‌رسد اما چگونه می‌تواند چنین باشد؟ یک حالت مبنا در وضعیت $n==0$ وجود دارد. مشکل اینجا است که مقادیر n حالت مبنا را گم می‌کنند. در اولین فراخوانی بازگشتی مقدار n برابر با 0.5 است. در فراخوانی بعدی مقدار n برابر با -0.5 است و از این لحظه مقدار کوچک و کوچک‌تر می‌شود اما هیچ‌گاه به صفر نمی‌رسد.

ما دو انتخاب داریم؛ می‌توانیم تابع **factorial** را برای کار بر روی اعداد اعشاری تعمیم دهیم و یا **factorial** را به گونه‌ای بسازیم که نوع آرگومان‌های خود را بررسی کند. اولین راه را «تابع گاما» می‌نامند و کمی از بحث این کتاب خارج است. بنابراین راه دوم را در پیش می‌گیریم. ما می‌توانیم از تابع **type** برای مقایسه نوع پارامتر با نوع مقدار صحیح مشخصی (مانند 1) استفاده کنیم. حال که ما بر روی این موضوع کار می‌کنیم، می‌توانیم مثبت بودن پارامتر را هم بررسی کنیم:

```
def factorial (n):
    if type(n) != type(1):
        print "Factorial is only defined for integers."
        return -1
    elif n < 0:
        print "Factorial is only defined for positive integers."
        return -1
    elif n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

اکنون ما سه حالت مبنا داریم. اولین حالت مبنا اعداد غیر صحیح را جدا می‌کند (یا به زبان عامیانه «غریبال می‌کند») و دومی اعداد صحیح منفی را. در هر دو حالت، برنامه پیغام خطایی را چاپ می‌کند و برای اینکه غلط بودن چیزی را نشان بدهد، مقدار مشخص -1 را بر می‌گرداند:

```
>>> factorial ("fred")
Factorial is only defined for integers.
-1
>>> factorial (-2)
Factorial is only defined for positive integers.
-1
```

اگر ما هر دو بررسی را انجام دهیم، آنگاه می‌دانیم که n یک عدد صحیح مثبت است و می‌توانیم ثابت کنیم که بازگشت پایان می‌یابد.

این برنامه الگویی را نشان می‌دهد که گاهی **نگهبان** نامیده می‌شود. دو شرط اول به‌عنوان نگهبان عمل می‌کنند و از کد برنامه وقتی که با مقادیر اشکال‌آفرین کار می‌کند محافظت می‌نمایند. نگهبان‌ها اثبات درستی کد را ممکن می‌سازند.

۵-۹- واژه‌نامه

fruitful function (تابع نتیجه‌دار)

تابعی که مقداری را برمی‌گرداند.

return value (مقدار برگشتی)

مقداری که از حاصل فراخوانی یک تابع تولید شده است.

temporary variable (متغیر موقت)

متغیر استفاده شده برای ذخیره‌سازی یک مقدار واسطه در یک محاسبه پیچیده.

dead code (کد مرده)

قسمتی از برنامه که هیچ‌گاه اجرا نمی‌شود و اغلب به خاطر این است که بعد از دستور **return** قرار گرفته است.

None

در توابعی که دستور **return** وجود ندارد و یا دستور **return** موجود دارای آرگومان نیست، پایتون مقدار ویژه‌ای به‌نام **None** را برمی‌گرداند.

incremental development (توسعه افزایشی)

یک طرح توسعه برنامه به‌منظور خودداری از اشکال‌زدایی، به این صورت که قطعه کوچکی از کد برنامه در هر زمان اضافه و آزمایش شود.

scaffolding (داربست)

کدی که در طول توسعه برنامه استفاده شده، اما جزء نسخه پایانی برنامه نیست.

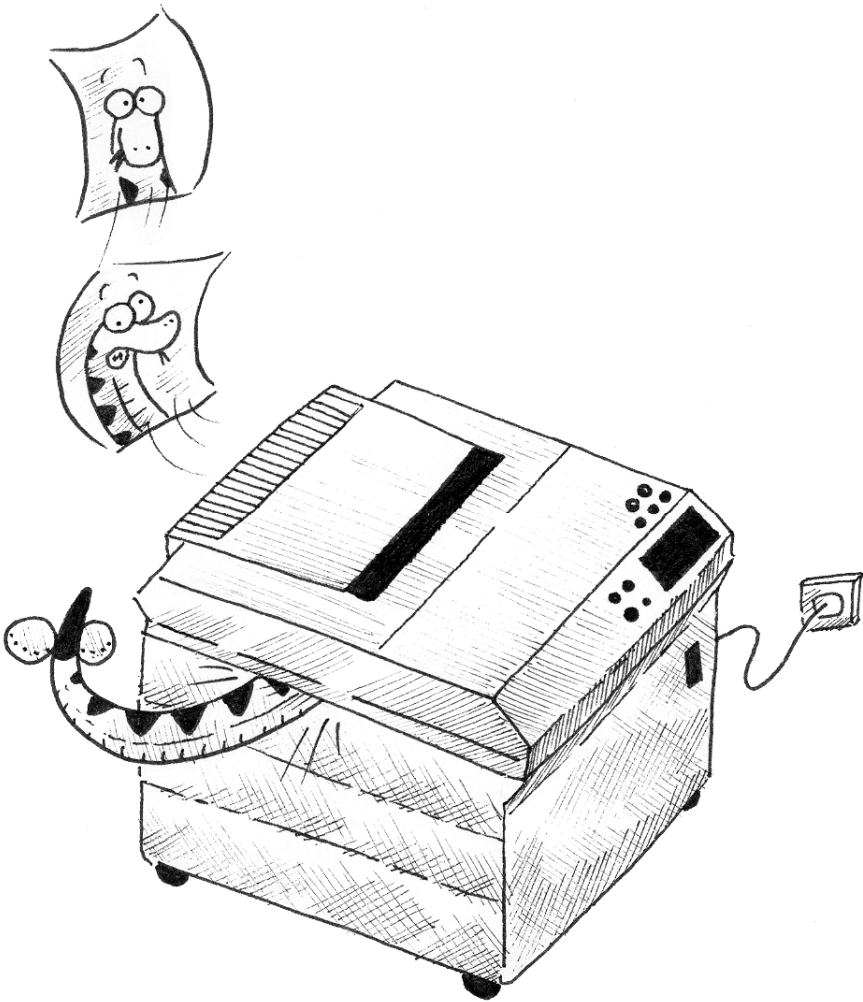
composition (ترکیب)

فراخوانی یک تابع از درون تابعی دیگر.

guardian (نگهبان)

شرطی که پیش‌آمدهای اشکال‌آفرین برنامه را بررسی می‌کند و مدیریت می‌کند.

تکرار



کامپیوترها اغلب جهت خودکارسازی وظایف تکراری استفاده می‌شوند. تکرار اعمال دقیقاً یکسان یا مشابه بدون تولید خطا، کاری است که کامپیوترها به راحتی انجام می‌دهند اما انسان‌ها از انجام آن عاجزند. باید توجه داشته باشید که خطاهای موجود در نتیجه برنامه‌ها به هیچ عنوان به کامپیوتر مربوط نمی‌شود، بلکه این برنامه‌نویس است که در صورت اشتباه در دستوردهی به کامپیوتر باعث بروز خطاها می‌شود.

در این فصل، شما با چگونگی انجام اعمال تکراری در کامپیوتر آشنا می‌شوید و از آنجا که متغیرها ارتباط زیادی با این مقوله دارند، می‌آموزید که مقدار یک متغیر پس از تشکیل چگونه تغییر می‌یابد و حوزه دسترسی به آنها در چه محدوده‌ای است.

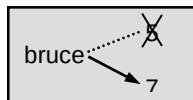
۶-۱- انتساب چندگانه

همان‌طور که دریافته‌اید، در پایتون بیش از یک انتساب به یک متغیر، عملی قانونی است. انتساب جدید، متغیر موجود را به مقدار جدید ارجاع می‌دهد (و ارجاع به متغیر قدیمی را متوقف می‌کند).

```
bruce = 5
print bruce,
bruce = 7
print bruce
```

خروجی این برنامه 5 7 است، زیرا اولین بار که **bruce** چاپ می‌شود مقدار آن 5 است و دفعه دوم 7. علامت کامایی که در آخر اولین دستور **print** آمده است، مانع رفتن به خط جدید پس از چاپ خروجی می‌شود و علت اینکه خروجی‌ها هر دو در یک سطر نمایش داده شده‌اند نیز همین است.

در اینجا انتساب چندگانه در یک نمودار حالت نشان داده شده است:



شکل ۶-۱

در میحث انتساب چندگانه این مطلب بسیار مهم است که یک عملیات انتساب و یک دستور تساوی را از هم تمیز دهیم، زیرا پایتون از علامت مساوی (=) برای انتساب استفاده می‌کند و بسیار فریبنده است که دستوری شبیه به $a = b$ به عنوان یک دستور تساوی مطرح شود که این‌طور نیست.

اول این که تساوی جابجایی پذیر است ولی انتساب نه. برای مثال در ریاضیات اگر $aa=a7$ آنگاه $7=a$ است، اما در پایتون دستور $a=7$ قانونی و $7=a$ غیرمجاز است. همچنین در ریاضی دستور تساوی همیشه درست است. اگر در حال حاضر $aa=ab$ باشد، همیشه a با b برابر خواهد بود. در پایتون یک دستور انتساب می تواند دو متغیر را با هم برابر سازد اما آنها مجبور نیستند همیشه به همین حالت بمانند:

```
a = 5
b = a # a and b are now equal
a = 3 # a and b are no longer equal
```

خط سوم مقدار a را تغییر می دهد اما تغییری در مقدار b به وجود نمی آورد، بنابراین آنها دیگر با هم برابر نیستند. (در بعضی از زبان های برنامه نویسی، برای جلوگیری از اشتباه الگوی متفاوتی جهت انتساب استفاده می شود از قبیل $=$ یا $<:=$.) اگرچه انتساب چندگانه اغلب مفید است، اما باید با احتیاط از آن استفاده کنیم. اگر مقادیر متغیرها مکرراً تغییر یابند، کد برنامه را از لحاظ خوانایی و رفع اشکال دشوار می سازند.

۶-۲- دستور while

تا به حال دو برنامه `nLines` و `countdown` را که از بازگشت برای انجام عمل تکرار استفاده می کنند، دیده ایم. این گونه برنامه ها نیز **تکرار** نامیده می شوند. از آنجا که عمل تکرار بسیار رایج است، پایتون چندین خصیصه زبان برای ساده تر کردن آن فراهم می کند. اولین خصیصه ای که قصد داریم بررسی کنیم، دستور `while` است. در اینجا چیزی شبیه به برنامه `countdown` را با یک دستور `while` می بینید:

```
def countdown(n):
    while n > 0:
        print n
        n = n-1
    print "Blastoff!"
```

از آنجا که ما فراخوانی بازگشتی را حذف کرده ایم، این تابع بازگشتی نیست. شما می توانید دستور `while` را همان طور که در زبان انگلیسی بیان می شود، استفاده کنید. این کد یعنی «تا وقتی که $n > 0$ است، مقدار n را چاپ کرده و آنگاه 1 را از آن کم کن. وقتی n به 0 رسید، کلمه **Blastoff!** را نمایش بده.»

در اینجا روند اجرای یک دستور **while** را به صورتی دیگر می‌بینید:

- ارزیابی شرط و به دست آوردن 0 یا 1
- ۷. اگر شرط **false** (0) بود، از دستور **while** خارج شو و دستور بعدی را اجرا کن.
- ۸. اگر شرط **true** (1) بود، تمام دستورات موجود در بدنه را اجرا کن و به مرحله ۱ برگرد.

منظور از بدنه تمام دستوراتی است که در زیر عنوان قرار گرفته‌اند و دارای تورفتگی یکسان هستند.

این نوع روند را یک **حلقه** می‌نامند، زیرا مرحله سوم به صورت یک حلقه به مرحله اول پیوند داده شده است. توجه کنید که اگر شرط در اولین بررسی حلقه **false** باشد، دستورات درون حلقه هرگز اجرا نمی‌شوند.

بدنه حلقه باید یک یا چند مقدار را تغییر دهد، به طوری که نهایتاً شرط **false** شود و حلقه پایان یابد. در غیر این صورت حلقه برای همیشه تکرار می‌شود، که به این حالت **حلقه بی‌انتهای** گفته می‌شود.

در مورد **countdown** می‌توانیم ثابت کنیم که حلقه پایان می‌یابد، زیرا می‌دانیم که مقدار **n** متناهی است و می‌بینیم که در طول حلقه هر بار کوچک‌تر می‌شود بنابراین سرانجام به مقدار 0 می‌رسیم. در برخی موارد به راحتی نمی‌توان متناهی بودن حلقه را اثبات کرد:

```
def sequence(n):
    while n != 1:
        print n,
        if n%2 == 0:           # n is even
            n = n/2
        else:                  # n is odd
            n = n*3+1
```

شرط این حلقه **n != 1** است، بنابراین حلقه تا آنجا که **n** برابر با 1 شود، ادامه می‌یابد چرا که این حالت شرط حلقه را **false** می‌کند.

در هر بار اجرای حلقه، برنامه مقدار **n** را به عنوان خروجی چاپ می‌کند و سپس زوج یا فرد بودن آن را بررسی می‌نماید. اگر زوج باشد، مقدار **n** بر 2 تقسیم می‌شود و اگر فرد باشد مقدار با عبارت **n*3+1** جایگزین می‌گردد. برای مثال اگر مقدار آغازین (آرگومانی که به تابع **sequence** فرستاده می‌شود) 3 باشد، دنباله حاصل به این صورت خواهد بود:

3 10 5 16 8 4 2 1

از آنجا که n گاهی کاهش و گاهی افزایش می‌یابد، هیچ دلیل آشکاری برای رسیدن n به مقدار 1 یا پایان برنامه، وجود ندارد. برای برخی مقادیر ویژه n ، می‌توانیم پایان یافتن برنامه را تضمین کنیم. برای مثال اگر مقدار آغازین توانی از 2 باشد، آنگاه مقدار n تا وقتی که به 1 برسد، در هر بار اجرای حلقه زوج خواهد بود. در مثال قبل هنگامی که در دنباله به عدد 16 رسیدیم، می‌توانیم مطمئن باشیم که این دنباله پایان می‌پذیرد (یعنی شرط حلقه $n == 1$ می‌شود).

صرف‌نظر از مقادیر ویژه، سؤال جالب آن است که آیا می‌توانیم ثابت کنیم که این برنامه برای تمام مقادیر n پایان می‌یابد؟ تاکنون کسی نتوانسته این موضوع را ثابت یا رد کند.

تمرین ۶-۱: تابع `nLines` از بخش ۴-۹ را به جای بازگشت با استفاده از تکرار بازنویسی کنید.

۶-۳- جدول‌ها

یکی از کاربردهای حلقه‌ها در جدول‌بندی داده‌ها است. پیش از آنکه کامپیوترها به خوبی در دسترس قرار گیرند، مردم مجبور بودند که لگاریتم‌ها، سینوس‌ها، کسینوس‌ها و دیگر توابع ریاضی را به صورت دستی محاسبه کنند. برای آسان‌تر کردن این کار کتاب‌های ریاضی شامل جداولی طولانی بودند که مقادیر این گونه توابع را فهرست کرده بودند. ساختن جدول‌ها به کندی انجام می‌گرفت و خسته‌کننده و مملو از خطا بود.

وقتی کامپیوترها وارد صحنه شدند، یکی از واکنش‌های نخست این بود: «این عالی است! ما می‌توانیم از کامپیوترها برای تولید جدول‌ها استفاده کنیم. همچنین آنها بدون خطا خواهند بود.» کمی پس از آن، کامپیوترها و ماشین‌حساب‌ها بسیار فراگیر شدند، به طوری که جدول‌ها از رده خارج شدند. در برخی اعمال، کامپیوترها از جداول مقادیر برای بدست آوردن یک جواب تقریبی استفاده می‌کنند و سپس از محاسباتی برای بهبود تقریب استفاده می‌نمایند. در بعضی موارد خطاهایی در جدول‌های مقدماتی وجود داشته است. مشهورترین آنها جدول «پنتیوم اینتل» بوده است که برای انجام تقسیم اعشاری استفاده شده است.

اگرچه یک جدول لگاریتمی آن‌چنان‌که در گذشته بود مفید نیست، اما هنوز مثال خوبی در مبحث تکرار است. خروجی برنامه زیر دنباله‌ای از مقادیر در ستونی در سمت چپ و لگاریتم آن اعداد در ستونی دیگر در سمت راست می‌باشد:

```
x = 1.0
while x < 10.0:
    print x, '\t', math.log(x)
    x = x + 1.0
```

رشته `'\t'` یک کاراکتر `tab` را نشان می‌دهد.

در حالی که کاراکترها و رشته‌ها روی صفحه نمایش نشان داده می‌شوند، یک نشانگر نامرئی به نام مکان‌نما `print` مکان نما به طور معمول به ابتدای سطر بعد می‌رود. کاراکتر `tab` مکان‌نما را تا وقتی به یک `tab stop` برسد، به سمت راست حرکت می‌دهد. `tab stop` مکانی است که مکان‌نما در آنجا توقف می‌کند. کاراکتر `tab` برای ساختن ستون‌هایی از خطوط متنی بسیار مفید است. همان‌طور که در خروجی برنامه قبل مشاهده می‌شود:

```
1.0      0.0
2.0      0.69314718056
3.0      1.09861228867
4.0      1.38629436112
5.0      1.60943791243
6.0      1.79175946923
7.0      1.94591014906
8.0      2.07944154168
9.0      2.19722457734
```

اگر این مقادیر، عجیب به نظر می‌رسند، به خاطر داشته باشید که تابع `log` از مبنای `e` استفاده می‌کند. از آنجا که توان‌های 2 در علم کامپیوتر بسیار مهمند، ما اغلب می‌خواهیم لگاریتم‌ها را در مبنای 2 به دست آوریم. برای این کار می‌توانیم از فرمول زیر استفاده کنیم:

با تغییر دستور خروجی به صورت:

```
print x, '\t', math.log(x)/math.log(2.0)
```

نتیجه زیر حاصل می‌شود:

```
1.0      0.0
2.0      1.0
3.0      1.58496250072
4.0      2.0
5.0      2.32192809489
6.0      2.58496250072
7.0      2.80735492206
8.0      3.0
9.0      3.16992500144
```


می‌توانیم ببینیم که 1، 2، 4 و 8 توان‌های 2 هستند، زیرا لگاریتم این اعداد در مبنای 2 اعداد کامل و گرد شده‌ای می‌باشند.

اگر می‌خواستیم لگاریتم دیگر توان‌های 2 را پیدا کنیم، می‌توانستیم برنامه را به این شکل بنویسیم:

```
x = 1.0
while x < 100.0:
    print x, '\t', math.log(x)/math.log(2.0)
    x = x * 2.0
```

حال به جای اینکه در میان حلقه، هر بار مقداری را با x جمع کنیم و یک دنباله حسابی را نتیجه بگیریم، x را در مقداری ضرب می‌کنیم و یک دنباله هندسی را به‌دست می‌آوریم. نتیجه بدین صورت است:

1.0	0.0
2.0	1.0
4.0	2.0
8.0	3.0
16.0	4.0
32.0	5.0
64.0	6.0

به‌واسطه وجود کاراکترهای **tab** میان ستون‌ها، مکان دوم ستون‌ها به تعداد ارقام اولین ستون ارتباطی ندارد. جداول لگاریتمی ممکن است استفاده چندانی نداشته باشند ولی برای متخصصین کامپیوتر دانستن توان‌های 2 ضروری است.

تمرین ۶-۲: این برنامه را طوری اصلاح کنید که خروجی، توان‌های 2 تا 65536 (2^{16}) باشد. آن را چاپ کنید و به خاطر بسپارید.

کاراکتر ممیز وارون (\) در '**\t**' ابتدای یک **کاراکتر کنترلی** را نمایش می‌دهد. کاراکترهای کنترلی برای نشان دادن کاراکترهای نامرئی مثل **tab** و کاراکتر **خط جدید** به‌کار می‌رود (کاراکتر کنترلی '**\n**' برای نمایش خط جدید استفاده می‌شود). یک کاراکتر کنترلی می‌تواند در هر جای یک رشته نمایش داده شود. در مثال فوق کاراکتر کنترلی **tab** تنها کاراکتر نامرئی درون رشته می‌باشد. فکر می‌کنید چگونه می‌توان یک کاراکتر '****' را در یک رشته نمایش داد؟

تمرین ۶-۳: یک رشته واحد بنویسید که خروجی زیر را تولید کند:

```
Python
    is a
programming language.
```

۶-۴- جداول دو بعدی

یک جدول دو بعدی، جدولی است که شما مقادیر را از تقاطع سطر و ستون‌های آن می‌خوانید. را یک جدول ضرب مثال خوبی است. فرض کنید می‌خواهید جدول ضربی برای مقادیر 1 تا 6 چاپ کنید.

راه خوبی برای شروع، نوشتن حلقه‌ای است که مضرب‌های 2 را برای این اعداد، در یک خط چاپ کند:

```
i = 1
while i <= 6:
    print 2*i, ' ',
    i = i + 1
print
```

اولین خط با متغیری به نام **i** آغاز می‌شود که به عنوان شمارنده و یا متغیر حلقه عمل می‌کند. همچنان که حلقه اجرا می‌شود، **print** میان حلقه مقدار $2*i$ را نمایش می‌دهد و سه فضای خالی پس از آن قرار می‌دهد. باز هم کامای دستور **print** از رفتن مکان‌نما به خط جدید جلوگیری می‌کند. بعد از اینکه حلقه کامل شد، دومین دستور **print** یک خط جدید را آغاز می‌کند. خروجی برنامه به صورت زیر است:

```
2      4      6      8      10     12
```

تا اینجا کار به خوبی انجام گرفته است. گام بعد بسته‌بندی و تعمیم است.

۶-۵- بسته‌بندی و تعمیم

بسته‌بندی، فرایند پنهان‌سازی قطعه‌ای از کد برنامه درون یک تابع است که به شما اجازه می‌دهد از تمام مزایای توابع بهره بگیرید. شما تا به حال دو مثال درباره بسته‌بندی دیده‌اید: **printParity** در بخش ۴-۵ و **isDivisible** در بخش ۵-۴.

تعمیم، یعنی مشخص کردن یک ویژگی مانند چاپ مضرب‌های ۲ و کلی ساختن آن مانند چاپ مضرب هر عدد صحیح.

این تابع حلقه قبل را بسته‌بندی می‌کند و برای چاپ مضرب n تعمیم می‌دهد:

```
def printMultiples(n):
    i = 1
    while i <= 6:
        print n*i, '\t',
        i = i + 1
    print
```

برای بسته‌بندی، تمام کاری که باید انجام می‌دادیم اضافه کردن خط اول بود که نامی را برای تابع و لیست پارامترها اعلان می‌کرد. برای تعمیم، همه کاری که مجبور بودیم انجام دهیم، جایگزین کردن مقدار 2 با پارامتر n بود.

اگر ما این تابع را با آرگومان 2 صدا بزنیم، همان خروجی قبل را نتیجه می‌گیریم و اگر آن را با آرگومان 3 صدا بزنیم خروجی به‌صورت زیر خواهد بود:

3	6	9	12	15	18
---	---	---	----	----	----

با آرگومان 4 خروجی به‌صورت زیر است:

4	8	12	16	20	24
---	---	----	----	----	----

در این زمان می‌توانید حدس بزنید که چگونه یک جدول را چاپ کنید - به وسیله فراخوانی مکرر تابع `printMultiples` با آرگومان‌های متفاوت - در حقیقت می‌توانیم از حلقه دیگری استفاده کنیم:

```
i = 1
while i <= 6:
    printMultiples(i)
    i = i + 1
```

به شباهت این حلقه با حلقه داخلی تابع `printMultiples` دقت کنید. تمام کاری را که انجام دادیم جایگزین کردن دستور `print` با یک فراخوانی تابع بود. خروجی این برنامه یک جدول ضرب است:

1	2	3	4	5	6
2	4	6	8	10	12
3	6	9	12	15	18
4	8	12	16	20	24
5	10	15	20	25	30
6	12	18	24	30	36

۶-۶- یک بسته‌بندی دیگر

برای اینکه یک بسته‌بندی را دوباره نشان دهیم بیاید کد آخر بخش ۵-۶ را بگیریم و در یک تابع بپوشانیم:

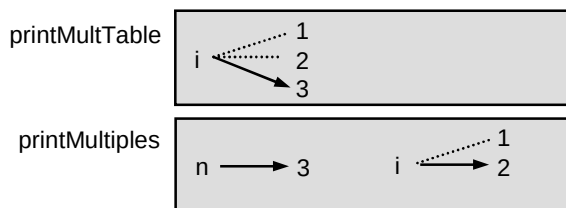
```
def printMultTable():
    i = 1
    while i <= 6:
        printMultiples(i)
        i = i + 1
```

این فرایند، یک **طرح توسعه** رایج است. ما کدی را به‌وسیله نوشتن خطوط کوتاه کد بیرون تابع یا تایپ کردن آنها در مفسر ایجاد می‌کنیم. وقتی که دیدیم برنامه کار می‌کند، آن را خارج می‌کنیم و در یک تابع بسته‌بندی می‌نماییم. وقتی شروع به برنامه‌نویسی می‌کنید، اگر نمی‌دانید چگونه برنامه را در توابع تقسیم کنید، این طرح توسعه به‌طور ویژه‌ای مفید است. این خط مشی به شما اجازه می‌دهد همان‌طور که پیش می‌روید، این عمل را طراحی کنید.

۶-۷- متغیرهای محلی

شاید تعجب کرده باشید که چگونه می‌توانیم متغیر مشابهی همچون **i** را در هر دو تابع **printMultTable** و **printMultiples** استفاده کنیم. آیا هنگامی که یکی از توابع، مقدار متغیر را تغییر می‌دهد، مشکلی ایجاد نمی‌شود؟

پاسخ منفی است، زیرا **I** در **printMultiples** و **I** در **printMultTable** یکسان نیستند. متغیرهای ایجاد شده درون یک تابع، محلی هستند و شما نمی‌توانید به متغیر محلی از بیرون محیط تابع دسترسی داشته باشید. یعنی مختارید که چندین متغیر با نام‌های مشابه داشته باشید به شرطی که آنها درون یک تابع نباشند. نمودار پشته برای این برنامه نشان می‌دهد که دو متغیر با نام **i**، متغیرهای یکسانی نیستند. آنها می‌توانند به مقادیر متفاوتی اشاره کنند و تغییر یکی از آنها در دیگری تأثیری ندارد.



شکل ۶-۲

مقدار **i** در **printMultTable** از 1 به سمت 6 می‌رود. در نمودار این اتفاق تا زمان رسیدن به 3 رخ داده است. در اجرای بعدی حلقه مقدار آن 4 خواهد بود. هر بار در طول حلقه **printMultTable** تابع **printMultiples** را با مقدار جاری **i**، به‌عنوان یک آرگومان فراخوانی می‌کند. آن مقدار به پارامتر **n** انتساب می‌یابد.

درون تابع **printMultiples** مقدار **i** از 1 به طرف 6 می‌رود. در نمودار این اتفاق تا رسیدن به عدد 2 رخ داده است. تغییر این متغیر هیچ تأثیری روی مقدار **i** در تابع **printMultTable** ندارد.

داشتن متغیرهای محلی مختلف با نام‌های یکسان امری کاملاً قانونی و رایج است. مخصوصاً نام‌هایی مانند **i** و **j** که برای تکرار به عنوان متغیرهای حلقه استفاده می‌شوند. اگر شما از به‌کار بردن آنها در یک تابع بپرهیزید فقط به خاطر اینکه در جای دیگری استفاده کرده‌اید، ممکن است کد برنامه را از لحاظ خواندن دشوار سازید.

۶-۸- یک تعمیم دیگر

به عنوان مثال دیگری برای تعمیم فرض کنید برنامه‌ای می‌خواهید که جدول ضربی را در هر اندازه دلخواه چاپ کند، نه فقط یک جدول شش در شش را. شما می‌توانستید پارامتری به **printMultTable** اضافه کنید:

```
def printMultTable(high):
    i = 1
    while i <= high:
        printMultiples(i)
        i = i + 1
```

ما مقدار 6 را با متغیر **high** جایگزین نمودیم. حال اگر **printMultTable** را با مقدار 7 فراخوانی کنیم، جدول زیر را خواهیم داشت:

1	2	3	4	5	6
2	4	6	8	10	12
3	6	9	12	15	18
4	8	12	16	20	24
5	10	15	20	25	30
6	12	18	24	30	36
7	14	21	28	35	42

این خوب است، مگر اینکه بخواهیم جدولی به صورت مربع -با تعداد سطر و ستون مشابه- داشته باشیم. برای این کار ما پارامتر دیگری را به `printMultiples` اضافه می‌نماییم تا مشخص کنیم در جدول چند ستون باید داشته باشیم.

فقط برای شیطنیت، ما این تابع را `high` می‌نامیم تا ثابت کنیم توابع مختلف می‌توانند پارامترهایی با نام یکسان داشته باشند (درست مانند متغیرهای محلی). در اینجا کد کامل برنامه را می‌بینیم:

```
def printMultiples(n, high):
    i = 1
    while i <= high:
        print n*i, '\t',
        i = i + 1
    print

def printMultTable(high):
    i = 1
    while i <= high:
        printMultiples(i, high)
        i = i + 1
```

توجه کنید که وقتی ما یک پارامتر جدید را اضافه می‌کنیم باید اولین خط تابع (عنوان تابع) را تغییر دهیم و همچنین باید کد برنامه را در مکانی که تابع فراخوانی می‌شود (در داخل `printMultTable`) تغییر دهیم.

همان‌طور که انتظار داشتیم این برنامه یک جدول مربع‌شکل هفت در هفت تولید می‌کند:

1	2	3	4	5	6	7
2	4	6	8	10	12	14
3	6	9	12	15	18	21
4	8	12	16	20	24	28
5	10	15	20	25	30	35
6	12	18	24	30	36	42
7	14	21	28	35	42	49

وقتی تابعی را به‌طور مناسب تعمیم می‌دهید، برنامه‌ای که به‌دست می‌آورید، اغلب دارای توانایی‌هایی است که از قبل طراحی نکرده بودید. برای نمونه شما می‌بایست توجه کرده باشید، چون `ab=ba`، همهٔ ارقام در جدول دو بار نشان داده شده‌اند. پس می‌توانستید به‌وسیلهٔ چاپ تنها نیمی از جدول بقیهٔ جوهر خود را ذخیره کنید!

برای این کار کافی است خط اول تابع `printMultiples` را تغییر دهید.

پس با تغییر `printMultiples(i, ahigh)` به `printMultiples(i, ai)` خواهیم داشت:

1						
2	4					
3	6	9				
4	8	12	16			
5	10	15	20	25		
6	12	18	24	30	36	
7	14	21	28	35	42	49

تمرین ۶-۴: اجرای این نسخه از `printMultTable` را ردیابی کنید تا دریابید که چگونه کار می‌کند.

۶-۹- توابع

تاکنون چند بار به “محاسن استفاده از توابع” اشاره کردیم. شاید شما در مورد اینکه آنها دقیقاً چه چیزهایی هستند، فکر کرده باشید. تعدادی از آنها در اینجا آورده شده است:

- دادن نام به دنباله‌ای از دستورات، برنامه‌تان را برای خواندن و اشکال‌زدایی ساده‌تر می‌سازد.
- قسمت کردن یک برنامه طولانی در توابع متعدد به شما امکان می‌دهد قسمت‌های جداگانه برنامه را به‌طور مجزا اشکال‌زدایی کنید و سپس آنها را در یک برنامه کامل با هم ترکیب کنید.
- توابع بازگشت و تکرار را آسان می‌سازند.
- توابعی که خوب طراحی شده‌اند برای تعداد زیادی از برنامه‌ها مفیدند. یک بار که تابعی را بنویسید و اشکال‌زدایی کنید، می‌توانید باز از آن استفاده کنید.

۶-۱۰- واژه‌نامه

multiple assignment (انتساب چندگانه)

نسبت‌دهی بیش از یک بار به متغیری یکسان در طول اجرای برنامه.

iteration (تکرار)

تکرار کردن اجرای مجموعه‌ای از دستورات استفاده شده در یک تابع بازگشتی و یا یک حلقه.

loop (حلقه)

دستور یا مجموعه‌ای از دستورات که مکرراً اجرا شوند تا وقتی که به‌وسیله شرط پایان حلقه متوقف گردند.

infinite loop (حلقه بی‌انتهای)

حلقه‌ای که شرط پایانی آن هیچ‌گاه برقرار نمی‌شود.

tab

یک کاراکتر ویژه که مکان‌نما را در خط جاری به محل توقف بعدی انتقال می‌دهد.

cursor (مکان‌نما)

یک نشانگر نامرئی که ردّ جایی را که کاراکتر بعدی می‌خواهد چاپ شود، نگه می‌دارد.

escape sequence (کاراکتر کنترلی)

یک کاراکتر " / " که به‌وسیله یک یا چند کاراکتر قابل چاپ برای مشخص کردن یک کاراکتر غیرقابل چاپ دنبال شده است.

newline (خط جدید)

یک کاراکتر ویژه که مکان‌نما را به ابتدای خط بعد انتقال می‌دهد.

loop variable (متغیر حلقه)

متغیر استفاده شده در قسمت شرط پایان یک حلقه.

encapsulate (بسته‌بندی)

قسمت کردن یک برنامه بزرگ و پیچیده درون قطعاتی مانند تابع و مجزا کردن آن قطعات از یکدیگر (مثلاً به‌وسیله استفاده از متغیرهای محلی).

generalize (تعمیم)

جایگزین کردن هر چیز غیرضروری مشخص (مانند یک مقدار ثابت) با هر چیز مناسب کلی (مانند یک متغیر یا یک پارامتر). تعمیم، کد برنامه را روان، توانا برای استفاده مجدد و حتی گاهی اوقات آسان تر برای نوشتن می سازد.

development plan (طرح توسعه)

فرآیندی برای توسعه دادن یک برنامه. در این فصل طرح توسعه ما این چنین بود که به توسعه کدهایی برای چیزهای ساده و خاص و سپس بسته بندی و تعمیم پرداختیم.

رشته‌ها



در فصل‌های گذشته، با انتساب متغیرها و در نتیجه با تغییر و تحول انواع داده‌ای آشنا شدید و همچنین روشی را برای تکرار دستورات مشخص به کار بردید. انواع داده‌ای استفاده شده در فصل‌های اخیر بسیار اندک بودند. در چند فصل آینده با چند نوع داده‌ای جدید آشنا می‌شوید که کاربرد بسیاری دارند و توانمندی شما را در انجام کارهایتان افزایش می‌دهند. علاوه بر آشنایی با این انواع داده‌ای، روش جدیدی برای انجام اعمال تکراری می‌آموزید. با این روش، خوانایی برنامه و توانایی شما در کوتاه‌سازی برنامه نیز افزایش می‌یابد.

۷-۱- نوع داده‌ای مرکب

تاکنون سه نوع داده‌ای دیده‌ایم: **int**، **float** و **string** (اعداد صحیح، اعشاری و رشته‌ها). رشته‌ها از لحاظ کیفی با دو نوع دیگر متفاوتند زیرا آنها از قطعات کوچک‌تری به نام کاراکتر ساخته شده‌اند.

انواع داده‌ای که از قسمت‌های کوچک‌تر تشکیل شده‌اند، نوع داده‌ای مرکب نامیده می‌شوند. بسته به عملی که انجام می‌دهیم ممکن است نوع داده‌ای مرکب را به عنوان داده‌ای واحد تلقی کنیم یا اینکه بخواهیم به اجزای آن دسترسی داشته باشیم. این دوگانگی مفید است. عملگر براکت یک کاراکتر واحد را از یک رشته انتخاب می‌کند:

```
>>> fruit = "banana"
>>> letter = fruit[1]
>>> print letter
```

عبارت **fruit[1]** کاراکتر شماره ۱ را از **fruit** انتخاب می‌کند. متغیر **letter** به نتیجه اشاره می‌کند. وقتی **letter** را نمایش می‌دهیم غافلگیر می‌شویم:

```
a
```

حرف اول "banana"، 'a' نیست، مگر اینکه شما یک متخصص کامپیوتر باشید. به دلایلی متخصصین کامپیوتر همواره شمارش را از صفر شروع می‌کنند. صفرمین حرف "banana"، 'b' است، یکمین حرف 'a' و دومین حرف 'n'. اگر صفرمین حرف یک رشته را می‌خواهید، تنها عدد ۰ و یا هر عبارت با مقدار ۰ را در براکت قرار دهید:

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

عبارت داخل براکت اندیس نامیده می‌شود. یک اندیس عضوی از یک مجموعه مرتب را مشخص می‌کند که در این مثال مجموعه کاراکترهای درون رشته مورد نظر است. اندیس مشخص می‌کند که شما کدام کاراکتر را انتخاب کرده‌اید و می‌تواند هر عبارت صحیحی باشد.

۷-۲- طول رشته

تابع **len** تعداد کاراکترهای یک رشته را برمی‌گرداند:

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

برای گرفتن آخرین حرف یک رشته ممکن است آزمایش چنین عملی وسوسه‌انگیز باشد:

```
length = len(fruit)
last = fruit[length]           # ERROR!
```

این کد کار نمی‌کند و موجب خطای زمان اجرای زیر می‌شود:

IndexError: string index out of range

علت بروز این خطا عدم وجود حرف ششم در رشته "banana" است. از آنجا که ما شمارش را از صفر آغاز کرده‌ایم، شش حرف این رشته از 0 تا 5 شماره‌گذاری شده‌اند. به منظور گرفتن آخرین کاراکتر مجبوریم یک واحد از **length** کم کنیم:

```
length = len(fruit)
last = fruit[length-1]
```

روش دیگر این است که ما می‌توانیم از اندیس‌های منفی که عمل شمارش را از انتهای رشته انجام می‌دهند، استفاده کنیم. عبارت **fruit[-1]** آخرین حرف را باز می‌گرداند، **fruit[-2]** دومین حرف از آخر و ...

۷-۳- پیمایش و حلقه for

بسیاری از محاسبات شامل پردازش کاراکترهای یک رشته به صورت جدا جدا است. این محاسبات از ابتدای رشته آغاز می‌شوند، کاراکترها را یکی یکی و به ترتیب انتخاب می‌کنند، عملی روی

آن انجام می‌دهند و این کار را تا انتهای رشته ادامه می‌دهند. این الگوی پردازش را **پیمایش** می‌گویند. یکی از راه‌های پیاده‌سازی پیمایش استفاده از یک دستور **while** است:

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

این حلقه طول رشته را می‌پیماید و هر حرف آن را در یک خط نمایش می‌دهد. شرط حلقه به صورت **index < len(fruit)** می‌باشد، بنابراین هرگاه **index** برابر با طول رشته شد، شرط **false** می‌شود و بدنه حلقه دیگر اجرا نمی‌گردد. آخرین اندیس قابل دسترسی کاراکتری با اندیس **len(fruit)-1** است که آخرین کاراکتر درون رشته می‌باشد.

تمرین ۷-۱: تابعی بنویسید که رشته‌ای را به عنوان آرگومان بگیرد و هر حرف آن را از انتها در هر خط چاپ کند.

استفاده از یک اندیس جهت پیمایش مجموعه‌ای از مقادیر به حدی رایج است که پایتون راه دیگری را با نحوه نگارش ساده‌تر تدارک دیده است:

```
for char in fruit:
    print char
```

در هر بار اجرای حلقه، کاراکتر بعدی رشته به متغیر **char** اختصاص داده می‌شود و حلقه تا زمانی که هیچ کاراکتری باقی نمانده باشد ادامه می‌یابد.

مثال زیر نشان می‌دهد که چگونه می‌توان جهت تولید مجموعه‌های مرتب -بر اساس حروف الفبا- از یک حلقه **for** و عمل الحاق استفاده کرد. برای نمونه در کتاب «رابرت مک‌کلوسکی» با عنوان «راه را به جوجه‌اردک‌ها نشان بده»، نام جوجه‌اردک‌ها **Mack, Lack, Kack, Jack**، **Nack, Ouack, Pack** و **Quack** می‌باشد. این حلقه اسامی را به ترتیب چاپ می‌کند:

```
prefixes = "JKLMNOPQ"
suffix = "ack"

for letter in prefixes:
    print letter + suffix
```

خروجی برنامه به این صورت است:

```
Jack
Kack
Lack
Mack
Nack
Oack
Pack
Qack
```

البته این اسامی کاملاً هم درست نیستند زیرا "Ouack" و "Quack" با غلط املائی به کار رفته‌اند.

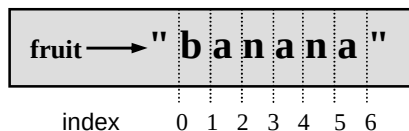
تمرین ۷-۲: برنامه را طوری تغییر دهید که این اشکال برطرف شود.

۷-۴- برش‌های رشته

قطعه‌ای از یک رشته را برش می‌نامند. انتخاب یک برش شبیه به انتخاب یک کاراکتر است:

```
>>> s = "Peter, Paul, and Mary"
>>> print s[0:5]
Peter
>>> print s[7:11]
Paul
>>> print s[17:21]
Mary
```

عملگر `[n:m]` قسمتی از یک رشته را از کاراکتر `n`ام تا کاراکتر `m`ام برمی‌گرداند که خود `n`امین کاراکتر را هم شامل می‌شود اما `m`امین کاراکتر جزء آن نیست. این رفتار غیرشهودی است. اگر تصور کنید اندیس‌ها به وسط دو رشته اشاره می‌کنند ایدهٔ بیشتری می‌گیرید. به شکل ۷-۱ توجه کنید:



شکل ۷-۱

اگر شما اندیس اول (قبل از دو نقطه) را حذف کنید برش از ابتدای رشته شروع خواهد شد و اگر از دومین اندیس صرف‌نظر کنید برش تا آخر رشته پیش خواهد رفت. بر این اساس:

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

فکر می‌کنید `s[:]` به چه معنی است؟

۷-۵- مقایسه رشته‌ها

عملگرهای مقایسه‌ای روی رشته‌ها نیز عمل می‌کنند، برای اینکه ببینیم رشته‌ها با هم برابرند یا نه:

```
if word == "banana":
    print "Yes, we have no bananas!"
```

دیگر عملگرهای مقایسه‌ای جهت مرتب کردن کلمه‌ها به ترتیب حروف الفبای انگلیسی مفیدند:

```
if word < "banana":
    print "Your word," + word + ", comes before banana."
elif word > "banana":
    print "Your word," + word + ", comes after banana."
else:
    print "Yes, we have no bananas!"
```

توجه کنید که پایتون با حروف بزرگ و کوچک آنگونه که مردم برخورد می‌کنند، رفتار نمی‌کند. تمام حروف بزرگ قبل از تمام حروف کوچک می‌آیند. به عنوان نتیجه:

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

یک راه معمول برای حل این مشکل تبدیل رشته به یک فرمت خاص است. مثلاً همه حروف را قبل از انجام مقایسه به صورت کوچک درآوریم. مشکل دشوارتر این است که به برنامه بفهمانیم گوره‌ها میوه نیستند!

۷-۶- رشته‌ها تغییرناپذیرند

استفاده از عملگر `[]` در سمت چپ یک انتساب با انگیزه تعویض یک کاراکتر در رشته وسوسه‌انگیز است. برای نمونه:


```
greeting = "Hello, world!"
greeting[0] = 'J'           # ERROR!
print greeting
```

این کد به جای تولید خروجی **jello,aworld!**، خطای زمان اجرای زیر را تولید می‌کند:

```
TypeError: object doesn't support item assignment.
```

رشته‌ها **تغییرناپذیر** هستند، به این معنی که شما نمی‌توانید یک رشته موجود را تغییر دهید. بهترین کاری که می‌توانید انجام دهید ساختن رشته جدیدی است که شامل تغییر روی رشته اصلی است:

```
greeting = "Hello, world!"
newGreeting = 'J' + greeting[1:]
print newGreeting
```

در این مثال، راه‌حل الحاق حرف جدیدی به اول یک برش از رشته **greeting** است. این عمل هیچ تأثیری بر روی رشته اصلی ندارد.

۷-۷- یک تابع **find**

تابع زیر چه می‌کند؟

```
def find(str, ch):
    index = 0
    while index < len(str):
        if str[index] == ch:
            return index
        index = index + 1
    return -1
```

به طور کلی **find** بر عکس عملگر **[]** عمل می‌کند. به جای اینکه اندیسی بگیرد و کاراکتر نظیر را بیرون بکشد، کاراکتری را می‌گیرد و اندیس اولین محل وقوع آن را پیدا می‌کند. اگر کاراکتری پیدا نشد تابع **-1** را بازمی‌گرداند.

این اولین مثالی است که ما در آن، دستور **return** را درون یک حلقه می‌بینیم. اگر شرط **str[index] == ch** برقرار شد، تابع با خروج و قطع عملیات سریعاً برمی‌گردد. اگر کاراکتری در رشته پیدا نشد برنامه به طور عادی از حلقه خارج شده و **-1** را برمی‌گرداند.

این الگوی محاسبه برخی اوقات پیمایش «eureka»^۳ نامیده می‌شود، زیرا به محض اینکه مجهول را یافتیم می‌توانیم فریاد بزنیم “هورا” و جستجو را پایان بخشیم.

تمرین ۷-۳: تابع `find` را طوری تغییر دهید که پارامتر `sومی` را به عنوان مکان جستجو بگیرد.

۷-۸- چرخش و شمارش

برنامه زیر تعداد دفعات ظاهر شدن حرف 'a' را در یک رشته می‌شمارد:

```
fruit = "banana"
count = 0
for char in fruit:
    if char == 'a':
        count = count + 1
print count
```

این برنامه الگوی دیگری از محاسبه، به نام **شمارنده** را نشان می‌دهد. شمارش متغیر از صفر شروع می‌شود و هر بار که یک 'a' پیدا می‌کند، افزایش می‌یابد. وقتی حلقه پایان می‌یابد، محتوی متغیر `count` شامل نتیجه نهایی می‌باشد که همان تعداد 'a' است.

تمرین ۷-۴: این کد را در یک تابع با نام `counterLetter` بسته‌بندی کنید و آن را تعمیم دهید، بنابراین این تابع یک رشته و یک حرف را به عنوان پارامترهای خود می‌گیرد.

تمرین ۷-۵: این تابع را به صورتی بازنویسی کنید که به جای پیمایش رشته، از نسخه سه پارامتری تابع قبلی `find` استفاده کند.

۷-۹- ماژول string

ماژول `string` شامل توابع مفیدی است که رشته‌ها را با مهارت دستکاری می‌کنند. مثل همیشه باید ماژول را قبل از استفاده وارد محیط کاری کنیم:

```
>>> import string
```

ماژول **string** شامل تابعی به نام **find** است که مانند تابع **ma** کار می‌کند. به‌منظور فراخوانی آن باید نام ماژول و سپس نام تابع را پس از گذاشتن یک نقطه مشخص کنیم:

```
>>> fruit = "banana"
>>> index = string.find(fruit, "a")
>>> print index
1
```

این مثال یکی از مزایای ماژول‌ها را ثابت می‌کند. آنها به جلوگیری از ایجاد برخورد میان نام توابع پیش‌ساخته و توابع کاربر-تعریف کمک می‌کنند. با استفاده از نمادگذاری نقطه می‌توانیم مشخص کنیم که کدام نسخه از تابع **find** موردنظر است. در حقیقت **string.find** جامع‌تر از نسخهٔ ما است. اول اینکه این تابع می‌تواند علاوه بر کاراکترها، زیر رشته‌ها را هم پیدا کند:

```
>>> string.find("banana", "na")
2
```

همچنین این تابع یک آرگومان سوم اضافه هم می‌گیرد که اندیس شروع جستجو را مشخص می‌کند:

```
>>> string.find("banana", "na", 3)
4
```

یا می‌تواند دو آرگومان اضافه بگیرد که بازه‌ای از اندیس‌ها را برای جستجو مشخص می‌کند:

```
>>> string.find("bob", "b", 1, 2)
-1
```

در این مثال جستجو منحل می‌شود زیرا حرف **b** در بازهٔ اندیس بین 1 و 2 ظاهر نمی‌شود (با توجه به اینکه خود 2 را شامل نمی‌شود).

۷-۱- طبقه‌بندی کاراکترها

آزمایش کاراکترها از لحاظ کوچک یا بزرگ بودن حروف و یا کاراکتر یا رقم‌بودن آنها اغلب مفید است. ماژول **string** ثابت‌های بسیاری ارائه می‌دهند که برای رسیدن به این اهداف سودمندند.

رشته `string.lowercase` شامل تمام حروفی است که سیستم آنها را حروف کوچک به حساب می‌آورد. به‌طور مشابهی `string.uppercase` شامل تمام حروف بزرگ می‌باشد. دستورات زیر را امتحان کنید و ببینید چه نتیجه‌ای می‌گیرید:

```
>>> print string.lowercase
>>> print string.uppercase
>>> print string.digits
```

ما می‌توانیم از این ثابت‌ها و تابع `find` استفاده کنیم و کاراکترها را طبقه‌بندی نماییم. برای مثال اگر `find(lowercase, ch)` مقداری غیر از `-1` را برگرداند، آنگاه `ch` باید از حروف کوچک باشد:

```
def isLower(ch):
    return string.find(string.lowercase, ch) != -1
```

متناوباً می‌توانیم از عملگر `in` که تعیین می‌کند آیا فلان کاراکتر در رشته وجود یا نه، بهره جوییم:

```
def isLower(ch):
    return ch in string.lowercase
```

هنوز هم راه‌حلهایی وجود دارد. ما می‌توانیم از عملگر مقایسه‌ای استفاده کنیم:

```
def isLower(ch):
    return 'a' <= ch <= 'z'
```

اگر `ch` بین `a` تا `z` باشد حتماً از حروف کوچک است.

تمرین ۷-۶: در مورد اینکه کدام نسخه از تابع سریع‌تر است بحث کنید. آیا می‌توانید در مورد علل برتری یکی بر دیگری بحث کنید؟

ثابت دیگری در ماژول `string` تعریف شده که ممکن است موجب تعجب شما شود:

```
>>> print string.whitespace
```

کاراکترهای فضای خالی، مکان نما را بدون چاپ چیزی جابجا می‌کنند. آنها فضاهایی خالی میان کاراکترهای قابل رؤیت ایجاد می‌کنند. ثابت `string.whitespace` شامل تمام کاراکترهای فضای خالی از جمله `space`، `tab` (`\t`) و خط جدید (`\n`) می‌باشد.

۷-۱۱- واژه‌نامه

compound data type (نوع داده‌ای مرکب)

یک نوع داده‌ای که در آن مقادیر از اجزاء یا عناصری که خود مقدار هستند، ساخته می‌شود.

index (اندیس)

متغیر یا مقداری که جهت انتخاب عضوی از یک مجموعه مرتب استفاده می‌شود، نظیر یک کاراکتر از یک رشته.

traverse (پیمایش)

مرور کردن عناصر یک مجموعه و انجام عملیات مشابهی بر روی هر کدام.

slice (برش)

قسمتی از یک رشته که به وسیله بازه‌ای از اندیس‌ها مشخص شده است.

immutable (تغییرناپذیر)

نوع داده‌ای مرکبی که عناصر آن نمی‌توانند مقادیر جدیدی به خود بگیرند.

counter (شمارنده)

متغیری که جهت شمارش چیزی استفاده می‌شود و معمولاً از صفر شروع شده، افزایش می‌یابد.

whitespace (فضای خالی)

هر کاراکتری که مکان نما را بدون چاپ کاراکترهای قابل رؤیت جابجا کند. ثابت `string.whitespace` تمام کاراکترهای فضای خالی را شامل می‌شود.

لیست‌ها



در فصل گذشته با رشته‌ها به‌عنوان یک نوع داده‌ای مرکب آشنا شدید. در این فصل پر کاربردترین نوع داده‌ای پایتون را بررسی می‌کنیم. بی شک استفاده از این نوع داده‌ای برای شما از جذاب‌ترین قسمت‌های برنامه‌نویسی خواهد بود. این نوع داده‌ای **لیست** نام دارد.

لیست مجموعه‌ای از مقادیر مرتب است که در آن هر عضو به وسیله یک اندیس مشخص شده است. مقادیری که یک لیست را می‌سازند، **اعضا**، یا **عناصر** لیست نامیده می‌شوند. لیست‌ها شبیه رشته‌ها - که مجموعه مرتبی از کاراکترها هستند - می‌باشند، با این تفاوت که اعضای لیست می‌توانند از هر نوعی باشند. لیست‌ها و رشته‌ها - و دیگر چیزهایی که شبیه مجموعه‌های مرتب رفتار می‌کنند - دنباله نامیده می‌شوند.

۸-۱- مقادیر لیست

برای ساختن یک لیست جدید چندین راه وجود دارد که ساده‌ترین آنها قرار دادن اعضا، لیست در میان یک جفت براکت ([و]) است:

```
[10, 20, 30, 40]
["spam", "bungee", "swallow"]
```

اولین مثال، لیستی متشکل از چهار عدد صحیح و دومین مثال لیستی شامل سه رشته است. لازم نیست اعضای یک لیست حتماً از یک نوع باشند. لیست زیر شامل یک رشته، یک عدد اعشاری، یک عدد صحیح و (به طور شگفت انگیزی) یک لیست دیگر است:

```
["hello", 2.0, 5, [10, 20]]
```

لیستی که درون لیست دیگر قرار دارد، **لیست تورفته** یا **لیست درونی** نامیده می‌شود. (به مجموعه این لیست‌ها، **لیست‌های تو در تو** گفته می‌شود)

لیست‌هایی که شامل اعداد صحیح متوالی هستند بسیار متداولند، لذا پایتون راه ساده‌ای برای تولید آنها فراهم ساخته است:

```
>>> range(1,5)
[1, 2, 3, 4]
```


تابع **range** دو آرگومان می‌گیرد و لیستی شامل همه اعداد صحیح بین اولین آرگومان و آرگومان دوم را باز می‌گرداند. این لیست، اولین آرگومان را شامل می‌شود اما آرگومان دوم را در بر نمی‌گیرد.

از تابع **range** به دو صورت دیگر هم می‌توان استفاده کرد. با یک آرگومان، که در آن صورت لیست ساخته شده از صفر شروع می‌شود و به یک واحد کمتر از مقدار آرگومان خاتمه می‌یابد:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

اگر آرگومان سومی وجود داشته باشد، این آرگومان اختلاف میان مقادیر متوالی را مشخص می‌کند که اندازه گام نامیده می‌شود. این مثال از 1 تا 10 را با گام 2 می‌شمارد:

```
>>> range(1, 10, 2)
[1, 3, 5, 7, 9]
```

و سرانجام اینکه لیست ویژه‌ای وجود دارد که شامل هیچ عنصری نیست. این لیست، «لیست تهی» نامیده می‌شود و به وسیله `[]` مشخص می‌شود. علاوه بر تمامی راه‌هایی که برای ساختن لیست وجود دارد قادریم مقادیر لیست‌ها را به متغیرها نسبت دهیم و یا آنها را به عنوان آرگومان به توابع بفرستیم.

```
vocabulary = ["ameliorate", "castigate", "defenestrate"]
numbers = [17, 123]
empty = []
print vocabulary, numbers, empty
['ameliorate', 'castigate', 'defenestrate'] [17, 123] []
```

۸-۲- دستیابی به اعضا

نحوه دستیابی به عناصر یک لیست شبیه نحوه دستیابی به کاراکترهای یک رشته است، یعنی عملگر براکت (`[]`) عبارات داخل لیست‌ها را مشخص می‌کند. به خاطر داشته باشید که اندیس‌ها از 0 شروع می‌شوند:

```
print numbers[0]
numbers[1] = 5
```

عملگر براکت می‌تواند در هر کجای یک عبارت ظاهر شود. وقتی این عملگر در سمت چپ یک انتساب نشان داده می‌شود، یکی از عناصر لیست را تغییر می‌دهد، بنابراین یکمین عنصر **numbers** که **123** بود، حال **5** است.

هر عبارت صحیح می‌تواند به عنوان یک اندیس استفاده شود:

```
>>> numbers[3-2]
5
>>> numbers[1.0]
TypeError: sequence index must be integer
```

اگر بخواهید عضوی که در لیست وجود ندارد را بخوانید یا بنویسید، یک خطای زمان اجرا دریافت خواهید کرد:

```
>>> numbers[2] = 5
IndexError: list assignment index out of range
```

اگر اندیس منفی باشد از آخر به اول لیست را می‌شمارد:

```
>>> numbers[-1]
5
>>> numbers[-2]
17
>>> numbers[-3]
IndexError: list index out of range
```

numbers[-1] آخرین عنصر لیست و **numbers[-2]** دومین عنصر از آخر لیست است و **numbers[-3]** وجود ندارد.

استفاده از یک متغیر حلقه به عنوان اندیس لیست بسیار متداول است:

```
horsemen = ["war", "famine", "pestilence", "death"]

i = 0
while i < 4:
    print horsemen[i]
    i = i + 1
```

این حلقه **while** از ۰ تا ۴ را می‌شمارد. وقتی متغیر حلقه، **i**، برابر با ۴ باشد، شرط دیگر برقرار نیست و حلقه پایان می‌یابد. بنابراین بدنه حلقه تنها وقتی **i** برابر با ۰، ۱، ۲ و ۳ است، اجرا می‌شود.

هر بار که در میان حلقه، متغیر **i** به عنوان یک اندیس در لیست استفاده می‌شود، **i**امین عنصر چاپ می‌شود. این الگوی محاسبه را پیمایش لیست می‌نامند.

۸-۳- اندازه لیست

تابع **len** طول یک لیست را بازمی‌گرداند. استفاده از این مقدار به عنوان کران بالای یک حلقه به جای یک ثابت بسیار مفید است. در این روش، اگر اندازه لیست تغییر کند، شما مجبور نخواهید بود که همه حلقه‌ها را در میان برنامه عوض کنید. آنها برای هر اندازه لیست به خوبی کار می‌کنند:

```
horsemen = ["war", "famine", "pestilence", "death"]

i = 0
while i < len(horsemen):
    print horsemen[i]
    i = i + 1
```

آخرین باری که بدنه حلقه اجرا می‌شود، **i** برابر است با **len(horsemen) - 1**، که برابر با اندیس آخرین عنصر است. وقتی **i** برابر با **len(horsemen)** شد، شرط حلقه منتفی می‌گردد و بدنه اجرا نمی‌شود. این خوب است، زیرا **len(horsemen)** اندیس مجازی نیست. اگرچه یک لیست می‌تواند شامل یک لیست دیگر باشد، لیست تورفته (درونی) هنوز به عنوان یک عضو واحد شمرده می‌شود. اندازه این لیست برابر با چهار است:

```
['spam!', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```

تمرین ۸-۱: حلقه‌ای بنویسید که لیست قبلی را پیمایش نماید و اندازه هر عضو را چاپ کند. تحقیق کنید که اگر یک عدد صحیح به تابع **len** بفرستید چه اتفاقی می‌افتد؟

۸-۴- عضویت لیست

in یک عملگر بولی است که عضویت را در یک دنباله آزمایش می‌کند. ما آن را در بخش ۱۰-۷ استفاده نمودیم اما این عملگر با لیست‌ها و دیگر دنباله‌ها نیز کار می‌کند:

```
>>> horsemen = ['war', 'famine', 'pestilence', 'death']
>>> 'pestilence' in horsemen
1
>>> 'debauchery' in horsemen
0
```

از آنجا که **pestilence** یکی از اعضای لیست **horsemen** است عملگر **in** مقدار **true** را برمی‌گرداند و از آنجا که **debauchery** در لیست وجود ندارد، **in** مقدار **false** را برمی‌گرداند. ما می‌توانیم از عملگر **not** در ترکیب با **in** استفاده کنیم و آن را جهت آزمایش عدم وجود عنصری در لیست به کار ببریم:

```
>>> 'debauchery' not in horsemen
1
```

۸-۵- لیست‌ها و حلقه‌های **for**

حلقه **for**، که در بخش ۷-۳ دیدیم، با لیست‌ها هم کار می‌کند. نحوه نگارش کلی برای حلقه **for** بدین شکل است:

```
for VARIABLE in LIST:
    BODY
```

این دستور برابر با این کد است:

```
i = 0
while i < len(LIST):
    VARIABLE = LIST[i]
    BODY
    i = i + 1
```

حلقه **for** فشرده‌تر است زیرا می‌توانیم متغیر حلقه یعنی **i** را حذف کنیم. در اینجا حلقه قبلی را با استفاده از یک حلقه **for** می‌نویسیم:

```
for horseman in horsemen:
    print horseman
```

این عبارت تقریباً شبیه انگلیسی خوانده می‌شود: “برای (هرمقدار) **horseman** در (لیست) **horsemen** (نام) **horseman** را چاپ کن.”

هر عبارت لیست می‌تواند در یک حلقه **for** استفاده شود:

```
for number in range(20):
    if number % 2 == 0:
        print number

for fruit in ["banana", "apple", "quince"]:
    print "I like to eat " + fruit + "s!"
```

مثال اول، تمام اعداد زوج بین 0 تا 19 را چاپ می‌کند. دومین مثال، اشتیاق به میوه‌های گوناگون را بیان می‌کند.

۸-۶- عملگرهای لیست

عملگر + لیست‌ها را به هم ملحق می‌کند:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print c
[1, 2, 3, 4, 5, 6]
```

به طور مشابه عملگر * یک لیست را به تعداد مشخص تکرار می‌کند:

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

اولین مثال [0] را 4 بار و دومین مثال [1, 2, 3] را 3 بار تکرار می‌کند.

۸-۷- برش‌های لیست

عملگر برش را که ما در بخش ۷-۴ دیدیم، بر روی لیست‌ها نیز کار می‌کند:

```
>>> list = ['a', 'b', 'c', 'd', 'e', 'f']
>>> list[1:3]
['b', 'c']
>>> list[:4]
['a', 'b', 'c', 'd']
>>> list[3:]
```

```
['d', 'e', 'f']  
>>> list[:]  
['a', 'b', 'c', 'd', 'e', 'f']
```

۸-۸- لیست‌ها تغییرپذیرند

بر خلاف رشته‌ها، لیست‌ها قابل انعطافند، یعنی ما می‌توانیم اعضای آنها را تغییر دهیم. با استفاده از عملگر براکت در سمت چپ یک نسبت‌دهی می‌توانیم عضو مشخصی را به‌روز درآوریم:

```
>>> fruit = ["banana", "apple", "quince"]  
>>> fruit[0] = "pear"  
>>> fruit[-1] = "orange"  
>>> print fruit  
['pear', 'apple', 'orange']
```

با عملگر برش می‌توانیم چند عضو را یکباره به‌روز درآوریم:

```
>>> list = ['a', 'b', 'c', 'd', 'e', 'f']  
>>> list[1:3] = ['x', 'y']  
>>> print list  
['a', 'x', 'y', 'd', 'e', 'f']
```

همچنین ما می‌توانیم اعضای یک لیست را با استفاده از نسبت‌دهی یک لیست خالی به‌آنها، حذف کنیم:

```
>>> list = ['a', 'b', 'c', 'd', 'e', 'f']  
>>> list[1:3] = []  
>>> print list  
['a', 'd', 'e', 'f']
```

همچنین ما می‌توانیم عناصری را با جا دادن در یک برش خالی به اندیس مورد نظر در لیست اضافه کنیم:

```
>>> list = ['a', 'd', 'f']  
>>> list[1:1] = ['b', 'c']  
>>> print list  
['a', 'b', 'c', 'd', 'f']  
>>> list[4:4] = ['e']  
>>> print list  
['a', 'b', 'c', 'd', 'e', 'f']
```

۸-۹- حذف لیست

به کار بردن برش‌ها برای حذف اعضای لیست می‌تواند دشوار و به همین دلیل خطاساز باشد. پایتون روشی مهیا می‌کند که خواناتر است. **del** یک عضو را از لیست حذف می‌کند:

```
>>> a = ['one', 'two', 'three']
>>> del a[1]
>>> a
['one', 'three']
```

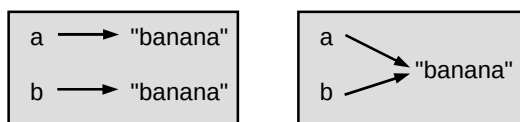
به طور معمول، برش‌ها همهٔ عناصر را (در محدودهٔ اندیس اول تا اندیس دوم) شامل می‌شوند، اما خود اندیس دوم را در بر نمی‌گیرند.

۸-۱۰- اشیاء و مقادیر

به دستورات انتساب زیر توجه کنید:

```
a = "banana"
b = "banana"
```

می‌دانیم که **a** و **b** به یک رشته با حروف "banana" اشاره خواهند کرد، اما نمی‌توانیم بگوییم که آنها به رشتهٔ یکسانی اشاره می‌کنند یا نه. دو حالت ممکن است وجود داشته باشد:



شکل ۸-۱

در اولین مورد، **a** و **b** به دو چیز مختلف اشاره می‌کنند که مقدار مشابهی دارند، اما در دومین مورد هر دو به یک چیز اشاره می‌کنند. این چیزها اسم دارند و **شیء** نامیده می‌شوند. یک شیء هر چیزی است که متغیری بتواند به آن اشاره کند.

هر شیء یک شناسهٔ منحصر به فرد دارد که ما می‌توانیم آن را به وسیلهٔ تابع **id** بدست آوریم. با چاپ کردن شناسهٔ **a** و **b** می‌توانیم بگوییم که آیا آنها به یک شیء یکسان اشاره می‌کنند یا نه:

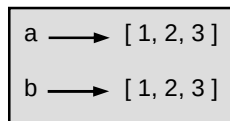
```
>>> id(a)
135044008
>>> id(b)
135044008
```

در حقیقت ما یک شناسه یکسان را دو بار به دست آورده ایم. یعنی پایتون فقط یک رشته ساخته است و هر دو متغیر **a** و **b** به آن تک رشته اشاره می کنند.

جالب است بدانید که لیست ها به طور متفاوتی رفتار می کنند. وقتی ما دو لیست را می سازیم،

دو شیء به دست می آوریم:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> id(a)
135045528
>>> id(b)
135041704
```



شکل ۸-۲

بنابراین نمودار حالت به صورت زیر است:

a و **b** مقدار مشابهی دارند، اما به شیء یکسانی اشاره نمی کنند.

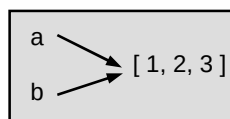
۸-۱۱- بدل سازی

از آنجا که متغیرها به اشیاء اشاره می کنند، اگر ما یک متغیر را به متغیر دیگری نسبت دهیم،

هر دو به یک چیز اشاره می کنند:

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

در این مورد، نمودار حالت مطابق شکل ۸-۳ است:



شکل ۸-۳

چون یک لیست یکسان دو نام متفاوت **a** و **b** دارد، می‌گوییم آن لیست **بدل** شده است. تغییرات به‌وجود آمده در یک بدل در دیگری تأثیر می‌گذارد:

```
>>> b[0] = 5
>>> print a
[5, 2, 3]
```

اگرچه این رفتار می‌تواند مفید باشد، گاهی اوقات غیرمنتظره یا نامطلوب است. به‌طور کلی خودداری از بدل‌سازی هنگام کار با اشیاء تغییرپذیر مطمئن‌تر است. البته برای اشیاء تغییرناپذیر مشکلی وجود ندارد. به این دلیل است که پایتون هرگاه فرصت را برای صرفه‌جویی مناسب ببیند، در مورد بدل‌سازی رشته‌ها آزادانه عمل می‌کند.

۸-۱۲- تکثیر لیست‌ها

اگر بخواهیم لیستی را تغییر دهیم و همچنین یک کپی از لیست اصلی نگه داریم، به یک توانایی برای ساختن کپی از خود لیست (نه فقط آدرس آن)، نیاز داریم. برای جلوگیری از ابهام کلمه کپی به این فرایند اصطلاحاً **تکثیر** گفته می‌شود. ساده‌ترین راه تکثیر یک لیست استفاده از عملگر برش است:

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

گرفتن هر برش از **a** یک لیست جدید می‌سازد. در این حالت، برش برای محتوای کل لیست اتفاق افتاده است.

اکنون بدون نگرانی در مورد لیست **a** می‌توانیم لیست **b** را آزادانه تغییر دهیم:

```
>>> b[0] = 5
>>> print a
[1, 2, 3]
```

تمرین ۸-۲: یک نمودار حالت برای **a** و **b** قبل و بعد از این تغییر رسم کنید.

۸-۱۳- لیست‌ها به عنوان پارامتر

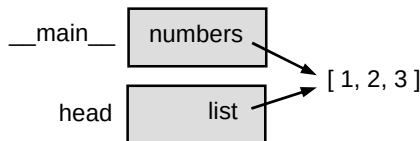
فرستادن یک لیست به عنوان آرگومان در واقع یک آدرس از لیست را به تابع می‌فرستد، نه یک کپی از آن. مثلاً تابع **head** یک لیست را به عنوان پارامتر می‌گیرد و اولین عضو آن را برمی‌گرداند:

```
def head(list):
    return list[0]
```

در اینجا طرز استفاده از این تابع نشان داده شده است:

```
>>> numbers = [1, 2, 3]
>>> head(numbers)
1
```

پارامتر **list** و متغیر **numbers** هر دو بدل‌هایی برای یک شیء یکسان هستند. نمودار



شکل ۸-۴

حالت به صورت زیر است:

از آنجا که شیء لیست برای هر دو قاب مشترک است، آن را در بین دو قاب رسم کرده‌ایم. اگر یک تابع تغییری در پارامتر لیست بدهد، فراخواننده تغییرات را مشاهده می‌کند. برای مثال **deleteHead** اولین عضو یک لیست را حذف می‌کند:

```
def deleteHead(list):
    del list[0]
```

در اینجا طرز استفاده از این تابع را می‌بینید:

```
>>> numbers = [1, 2, 3]
>>> deleteHead(numbers)
>>> print numbers
[2, 3]
```

اگر یک تابع لیستی را بازگرداند، آدرس لیست را باز می‌گرداند. برای نمونه، **tail** لیستی شامل همهٔ اعضاء غیر از عضو اول را باز می‌گرداند:

```
def tail(list):
    return list[1:]
```

در اینجا چگونگی کار **tail** را می‌بینید:

```
>>> numbers = [1, 2, 3]
>>> rest = tail(numbers)
>>> print rest
[2, 3]
```

چون مقدار بازگشتی با یک عملگر برش ساخته شده است، بنابراین یک لیست جدید است و هر تغییری که از این پس در **rest** داده شود هیچ تأثیری در **numbers** نخواهد گذاشت.

۸-۱۴- لیست‌های تودرتو

یک لیست تورفته لیستی است که به عنوان یک عضو در لیست دیگری نمایش داده شود. در این لیست عضو با اندیس 3، لیست تورفته است:

```
>>> list = ["hello", 2.0, 5, [10, 20]]
```

اگر ما **list[3]** را چاپ کنیم، **[10, 20]** را نتیجه می‌گیریم. برای به‌دست آوردن اعضای لیست تورفته می‌توانیم به دو طریق عمل کنیم:

```
>>> elt = list[3]
>>> elt[0]
10
```

یا حتی می‌توانیم آنها را با هم ترکیب کنیم:

```
>>> list[3][1]
20
```

عملگرهای براکت (**[]**)، از چپ به راست ارزیابی می‌شوند و بنابراین این عبارت سومین عضو لیست را می‌گیرد و اولین عنصر آن را بیرون می‌کشد.

۸-۱۵- ماتریس‌ها

لیست‌های تودرتو اغلب برای نشان دادن ماتریس‌ها استفاده می‌شوند. برای نمونه ماتریس:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

ممکن است به صورت زیر نمایش داده شود:

```
>>> matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

matrix لیستی با سه عضو است که هر عضو یکی از سطرهای ماتریس می‌باشد. ما می‌توانیم یک سطر کامل از ماتریس را به روش معمول به دست آوریم:

```
>>> matrix[1]
[4, 5, 6]
```

یا اینکه می‌توانیم با استفاده از روش دو اندیسی یک عضو واحد را از ماتریس خارج کنیم:

```
>>> matrix[1][1]
5
```

اندیس اول، سطر و اندیس دوم، ستون را انتخاب می‌کند. اگرچه این راه نمایش ماتریس‌ها معمول است اما تنها راه ممکن نیست. یک تغییر کوچک، برای استفاده از لیست ستون‌ها به جای لیست سطرها وجود دارد. در آینده با استفاده از یک دیکشنری چاره بنیادی تری خواهیم دید.

۸-۱۶- رشته‌ها و لیست‌ها

دو مورد از مفیدترین توابع ماژول **string**، لیست‌هایی از رشته‌ها را به هم می‌آمیزد. تابع **split** یک رشته را در لیستی از کلمات می‌شکند. به طور پیش فرض هر تعداد از کاراکترهای فضای خالی، یک مرز در بین کلمات در نظر گرفته شده است:

```
>>> import string
>>> song = "The rain in Spain..."
>>> string.split(song)
['The', 'rain', 'in', 'Spain...']
```

یک آرگومان اختیاری با نام **حایل** می‌تواند برای مشخص کردن اینکه کدام کاراکترها مرز کلمات در نظر گرفته شوند، استفاده شود. مثال زیر رشته **'ai'** را به عنوان حایل استفاده می‌کند:

```
>>> string.split(song, 'ai')
['The r', 'n in Sp', 'n...']
```

توجه کنید که حایل در لیست ظاهر نمی‌شود.

تابع **join** برعکس تابع **split** عمل می‌کند. این تابع لیستی از رشته‌ها را می‌گیرد و آنها را با یک فضای خالی میان هر دو جفت آن به هم متصل می‌کند:

```
>>> list = ['The', 'rain', 'in', 'Spain...']
>>> string.join(list)
'The rain in Spain...'
```

مانند **split**، تابع **join** یک حایل اختیاری می‌گیرد که بین اعضاء گذاشته می‌شود:

```
>>> string.join(list, '_')
'The_rain_in_Spain...'
```

تمرین ۸-۳: ارتباط میان **string.join(string.split(song))** و **song** را شرح دهید. آیا آنها برای همه رشته‌ها یکسان هستند؟ چه موقع آنها متفاوت می‌شوند؟

۸-۱۷- واژه‌نامه

list (لیست)

مجموعه‌ای دارای نام از اشیاء که هر شیء با یک اندیس مشخص می‌شود.

element (عنصر، عضو)

یکی از مقادیر لیست (یا دیگر دنباله‌ها). عملگر براکت عناصر یک لیست را انتخاب می‌کند.

sequence (دنباله)

هر یک از انواع داده که شامل مجموعه‌ای از عناصر مرتب باشد، به‌طوری که هر عضو به‌وسیله اندیسی مشخص شده باشد.

nested list (لیست تورفته، لیست درونی)

لیستی که عضوی از یک لیست دیگر باشد.

list traversal (پیمایش لیست)

دستیابی متوالی و مرتب به هر یک از عناصر لیست.

object (شیء)

چیزی که یک متغیر بتواند به آن اشاره کند.

aliases (بدل‌ها)

متغیرهای چندگانه‌ای که آدرس شیء واحدی را در بر دارند.

clone (تکثیر)

ساختن شیء جدیدی که مقدار مشابه آن وجود آن دارد. کپی کردن آدرس در یک شیء، یک بدل می‌سازد اما شیء را تکثیر نمی‌کند.

delimiter (حائل)

کاراکتر یا رشته‌ای که برای نشان دادن محل تفکیک (یا الحاق) رشته بکار می‌رود.

چندتایی‌ها



تا به حال دو نوع داده‌ای مرکب دیده‌اید: رشته‌ها که از کاراکترها تشکیل شده‌اند و لیست‌ها که شامل عناصری از هر نوع هستند. در این فصل یک نوع داده‌ای مرکب جدید را به شما معرفی می‌نمایم و سپس خصوصیات آن را بررسی می‌کنیم.

یکی از تفاوت‌هایی که میان لیست‌ها و رشته‌ها وجود دارد، این است که عناصر یک لیست می‌توانند تغییر یابند، اما کاراکترهای یک رشته نمی‌توانند. به بیان دیگر رشته‌ها **تغییرناپذیر** و لیست‌ها **تغییرپذیر** هستند.

۹-۱- تغییرپذیری و چندتایی‌ها

در پایتون نوع داده‌ای مرکبی به نام **چندتایی** وجود دارد که شبیه لیست است، با این تفاوت که تغییرناپذیر می‌باشد. از لحاظ نگارشی یک چندتایی لیستی از مقادیر است که به وسیله کاما از هم جدا شده‌اند:

```
>>> tuple = 'a', 'b', 'c', 'd', 'e'
```

اگرچه ضروری نیست، اما مرسوم است که چندتایی‌ها را در پرانتز قرار می‌دهند:

```
>>> tuple = ('a', 'b', 'c', 'd', 'e')
```

برای ساختن یک چندتایی با یک عضو، باید کامایی در آخر آن منظور کنیم:

```
>>> t1 = ('a',)
>>> type(t1)
<type 'tuple'>
```

بدون گذاشتن کاما، پایتون با **('a')** به عنوان رشته‌ای در پرانتز رفتار می‌کند:

```
>>> t2 = ('a')
>>> type(t2)
<type 'string'>
```

تنها از نحوه نگارش این‌طور بر می‌آید که عملیات بر روی چندتایی‌ها شبیه عملیات بر روی لیست‌ها است. عملگر اندیس عنصری از یک چندتایی را انتخاب می‌کند:


```
>>> tuple = ('a', 'b', 'c', 'd', 'e')
>>> tuple[0]
'a'
```

و عملگر برش محدوده‌ای از عناصر را انتخاب می‌کند:

```
>>> tuple[1:3]
('b', 'c')
```

اما اگر سعی کنیم عضوی از چندتایی را تغییر دهیم، یک پیغام خطا می‌گیریم:

```
>>> tuple[0] = 'A'
TypeError: object doesn't support item assignment
```

البته اگر حتی اگر نتوانیم اعضای یک چندتایی را تغییر دهیم، می‌توانیم آن را با یک چندتایی متمایز جایگزین کنیم:

```
>>> tuple = ('A',) + tuple[1:]
>>> tuple
('A', 'b', 'c', 'd', 'e')
```

۹-۲- نسبت‌دهی یک چندتایی

بعضی اوقات تعویض مقادیر دو متغیر مفید است. با دستورات نسبت‌دهی مرسوم، باید از یک متغیر موقت استفاده کنیم. برای مثال جهت تعویض مقادیر **a** و **b**:

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

اگر مجبور باشیم این عمل را اغلب اوقات انجام دهیم، این مسیر طاقت‌فرسا و خسته کننده می‌شود. پایتون فرمی از **انتساب چندتایی** را تدارک دیده است که این مشکل را با ظرافت حل می‌کند:

```
>>> a, b = b, a
```

سمت چپ عبارت، یک چندتایی از متغیرها است و در سمت راست یک چندتایی از مقادیر. هر مقدار به متغیر نظیرش اختصاص می‌یابد. تمام عبارات سمت راست قبل از هر نسبت‌دهی بررسی می‌شوند. این خصیصه به انتساب چندتایی‌ها تنوع می‌بخشد.

طبیعتاً تعداد متغیرها در سمت چپ باید با تعداد مقادیر در سمت راست برابر باشد:

```
>>> a, b, c, d = 1, 2, 3
ValueError: unpack tuple of wrong size
```

۹-۳- چندتایی‌ها به عنوان مقادیر بازگشتی

توابع می‌توانند چندتایی‌ها را به عنوان مقادیر بازگشتی برگردانند. برای مثال می‌توانیم تابعی بنویسیم که دو پارامتر را جابجا کند:

```
def swap(x, y):
    return y, x
```

سپس می‌توانیم مقدار برگشتی را به یک چندتایی با دو متغیر نسبت دهیم:

```
a, b = swap(a, b)
```

در این مورد، تبدیل **swap** به یک تابع کار بی‌هوده‌ای است. در حقیقت احتمال اشتباه وسوسه‌انگیزی در زمان بسته‌بندی **swap** وجود دارد:

```
def swap(x, y):                # incorrect version
    x, y = y, x
```

اگر ما تابع **swap** را به صورت زیر فراخوانی کنیم:

```
swap(a, b)
```

آنگاه **a** و **x** بدل‌هایی برای یک مقدار یکسانند. تغییر **x** درون **swap** باعث می‌شود **x** به مقدار متفاوتی رجوع کند، اما تأثیری روی متغیر **a** در **__main__** ندارد. به‌طور مشابه، تغییر دادن **y** تأثیری روی مقدار **b** ندارد. این تابع بدون تولید هیچ پیغام خطایی اجرا می‌شود، اما کاری را که ما می‌خواستیم انجام نمی‌دهد. این مورد، مثالی برای خطاهای معنایی است.

تمرین ۹-۱: نمودار حالت را برای این تابع رسم کنید تا ببینید چرا این تابع درست کار نمی‌کند.

۹-۴- اعداد تصادفی

بسیاری از برنامه‌های کامپیوتری هر بار که اجرا می‌شوند عمل یکسانی انجام می‌دهند، لذا آنها را **قطعی** می‌نامند. قطعی بودن برنامه تا وقتی خوب است که ما محاسبه یکسانی را برای گرفتن نتیجه‌ای یکسان انتظار داریم. با این وجود برای برخی کاربردها، ما می‌خواهیم عمل کامپیوتر غیرقابل پیش‌گویی باشد. بازی‌های کامپیوتری مثالی آشکار است، اما استفاده‌های بیشتری هم وجود دارد. غیرقطعی کردن برنامه به نحو درست آسان به نظر نمی‌رسد، اما راه‌هایی وجود دارد که بتوان حداقل آن را شبیه به برنامه‌های غیرقطعی کرد. یکی از این راه‌ها تولید اعداد تصادفی و استفاده از آنها جهت تعیین خروجی برنامه است. پایتون تابع پیش‌ساخته‌ای ارائه می‌دهد که اعداد شبه‌تصادفی تولید می‌کند. این اعداد از لحاظ ریاضی واقعاً تصادفی نیستند، اما برای منظور ما کار می‌کنند. ماژول **random** شامل تابعی با نام **random** است که عددی اعشاری بین **0.0** و **1.0** برمی‌گرداند. هر بار که تابع **random** را فرا می‌خوانید، عدد بعدی از یک سری طولانی را دریافت می‌کنید. برای دیدن یک مثال، این حلقه را اجرا کنید:

```
import random
```

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

برای تولید یک عدد تصادفی بین **0.0** و یک کران بالاتر مثل **high**، **x** را در **high** ضرب کنید.

تمرین ۹-۲: یک عدد تصادفی بین **high** و **low** پیدا کنید.

تمرین ۹-۳: عدد تصادفی صحیحی بین **low** و **high** تولید کنید که دو نقطه را هم (به‌عنوان کران بالا و پایین) شامل شود.

۹-۵- لیستی از اعداد تصادفی

قدم اول تولید لیستی از مقادیر تصادفی است. تابع **randomList** یک پارامتر صحیح می‌گیرد و لیستی از اعداد تصادفی با طول داده شده را برمی‌گرداند. این کار با لیستی شامل **n** عدد

صفر شروع می‌شود. هر بار که حلقه اجرا می‌شود یکی از اعضاها با عددی تصادفی جایگزین می‌شود. مقدار برگشتی، ارجاعی به لیست کامل است:

```
def randomList(n):
    s = [0] * n
    for i in range(n):
        s[i] = random.random()
    return s
```

این تابع را با لیستی هشت عضوی آزمایش می‌کنیم. برای اشکال‌زدایی بهتر است از یک لیست کوچک شروع کنیم:

```
>>> randomList(8)
[0.15156642489, 0.498048560109, 0.810894847068, 0.360371157682,
0.275119183077, 0.328578797631, 0.759199803101, 0.800367163582]
```

اعداد تولید شده توسط تابع **random** باید به‌طور یکنواخت توزیع شده باشند. یعنی احتمال همه مقادیر برابر باشد.

اگر بازه مقادیر ممکن را به طبقات مساوی تقسیم کنیم و تعداد دفعاتی که یک مقدار تصادفی در هر طبقه قرار گرفته را بشماریم، باید تقریباً عدد یکسانی از هر طبقه به‌دست آوریم. می‌توانیم این فرضیه را با نوشتن برنامه‌ای که بازه مقادیر را به طبقاتی تقسیم کند و تعداد مقادیر را در هر یک بشمارد، آزمایش کنیم.

۹-۶- شمارش

یک راه مناسب برای دستیابی به این‌گونه مسائل، تقسیم مسئله‌ها به زیرمسئله‌ها و نیز یافتن زیرمسئله‌ای است که با الگوهای محاسباتی که قبلاً دیده‌اید سازگار باشند. در این مورد می‌خواهیم لیستی از اعداد را پیمایش کنیم و تعداد دفعات وقوع مقداری را در یک بازه بشماریم. این برنامه آشنا به نظر می‌رسد؛ در بخش ۷-۸ برنامه‌ای نوشتیم که یک رشته را می‌پیمود و تعداد دفعات ظاهر شدن حرف خاصی را می‌شمرد. بنابراین می‌توانیم به کپی کردن برنامه قبلی اقدام کنیم و آن را با مسئله کنونی وفق دهیم. برنامه اصلی چنین بود:

```
count = 0
for char in fruit:
    if char == 'a':
        count = count + 1
print count
```

قدم اول جایگزین کردن **fruit** با **list** و **char** با **num** است. این تغییر برنامه را عوض نمی‌کند بلکه تنها آن را خوانا تر می‌سازد.

قدم دوم تغییر آزمایش است. ما علاقه‌ای به یافتن حروف نداریم بلکه می‌خواهیم **num** را در صورت وجود بین مقادیر داده شده **low** و **high** پیدا کنیم.

```
count = 0
for num in list:
    if low < num < high:
        count = count + 1
print count
```

قدم آخر بسته‌بندی این کد در تابعی به نام **inBucket** است. پارامترها **list**، **low** و **high** هستند:

```
def inBucket(list, low, high):
    count = 0
    for num in list:
        if low < num < high:
            count = count + 1
    return count
```

با کپی کردن و تغییر یک برنامه موجود، قادر شدیم که این تابع را سریع‌تر بنویسیم و از اتلاف وقت در خطایابی جلوگیری کنیم. این طرح توسعه را **تطابق الگویی** می‌نامند. اگر تصمیم گرفتید روی مسئله‌ای که قبلاً حل کرده‌اید کار کنید، می‌توانید راه‌حل آن را دوباره به کار بندید.

۹-۷- طبقات متعدد

همچنان که تعداد طبقات افزایش می‌یابد، **inBucket** کمی کند می‌شود. این تابع با دو طبقه بد کار نمی‌کند:

```
low = inBucket(a, 0.0, 0.5)
high = inBucket(a, 0.5, 1)
```

اما با چهار طبقه کند می‌شود.

```
bucket1 = inBucket(a, 0.0, 0.25)
bucket2 = inBucket(a, 0.25, 0.5)
bucket3 = inBucket(a, 0.5, 0.75)
bucket4 = inBucket(a, 0.75, 1.0)
```

در اینجا دو مشکل وجود دارد، یکی اینکه ما مجبوریم نام‌های جدیدی برای متغیرهای هر نتیجه داشته باشیم و دیگر اینکه باید عرض طبقات را محاسبه کنیم.

ابتدا مشکل دوم را حل می‌کنیم. اگر تعداد طبقات **numBuckets** باشد، آنگاه عرض هر طبقه **1/numBuckets** خواهد بود.

برای محاسبه عرض هر طبقه از یک حلقه استفاده می‌کنیم. متغیر حلقه، **i**، از 1 تا **numBuckets-1** را می‌شمارد:

```
bucketWidth = 1.0 / numBuckets
for i in range(numBuckets):
    low = i * bucketWidth
    high = low + bucketWidth
    print low, "to", high
```

برای محاسبه حد پایین هر طبقه، متغیر حلقه را در عرض طبقه ضرب می‌کنیم. حد بالای طبقه تنها به اندازه **bucketWidth** (عرض طبقه)، از حد پایین فاصله دارد. با احتساب **numBuckets = 8** خروجی به صورت زیر است:

```
0.0 to 0.125
0.125 to 0.25
0.25 to 0.375
0.375 to 0.5
0.5 to 0.625
0.625 to 0.75
0.75 to 0.875
0.875 to 1.0
```

شما می‌توانید هم‌عرض بودن طبقات را که باعث می‌شود با هم نقطه اشتراکی (تداخلی) نداشته باشند، بررسی و تأیید نمایید. همچنین واضح است که تمام بازه **0.0** و **1.0** پوشش داده شده است. اکنون به مشکل اول باز می‌گردیم. ما به راهی جهت ذخیره هشت عدد صحیح با استفاده از متغیر حلقه که به نوبت به تمام طبقات اشاره می‌کند نیاز داریم. تا به حال شما باید به فکر لیست

افتاده باشید. لیست طبقات باید خارج از حلقه ساخته شود، زیرا تنها یک بار نیاز به انجام این کار است. در میان حلقه، ما `inBucket` را مکرراً فراخوانی می‌کنیم و `inBucket` عنصر آن را به‌هنگام می‌سازیم:

```
numBuckets = 8
buckets = [0] * numBuckets
bucketWidth = 1.0 / numBuckets
for i in range(numBuckets):
    low = i * bucketWidth
    high = low + bucketWidth
    buckets[i] = inBucket(list, low, high)
print buckets
```

این کد با یک لیست 1000 عضوی، لیست طبقات زیر را تولید می‌کند:

```
[138, 124, 128, 118, 130, 117, 114, 131]
```

این اعداد تقریباً نزدیک به ۱۲۵ هستند و این همان چیزی است که انتظار داشتیم. لاقبل آنها آنقدر نزدیک هستند که به کارکرد مولد اعداد تصادفی ایمان بیاوریم.

تمرین ۹-۴: این تابع را با چند لیست طولانی‌تر آزمایش کنید و ببینید که آیا تعداد مقادیر هر طبقه به سمت عدد ثابتی میل می‌کنند یا خیر.

۹-۸- یک راه‌حل تک‌گذری

اگرچه این برنامه کار می‌کند، اما از نظر کارایی، توانایی چندانی ندارد. هر بار که این برنامه `inBucket` را صدا می‌زند، سراسر لیست را می‌پیماید. همچنان که تعداد طبقات افزایش می‌یابد، پیمایش لیست به دفعات زیادی تکرار می‌شود.

بهتر است عبور واحدی از روی لیست داشته باشیم و برای هر مقدار، اندیس طبقه‌ای را که در آن قرار گرفته محاسبه نماییم. سپس می‌توانیم شمارندهٔ مربوط را افزایش دهیم.

در بخش قبل، اندیسی به‌نام `i` گرفتیم و آن را در `bucketWidth` ضرب کردیم تا حد پایین طبقهٔ داده شده را بیابیم. اکنون می‌خواهیم مقداری در بازهٔ ۰.۰ تا ۱.۰ بگیریم و اندیس طبقه‌ای را که در آن قرار گرفته پیدا کنیم. از آنجا که این مسئله عکس مسئلهٔ قبل است، ممکن است حدس زده باشید که به‌جای ضرب در `bucketWidth` باید عمل تقسیم را انجام دهید. این حدس درست است.

از آنجا که `bucketWidth = 1.0 / numBuckets`، تقسیم بر `bucketWidth` معادل ضرب در `numBuckets` است. اگر عددی را از بازه `0.0` تا `0.1` در `numBucket` ضرب کنیم عددی در بازه `0.0` تا `numbuckets` به دست می آوریم. اگر آن عدد را به کوچک ترین عدد صحیح بعدی گرد کنیم، دقیقاً آنچه را که به دنبال آن بودیم یعنی یک اندیس طبقه را به دست می آوریم:

```
numBuckets = 8
buckets = [0] * numBuckets
for i in list:
    index = int(i * numBuckets)
    buckets[index] = buckets[index] + 1
```

ما از تابع `int` به منظور تبدیل یک عدد اعشاری به عددی صحیح استفاده کرده ایم. آیا امکان دارد محاسبات فوق، اندیسی خارج از محدوده تولید کند؟ (اندیسی منفی یا بزرگ تر از `len(buckets) - 1`) لیستی شبیه به `buckets` که شمارش تعداد مقادیر در هر بازه را نگه می دارد `histogram` نامیده می شود.

تمرین ۹-۵: تابعی به نام `histogram` بنویسید که یک لیست و تعداد طبقات را به عنوان پارامتر بگیرد و یک `histogram` با تعداد طبقات داده شده برگرداند.

۹-۹- واژه نامه

immutable type (نوع تغییر ناپذیر)

نوعی که در آن عناصر، قابل تغییر و تحول نیستند. نسبت دهی به عناصر یا برش های انواع تغییر ناپذیر به خطا منجر می شود.

mutable type (نوع تغییر پذیر)

نوعی از داده ها که در آن عناصر قابل تغییرند. تمام انواع داده های قابل تغییر از نوع ترکیبی هستند. لیست ها و دیکشنری ها از نوع تغییر پذیر و رشته ها و چندتایی ها تغییر ناپذیرند.

tuple (چندتایی)

نوعی داده‌ترتیبی که شبیه لیست می‌باشد، با این تفاوت که تغییرناپذیر است. چندتایی‌ها می‌توانند هر کجا که به یک داده‌تغییرناپذیر نیاز است، استفاده شوند؛ نظیر یک کلید در یک دیکشنری.

tuple assignment (انتساب چندتایی)

نوعی نسبت‌دهی به تمام عناصر یک چندتایی با استفاده از یک دستور نسبت‌دهی واحد. نسبت‌دهی چندتایی بیشتر به صورت موازی اتفاق می‌افتد تا به صورت توالی و تسلسل، که این ویژگی چندتایی‌ها را به‌منظور جابجایی مقادیر مفید می‌سازد.

deterministic (قطعی)

برنامه‌ای که در هر بار فراخوانی عمل یکسانی را انجام می‌دهد.

pseudorandom (شبیه تصادفی)

دنباله‌ای از اعداد که به نظر تصادفی می‌رسند اما در حقیقت نتیجه یک سری محاسبات قطعی هستند.

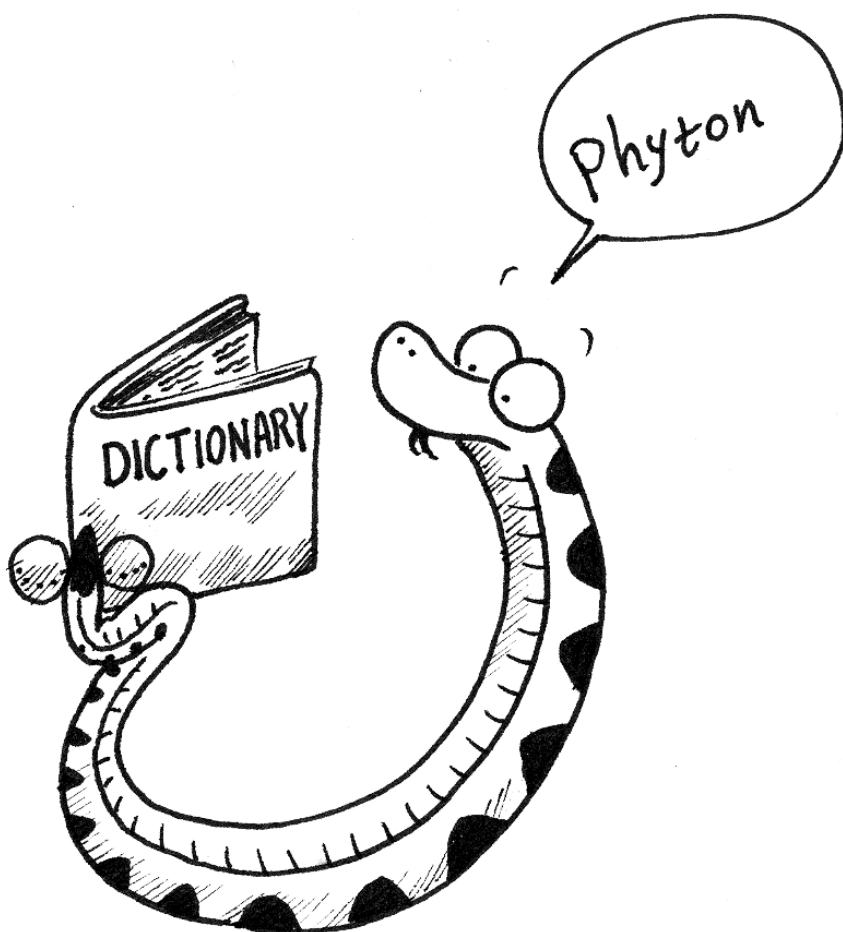
patern matching (تطابق الگویی)

طرحی برای توسعه برنامه که در برگیرنده شناسایی الگوی شناخته‌شده و کپی کردن آن راه‌حل‌ها برای مسائل مشابه می‌باشد.

histogram

لیستی از اعداد صحیح که در آن هر عنصر دفعات وقوع عملی را می‌شمارند.

دیکشنری‌ها



انواع داده‌های مرکبی که تا به حال آموخته‌اید - رشته‌ها، لیست‌ها و چندتایی‌ها - از اعداد صحیح به عنوان اندیس استفاده می‌کنند. اگر سعی کنید انواع دیگر داده‌ها را به عنوان اندیس به کار ببرید، پیغام خطا دریافت می‌کنید.

دیکشنری‌ها همچون دیگر انواع داده‌های مرکب هستند با این تفاوت که می‌توانند هر نوع داده‌ی تغییر ناپذیری را به عنوان اندیس به کار برند. برای مثال، یک دیکشنری جهت ترجمه‌ی کلمات انگلیسی به اسپانیایی می‌سازیم. رشته‌ها، اندیس‌های این دیکشنری هستند.

یک راه ساختن دیکشنری این است که کار را با یک دیکشنری تهی شروع کنیم و اعضا را به آن بیافزاییم. دیکشنری تهی با علامت `{}` مشخص می‌شود.

```
>>> eng2sp = {}
>>> eng2sp['one'] = 'uno'
>>> eng2sp['two'] = 'dos'
```

اولین انتساب، یک دیکشنری با نام `eng2sp` می‌سازد؛ دیگر انتساب‌ها عناصر جدیدی به دیکشنری اضافه می‌کنند. می‌توانیم مقدار جاری را به روش معمول چاپ کنیم:

```
>>> print eng2sp
{'one': 'uno', 'two': 'dos'}
```

عناصر یک دیکشنری در لیستی که اعضای آن به‌وسیله‌ی کاما از هم تفکیک شده‌اند، ظاهر می‌شوند. هر قلم داده شامل یک اندیس و یک مقدار است که این دو به‌وسیله‌ی علامت کولن (`:`) از هم جدا شده‌اند. در یک دیکشنری اندیس‌ها **کلید** نامیده می‌شوند. بنابراین اعضا را **جفت‌های کلید-مقدار** می‌نامند.

راه دیگری برای ساختن یک دیکشنری، فراهم کردن لیستی از جفت‌های کلید-مقدار با استفاده از نحوه‌ی نگارشی همچون خروجی اخیر است:

```
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

اگر مقدار `eng2sp` را مجدداً چاپ کنیم، شگفت‌زده خواهیم شد:

```
>>> print eng2sp
{'one': 'uno', 'three': 'tres', 'two': 'dos'}
```

جفت‌های کلید-مقدار به ترتیب نیستند! خوشبختانه، تا هنگامی که عناصر یک دیکشنری با اعداد صحیح اندیس‌گذاری نشده‌اند دلیلی برای حفظ ترتیب اعضا وجود ندارد، در عوض ما از کلیدها جهت مراجعه به مقادیر متناظر بهره می‌گیریم:

```
>>> print eng2sp['two']  
'dos'
```

کلید 'two' مقدار 'dos' را نتیجه می‌دهد، هرچند در مکانِ جفت کلید-مقدار سوم ظاهر شده باشد.

۱۰-۱- عملیات بر روی دیکشنری‌ها

دستور **del** یک جفت کلید-مقدار را از دیکشنری حذف می‌کند. برای مثال، دیکشنری زیر شامل نام میوه‌های گوناگون و تعداد هر میوه در انبار است:

```
>>> inventory = {'apples': 430, 'bananas': 312, 'oranges': 525,  
'pears': 217}  
>>> print inventory  
{'oranges': 525, 'apples': 430, 'pears': 217, 'bananas': 312}
```

اگر شخصی تمام گلابی‌ها را بخرد، ما می‌توانیم این قلم را از دیکشنری حذف کنیم:

```
>>> del inventory['pears']  
>>> print inventory  
{'oranges': 525, 'apples': 430, 'bananas': 312}
```

یا اگر ما منتظر رسیدن گلابی‌های دیگری در عوض فروختن این‌ها هستیم، تنها کافی است مقدار مرتبط با آن را تغییر دهیم:

```
>>> inventory['pears'] = 0  
>>> print inventory  
{'oranges': 525, 'apples': 430, 'pears': 0, 'bananas': 312}
```

تابع **len** بر روی دیکشنری‌ها هم کار می‌کند و تعداد جفت‌های کلید-مقدار را باز می‌گرداند:

```
>>> len(inventory)  
4
```

۱۰-۲- متدهای دیکشنری

یک متد شبیه به یک تابع است - پارامتری می‌گیرد و مقداری بر می‌گرداند - اما نحوه نگارش آن متفاوت است. برای مثال متد **keys** یک دیکشنری می‌گیرد و لیستی از کلیدهای موجود را برمی‌گرداند، اما به جای نحوه نگارش تابع **keys(eng2sp)** از نحوه نگارش متد **eng2sp.keys()** بهره می‌گیرد:

```
>>> eng2sp.keys()
['one', 'three', 'two']
```

این شکل نمادگذاری نقطه، نام تابع (**keys**) و نام شیئی که از تابع استفاده می‌کند (**eng2sp**) را مشخص می‌کند. پرانتزها نشان می‌دهند که این متد پارامتری نمی‌گیرد. فراخوانی یک متد را **احضار** می‌نامند. در این مورد می‌گوییم که ما متد **keys** را بر روی شیء **eng2sp** احضار کرده‌ایم. متد **values** مشابه متد **keys** است با این تفاوت که لیستی از مقادیر درون دیکشنری را برمی‌گرداند:

```
>>> eng2sp.values()
['uno', 'tres', 'dos']
```

متد **items** هر دو را در قالب لیستی از چندتایی‌ها (که هر کدام یک جفت کلید-مقدار هستند) برمی‌گرداند:

```
>>> eng2sp.items()
[('one', 'uno'), ('three', 'tres'), ('two', 'dos')]
```

نحوه نگارش، اطلاعات مفیدی درباره نوع داده‌ها فراهم می‌کند. جفت براکت‌ها نشان می‌دهد که این یک لیست است و پرانتزها مشخص می‌کند که عناصر لیست، چندتایی هستند. اگر متدی آرگومان بگیرد، از نحوه نگارشی مشابه با فراخوانی توابع استفاده می‌کند. برای مثال، متد **has_key** کلیدی را می‌گیرد و در صورت وجود آن در دیکشنری مقدار **true**، (1) را برمی‌گرداند:

```
>>> eng2sp.has_key('one')
1
>>> eng2sp.has_key('deux')
0
```

اگر سعی کنید متدی را بدون مشخص کردن یک شیء احضار کنید، خطایی دریافت می‌کنید. در این مورد پیغام خطا چندان مفید نیست:

```
>>> has_key('one')
NameError: has_key
```

۱۰-۳- بدل‌سازی و کپی‌برداری

از آنجا که دیکشنری‌ها تغییرپذیرند، لازم است از بدل‌سازی آگاه باشید. هرگاه دو متغیر به شیء واحدی اشاره کنند اعمال تغییر در یکی، در دیگری هم تأثیر می‌گذارد. اگر می‌خواهید یک دیکشنری را تغییر دهید و یک کپی از نسخه اصلی را نگه دارید، از متد **copy** استفاده کنید. برای نمونه، **opposites** یک دیکشنری است که جفت‌های متضاد را نگه می‌دارد:

```
>>> opposites={'up': 'down', 'right': 'wrong', 'true': 'false'}
>>> alias = opposites
>>> copy = opposites.copy()
```

alias و **opposites** به شیء یکسانی اشاره می‌کنند و **copy** به نمونه جدیدی از همان دیکشنری. اگر ما **alias** را تغییر دهیم **opposites** هم تغییر می‌کند:

```
>>> alias['right'] = 'left'
>>> opposites['right']
'left'
```

و اگر **copy** را تغییر دهیم، **opposites** بدون تغییر باقی می‌ماند:

```
>>> copy['right'] = 'privilege'
>>> opposites['right']
'left'
```

۱۰-۴- ماتریس‌های پراکنده

در بخش ۸-۱۴، ما از لیستی از لیست‌ها برای نشان دادن یک ماتریس استفاده کردیم. این برای ماتریسی با مقادیر اکثراً غیرصفر انتخاب خوبی است، اما ماتریس پراکنده‌ای نظیر ماتریس صفحه بعد را مجسم کنید:

$$\begin{pmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \end{pmatrix}$$

در نحوه نمایش به صورت لیست، تعداد زیادی 0 به کار رفته است:

```
matrix = [ [0,0,0,1,0],
            [0,0,0,0,0],
            [0,2,0,0,0],
            [0,0,0,0,0],
            [0,0,0,3,0] ]
```

چاره دیگر، استفاده از یک دیکشنری است، برای کلیدها می‌توانیم از چندتایی‌هایی که شامل تعداد سطر و ستون‌ها هستند، استفاده کنیم. در اینجا نمایش همان ماتریس را در قالب دیکشنری مشاهده می‌کنید:

```
matrix = {(0,3): 1, (2, 1): 2, (4, 3): 3}
```

ما تنها به سه جفت کلید-مقدار نیاز داریم که هر کدام به یکی از عناصر غیرصفر ماتریس اشاره می‌کنند. هر کلید یک چندتایی است و هر مقدار یک عدد صحیح است. جهت دستیابی به عنصری از ماتریس می‌توانیم از عملگر [] استفاده کنیم:

```
matrix[0,3]
1
```

توجه کنید که نحوه نگارش، برای نمایش ماتریس‌ها در قالب دیکشنری با نحوه نگارش برای نشان دادن آنها در قالب لیست‌های تودرتو یکسان نیست. به جای دو اندیس صحیح، ما از یک اندیس که یک چندتایی از اعداد صحیح است، استفاده می‌کنیم. یک مشکل وجود دارد. اگر عضوی را مشخص کنیم که مقدار آن 0 باشد پیغام خطا دریافت می‌کنیم، زیرا هیچ قلم داده‌ای شامل آن کلید در دیکشنری وجود ندارد:

```
>>> matrix[1, 3]
KeyError: (1, 3)
```


متد **get** این مشکل را حل می‌کند:

```
>>> matrix.get((0,3), 0)
1
```

آرگومان اول کلید است و آرگومان دوم مقداری است که **get** در صورت عدم وجود کلید در دیکشنری باید برگرداند:

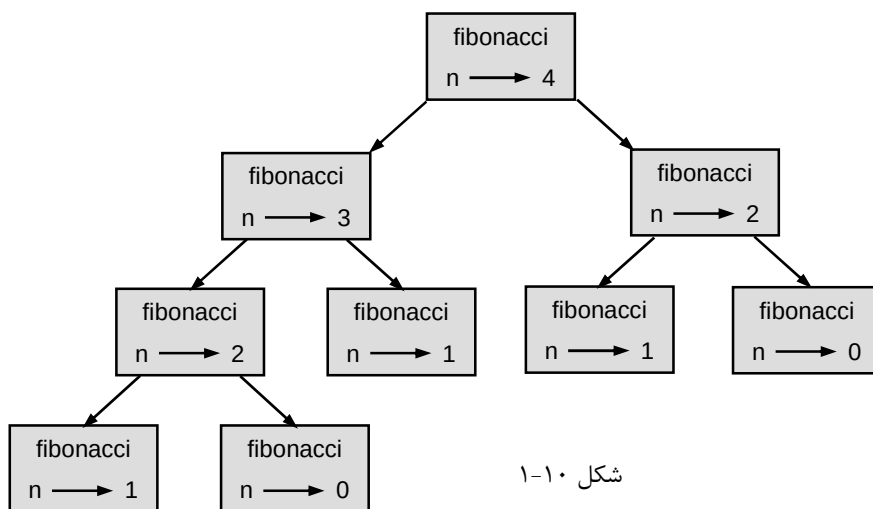
```
>>> matrix.get((1,3), 0)
0
```

متد **get** معنای دسترسی به یک ماتریس پراکنده را کاملاً بهبود می‌بخشد اما درباره نحوه نگارش آن شرمنده‌ایم!

۱-۵- دستاوردها

اگر با تابع **fibonacci** از بخش ۵-۷ کار کرده باشید، احتمالاً متوجه شده‌اید که هر چه آرگومان مقرر بزرگ‌تر باشد زمان بیشتری صرف اجرای تابع می‌شود. به علاوه زمان اجرا خیلی سریع افزایش می‌یابد. بر روی یکی از ماشین‌های ما، **fibonacci(20)** فوراً اجرا می‌شود، **fibonacci(30)** حدود ۱ ثانیه زمان می‌برد و **fibonacci(40)** تقریباً تا ابد به طول می‌انجامد.

برای درک علت، گراف فراخوانی زیر را برای تابع **fibonacci** با $n = 4$ در نظر بگیرید:



شکل ۱-۱۰

یک گراف فراخوانی، مجموعه قاب‌های تابع را با خطوطی که هر قاب را به قاب‌های توابع فراخوانده شده وصل می‌کنند نمایش می‌دهد. در بالای گراف `fibonacci` با `na=a4`، `fibonacci` با `n = 3` و `n = 2` را صدا می‌زند. به همین ترتیب `fibonacci` با `na=a3`، `fibonacci` با `n = 2` و `n = 1` را فرا می‌خواند و ...

تعداد دفعات فراخوانی `fibonacci(0)` و `fibonacci(1)` را بشمارید. این یک راه حل ناکارآمد مسأله است و همین‌طور که آرگومان تابع بزرگ‌تر می‌گردد، وضع بدتر و وخیم‌تر می‌شود. یک راه حل مناسب برای این مسأله حفظ مقادیر از پیش محاسبه شده به‌وسیله مرتب کردن آنها در یک دیکشنری است. به مقداری که قبلاً محاسبه شده و برای استفاده در محاسبات بعدی به کار می‌رود **دست‌آورد** می‌گویند. در اینجا اجرایی از `fibonacci` را با استفاده از دستاوردها می‌بینیم:

```
previous = {0:1, 1:1}

def fibonacci(n):
    if previous.has_key(n):
        return previous[n]
    else:
        newValue = fibonacci(n-1) + fibonacci(n-2)
        previous[n] = newValue
        return newValue
```

دیکشنری `previous` اعداد فیبوناچی را که تاکنون به‌دست آورده‌ایم، ثبت می‌کند. تنها با دو جفت، عملیات را شروع می‌کنیم: `0` به `1` و `1` به `1` نگاشت می‌کند.

هر گاه که `fibonacci` فراخوانی شود، دیکشنری را چک می‌کند تا تعیین کند آیا حاوی نتیجه هست یا نه. اگر نتیجه در دیکشنری موجود بود تابع می‌تواند بدون هیچ‌گونه فراخوانی بازگشتی سریعاً بازگردد، در غیر این صورت باید مقدار جدید را محاسبه نماید. این مقدار جدید قبل از بازگشت تابع به دیکشنری اضافه می‌شود.

با استفاده از این نسخهٔ `fibonacci` ماشین‌های ما `fibonacci(40)` را در یک چشم به هم زدن محاسبه می‌کنند. اما هنگامی که سعی می‌کنیم `fibonacci(50)` را محاسبه کنیم به اشکالی متفاوت بر می‌خوریم:

```
>>> fibonacci(50)
OverflowError: integer addition
```

پاسخی که تا یک دقیقه دیگر خواهید دید **20,365,011,074** می‌باشد. مشکل آنجا است که این عدد برای جا شدن در متغیرهای صحیح پایتون (**integer**) خیلی بزرگ است و سرریز می‌شود. خوشبختانه این مشکل راه حل ساده‌ای دارد.

۱۰-۶- اعداد صحیح بزرگ

پایتون یک نوع داده‌ای به نام **long int** تدارک دیده است که می‌تواند اعداد صحیح در هر اندازه‌ای را به کار برد. دو راه برای ساختن یک مقدار **long int** وجود دارد. یک راه، نوشتن یک **L** در آخر یک عدد صحیح است:

```
>>> type(1L)
<type 'long int'>
```

راه دیگری برای تبدیل یک مقدار به **long int** استفاده از تابع **long** است. **long** می‌تواند هر نوع داده عددی و حتی رشته‌های حاوی ارقام را بپذیرد:

```
>>> long(1)
1L
>>> long(3.9)
3L
>>> long('57')
57L
```

تمام اعمال ریاضی بر روی **long int**‌ها کار می‌کنند، بنابراین مجبور نیستیم کار چندانی برای اصلاح **fibonacci** انجام دهیم:

```
>>> previous = {0:1L, 1:1L}
>>> fibonacci(50)
20365011074L
```

تنها با تغییر مقادیر اولیه **previous**، رفتار **fibonacci** را تغییر می‌دهیم. دو عدد نخست دنباله از نوع **long int** هستند، بنابراین تمام اعداد بعدی دنباله نیز **long int** خواهند بود.

تمرین ۱۰-۱): factorial را طوری تغییر دهید تا اعداد را به صورت **longint** به عنوان نتیجه تولید کند.

۱۰-۷- شمارش حروف

در فصل ۷ تابعی نوشتیم که دفعات رخداد حرفی را در یک رشته می‌شمارد. یک نسخهٔ جامع‌تر این مسئله تشکیل یک **histogram** از تعداد رخدادهای حروف درون رشته می‌باشد. چنین نموداری می‌تواند جهت فشرده‌سازی یک فایل متنی مفید باشد. از آنجا که حروف متفاوت، با دفعات تکرار متفاوت ظاهر می‌شوند، می‌توانیم با استفاده از کدهای کوتاه‌تر برای حروف عادی و کدهای طولانی‌تر برای حروفی که کمتر تکرار شده‌اند یک فایل را فشرده‌سازی کنیم. دیکشنری‌ها راه زیبایی برای تعمیم یک **histogram** تدارک دیده‌اند:

```
>>> letterCounts = {}
>>> for letter in "Mississippi":
...     letterCounts[letter] = letterCounts.get (letter, 0) + 1
...
>>> letterCounts
{'M': 1, 's': 4, 'p': 2, 'i': 4}
```

عملیات را با یک دیکشنری خالی شروع می‌کنیم. برای هر حرف در رشته ما تعداد فعلی (شاید صفر) را یافته و آن را افزایش می‌دهیم. در پایان دیکشنری شامل جفت‌های حروف و تعداد تکرار آنها است.

ممکن است نمایش **histogram** به ترتیب حروف الفبا جذاب‌تر باشد. می‌توانیم این کار را با استفاده از متدهای **sort** و **items** انجام دهیم:

```
>>> letterItems = letterCounts.items()
>>> letterItems.sort()
>>> print letterItems
[('M', 1), ('i', 4), ('p', 2), ('s', 4)]
```

شما متد **items** را قبلاً دیده‌اید، اما **sort** اولین متدی است که می‌بینید بر روی لیست‌ها کار می‌کند. متدهای بسیاری از جمله **append**، **extend** و **reverse** وجود دارد.

۱۰-۸- واژه‌نامه

dictionary (دیکشنری)

مجموعه‌ای از جفت‌های کلید-مقدار که کلیدها را به مقادیر نگاشت می‌کند. کلیدها می‌توانند از هر نوع تغییرناپذیری باشند و مقادیر می‌توانند هر نوعی را به خود اختصاص دهند.

key (کلید)

مقداری که جهت یافتن یک عنصر در دیکشنری استفاده می‌شود.

key-value pair (جفت کلید-مقدار)

یکی از اقلام درون دیکشنری.

method (متد)

نوع دیگری از تابع که با نحوه نگارش متفاوتی فراخوانی شده و بر روی یک شیء احضار می‌شود.

invoke (احضار)

فراخوانی یک متد.

call graph (گراف فراخوانی)

نموداری که در آن فراخوانی یک تابع توسط دیگر توابع به صورت گرافیکی نمایش داده شده است.

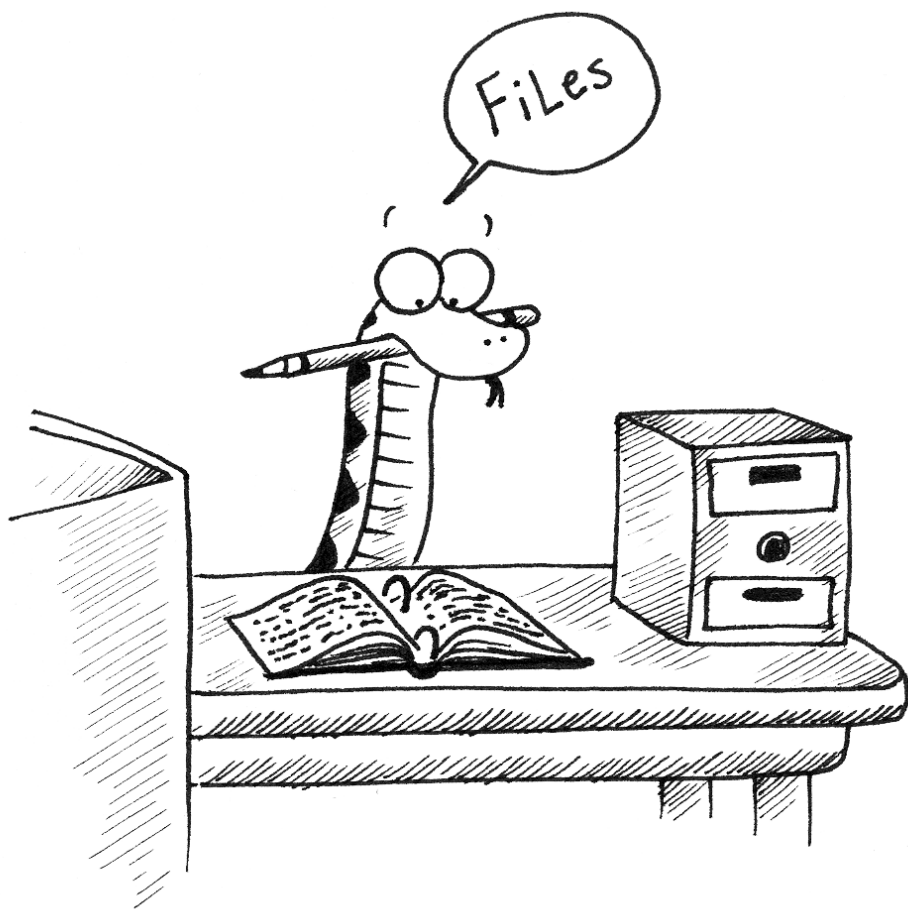
hint (دست‌آورد)

ذخیره‌سازی موقت یک مقدار از پیش محاسبه شده جهت جلوگیری از محاسبه مفرط.

overflow (سرریز)

یک نتیجه عددی که برای نمایش در یک قالب عددی خیلی بزرگ است.

فایل‌ها و اعتراض‌ها



مادامی که یک برنامه در حال اجرا است، داده‌هایش در حافظه قرار دارد. وقتی برنامه خاتمه یافت و یا کامپیوتر خاموش شد، داده‌ها در حافظه ناپدید می‌شوند. برای نگهداری داده‌ها به صورت دائمی باید آنها را در یک فایل بگذارید. فایل‌ها معمولاً بر روی دیسک سخت، دیسک لرزان و یا CD ذخیره می‌شوند.

وقتی تعداد زیادی فایل وجود دارد، آنها در داخل **دایرکتوری‌ها** (پوشه‌ها) سازمان‌دهی می‌شوند. هر فایل به وسیله یک نام منحصر به فرد و یا ترکیبی از نام فایل و نام دایرکتوری مشخص می‌شود.

با خواندن و نوشتن فایل‌ها، برنامه‌ها می‌توانند اطلاعات را با هم مبادله کنند و قالب‌های قابل چاپ، مانند PDF، تولید کنند.

کار کردن با فایل‌ها بسیار شبیه به کار با کتاب‌ها است. برای استفاده از یک کتاب شما باید آن را باز کنید. وقتی کارتان با کتاب تمام شد باید آن را ببندید. تا وقتی که کتاب باز است می‌توانید در آن بنویسید و یا از مطالبش بخوانید. در هر وضعیت، شما می‌دانید که در کجای کتاب هستید. اغلب اوقات، شما سراسر کتاب را با ترتیب طبیعی آن می‌خوانید، اما می‌توانید در بخش‌های مختلف هم جست‌وجو کنید.

همه این موارد در مورد فایل‌ها هم صادق است. برای باز کردن فایل، شما اسم آن را مشخص می‌کنید و تعیین می‌نمایید که آیا می‌خواهید بر روی آن بنویسید و یا چیزی را از آن بخوانید. باز کردن یک فایل، یک شیء فایل می‌سازد. در این مثال ما متغیر **f** را به شیء فایل جدید ارجاع می‌دهیم:

```
>>> f = open("test.dat", "w")
>>> print f
<open file 'test.dat', mode 'w' at fe820>
```

تابع **open** دو آرگومان می‌گیرد. اولی نام فایل (یا مسیر آن فایل همراه با نامش) و دومی سبک کار با فایل است. سبک **w** یعنی ما فایل را برای نوشتن (**write**) باز می‌کنیم. اگر فایلی به نام **test.dat** وجود نداشته باشد، ساخته خواهد شد و اگر از قبل وجود داشته باشد با فایلی که ما می‌نویسیم جایگزین می‌شود. وقتی ما شیء فایل را چاپ می‌کنیم نام فایل، سبک و محل شیء را (در حافظه مفسر) ملاحظه می‌کنیم.

برای گذاشتن داده‌ها در فایل ما متد **write** را بر روی شیء فایل احضار می‌کنیم:

```
>>> f.write("Now is the time")
>>> f.write("to close the file")
```


بستن فایل به سیستم می‌گوید که ما عمل نوشتن را انجام دادیم و فایل را برای خواندن قابل دستیابی می‌سازد:

```
>>> f.close()
```

حال می‌توانیم فایل را دوباره باز کنیم، اما این بار برای خواندن. حال آرگومان سبک، "r"، برای خواندن (read) است و از این طریق می‌توانیم محتویات فایل را درون یک رشته بخوانیم:

```
>>> f = open("test.dat", "r")
```

اگر بخوایم فایلی را که وجود ندارد باز کنیم، پیغام خطا دریافت خواهیم کرد:

```
>>> f = open("test.cat", "r")
IOError: [Errno 2] No such file or directory: 'test.cat'
```

متد **read**، داده‌ها را از فایل می‌خواند. اگر هیچ آرگومانی به این متد ندهیم، تمام محتویات فایل را می‌خواند:

```
>>> text = f.read()
>>> print text
Now is the timeto close the file
>>> f.close()
```

هیچ فضایی بین **time** و **to** وجود ندارد، زیرا ما هیچ فضایی بین آنها وارد نکردیم. **read** همچنین یک آرگومان می‌گیرد که مشخص می‌کند چه مقدار از کاراکترها خوانده شود:

```
>>> f = open("test.dat", "r")
>>> print f.read(5)
Now i
```

اگر کاراکترهای کافی در سمت چپ فایل وجود نداشته باشد، **read** کاراکترهای باقیمانده را باز می‌گرداند و هنگامی که به آخر فایل می‌رسیم، **read** رشته تهی برمی‌گرداند:

```
>>> print f.read(1000006)
s the timeto close the file
>>> print f.read()
>>>
```

تابع زیر یک فایل را کپی می‌کند، این تابع در هر بار 50 کاراکتر را می‌خواند و می‌نویسد. اولین آرگومان نام **file** اصلی و دومین آرگومان نام فایل جدید است:

```
def copyFile(oldFile, newFile):  
    f1 = open(oldFile, "r")  
    f2 = open(newFile, "w")  
    while 1:  
        text = f1.read(50)  
        if text == "":  
            break  
        f2.write(text)  
    f1.close()  
    f2.close()  
    return
```

دستور break جدید است. اجرای آن باعث خروج از حلقه شده و روند اجرا از اولین دستور بعد از حلقه از سر گرفته می‌شود.
در این مثال حلقه **while** بی‌انتهاست، زیرا مقدار 1 همیشه **true** است. تنها راه خروج از حلقه اجرای دستور **break** است که وقتی **text** برابر با یک رشته تهی شد اتفاق می‌افتد؛ یعنی زمانی که به پایان فایل رسیدیم.

۱۱-۱- فایل‌های متنی

یک فایل متنی، فایلی است شامل کاراکترهای قابل چاپ و فضای خالی؛ این کاراکترها در خطوط جدا شده توسط کاراکترهای خط جدید سازمان‌دهی شده‌اند. از آنجا که پایتون خصوصاً برای پردازش فایل‌های متنی طراحی شده، متدهایی برای ساده سازی انجام این کار فراهم ساخته است. برای اثبات این موضوع، ما یک فایل متنی با سه خط می‌سازیم که خطوط آن توسط کاراکتر خط جدید از هم جدا شده‌اند:

```
>>> f = open("test.dat", "w")  
>>> f.write("line one\nline two\nline three\n")  
>>> f.close()
```

متد **readline** تمام کاراکترها را تا رسیدن به یک کاراکتر خط جدید می‌خواند و خود کاراکتر خط جدید را هم شامل می‌شود:

```
>>> f = open("test.dat", "r")
>>> print f.readline()
line one
>>>
```

متد **readlines** تمام خطوط باقیمانده را به عنوان لیستی از رشته‌ها باز می‌گرداند:

```
>>> print f.readlines()
['line two\012', 'line three\012']
```

در این مورد خروجی در قالب لیست می‌باشد، یعنی رشته‌ها با علامت کوتیشن و کاراکترها به وسیله کاراکتر کنترلی **012** نمایش داده می‌شود.

در پایان فایل، **readline** یک رشته تهی و **readlines** یک لیست تهی را باز می‌گرداند:

```
>>> print f.readline()
>>> print f.readlines()
[]
```

در زیر مثالی از یک برنامه پردازش خطی آمده است. **filterFile** یک کپی از فایل اصلی می‌گیرد و خطوطی که با **#** مشخص شده‌اند را حذف می‌کند:

```
def filterFile(oldFile, newFile):
    f1 = open(oldFile, "r")
    f2 = open(newFile, "w")
    while 1:
        text = f1.readline()
        if text == "":
            break
        if text[0] == '#':
            continue
        f2.write(text)
    f1.close()
    f2.close()
    return
```

دستور continue تکرار جاری حلقه را خاتمه می‌دهد، اما اجرای حلقه ادامه می‌یابد. روند اجرا به ابتدای حلقه می‌رود و تنها در صورتی که هر دو شرط مردود شود، ما **text** را در یک فایل جدید کپی می‌کنیم.

۱۱-۲- نوشتن متغیرها

آرگومان متد **write** باید یک رشته باشد، بنابراین اگر بخواهیم مقادیر دیگری در فایل بگذاریم باید اول آنها را به رشته تبدیل کنیم. ساده‌ترین راه برای این کار استفاده از تابع **str** است:

```
>>> x = 52
>>> f.write (str(x))
```

راه دیگر، استفاده از **عملگر قالب‌بندی** % است. وقتی عملگر % را با اعداد صحیح به کار می‌بریم، عملگر باقیمانده است، اما وقتی اولین عملوند یک رشته باشد، یک عملگر قالب‌بندی است. اولین عملوند، یک **رشته قالب‌بندی** است و دومین عملوند یک چندتایی از عبارات است. نتیجه، رشته‌ای شامل مقادیر عبارات با قالبی مطابق با رشته قالب‌بندی است. به عنوان یک مثال ساده، دنباله قالب‌بندی **"%d"** یعنی اولین عبارت در چندتایی باید به عنوان یک عدد صحیح قالب‌بندی شود. در اینجا **d** نمادی برای نشان دادن **"decimal"** است:

```
>>> cars = 52
>>> "%d" % cars
'52'
```

نتیجه، رشته **'52'** است که نباید با مقدار صحیح **52** اشتباه گرفته شود. یک دنباله قالب‌بندی می‌تواند در هر جای رشته مورد نظر قرار گیرد. بنابراین، ما می‌توانیم یک مقدار را در یک رشته جاسازی کنیم:

```
>>> cars = 52
>>> "In July we sold %d cars." % cars
'In July we sold 52 cars.'
```

دنباله قالب‌بندی **"%f"**، بخش بعدی را در چندتایی به عنوان یک عدد اعشاری و دنباله قالب‌بندی **"%s"**، بخش بعدی را در چندتایی به عنوان یک رشته، قالب‌بندی می‌کند:

```
>>> "In %d days we made %f million %s." % (34,6.1,'dollars')
'In 34 days we made 6.100000 million dollars.'
```

بنابراین قرارداد، قالب‌بندی اعداد اعشاری با شش رقم اعشار چاپ می‌شود. تعداد عباراتی که در چندتایی وجود دارد باید با تعداد دنباله‌های قالب‌بندی که در رشته وجود دارد، مطابق باشد:

```
>>> "%d %d %d" % (1,2)
TypeError: not enough arguments for format string
>>> "%d" % 'dollars'
TypeError: illegal argument type for built-in operation
```

در مثال اول، عبارات کافی وجود نداشت و در مثال دوم نوع عبارت غلط است. برای کنترل بیشتر بر روی قالب‌بندی اعداد، ما می‌توانیم تعداد ارقام را به‌عنوان قسمتی از دنباله قالب‌بندی مشخص کنیم:

```
>>> "%6d" % 62
'   62'
>>> "%12f" % 6.1
' 6.100000'
```

شماره بعد از علامت درصد حداقل تعداد مکان‌هایی است که عدد، خواهد گرفت. اگر مقدار تولید شده رقم‌های کمتری جا گرفت، فضای خالی قبل از عدد اضافه می‌شوند و اگر مکان‌ها منفی بود فضای خالی بعد از عدد اضافه می‌شوند:

```
>>> "%-6d" % 62
'62   '
```

برای اعداد اعشاری ما می‌توانیم تعداد ارقام بعد از ممیز را هم مشخص کنیم:

```
>>> "%12.2f" % 6.1
' 6.10'
```

در این مثال نتیجه، دوازده فضای خالی می‌گیرد و دو رقم بعد از ممیز را شامل می‌شود. این قالب‌بندی برای چاپ مبالغ دلار با ممیزهای هم‌تراز مفید است. برای مثال یک دیکشنری شامل نام دانشجویان به‌عنوان کلید و ساعت کار آنها به‌عنوان مقدار در نظر بگیرید. در اینجا تابعی که محتویات آن دیکشنری را به‌طور گزارشات قالب‌بندی شده چاپ می‌کند می‌بینید:

```
def report (wages):
    students = wages.keys()
    students.sort()
    for student in students:
        print "%-20s %12.02f" % (student, wages[student])
```

برای آزمایش این تابع ما یک دیکشنری کوچک می‌سازیم و محتویات آن را چاپ می‌کنیم:

```
>>> wages = {'mary': 6.23, 'joe': 5.45, 'joshua': 4.25}
>>> report (wages)
joe                5.45
joshua             4.25
mary               6.23
```

تا زمانی که نام‌ها کمتر از ۲۱ کاراکتر و دستمزدها کمتر از یک میلیارد دلار در ساعت هستند با کنترل عرض هر مقدار تضمین می‌کنیم که ستون‌ها در یک ردیف قرار می‌گیرند.

۱۱-۳- دایرکتوری‌ها

وقتی که شما فایلی را به‌وسیله بازکردن و نوشتن بر روی آن می‌سازید، فایل جدید به دایرکتوری جاری می‌رود (هر جا که شما برنامه را اجرا کرده‌اید). به‌طور مشابه وقتی فایلی را برای خواندن باز می‌کنید، پایتون در دایرکتوری جاری به دنبال آن می‌گردد. اگر می‌خواهید فایلی را در جای دیگری باز کنید، باید مسیری برای فایل مشخص کنید که این مسیر دایرکتوری (پوشه‌ای) است که فایل در آن قرار گرفته است:

```
>>> f = open("C:\\programming\\Python\\Tests\\test1.dat", "r")
>>> print f.readline()
This is a test.
```

در درایو **C** پوشه‌ای به نام **programming** که در آن پوشه‌ای به نام **Python**، که در آن پوشه‌ای به نام **Tests** وجود دارد که شامل فایلی به نام **test1.dat** است. سطر اول این فایل جمله **This is a test.** می‌باشد که ما توسط متد **readline()** آن را خواندیم.

همان‌طور که می‌بینید، در آخر هر پوشه از دو علامت **"\"** استفاده شده است، زیرا این علامت یک کاراکتر کنترلی است و برای قرار دادن آن در رشته باید از دو علامت **"\"** استفاده کنیم. شما نمی‌توانید کاراکتر **"\"** را برای نام یک فایل استفاده کنید، زیرا این کاراکتر برای جداکردن اسامی فایل‌ها و دایرکتوری‌ها در نظر گرفته شده است.

۱۱-۴- Pickling

به‌منظور گذاشتن مقادیر در یک فایل، باید آنها را به رشته تبدیل کنید. قبلاً دیده‌اید که چگونه می‌توان با تابع **str** کار کرد:

```
>>> f.write(str(12.3))
>>> f.write(str([1,2,3]))
```

مشکل اینجا است که وقتی شما مقدار را در آخر می‌خوانید، یک رشته به‌دست می‌آورید. اطلاعات اصلی نوع داده از بین رفته‌اند. در حقیقت شما نمی‌توانید حتی بگویید که یک مقدار در کجا تمام شده و مقدار بعدی از کجا آغاز گشته است:

```
>>> f.readline()
'12.3[1, 2, 3]'
```

راه حل نگهداری اطلاعات اصلی نوع داده، استفاده از روش **Pickling** است. **Pickling** به معنای در نمک نگه‌داشتن است و دلیل نام‌گذاری به چنین کلمه‌ای، این است که ساختار داده‌ها در این روش حفظ می‌شود. ماژول **pickle** فرمان‌های لازم را نگهداری می‌کند. برای استفاده از آن **pickle** را وارد محیط کاری کنید و سپس فایل را به طریقه معمول باز کنید:

```
>>> import pickle
>>> f = open("test.pck", "w")
```

برای ذخیره‌سازی ساختار داده‌ها از متد **dump** استفاده کنید و سپس فایل را به روش معمول ببندید:

```
>>> pickle.dump(12.3, f)
>>> pickle.dump([1,2,3], f)
>>> f.close()
```

سپس ما می‌توانیم فایل را برای خواندن باز کرده و ساختار داده‌هایی که موقتاً ذخیره کرده‌ایم را بار کنیم:

```
>>> f = open("test.pck", "r")
>>> x = pickle.load(f)
>>> x
12.3
>>> type(x)
<type 'float'>
>>> y = pickle.load(f)
>>> y
[1, 2, 3]
>>> type(y)
<type 'list'>
```

هر بار که ما **load** را احضار می‌کنیم، یک مقدار واحد را از فایل دریافت می‌کنیم که با نوع اصلی‌اش همراه می‌باشد.

۱۱-۵- اعتراض

هر زمان که یک خطای زمان اجرا اتفاق می‌افتد، مفسر پایتون یک **اعتراض** را تولید می‌کند. در این حالت معمولاً برنامه متوقف می‌شود و پایتون پیغام خطایی را چاپ می‌کند. برای مثال تقسیم بر صفر یک اعتراض می‌سازد:

```
>>> print 55/0
ZeroDivisionError: integer division or modulo
```

همچنین دستیابی به عنصری از لیست که وجود ندارد:

```
>>> a = []
>>> print a[5]
IndexError: list index out of range
```

یا دستیابی به کلیدی که در دیکشنری نیست:

```
>>> b = {}
>>> print b['what']
KeyError: what
```

در هر مورد، پیغام خطا دارای دو بخش است: نوع خطا که قبل از علامت کولن قرار دارد و جزئیات خطا که بعد از علامت کولن قرار دارد. معمولاً پایتون یک پس‌یابی از جایی که برنامه در آن محل مشکل داشته است را هم چاپ می‌کند، اما ما آنها را در مثال‌ها حذف کرده‌ایم.

گاهی اوقات ما می‌خواهیم عملیاتی را که می‌تواند به یک اعتراض منجر شود، اجرا کنیم، اما نمی‌خواهیم برنامه متوقف شود. ما می‌توانیم به‌وسیلهٔ دستورات **try** و **except** از اعتراض پیش‌گیری کنیم.

برای مثال ممکن است ما به کاربر اعلان کنیم که نام فایلی را وارد کند و سپس سعی کنیم که آن فایل را باز کنیم. اگر فایل وجود نداشت، ما نمی‌خواهیم که برنامه خراب شود بلکه می‌خواهیم از اعتراض پیش‌گیری کنیم:

```
filename = raw_input('Enter a file name: ')
try:
    f = open (filename, "r")
except:
    print 'There is no file named', filename
```


دستور **try**، فرامین بلوک اول را اجرا می‌کند. اگر هیچ خطایی رخ ندهد، از دستور **except** چشم‌پوشی می‌کند و اگر هر گونه خطایی رخ دهد، شاخه دستور **except** اجرا می‌شود و ادامه می‌یابد.

ما می‌خواهیم این قابلیت را در یک تابع بسته‌بندی کنیم: تابع **exists** نام فایل را می‌گیرد و در صورتی که وجود داشت **true** و در غیر این صورت **false** را برمی‌گرداند:

```
def exists(filename):
    try:
        f = open(filename)
        f.close()
        return 1
    except:
        return 0
```

شما می‌توانید از چند بلوک **except** برای کنترل انواع مختلف اعتراض استفاده کنید. برای جزئیات بیشتر می‌توانید به کتاب مرجع راهنمای پایتون یا پیوست ب مراجعه کنید. اگر برنامه شما وضعیت خطا را مشخص می‌کند، می‌توانید آن را مولد اعتراض بسازید. در اینجا مثالی عنوان شده که ورودی را از کاربر می‌گیرد و آن را با مقدار 17 چک می‌کند. به فرض اینکه 17 به هر دلیل یک ورودی نامعتبر باشد، اعتراضی را (از طرف برنامه) ارائه می‌دهیم:

```
def inputNumber ():
    x = input ('Pick a number: ')
    if x == 17:
        raise 'BadNumberError', '17 is a bad number'
    return x
```

دستور **raise** دو آرگومان دریافت می‌کند: نوع اعتراض و اطلاعات ویژه درباره خطا. **BadNumberError** یک نوع جدید اعتراض است که برای این کاربرد ساخته‌ایم. اگر تابعی که **inputNumber** را فرا خوانده، از بروز خطا پیش‌گیری کند، برنامه می‌تواند ادامه یابد و در غیر این صورت پایتون پیغام خطایی را گزارش می‌کند و خارج می‌شود:

```
>>> inputNumber ()
Pick a number: 17
BadNumberError: 17 is a bad number
```

پیغام خطا شامل نوع اعتراض و اطلاعات اضافه‌ای است که شما تدارک دیده‌اید.

تمرین ۱۱-۱: تابعی بنویسید که از `inputNumber` برای وارد کردن عددی از صفحه کلید استفاده کند و سپس از اعتراض `BadNumberError` پیش‌گیری کند.

۱۱-۶- واژه‌نامه

file (فایل)

یک واحد داده‌دارای نام که معمولاً بر روی دیسک سخت، دیسک لِرزان و یا **CD-ROM** ذخیره می‌شود و شامل جریانی از کاراکترها است.

directory (دایرکتوری)

مجموعه‌ای دارای نامی از فایل‌ها که پوشه (**folder**) هم نامیده می‌شود.

break statement

دستوری که باعث می‌شود روند اجرا از درون حلقه خارج شود.

text file (فایل متنی)

فایلی شامل کاراکترهای قابل چاپ، سازمان‌یافته در خطوطی که با کاراکترهای خط جدید از هم جدا شده‌اند.

continue statement

دستوری که باعث می‌شود تکرار جاری حلقه پایان یابد. روند اجرا به بالای حلقه می‌رود، شرط را ارزیابی می‌کند و بر اساس آن ادامه می‌یابد.

format operator (عملگر قالب‌بندی)

عملگر % که رشته قالب‌بندی و یک چندتایی از عبارات را می‌گیرد و رشته‌ای شامل عبارت قالب‌بندی شده براساس رشته قالب‌بندی را برمی‌گرداند.

format string (رشته قالب‌بندی)

رشته‌ای شامل کاراکترهای قابل چاپ و دنباله قالب‌بندی که چگونگی قالب‌بندی مقادیر را مشخص می‌کند.

format sequence (دنباله قالب‌بندی)

دنباله‌ای از کاراکترها که با علامت % شروع می‌شوند و چگونگی قالب‌بندی یک مقدار را مشخص می‌کنند.

path (مسیر)

دنباله‌ای از اسامی دایرکتوری‌ها که محل دقیق یک فایل را مشخص می‌کنند.

pickling

نوشتن مقدار یک داده در یک فایل همراه با اطلاعات نوع آن به طوری که می‌تواند بعداً دوباره بازیابی شود.

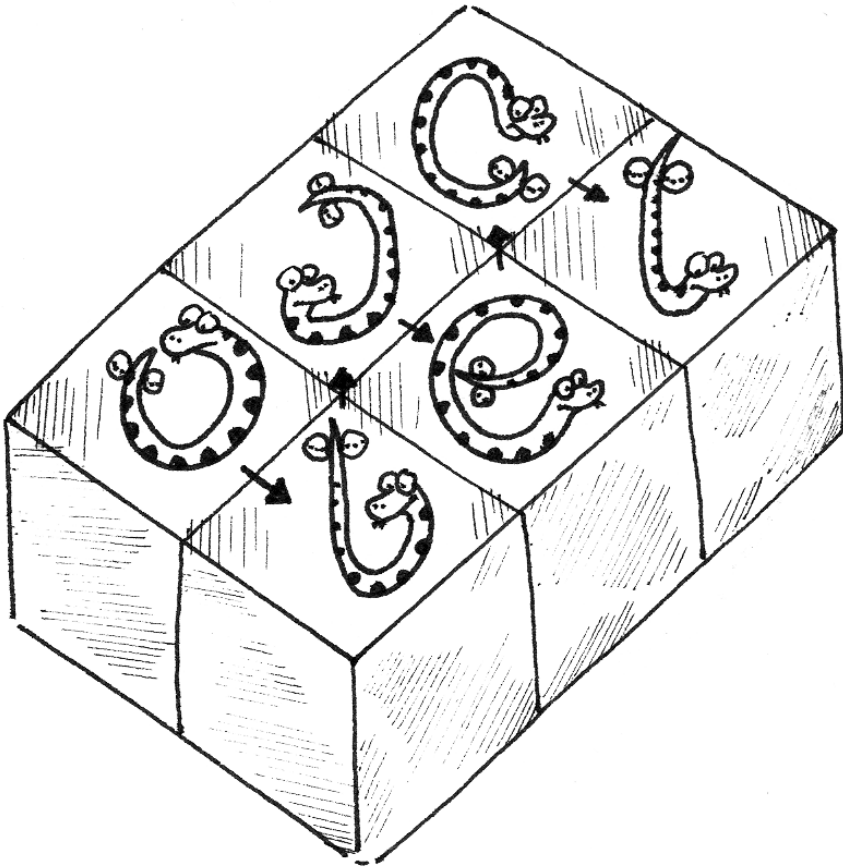
exception (اعتراض)

خطایی که در زمان اجرا رخ می‌دهد.

handle (پیشگیری)

جلوگیری از اعتراضی که به برنامه پایان می‌دهد، با استفاده از دستورات **try** و **except**.

کلاس و اشیاء



در یازده فصلی که مطالعه کردید با انواع داده‌ای متنوعی آشنا شدید. اعداد صحیح و اعشاری، رشته‌ها، لیست‌ها، چندتایی‌ها و دیکشنری‌ها از جمله انواع پیش‌ساخته‌ای بودند که هر یک بنابه سرعت و قابلیت‌هایشان مورد استفاده قرار می‌گیرند.

در بسیاری از برنامه‌ها، به توانایی ساختن انواع داده‌ای کاربر-تعریف نیاز است. در این فصل شما انجام این کار را خواهید آموخت و همچنین برای یادگیری روش دیگری از برنامه‌نویسی آماده می‌شوید.

۱۲-۱ انواع ترکیبی کاربر-تعریف

بعضی از انواع پیش‌ساخته پایتون را به کار بردیم. حال ما آماده‌ایم که یک نوع کاربر-تعریف را بسازیم: نقطه.

به مفهوم ریاضی نقطه فکر کنید. در فضای دو بعدی، یک نقطه عبارت است از دو عدد (مختصات) است که مجتمعاً به عنوان یک شیء واحد تلقی می‌شود. در نمادگذاری ریاضی، نقطه‌ها اغلب در پرانتز نوشته می‌شوند که یک علامت کاما مختصات را از هم جدا می‌کند. برای نمونه $(0a, 0)$ مبدأ را نشان می‌دهد.

یک راه طبیعی برای نمایش دادن نقطه در پایتون، نشان دادن آن توسط دو مقدار اعشاری است. سؤال این است که چگونه این دو مقدار در یک شیء مرکب گروه‌بندی می‌شوند. یک راه حل سریع و بدترکیب، استفاده از لیست یا چندتایی است که ممکن است برای بعضی از کاربردها بهترین انتخاب باشد.

یک راه دیگر تعریف یک نوع ترکیبی توسط کاربر است که **کلاس** نامیده می‌شود. این قابلیت یک مقدار کوشش بیشتری نیاز دارد، اما نتایج آن به زودی آشکار خواهد شد. تعریف یک کلاس به صورت زیر است:

class Point:
pass

تعاریف کلاس می‌توانند در هر جای برنامه ظاهر شوند اما معمولاً در آغاز برنامه هستند (بعد از دستور `import`). قوانین نگارشی برای تعریف یک کلاس شبیه به دیگر دستورات ترکیبی می‌باشند (به بخش ۴-۴ مراجعه کنید).

این تعریف، یک کلاس جدید به نام **Point** می‌سازد. دستور **pass** هیچ تأثیری ندارد. این دستور تنها به این خاطر لازم است که یک دستور ترکیبی، باید در بدنه‌اش چیزی وجود داشته باشد.

با ساختن کلاس **Point** ما یک نوع جدید ساخته‌ایم که آن هم **Point** نام دارد. اعضای این نوع **وهله‌های** نوع یا اشیاء نام دارند. ساختن یک **وهله** جدید را **وهله‌سازی** می‌نامند. برای **وهله‌سازی** شیء **Point** ما تابعی به نام **Point** را فراخوانی می‌کنیم:

```
blank = Point()
```

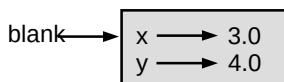
متغیر **blank** به یک شیء جدید **Point** آدرسی را اختصاص می‌دهد. تابعی شبیه **Point** که یک شیء جدید می‌سازد، **سازنده** نامیده می‌شود.

۱۲-۲- مشخصه‌ها

ما می‌توانیم با استفاده از نمادگذاری نقطه‌ای داده جدیدی را به یک **وهله** اضافه کنیم:

```
>>> blank.x = 3.0
>>> blank.y = 4.0
```

این نحوه نگارش شبیه به انتخاب یک متغیر از ماژول است، مثل **math.pi** و **string.uppercase**. در این مثال ما یک قلم داده را از یک **وهله** انتخاب کرده‌ایم. به این قلم داده‌های دارای نام، **مشخصه** گفته می‌شود. نمودار حالت زیر نتیجه این انتساب‌ها را نشان می‌دهد:



شکل ۱۲-۱

متغیر **blank** به شیء **Point** اشاره می‌کند که شامل دو مشخصه است. هر مشخصه به یک عدد اعشاری اشاره می‌کند. ما می‌توانیم یک مشخصه را با استفاده از نحوه نگارش مشابهی بخوانیم:

```
>>> print blank.y
4.0
>>> x = blank.x
>>> print x
3.0
```

عبارت `blank.x` یعنی «به شیء `blank` برو و مقدار `x` را بگیر.» در این مثال، ما مقدار بدست آمده را به متغیر `x` نسبت می‌دهیم. هیچ برخوردی بین `x` و مشخصه `x` به وجود نمی‌آید. هدف نمادگذاری نقطه مشخص کردن متغیری است که شما دقیقاً به آن اشاره کرده‌اید. می‌توانید از نمادگذاری نقطه به عنوان بخشی از هر عبارت استفاده کنید، بنابراین این دستورات قانونی هستند:

```
print '(' + str(blank.x) + ', ' + str(blank.y) + ')\n'
distanceSquared = blank.x * blank.x + blank.y * blank.y
```

اولین خط `(3.0, 4.0)` را چاپ می‌کند و دومین خط مجموع مربعات مختصات نقطه یعنی `25.0` را محاسبه می‌نماید. شاید وسوسه شوید مقدار خود `blank` را هم چاپ کنید که در این صورت خواهید داشت:

```
>>> print blank
<__main__.Point instance at 80f8e70>
```

نتیجه مشخص می‌کند `blank` وهله‌ای از کلاس `Point` است که در `__main__` تعریف شده است. `80f8e70` شناسه منحصره‌فردی برای این شیء است که در هگزادسیمال (مبنای ۱۶) نوشته شده است. شاید این راه چندان آموزنده‌ای برای نمایش شیء `Point` نباشد اما به زودی نحوه تغییر آن را خواهید دید.

تمرین ۱۲-۱: یک شیء `Point` بسازید و چاپ کنید و سپس از تابع `id` برای مشخص کردن شناسه منحصره‌فرد شیء استفاده کنید.

۱۲-۳- وهله‌ها به عنوان پارامترها

شما می‌توانید یک وهله را با روش معمول به‌عنوان پارامتر به تابع بفرستید. برای مثال:

```
def printPoint(p):
    print '(' + str(p.x) + ', ' + str(p.y) + ')\n'
```

تابع `printPoint`، نقطه‌ای را به عنوان آرگومان می‌گیرد و آن را در قالب استاندارد نمایش می‌دهد. اگر `printPoint(blank)` را فراخوانی کنیم، خروجی `(3.0, 4.0)` است.

تمرین ۱۲-۲: تابع **distance** از بخش ۵-۲ را طوری بازنویسی کنید که به جای چهار عدد، دو نقطه را به عنوان آرگومان بگیرد.

۱۲-۴- تشابه و وحدت

اگر دو نقطه با مختصات یکسان وجود داشته باشد، این دو نقطه می‌توانند یک شیء واحد باشند یا دو شیء مختلف با مختصات مشابه باشند. برای فهمیدن اینکه آیا دو آدرس به یک شیء واحد اشاره می‌کنند یا نه از عملگر `==` استفاده کنید. برای نمونه:

```
>>> p1 = Point()
>>> p1.x = 3
>>> p1.y = 4
>>> p2 = Point()
>>> p2.x = 3
>>> p2.y = 4
>>> p1 == p2
0
```

هر چند **p1** و **p2** دارای مختصات مشابهی هستند، اما شیء واحدی نیستند. اگر ما **p1** را به **p2** انتساب دهیم، آنگاه دو متغیر بدل‌هایی از یک شیء واحدند:

```
>>> p2 = p1
>>> p1 == p2
1
```

این نوع برابری را **مساوات سطحی** می‌نامند، زیرا تنها آدرس‌ها مقایسه شده‌اند، نه مضمون اشیاء.

برای مقایسهٔ مضمون اشیاء - **مساوات عمقی** - ما می‌توانیم تابعی به نام **samePoint** بنویسیم:

```
def samePoint(p1, p2):
    return (p1.x == p2.x) and (p1.y == p2.y)
```

حال اگر ما دو شیء مختلف بسازیم که شامل داده‌های مشابه هستند می‌توانیم از **samePoint** برای فهمیدن اینکه آیا آنها نقطهٔ مشابهی را نشان می‌دهند یا نه، استفاده کنیم:

```
>>> p1 = Point()
>>> p1.x = 3
>>> p1.y = 4
>>> p2 = Point()
```

```
>>> p2.x = 3
>>> p2.y = 4
>>> samePoint(p1, p2)
1
```

البته اگر دو متغیر به شیء واحدی اشاره کنند، آنها هر دو مساوات سطحی و عمقی را دارا هستند.

۱۲-۵- مستطیل

در اینجا ما به کلاسی برای نمایش مستطیل نیاز داریم. سؤال این است که چه اطلاعاتی را مجبوریم برای مشخص کردن یک مستطیل فراهم کنیم. برای ساده کردن کار فرض کنید که مستطیل تنها در جهت عمودی یا افقی است و زاویه‌دار نمی‌باشد. چند امکان وجود دارد: ما می‌توانستیم وسط مستطیل (دو مختصات) و اندازه (طول و عرض) آن را مشخص کنیم یا می‌توانستیم یکی از رأس‌ها و اندازه را تعیین کنیم. همچنین می‌توانستیم دو رأس مقابل هم را مشخص کنیم. یک انتخاب قراردادی هم مشخص کردن رأس سمت چپ-بالای مستطیل و اندازه است. پس ما یک کلاس جدید تعریف می‌کنیم:

```
class Rectangle:
    pass
```

و آن را وهله‌سازی می‌نماییم:

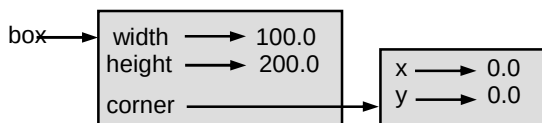
```
box = Rectangle()
box.width = 100.0
box.height = 200.0
```

این کد یک شیء **Rectangle** جدید با دو مشخصهٔ اعشاری ایجاد می‌کند. برای مشخص کردن رأس سمت چپ-بالا می‌توانیم یک شیء را درون یک شیء جاسازی کنیم!

```
box.corner = Point()
box.corner.x = 0.0
box.corner.y = 0.0
```

عملگر نقطه، عمل جاسازی را انجام می‌دهد. عبارت **box.corner.x** یعنی «به شیء **box** برو و به مشخصه‌ای به نام **corner** اشاره کن و سپس به شیء **corner** برو و مشخصه‌ای به نام **x** را انتخاب کن.»

شکل زیر وضعیت این شیء را نشان می‌دهد:



شکل ۱۲-۲

۱۲-۶- وهله‌ها به عنوان مقادیر برگشتی

توابع می‌توانند وهله‌ها را بازگردانند. برای نمونه، `findCenter` یک `Rectangle` را به عنوان آرگومان می‌گیرد و یک `Point` را که شامل مختصات مرکز آن `Rectangle` است برمی‌گرداند:

```
def findCenter(box):
    p = Point()
    p.x = box.corner.x + box.width/2.0
    p.y = box.corner.y + box.height/2.0
    return p
```

برای فراخوانی این تابع، `box` را به عنوان آرگومانی به آن بفرستید و نتیجه را به متغیری نسبت دهید:

```
>>> center = findCenter(box)
>>> printPoint(center)
(50.0, 100.0)
```

۱۲-۷- اشیاء تغییرپذیرند

ما می‌توانیم حالت یک شیء را به وسیلهٔ انتساب به یکی از مشخصه‌های آن تغییر دهیم. برای نمونه، جهت تغییر اندازهٔ یک مستطیل بدون تغییر مکان آن می‌توانیم مقادیر `width` و `height` را عوض کنیم:

```
box.width = box.width + 50
box.height = box.height + 100
```

می‌توانیم این کد را در یک متد بسته‌بندی کنیم و آن را برای تغییر اندازه مستطیل به وسیله هر مقدار، تعمیم دهیم:

```
def growRect(box, dwidth, dheight):
    box.width = box.width + dwidth
    box.height = box.height + dheight
```

متغیرهای **dwidth** و **dheight** نشان می‌دهند که مستطیل در هر جهت چقدر باید تغییر اندازه دهد. نتیجه احضار این متد تغییر **Rectangle** است که به عنوان آرگومانی به آن فرستاده شده است.

برای مثال می‌توانیم یک **Rectangle** جدید به نام **bob** بسازیم و آن را به **growRect** بفرستیم:

```
>>> bob = Rectangle()
>>> bob.width = 100.0
>>> bob.height = 200.0
>>> bob.corner = Point()
>>> bob.corner.x = 0.0
>>> bob.corner.y = 0.0
>>> growRect(bob, 50, 100)
```

تا وقتی که **growRect** در حال اجرا است، پارامتر **box** بدلی برای **bob** است. هر تغییری که بر روی **box** صورت گیرد، بر **bob** هم تأثیر می‌گذارد.

تمرین ۱۲-۳: یک تابع به نام **moveRect** بنویسید که یک **Rectangle** و دو پارامتر به نام‌های **dx** و **dy** بگیرد و مکان مستطیل را با اضافه کردن **dx** به مختصات **x** و **dy** به مختصات **y** به‌دست آورد.

۱۲-۸- کپی‌برداری

بدل‌سازی می‌تواند برنامه را برای خواندن مشکل کند. زیرا ممکن است تغییرات در یک مکان تأثیرات غیرمنتظره‌ای در مکان دیگر داشته باشد. ثبت سابقه همه متغیرهایی که ممکن است به یک شیء معطوف شوند مشکل است.

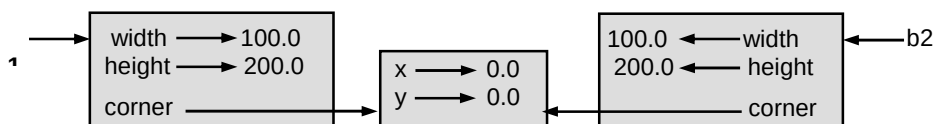
کپی کردن یک شیء اغلب چاره دیگری برای بدل‌سازی است. ماژول **copy** شامل تابعی به نام **copy** است که می‌تواند هر شیئی را تکثیر کند:

```
>>> import copy
>>> p1 = Point()
>>> p1.x = 3
>>> p1.y = 4
>>> p2 = copy.copy(p1)
>>> p1 == p2
0
>>> samePoint(p1, p2)
1
```

به محض اینکه ماژول **copy** را وارد محیط کاری کنیم، می‌توانیم از متد **copy** برای ساختن یک **Point** جدید استفاده کنیم. **p1** و **p2** یک نقطهٔ واحد نیستند، اما دارای داده‌های مشابهی هستند.

برای کپی کردن یک شیء ساده مانند **Point** که شامل اشیاء جاسازی شده‌ای نیست، **copy** کافی است. به این نوع کپی کردن، **کپی سطحی** گفته می‌شود.

برای اشیائی همچون **Rectangle** که شامل آدرسی به یک **Point** است، کپی کار کاملاً درست را انجام نمی‌دهد. این تابع آدرس شیء **Point** را کپی می‌کند، بنابراین هر دو **Rectangle** قبلی و **Rectangle** جدید به یک **Point** واحد معطوف می‌شوند. اگر ما مستطیلی به نام **b1** بسازیم و از راه معمول (با استفاده از کپی) یک کپی به نام **b2** از آن بگیریم. نمودار حالت حاصله به صورت زیر است:



شکل ۱۲-۳

این به‌طور حتم آن چیزی نیست که ما می‌خواستیم. در این مورد احضار **growRect** بر روی یکی از **Rectangle**‌ها تأثیری بر دیگری نخواهد گذاشت، اما احضار **moveRect** بر روی یکی، در هر دو مؤثر خواهد بود. این رفتار گیج‌کننده و زمینه‌ساز خطا می‌باشد.

خوشبختانه ماژول **copy** شامل متد دیگری به نام **deepcopy** است که علاوه بر خود شیء، هر شیء جاسازی شده درون آن را هم کپی می‌کند. شما از اینکه چرا این عملیات **کپی عمقی** نامیده می‌شود غافلگیر نخواهید شد:

```
>>> b2 = copy.deepcopy(b1)
```

حال **b1** و **b2** اشیاء کاملاً جداگانه‌ای هستند.

می‌توانیم از **deepcopy** برای بازنویسی **growRect** استفاده کنیم، به‌طوری‌که به جای تغییر دادن یک **Rectangle** موجود، یک **Rectangle** جدید بسازد. این **Rectangle** از نظر مکانی مشابه نوع قبلی است، اما ابعاد جدیدی دارد:

```
def growRect(box, dwidth, dheight):
    import copy
    newBox = copy.deepcopy(box)
    newBox.width = newBox.width + dwidth
    newBox.height = newBox.height + dheight
    return newBox
```

تمرین ۱۲-۴: **moveRect** را طوری بازنویسی کنید که به جای تغییر یک **Rectangle** قدیمی، **Rectangle** جدیدی را بسازد و برگرداند.

۱۲-۹- واژه‌نامه

class (کلاس)

یک نوع مرکب تعریف شونده توسط کاربر. یک کلاس همچنین می‌تواند به عنوان قالبی برای اشیایی که وهله‌هایی از آن هستند، در نظر گرفته شود.

instance (وهله)

شیئی که به یک کلاس تعلق دارد.

object (شیء)

یک نوع دادهٔ مرکب که اغلب برای مدل‌سازی چیزی یا مفهومی در دنیای واقعی استفاده می‌شود.

instantiate (وهله‌سازی)

ساختن وهله‌ای از یک کلاس.

constructor (سازنده)

متدی که برای ساختن یک شیء جدید استفاده می‌شود.

attribute (مشخصه)

یکی از اقلام داده‌ای دارای نام که وهله را تشکیل می‌دهد.

shallow equality (مساوات سطحی)

برابری آدرس‌ها و یا دو آدرس که به شیء واحدی اشاره می‌کنند.

deep equality (مساوات عمقی)

تساوی مقادیر یا دو آدرس که به اشیایی با مقادیر یکسان اشاره می‌کنند.

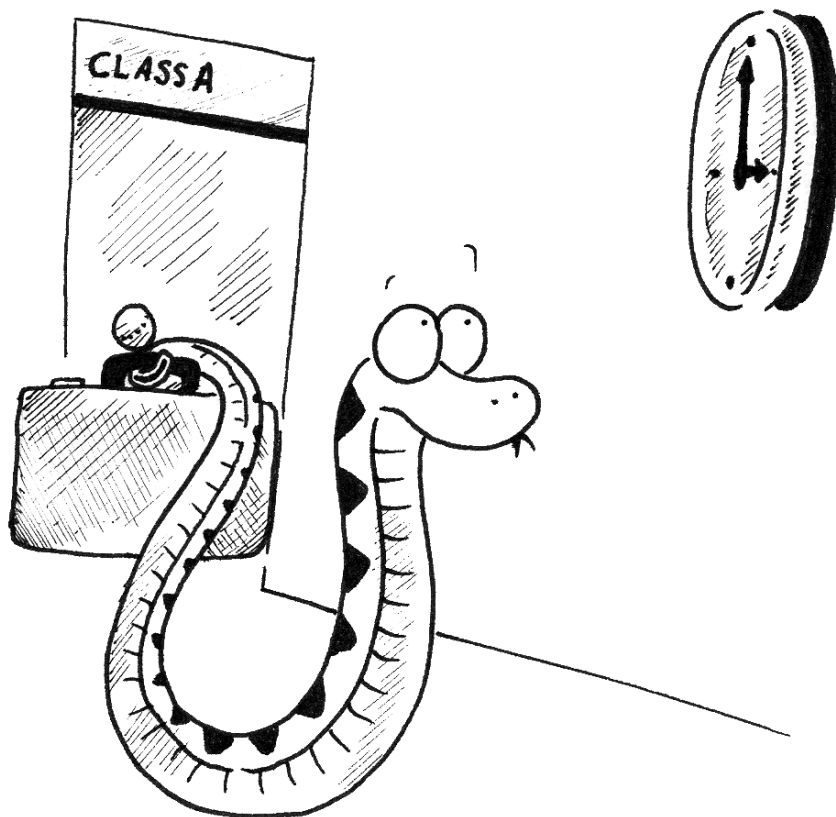
shallow copy (کپی سطحی)

کپی کردن مضمون یک شیء که در بر گیرنده هر یک از آدرس‌های اشیاء جاسازی شده در آن است؛ این کار توسط تابع **copy** که در ماژول **copy** قرار دارد انجام می‌شود.

deep copy (کپی عمیق)

کپی کردن محتویات یک شیء به طوری که اشیاء جاسازی شده در آن را هم در بر گیرد و نیز اشیاء جاسازی شده در آن اشیاء را هم در بر می‌گیرد و به همین ترتیب؛ این کار توسط تابع **deepcopy** از ماژول **copy** انجام می‌شود.

کلاس‌ها و توابع



در فصل گذشته توانایی جدیدی به مهارت‌هایتان در برنامه‌نویسی اضافه شد. شما آموختید که چگونه یک نوع داده‌ای جدید بسازید و از آن استفاده کنید. از آنجا که نقش توابع در به‌کارگیری انواع داده‌ای بسیار حائز اهمیت است، بنابراین باید بتوانید انواع داده‌ای که خود ساخته‌اید را در توابع به‌کار ببرید. هدف ما در این فصل ارتباط اشکال مختلف توابع با انواع داده‌ای کاربر-تعریف است. در پایان روش جدیدی برای طراحی برنامه‌ها معرفی می‌کنیم که به شما کمک می‌کند برنامه‌های اصولی‌تری بنویسید.

۱۳-۱- زمان

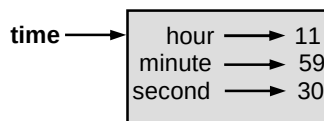
به‌عنوان مثالی دیگر برای یک نوع کاربر-تعریف، می‌خواهیم کلاسی به نام **Time** تعریف کنیم که وقت روز را ثبت کند. تعریف کلاس به این صورت است:

```
class Time:
    pass
```

می‌توانیم یک شیء **Time** جدید بسازیم و مشخصه‌هایی برای ساعت، دقیقه و ثانیه به آن نسبت دهیم:

```
time = Time()
time.hours = 11
time.minutes = 59
time.seconds = 30
```

نمودار حالت برای شیء **Time** به صورت زیر است:



شکل ۱۳-۱

تمرین ۱۳-۱: تابعی به نام **printTime** بنویسید که یک شیء **Time** را به‌عنوان آرگومان بگیرد و آن را به فرم **hours:minutes:seconds** چاپ کند.

تمرین ۱۳-۲: یک تابع بولی به نام **after** بنویسید که دو شیء **Time** به نام‌های **t1** و **t2** بگیرد و در صورتی که **t1** از نظر زمانی بعد از **t2** بود (1) **true** و در غیر این صورت (0) **false** را برگرداند.

۱۳-۲- توابع محض

به زودی ما دو نسخه از یک تابع به نام **addTime** را خواهیم نوشت که مجموع دو زمان را محاسبه می‌کند. آنها دو نوع از توابع را نشان خواهند داد: توابع محض و تغییردهنده‌ها. در زیر نسخهٔ اجمالی از تابع **addTime** را می‌بینید:

```
def addTime(t1, t2):
    sum = Time()
    sum.hours = t1.hours + t2.hours
    sum.minutes = t1.minutes + t2.minutes
    sum.seconds = t1.seconds + t2.seconds
    return sum
```

تابع یک شیء **Time** جدید می‌سازد، مشخصه‌های آن را مقدار دهی می‌کند و آدرسی به شیء جدید را برمی‌گرداند. این تابع، یک **تابع محض** نامیده می‌شود زیرا هیچ کدام از اشیائی که به عنوان پارامتر به آن فرستاده شده را تغییر نمی‌دهد و هیچ اثرات جانبی از قبیل نمایش یک مقدار یا گرفتن یک ورودی از کاربر ندارد.

در اینجا مثالی از چگونگی استفاده از این تابع را می‌بینید. ما دو شیء **Time** می‌سازیم: **currentTime** که شامل زمان جاری است و **breadTime** که مدت زمان برای پختن نان است. سپس ما از **addTime** برای به‌دست آوردن لحظه‌ای که نان پخته می‌شود استفاده می‌کنیم. اگر هنوز نوشتن تابع **printTime** را تمام نکرده‌اید، قبل از اینکه شروع به کار کنید ۱۴-۲ نگاهی بیاندازید:

```
>>> currentTime = Time()
>>> currentTime.hours = 9
>>> currentTime.minutes = 14
>>> currentTime.seconds = 30

>>> breadTime = Time()
>>> breadTime.hours = 3
>>> breadTime.minutes = 35
>>> breadTime.seconds = 0

>>> doneTime = addTime(currentTime, breadTime)
>>> printTime(doneTime)
```

خروجی برنامه 12:49:30 است که درست می‌باشد. از طرف دیگر مواردی وجود دارد که نتیجه درست نخواهد بود. آیا می‌توانید یکی از این موارد را حدس بزنید؟

مشکل آنجا است که این تابع وقتی که تعداد ثانیه‌ها یا دقیق بیشتر از شصت می‌شود درست عمل نمی‌کند. وقتی این حالت اتفاق می‌افتد ما مجبوریم ثانیه‌های اضافی را به ستون دقیقه‌ها و یا دقیقه‌های اضافی را به ستون ساعت‌ها انتقال دهیم.

در اینجا نسخه دوم و تصحیح شده تابع را می‌بینید:

```
def addTime(t1, t2):
    sum = Time()
    sum.hours = t1.hours + t2.hours
    sum.minutes = t1.minutes + t2.minutes
    sum.seconds = t1.seconds + t2.seconds

    if sum.seconds >= 60:
        sum.seconds = sum.seconds - 60
        sum.minutes = sum.minutes + 1

    if sum.minutes >= 60:
        sum.minutes = sum.minutes - 60
        sum.hours = sum.hours + 1

    return sum
```

اگرچه این تابع درست است، اما تعداد خطوط کد آن رو به افزایش است. در آینده راه حل دیگری پیشنهاد می‌کنیم که کد کوتاه‌تری را نتیجه می‌دهد.

۱۳-۳- تغییردهنده‌ها

مواردی وجود داشت که تغییر یک یا چند شیء که به عنوان پارامتر به تابع داده شده بودند برای آن بسیار مفید بود. معمولاً فراخواننده تابع آدرسی از اشیائی که به آن می‌فرستد را نگه می‌دارد، بنابراین هر تغییری که تابع به وجود آورد از دید فراخوان قابل رؤیت است. توابعی که به این صورت کار می‌کنند **تغییردهنده** نامیده می‌شوند.

تابع **increment** که تعداد معینی ثانیه را به شیء **Time** اضافه می‌کند، اگر به عنوان یک تغییردهنده نوشته شود به طور طبیعی تری ظاهر می‌شود. پیش‌نویس موقتی از این تابع به این صورت است:

```
def increment(time, seconds):
    time.seconds = time.seconds + seconds

    if time.seconds >= 60:
        time.seconds = time.seconds - 60
        time.minutes = time.minutes + 1

    if time.minutes >= 60:
        time.minutes = time.minutes - 60
        time.hours = time.hours + 1
```

اولین خط عمل اصلی را انجام می‌دهد و بقیه خطوط موارد خاصی را که قبلاً با آن مواجه شده‌ایم به کار می‌برند.

آیا این تابع درست است؟ اگر پارامتر **seconds** خیلی بزرگ‌تر از شصت باشد چه اتفاقی می‌افتد؟ در این صورت انتقال یک رقم نقلی کافی نیست؛ ما مجبوریم این کار را تا آنجا که **seconds** از 60 کمتر شود انجام دهیم. یک راه حل برای استفاده از دستور **while** به جای **if** است:

```
def increment(time, seconds):
    time.seconds = time.seconds + seconds

    while time.seconds >= 60:
        time.seconds = time.seconds - 60
        time.minutes = time.minutes + 1

    while time.minutes >= 60:
        time.minutes = time.minutes - 60
        time.hours = time.hours + 1
```

حال این تابع درست است، اما مؤثرترین راه ممکن نیست.

تمرین ۱۳-۳: این تابع را طوری بازنویسی کنید که هیچ حلقه‌ای در آن نباشد.

تمرین ۱۳-۴: تابع **increment** را به عنوان یک تابع محض بنویسید و فراخوانی‌های تابع را برای هر دو نسخه بنویسید.

۱۳-۴- کدام بهتر است؟

هر کاری که بتواند با تغییردهنده‌ها انجام گیرد می‌تواند با توابع محض هم انجام شود. در حقیقت بعضی از زبان‌های برنامه‌نویسی تنها توابع محض را پذیرا هستند. در اینجا شواهدی وجود دارد که نشان می‌دهد برنامه‌هایی که از توابه محض استفاده می‌کنند برای توسعه سریع‌ترند و نسبت به

برنامه‌هایی که از تغییردهنده‌ها استفاده می‌کنند، کمتر مستعد خطا هستند. با این حال، تغییردهنده‌ها گاهی مناسبند و در بعضی موارد برنامه‌های تابعمند کارایی کمتری دارند. در کل، ما پیشنهاد می‌کنیم که توابع محض را هرگاه که توجیه‌پذیرند بنویسید و تنها زمانی که یک مزیت اجتناب ناپذیر وجود دارد به تغییردهنده‌ها متوسل شوید. این روش برنامه‌نویسی را **سبک برنامه‌نویسی تابعمند** می‌نامند.

۱۳-۵- توسعه پیش‌نمونه در برابر برنامه طرح‌ریزی شده

در این فصل، ما راهی را برای توسعه برنامه نشان دادیم که آن را **توسعه پیش‌نمونه** می‌نامیم. در هر مورد، ما یک پیش‌نویس موقت (یا پیش‌نمونه) نوشتیم که محاسبات اساسی را انجام می‌داد و سپس آن را بر روی تعدادی از موارد امتحان می‌کردیم و همین‌که نقصی در برنامه می‌یافتیم آن را تصحیح می‌کردیم.

اگرچه این راه می‌تواند مؤثر باشد، اما ممکن است کد برنامه را طولانی و پیچیده سازد - چون با موارد ویژه و غیر قابل اعتماد زیادی سر و کار دارد - لذا برای آگاهی از همه خطاهایی که پیدا کرده‌اید بسیار مشکل خواهد شد.

یک راه دیگر برنامه‌نویسی **توسعه برنامه طرح‌ریزی شده** است که با داشتن یک بینش برتر در مسئله می‌تواند برنامه‌نویسی را بسیار آسان‌تر سازد. در این مثال، نحوه نگارش این‌طور است که یک شیء **Time** عددی سه رقمی در مبنای 60 می‌باشد! مؤلفه **second** ستون یکان است، مؤلفه **minute** ستون 60 تایی‌ها و مؤلفه **hour** ستون 3600 تایی‌ها.

وقتی که ما **addTime** و **increment** را نوشتیم، به طور کارآمدی جمع کردن در مبنای 60 را انجام دادیم که این، دلیل انتقال رقم نقلی از یک ستون به ستون دیگر است.

این نحوه نگارش راه دیگری را برای کل مسئله پیشنهاد می‌کند - ما می‌توانیم یک شیء **Time** را به عددی واحد تبدیل کنیم و از این واقعیت که کامپیوتر می‌داند چگونه اعمال حسابی را انجام دهد، نتیجه بگیریم. تابع زیر یک شیء **Time** را به عددی صحیح تبدیل می‌کند:

```
def convertToSeconds(t):
    minutes = t.hours * 60 + t.minutes
    seconds = minutes * 60 + t.seconds
    return seconds
```

حال، تمام آنچه نیاز داریم راهی برای تبدیل یک عدد صحیح به یک شیء **Time** است:

```
def makeTime(seconds):
    time = Time()
    time.hours = seconds/3600
    seconds = seconds - time.hours * 3600
    time.minutes = seconds/60
    seconds = seconds - time.minutes * 60
    time.seconds = seconds
    return time
```

ممکن است کمی فکر کنید تاخودتان را متقاعد سازید که این تکنیک تبدیل از یک مبنای به مبنای دیگر، صحیح است. به فرض اینکه متقاعد شده‌اید، حال می‌توانید از این تابع برای بازنویسی `addTime` استفاده کنید:

```
def addTime(t1, t2):
    seconds = convertToSeconds(t1) + convertToSeconds(t2)
    return makeTime(seconds)
```

این نسخه بسیار کوتاه‌تر از نسخه اصلی است. همچنین اثبات درستی آن بسیار ساده‌تر است (طبق معمول، با فرض اینکه توابع فراخوانی شده درست باشند).

تمرین ۱۳-۵: `increment` را به طریق مشابه بازنویسی کنید.

۱۳-۶- تعمیم

در برخی از روش‌ها، تبدیل از مبنای ۶۰ به مبنای ۱۰ و برگشت، سخت‌تر از کارکردن با زمان‌ها است. تبدیل مبنای خلاصه‌تر است پس نحوه نگارش ما نسبت به کار با زمان‌ها بهتر است. اما اگر ایده‌ای نسبت به رفتار با زمان‌ها به عنوان اعداد مبنای ۶۰ داشته باشیم و برای نوشتن توابع مبدل (`makeTime` و `convertToSeconds`) سرمایه‌گذاری بیشتری انجام دهیم، برنامه‌ای بسیار کوتاه‌تر، خواناتر، ساده‌تر برای اشکال‌زدایی و همچنین قابل اطمینان‌تر به‌دست خواهیم آورد. همچنین اضافه کردن ویژگی‌های دیگر در آینده بسیار آسان‌تر است. مثلاً تفريق دو شیء `Time` را برای پیدا کردن مدت زمان بین آنها تصور کنید. راه بسیار ساده‌ای برای این کار انجام عمل تفريق با استفاده از روش قرض‌گیری است. برای درک مفهوم قرض‌گیری به تفريق ۱۲-۱۵۰ توجه کنید. برای انجام این تفريق ۵ به ۰ یک واحد قرض می‌دهد. استفاده از توابع مبدل بسیار ساده‌تر و احتمالاً صحیح‌تر است.

جالب است بدانید گاهی دشوارسازی (یا کلی تر ساختن) یک برنامه آن را ساده تر می سازد (زیرا موارد خاص کاهش می یابد و همچنین احتمال خطا کمتر می شود).

۱۳-۷- الگوریتم‌ها

وقتی که یک راه حل کلی برای دسته‌ای از مسائل در مقابل یک راه حل خاص برای یک مسئله واحد می نویسد، شما یک **الگوریتم** نوشته‌اید. ما قبلاً به این کلمه اشاره کردیم اما به دقت آن را تعریف نمودیم. تعریف این واژه آسان نیست، بنابراین تعدادی راه حل را آزمایش می کنیم. اول چیزی را که تصور کنید که الگوریتم نیست. وقتی شما ضرب اعداد تک رقمی را یاد گرفتید، احتمالاً جدول ضرب را حفظ کردید. در حقیقت شما صد راه حل ویژه را به خاطر سپردید. این نوع دانش بر اساس الگوریتم نیست.

اگر تنبل بوده باشید، شاید به وسیله یاد گرفتن چند حقه کلک زده باشید. مثلاً برای پیدا کردن حاصل ضرب n و 9 می توانید $(n-1)$ را به عنوان اولین رقم و $n-10$ را به عنوان رقم دوم بنویسید. این حقه یک راه حل کلی برای ضرب هر عدد تک رقمی در 9 می باشد. مثلاً برای ضرب 6 در 9 می توانیم به این صورت عمل کنیم:

54	è	4 à 10-6	5 à 1-6
----	---	----------	---------

این یک الگوریتم است!

به طور مشابه شیوه‌هایی که برای جمع اعداد به وسیله ارقام نقلی، تفريق به وسیله قرضگیری و همچنین تقسیم اعداد چند رقمی یاد گرفته‌اید، همگی الگوریتم هستند. یکی از مشخصات الگوریتم‌ها این است که هیچ نیازی به هوشمندی برای انجام کار ندارند. آنها فرایندهایی مکانیکی و غیر فکری هستند که هر گام را بر طبق یک سری قوانین ساده دنبال می کنند.

به نظر ما، این شرم آور است که اشخاص وقت زیادی را در مدرسه صرف کنند تا نحوه اجرای الگوریتم‌ها را بیاموزند، چراکه این امر واقعاً به هیچ ذکاوتی نیاز ندارد. از طرف دیگر فرایند طراحی الگوریتم‌ها بسیار جالب است. این امر مبارزه‌ای خردمندانه و هسته مرکزی چیزی است که ما آن را برنامه نویسی می نامیم.

برخی از کارهایی که مردم به طور طبیعی و بدون هیچ سختی و آگاهی فکری انجام می دهند در بیان الگوریتمی بسیار دشوارند. فهمیدن زبان طبیعی مثال خوبی است. همه ما این کار را انجام می دهیم اما تا به حال هیچ کس نتوانسته است که توضیح دهد این کار چگونه صورت می گیرد، چه برسد به اینکه آن را در قالب یک الگوریتم بیان کند.



۱۳-۸- واژه‌نامه

pure function (تابع محض)

تابعی که تغییری در اشیائی که به عنوان پارامتر می‌گیرد، نمی‌دهد. اغلب توابع محض، نتیجه‌دار هستند.

modifier (تغییر دهنده)

تابعی که در یک یا چند شیء گرفته شده به عنوان پارامتر تغییر حاصل کند. بسیاری از تغییرردهنده‌ها چیزی را بر نمی‌گردانند.

functional programming style (سبک برنامه‌نویسی تابع‌مند)

سبکی از برنامه‌نویسی که در آن اکثریت توابع محض هستند.

prototype development (توسعه پیش‌نمونه)

یک راه توسعه برنامه که با یک مقدار اولیه آغاز می‌شود و با آزمایش تدریجی و توسعه افزایشی ادامه می‌یابد.

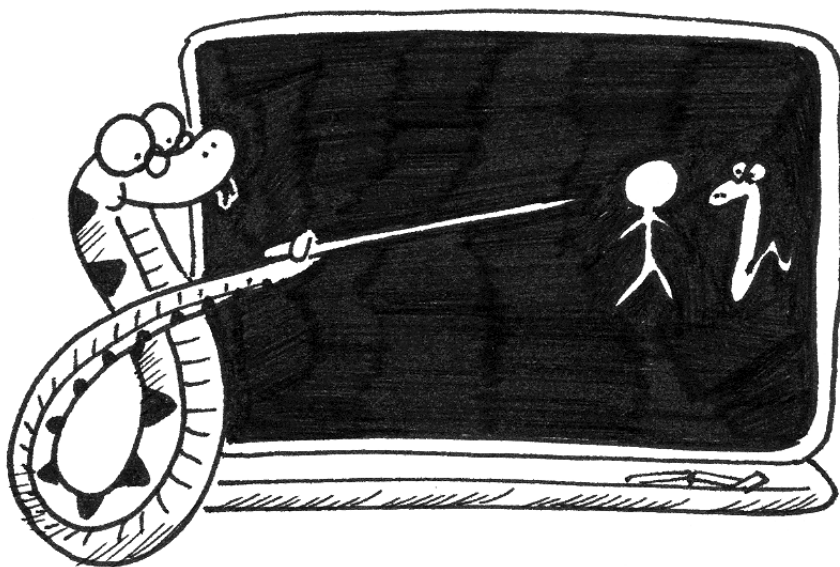
planned development (توسعه برنامه طراحی شده)

یک راه توسعه برنامه که مستلزم بینشی وسیع نسبت به مسئله است و شامل طراحی بیشتری نسبت به روش‌های توسعه افزایشی یا توسعه پیش‌نمونه می‌باشد.

algorithm (الگوریتم)

مجموعه‌ای از دستورات برای حل یک دسته از مسائل به‌وسیله فرایندی مکانیکی و غیرهوشمند.

کلاس‌ها و متدها



در فصل گذشته روش جدیدی را برای طراحی برنامه‌ها به کار بردیم و همچنین آموختیم که چگونه انواع داده‌ای کاربر-تعریف را در توابع استفاده کنیم. در این فصل قصد داریم یکی از مهمترین خصوصیات زبان پایتون را مطرح کنیم. این خصوصیت باعث شده است که قدرت توسعه برنامه‌ها در پایتون با قدرت توسعه برنامه‌ها در زبانی چون ++C مقایسه گردد.

۱۴-۱- خصوصیات شیء‌گرا

پایتون یک **زبان برنامه‌نویسی شیء‌گرا** است، یعنی خصوصياتی را دارا است که **برنامه‌نویسی شیء‌گرا** را پشتیبانی می‌کند. تعریف برنامه‌نویسی شیء‌گرا آسان نیست اما ما قبلاً بعضی از خصوصیات آن را دیده‌ایم:

- برنامه‌ها از تعاریف تابع و تعاریف شیء ساخته شده‌اند و اغلب محاسبات در روابطی از عملکرد بر روی اشیاء بیان شده‌اند.
- هر تعریف شیء با چند شیء یا مفهوم در دنیای واقعی همخوانی دارد و توابعی که بر روی آن شیء عمل می‌کنند متقابلاً روش‌هایی برای کار بر روی اشیاء در دنیای واقعیند.

برای مثال شیء **Time** که در فصل ۱۳ تعریف شد مطابق با راهی است که مردم برای ثبت اوقات روز استفاده می‌کنند و توابعی که ما تعریف کردیم مطابق با انواع کارهایی است که مردم با وقت انجام می‌دهند. به‌طور مشابه، کلاس‌های **Point** و **Rectangle** مطابق با مفاهیم ریاضی نقطه و مستطیل هستند.

تاکنون ما از خصوصياتی که پایتون برای پشتیبانی برنامه‌نویسی شیء‌گرا تدارک دیده بهره‌برداری نکرده‌ایم. اگر بخواهیم با صراحت صحبت کنیم این خصوصیات ضروری نیستند معمولاً آنها راه‌حل دیگری برای کارهایی که مثلاً انجام داده‌ایم فراهم می‌کنند، اما در اکثر موارد این راه‌حل بسیار مختصرتر است و ساختار برنامه را به‌طور دقیق‌تری بیان می‌کند.

برای مثال، در برنامه **Time** هیچ وابستگی آشکاری بین تعریف کلاس و تعاریف تابعی که دنبال می‌شد وجود ندارد. با انجام چند آزمایش ملاحظه می‌شود که هر تابع حداقل یک شیء **Time** را به‌عنوان پارامتر دریافت می‌کند. این نگرش انگیزه‌ای برای استفاده از متدها است. ما قبلاً چند متد از قبیل **values** و **keys** را دیده‌ایم که بر روی دیشکنری‌ها احضار می‌شدند. هر متد به یک کلاس متصل است و برای احضار بر روی وهله‌ای از کلاس در گرفته شده است. متدها درست شبیه به توابعند اما دو تفاوت دارند:

- متدها درون تعریف یک کلاس تعریف می‌شوند، به‌طوری‌که رابطه‌ای بین کلاس و متد صریح به‌وجود می‌آید.
 - نحوه نگارش برای احضار یک متد با نحوه فراخوانی یک تابع متفاوت است.
- در بخش‌های بعد ما توابعی از فصل‌های قبل می‌گیریم و آنها را به متد تبدیل می‌کنیم. این تبدیل کاملاً غیرهوشمند است. شما می‌توانید این کار را به سادگی و با دنبال کردن چند مرحله انجام دهید. اگر شما در تبدیل یک فرم به فرمی دیگر راحت باشید، قادر خواهید بود بهترین شکل برای هر آنچه انجام می‌دهید را انتخاب کنید.

۱۴-۲ printTime

در فصل ۱۳ ما کلاسی به‌نام **Time** تعریف کردیم و شما تابعی به نام **printTime** برای آن نوشتید که باید چیزی شبیه به این کد به‌نظر برسد:

```
class Time:
    pass

def printTime(time):
    print str(time.hours) + ":" +
          str(time.minutes) + ":" +
          str(time.seconds)
```

برای فراخوانی این تابع ما شیئی را به‌عنوان پارامتر به آن می‌فرستیم:

```
>>> currentTime = Time()
>>> currentTime.hours = 9
>>> currentTime.minutes = 14
>>> currentTime.seconds = 30
>>> printTime(currentTime)
```

برای اینکه **printTime** را به یک متد تبدیل کنیم، تمام کاری که باید انجام دهیم انتقال تعریف تابع درون تعریف کلاس است. توجه کنید که در این صورت کنگره‌گذاری رعایت شود.

```
class Time:
    def printTime(time):
        print str(time.hours) + ":" +
              str(time.minutes) + ":" +
              str(time.seconds)
```

حال ما می‌توانیم با استفاده از نشانه‌گذاری نقطه **printTime** را احضار کنیم:

```
>>> currentTime = Time()
>>> currentTime.hours = 9
>>> currentTime.minutes = 14
>>> currentTime.seconds = 30
>>> currentTime.printTime()
```

به‌طور معمول شیئی که متد بر روی آن احضار شده قبل از نقطه و نام متد پس از نقطه نشان داده می‌شود.

شیئی که متد بر روی آن احضار شده، به پارامتر اول نسبت داده می‌شود، بنابراین در این مثال **currentTime** به پارامتر **time** اختصاص می‌یابد.

بنابه قرارداد پارامتر اول متد **self** نامیده می‌شود، دلیل این نام‌گذاری کمی پیچیده است، اما بر اساس کنایه مفیدی بنا نهاده شده است.

نحوه نگارش برای فراخوانی یک تابع، **printTime(currentTime)**، این‌طور به ما القاء می‌کند که تابع یک خدمتکار فعال است و به او چیزی شبیه به این را می‌گوید: «آهای **printTime**، در اینجا شیئی وجود دارد که باید آن را چاپ کنی.»

در برنامه‌نویسی شیء‌گرا خدمتکاران فعال، شیء‌ها هستند. احضاری شبیه به **currentTime.printTime()** می‌گوید: «آهای **currenttime**، لطفاً خودت را چاپ کن.»

این تغییر ممکن است مؤدبانه‌تر به‌نظر برسد اما معلوم نیست مفید باشد. ممکن است در مثال‌هایی که تاکنون دیده‌ایم این‌طور نباشد، اما گاهی انتقال مسئولیت از توابع به اشیاء نوشتن توابع متنوع را ممکن ساخته و نگهداری و استفاده مجدد کدشان را آسان‌تر می‌سازد.

۱۴-۳- مثال دیگر

در اینجا تابع **increment** (از بخش ۱۳-۳) را به یک متد تبدیل می‌کنیم. برای جلوگیری از اتلاف فضا، ما از نوشتن متدهای تعریف شده قبلی صرف‌نظر می‌کنیم، اما شما باید آنها را در نسخه اجرایی خودتان منظور کنید:

```
class Time:
    #previous method definitions here...

    def increment(self, seconds):
        self.seconds = seconds + self.seconds

    while self.seconds >= 60:
        self.seconds = self.seconds - 60
        self.minutes = self.minutes + 1
```

```
while self.minutes >= 60:
    self.minutes = self.minutes - 60
    self.hours = self.hours + 1
```

تغییر شکل تابع به متد کاملاً غیرهوشمند است. ما تعریف متد را به داخل تعریف کلاس انتقال دادیم و پارامتر اول را تغییر دادیم. حال می‌توانیم **increment** را به‌عنوان یک مثال احضار کنیم:

```
currentTime = Time()
currentTime.hours = 9
currentTime.minutes = 14
currentTime.seconds = 30
currentTime.increment(500)
```

باز هم شیئی که متد بر روی آن احضار شده به پارامتر اول، **self**، اختصاص می‌یابد و دومین پارامتر، **seconds**، مقدار 500 را می‌گیرد.

تمرین ۱۴-۱: تابع **convertToSeconds** (از بخش ۱۸-۵) را به متدی درون کلاس **Time** تبدیل کنید.

۱۴-۶- یک مثال پیچیده‌تر

تابع **after** کمی پیچیده‌تر است، زیرا بر روی دو شیء **Time** عمل می‌کند، نه فقط یکی. ما تنها می‌توانیم یکی از پارامترها را به **self** تبدیل کنیم. پارامتر دوم در جای خود باقی می‌ماند:

```
class Time:
    #previous method definitions here...
    def after(self, time2):
        if self.hour > time2.hour:
            return 1
        if self.hour < time2.hour:
            return 0

        if self.minute > time2.minute:
            return 1
        if self.minute < time2.minute:
            return 0

        if self.second > time2.second:
            return 1
        return 0
```

ما این متد را بر روی یکی از اشیاء احضار می‌کنیم و دیگری را به‌عنوان یک آرگومان به آن

می‌فرستیم:

```
if doneTime.after(currentTime):
    print "The bread will be done after it starts."
```

می‌توانید احضار متد را تقریباً شبیه به یک جمله انگلیسی بخوانید: «اگر زمان پایان کار بعد از زمان کنونی است، آنگاه ...»

۱۴-۵- آرگومان‌های اختیاری

ما قبلاً توابع پیش‌ساخته‌ای دیده‌ایم که تعداد آرگومان‌های آن متغیر بود. برای مثال `string.find` می‌تواند دو، سه و یا چهار آرگومان بگیرد. نوشتن توابع کاربر-تعریف با لیست‌های آرگومان اختیاری امکان‌پذیر است. برای نمونه ما می‌توانیم نسخه تابع `find` خودمان را طوری به‌هنگام سازیم که مانند `find.string` عمل کند. این نسخه اصلی از بخش ۷-۷ است:

```
def find(str, ch):
    index = 0
    while index < len(str):
        if str[index] == ch:
            return index
        index = index + 1
    return -1
```

پارامتر سوم، `start`، اختیاری است زیرا یک مقدار پیش‌فرض یعنی ۰ برای آن تدارک دیده شده است. اگر ما `find` را تنها با دو آرگومان احضار کنیم، از مقدار پیش‌فرض استفاده کرده و از ابتدای رشته شروع می‌کنیم:

```
>>> find("apple", "p")
1
```

اگر پارامتر سوم را هم منظور کنیم، این مقدار، عمل مقدار پیش‌فرض را لغو می‌کند:

```
>>> find("apple", "p", 2)
2
>>> find("apple", "p", 3)
-1
```


تمرین ۱۴-۲: پارامتر چهارمی با نام `end` اضافه کنید که محل خاتمه جستجو را مشخص کند.

اخطار: این تمرین کمی فریبنده است. مقدار پیش‌فرض `end` باید `len(str)` باشد اما این مقدار کار نمی‌کند. مقادیر پیش‌فرض در زمان تعریف تابع ارزیابی می‌شوند نه در زمان فراخوانی آن. وقتی `find` تعریف شده هنوز `str` به نیامده است، لذا نمی‌توانیم طول آن را پیدا کنیم.

۱۴-۶- متد مقداردهی اولیه

متد مقداردهی اولیه متد ویژه‌ای است که هنگام ساختن یک شیء احضار می‌شود. نام این متد `__init__` است (با دو کاراکتر خط زیر در ابتدا، حروف `init` و دو کاراکتر خط زیر در انتها). یک متد مقدار اولیه برای کلاس `Time` به این صورت است:

```
class Time:
    def __init__(self, hours=0, minutes=0, seconds=0):
        self.hours = hours
        self.minutes = minutes
        self.seconds = seconds
```

هیچ تضادی میان مشخصه `self.hours` و پارامتر `hours` وجود ندارد. نمادگذاری نقطه متغیری را که ما به آن اشاره می‌کنیم مشخص می‌کند. وقتی که ما سازه `Time` را احضار می‌کنیم، آرگومان‌هایی که از قبل تدارک دیده‌ایم به `init` فرستاده می‌شوند:

```
>>> currentTime = Time(9, 14, 30)
>>> currentTime.printTime()
>>> 9:14:30
```

از آنجا که پارامترها اختیاری هستند، می‌توانیم از آنها صرف‌نظر کنیم:

```
>>> currentTime = Time()
>>> currentTime.printTime()
>>> 0:0:0
```

یا می‌توانیم فقط آرگومان اول را منظور کنیم:

```
>>> currentTime = Time(9)
>>> currentTime.printTime()
>>> 9:0:0
```

یا تنها دو آرگومان اول:

```
>>> currentTime = Time(9, 14)
>>> currentTime.printTime()
>>> 9:14:0
```

و سرانجام ما می‌توانیم مجموعه‌ای از پارامترها را با مشخص کردن نام صریح آنها استفاده کنیم:

```
>>> currentTime = Time(seconds = 30, hours = 9)
>>> currentTime.printTime()
>>> 9:0:30
```

۱۴-۷- بازگشتی به Point

بیایید کلاس **Point** از بخش ۱۲-۱ را به سبک شیء‌گراتری بنویسیم:

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __str__(self):
        return '(' + str(self.x) + ', ' + str(self.y) + ')'
```

متد مقداردهی اولیه مقادیر **x** و **y** را به‌عنوان پارامترهای اختیاری می‌گیرد. مقدار پیش‌فرض برای هر پارامتر، **0** است.

متد بعدی، **__str__**، نمایش رشته‌ای شیء **Point** را می‌گرداند. اگر یک کلاس متدی به نام **str** را تدارک ببیند، رفتار پیش‌ساخته **str** را لغو می‌کند.

```
>>> p = Point(3, 4)
>>> str(p)
'(3, 4)'
```

چاپ یک شیء **Point** مطلقاً **__str__** را بر روی شیء احضار می‌کند، بنابراین تعریف **__str__** رفتار تابع **print** را هم تغییر می‌دهد:

```
>>> p = Point(3, 4)
>>> print p
(3, 4)
```

وقتی ما یک کلاس جدید می‌نویسیم، تقریباً همیشه با نوشتن `__init__` شروع می‌کنیم که این کار کلاس را برای وهله‌سازی شیء ساده‌تر می‌سازد و `__str__` که تقریباً همیشه برای اشکال‌زدایی مفید است.

۱۴-۸- باردهی اضافی عملگر

بعضی زبان‌ها، تغییر تعریف عملگرهای پیش‌ساخته را وقتی که به‌وسیلهٔ انواع داده‌ای کاربر-تعریف استفاده می‌شوند، ممکن می‌سازند. این خصیصه **باردهی اضافی عملگر** نامیده می‌شود. این کار مخصوصاً هنگام تعریف یک نوع داده‌ای جدید ریاضی بسیار مفید است. برای مثال، جهت لغو عمل عملگر جمع، `+`، ما متدی به نام `__add__` را تهیه می‌کنیم:

```
class Point:
    # previously defined methods here...

    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)
```

به‌طور معمول، اولین پارامتر شیئی است که متد بر روی آن احضار می‌شود. دومین پارامتر به نحو مناسبی برای تمیز دادن آن از `self`، `other` نام‌گذاری شده است. برای جمع دو `Point` ما یک `Point` جدید که شامل جمع مختص‌های `x` و مختص‌های `y` است برمی‌گردانیم. حال وقتی عملگر `+` را برای شیء `Point` به کار می‌بریم، پایتون `__add__` را احضار می‌کند:

```
>>> p1 = Point(3, 4)
>>> p2 = Point(5, 7)
>>> p3 = p1 + p2
>>> print p3
(8, 11)
```

عبارت `p1+p2` معادل `p1.__add__(p2)` است، اما همان‌طور که می‌بینید به صورتی بسیار زیباتر.

تمرین ۱۴-۳: متدی به نام `__sub__(self, other)` به کلاس `Point` اضافه کنید که عملگر تفریق را باردهی اضافی کند و سپس آن را آزمایش نمایید.

برای لغو عمل عملگر ضرب چندین راه وجود دارد. یکی از آنها تعریف متدی به نام `__mul__` یا `__rmul__` یا هر دو است. اگر عملوند سمت چپ `*`، یک `Point` باشد، پایتون `__mul__` را احضار می‌کند، چرا که فرض می‌کند عملوند دیگر هم یک `Point` است. این متد حاصل ضرب نقطه‌ای دو نقطه را بر اساس قوانین جبر خطی محاسبه می‌کند:

```
def __mul__(self, other):
    return self.x * other.x + self.y * other.y
```

اگر عملوند سمت چپ `*` یکی از انواع داده‌ای اولیه و عملوند سمت راست، یک `Point` باشد، پایتون `__mul__` را احضار می‌کند که ضرب اسکالر را انجام می‌دهد:

```
def __rmul__(self, other):
    return Point(other * self.x, other * self.y)
```

نتیجه یک `Point` جدید است که مختص‌های آن یک مضرب از مختص‌های اصلی است. اگر `other` از نوعی باشد که نتواند در یک عدد اعشاری ضرب شود، `__rmul__` پیغام خطایی را نمایش خواهد داد.

این مثال هر دو نوع ضرب را نشان می‌دهد:

```
>>> p1 = Point(3, 4)
>>> p2 = Point(5, 7)
>>> print p1 * p2
43
>>> print 2 * p2
(10, 14)
```

اگر بخواهیم `p2*2` را محاسبه کنیم، چه اتفاقی می‌افتد؟ از آنجا که پارامتر اول یک `Point` است، پایتون `__mul__` را احضار می‌کند و `2` را به‌عنوان آرگومان دوم در نظر می‌گیرد. درون `__mul__` برنامه سعی می‌کند به مختص `x` از `other` دست یابد که مردود است، زیرا یک عدد صحیح هیچ مشخصه‌ای ندارد:

```
>>> print p2 * 2
AttributeError: 'int' object has no attribute 'x'
```

متأسفانه پیغام خطا کمی مبهم است. این برنامه بعضی از مشکلات برنامه‌نویسی شیء‌گرا را نشان می‌دهد. گاهی کشف کردن اینکه کدام کد در حال اجرا است به حد کافی مشکل است.

۱۴-۹- چندریختی

اغلب متدهایی که تا به حال نوشته‌ایم تنها برای یک نوع مخصوص کار می‌کنند. وقتی که یک شیء جدید می‌سازید، متدهایی می‌نویسید که روی آن نوع عمل می‌کنند. اما عملیات مشخصی وجود دارد که شما می‌خواهید بر روی بسیاری از انواع انجام دهید؛ از قبیل عملیات ریاضی که در بخش قبل دیدید. اگر انواع متعددی، یک مجموعه از عملیات یکسان را پشتیبانی کنند شما می‌توانید توابعی بنویسید که بر روی هر کدام از آن انواع کار کنند. برای نمونه عملیات **multadd** (که در جبر خطی رایج است) سه پارامتر دریافت می‌کند، اولی را در دومی ضرب کرده سپس با سومی جمع می‌کند. ما می‌توانیم آن را در پایتون به این صورت بنویسیم:

```
def multadd (x, y, z):
    return x * y + z
```

این متد برای هر مقدار **x** و **y** که بتوانند در هم ضرب شوند و برای هر مقدار **z** که بتواند با حاصل ضرب جمع شود کار خواهد کرد. ما می‌توانیم آن را با مقادیر عددی احضار کنیم:

```
>>> multadd (3, 2, 1)
7
```

یا با **Point**‌ها:

```
>>> p1 = Point(3, 4)
>>> p2 = Point(5, 7)
>>> print multadd (2, p1, p2)
(11, 15)
>>> print multadd (p1, p2, 1)
44
```

در اولین مورد **Point** در یک عدد ضرب می‌شود و سپس با یک **Point** دیگر جمع می‌شود و در دومین مورد حاصل ضرب نقطه‌ای یک مقدار عددی را نتیجه می‌دهد، بنابراین پارامتر سوم هم باید یک مقدار عددی باشد.

تابعی شبیه به این، که می‌تواند پارامترهایی با انواع مختلف را دریافت کند، چندریختی نامیده می‌شود.

به عنوان مثالی دیگر، متد **frontAndBack** را ملاحظه کنید که یک لیست را از اول به آخر و معکوس چاپ می کند:

```
def frontAndBack(front):
    import copy
    back = copy.copy(front)
    back.reverse()
    print str(front) + str(back)
```

از آنجا که متد **reverse** یک تغییردهنده است، ما قبل از معکوس کردن لیست یک **copy** از آن می سازیم. با این روش، متد لیستی را که به عنوان پارامتر وارد شده تغییر نمی دهد. در اینجا مثالی که **frontAndBack** را روی یک لیست به کار می برد ملاحظه می کنید:

```
>>> myList = [1, 2, 3, 4]
>>> frontAndBack(myList)
[1, 2, 3, 4][4, 3, 2, 1]
```

البته ما سعی کردیم این تابع را برای لیست ها به کار ببریم، لذا اگر این متد کار کند موجب شگفتی نیست. آنچه که ممکن است باعث تعجب شود آن است که ما بتوانیم متد را برای یک **Point** به کار ببریم.

برای تعیین اینکه آیا یک تابع می تواند برای یک نوع جدید به کار رود یا نه، ما از قوانین بنیادی چندریختی ها استفاده می کنیم:

اگر همه عملیات درون یک تابع بتواند برای یک نوع به کار رود تابع می تواند بر روی آن نوع به کار گرفته شود.

عملیات داخل متد شامل **copy**، **reverse** و **print** هستند.

copy بر روی هر شیء کار می کند و قبلاً متد **__str__** را هم برای همه **Point** ها نوشته ایم. حال تمام آنچه ما نیاز داریم نوشتن متد **reverse** در کلاس **Point** است:

```
def reverse(self):
    self.x , self.y = self.y, self.x
```

آنگاه می توانیم **Point** ها را به **frontAndBack** بفرستیم:

```
>>> p = Point(3, 4)
>>> frontAndBack(p)
(3, 4)(4, 3)
```

بهترین نوع چند شکلی نوع غیرعمدی است؛ در جایی که شما درمی‌یابید تابعی که قبلاً نوشته‌اید می‌تواند برای نوعی که اصلاً طراحی نکرده‌اید هم به کار رود.

۱۴-۱۰- واژه‌نامه

object-oriented programming language (زبان برنامه‌نویسی شیء‌گرا)

زبانی که خصوصیتی از قبیل کلاس‌های کاربر-تعریف و وراثت، که به برنامه‌نویسی شیء‌گرا کمک می‌کند را فراهم می‌نماید.

object-oriented programming (برنامه‌نویسی شیء‌گرا)

یک سبک برنامه‌نویسی که در آن داده‌ها و عملیاتی که آنها را اداره می‌کنند در کلاس‌ها و متدها سازمان می‌یابند.

method (متد)

تابعی که درون تعریف یک کلاس، تعریف شده و بر روی وهله‌های آن کلاس احضار می‌شود.

override (باطل کردن، لغو کردن)

جایگزینی با پیش‌فرض. مثال‌های این فصل شامل جایگزینی یک پارامتر پیش‌فرض با یک آرگومان خاص و یا جایگزینی یک متد پیش‌فرض به وسیله یک متد جدید تهیه شده با نام مشابه هستند.

initialization method (متد مقداردهی اولیه)

متد ویژه‌ای که هنگام ساخته شدن یک شیء جدید به طور خودکار احضار می‌شود و مشخصه‌های آن شیء را مقداردهی اولیه می‌کند.

operator overloading (باردهی اضافی عملگر)

توسعه عملگرهای پیش‌ساخته (+، -، *، <، > و غیره) به طوری که با انواع کاربر-تعریف هم کار کنند.

dot product (حاصل ضرب نقطه‌ای)

عملیاتی تعریف شده در جبر خطی که دو Point را در هم ضرب می‌کند و یک مقدار عددی را نتیجه می‌دهد.

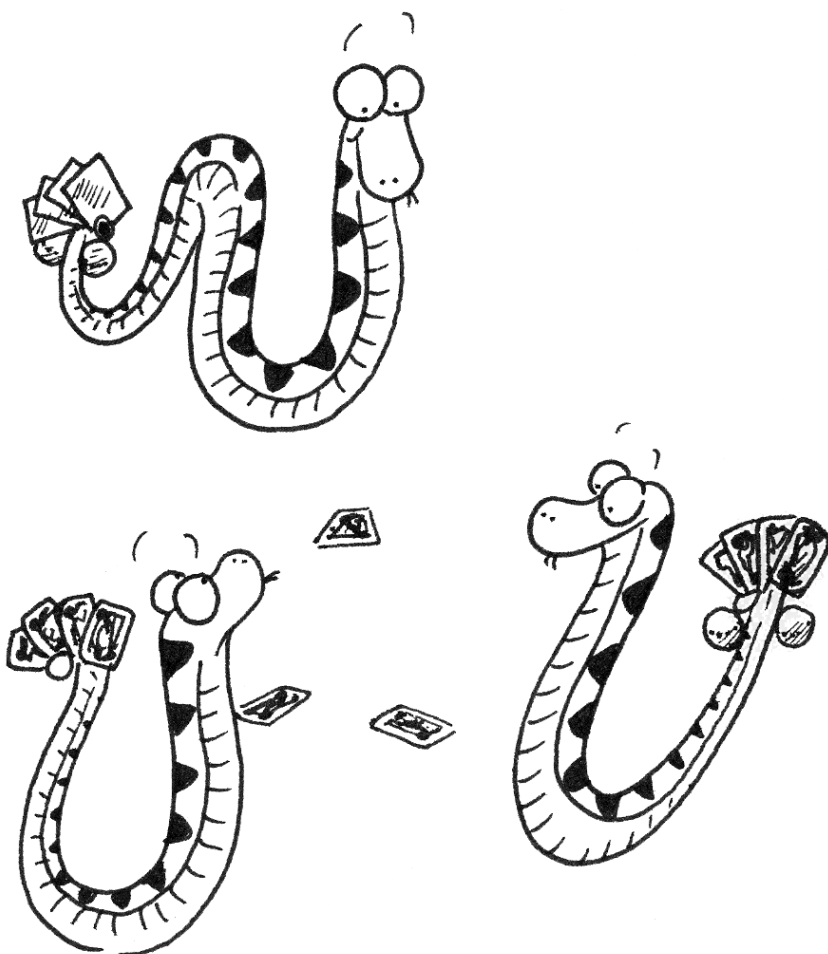
scalar multiplication (ضرب اسکالر)

عملیاتی تعریف شده در جبر خطی که هر یک از مختص‌های **Point** را در یک مقدار عددی ضرب می‌کند.

polymorphic (چندریختی، چند شکلی)

تابعی که می‌تواند بر روی بیش از یک نوع داده‌ای عمل کند. اگر همهٔ عملیات داخل یک تابع بتوانند برای یک نوع داده‌ای به کار روند تابع می‌تواند برای آن نوع داده‌ای به کار برده شود.

مجموعه‌های اشیاء



در فصل گذشته به بررسی مفهوم برنامه‌نویسی شیء‌گرا پرداختیم و با مفهوم کلاس و متد به‌عنوان خصیصه‌هایی از یک زبان برنامه‌نویسی شیء‌گرا آشنا شدیم. در این فصل ضمن مروری بر مبحث ترکیب، قصد داریم با طرح یک مثال، خصوصیات دیگری از برنامه‌نویسی شیء‌گرا را بررسی کنیم.

۱۵-۱- ترکیب

تاکنون مثال‌های زیادی از ترکیب دیده‌اید. یکی از اولین مثال‌ها، استفاده از یک احضار متد به عنوان قسمتی از یک عبارت بود. مثال دیگر، ساختار تودرتوی دستورات است؛ شما می‌توانید یک دستور **if** را درون یک حلقه **while** یا درون دستور **if** دیگری قرار دهید و

با دیدن این الگو و آموختن در مورد لیست‌ها و اشیاء نباید از فهمیدن این موضوع که می‌توانید لیست‌هایی از اشیاء بسازید، متعجب شوید. شما همچنین می‌توانید شیء‌هایی بسازید که شامل لیست‌ها (به عنوان مشخصه‌ها) باشند. شما قادرید لیست‌هایی بسازید که شامل لیست‌های دیگری باشند و می‌توانید اشیائی بسازید که شامل شیء‌های دیگری باشند و

در این فصل و فصل آینده با استفاده از اشیاء **Card** به عنوان نمونه، مثال‌هایی از این ترکیب‌ها خواهید دید.

۱۵-۲- شیء Card

ما در اینجا قصد نداریم به شما نحوه استفاده از کارت‌های بازی را آموزش دهیم، بلکه مثال‌های این فصل تنها برای آشنایی بیشتر شما با مفهوم شیء و برنامه‌نویسی شیء‌گرا است.

یک دسته ورق شامل پنجاه و دو کارت است. این کارت‌ها به چهارگروه سیزده‌تایی با چهار خال متفاوت تقسیم شده‌اند. در برخی از بازی‌ها ارزش هر خال نسبت به دیگر خال‌ها متفاوت است. مثلاً به این صورت:

Clubs < Diamonds < Hearts < Spades

هر دسته سیزده‌تایی از خال‌ها به این ترتیب است:

Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King

بسته به نوع بازی ممکن است امتیاز **Ace** از **King** بیشتر، یا از 2 کمتر باشد.

اگر بخواهیم شیء جدیدی برای نمایش یک کارت بازی تعریف کنیم، واضح است که مشخصه‌ها باید چه باشند: **rank** (رتبه کارت) و **suit** (خال کارت). اما چندان معلوم نیست که مشخصه‌ها باید از چه نوع داده‌ای باشند. یک راه، استفاده از رشته‌هایی شامل کلماتی نظیر **"Spades"** برای نوع خال و **"Queen"** برای رتبه کارت است. یکی از مشکلات این کار این است که مقایسه کارت‌ها از لحاظ رتبه (**rank**) و خال (**suit**) چندان آسان نیست.

چاره دیگر، استفاده از اعداد صحیح برای به رمز درآوردن رتبه‌ها و خال‌ها است. منظور ما از به رمز درآوردن مطابق با آنچه مردم تصور می‌کنند، یعنی تبدیل به یک سری علائم نیست. آنچه یک متخصص کامپیوتر از «به رمز درآوردن» تعبیر می‌کند، «تعریف یک نگاشت میان دنباله‌ای از اعداد و عناصری که می‌خواهیم نمایش دهیم» است. برای مثال:

Spades	à	3
Hearts	à	2
Diamonds	à	1
Clubs	à	0

یک صورت آشکار این نگاشت آن است که خال‌ها به ترتیب به اعداد صحیح اشاره کنند، بنابراین می‌توانیم خال‌ها را به وسیله مقیاس اعداد صحیح، با هم مقایسه کنیم. نگاشت رتبه خال‌ها نسبتاً مشخص است؛ هر خال عددی به عدد صحیح نظیرش مرتبط است و برای کارت‌های صورت‌دار داریم:

Jack	à	11
Queen	à	12
King	à	13

علت استفاده از نمادگذاری ریاضی (استفاده از پیکان‌ها) برای نگاشت‌ها این است که آنها بخشی از برنامه پایتون نیستند بلکه قسمتی از طراحی برنامه‌اند و هرگز به‌طور صریح در کد برنامه ظاهر نمی‌شوند. تعریف کلاس برای نوع داده‌ای **Card** به صورت زیر است:

```
class Card:
    def __init__(self, suit=0, rank=0):
        self.suit = suit
        self.rank = rank
```

طبق معمول یک متد مقداره‌ی اولیه در نظر گرفته‌ایم که پارامترهای اختیاری را برای هر مشخصه دریافت می‌کند.

برای ساختن شیئی که 3 از Clubs را نمایش دهد، از این فرمان استفاده کنید:

```
threeOfClubs = Card(0,3)
```

آرگومان اول، 0، خال Clubs را نمایش می‌دهد.

۱۵-۳- مشخصه‌های کلاس و متد __str__

به‌منظور چاپ شیء‌های Card، آن هم به صورتی که هر کس بتواند آن‌ها را به راحتی بخواند، لازم است کدهایی صحیح (integer) به کلمات نگاشت شوند. یک راه طبیعی برای انجام این کار استفاده از لیست‌هایی از رشته‌ها است. ما در قسمت ابتدای تعریف کلاس، این لیست‌ها را به مشخصه‌های کلاس نسبت می‌دهیم:

```
class Card:
    suitList = ["Clubs", "Diamonds", "Hearts", "Spades"]
    rankList = ["narf", "Ace", "2", "3", "4", "5", "6", "7",
                "8", "9", "10", "Jack", "Queen", "King"]

    #init method omitted

    def __str__(self):
        return (self.rankList[self.rank] + " of " +
                self.suitList[self.suit])
```

درون __str__ ما می‌توانیم از suitList و rankList جهت نگاشتن مقادیر عددی suit و rank به رشته استفاده کنیم. برای نمونه عبارت self.suitList[self.suit] به این معنی است که: «مشخصهٔ suit از شیء self را به عنوان اندیسی در مشخصهٔ کلاسی به نام suitList به کار ببر و رشتهٔ متناظر را انتخاب کن.»

علت وجود "narf" در اولین عنصر rankList، نگه داشتن محل صفرمین عنصر لیست است که هرگز استفاده نمی‌شود. تنها رتبه‌های معتبر، 1 تا 13 هستند. این قلم دادهٔ هرز کاملاً ضروری نیست. ما می‌توانستیم از 0 شروع کنیم، اما رمزگذاری 2 به عنوان 2 و 3 به عنوان 3 و ...، کمتر گیج کننده است.

با متدهایی که تاکنون داشته‌ایم می‌توانیم کارت‌ها را بسازیم و چاپ کنیم:

```
>>> card1 = Card(1, 11)
>>> print card1
Jack of Diamonds
```

مشخصه‌های کلاس، همچون **suitList** با تمام اشیاء **Card** در اشتراکند. مزیت این خاصیت آن است که می‌توانیم هر شیء **Card** را برای دسترسی به مشخصه‌های کلاس به کار ببریم:

```
>>> card2 = Card(1, 3)
>>> print card2
3 of Diamonds
>>> print card2.suitList[1]
Diamonds
```

عیب این روش آن است که اگر ما یک مشخصه کلاس را تغییر دهیم، بر روی تمام وهله‌های کلاس اثر می‌گذارد. برای مثال، اگر تصمیم بگیریم که کارت **"Jack of Diamonds"**، از این پس **"Jack of Swirly Whales"** خوانده شود، می‌توانیم به این صورت عمل کنیم:

```
>>> card1.suitList[1] = "Swirly Whales"
>>> print card1
Jack of Swirly Whales
```

مشکل آن است که تمام **Diamonds**‌ها به **Swirly Whales** تبدیل می‌شوند:

```
>>> print card2
3 of Swirly Whales
```

این کار معمولاً ایده خوبی برای تغییر مشخصه‌های کلاس نیست.

۱۵-۸- مقایسهٔ کارت‌ها

برای انواع داده‌ای قبلی عملگرهای شرطی (**=**، **>**، **<** و غیره) وجود دارد که مقادیر را مقایسه می‌کنند و مشخص می‌کنند که چه زمان یکی نسبت به دیگری بزرگ‌تر، کوچک‌تر و یا مساوی است. برای انواع داده‌ای کاربر-تعریف می‌توانیم با ارائهٔ متدی به نام **__cmp__** رفتار عملگرهای پیش‌ساخته را لغو کنیم. طبق قرارداد، **__cmp__** دو پارامتر **self** و **other** را می‌گیرد و اگر شیء اول بزرگ‌تر باشد **1**، در صورتی که شیء دوم بزرگ‌تر باشد **-1** و در حالتی که برابر باشند **0** را برمی‌گرداند.

برخی از انواع داده‌ای کاملاً به صورت متوالی هستند، یعنی شما می‌توانید هر دو عضوی را مقایسه کنید و بگویید کدام یک بزرگ‌تر است. برای مثال اعداد صحیح و اعداد اعشاری کاملاً متوالیند. برخی مجموعه‌ها نامتوالیند، به این معنی که برای تعیین بزرگ‌تری عضوی نسبت به دیگری راه معینی

وجود ندارد. مثلاً میوه‌ها نامتوالیند، به همین دلیل شما نمی‌توانید سیب یا پرتقال را از نظر اندازه با هم مقایسه کنید.

مجموعه کارت‌های بازی پاره‌مرتب هستند، یعنی برخی اوقات می‌توانید کارت‌ها را با هم مقایسه کنید و گاهی هم نمی‌توانید. برای مثال، می‌دانید 3 از Clubs بزرگ‌تر از 2 از Clubs است و 3 از Diamonds بیشتر از 3 از Clubs می‌باشد. اما کدام یک بهتر است، 3 از Clubs یا 2 از Diamonds؟ یکی از لحاظ عددی بزرگ‌تر است و دیگری خال با ارزش‌تری دارد.

به منظور قیاس‌پذیر ساختن کارت‌ها باید تصمیم بگیرید که کدام یک مهم‌تر است، رتبه یا خال. در حقیقت این انتخاب، اختیاری و قراردادی است. جهت انجام این انتخاب ما فرض می‌کنیم که خال مهم‌تر است، زیرا یک دست کارت نو بر اساس خال‌ها مرتب شده‌اند، مثلاً مجموعه Clubs ها در کنار هم، پس از آن مجموعه Diamonds ها و به همین ترتیب.

با این قرارداد می‌توانیم __cmp__ را بنویسیم:

```
def __cmp__(self, other):
    # check the suits
    if self.suit > other.suit: return 1
    if self.suit < other.suit: return -1
    # suits are the same... check ranks
    if self.rank > other.rank: return 1
    if self.rank < other.rank: return -1
    # ranks are the same... it's a tie
    return 0
```

در این مرتب‌سازی، Ace ها (1ها) دارای ارزش کمتری نسبت به 2ها هستند.

تمرین ۱۵-۱: __cmp__ را طوری تغییر دهید که ارزش Ace ها بیشتر از King ها باشد.

۱۵-۵- دسته‌های ورق

اکنون که اشیائی برای Card ها داریم، قدم بعدی تعریف کلاسی برای نمایش Deck است. از آنجا که یک دسته ورق از تعدادی کارت تشکیل شده، بنابراین هر شیء Deck شامل لیستی از کارت‌ها به عنوان یک مشخصه است.

در ادامه یک تعریف برای کلاس Deck می‌بینید. متد مقداردی اولیه مشخصه Cards را می‌سازد و مجموعه استاندارد از ۵۲ کارت را تولید می‌کند:

```
class Deck:
    def __init__(self):
        self.cards = []
        for suit in range(4):
            for rank in range(1, 14):
                self.cards.append(Card(suit, rank))
```

آسان‌ترین راه برای جمع کردن یک دسته ورق، استفاده از حلقه‌های تودرتو است. حلقه بیرونی، خال‌ها را یکی‌یکی از ۰ تا ۳ و حلقه درونی، رتبه‌ها را از ۱ تا ۱۳ می‌شمارد. از آنجا که حلقه خارجی چهار بار و حلقه درونی سیزده بار تکرار می‌شود، تعداد کل دفعاتی که بدنه اجرا می‌شود، ۵۲ مرتبه است. هر بار تکرار حلقه، وهله **Card** جدیدی با خال و رتبه کنونی می‌سازد و آن کارت را به لیست **cards** اضافه می‌کند. متد **append** بر روی لیست‌ها (و نه چندتایی‌ها) کار می‌کند.

۱۵-۶ چاپ یک دسته ورق

طبق معمول، هر گاه شیئی از یک نوع جدید را تعریف می‌کنیم، به متدی احتیاج داریم که محتوای شیء را چاپ کند. برای چاپ یک **Deck** لیست را پیمایش کرده و هر **Card** را چاپ می‌نماییم:

```
class Deck:
    ...
    def printDeck(self):
        for card in self.cards:
            print card
```

در اینجا و از این پس، علامت سه‌نقطه (...) نشان می‌دهد که ما متدهای دیگر را در کلاس حذف کرده‌ایم. راه دیگری برای **printDeck**، نوشتن یک متد **__str__** برای کلاس **Deck** است. مزیت **__str__** انعطاف‌پذیری بیشتر آن است. مهم‌تر از چاپ محتوا اینک، این کار یک نمایش رشته‌ای تولید می‌کند که می‌تواند توسط بخش‌های دیگر برنامه قبل از چاپ دستکاری و یا برای استفاده‌های بعدی ذخیره شود.

در اینجا نسخه‌ای از **__str__** را می‌بینید که نمایش رشته‌ای یک **Deck** را برمی‌گرداند. این متد کارت‌ها را به صورت آبشاری می‌چیند که در این حالت هر کارت به اندازه یک فاصله بیشتر از کارت قبلی کنگره‌گذاری شده است:

```
class Deck:
    ...
    def __str__(self):
        s = ""
        for i in range(len(self.cards)):
            s = s + " " + str(self.cards[i]) + "\n"
        return s
```

این مثال چندین خصیصه را نمایش می‌دهد. اول، به جای پیمایش **self.cards** و نسبت‌دهی هر کارت به یک متغیر، از **i** به عنوان یک متغیر حلقه و اندیسی برای لیست کارت‌ها استفاده می‌کنیم.

دوم، ما از ضرب رشته‌ها استفاده می‌کنیم تا هر کارت را با یک فاصله بیشتر نسبت به قبلی کنگره‌گذاری کنیم.

سوم، به جای استفاده از دستور **print** برای چاپ کارت‌ها، از تابع **__str__** استفاده می‌کنیم. ارسال یک شیء به **str** به عنوان یک آرگومان، معادل احضار متد **__str__** بر روی شیء است.

در نهایت ما از متغیر **s** به عنوان یک انباشتگر استفاده می‌کنیم. در آغاز **s** یک رشته تهی است. در هر بار اجرای حلقه یک رشته جدید تولید می‌شود و به مقدار قبلی **s** می‌پیوندد تا مقدار جدیدی بگیرد. زمانی که حلقه پایان می‌یابد، **s** حاوی نمایش رشته‌ای و کامل **Deck** می‌باشد که به صورت زیر است:

```
>>> deck = Deck()
>>> print deck
Ace of Clubs
2 of Clubs
3 of Clubs
4 of Clubs
5 of Clubs
6 of Clubs
7 of Clubs
8 of Clubs
9 of Clubs
10 of Clubs
Jack of Clubs
Queen of Clubs
King of Clubs
Ace of Diamonds
```

و به همین صورت.

حتی اگر نتیجه در ۵۲ خط چاپ شود، تنها یک رشته طولانی است که شامل کاراکترهای خط جدید می‌باشد.

۱۵-۷- بُر زدن یک دسته ورق

اگر یک دسته ورق ماهرانه بُر خورده باشد، آنگاه هر کارت با احتمالی مساوی در هر جای دسته ورق ظاهر می‌شود و هر مکان در دسته، احتمال برابری برای شامل بودن هر کارت را دارد. به منظور بُر زدن یک دسته ورق، ما از تابع **randrange** از ماژول **random** استفاده می‌کنیم. با دو آرگومان صحیح **a** و **b**، **randrange** یک عدد صحیح در بازه $a \leq x < b$ انتخاب می‌کند. از آنجا که کران بالایی حتماً از **b** کمتر است، می‌توانیم از طول لیست به عنوان پارامتر دوم استفاده نماییم و تضمین کنیم که اندیس مجازی دریافت می‌کنیم. برای مثال، این عبارت اندیس یک کارت را در یک دسته ورق انتخاب می‌کند:

```
random.randrange(0, len(self.cards))
```

یک راه ساده برای بر زدن یک دسته ورق، پیمایش کارت‌ها و جابجایی هر کارت با یک کارت تصادفی انتخاب شده از دسته است. امکان دارد کارتی با خودش جابجا شود، اما باز هم خوب است. در حقیقت اگر از این امکان جلوگیری کنیم، ترتیب کارت‌ها به‌طور کامل اتفاقی نخواهد بود:

```
class Deck:
    ...
    def shuffle(self):
        import random
        nCards = len(self.cards)
        for i in range(nCards):
            j = random.randrange(i, nCards)
            self.cards[i], self.cards[j]=self.cards[j], self.cards[i]
```

بدون اینکه فرض کنیم در دسته ورق ۵۲ کارت وجود دارد، طول واقعی لیست را می‌گیریم و در **nCards** ذخیره می‌کنیم.

برای هر کارت در دسته ورق، ما کارتی را از میان کارت‌ها انتخاب می‌کنیم که تا به حال بُر نخورده باشد. آنگاه کارت جاری (**i**) را با کارت انتخابی (**j**) عوض می‌کنیم. جهت معاوضه کارت‌ها همان‌طور که در بخش ۹-۲ دیدید از یک نسبت‌دهی چندتایی استفاده می‌کنیم:

```
self.cards[i], self.cards[j] = self.cards[j], self.cards[i]
```

تمرین ۱۵-۲: این خط کد را بدون استفاده از نسبت‌دهی چندتایی بازنویسی کنید.

۱۵-۸- حذف و تقسیم کارتها

متد دیگری که برای کلاس **Deck** مفید خواهد بود، **removeCard** است. این متد یک کارت را به عنوان پارامتر می‌گیرد، آن را حذف می‌کند و مقدار **true** (1) را در صورت وجود و **false** (0) را در صورت عدم وجود کارت برمی‌گرداند:

```
class Deck:
    ...
    def removeCard(self, card):
        if card in self.cards:
            self.cards.remove(card)
            return 1
        else:
            return 0
```

در صورتی که عملوند اول، درون عملوند دوم (که باید یک لیست یا چندتایی باشد) وجود داشته باشد، عملگر **in** مقدار **true** را برمی‌گرداند. اگر عملوند اول یک شیء باشد، پایتون از متد **__cmp__** متعلق به شیء استفاده می‌کند تا برابری اقلام لیست را معین کند. از آنجا که **__cmp__** در کلاس **Card** مساوات عمقی را چک می‌کند، متد **removeCard** هم مساوات عمقی را بررسی می‌کند.

برای توزیع کارتها، قصد داریم کارت بالایی را حذف کنیم و برگردانیم. متد **pop** که بر روی لیست‌ها عمل می‌کند، راه مناسبی برای انجام این کار ارائه می‌دهد:

```
class Deck:
    ...
    def popCard(self):
        return self.cards.pop()
```

در حقیقت، **pop** کارت آخر لیست را حذف می‌کند. بنابراین ما در حال توزیع فعال از پایین دسته ورق هستیم.

عمل دیگری که مایلیم دسته باشیم، تابع بولی **isEmpty** است. در صورتی که دسته ورق حاوی هیچ کارتی نباشد، این تابع مقدار **true** را برمی‌گرداند:

```
class Deck:
    ...
    def isEmpty(self):
        return (len(self.cards) == 0)
```

encode (به رمز در آوردن)

نمایش مجموعه‌ای از مقادیر با استفاده از مجموعه دیگری از مقادیر به وسیله ساختن نگاشتی میان آنها.

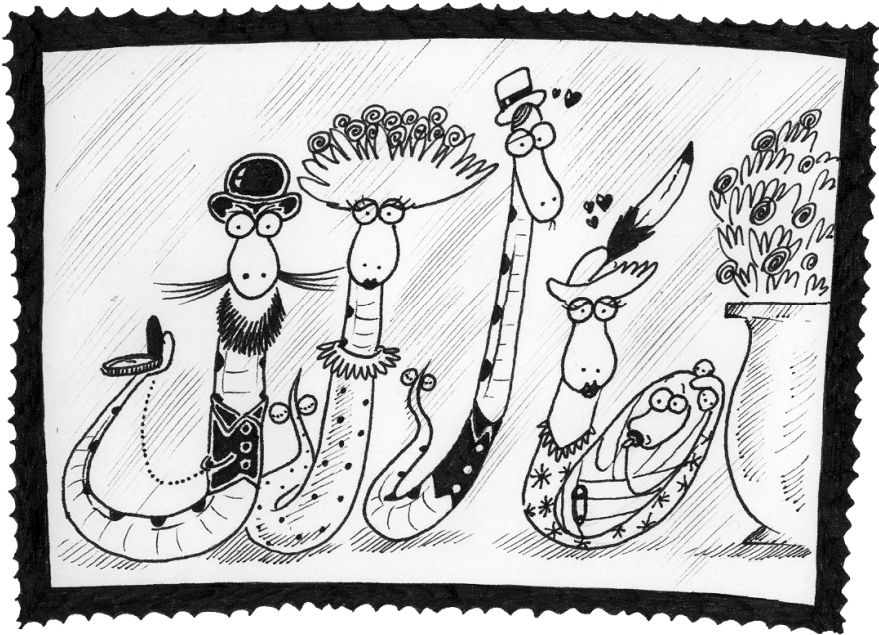
class attribute (مشخصه کلاس)

متغیری که درون تعریف یک کلاس، اما خارج از هر متد، تعریف شده است. مشخصه‌های کلاس از هر متد در کلاس قابل دسترسی هستند و میان تمام وهله‌های کلاس مشترکند.

accumulator (انباشتگر)

متغیری که درون حلقه به منظور انباشتن یک سری از مقادیر انباشته می‌شود؛ مانند به هم پیوستن آنها به صورت یک رشته یا اضافه کردن آنها به یک مجموع جاری.

وراثت



در فصل گذشته با عنوان کردن مثال **Card** ها توانستیم یک شی، واقعی را بدل سازی کنیم. در این فصل نیز با توسعه این مثال قصد داریم یکی از ویژگی های اساسی زبان که اغلب بیشترین ارتباط را با برنامه نویسی شیء گرا دارد عنوان کنیم.

۱۶-۱- وراثت

وراثت توانایی تعریف کلاس جدیدی است که نسخه تغییر یافته ای از یک کلاس موجود می باشد. مزیت اصلی این ویژگی آن است که شما می توانید متدهای جدیدی را به یک کلاس اضافه کنید، بدون اینکه کلاس موجود را تغییر دهید. از آنجایی که کلاس جدید تمام متدهای کلاس موجود را به ارث می برد، این عمل وراثت نامیده می شود. با توسعه این استعاره، کلاس موجود گاهی **کلاس والد** نامیده می شود. به کلاس جدید، گاهی **کلاس فرزند** یا «زیر کلاس» گفته می شود.

وراثت یک ویژگی قدرتمند است. برخی برنامه ها که ممکن است بدون وراثت پیچیده باشند، می توانند توسط این خصوصیت به طور ساده تر و مختصرتری نوشته شوند. وراثت همچنین می تواند استفاده مجدد از کد را آسان سازد، زیرا شما می توانید رفتار کلاس های والد را بدون دستکاری آنها تغییر دهید. در برخی موارد، ساختار وراثت بر روی ساختار طبیعی مسئله تأثیر می گذارد که این موضوع فهم برنامه را آسان تر می سازد.

از سوی دیگر، وراثت می تواند خوانایی برنامه را دشوارتر سازد. هنگامی که یک متد احضار می شود، گاهی یافتن محل تعریف آن واضح نیست. کد مربوطه ممکن است میان چندین ماژول پراکنده باشد. همچنین بسیاری از کارهایی که می توانند با استفاده از وراثت انجام شوند، بدون استفاده از آن هم می توانند به همان زیبایی (و یا حتی بیشتر) انجام شوند. اگر ساختار طبیعی مسئله برای وراثت مناسب نباشد، این سبک برنامه نویسی می تواند زیان آور باشد.

در این فصل ما کاربرد وراثت را با کمک برنامه ای که بازی **Old Maid** را انجام می دهد، نمایش خواهیم داد. یکی از اهداف ما نوشتن کدی است که بتوان در پیاده سازی بازی های دیگر از آن استفاده کرد.

۱۶-۲- یک دست کارت

تقریباً برای هر بازی، به نمایش یک دست کارت نیاز داریم. یک دست کارت شبیه به یک دسته ورق است. هر دو از مجموعه کارت ها تشکیل شده اند و هر دو به اعمالی نظیر اضافه و حذف کردن کارت ها نیاز دارند، همچنین ممکن است به توانایی بُر زدن دسته ورق ها و هر دست از کارت ها نیاز داشته باشیم.

به علاوه یک دست کارت با یک دسته ورق متفاوت است. بسته به نوع بازی که انجام می شود، ممکن است بخواهید اعمالی روی یک دست کارت انجام دهید که انجام آنها بر روی یک دسته ورق معنی نداشته باشد. برای مثال، در بازی **Poker** ممکن است یک دست کارت را دسته بندی کنیم و یا آن را با یک دست کارت دیگر مقایسه کنیم. در بازی **Bridge** ممکن است بخواهیم به منظور حکم کردن، امتیازی را برای یک دست کارت محاسبه کنیم.

در این حالت می توانیم از وراثت استفاده کنیم. اگر **Hand** زیر کلاسی از **Deck** باشد، تمام متدهای آن را خواهد داشت و می توان متدهای جدیدی را هم به آن اضافه کرد. در تعریف کلاس، نام کلاس والد در پرانتز ظاهر می شود:

```
class Hand(Deck):
    pass
```

این دستور نشان می دهد که کلاس جدید **Hand** از کلاس **Deck** موجود ارث بری دارد. سازنده **Hand** مشخصه هایی را برای یک دست کارت مقداردهی اولیه می کند که این مقادیر **name** و **cards** هستند. رشته **name**، این دست کارت را احتمالاً به وسیله نام بازیکن صاحب آن مشخص می کند. **name**، یک پارامتر اختیاری با مقدار پیش فرض تهی است. **cards**، لیست کارت های درون یک دست است که با لیستی تهی مقداردهی اولیه شده است:

```
class Hand(Deck):
    def __init__(self, name=""):
        self.cards = []
        self.name = name
```

تقریباً برای هر کارت بازی، لازم است کارت هایی از یک دسته ورق اضافه و حذف کنیم. حذف کارت ها قبلاً در نظر گرفته شده، زیرا **Hand.removeCard** را از **Deck** به ارث برده است، اما **addCard** را باید بنویسیم:

```
class Hand(Deck):
    ...
    def addCard(self, card):
        self.cards.append(card)
```

باز هم یادآوری می کنیم که علامت (...) نشان می دهد که از نوشتن متدهای دیگر صرف نظر کرده ایم. متد **append** کارت جدیدی را به انتهای لیست کارت ها اضافه می کند.

۱۶-۳- توزیع کارتها

اکنون که یک کلاس **Hand** داریم، می‌خواهیم کارتها را از **Deck** به دست‌ها توزیع کنیم. در ابتدا مشخص نیست که این متد باید در کلاس **Hand** باشد یا کلاس **Deck**، اما از آنجا که این متد روی یک دسته ورق واحد و احتمالاً روی چند دست کارت عمل می‌کند، طبیعی‌تر است که آن را در **Deck** قرار دهیم.

deal باید به اندازه کافی جامع باشد، زیرا بازی‌های گوناگون نیازمندی‌های مختلفی دارند. ممکن است بخواهیم یک مجموعه ورق دست‌نخورده را یکباره توزیع کنیم، یا اینکه کارتها را یکی‌یکی به هر دست اضافه کنیم.

deal دو پارامتر می‌گیرد. پارامتر اول یک لیست (یا چندتایی) از دست‌ها و تعداد کل کارتهایی است که باید توزیع شود. اگر کارتهای کافی درون دسته ورق نباشد، متد تمام کارتها را توزیع می‌کند و متوقف می‌شود:

class Deck:

```
...
def deal(self, hands, nCards=999):
    nHands = len(hands)
    for i in range(nCards):
        if self.isEmpty(): break # break if out of cards
        card = self.popCard() # take the top card
        hand = hands[i % nHands] # whose turn is next?
        hand.addCard(card) # add the card to the hand
```

پارامتر دوم، **nCards**، اختیاری است: مقدار پیش‌فرض عدد بزرگی است که یعنی تمام کارتهای درون دسته ورق توزیع خواهد شد.

متغیر حلقه، **i**، از ۰ تا **nCards-1** را سیر می‌کند. هر بار در طول حلقه یک کارت از دسته ورق با استفاده از متد **pop** حذف می‌شود. این متد آخرین عنصر را حذف می‌کند و برمی‌گرداند. عملگر باقی‌مانده (%) به ما این اجازه را می‌دهد که کارتها را با نوبت گردشی توزیع کنیم (هر کارت در هر نوبت به یک دست). هنگامی که **i** با تعداد دست‌های داخل لیست برابر می‌شود، عبارت **i%nHands** به ابتدای لیست (اندیس ۰) می‌چرخد.

۱۶-۴ چاپ یک دست کارت

برای چاپ محتوای یک دست کارت، می‌توانیم از متدهای `printDeck` و `__str__` که از `Deck` ارث‌بری دارند، سود ببریم. به عنوان مثال:

```
>>> deck = Deck()
>>> deck.shuffle()
>>> hand = Hand("frank")
>>> deck.deal([hand], 5)
>>> print hand
Hand frank contains
2 of Spades
3 of Spades
4 of Spades
Ace of Hearts
9 of Clubs
```

اگرچه ارث‌بری از متدهای موجود بی‌دردسر و مناسب است، اما اطلاعات اضافی در یک شیء `Hand` موجود است که ممکن است بخواهیم در زمان چاپ، آنها را منظور کنیم. برای انجام این کار می‌توانیم یک متد `__str__` در کلاس `Hand` تدارک ببینیم که هم‌نوع خود در کلاس `Deck` را لغو کند:

```
class Hand(Deck)
...
def __str__(self):
    s = "Hand " + self.name
    if self.isEmpty():
        s = s + " is empty\n"
    else:
        s = s + " contains\n"
    return s + Deck.__str__(self)
```

در آغاز `s` رشته‌ای است که یک دست را شناسایی می‌کند. اگر دست خالی باشد، برنامه کلمات `is empty` را اضافه می‌کند و `s` را برمی‌گرداند. در غیر این صورت برنامه کلمه `contains` و نمایش رشته‌ای `Deck` را اضافه می‌کند. نمایش رشته‌ای `Deck` با احضار متد `__str__` بر روی `self` در کلاس `Deck` محاسبه شده است.

ارسال `self` (که به `Hand` کنونی اشاره می‌کند) به یک متد `Deck`، تا زمانی که به یاد می‌آورید `Hand` نوعی `Deck` است، ممکن است عجیب به نظر برسد. در کل استفاده از وهلهٔ یک زیرکلاس به جای وهلهٔ یک کلاس والد عملی مجاز است.

۱۶-۵- کلاس CardGame

کلاس **CardGame** برخی از اعمال رایج و مشترک در تمام بازی‌ها همچون ساختن یک دسته ورق و بُر زدن آن را پشتیبانی می‌کند:

```
class CardGame:
    def __init__(self):
        self.deck = Deck()
        self.deck.shuffle()
```

این اولین موردی است که مشاهده می‌کنیم متد مقداردهی اولیه در جایی خارج از مشخصه‌های مقداردهی اولیه، محاسبات مهمی را انجام می‌دهد.

به‌منظور پیاده‌سازی بازی‌های بخصوص می‌توانیم از **CardGame** ارث‌بری داشته باشیم و ویژگی‌هایی به بازی جدید اضافه کنیم. به عنوان یک مثال، بازی **Old Maid** را شبیه‌سازی می‌کنیم. شما با جفت و جور کردن کارت‌ها بر اساس رتبه و رنگ از شر آنها خلاص می‌شوید. برای مثال **4** از **Clubs** با **4** از **Spades** جور است، زیرا هر دو دارای خال مشکی هستند. **Jack** از **Hearts** هم با **Jack** از **Diamonds** جور است، زیرا هر دو قرمز هستند.

برای شروع بازی، **Queen** از **Clubs** از دسته ورق حذف می‌شود، بنابراین **Queen** از **Spades** هیچ جفتی ندارد. پنجاه و یک کارت باقی‌مانده به نوبت چرخشی بین بازیکنان تقسیم شده است. پس از توزیع کارت‌ها تمام بازیکنان هر تعداد کارت را که ممکن باشد جفت کرده و می‌اندازند. وقتی که دیگر نتوان جفت کارتی تشکیل داد بازی آغاز می‌شود. هر بازیکن به نوبت یک کارت را (بدون نگاه کردن) از نزدیک‌ترین بازیکن مجاور سمت چپ خود که هنوز کارت در دست دارد، انتخاب می‌کند و برمی‌دارد. اگر کارت انتخاب شده، با کارتی در دست بازیکن جفت شود، هر دو کارت از دست خارج می‌شود. در غیر این صورت کارت به دست بازیکن اضافه می‌شود. سرانجام تمام جفت‌های ممکن ساخته می‌شوند و تنها **Queen** از **Spades** در دست بازنده باقی می‌ماند.

در مشابه کامپیوتری این بازی، تمام دست‌ها را کامپیوتر بازی می‌کند. متأسفانه برخی از نکات ظریف واقعی از دست رفته‌اند. در یک بازی واقعی، بازیکنی که **Old Maid** را در دست دارد تلاش می‌کند با برجسته‌تر نشان دادن این کارت بازیکن مجاور خود را به انتخاب آن ترغیب کند، اما کامپیوتر به سادگی کارتی را به طور اتفاقی از دست بازیکن مجاور برمی‌دارد.

۱۶-۶- کلاس OldMaidHand

یک دست کارت برای بازی **OldMaid** علاوه بر قابلیت‌های عمومی یک **Hand**، نیاز به توانایی‌های دیگری هم دارد. ما کلاس جدیدی به نام **OldMaidHand** را تعریف می‌کنیم که از **Hand** ارث‌بری دارند و یک متد اضافه به نام **removeMatches** را ارائه می‌دهد:

```
class OldMaidHand(Hand):
    def removeMatches(self):
        count = 0
        originalCards = self.cards[:]
        for card in originalCards:
            match = Card(3 - card.suit, card.rank)
            if match in self.cards:
                self.cards.remove(card)
                self.cards.remove(match)
                print "Hand %s: %s matches %s" % (self.name, card, match)
                count = count + 1
        return count
```

کار را با ساختن یک کپی از کارت‌ها شروع می‌کنیم، به‌طوری‌که می‌توانیم در حالی که کارت‌ها را از لیست اصلی حذف می‌کنیم، کپی را پیمایش نماییم. از آنجا که **self.cards** در حلقه تغییر یافته است، نمی‌خواهیم از آن برای کنترل پیمایش استفاده کنیم. پایتون ممکن است کاملاً گیج شود، اگر لیستی را پیمایش نماید که در حال تغییر است.

برای هر کارت موجود در دست، مشخص می‌کنیم چه کارتی با آن جفت می‌شود و به دنبال آن می‌گردیم. کارت هم‌تا (جفت شده) دارای رتبه‌ای برابر و خالی دیگر از همان رنگ است.

عبارت **3-crad.suit**، یک **Club** (خال ۰) را به یک **Spade** (خال ۱) را به یک **Heart** (خال ۲) تبدیل می‌کند. شما باید خود را متقاعد کنید که عملگرهای متضاد نیز کار می‌کنند. اگر کارت هم‌تا هم در دست باشد هر دو حذف می‌شوند.

مثال زیر چگونگی استفاده از **removeMatches** را نمایش می‌دهد:

```
>>> game = CardGame()
>>> hand = OldMaidHand("frank")
>>> game.deck.deal([hand], 13)
>>> print hand
Hand frank contains
Ace of Spades
2 of Diamonds
7 of Spades
8 of Clubs
```

```
6 of Hearts
8 of Spades
7 of Clubs
Queen of Clubs
7 of Diamonds
5 of Clubs
Jack of Diamonds
10 of Diamonds
10 of Hearts
```

```
>>> hand.removeMatches()
Hand frank: 7 of Spades matches 7 of Clubs
Hand frank: 8 of Spades matches 8 of Clubs
Hand frank: 10 of Diamonds matches 10 of Hearts
>>> print hand
Hand frank contains
Ace of Spades
2 of Diamonds
6 of Hearts
Queen of Clubs
7 of Diamonds
5 of Clubs
Jack of Diamonds
```

دقت کنید در کلاس `OldMaidHand` متد `__init__` وجود ندارد. ما از `Hand` ارث‌بری داریم.

۱۶-۷- کلاس `OldMaidGame`

حال ما می‌توانیم ذهنمان را متوجه خود بازی کنیم. `OldMaidGame` زیرکلاسی از `CardGame` است با متد جدیدی به نام `play` که لیستی از بازیکنان را به‌عنوان پارامتر دریافت می‌کند.

از آنجا که `__init__` از `CardGame` ارث‌بری شده، شیء `OldMaidGame` جدید شامل یک دست کارت جدید بُرخورده است:

```
class OldMaidGame(CardGame):
    def play(self, names):
        # remove Queen of Clubs
        self.deck.removeCard(Card(0,12))

        # make a hand for each player
        self.hands = []
```

```

for name in names:
    f.hands.append(OldMaidHand(name))

# deal the cards
self.deck.deal(self.hands)
print "----- Cards have been dealt"
self.printHands()

# remove initial matches
matches = self.removeAllMatches()
print "----- Matches discarded, play begins"
self.printHands()

# play until all 50 cards are matched
turn = 0
numHands = len(self.hands)
while matches < 25:
    matches = matches + self.playOneTurn(turn)
    turn = (turn + 1) % numHands

print "----- Game is Over"
self.printHands()
    
```

برخی از گام‌های بازی در متدها تفکیک شده‌اند. `removeAllMatches` لیست دست‌ها را پیمایش می‌کند و `removeMatches` را روی هر کدام احضار می‌نماید:

```

class OldMaidGame(CardGame):
    ...
    def removeAllMatches(self):
        count = 0
        for hand in self.hands:
            count = count + hand.removeMatches()
        return count
    
```

تمرین ۱۶-۱: یک `printHands` را بنویسید که `self.hands` را پیمایش می‌کند و هر دست را چاپ می‌نماید.

`count` یک انباشتگر است که تعداد جفت‌ها در هر دست را جمع می‌زند و مجموع را برمی‌گرداند.

هنگامی که تعداد کل جفت‌ها به بیست و پنج رسید، پنجاه کارت از دست‌ها حذف شده که یعنی تنها یک کارت باقی مانده و بازی تمام است.

متغیر **turn** پیگیری نوبت بازیکنان را بر عهده دارد. این متغیر از 0 شروع می‌شود و هر بار یک واحد افزایش می‌یابد. هنگامی که به **numHand** رسید، عملگر باقی‌مانده آن را دوباره به 0 برمی‌گرداند.

متد **playOneTurn** یک پارامتر می‌گیرد که مشخص می‌کند نوبت چه کسی است. مقدار بازگشتی تعداد جفت‌های تشکیل یافته در طول این دور است:

```
class OldMaidGame(CardGame):
    ...
    def playOneTurn(self, i):
        if self.hands[i].isEmpty():
            return 0
        neighbor = self.findNeighbor(i)
        pickedCard = self.hands[neighbor].popCard()
        self.hands[i].addCard(pickedCard)
        print "Hand", self.hands[i].name, "picked", pickedCard
        count = self.hands[i].removeMatches()
        self.hands[i].shuffle()
        return count
```

اگر دست بازیکنی خالی شد، آن بازیکن از دور بازی خارج است، بنابراین عملی انجام نمی‌دهد و 0 را برمی‌گرداند.

در غیر این صورت یک دور بازی عبارت است از یافتن اولین بازیکن سمت چپ دارای کارت، گرفتن یک کارت از دست مجاور و بررسی جفت‌ها. قبل از بازگشت، کارت‌های درون دست بُرخورده‌اند، به‌طوری‌که کارت انتخابی بازیکن بعدی اتفاقی است.

متد **findNeighbor** با بازیکن مجاور سمت چپ شروع می‌کند و در یک مسیر حلقوی تا زمانی که بازیکن دارای کارتی را بیابید، ادامه می‌دهد:

```
class OldMaidGame(CardGame):
    ...
    def findNeighbor(self, i):
        numHands = len(self.hands)
        for next in range(1, numHands):
            neighbor = (i + next) % numHands
            if not self.hands[neighbor].isEmpty():
                return neighbor
```

اگر `findNeighbor` یک دور کامل چرخید و هیچ کارتی پیدا نکرد `None` را برمی‌گرداند و موجب رخداد یک خطا می‌شود، مگر اینکه درون برنامه باشیم. خوشبختانه می‌توانیم برنامه را طوری ارتقاء دهیم که این اتفاق هرگز رخ ندهد (تا زمانی که پایان بازی به درستی شناسایی شود). ما متد `printHand` را حذف کرده‌ایم. شما می‌توانید خودتان آن را بنویسید.

خروجی زیر صورتی از یک بازی ناقص است که در آن تنها پانزده کارت فوقانی (10ها و کارت‌های بالاتر) بین سه بازیکن تقسیم شده است. با این دسته ورق کوچک بازی به جای اینکه پس از بیست و پنج جفت تمام شود بعد از هفت جفت پایان می‌یابد:

```
>>> import cards
>>> game = cards.OldMaidGame()
>>> game.play(["Allen", "Jeff", "Chris"])
----- Cards have been dealt
Hand Allen contains
King of Hearts
Jack of Clubs
  Queen of Spades
    King of Spades
      10 of Diamonds

Hand Jeff contains
Queen of Hearts
Jack of Spades
  Jack of Hearts
    King of Diamonds
      Queen of Diamonds

Hand Chris contains
Jack of Diamonds
King of Clubs
  10 of Spades
    10 of Hearts
      10 of Clubs

Hand Jeff: Queen of Hearts matches Queen of Diamonds
Hand Chris: 10 of Spades matches 10 of Clubs
----- Matches discarded, play begins
Hand Allen contains
King of Hearts
Jack of Clubs
  Queen of Spades
    King of Spades
      10 of Diamonds

Hand Jeff contains
```

```

Jack of Spades
Jack of Hearts
  King of Diamonds

Hand Chris contains
Jack of Diamonds
King of Clubs
  10 of Hearts

Hand Allen picked King of Diamonds
Hand Allen: King of Hearts matches King of Diamonds
Hand Jeff picked 10 of Hearts
Hand Chris picked Jack of Clubs
Hand Allen picked Jack of Hearts
Hand Jeff picked Jack of Diamonds
Hand Chris picked Queen of Spades
Hand Allen picked Jack of Diamonds
Hand Allen: Jack of Hearts matches Jack of Diamonds
Hand Jeff picked King of Clubs
Hand Chris picked King of Spades
Hand Allen picked 10 of Hearts
Hand Allen: 10 of Diamonds matches 10 of Hearts
Hand Jeff picked Queen of Spades
Hand Chris picked Jack of Spades
Hand Chris: Jack of Clubs matches Jack of Spades
Hand Jeff picked King of Spades
Hand Jeff: King of Clubs matches King of Spades
----- Game is Over
Hand Allen is empty

Hand Jeff contains
Queen of Spades

Hand Chris is empty

```

بنابراین Jeff می‌بازد.

۱۶-۸- واژه‌نامه

inheritance (وراثت)

توانایی تعریف یک کلاس جدید که نسخه‌ی تغییر یافته‌ای از یک کلاس تعریف شده است.

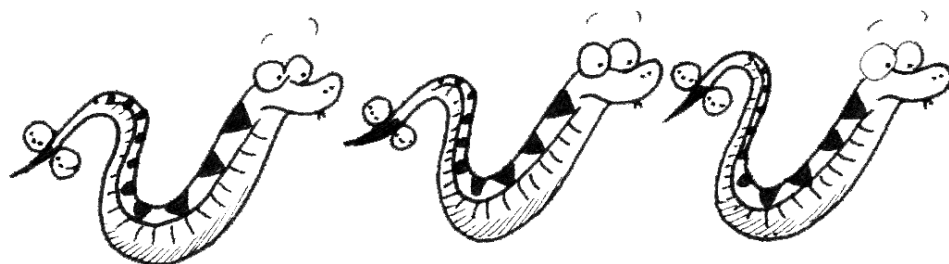
parent class (کلاس والد)

کلاسی که یک کلاس فرزند از آن ارث‌بری دارد.

child class (کلاس فرزند)

کلاس جدیدی که به وسیله ارث‌بری از یک کلاس موجود، ساخته شده و “زیر کلاس” هم نامیده می‌شود.

لیست‌های پیوندی



در فصل ۱۵ و ۱۶ با بررسی مثال **Card** ها تقریباً به طور کامل با برنامه نویسی شیء گرا آشنا شدید. حال می توانیم شروع به طراحی ساختمان های داده کنیم. ساختمان داده، یک تصویر سازمانی است به داده اعمال می شود تا تفسیر آن یا انجام اعمال بر روی آن آسان شود. در چهار فصل آینده تصمیم داریم که شما را با ساختمان داده های مختلفی آشنا کنیم و در این میان با ویژگی های مختلف برنامه نویسی شیء گرا آشنا شدیم.

۱۷-۱- ارجاع های توکار

ما تاکنون مثال هایی دیده ایم که مشخصه ها به دیگر اشیاء ارجاع می کنند، که آنها را **ارجاع های توکار** نامیدیم (به بخش ۸-۱۲ نگاه کنید). یک ساختمان داده رایج به نام **لیست پیوندی** از این خصوصیت بهره می برد.

لیست های پیوندی از **گره ها** تشکیل شده اند که هر گره شامل آدرسی به گره بعدی لیست است. در مجموع، هر گره شامل واحدی از داده به نام **بار** است.

لیست پیوندی، یک ساختمان داده بازگشتی است زیرا یک تعریف بازگشتی دارد.

یک لیست پیوندی چنین است:

- لیست تهی، نمایش داده شده بوسیله **None**، یا
- یک گره که شامل یک شیء بار و آدرسی به لیست پیوندی است.

ساختمان داده های بازگشتی خودشان را به متدهای بازگشتی معطوف می دارند.

۱۷-۲- کلاس Node

به طور معمول هنگام نوشتن یک کلاس جدید، کار را با متد مقداردهی اولیه و متد **__str__** شروع می کنیم، به طوری که بتوانیم مکانیسم مقدماتی ساخت و نمایش نوع جدید را آزمایش کنیم:

```
class Node:
    def __init__(self, cargo=None, next=None):
        self.cargo = cargo
        self.next = next

    def __str__(self):
        return str(self.cargo)
```

مطابق معمول، پارامترها برای متد مقداردهی اولیه اختیاری هستند. به‌طور پیش‌فرض، هر دو مقدار بار (**cargo**) و پیوند (**next**)، در آغاز **None** قرار داده می‌شوند.

نمایش رشته‌ای برای یک گره، تنها نمایش رشته‌ای بار آن گره است. از آنجا که هر مقداری می‌تواند به تابع **str** فرستاده شود، می‌توانیم آنها را در یک لیست ذخیره کنیم.

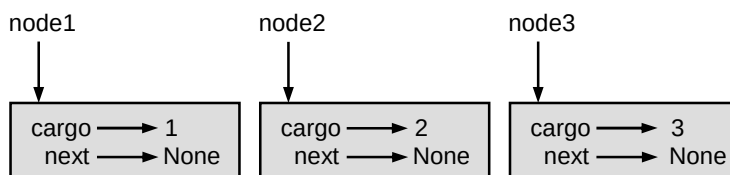
برای آزمایش اجرای برنامه تا این مرحله، می‌توانیم یک **Node** بسازیم و آن را چاپ کنیم.

```
>>> node = Node("test")
>>> print node
test
```

برای اینکه آزمایش را جالب‌تر کنیم، به لیستی با گره‌های بیشتر نیاز داریم:

```
>>> node1 = Node(1)
>>> node2 = Node(2)
>>> node3 = Node(3)
```

این کد سه گره تولید می‌کند اما هنوز یک لیست نداریم، زیرا گره‌ها به هم پیوند نشده‌اند. نمودار حالت بدین شکل است:

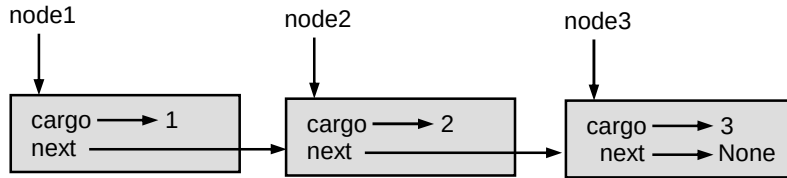


شکل ۱۷-۱

برای به هم پیوستن گره‌ها مجبوریم اولین گره را به دومی و دومی را به سومی ارجاع دهیم:

```
>>> node1.next = node2
>>> node2.next = node3
```

سومین گره به **None** اشاره می‌کند که این نمایانگر پایان لیست است. اکنون نمودار حالت مطابق شکل ۱۷-۲ است.



شکل ۱۷-۲

حال می‌دانید چگونه گره‌ها را بسازید و آنها را به لیست‌ها پیوند دهید. آنچه در اینجا چندان واضح نیست چرایی این مطلب است.

۱۷-۳- لیست‌ها به عنوان مجموعه

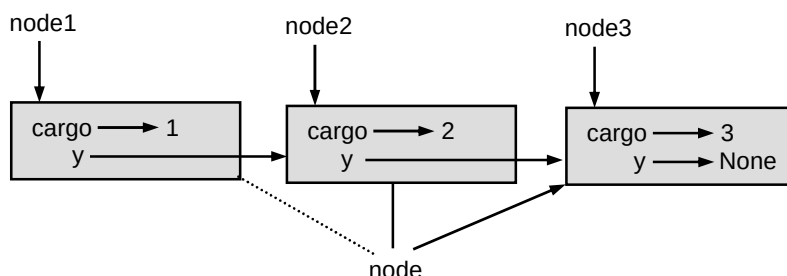
لیست‌ها بسیار مفیدند زیرا راهی را برای جمع‌آوری اشیاء چندگانه درون یک نهاد واحد فراهم می‌کنند. این نهادهای واحد گاهی اوقات مجموعه نامیده می‌شوند. در مثال، اولین گره از لیست به عنوان ارجاعی به کل لیست به کار می‌رود. برای فرستادن لیست به عنوان پارامتر، تنها باید آدرسی به اولین گره را به تابع بفرستیم. برای مثال، تابع **printList** یک گره واحد را به عنوان آرگومان می‌گیرد، از آن گره به عنوان رأس لیست شروع می‌کند و تا آخر لیست یکی یکی بار گره‌ها را چاپ می‌کند.

```
def printList(node):
    while node:
        print node,
        node = node.next
    print
```

برای احضار این متد ما آدرسی به اولین گره را به تابع می‌فرستیم.

```
>>> printList(node1)
1 2 3
```

درون **printList** ما آدرسی به اولین گره را داریم، اما هیچ متغیری که به دیگر گره‌ها اشاره کند وجود ندارد. ما مجبوریم برای دستیابی به گره بعد از مقدار **next** هر گره استفاده کنیم. برای پیمایش یک لیست پیوندی، رایج است که از یک متغیر حلقه مانند **node** برای ارجاع به هر یک از گره‌های پی‌درپی استفاده شود. نمودار شکل ۱۷-۳ مقدار **list** و مقادیری که **node** می‌گیرد را نشان می‌دهد.



شکل ۱۷-۳

تمرین ۱۷-۱: به‌طور قراردادی، لیست‌ها اغلب در براکت‌ها با کاماهایی بین عناصرشان چاپ می‌شوند، مثلاً به‌صورت `[1, a2, a3]`. تابع `printList` را طوری تغییر دهید که خروجی را در چنین قالبی تولید کند.

۱۷-۴- لیست‌ها و بازگشت

بیان خیلی از عملیات لیست‌ها با استفاده از متدهای بازگشتی بسیار عادی است. برای نمونه، در زیر یک الگوریتم بازگشتی را برای چاپ یک لیست به‌طور معکوس مشاهده می‌کنید.

۱. لیست را به دو قطعه تفکیک کن: اولین گره (که رأس نامیده می‌شود) و بقیه آن (که دنباله نامیده می‌شود)
۲. دنباله را به‌طور معکوس چاپ کن.
۳. رأس را چاپ کن.

البته در گام ۲ (فراخوانی بازگشتی)، فرض می‌شود که ما راهی برای چاپ یک لیست به‌طور معکوس در اختیار داریم. اما اگر فرض کنیم که فراخوانی بازگشتی کار می‌کند -بر اساس جهش با اطمینان- آنگاه می‌توانیم خودمان را قانع کنیم که این الگوریتم کار می‌کند. تمام آنچه که نیاز داریم یک حالت مبنا و راهی برای اثبات این موضوع است که به‌ازاء هر لیست، ما سرانجام به حالت مبنا می‌رسیم. با در نظر گرفتن تعریف بازگشتی یک لیست یک حالت مبنای معمول، لیستی تهی است که با `None` نمایش داده می‌شود:

```
def printBackward(list):
    if list == None: return
    head = list
    tail = list.next
    printBackward(tail)
    print head,
```

اولین خط به عنوان حالت مبنا رفتار می‌کند که در عمل کاری انجام نمی‌دهد. دو خط بعدی لیست را به رأس و دنباله تفکیک می‌کند و دو خط آخر لیست را چاپ می‌کند. کمایی که در پایان آخرین خط وجود دارد، پایتون را از چاپ خط جدید بعد از هر گره باز می‌دارد. ما همانطور که **printList** را فراخوانی کردیم، این تابع را فراخوانی می‌کنیم:

```
>>> printBackward(node1)
3 2 1
```

نتیجه یک لیست معکوس است.

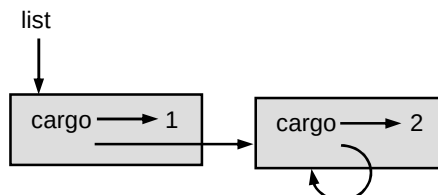
ممکن است تعجب کنید که چرا **printList** و **printBackward** تابع هستند و نه متدهایی در کلاس **Node**.

دلیل این امر آن است که ما می‌خواهیم از **None** برای نمایش یک لیست تهی استفاده کنیم و احضار یک متد بر روی **None** قانونی نیست. این محدودیت توابع را برای نوشتن کدی زیبا و قابل دستکاری به سبک شیء‌گرا نامناسب می‌سازد.

آیا می‌توانیم ثابت کنیم که **printBackward** همیشه پایان می‌یابد؟ به بیانی دیگر آیا همیشه به حالت مبنا می‌رسیم؟ در حقیقت پاسخ منفی است. بعضی از لیست‌ها این متد را خراب می‌کنند.

۱۷-۵- لیست‌های نامتناهی

هیچ راهی برای جلوگیری از گرهی که به یک گره قبل از خود (از جمله خود گره) اشاره می‌کند، وجود ندارد. برای نمونه این شکل لیستی با دو گره را نمایش می‌دهد که یکی از آنها به خودش اشاره می‌کند:



شکل ۱۷-۴

اگر **printList** را بر روی این لیست احضار کنیم، یک حلقه نامتناهی را دور خواهد زد و اگر **printBackward** را احضار کنیم یک بازگشت بی‌انتهای خواهیم داشت. این نوع رفتار کار با لیست‌های متناهی را مشکل می‌سازد.

با این وجود این لیست‌ها گاهی اوقات مفیدند. ممکن است یک عدد را به صورت لیستی از ارقام نشان دهیم و از یک لیست نامتناهی برای نمایش یک کسر مکرر استفاده کنیم.

علیرغم این، گیج‌کننده آن است که ما نمی‌توانیم ثابت کنیم **printList** و **PrintBackward** پایان می‌یابند. بهترین کاری که می‌توانیم انجام دهیم گزاره فرضی زیر است: “اگر لیست شامل حلقه‌ای نباشد، این متدها پایان می‌یابند.” این نوع ادعا یک پیش‌شرط نامیده می‌شود. این گزاره محدودیتی را بر روی یکی از عناصر اعمال می‌کند و رفتار متد را در صورتی که این محدودیت به‌وجود آید توصیف می‌نماید. به زودی مثال‌های بیشتری در این باره خواهید دید.

۱۷-۶- قضیه ابهام بنیادی

ممکن است بخشی از **printBackward** موجب تعجب شود:

```
head = list
tail = list.next
```

بعد از اولین نسبت‌دهی، **head** و **list** دارای یک نوع و یک مقدار می‌باشند. بنابراین چرا ما یک متغیر جدید ساختیم؟

علت آن است که دو متغیر نقش‌های متفاوتی را بازی می‌کنند. ما به **head** به‌عنوان آدرسی به یک گره واحد فکر می‌کنیم و **list** را به‌عنوان آدرسی به اولین گره تلقی می‌کنیم. این نقش‌ها قسمتی از برنامه نیستند، بلکه در ذهن برنامه‌نویس شکل می‌گیرند.

به‌طور کلی ما نمی‌توانیم با نگاه کردن به یک برنامه بگوییم یک متغیر چه نقشی بازی می‌کند. این ابهام می‌تواند مفید باشد، اما همچنین می‌تواند خوانایی برنامه را مشکل سازد. ما اغلب از متغیری به نام **node** و **list** استفاده می‌کنیم، تا به آنچه که قصد داریم متغیر را به آن جهت به کار ببریم، سندیت دهیم. همچنین گاهی اوقات متغیرهایی اضافی برای برطرف کردن ابهام می‌سازیم.

ما می‌توانستیم **printBackward** را بدون **head** و **tail** هم بنویسیم، که در این صورت کد مختصرتر می‌شد اما ممکن است این روش چندان واضح نباشد:

```
def printBackward(list):
    if list == None: return
    printBackward(list.next)
    print list,
```

به دو فراخوانی تابع نگاه کنید. ما باید به خاطر داشته باشیم که `printBackward` با آرگومان‌هایش به عنوان یک مجموعه رفتار می‌کند و `print` آرگومان‌هایش را به عنوان یک شیء واحد تلقی می‌کند.

قضیهٔ ابهام بنیادی، ابهامی را شرح می‌دهد که به طور ذاتی در ارجاع به یک گره وجود دارد:

متغیری که به یک گره اشاره می‌کند ممکن است با گره به عنوان یک شیء واحد و یا اولین گره در لیستی از گره‌ها رفتار کند.

۱۷-۷- تغییر دادن لیست‌ها

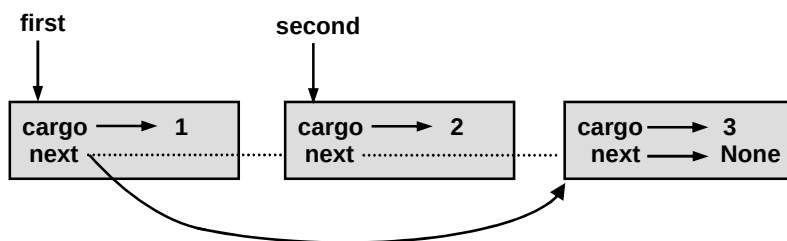
دو راه برای تغییر یک لیست پیوندی وجود دارد. بدیهی است که ما می‌توانیم بار (اطلاعات داخل هر گره) هر یک از گره‌ها را تغییر دهیم. آنچه جالب‌تر است، جمع کردن، حذف کردن و یا مرتب‌سازی مجدد گره‌ها است. به عنوان مثال، اجازه دهید متدی بنویسیم که گره دوم در لیست را حذف کند و آدرسی به گره حذف شده را بازگرداند:

```
def removeSecond(list):
    if list == None: return
    first = list
    second = list.next
    # make the first node refer to the third
    first.next = second.next
    # separate the second node from the rest of the list
    second.next = None
    return second
```

باز هم از یک متغیر موقت برای خواناتر سازی کد برنامه استفاده کردیم. در اینجا نحوهٔ استفاده از این متد را می‌بینید:

```
>>> printList(node1)
1 2 3
>>> removed = removeSecond(node1)
>>> printList(removed)
2
>>> printList(node1)
1 3
```

این نمودار حالت، مفهوم عملیات را نشان می‌دهد:



شکل ۱۷-۵

اگر شما این متد را احضار کنید و یک لیست تک‌عضوی به آن بفرستید چه اتفاقی می‌افتد؟ اگر یک لیست خالی را به‌عنوان آرگومان به آن بفرستید چه روی می‌دهد؟ آیا پیش‌شرطی برای این متد وجود دارد؟ اگر چنین است، متد را برای جداسازی خطای پیش‌شرط از یک راه معقول تصحیح کنید.

۱۷-۸- بسته‌سازها و کمک‌کننده‌ها

تقسیم عملیات یک لیست در دو متد اغلب مفید است. مثلاً برای چاپ معکوس یک لیست در قالب مرسوم لیست `[3, 2, 1]` می‌توانیم از متد `printBackward` برای چاپ `3, 2, 1` استفاده کنیم اما به یک متد جداگانه برای چاپ براکت‌ها و گره اول نیاز داریم. بیا یاد نام آن را `printBackwardNicely` بگذاریم:

```
def printBackwardNicely(list):
    print "[",
    if list != None:
        head = list
        tail = list.next
        printBackward(tail)
        print head,
    print "]",
```

باز هم خوب است که متدهایی شبیه این را بررسی کنیم تا ببینیم با موارد ویژه‌ای چون یک لیست خالی و یا یک لیست کار می‌کند یا نه.

وقتی که ما این متد را در جای دیگری از برنامه استفاده می‌کنیم، مستقیماً `printBackwardNicely` را احضار می‌کنیم و این متد `printBackward` را از طرف ما احضار می‌کند. از آن جهت `printBackwardNicely` به‌عنوان یک بسته‌ساز رفتار می‌کند و `printBackward` را به‌عنوان یک کمک‌کننده به‌کار می‌برد.

۱۷-۹- کلاس LinkedList

چند مشکل ظریف در روشی که برای پیاده‌سازی لیست‌ها داشتیم، وجود دارد. در نقض علت و معلول، ما در ابتدا یک روش پیاده‌سازی جایگزین را پیشنهاد می‌کنیم و سپس توضیح می‌دهیم که این روش چه مشکلاتی را حل می‌کند.

نخست یک کلاس جدید به نام **LinkedList** می‌سازیم. مشخصه‌های این کلاس یک عدد صحیح که شامل طول لیست است و یک آدرس به گره اول می‌باشد. اشیاء **LinkedList** به‌عنوان ابزارهایی برای دستکاری لیست‌هایی از اشیاء **Node** به کار می‌روند:

```
class LinkedList:
    def __init__(self):
        self.length = 0
        self.head = None
```

یک نکته جالب درباره کلاس **LinkedList** این است که این کلاس یک محل طبیعی برای قرار دادن توابع بسته‌سازی نظیر **printBackwardNicely** فراهم می‌کند که ما می‌توانیم متدی از کلاس **LinkedList** بسازیم:

```
class LinkedList:
    ...
    def printBackward(self):
        print "[",
        if self.head != None:
            self.head.printBackward()
        print "]",

class Node:
    ...
    def printBackward(self):
        if self.next != None:
            tail = self.next
            tail.printBackward()
        print self.cargo,
```

تنها برای پیچیده‌تر کردن ماجرا، ما **printBackwardNicely** را تغییر نام دادیم. حال دو متد به نام **printBackward** وجود دارد؛ یکی در کلاس **Node** (کمک کننده) و یکی در کلاس **LinkedList** (بسته‌ساز). وقتی بسته‌ساز، **self.head.printBackward**، را احضار می‌کند، کمک‌کننده را احضار کرده است، زیرا **self.head** یک شیء **Node** است.

مزیت دیگر کلاس **LinkedList** آن است که جمع و حذف کردن اولین عضو لیست را آسان‌تر می‌سازد. برای مثال **addFirst** متدی برای **LinkedList**‌ها است. این متد یک عضو بار را می‌گیرد و در آغاز لیست قرار می‌دهد:

```
class LinkedList:
    ...
    def addFirst(self, cargo):
        node = Node(cargo)
        node.next = self.head
        self.head = node
        self.length = self.length + 1
```

به‌طور معمول، شما باید چنین کدی را بررسی کنید تا ببینید آیا می‌تواند در حالت‌های ویژه هم کار کند. برای نمونه، اگر لیست در آغاز تهی باشد چه اتفاقی می‌افتد؟

۱۷-۱۰- نامتغیرها

بعضی از لیست‌ها “خوش‌فرم” هستند و برخی دیگر نه. برای مثال اگر لیستی شامل یک حلقه باشد، منجر به خراب شدن بسیاری از متدهای ما می‌شود. بنابراین ممکن است بخواهیم که لیست‌ها شامل هیچ حلقه‌ای نباشند. نیاز دیگر ما این است که مقدار **length** در شیء **LinkedList** باید برابر با تعداد واقعی گره‌ها در لیست باشد.

نیازمندی‌هایی همچون اینها **نامتغیر** نامیده می‌شوند، زیرا در شرایط مطلوب آنها باید برای هر شیء تمام مدت درست باشند. مشخص کردن نامتغیرها برای اشیاء یک تمرین برنامه‌نویسی مفید است، زیرا آنها درستی کد برنامه را ثابت می‌کنند، صحت ساختمان‌های داده را بررسی می‌کنند و خطاها را شناسایی می‌نمایند.

چیزی که گاهی اوقات درباره نامتغیرها گیج‌کننده است، این است که زمان‌هایی وجود دارد که آنها مختل می‌شوند. برای مثال در وسط **addFirst** بعد از اینکه ما گرهی را اضافه کردیم و قبل از اینکه **length** را افزایش دهیم، نامتغیر مختل شده است. این نوع اختلال پذیرفتنی است؛ در واقع اغلب اوقات، تغییر یک شیء بدون اینکه نامتغیر حتی برای مدت کوتاهی تغییر کند، غیرممکن است. به‌طور معمول لازم است هر متدی که نامتغیر را مختل می‌کند دوباره آن را به حال اول برگرداند.

embedded reference (آدرس‌های توکار، ارجاع‌های توکار)

آدرسی که در مشخصه‌ای از یک شیء ذخیره شده است.

linked list (لیست پیوندی)

ساختمان داده‌ای که یک مجموعه را با استفاده از دنباله‌ای از گره‌های پیوسته پیاده‌سازی می‌کند.

node (گره)

عضوی از لیست، معمولاً پیاده‌سازی شده به عنوان شیئی که شامل آدرسی به شیء دیگر و از نوع مشابه است.

cargo (بار)

قلم داده‌ای در یک گره.

link (پیوند)

آدرس توکاری که برای پیوستن یک شیء به شیئی دیگر به کار می‌رود.

precondition (پیش شرط)

ادعایی (عبارتی) که برای درست کار کردن یک متد باید صحیح باشد.

fundamental ambiguity theorem (قضیه ابهام بنیادی)

یک آدرس به گرهی از لیست، می‌تواند به عنوان شیئی واحد یا اولین گره در لیست گره‌ها رفتار کند.

wrapper (بسته‌ساز)

متدی که به عنوان حد واسطی میان یک متد فراخوان و کمک‌کننده رفتار می‌کند، اغلب متد را برای احضار ساده‌تر و یا کم‌خطاتر می‌سازد.

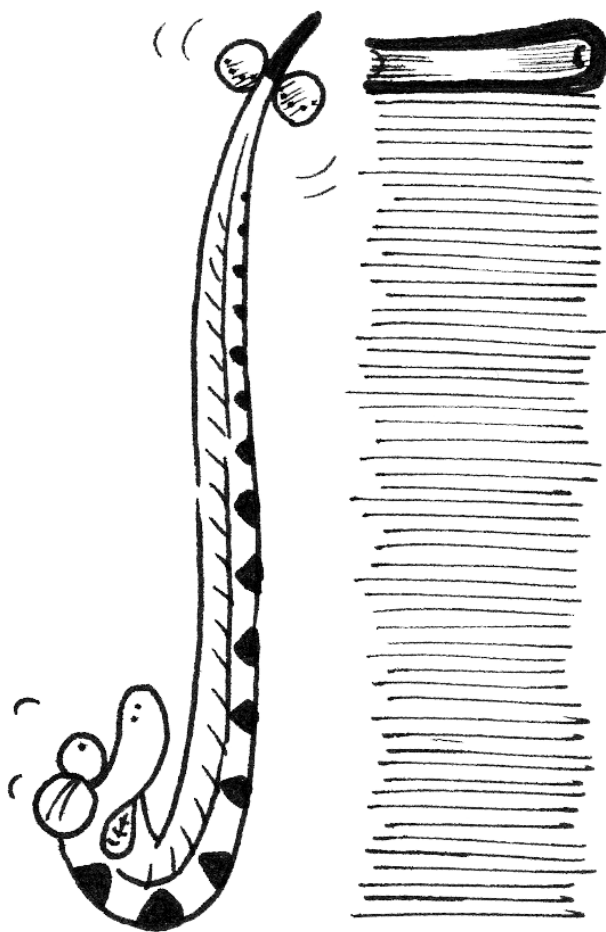
helper (کمک‌کننده)

متدی که توسط فراخوان مستقیماً احضار نمی‌شود، اما بوسیلهٔ یک متد دیگر برای انجام بخشی از یک عملیات استفاده می‌شود.

invariant (نامتغیر)

ادعایی (عبارتی) که برای یک شیء در هر زمان باید درست باشد (مگر در مواقعی که شیء احتمالاً تغییر کرده باشد).

پشته‌ها



از ابتدای کتاب تاکنون با انواع مختلف داده‌ای آشنا شده‌اید. گروهی از انواع داده‌ای پیش‌ساخته پایتون را به کار بردید و حتی یاد گرفتید که چگونه انواع داده‌ای دلخواهی را بسازید. در این فصل شما را با نوع دیگری از انواع داده‌ای آشنا می‌کنیم و در ادامه، یک ساختمان داده‌ای دیگر را مورد بررسی قرار می‌دهیم.

۱۸-۱- نوع داده‌ای انتزاعی

نوع داده‌هایی که تاکنون دیده‌اید همگی واضح و مشخص هستند. (به این معنی که ما دقیقاً نحوه پیاده‌سازی آنها را مشخص می‌کردیم.) برای مثال، کلاس **Card** یک کارت را با استفاده از دو عدد صحیح نمایش می‌دهد. همانطور که در تمام مدت بحث کردیم، این تنها راه نمایش یک کارت نیست و راه‌های زیادی برای انجام این کار وجود دارد.

یک نوع داده‌ای انتزاعی یا **ADT (Abstract Data Type)** مجموعه‌ای از عملیات و مفهوم عملیات (اینکه چه کاری انجام می‌دهند) را مشخص می‌کند اما پیاده‌سازی عملیات را نشان نمی‌دهند. این چیزی است که آنها را خلاصه و انتزاعی می‌سازد.

چرا این نوع داده‌ای مفید است؟

- اگر بتوانید عملیاتی که نیاز دارید را مشخص نمایید، بدون اینکه مجبور باشید در مورد نحوه اجرای عملیات در همان زمان فکر کنید این نوع داده‌ای وظیفه تعیین الگوریتم را ساده می‌سازد.
- از آنجا که معمولاً راه‌های زیادی جهت پیاده‌سازی یک **ADT** وجود دارد، ممکن است نوشتن یک الگوریتم که بتواند با تمام اجراهای ممکن استفاده شود، مفید باشد.
- **ADT**های مشهور نظیر پشته در این فصل معمولاً در کتابخانه‌های استاندارد پیاده‌سازی شده‌اند، بنابراین می‌توانند یک بار نوشته شوند و توسط برنامه‌نویسان زیادی مورد استفاده قرار گیرند.
- عملیات بر روی **ADT**ها، یک زبان سطح بالای مشترک برای تعیین الگوریتم‌ها و صحبت در مورد آنها ارائه می‌دهند. وقتی در مورد **ADT**ها صحبت می‌کنیم، اغلب میان کدی که **ADT** را به کار می‌برد - کد مشتری - با کدی که **ADT** را پیاده‌سازی می‌کند - کد فراهم‌کننده - تفاوت قائل می‌شویم.

۱۸-۲- پشته

در این فصل به یک **ADT** رایج به نام پشته می‌پردازیم. یک پشته شامل یک مجموعه است، یعنی ساختمان داده‌ای که از عناصر چندگانه تشکیل شده است. مجموعه‌های دیگری که تاکنون دیده‌ایم شامل دیکشنری‌ها و لیست‌ها بوده‌اند. یک **ADT** به وسیله عملیاتی که می‌تواند روی آن اجرا شود و **واسط** نام دارد، تعریف شده است. واسط برای یک پشته شامل عملیات زیر است:

__init__: مقداردهی اولیه یک پشته تهی جدید.

push: اضافه کردن یک عنصر جدید به پشته.

pop: حذف و برگرداندن یک عنصر از پشته. عنصری که برگردانده می‌شود همیشه آخرین عنصر اضافه شده است.

isEmpty: بررسی اینکه آیا پشته تهی است یا خیر.

یک پشته گاهی "**LastsIn First Out**" یا یک ساختمان داده‌ای **LIFO** نامیده می‌شود، زیرا آخرین عنصر اضافه شده به پشته اولین عنصری است که حذف می‌شود.

۱۸-۳- پیاده‌سازی پشته‌ها با لیست‌های پایتون

عملیات لیستی که پایتون ارائه می‌دهد شبیه عملیاتی است که یک پشته را تعریف می‌کند. واسط دقیقاً آنچه که باید باشد نیست اما می‌توانیم کدی برای ترجمه **ADT** پشته به عملیات از پیش ساخته بنویسیم.

این کد یک پیاده‌سازی از **ADT** پشته نامیده می‌شود. در کل یک پیاده‌سازی مجموعه‌ای از متدهایی است که نیازهای نحوی و معنایی یک واسط را برطرف می‌کند.

در اینجا یک پیاده‌سازی از **ADT** پشته را می‌بینید که از لیست پایتون استفاده می‌کند:

```
class Stack:
    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def isEmpty(self):
        return (self.items == [])
```

یک شیء پشته شامل مشخصه‌ای به نام **items** است که لیستی از عناصر درون پشته را در بر دارد. متد مقداردهی اولیه **items** را لیست تهی قرار می‌دهد.

برای وارد کردن یک عنصر جدید به پشته، **push** آن را به **items** اضافه می‌کند. جهت برداشتن یک عنصر از پشته **pop** از متد لیست هم‌نامی برای حذف و برگرداندن آخرین عنصر روی لیست استفاده می‌کند.

در نهایت به منظور بررسی تهی بودن پشته **isEmpty**، **items** را با لیستی تهی مقایسه می‌کند.

یک پیاده‌سازی شبیه به این که متدهای درون آن شامل احضارهای ساده‌ای بر روی متدهای موجود هستند، یک **ونیر** نامیده می‌شود. در زندگی واقعی، ونیر پوشش نازکی از چوب با کیفیت خوب است که در مبلمان‌سازی برای پوشاندن چوب با کیفیت کمتر در زیر آن استفاده می‌شود. متخصصین کامپیوتر از این استعاره به منظور توصیف قطعه کوچکی از کد که جزئیات یک پیاده‌سازی را پنهان می‌سازد و واسط ساده‌تر و استانداردتری فراهم می‌کند، استفاده می‌نمایند.

۱۸-۲ گذاشتن و برداشتن عناصر در پشته‌ها

هر پشته یک **ساختمان داده‌ای همه‌پوش (generic)** است. یعنی ما می‌توانیم عناصری از هر نوع داده‌ای را به آن اضافه کنیم. مثال زیر دو عدد صحیح و یک رشته را روی پشته قرار می‌دهد:

```
>>> s = Stack()
>>> s.push(54)
>>> s.push(45)
>>> s.push("+")
```

ما می‌توانیم با استفاده از **isEmpty** و **pop** تمام عناصر پشته را حذف کرده و چاپ نماییم:

```
while not s.isEmpty():
    print s.pop(),
```

خروجی ۵۴ ۴۵ + است. به بیان دیگر ما تنها از پشته جهت چاپ عناصر به طور معکوس استفاده کردیم. مسلماً این قالب استاندارد برای چاپ یک لیست نیست، اما انجام این کار با استفاده از یک پشته به‌طور چشمگیری ساده بود. شما می‌توانید این کد کوتاه را با پیاده‌سازی **printBackward** از بخش ۱۷-۴ مقایسه کنید. در اینجا یک تشابه و توازن طبیعی میان نسخه بازگشتی **printBackward** و الگوریتم پشته وجود دارد، اما تفاوت این است که **printBackward** از یک پشته زمان اجرا برای نگه داشتن رد گره‌ها در طول پیمایش لیست

استفاده می‌کند و سپس آنها را به روش بازگشت از انتها چاپ می‌کند. الگوریتم پشته، عمل یکسانی انجام می‌دهد، با این تفاوت که به جای استفاده از پشته زمان اجرا از یک شیء **Stack** استفاده می‌کند.

۱۸-۵- استفاده از یک پشته برای ارزیابی postfix

در اغلب زبان‌های برنامه‌نویسی، عبارات محاسباتی ریاضی به صورتی نوشته می‌شوند که عملگر عملگر میان دو عملوند قرار دارد مانند $2 + 1$. این غالب **infix** نامیده می‌شود. روش دیگری که به وسیله برخی از ماشین‌حساب‌ها به کار می‌رود، **postfix** نام دارد. در روش **postfix**، عملگر پس از عملوندها منظور می‌شود، نظیر $1\ 2\ +$.

علت اینکه روش **postfix** گاهی مفید واقع می‌شود این است که یک راه طبیعی برای ارزیابی عبارت **postfix** با استفاده از پشته وجود دارد:

- با شروع از ابتدای عبارت، در هر بار یک جمله (عملگر یا عملوند) را بگیرید.
- اگر جمله یک عملوند باشد، آن را به روی پشته **push** کنید.
- اگر جمله یک عملگر باشد دو عملوند را از روی پشته بردارید و عملیات را بر روی آنها اجرا کنید و نتیجه را روی پشته اضافه کنید.
- هنگامی که به پایان عبارت رسیدید، باید دقیقاً یک عملوند در پشته باقی‌مانده باشد. آن عملوند نتیجه عبارت است.

تمرین ۱۸-۱: این الگوریتم را روی عبارت $1\ 2\ +\ 3\ *$ اعمال کنید.

این مثال یکی از مزایای **postfix** را نمایش می‌دهد: جهت کنترل ترتیب عملیات نیازی به استفاده از پرانتز نداریم. برای گرفتن نتیجه یکسان، در روش **infix** مجبوریم عبارت را به صورت $(1+2)*3$ بنویسیم.

تمرین ۱۸-۲: یک عبارت **postfix** بنویسید که معادل $1 + 2 * 3$ باشد.

۱۸-۶- تجزیه

برای پیاده کردن الگوریتم قبلی لازم است بتوانیم یک رشته را پیمایش کنیم و آن را به عملوند و عملگرها تقسیم نماییم. این فرایند نمونه‌ای از **تجزیه** است و نتایج -قطعات غیرقابل تجزیه رشته-، **توکن** (نشان، نماد) نامیده می‌شود. ممکن است این کلمات را از فصل ۱ به خاطر آورید.

پایتون متدی به نام **split** را در ماژول‌های **string** و **re (regular expression)** ارائه می‌دهد. تابع **string.split** یک رشته را با استفاده از یک کاراکتر واحد (**delimiter**) به عنوان حائل در یک لیست تجزیه می‌کند. برای مثال:

```
>>> import string
>>> string.split("Now is the time", " ")
['Now', 'is', 'the', 'time']
```

در این مورد حائل یک کاراکتر فضای خالی است. بنابراین رشته از محل فواصل تکه‌تکه شده است.

تابع **re.split** قوی‌تر است زیرا به ما اجازه می‌دهد یک عبارت باقاعده را به جای یک حائل ارائه دهیم. یک عبارت باقاعده راهی برای مشخص کردن یک مجموعه از رشته‌ها است. برای مثال **[A-z]** مجموعه تمام حروف است و **[0-9]** مجموعه‌ای از تمام اعداد است. عملگر **^** یک مجموعه را نفی می‌کند، بنابراین **[^0-9]** مجموعه‌ای شامل همه چیز به جز اعداد است. این دقیقاً مجموعه‌ای است که ما برای تفکیک عبارات **postfix** نیاز داریم:

```
>>> import re
>>> re.split("[^0-9]", "123+456*/")
['123', '+', '456', '*', '/', '']
```

توجه کنید که ترتیب آرگومان‌ها با **string.split** متفاوت است. جدا کننده قبل از رشته می‌آید.

لیست حاصله شامل عملوندهای **123** و **456** و عملگرهای ***** و **/** است. همچنین دو رشته تهی دارد که پس از عملگرها وارد شده‌اند.

۱۸-۷- ارزیابی روش postfix

به منظور ارزیابی یک عبارت **postfix** ما از تجزیه کننده بخش قبل و الگوریتم مربوط به بخش قبل از آن (دو بخش پیش) استفاده می‌کنیم. برای حفظ سادگی، کار را با یک ارزیاب که تنها عملگرهای **+** و ***** را اجرا می‌کند، آغاز می‌نماییم:

```
def evalPostfix(expr):
    import re
    tokenList = re.split("([^\0-9])", expr)
    stack = Stack()
    for token in tokenList:
        if token == '' or token == ' ':
            continue
        if token == '+':
            sum = stack.pop() + stack.pop()
            stack.push(sum)
        elif token == '*':
            product = stack.pop() * stack.pop()
            stack.push(product)
        else:
            stack.push(int(token))
    return stack.pop()
```

شرط اول فواصل خالی و رشته‌های تهی را بررسی می‌کند و دو شرط بعدی عملگرها را کنترل می‌کنند. در حال حاضر فرض می‌کنیم هر کاراکتر دیگر باید یک عملوند باشد. البته بهتر است ورودی‌های نادرست را بررسی نماییم و پیغام خطایی گزارش دهیم. اما در آینده به این موضوع می‌پردازیم.

بیایید این کد را با ارزیابی فرم **postfix** عبارت **(56+47)*2** آزمایش نماییم:

```
>>> print evalPostfix ("56 47 + 2 *")
206
```

به اندازه کافی دقیق است.

۱۸-۸- فراهم‌گرها و مشتری‌ها

یکی از اهداف اساسی هر **ADT** جدا کردن علایق فراهم‌گر -یعنی کسی که کد اجراکننده **aADT** را می‌نویسد- و مشتری -یعنی کسی که از **ADT** استفاده می‌کند- است. تنها نگرانی فراهم‌گر صحت پیاده‌سازی با توجه به خصوصیت **ADT** است و نه چگونگی استفاده از آن. در مقابل مشتری فرض می‌کند که پیاده‌سازی **ADT** درست است و در مورد جزئیات نگرانی ندارد. هنگامی که شما از انواع داده‌ای پیش‌ساخته پایتون استفاده می‌کنید، از اندیشیدن به‌عنوان صرفاً یک مشتری لذت می‌برید.

البته، هنگامی که شما یک **ADT** را پیاده‌سازی می‌کنید، باید کد مشتری را هم برای امتحان کردن آن بنویسید. در آن مورد شما هر دو نقش را بازی می‌کنید که ممکن است گیج‌کننده باشد. شما باید مقداری سعی کنید تا نقشی که در هر لحظه بازی می‌کنید، حفظ نمایید.

۱۸-۹- واژه‌نامه

abstract data type (ADT) (نوع داده‌ای انتزاعی)

یک نوع داده‌ای (اغلب مجموعه‌ای از اشیاء) که به وسیله دسته‌ای از عملیات تعریف شده، اما می‌تواند به طرق گوناگون پیاده‌سازی و اجرا شود.

client (مشتری)

یک برنامه (یا نویسنده کد) که از یک **ADT** استفاده می‌کند.

provider (فراهم‌گر)

یک کد (یا نویسنده آن) که یک **ADT** را پیاده‌سازی می‌کند.

stack (پشته)

یک نوع **ADT** که در آن آخرین عضوی که وارد می‌شود، اولین عضو خروجی است.

interface (واسط، رابط)

مجموعه عملیاتی که یک **ADT** را تعریف می‌کند.

implementation (پیاده‌سازی)

کدی که نیازمندی‌های نحوی و معنایی را برای واسط تأمین می‌کند.

veneer (ونیر)

تعریفی از کلاس که یک **ADT** را با تعاریف متد که خود احضارهایی از متدهای دیگر (گاهی هم با جابجایی ساده‌تر) هستند، پیاده‌سازی می‌کند. ونیر کار خاص باارزشی انجام نمی‌دهد، اما رابطی که مشتری می‌بیند را بهبود می‌بخشد و استاندارد می‌کند.

generic data structure (ساختمان داده عمومی)

یک نوع ساختمان داده که می‌تواند هر نوع داده‌ای را شامل شود.

infix

یک راه نوشتن عبارات ریاضی که در آن از عملگرها در بین عملوندها استفاده می‌شود.

postfix

یک راه نوشتن عبارات ریاضی که در آن از عملگرها پس از عملوندها استفاده می‌شود.

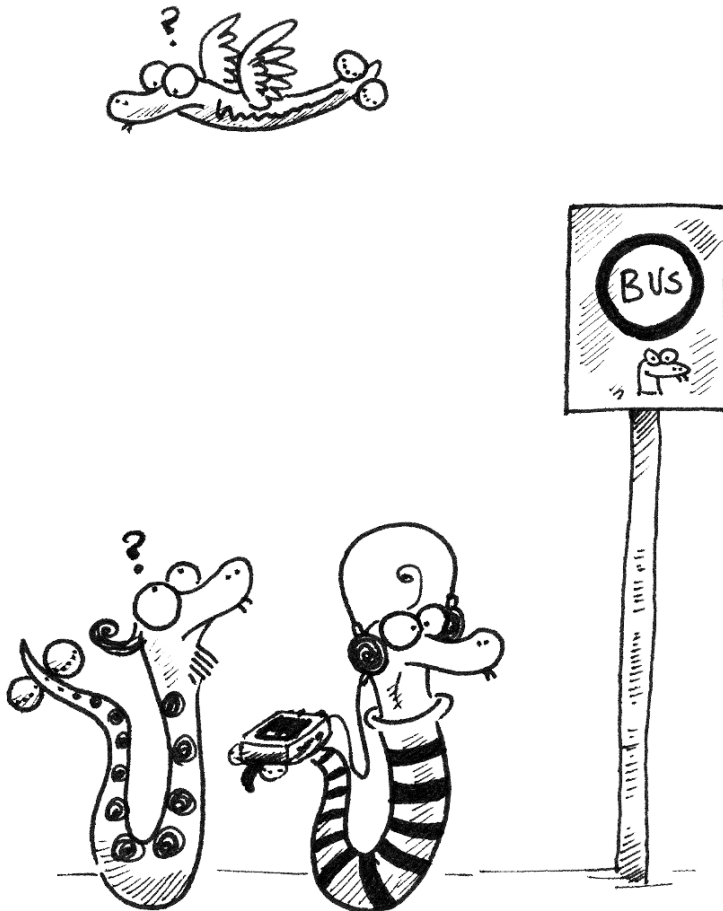
parse (تجزیه)

خواندن رشته‌ای از کاراکترها یا توکن‌ها و تحلیل ساختار دستوری آنها.

token (توکن)

مجموعه‌ای از کاراکترها که برای اهداف تجزیه به عنوان یک واحد عمل می‌کند؛ مانند کلمات در زبان طبیعی.

صفها



در فصل گذشته با **ADT** آشنا شدید و یک **ADT** مشهور به نام پشته را مورد بررسی قرار دادید. در این فصل دو **ADT** دیگر ارائه می‌شود: **صف** و **صف اولویت**. در زندگی واقعی، صف ردیفی است از مشتری‌هایی که در انتظار نوع خاصی از خدمات هستند. در اغلب موارد نفر اول صف، مشتری بعدی سرویس‌گیرنده خواهد بود. در فرودگاه‌ها، مسافرانی که پروازشان زودتر است گاهی از میان صف بیرون کشیده می‌شوند. در سوپرمارکت‌ها یک مشتری مؤدب ممکن است به شخصی که اجناس کمی برای خرید دارد اجازه دهد که کارش را زودتر انجام دهد.

قاعده‌ای که تعیین می‌کند چه کسی نفر بعدی است، **سیاست صف‌بندی** و ساده‌ترین سیاست صف‌بندی **FIFO** نامیده می‌شود (**First In, First Out**). کلی‌ترین سیاست صف‌بندی، صف‌بندی اولویتی است. در این صف‌بندی به هر مشتری اولویتی داده شده است و مشتری با اولویت بالاتر صرف‌نظر از ترتیب ورودش اول مرخص می‌شود. ما می‌گوییم این کلی‌ترین سیاست است، چرا که اولویت می‌تواند بر اساس هر چیزی باشد؛ چه زمانی پرواز، فرودگاه را ترک می‌کند، مشتری چه مقدار جنس برای خرید دارد و یا مشتری چقدر مهم است. البته تمام سیاست‌های صف‌بندی «عادلانه» نیستند، اما انصاف از دید بیننده معنا دارد.

ADT صف و **ADT** صف اولویت، مجموعه‌ای عملیات مشابهی دارند. تفاوت در معنای عملیات است: یک صف از سیاست **FIFO** استفاده می‌کند و صف اولویت (همان‌طور که از نامش پیدا است) از سیاست صف‌بندی اولویتی استفاده می‌کند.

۱۹-۱- ADT صف

ADT صف با عملیات زیر تعریف شده است:

- __init__**: یک صف جدید تهی را مقداردهی اولیه می‌کند.
- insert**: یک قلم داده جدید به صف اضافه می‌کند.
- remove**: یک قلم داده را از صف حذف می‌کند و برمی‌گرداند. آن قلم دادای که برگردانده شده، اولین قلم داده‌ای است که اضافه شده بود.
- isEmpty**: بررسی می‌کند که آیا صف تهی است یا نه.

۱۹-۲- صف پیوندی

اولین پیاده‌سازی **ADT** صفی که ما می‌خواهیم ببینیم، **صف پیوندی** نامیده می‌شود، زیرا از یک سری شیء **Node** به هم پیوسته تشکیل شده است. در اینجا تعریف کلاس را می‌بینید:

```
class Queue:
    def __init__(self):
        self.length = 0
        self.head = None

    def isEmpty(self):
        return (self.length == 0)

    def insert(self, cargo):
        node = Node(cargo)
        node.next = None
        if self.head == None:
            # if list is empty the new node goes first
            self.head = node
        else:
            # find the last node in the list
            last = self.head
            while last.next: last = last.next
            # append the new node
            last.next = node
        self.length = self.length + 1

    def remove(self):
        cargo = self.head.cargo
        self.head = self.head.next
        self.length = self.length - 1
        return cargo
```

متدهای `isEmpty` و `remove` معادل متدهای `isEmpty` و `removeFirst` لیست‌های پیوندی هستند. متد `insert` جدید و کمی پیچیده‌تر است.

ما می‌خواهیم یک قلم داده جدید به آخر لیست اضافه کنیم. اگر تهی باشد، ما فقط `head` را برای اشاره به گره جدید تنظیم می‌کنیم.

در غیر این صورت، ما لیست‌ها را تا گره آخر پیمایش می‌کنیم و گره جدید را به آخر لیست ضمیمه می‌کنیم. ما می‌توانیم گره آخر را از جایی که مشخصه `next` آن `None` است، تشخیص دهیم. برای یک شیء `Queue` خوش‌فرم و خوش‌حالت، دو نامتغیر وجود دارد. مقدار `length` باید تعداد گره‌های صف باشد و گره آخر باید یک مشخصه `next` برابر با `None` داشته باشد.

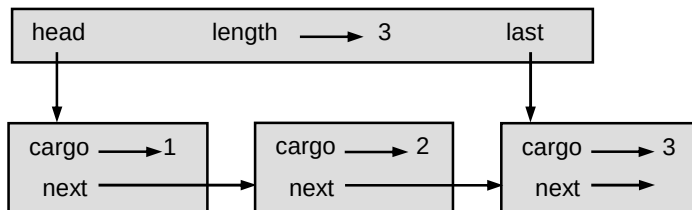
۱۹-۳- ویژگی‌های کارکرد

معمولاً وقتی یک متد را احضار می‌کنیم، نگران جزئیات پیاده‌سازی آن نیستیم. اما ممکن است که یک «نکته ظریف» وجود داشته باشد که بخواهیم بدانیم: ویژگی‌های کارکرد. همچنان که تعداد اقلام درون مجموعه افزایش می‌یابد، مدت زمان اجرا چقدر خواهد بود و به چه صورت تغییر می‌کند؟ ابتدا به **remove** نگاه کنید. هیچ حلقه یا فراخوانی تابعی در اینجا وجود ندارد، یعنی زمان اجرای این متد همواره یکسان است. چنین متدی یک عملیات **ثابت-زمانی** نامیده می‌شود. در حقیقت ممکن است که متد وقتی لیست خالی است اندکی سریع‌تر باشد چراکه از بدنه شرط‌ها عبور می‌کند، اما این تفاوت چندان قابل توجه نیست.

کارکرد **insert** بسیار متفاوت است. در حالت کلی، ما مجبوریم برای پیدا کردن آخرین عضو، لیست را پیمایش کنیم. این پیمایش بسته به طول لیست وقت‌گیر است. از آنجا که زمان اجرا یک تابع خطی از طول است، این متد **زمان-خطی** نامیده می‌شود و در مقایسه با متدهای عملیات ثابت-زمانی بسیار بد است.

۱۹-۴- صف پیوندی اصلاح شده

ما مایلیم یک پیاده‌سازی **ADT** صف بتواند همه عملیات را به صورت ثابت-زمانی انجام دهد. یک راه برای انجام این کار تغییر دادن کلاس صف به‌طوری است که آدرسی به اولین و آخرین گره را همان‌طور که در شکل ۱۹-۱ نمایش داده شده است، نگهداری کند.



شکل ۱۹-۱

پیاده‌سازی صف اصلاح‌شده به این صورت است:

```

class ImprovedQueue:
    def __init__(self):
        self.length = 0
        self.head = None
        self.last = None

    def isEmpty(self):
        return (self.length == 0)

```

تاکنون تنها تغییر مشخصه **last** است که در متدهای **remove** و **insert** استفاده شده است:

```
class ImprovedQueue:
    ...
    def insert(self, cargo):
        node = Node(cargo)
        node.next = None
        if self.length == 0:
            # if list is empty, the new node is head and last
            self.head = self.last = node
        else:
            # find the last node
            last = self.last
            # append the new node
            last.next = node
            self.last = node
        self.length = self.length + 1
```

از آنجا که **last** ردّ آخرین گره را نگه می‌دارد، ما مجبور نیستیم آن را جستجو کنیم. در نتیجه این متد ثابت-زمانی است. برای بدست آوردن این سرعت باید هزینه‌ای پرداخت. ما مجبوریم برای متغیر **last** به **None** در زمانی که آخرین گره حذف شده است به **remove** حالت خاصی را اضافه کنیم:

```
class ImprovedQueue:
    ...
    def remove(self):
        cargo = self.head.cargo
        self.head = self.head.next
        self.length = self.length - 1
        if self.length == 0:
            self.last = None
        return cargo
```

این پیاده‌سازی نسبت به پیاده‌سازی صف پیوندی پیچیده‌تر و اثبات آن هم دشوارتر است. نتیجه این است که ما به هدفمان دست یافتیم؛ هر دو متد **insert** و **remove** عملیات ثابت-زمانی هستند.

تمرین ۱۹-۱: یک پیاده‌سازی از **ADT** صف را با استفاده از لیست‌های پاتون بنویسید. کارایی این پیاده‌سازی را با کلاس **ImprovedQueue** برای محدوده‌ای از طول‌های صف مقایسه کنید.

۱۹-۵- صف اولویت

ADT صف اولویت رابط مشابهی به عنوان **ADT** صف دارد، اما از لحاظ معنایی متفاوت است. رابط باز هم به صورت زیر است:

__init__: یک صف خالی جدید را مقداردهی می‌کند.
insert: یک عضو جدید به صف اضافه می‌کند.
remove: یک عضو از صف حذف می‌کند و برمی‌گرداند. عضوی که برگردانده می‌شود، آن است که اولویت بالاتری دارد.
isEmpty: بررسی می‌کند که آیا صف خالی است یا نه.

تفاوت معنایی در اینجا است که عضو حذف شونده الزاماً اولین عضو اضافه شده نیست. این عضو، عنصری است که بالاترین اولویت را دارد. اینکه اولویت‌ها چیستند و چگونه با یکدیگر مقایسه می‌شوند به وسیله پیاده‌سازی صف اولویت مشخص نمی‌شود. این موضوع به عنصری که در صف وجود دارند بستگی دارد.

برای مثال اگر اعضای درون صف، اسم‌ها باشند، ما احتمالاً آنها را بر اساس ترتیب حروف الفبا انتخاب می‌کنیم. اگر امتیازهای بازی بولینگ باشد، احتمالاً از بالاترین عدد به سمت پایین‌ترین عدد حرکت می‌کنیم ولی اگر امتیازهای بازی گلف باشند از پایین‌ترین عدد به سمت بالاترین عدد می‌رویم. در طول مدتی که ما می‌توانیم عضوهای داخل صف را با هم مقایسه کنیم، می‌توانیم عضوی را که بالاترین اولویت را دارا است پیدا کرده و حذف کنیم. این پیاده‌سازی صف اولویت، یک لیست پایتون را به عنوان مشخصه دارد که شامل اقلام داخل صف می‌باشد:

```
class PriorityQueue:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def insert(self, item):
        self.items.append(item)
```

متد مقداردهی اولیه، **isEmpty** و **insert** همگی ونیرهایی بر روی عملیات لیست هستند. تنها متد جالب **remove** است:


```
class PriorityQueue:
    ...
    def remove(self):
        maxi = 0
        for i in range(1, len(self.items)):
            if self.items[i] > self.items[maxi]:
                maxi = i
            item = self.items[maxi]
            self.items[maxi:maxi+1] = []
        return item
```

در آغاز هر تکرار، **maxi** اندیس بزرگ‌ترین عضوی را که تاکنون دیده‌ایم (بالاترین اولویت) نگه می‌دارد. در هر بار تکرار حلقه برنامه **i** امین عضو را با عضو منتخب مقایسه می‌کند. اگر عضو جدید بزرگ‌تر است، مقدار **maxi** به **i** تغییر خواهد یافت.

وقتی دستور **for** کامل شد، **maxi** اندیس بزرگ‌ترین عضو است. این عضو از لیست حذف شده و برگردانده می‌شود. بیا یاد پیاده‌سازی را آزمایش کنیم:

```
>>> q = PriorityQueue()
>>> q.insert(11)
>>> q.insert(12)
>>> q.insert(14)
>>> q.insert(13)
>>> while not q.isEmpty(): print q.remove()
14
13
12
11
```

اگر صف شامل رشته‌ها و اعداد ساده باشد آنها به ترتیب الفبایی یا شمارشی از زیاد به کم حذف می‌شوند. پایتون می‌تواند بزرگ‌ترین عدد صحیح یا رشته را پیدا کند زیرا آنها را بر اساس عملگرهای مقایسه‌ای پیش‌ساخته می‌سنجند.

اگر صف شامل یک نوع شیئی باشد، مجبور است یک متد **__cmp__** فراهم می‌کند. وقتی که **remove** از عملگر **>** برای مقایسهٔ اقلام استفاده می‌کند، متد **__cmp__** را بر روی یکی از اعضا احضار می‌کند و عضو دیگر را به عنوان پارامتر به آن می‌فرستد. تا زمانی که متد **__cmp__** به درستی کار کند، صف اولویت کار می‌کند.

۱۹-۶- کلاس Golfer

به عنوان یک مثال از شیئی با یک تعریف غیرمعمول از اولویت، بیایید کلاسی به نام **Golfer** را که رتبه اسامی و امتیازهای بازیکنان گلف را می‌دارد، پیاده‌سازی کنیم. به طور معمول، کار را با تعریف `__init__` و `__str__` شروع می‌کنیم:

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

    def __str__(self):
        return "%-16s: %d" % (self.name, self.score)
```

`__str__` از عملکرد قالب‌بندی برای گذاشتن اسامی و امتیازها در ستون‌های مرتب استفاده می‌کند.

سپس ما نسخه‌ای از `__cmp__` را تعریف می‌کنیم که پایین‌ترین امتیاز، بالاترین اولویت را بگیرد. به‌طور معمول، `__cmp__` در صورتی که **self** “بزرگ‌تر از” **other** باشد 1، اگر “کوچک‌تر از” **other** باشد -1 و اگر مساوی با آن باشد 0 را برمی‌گرداند.

```
class Golfer:
    ...
    def __cmp__(self, other):
        if self.score < other.score: return 1 # less is more
        if self.score > other.score: return -1
        return 0
```

حال ما آماده‌ایم که صف اولویت را با کلاس **Golfer** آزمایش کنیم:

```
>>> tiger = Golfer("Tiger Woods", 61)
>>> phil = Golfer("Phil Mickelson", 72)
>>> hal = Golfer("Hal Sutton", 69)
>>>
>>> pq = PriorityQueue()
>>> pq.insert(tiger)
>>> pq.insert(phil)
>>> pq.insert(hal)
>>> while not pq.isEmpty(): print pq.remove()
Tiger Woods: 61
Hal Sutton: 69
Phil Mickelson: 72
```

تمرین ۱۹-۲: یک پیاده‌سازی از **ADT** صف اولویت را با استفاده از یک لیست پیوندی بنویسید. شما باید لیست را به صورت مرتب نگه دارید، به‌طوری که عمل حذف عملیاتی ثابت-زمانی باشد. کارایی این پیاده‌سازی را با پیاده‌سازی لیست‌های پایتون مقایسه کنید.

۱۹-۷- واژه‌نامه

Queue (صف)

یک **ADT** که عملیاتی مشابه با عملیات یک صف واقعی، انجام می‌دهد.

Priority Queue (صف اولویت)

یک **ADT** که در آن عضو که بالاترین اولویت را دارا است، اولین عضوی است که خارج می‌شود.

queueing policy (سیاست صف‌بندی)

قانونی که معین می‌کند عضو بعدی که از لیست حذف می‌شود، کدام است.

FIFO

" **First In First Out** ". یک سیاست صف‌بندی به این صورت که اولین عضو وارد شده اولین عضوی است که خارج می‌شود.

linked queue (صف پیوندی)

پیاده‌سازی یک صف با استفاده از لیست پیوندی.

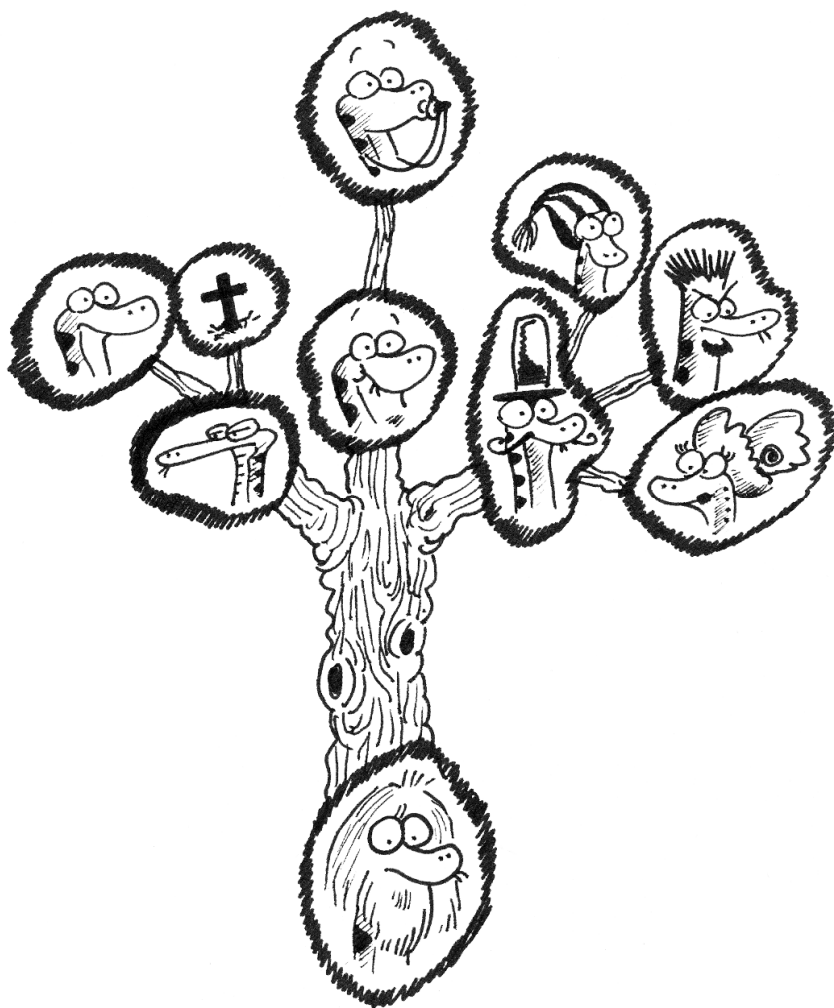
constant time (ثابت-زمانی)

عملیاتی که زمان اجرای آن وابسته به اندازه ساختار داده‌ها نیست.

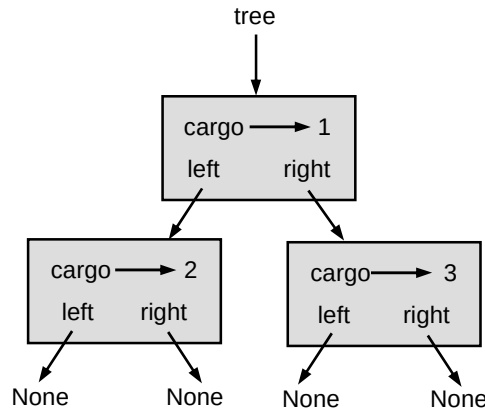
linear time (زمان-خطی)

عملیاتی که زمان اجرای آن یک تابع خطی از اندازه ساختار داده‌ها است.

درخت‌ها



در فصل ۱۷ با ساختمان داده‌ای به نام لیست پیوندی آشنا شدید. درخت‌ها نیز همچون لیست‌های پیوندی از تعدادی گره تشکیل شده‌اند. یک نوع رایج از درخت‌ها، **درخت دودویی** است که در آن هر گره شامل آدرسی به دو گره دیگر (که ممکن است پوچ باشند) می‌باشد. این آدرس‌ها (ارجاع‌ها) به زیردرخت‌های چپ و راست اشاره می‌کنند. گره‌های درخت نیز مانند گره‌های لیست شامل بار (**cargo**) هستند. نمودار حالت برای یک درخت به صورت زیر است:



شکل ۲۰-۱

به منظور جلوگیری از بی نظمی و پیچیدگی شکل، معمولاً **None** را حذف می‌کنیم. بالای درخت (گره‌ی که **tree** به آن اشاره می‌کند) **ریشه** نامیده می‌شود. در جهت حفظ هماهنگی استعاره درخت، گره‌های دیگر شاخه نامیده می‌شوند و گره‌هایی که در انتها قرار گرفته و آدرس‌هایی به **None** دارند، **برگ** نام دارند. ممکن است کمی عجیب به نظر برسد که ما تصویری با ریشه در بالا و برگ‌هایی در پایین رسم کنیم اما این عجیب‌ترین موضوع نیست! متخصصین کامپیوتر -این قالب را- با استعاره دیگری ترکیب می‌کنند: شجره‌نامه خانوادگی. گره فوقانی **پدر** یا **والد** و گره‌هایی که پدر به آنها اشاره می‌کند **فرزند** نامیده می‌شود. گره‌هایی با پدر واحد، **برادر** نام دارند.

در پایان اینکه، یک لغت‌نامه هندسی برای بحث در مورد درخت‌ها وجود دارد. قبلاً به مفاهیم چپ و راست اشاره کردیم اما مفاهیم دیگری هم تحت عنوان "بالا" (برای والد/ریشه) و "پایین" (برای فرزندان/برگ‌ها) وجود دارد. همچنین تمام گره‌هایی که فاصله یکسانی از ریشه دارند یک **سطح** از درخت را تشکیل می‌دهند.

ممکن است برای صحبت در مورد درخت‌ها به سه اصطلاح نیاز نداشته باشیم اما به هر حال این اصطلاح‌ها به کار می‌روند.

درخت‌ها هم مثل لیست‌های پیوندی دارای ساختمان‌های داده‌ای بازگشتی هستند، زیرا آنها به صورت بازگشتی تعریف شده‌اند.
یک درخت یا:

- درختی تهی که با **None** نمایش داده شده است، یا
- گرهی شامل آدرس شیء و دو آدرس برای اشاره به گره‌های درخت می‌باشد.

۲۰-۱- ساختن درخت‌ها

فرآیند تشکیل یک درخت شبیه به روند ساختن یک لیست پیوندی است. هر احضار سازنده یک تک گره ایجاد می‌کند.

```
class Tree:
    def __init__(self, cargo, left=None, right=None):
        self.cargo = cargo
        self.left = left
        self.right = right

    def __str__(self):
        return str(self.cargo)
```

cargo از هر جنسی می‌تواند باشد، اما پارامترهای **left** و **right** باید از جنس گره‌های درخت باشند. **left** و **right** اختیاری هستند، مقدار پیش‌فرض **None** است.
برای چاپ یک گره، ما تنها بار آن را چاپ می‌کنیم.
یک راه برای ساخت درخت حرکت از سمت پایین به بالا است. ابتدا گره‌های فرزند را نسبت‌دهی کنید:

```
left = Tree(2)
right = Tree(3)
```

سپس گره والد را بسازید و آن را به فرزندان متصل کنید:

```
tree = Tree(1, left, right)
```

می‌توانیم این کد را به وسیلهٔ احضارهای سازنده به صورت تودرتو مختصرتر بنویسیم:

```
>>> tree = Tree(1, Tree(2), Tree(3))
```

از هر دو راه، نتیجه درختی است که در ابتدای فصل دیدیم.

۲-۲۰- پیمایش درخت‌ها

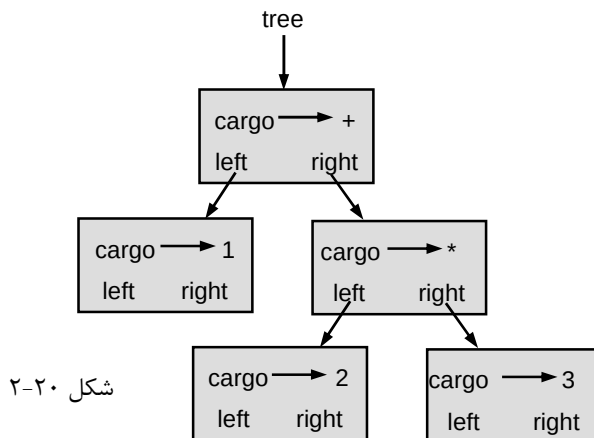
هر زمان که یک ساختمان داده‌ای جدید می‌بینید، اولین سؤال شما باید این باشد: «چگونه آن را پیمایش کنم؟» طبیعی‌ترین راه پیمایش یک درخت روش بازگشتی است. برای مثال، اگر درخت شامل اعداد صحیحی به عنوان بار باشد، این تابع مجموع آنها را برمی‌گرداند:

```
def total(tree):
    if tree == None: return 0
    return total(tree.left) + total(tree.right) + tree.cargo
```

حالت مبنا درختی تهی است که هیچ باری ندارد؛ بنابراین مجموع 0 است. مرحله بازگشتی، دو فراخوانی بازگشتی برای یافتن مجموع درخت‌های فرزند تولید می‌کند. وقتی فراخوانی‌های بازگشتی کامل شد، بار پدر را اضافه می‌کنیم و مجموع کل را برمی‌گردانیم.

۳-۲۰- درخت‌های عبارت

یک درخت راهی طبیعی برای نمایش ساختار یک عبارت است. برخلاف دیگر نمادگذاری‌ها، درخت می‌تواند محاسبات را به‌طور واضح و دقیق نمایش دهد. برای مثال عبارت $1+2*3$ (که یک عبارت infix می‌باشد) مبهم است، مگر اینکه بدانیم ضرب قبل از جمع انجام می‌شود. این درخت گزاره‌ای، همان محاسبه را نمایش می‌دهد:



شکل ۲-۲۰

گره‌های یک درخت عبارت می‌تواند عملوندهایی نظیر 1 و 2 یا عملگرهایی همچون + و * باشد. عملوندها گره‌های برگ هستند؛ گره‌های عملگر شامل آدرس‌هایی به عملوندهایشان می‌باشند. (تمام این عملگرها دودویی هستند، یعنی آنها دقیقاً دو عملوند دارند).

ما می‌توانیم این درخت را به صورت زیر بسازیم:

```
>>> tree = Tree('+', Tree(1), Tree('*', Tree(2), Tree(3)))
```

با توجه به شکل، هیچ سؤال و ابهامی در مورد ترتیب عملگرها وجود ندارد؛ به‌منظور محاسبهٔ عملگر دوم جمع، عمل ضرب اول صورت می‌گیرد. درخت‌های عبارت استفاده‌های زیادی دارند. مثال این فصل از درخت‌ها برای ترجمهٔ عبارات به **prefix**، **postfix** و **infix** استفاده می‌کند. درون کامپایلرها درخت‌های مشابهی برای تجزیه، بهینه‌سازی و ترجمهٔ برنامه‌ها به کار می‌رود.

۲۰-۴- پیمایش درختی

ما می‌توانیم یک درخت عبارت را پیمایش کنیم و محتویات آن را به صورت زیر چاپ کنیم:

```
def printTree(tree):  
    if tree == None: return  
    print tree.cargo,  
    printTree(tree.left)  
    printTree(tree.right)
```

به بیان دیگر، برای چاپ یک درخت، ابتدا محتویات ریشه را چاپ می‌کنیم، سپس سرتاسر زیردرخت سمت چپ و آنگاه سرتاسر زیردرخت سمت راست را چاپ می‌نماییم. این روش پیمایش درخت **preorder** نامیده می‌شود، زیرا محتویات ریشه قبل از محتویات فرزندان نمایش داده می‌شود. خروجی مثال قبل به این صورت است:

```
>>> tree = Tree('+', Tree(1), Tree('*', Tree(2), Tree(3)))  
>>> printTree(tree)  
+ 1 * 2 3
```

این قالب متفاوت از انواع **postfix** و **infix** است. این نوع دیگری از نمادگذاری به نام **prefix** است که در آن عملگرها قبل از عملوندها ظاهر می‌شوند. اگر درخت را به ترتیب دیگری پیمایش کنید، ممکن است به این خروجی شک کنید. چرا که عبارتی با یک نمادگذاری متفاوت به دست می‌آورد.

برای مثال اگر ابتدا زیردرخت‌ها را چاپ کنید و سپس گره ریشه، خواهید داشت:

```
def printTreePostorder(tree):
    if tree == None: return
    printTreePostorder(tree.left)
    printTreePostorder(tree.right)
    print tree.cargo,
```

نتیجه، $1\ 2\ 3\ * +$ ، به فرم postfix است! این ترتیب پیمایش postorder نامیده می‌شود.

در پایان اینکه، جهت پیمایش یک درخت به صورت inorder، درخت سمت چپ، سپس ریشه و آنگاه درخت سمت راست را چاپ می‌نمایید:

```
def printTreeInorder(tree):
    if tree == None: return
    printTreeInorder(tree.left)
    print tree.cargo,
    printTreeInorder(tree.right)
```

حاصل، $1 + 2 * 3$ می‌باشد که همان نمایش infix عبارت است. با دیدی منصفانه، باید متذکر شویم که ما اشکال مهمی را از قلم انداختیم. بعضی مواقع، وقتی یک عبارت infix را می‌نویسیم، مجبوریم برای حفظ ترتیب عملیات از پرانتز استفاده کنیم. بنابراین یک پیمایش inorder برای تولید یک عبارت infix کافی نیست. با این همه با کمی پیشرفت، درخت عبارت و سه پیمایش بازگشتی، روشی کلی برای ترجمه عبارات از یک قالب به قالب دیگر ارائه می‌دهند.

تمرین ۲۰-۱: printTreePreorder را طوری تغییر دهید که دو پرانتز به دور هر عملگر و جفت عملوند آن قرار دهد. آیا خروجی درست و غیر مبهم است؟ آیا پرانتزها همیشه ضروری هستند؟

اگر یک پیمایش inorder انجام دهیم و رده محلی را که در درخت بسر می‌بریم نگه داریم، می‌توانیم نمایشی گرافیکی از یک درخت تولید کنیم:

```
def printTreeIndented(tree, level=0):
    if tree == None: return
    printTreeIndented(tree.right, level+1)
    print ' '*level + str(tree.cargo)
    printTreeIndented(tree.left, level+1)
```

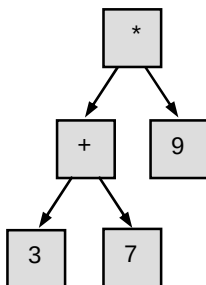
پارامتر **level** موقعیت ما در درخت را ضبط می‌کند. این پارامتر در آغاز به‌طور پیش‌فرض 0 است. هر بار که یک فراخوانی بازگشتی انجام دهیم، **level+1** را ارسال می‌کنیم، چرا که سطح فرزند همیشه یک واحد از سطح والد بزرگ‌تر است. هر عضو به‌وسیلهٔ دو فاصله در هر سطح کنگره‌گذاری می‌شود. نتیجه برای درخت نمونه بدین صورت است:

```
>>> printTreeIndented(tree)
  3
 *
  2
+
  1
```

اگر از پهلوی به خروجی نگاه کنید، نسخهٔ ساده‌شده‌ای از شکل اصلی را می‌بینید.

۵-۲۰- ساختن یک درخت عبارت

در این بخش، عبارات **infix** را تجزیه می‌کنیم و درخت عبارت متناظر با آن را ایجاد می‌کنیم. برای مثال، عبارت **(3+7)*9** درخت زیر را نتیجه می‌دهد:



شکل ۳-۲۰

توجه کنید که ما با حذف اسامی مشخصه‌ها نمودار را ساده‌تر کردیم. تجزیه‌گری که خواهیم نوشت، بر روی عباراتی شامل اعداد، پرانتزها و عملگرهای + و * کار می‌کند. ما فرض می‌کنیم رشتهٔ ورودی قبلاً درون یک لیست پایتون به تعدادی توکن تقسیم شده است. لیست توکن برای عبارت **(3+7)*9** به این صورت است:

```
['(', 3, '+', 7, ')', '*', 9, 'end']
```

توکن **end** مشخص می‌کند که لیست پایان یافته و تجزیه‌گر را از ادامهٔ کار باز می‌دارد.

تمرین ۲۰-۲: تابعی بنویسید که عبارتی را در قالب یک رشته بگیرد و لیستی از توکن‌ها را برگرداند.

اولین تابعی که می‌نویسیم **getToken** است که لیستی از توکن‌ها و یک توکن خاص را به عنوان پارامتر می‌گیرد. این تابع توکن مورد نظر را با اولین توکن در لیست مقایسه می‌کند، اگر با هم برابر بودند آن را حذف می‌کند و مقدار **true** را برمی‌گرداند؛ در غیر این صورت **false** را برمی‌گرداند:

```
def getToken(tokenList, expected):
    if tokenList[0] == expected:
        del tokenList[0]
        return 1
    else:
        return 0
```

از آنجا که **tokenList** به یک شیء تغییرپذیر اشاره می‌کند، تغییراتی که در اینجا به وجود می‌آید از هر متغیر دیگری که به همین شیء اشاره کند، قابل رؤیتند. تابع بعدی، **getNumber**، عملوندها را اداره می‌کند. اگر توکن بعدی در **tokenList** یک عدد باشد، **getNumber** آن را حذف می‌کند و یک گره برگ شامل آن عدد را برمی‌گرداند، در غیر این صورت **None** را برمی‌گرداند:

```
def getNumber(tokenList):
    x = tokenList[0]
    if type(x) != type(0): return None
    del tokenList[0]
    return Tree(x, None, None)
```

قبل از ادامه، باید **getNumber** را به تنهایی آزمایش کنیم. ما لیستی از اعداد را به **tokenList** نسبت می‌دهیم. عضو اول را در می‌آوریم، نتیجه را چاپ می‌کنیم و سپس هر چه که از لیست توکن‌ها باقی مانده بود را چاپ می‌نماییم:

```
>>> tokenList = [9, 11, 'end']
>>> x = getNumber(tokenList)
>>> printTreePostorder(x)
9
>>> print tokenList
[11, 'end']
```

متد دیگری که نیاز داریم، **getProduct** است که یک درخت عبارت برای حاصل‌ضرب‌ها می‌سازد. یک حاصل‌ضرب ساده دو عدد به عنوان عملوند دارد، مثل $3*7$.

در اینجا نسخه‌ای از **getProduct** آمده است که بر روی حاصل ضرب‌های ساده کار می‌کند.

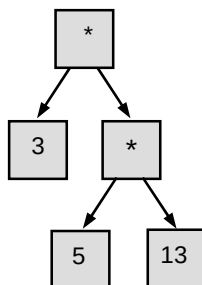
```
def getProduct(tokenList):  
    a = getNumber(tokenList)  
    if getToken(tokenList, '*'):  
        b = getNumber(tokenList)  
        return Tree('*', a, b)  
    else:  
        return a
```

با فرض بر اینکه **getNumber** درست کار می‌کند و یک درخت منحصر به فرد برمی‌گرداند، عملوند اول را به **a** نسبت می‌دهیم. اگر کاراکتر بعدی ***** باشد، عدد دوم را می‌گیریم و با **a** و عملگر یک درخت عبارت تولید می‌کنیم.

اگر کاراکتر بعدی چیز دیگری باشد، آنگاه تنها گره برگ با **a** را برمی‌گردانیم. در اینجا دو مثال آورده شده است:

```
>>> tokenList = [9, '*', 11, 'end']  
>>> tree = getProduct(tokenList)  
>>> printTreePostorder(tree)  
9 11 *  
>>> tokenList = [9, '+', 11, 'end']  
>>> tree = getProduct(tokenList)  
>>> printTreePostorder(tree)  
9
```

مثال دوم به این نکته اشاره دارد که ما یک عملوند واحد را به عنوان نوعی حاصل ضرب تلقی کرده‌ایم. این نوع تعریف از “حاصل ضرب”، غیر شهودی است، اما مفید به نظر می‌رسد. اکنون باید به ضرب‌های پیچیده‌تری نظیر $3*5*13$ بپردازیم. ما این عبارت را به عنوان حاصل ضرب ضرب‌ها در نظر می‌گیریم؛ مثلاً $3*(5*13)$. درخت حاصل به صورت زیر است:



شکل ۲۰-۴

با یک تغییر کوچک در **getProduct**، می‌توانیم بر روی یک ضرب طولانی دلخواه کار کنیم:

```
def getProduct(tokenList):
    a = getNumber(tokenList)
    if getToken(tokenList, '*'):
        b = getProduct(tokenList) # this line changed
        return Tree('*', a, b)
    else:
        return a
```

به بیان دیگر، یک ضرب می‌تواند یک ضرب یگانه باشد و یا یک درخت با * در ریشه، یک عدد در سمت چپ و یک ضرب در سمت راست. این نوع از تعریف بازگشتی باید تمرین شود تا مأنوس به‌نظر برسد.

اجازه دهید نسخه جدید را با یک ضرب پیچیده آزمایش کنیم:

```
>>> tokenList = [2, '*', 3, '*', 5, '*', 7, 'end']
>>> tree = getProduct(tokenList)
>>> printTreePostorder(tree)
2 3 5 7 * * *
```

در ادامه توانایی تجزیه اعمال جمع را اضافه می‌کنیم. باز هم از یک تعریف غیرشهودی “جمع” استفاده می‌نماییم. برای ما، یک جمع می‌تواند درختی باشد با یک + در ریشه، یک ضرب در سمت چپ و یک جمع در سمت راست؛ یا می‌تواند تنها از یک ضرب تشکیل شده باشد. اگر مایلید با این تعریف کار کنید، لازم است بدانید که خصوصیت زیبایی دارد: می‌توانیم هر عبارتی را (بدون پرانتزها) به‌عنوان مجموعی از حاصل ضرب‌ها نمایش دهیم. این ویژگی اساس الگوریتم تجزیه ما است.

getSum سعی می‌کند درختی بسازد که یک ضرب سمت چپ آن و یک جمع سمت راست آن باشد، اما اگر یک علامت + پیدا نکند، تنها حاصل ضرب را تولید می‌کند:

```
def getSum(tokenList):
    a = getProduct(tokenList)
    if getToken(tokenList, '+'):
        b = getSum(tokenList)
        return Tree('+', a, b)
    else:
        return a
```

بیایید آن را با عبارت $9 * 11 + 5 * 7$ آزمایش کنیم:

```
>>> tokenList = [9, '*', 11, '+', 5, '*', 7, 'end']
>>> tree = getSum(tokenList)
>>> printTreePostorder(tree)
9 11 * 5 7 * +
```

کار تقریباً تمام است، اما ما هنوز باید بر روی پرانتزها کارکنیم. در هر کجای یک عبارت که یک عدد بتواند وجود داشته باشد، یک جمع کامل هم می‌تواند درون پرانتز قرار گیرد. ما تنها نیاز داریم **getNumber** را طوری تغییر دهیم که زیر عبارت‌ها را هم پوشش دهد:

```
def getNumber(tokenList):
    if getToken(tokenList, '('):
        x = getSum(tokenList) # get the subexpression
        getToken(tokenList, ')') # remove the closing parenthesis
        return x
    else:
        x = tokenList[0]
        if type(x) != type(0): return None
        tokenList[0:1] = []
        return Tree (x, None, None)
```

بیا این کد را با عبارت $9 * (11 + 5) * 7$ آزمایش کنیم:

```
>>> tokenList = [9, '*', '(', 11, '+', 5, ')', '*', 7, 'end']
>>> tree = getSum(tokenList)
>>> printTreePostorder(tree)
9 11 5 + 7 * *
```

تجزیه‌گر، با پرانتزها به درستی برخورد می‌کند؛ عمل جمع قبل از عمل ضرب صورت می‌گیرد. در نسخه نهایی برنامه، بد نیست نامی به **getNumber** بدهیم که نقش جدید آن را بهتر توصیف کند.

۲۰-۶- اداره کردن خطاها

فرض ما بر این است که عبارت در سرتاسر فرایند تجزیه خوش‌فرم است. برای مثال وقتی به انتهای یک زیر عبارت می‌رسیم، فرض می‌کنیم کاراکتر بعدی یک پرانتز بسته است. اگر خطایی در عبارت وجود داشته باشد و کاراکتر بعدی چیز دیگری باشد، باید آن را مورد بررسی قرار دهیم:

```
def getNumber(tokenList):
    if getToken(tokenList, '('):
        x = getSum(tokenList)
        if not getToken(tokenList, ')'):
            raise 'BadExpressionError', 'missing parenthesis'
        return x
    else:
        # the rest of the function omitted
```

دستور **raise** یک اعتراض می‌سازد؛ در این مورد ما نوعی عبارت جدید به نام **BadExpressionError** می‌سازیم. اگر تابعی که **getNumber** نام دارد و یا یکی از توابع دیگر

در پس‌یابی، اعتراض را کنترل کنند، آنگاه برنامه ادامه می‌یابد. در غیر این‌صورت، پایتون یک پیغام خطا چاپ کرده و خارج می‌شود.

تمرین ۲۰-۳: دیگر محل‌های وقوع خطا در این توابع را پیدا کرده و دستورات **raise** مناسب را اضافه نمایید. برنامه خود را با عبارات نادرست و بدفرم آزمایش کنید.

۷-۲۰- درخت جانوران

در این بخش، ما برنامه کوچکی را توسعه می‌دهیم که از یک درخت برای نمایش یک اصل علمی استفاده می‌کند.

این برنامه برای ساختن درختی از پرسش‌ها و اسامی حیوانات، با کاربر ارتباط متقابلی دارد. در اینجا یک اجرای آزمایشی از این برنامه را می‌بینید:

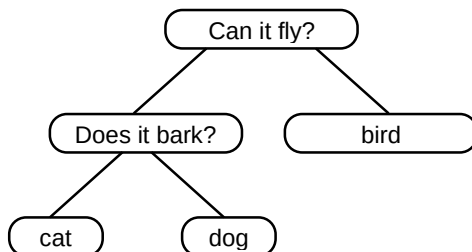
```
Are you thinking of an animal? y
Is it a bird? n
What is the animals name? dog
What question would distinguish a dog from a bird? Can it fly
If the animal were dog the answer would be? n

Are you thinking of an animal? y
Can it fly? n
Is it a dog? n
What is the animals name? cat
What question would distinguish a cat from a dog? Does it bark
If the animal were cat the answer would be? n

Are you thinking of an animal? y
Can it fly? n
Does it bark? y
Is it a dog? y
I rule!

Are you thinking of an animal? n
```

در شکل ۲۰-۵ درختی را که توسط این گفت و گو ساخته می‌شود، ملاحظه می‌کنید:



شکل ۲۰-۵

در آغاز هر دور، برنامه از بالای درخت شروع می‌کند و اولین سؤال را می‌پرسد. بسته به پاسخ، برنامه به فرزند سمت راست یا چپ منتقل می‌شود و تا زمانی که به یک گره برگ برسد این کار را ادامه می‌دهد. در این نقطه برنامه یک حدس می‌زند. اگر حدس نادرست باشد، برنامه نام یک حیوان جدید و سؤالی که حدس نادرست را از حیوان جدید تشخیص می‌دهد، از کاربر می‌پرسد. سپس یک گره با پرسش جدید و حیوان جدید به درخت اضافه می‌کند.

کد برنامه به این صورت است:

```

def animal():
    # start with a singleton
    root = Tree("bird")

    # loop until the user quits
    while 1:
        print
        if not yes("Are you thinking of an animal? "): break

    # walk the tree
    tree = root
    while tree.getLeft() != None:
        prompt = tree.getCargo() + "? "
        if yes(prompt):
            tree = tree.getRight()
        else:
            tree = tree.getLeft()

    # make a guess
    guess = tree.getCargo()
    prompt = "Is it a " + guess + "? "
    if yes(prompt):
        print "I rule!"
        continue

    # get new information
    
```

```

prompt = "What is the animal's name? "
animal = raw_input(prompt)
prompt = "What question would distinguish a %s from a %s? "
question = raw_input(prompt % (animal, guess))

# add new information to the tree
tree.setCargo(question)
prompt = "If the animal were %s the answer would be? "
if yes(prompt % animal):
    tree.setLeft(Tree(guess))
    tree.setRight(Tree(animal))
else:
    tree.setLeft(Tree(animal))
    tree.setRight(Tree(guess))

```

تابع **yes** یک کمک‌کننده است. این تابع پیایی را چاپ می‌کند و یک ورودی از کاربر می‌گیرد. اگر پاسخ با **y** یا **Y** آغاز شود تابع مقدار **true** را برمی‌گرداند:

```

def yes(ques):
    from string import lower
    ans = lower(raw_input(ques))
    return (ans[0] == 'y')

```

شرط حلقه بیرونی **1** است که یعنی تا زمان اجرای دستور **break** ادامه می‌یابد. این دستور در صورتی رخ می‌دهد که کاربر به یک حیوان فکر نکند.

حلقه **while** درونی با کمک گرفتن از پاسخ‌های کاربر، درخت را از بالا به پایین می‌پیماید. هنگامی که گره جدیدی به درخت اضافه می‌شود، پرسش جدید در بار **(cargo)** جایگزین می‌شود و دو فرزند شامل حیوان جدید و بار اولیه خواهند بود. یک از نارسایی‌های برنامه این است که وقتی از آن خارج شوید، هر آنچه که با دقت در موردش فکر کرده بودید، فراموش می‌شود.

تمرین ۲۰-۴: در مورد راه‌های مختلف ذخیره کردن درخت دانسته‌ها در یک فایل فکر کنید. روشی را که فکر می‌کنید از همه ساده‌تر است، پیاده‌سازی کنید.

۲۰-۸- واژه‌نامه

binary tree (درخت دودویی)

درختی که در آن هر گره به صفر، یک و یا دو گره وابسته، اشاره می‌کند.

root (ریشه)

فوقانی‌ترین گره یک درخت که هیچ والدی ندارد.

leaf (برگ)

پایینی‌ترین گره یک درخت که هیچ فرزندی ندارد.

parent (والد، پدر)

گره‌ای که به یک گره خاص اشاره می‌کند.

child (فرزند)

یکی از گره‌هایی که گره دیگری به آن اشاره می‌کند.

sibling (برادر)

گره‌هایی که یک والد مشترک دارند.

level (سطح)

مجموعه گره‌هایی که از ریشه فاصله یکسانی دارند.

binary operator (عملگر دودویی)

عملگری که دو عملوند می‌گیرند.

preorder

روشی برای پیمایش یک درخت که در آن هر گره قبل از فرزندانش مشاهده می‌شود.

prefix

روشی برای نوشتن عبارات ریاضی که در آن هر عملگر قبل از عملوندهایش ظاهر می‌شود.

postorder

روشی برای پیمایش یک درخت که در آن فرزندان هر گره قبل از خود گره بررسی می‌شوند.

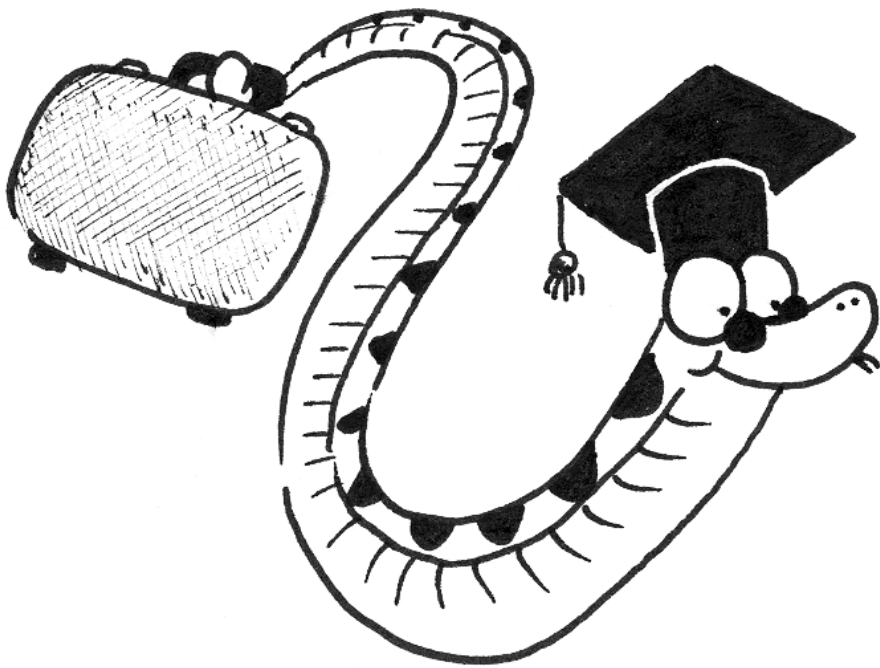
inorder

روشی برای پیمایش یک درخت که در آن زیردرخت سمت چپ، سپس ریشه و آنگاه زیردرخت سمت راست بررسی می‌شوند.

subexpression (زیر عبارت)

عبارتی که در پرانتز قرار دارد و در یک عبارت بزرگ‌تر به عنوان یک عملوند واحد عمل می‌کند.

از این پس، پایتون



تاکنون ۲۰ فصل از کتاب را مطالعه کرده‌اید. در این ۲۰ فصل شما مطالب زیادی را در برنامه‌نویسی به زبان پایتون آموخته‌اید. انواع مختلف داده‌ای، دستورات تصمیم و تکرار، روش‌های بسته‌بندی و تعمیم، چگونگی برنامه‌نویسی به روش شیء‌گرا و بررسی تعدادی از ساختمان‌های داده، مطالبی بودند که در فصل‌های گذشته به آنها پرداختیم.

پایتون نحوه‌های نگارش متفاوتی برای کوتاه و ساده‌سازی کد برنامه ارائه می‌دهد که در این فصل آنها را در طی مثال‌های متنوعی می‌آموزید. آنچه در فصل‌های گذشته آموختید برای نوشتن برنامه‌های کاربردی به زبان پایتون کافی بود. اما از این پس می‌توانید بر روی روش‌های ساده‌سازی و زیان‌نویسی کد تلاش کنید.

۲۱-۱- دنباله‌ها

در فصل‌های گذشته با مفهوم دنباله آشنا شدید و دنباله‌های پیش‌ساخته پایتون یعنی رشته‌ها، لیست‌ها، چندتایی‌ها و دیکشنری‌ها را هم ملاحظه کردید. همچنین در بخش ۸-۱ دیدید که می‌توانیم یک دنباله دیگری به کار ببریم این نوع دنباله‌ها را دنباله‌های تودرتو می‌نامیم.

در دیگر زبان‌های برنامه‌نویسی مفهومی به نام **آرایه** وجود دارد که تا حدودی مشابه دنباله‌های پایتون است، با این تفاوت که قدرت آنها بسیار کمتر است. آرایه، به گروهی از متغیرها گفته می‌شود که نام مشترکی دارند و نوع یکسانی از اطلاعات را نگهداری می‌کنند. دنباله‌های پایتون بسیار انعطاف‌پذیرترند، چراکه می‌توانند انواع مختلفی از داده‌ها را نگهداری کنند و حتی خود آنها نیز می‌توانند یکی از این انواع داده‌ای باشند.

تا به حال، ما هر کدام از دنباله‌های تعریف شده در پایتون را به عنوان یک **دنباله تک‌بعدی** به کار برده‌ایم. دنباله تک‌بعدی، دنباله‌ای است که تنها شامل اعضای از نوع داده‌ای تجزیه‌ناپذیر باشد. در فصل‌های اول با انواعی چون اعداد صحیح و اعداد اعشاری آشنا شدید که از جمله انواع داده‌ای تجزیه‌ناپذیر هستند. یکی دیگر از انواع تجزیه‌ناپذیر کاراکترها هستند که در دیگر زبان‌های برنامه‌نویسی بسیار مورد استفاده قرار می‌گیرند. برای دستیابی به یک کاراکتر ما از عملگر `[]` استفاده می‌کردیم. برای مثال:

```
>>> String="Python"
>>> print String[3]
h
```

بسته به نوع نگرش ما به یک تعداد کاراکتر، می‌توانیم آنها را به عنوان یک نوع تجزیه‌ناپذیر (رشته) و یا تجزیه‌پذیر (مجموعه کاراکترها) در نظر بگیریم. بنابراین دنباله‌ای مانند این مثال می‌تواند یک دنباله تک‌بعدی در نظر گرفته شود:

```
>>> S=['James', 'Michel', 'Guido']
```

مثال فوق دنباله‌ای تک‌بعدی از اسامی است.

۲۱-۲- دنباله‌های چندبعدی

اگر دنباله‌ها خود شامل دنباله باشند، دنباله‌های تودرتو یا چندبعدی نامیده می‌شوند. برای نمونه به دنباله زیر توجه کنید:

```
>>> matrix=[[2,3],[4,5],[6,7]]
>>> matrix[2][0]
6
```

همان‌طور که می‌بینید برای دستیابی به عناصر تجزیه‌ناپذیر موجود در این لیست تودرتو باید از دو عملگر `[]` استفاده کنیم. به همین خاطر ما به این دنباله، یک دنباله دوبعدی می‌گوییم. اگر رشته‌ها را به عنوان عناصر تجزیه‌پذیر استفاده کنیم، می‌توانیم هر رشته را یک دنباله تک‌بعدی ببینیم. در آن صورت دنباله‌ای مانند دنباله زیر هم یک دنباله دو بعدی است:

```
>>> S=["James", "Michel", "Guido"]
>>> S[2][0]
G
```

برای اینکه با دنباله‌های چندبعدی بیشتر آشنا شویم با طراحی یک مثال آنها را ادامه می‌دهیم. فرض کنید می‌خواهیم نمرات میان‌ترم و پایان‌ترم دانشجویی را در درس ریاضی ذخیره کنیم. به چند روش می‌توانیم این کار را انجام دهیم. یکی از این راه‌ها انتساب یک دیکشنری به یک متغیر است:

```
>>> mathematics = {'midterm':5, 'final':11}
```

همچنین اگر به‌طور قراردادی اولین عدد را نمره میان‌ترم و دومین عدد را نمره پایان‌ترم در نظر بگیریم، می‌توانیم با استفاده از یک لیست یا چندتایی هم این کار را انجام دهیم:

```
>>> mathematics = (5,11)
```

یا

```
>>> mathematics = [5,11]
```

برای دستیابی به نمرات می‌توانید از عملگر [] استفاده کنید:

```
>>> midterm = mathematics[0]
>>> print midterm
5
```

حال فرض کنید بخواهیم نمرات میان‌ترم و پایان‌ترم چندین درس این دانشجو را ذخیره کنیم. می‌توانیم با استفاده از یک دیکشنری که مقادیری از چندتایی‌ها را شامل می‌شود استفاده کنیم. ما این عمل را برای دو درس این دانشجو انجام می‌دهیم:

```
>>> student = {'mathemathics':[5,11], 'Phisycs':[3,10]}
```

و یا اگر به طور قراردادی به هر درس اندیسی نسبت دهیم، می‌توانیم از لیست‌های تودرتو استفاده کنیم:

```
>>> student = [[5,11], [3,10]]
```

با این قرارداد، مثلاً نمرهٔ درس میان‌ترم فیزیک را از این روش به‌دست می‌آوریم:

```
>>> print student[1][0]
3
```

اندیس 1 مشخص‌کنندهٔ درس فیزیک و اندیس 0 مشخص‌کنندهٔ نمرهٔ میان‌ترم است. با اینکه استفاده از لیست‌ها در مقابل دیکشنری‌ها کمی مبهم به نظر می‌رسد اما به زودی مزیت استفاده از این روش را در می‌یابید.

ما توانستیم نمرات میان‌ترم و پایان‌ترم دو درس یک دانشجو را ذخیره کنیم و می‌توانیم این کار را برای تعداد بیشتری از دروس هم انجام دهیم، اما آیا می‌توان تعداد دانشجویان را افزایش داد؟ فرض کنید در رشتهٔ تحصیلی کامپیوتر ۱۰۰ نفر دانشجو وجود دارد. برای ذخیرهٔ نمرات میان‌ترم و پایان‌ترم این ۱۰۰ دانشجو می‌توان هر دو روش استفاده از لیست و دیکشنری را به کار برد. ما این کار را مستقیماً با استفاده از اندیس‌گذاری هر دانشجو و استفاده از لیست‌ها، برای ۵ نفر از آنها انجام می‌دهیم. روند کلی برای ذخیرهٔ نمرات بقیه هم به همین صورت است:

```
>>>scomputings=s[[[5, 11], [3, 10]], [[2, 7], [3, 8]], [[4, 7], [3, 9]], [[3, 9], [2, 7]], [[6, 12], [5, 10]]]
```

حال که نگهداری نمرات به این سادگی است، بیایید نمرات تمام دانشجویان یک دانشکده در رشته‌های مختلف را ذخیره کنیم. برای سادگی، فرض کنید این دانشکده ۴ رشته، در هر رشته ۵ دانشجو و هر دانشجو ۲ درس دارد.

قبل از اینکه این کار را انجام دهیم، بهتر است دلیل استفاده از لیست‌ها به جای دیکشنری‌ها را بیان کنیم.

علیرغم اینکه دیکشنری‌ها نحوه نگارش ساده‌تری را برای دسترسی به عناصر تجزیه‌ناپذیر فراهم می‌سازند، اما استفاده از آنها در جهت نگهداری مقادیر دارای یک مشکل اساسی است.

در مثالی که عنوان شد، فرض کنید دانشکدهٔ مزبور دارای ۱۰ رشته و در هر رشته ۱۰۰ نفر دانشجو که هر کدام ۵ درس را امتحان داده‌اند، وجود داشته باشد. آنگاه برای نگهداری نمرات میان‌ترم و پایان‌ترم تمام دروس این تعداد دانشجو باید ۱۰۰۰۰ عدد را ذخیره کنیم. حال اگر بخواهیم این ۱۰۰۰۰ عدد را در یک دیکشنری ذخیره کنیم، باید کلمات **midterm** و **final** را هم هر کدام ۵۰۰۰ بار ذخیره کنیم. همچنین نام هر درس هم باید ۵۰۰ بار تکرار شود و اگر هر دانشجو را با کلمهٔ **student** و شماره‌ای در آخر آن (مثلاً **student23**) نشان دهیم، باید این کلمات را هر کدام ۵ بار ذخیره نماییم!

با این حساب، مقدار فضایی که برای نگهداری محل یک نمره مشخص می‌شود، از مقدار فضایی که برای خود نمره ذخیره می‌شود بیشتر است. بنابراین بهتر است برای جلوگیری از چنین فاجعه‌ای، از لیست‌های تودرتو استفاده کنیم و در عوض اسامی تکراری را اندیس‌گذاری کرده و در چند دیکشنری نگهداری کنیم. حال فرض کنید نمرات تمام دانشجویان یک دانشکده را هم به روش قبل در متغیری به نام **university** ذخیره کرده‌ایم. به طوری‌که هر رشته با یک اندیس مشخص شده است. با این فرض دیکشنری‌های مربوطه را مقداردهی می‌کنیم:

```
branches = {'Computing':0, 'Mathematics':1}
students = {'student1':0, 'student2':1, 'student3':2}
lessons = {'mathematics':0, 'phisycs':1, 'programming':2}
numKind = {'midterm':0, 'final':1}
```

با انجام این کار، می‌توانیم نحوه نگارش برای بازیابی اطلاعات را هم تصحیح کنیم:

```
>>>sprintsuniversity[branches['Computing']][students['student2']
][lessons['mathematics']][numKind['final']]
7
```

هر چند که این نحوه نگارش به زیبایی استفاده از خود دیکشنری‌ها نیست!

۲۱-۳- وارد کردن اطلاعات

در بخش‌های قبل برای ذخیره اطلاعات از انتساب یک لیست تودرتو به یک متغیر استفاده کردیم، اما بهتر است راه بهتری را برای این منظور تدارک ببینیم.

ابتدا باید مشخص کنیم که اطلاعات ورودی ما چقدر است. دانشکده‌ای با ۳ رشته که در هر رشته ۵ نفر دانشجوی دارد را در نظر بگیرید. اگر بخواهیم نمرات میان‌ترم و پایان‌ترم ۲ درس این دانشجویان را ذخیره کنیم، به دنباله‌ای چهاربعدی نیاز داریم. بعد اول رشته تحصیلی، بعد دوم دانشجوی مورد نظر، بعد سوم نام درس و بعد چهارم نوع نمره (میان‌ترم یا پایان‌ترم) خواهد بود.

بیا یاد بررسی کنیم که چگونه می‌توان این دنباله را ساخت. ابتدا با یک دنباله تک‌بعدی شروع می‌کنیم. ما تابعی می‌نویسیم که با گرفتن آرگومانی به‌عنوان تعداد اعضای این دنباله، لیستی با عناصر ۰ را برمی‌گرداند:

```
def zeroList(n):  
    zl=[]  
    for i in range(n):  
        zl.append(0)  
    return zl
```

با اینکه این تابع درست کار می‌کند اما کد آن کمی طولانی است!

۲۱-۴- اشتغال لیست‌ها

لیست‌ها از پرکاربردترین اشیاء پایتونند. با اینکه سرعت چندتایی‌ها بیشتر است، اما چون لیست‌ها انعطاف پذیری بیشتری دارند اغلب از آنها استفاده می‌کنیم. پایتون، برای کار با این اشیاء نحوه نگارش زیبایی فراهم کرده است که از آن به‌عنوان **اشتغال لیست‌ها** یاد می‌شود.

ما می‌توانیم دستورات تصمیم و تکرار را به سادگی درون لیست‌ها به کار ببریم. برای مثال، به منظور ساختن یک لیست با اعضای ۰ می‌توانیم از این نحوه نگارش استفاده کنیم:

```
>>> print [0 for i in range(5)]  
[0, 0, 0, 0, 0]
```

حلقه **for** درون لیست اجرا می‌شود و به ازای هر **i** در محدوده ۰ تا ۵ یک ۰ را درون لیست قرار می‌دهد. می‌توانید به جای ۰، از هر عدد یا تابع دیگری استفاده کنید و لیستی از مقادیر حاصله را نتیجه بگیرید:

```
>>> print [i**2+1 for I in range(1,10,2)]
[2, 10, 26, 50, 82]
```

می‌توانید یک اشتمال لیست را به راحتی به متغیری انتساب دهید. در این حالت لیست تولید شده توسط اشتمال به آن متغیر اختصاص می‌یابد:

```
>>> zl=[0 for i in range(10)]
>>> print zl
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

دستور **if** را هم می‌توان در اشتمال لیست‌ها به کار برد. برای مثال کد زیر دنباله‌ای از اعداد اول را تولید می‌کند:

```
>>> print [i for i in range(2,25) if i%2!=0 and i%0!=0]
[5, 7, 11, 13, 17, 19, 23]
```

با این قابلیت، تابع بخش قبل را به صورت زیر بازنویسی می‌کنیم:

```
def zeroList(n):
    return [0 for i in range(n)]
```

۲۱-۵- Lambda

تابع دو خطی بخش قبل شاید شما را وسوسه کرده باشد که کد کوتاه‌تری برای آن بنویسید. پایتون نحوه نگارشی را برای نوشتن توابع کوتاه فراهم ساخته که نیل به این هدف را ممکن می‌سازد:

```
>>> zeroList = lambda n:[0 for i in range(n)]
```

با مقایسه این کد با تابع **zeroList** در بخش قبل، به راحتی چگونگی کار با **lambda** را خواهید آموخت.

بعد از کلمه کلیدی **lambda** نام پارامترها ذکر می‌شوند و پس از علامت کولن (:)، مقدار برگشتی. با انتساب این کد به هر متغیر، آن متغیر قابلیت یک تابع را خواهد یافت:

```
>>> zeroList(4)
[0, 0, 0, 0]
```

می‌توانید آرگومان‌های اختیاری را هم در توابعی که به این صورت می‌سازید، منظور کنید:

```
>>> prefix=lambda x,y='Mr.': y+' '+x
>>> print prefix('Janson')
Mr. Janson
>>> print prefix('Janson','Mrs.')
Mrs. Janson
```

برای اینکه مثال زیباتری را به شما نشان دهیم، بیاید ابتدا تابع **reduce** که از توابع پیش‌ساخته پایتون است را بررسی کنیم. تابع **reduce** یک تابع و یک دنباله را به عنوان آرگومان دریافت می‌کند. تابعی که به عنوان آرگومان به **reduce** فرستاده می‌شود، حتماً باید مستقلاً دو آرگومان دریافت کند و مقداری را بازگرداند. **reduce** مقدار اول و دوم دنباله را برداشته و تابع مذکور را برای آن‌ها فرامی‌خواند. سپس مقدار برگشتی از تابع را با مقدار سوم از دنباله به تابع می‌فرستد و نتیجه را با عضو چهارم دنباله به تابع می‌فرستد. این کار تا زمانی که اعضای دنباله تمام شوند، ادامه می‌یابد. در آخر مقدار برگشتی تابع توسط **reduce** برگردانده می‌شود.

فرض کنید تابعی که به عنوان آرگومان در نظر گرفته می‌شود، تابعی برای محاسبه ضرب دو عدد باشد:

```
>>> mult=lambda x, y : x*y
```

این تابع را به همراه **range(1, -n+1)** به **reduce** می‌فرستیم. **n** می‌تواند هر عدد صحیحی باشد که در اینجا ما آن را با عدد 5 امتحان می‌کنیم:

```
>>> print reduce(mult,range(1, 5+1))
120
```

این کار را برای اعداد 4 و 6 نیز تکرار کنید. جواب‌ها 24 و 720 خواهند بود. حتماً تا به حال متوجه شده‌اید که این نتایج، فاکتوریل اعدادند. پس تابع **factorial** را بازنویسی می‌کنیم:

```
def factorial(n):
    return reduce(lambda x, y : x * y, range(1, n+1), 0)
```

آرگومان سوم، اختیاری است. این مقدار در صورتی که مقداری در دنباله فرستاده شده به عنوان آرگومان یافته نشود، برگردانده می‌شود. می‌دانید که فاکتوریل عدد 0 برابر با 1 است:

```
>>> factorial(0)
1
```

اما متأسفانه اشکالاتی هم در این کد وجود دارد:

```
>>> factorial(-1)
1
```

تمرین ۲۱-۱: با اضافه کردن چند خط کد نگاهی به این اشکالات را برطرف کنید.

۲۱-۶- یک تابع بازگشتی

کمی از بحث اصلی دور شدیم. در بخش ۲۱-۳ قصد داشتیم تابعی بنویسیم که با گرفتن آرگومانی به عنوان بعد یک دنباله تک بعدی، لیستی از صفرها را بازگرداند. حال اگر بخواهیم این تابع دو آرگومان بگیرد و یک دنباله دو بعدی را بازگرداند باید چه عملی انجام دهیم؟

```
def sequence(a, b):
    zeroList = lambda n : [0 for i in range(n)]
    zl = zeroList(a)
    for i in range(len(zl)):
        zl[i] = zeroList(b)
    return zl
```

در ابتدای تابع **sequence**، تابع **zeroList** را تعریف می‌کنیم و برای عضو اول یک دنباله تک بعدی از ۰ها می‌سازیم. حال این دنباله را پیمایش می‌کنیم و اعضای آن را به لیست‌های تک بعدی از بعد دوم، **b**، تغییر می‌دهیم. بنابراین هرگاه این تابع را فراخوانی کنیم، یک لیست دو بعدی از ۰ها را نتیجه می‌گیریم:

```
>>> print sequence(2, 3)
[[0, 0, 0], [0, 0, 0]]
```

حتماً تا به حال وسوسه شده‌اید، تابعی بنویسید که آرگومان‌های بیشتری را به عنوان ابعاد دریافت کند. اگر بخواهیم تعداد آرگومان‌ها اختیاری باشد، می‌توانیم آنها را در قالب یک چندتایی به تابع بفرستیم و از آنها استفاده کنیم.

برای ساختن دنباله چند بعدی هم تنها کاری که باید انجام دهیم این است که به جای استفاده از تابع **zeroList** در انتساب به اعضای لیست تک بعدی، تابع خودمان را فراخوانی کنیم. چرا که هر عضو این لیست باید به یک دنباله چند بعدی تبدیل شود. آرگومان‌های ارسالی، همه اعضای چندتایی، به جز عنصر اول خواهند بود:

```
def sequence(d):
    zeroList = lambda n:[0 for i in range(n)]
    zl = zeroList(d[0])
    if len(d) == 1:
        return zl
    else:
        for i in range(len(zl)):
            zl[i] = sequence(n[1:])
    return zl
```

خروجی تابع برای ساخت یک دنباله سه‌بعدی به این صورت خواهد بود:

```
>>> print sequence((2,3,2))
[[[0, 0],[0, 0],[0, 0]],[[0, 0],[0, 0],[0, 0]]]
```

۲۱-۷- آرگومان‌های اختیاری

در بخش ۱۴-۵، چگونگی استفاده از آرگومان‌های اختیاری را توضیح دادیم. در روش مزبور ما مجبور بودیم مقادیری را به پارامترها به طور پیش‌فرض نسبت دهیم تا در توابع استفاده شوند، اما نمی‌توانستیم بیش از آرگومان‌های مقداردهی شده آرگومانی را به تابع بفرستیم. بهتر است آن نسخه از تابع **sequence** را که ما با یک چندتایی ابعاد نوشتیم، طوری نوشته شود که نیازی به استفاده از چندتایی نباشد. این کار بسیار صریح‌تر هدف تابع **sequence** را مشخص می‌کند. پایتون نحوه نگارشی را تدارک دیده است که در آن آرگومان‌های تابع واقعاً اختیاری هستند. در یک مثال ساده این نحوه نگارش را ملاحظه کنید:

```
>>> def func(a, *b): print a ; print b
>>> func(2, 3)
2
(3, )
```

تابع **func** دو آرگومان می‌گیرد و هر کدام را در یک خط چاپ می‌کند. علامت سمی‌کلنی (**;**) که دو دستور **print** را از هم مجزا کرده است، نشان دهنده پایان دستور اول است. برای نشان دادن چند دستور در یک خط، می‌توانید از این علامت استفاده کنید. مفسر پایتون در صورت برخورد با این علامت دستور قبل از آن را اجرا می‌کند و سپس بقیه جمله را ادامه می‌دهد.

با نگاهی به خروجی برنامه نحوه عمل * را هم متوجه خواهید شد. دستور `print b` آرگومان 3 را به عنوان یک چندتایی چاپ کرده است. بار دیگر این تابع را با تعداد آرگومانهای متنوع‌تری، فراخوانی می‌کنیم:

```
>>> func(2,3,{'key':'value'},'string',('tuple',),['list'])
2
(3, {'key': 'value'}, 'string', ('tuple',), ['list'])
```

همان طور که می‌بینید، همه آرگومان‌ها به جز آرگومان اول به عنوان یک چندتایی چاپ شده‌اند. این چندتایی شامل یک عدد صحیح، یک دیکشنری، یک رشته، یک چندتایی و یک لیست است.

با قرار دادن دو علامت ستاره (**) در ابتدای پارامترهای یک تابع، می‌توانیم عمل جالب‌تری هم انجام دهیم. تابع زیر را ملاحظه کنید:

```
>>> def func2(a, *b, **c): print a ; print b ; print c
>>> func2(2,3,4,5,key='value')
2
(3, 4, 5)
{'key': 'value'}
```

اگر در این گونه توابع پارامتری را با انتساب یک مقدار به عنوان آرگومان به تابع بدهید، می‌توانید در تابع از آن به عنوان یک دیکشنری استفاده کنید. در این حالت پارامتر فرستاده شده، به عنوان یک کلید و مقدار انتسابی به عنوان یک مقدار در دیکشنری استفاده خواهد شد. باید توجه داشته باشید که در صورتی که بخواهید از هر سه نوع پارامتر (`a, *b, **c`) استفاده کنید، باید ترتیب آنها را حفظ کنید. پارامتری که با یک ستاره استفاده می‌شود پس از آرگومان‌های عادی و آرگومانی که با دو ستاره استفاده می‌شود همیشه در آخر قرار می‌گیرد. همچنین باید توجه داشته باشید که هر کدام از این دو نوع پارامتر را تنها یک بار می‌توانید در هر تابع به کار ببرید. با این قابلیت می‌توانیم تابع `sequence` را هم بازنویسی کنیم:

```
def sequence(*n):
    zeroList = lambda x:[0 for I in range(x)]
    zl = zeroList(n[0])
    if len(n) == 1:
        return zl
    for i in range(len(zl)):
        zl[i] = sequence(n[1:])
    return zl
```

این تابع کار نمی‌کند. به محلی که `zl[i]` در آن مقدار دهی می‌شود، نگاه کنید. ما یک فراخوانی بازگشتی داریم و یک چندتایی را به عنوان آرگومان به آن می‌فرستیم، اما این چندتایی در تابع `sequence` خود درون یک چندتایی دیگر قرار می‌گیرد، چرا که ما از فرم `*n` استفاده کرده‌ایم. برای تصحیح کد ما قبل از هر چیز بررسی می‌کنیم که اولین عضو چندتایی ساخته شده به وسیله فرم `*n` از چه نوعی است و در صورتی که این عضو یک چندتایی بود آن را بیرون می‌کشیم و آن را جایگزین چندتایی موجود می‌کنیم:

```
def sequence(*n):
    from types import TupleType
    if type(n[0]) == TupleType:
        n = n[0]
        zeroList = lambda x:[0 for i in range(x)]
        zl = zeroList(n[0])
        if len(n) == 1:
            return zl
        for i in range(len(zl)):
            zl[i] = sequence(n[1:])
    return zl
```

در بخش ۵-۸ برای اینکه نوع داده‌ها را با هم مقایسه کنیم، از تابع `type` استفاده کردیم و یک نوع داده‌ای مشخص را با نوع داده‌ای مورد نظرمان مقایسه کردیم. در اینجا ما از ماژول `types` استفاده نمودیم. این ماژول همه انواع داده‌ای پیش‌ساخته را در بر می‌گیرد. در صورتی که بخواهید یک نوع داده‌ای را به کار ببرید آن را از ماژول `types` وارد محیط کاری کنید. نام هر نوع داده‌ای با حروف بزرگ شروع می‌شود و با حروف کوچک ادامه می‌یابد و سپس کلمه `Type` در پایان آن قرار می‌گیرد، مانند `ListType` یا `TupleType`.

تابع `sequence` را برای ساختن یک دنباله چندبعدی امتحان کنید. برای مثال ما این تابع را با سه آرگومان فراخوانی می‌کنیم:

```
>>> sequence(2,3,2)
[[[0, 0], [0, 0], [0, 0]], [[0, 0], [0, 0], [0, 0]]]
```

حال، هر یک از عناصر را به راحتی می‌توان تغییر داد:

```
>>> d = sequence(2,3,2)
>>> d[2][1][0] = 123
>>> print d
[[[0, 0], [0, 0], [0, 0]], [[0, 0], [123, 0], [0, 0]]]
```

در اینجا بهتر است برای ورود اطلاعات هم تابعی بنویسیم تا مجبور نباشیم تک‌تک عناصر را به صورت فوق مقداردهی کنیم. باز هم می‌توانیم یک تابع بازگشتی بنویسیم.

این تابع اعضای یک دنباله را می‌پیماید. در صورتی که عنصر مورد بررسی، یک عنصر تجزیه‌ناپذیر باشد (دنباله نباشد) با تابع `input` یا `raw_input` آن را مقداردهی می‌کنیم و در صورتی که یک دنباله باشد تابع خودمان را برای آن فرا می‌خوانیم و در آخر دنباله مقداردهی شده را بر می‌گردانیم:

```
def inputItems(Sequence):
    from types import ListType
    for i in range(len(Sequence)):
        if type(Sequence[i]) != ListType:
            Sequence[i] = input()
        else:
            inputItems(Sequence[i])
    return Sequence
```

با داشتن توابع `sequence` و `inputItems` به راحتی می‌توانیم نمرات را ذخیره کنیم، اما باید توجه داشته باشید که در ورود اطلاعات اشتباه نکنید چراکه به‌خاطر سپردن اینکه نمره دانشجویی در محل مورد نظر وارد شده یا نه، در هنگام ورود اطلاعات در حجم بالا بسیار دشوار است. در این مثال هدف ما آموزش مفاهیم دنباله‌های چندبعدی و استفاده از نحوه‌های نگارش مختلف برای آسان‌سازی کد، مثل دستور `lambda`، آرگومان‌های اختیاری و اشتغال لیست‌ها بود. برای نگهداری اطلاعات راه‌های متنوع‌تر و کاربرپسندتری وجود دارد.

۲۱-۸- ماتریس‌ها

با اینکه زبان‌های برنامه‌نویسی مختلف راه‌های متنوعی برای کار با دنباله‌های چندبعدی تدارک دیده‌اند اما در عمل، بیشتر از دنباله‌های دوبعدی استفاده می‌شود. ماتریس‌ها از پرکاربردترین مفاهیم ریاضی در برنامه‌نویسی کامپیوتر هستند.

در این بخش با استفاده از مفهوم دنباله چندبعدی به بررسی آنها می‌پردازیم. همان‌طور که در بخش ۸-۱۵ و ۱۰-۴ گفته شد، می‌توان ماتریس‌ها را به‌وسیله لیست‌های تو در تو و دیکشنری‌ها نمایش داد. در این بخش ما از لیست‌ها استفاده می‌کنیم. دلیل این امر را به زودی متوجه خواهید شد. اگر یک ماتریس را به‌عنوان یک شیء در نظر بگیریم، می‌توانیم کلاسی برای آن بنویسیم و مفاهیم مختلف را در قالب متدها و توابع پیاده‌سازی کنیم. پس با این ایده، شروع به نوشتن کلاس ماتریس می‌کنیم:

```
class Matrix:
    pass
```

اولین نیاز ما، متدی برای مقداردهی اولیه ماتریس است. می‌توانیم تابع `sequence` از بخش قبل را در `__init__` استفاده کنیم و ابعاد ماتریس را به عنوان آرگومان به آن ارسال کنیم ولی راه

ساده‌تر، نوشتن متدی برای ساختن یک دنبالهٔ دوبعدی است. می‌توانیم آن نسخه از تابع `sequence` که در ابتدای بخش ۲۱-۶ نوشتیم را با اندکی تغییر به‌کار ببریم:

```
def __init__(self, dimension1=0, dimension2=0):
    zeroList = lambda n:[0 for i in range(n)]
    self.matrix = zeroList(dimension1)
    for i in range(len(self.matrix)):
        self.matrix[i] = zeroList(dimension2)
    self.dimensions = (dimension1, dimension2)
```

در متد `__init__` متغیر `matrix` دنبالهٔ دوبعدی مورد نظر ما است. در این متد ما نیازی به برگرداندن `matrix` نداریم، چراکه این متغیر به `self` اختصاص یافته و لذا در سراسر کلاس `Matrix` قابل رؤیت است و همهٔ متدهایی که در آینده تعریف می‌کنیم به آن دسترسی دارند. متغیر `dimensions` که در آخر متد `__init__` تعریف شده است هم مانند `matrix` در تمام کلاس قابل دسترسی است. این متغیر ابعاد ماتریس را در یک چندتایی نگهداری می‌کند. بنابراین با دسترسی به این متغیر می‌توانیم از ابعاد ماتریس استفاده کنیم. دومین متد مورد نیاز ما، `__str__` است. برای چاپ یک ماتریس، باید رشته‌ای بسازیم که پس از چاپ، اعضای ماتریس را در سطرها و ستون‌های و متمادی نشان دهد:

```
def __str__(self):
    s=''
    for i in range(self.dimensions[0]):
        for j in range(self.dimensions[1]):
            s+=self.matrix[i][j]+'\\t'
        s+= '\\n'
    return s[:-1]
```

ابتدا متغیر `s` را با یک رشتهٔ تهی مقداردهی می‌کنیم. متغیر `i` سطرها و متغیر `j` ستون‌ها را می‌شمارد. بنابراین متغیر `matrix[i][j]` به ترتیب اعضای ماتریس را مشخص می‌کند. ما این اعضا را به رشته تبدیل می‌کنیم و به متغیر `s` می‌افزاییم.

در این تابع دو عملگر جدید را ملاحظه می‌کنید: علامت کوتیشن معکوس (```) و علامت `+=`. کوتیشن معکوس، دقیقاً مانند تابع `str` عمل می‌کند. هر نوع داده‌ای که در میان دو کوتیشن معکوس قرار گیرد به یک رشته تبدیل می‌شود. این نحوهٔ نگارش در بعضی موارد زیباتر است و کد برنامه را کوتاه‌تر می‌سازد.

عملگر `+=` مقدار سمت راست خود را با متغیر سمت چپ جمع می‌کند و مجدداً به متغیر سمت چپ نسبت می‌دهد. این عملگر از یک عملگر `+` و یک عملگر `=` (که برای انتساب به

متغیرها به کار می‌رود) تشکیل شده است. می‌توانیم از موارد مشابه ($=$ ، $/$ و $=$) نیز به همین صورت استفاده کنیم. در حقیقت دو کد زیر با هم برابرند:

```
n += x
n = n+x
```

از آنجا که ممکن است طول درایه‌های ماتریس متفاوت باشد، ما از کاراکتر `tab` ('`\t`') در آخر هر درایه استفاده می‌کنیم و برای اینکه سطرها را در خطوط متوالی چاپ کنیم، کاراکتر خط جدید ('`\n`') را در پایان هر سطر قرار می‌دهیم.

در آخر چون به آخرین سطر هم یک کاراکتر '`\n`' اضافه شده است ما رشته‌ای مشتمل بر همه کاراکترها به جز کاراکتر آخر را برمی‌گردانیم.

در اینجا کلاس ماتریس ما می‌تواند با گرفتن ابعاد یک ماتریس ساخته و توسط دستور `print` چاپ شود. برای مثال:

```
>>> m = Matrix(2,3)
>>> print m
0      0      0
0      0      0
```

حال متدی برای مقداردهی عناصر ماتریس می‌نویسیم. این متد، عناصر تجزیه‌ناپذیر `matrix` (که در حقیقت درایه‌های ماتریس هستند) را می‌پیماید و آنها را مقداردهی می‌کند.

شاید شما تابع `inputItems`، از بخش ۲۱-۷ را برای این منظور مناسب ببینید، اما دقت کنید که آن تابع، یک تابع بازگشتی است و در آن همواره یک لیست به عنوان یک دنباله چندبعدی برگردانده می‌شود، بنابراین ما نمی‌خواهیم از آن استفاده کنیم، زیرا بعد از نسبت‌دهی عناصر نیازی به دنباله برگشتی نداریم. همچنین نحوه نگارش برای احضار این متد کمی غیرطبیعی خواهد بود، چراکه ما در هر بار باید متغیر `matrix` را به عنوان یک دنباله به `inputItems` بفرستیم. لذا راه حل دیگری برای مقداردهی ماتریس ارائه می‌دهیم:

```
class Matrix:
    ...
    def inputItems(self):
        for i in range(self.dimensions[0]):
            for j in range(self.dimensions[1]):
                while 1:
                    try:
                        self.matrix[i][j]=input('Item['+'`i`'+']['+'`j`'+']:')
                        break
                    except NameError:
                        pass
```

طریقهٔ پیمایش درایه‌های ماتریس در این متد، دقیقاً مشابه متد `__str__` است با این تفاوت که در حلقهٔ `for` دوم تا متغیر `matrix[i][j]` با یک نوع داده‌ای مجاز مقداردهی نشود، عنصر بعدی را پیمایش نمی‌کنیم.

شما باید در نظر داشته باشید که چون از دستور `input` استفاده شده است، تنها می‌توانیم یک عدد اعشاری یا صحیح را وارد ماتریس کنیم و کاراکترهای الفبایی نمی‌توانند اعضای ماتریس را تشکیل دهند.

رشته‌ای که در تابع `input` استفاده می‌شود، هر بار دنباله‌ای را که باید مقداردهی شود به کاربر اعلان می‌کند. حال کلاس `Matrix` را امتحان می‌کنیم:

```
>>> m = Matrix(2, 3)
>>> m.inputItems()
Element[0][0]:1
Element[0][1]:2
Element[0][2]:3
Element[1][0]:4
Element[1][1]:5
Element[1][2]:6
>>> print m
1      2      3
4      5      6
```

اگر بخواهیم عناصر بخصوصی از ماتریس را تغییر دهیم، چه کاری می‌توانیم انجام دهیم؟ یکی از این راه‌ها دسترسی به متغیر `matrix` و تغییر لیست است:

```
>>> m.matrix[0][1] = 300
>>> print m
1      300    3
4      5      6
```

اما این کار کمی خطرناک است. از آنجا که متغیر `matrix` به راحتی قابل دستیابی و تغییر می‌باشد، ممکن است آن را به این صورت تغییر دهیم:

```
>>> m.matrix[0] = 300
>>> print m
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in ?
    print m
  File "<pyshell#2>", line 32, in __str__
    s+=`self.matrix[i][j]`+' '
TypeError: unsubscriptable object
```

از این پس شیء ماتریس ما خراب شده است. با اینکه می‌توانیم با یک انتساب مجدد آن را ترمیم کنیم، اما بهتر است راه دیگری برای تغییر عناصر ماتریس پیدا کنیم و در عوض دستیابی به متغیر **matrix** را محدود کنیم.

۲۱-۹- متغیرها و توابع اختصاصی

اگر بخواهیم مهم‌ترین متغیر موجود در کلاس ماتریس را مشخص کنیم، بی‌شک متغیر **matrix** را نام خواهیم برد. زیرا همهٔ عناصر در این متغیر که از نوع لیست می‌باشد، وجود دارد. حال اگر این متغیر به نوعی توسط دیگر توابع و یا حتی توسط خود کاربر از بین برود و یا خراب شود، شیء ماتریس ما هیچ مورد استفاده‌ای ندارد.

برای جلوگیری از چنین حادثه‌ای ما باید دسترسی به **matrix** را محدود کنیم. در پایتون اگر دو علامت خط زیر (`_`) را در ابتدای نام متغیرها و یا توابع یک کلاس قرار دهیم، این متغیرها یا توابع دیگر به روش نمادگذاری نقطه‌ای معمول قابل دستیابی نخواهند بود و برای کلاسی که تعریف شده‌اند **اختصاصی** خواهند بود. البته باید توجه داشته باشید که اشیائی که از نوع کلاس حاضر هستند می‌توانند به این متغیرها دسترسی داشته باشند.

با این قابلیت ما نام متغیر **matrix** را به **__matrix__** تغییر می‌دهیم و در هر جا که این متغیر را به کار بردیم، از نام جدید آن استفاده می‌کنیم:

```
class Matrix:
    def __init__(self, dimension1=0, dimension2=0):
        zeroList = lambda n:[0 for i in range(n)]
        self.__matrix = zeroList(dimension1)
        for i in range(len(self.__matrix)):
            self.__matrix[i] = zeroList(dimension2)
        self.dimensions = (dimension1, dimension2)

    def __str__(self):
        s=''
        for i in range(self.dimensions[0]):
            for j in range(self.dimensions[1]):
                s+=self.__matrix[i][j]+'\\t'
            s+= '\\n'
        return s[:-1]

    def inputItems(self):
        for i in range(self.dimensions[0]):
            for j in range(self.dimensions[1]):
                while 1:
                    try:
                        self.__matrix[i][j]=input('Item['+str(i)+'']['+str(j)+'']: ')
                        break
                    except NameError:
                        pass
```

حال در صورتی که بخواهیم از روش نمادگذاری نقطه‌ای استفاده کنیم با اعتراض زیر مواجه خواهیم شد:

```
>>> m = Matrix(2, 3)
>>> print m.__matrix
AttributeError: Matrix instance has no attribute '__matrix'
```

با اینکه استفاده از این نحوه نگارش به منظور اختصاصی کردن نام متغیرها و توابع می‌باشد، اما باز هم راهی برای دسترسی به این متغیرها وجود دارد که ما پیشنهاد می‌کنیم از آن استفاده نکنید:

```
>>> m._Matrix__matrix
[[0, 0], [0, 0]]
```

اگر پس از نام شیء و علامت نقطه یک علامت زیرخط (_) قرار دهید و نام کلاس و سپس نام متغیر یا تابع را ذکر کنید، می‌توانید اختصاصی بودن یک تابع یا متغیر را منحل کنید. طراح این زبان معتقد است که این نه تنها نقطه ضعف پایتون نیست، بلکه کاملاً بر اساس طراحی می‌باشد.

تمرین ۲۱-۲: بدون اینکه از نحوه نگارش فوق استفاده کنید، تابعی بنویسید که کاربر را قادر به تغییر درایه مشخصی از ماتریس کند. توجه داشته باشید که شیء ماتریس با انتساب یک دنباله از بین نرود.

۲۱-۱۰- ضرب ماتریس‌ها

همان‌طور که می‌دانید دو ماتریس در صورتی می‌توانند در هم ضرب شوند که تعداد ستون‌های اولین ماتریس با تعداد سطرهای دومین ماتریس برابر باشد. درایه‌های هر سطر از ماتریس اول در درایه‌های متناظر در ستون ماتریس دوم ضرب می‌شوند و مجموع آنها در مکان درایه‌ای با عدد سطر ماتریس اول و عدد ستون ماتریس دوم در ماتریس حاصل ضرب قرار می‌گیرد. با این توضیح ما متدی به نام `__mul__` می‌نویسیم که عمل ضرب دو ماتریس را انجام می‌دهد. قبلاً با نحوه عمل `__mul__` آشنا شده‌اید:

```
def __mul__(self, other):
    product = Matrix(self.dimension[0], other.dimensions[1])
    for i in range(self.dimension[0]):
        for k in range(other.dimensions[1]):
            for j in range(self.dimensions[1]):
                product.__matrix[i][k] += (self.__matrix[i][j] * \
                                             other.__matrix[j][k])
    return product
```

در ابتدای متد، ما ماتریسی را می‌سازیم که از تعداد سطرهای ماتریس اول و تعداد ستون‌های ماتریس دوم تشکیل شده است. حال در سه حلقه متوالی اعمال ضرب و جمع درایه‌ها را انجام می‌دهیم. متغیر حلقه اول درایه‌های موجود در سطرهای ماتریس اول را می‌پیماید. متغیر حلقه دوم، هم ستون‌های ماتریس اول و هم سطرهای ماتریس دوم و متغیر حلقه سوم هم درایه‌های موجود در ستون‌های ماتریس دوم را می‌پیماید.

در آخرین حلقه یک به یک درایه‌های سطرهای ماتریس اول، در درایه‌های متناظر در ستون‌های ماتریس دوم ضرب شده و به درایه حاصل ضرب اضافه می‌شوند. در پایان، ماتریس حاصل ضرب برگردانده می‌شود و می‌توانیم آن را در هر متغیر دلخواهی ذخیره کنیم و یا با دستور **print** آن را چاپ کنیم:

```
>>> m1 = Matrix(3, 1)
>>> m1.inputItems()
Element[0][0] = 1
Element[1][0] = 2
Element[2][0] = 3
>>> m2 = Matrix(1*2)
>>> m2.inputItems()
Element[0][0] = 4
Element[0][1] = 5
>>> print m1 * m2
4      5
8      10
12     15
```

تمرین ۲۱-۳: با اطلاعاتی که در مورد عمل متدهای **__add__**، **__sub__** و **__rmul__** دارید، هر یک از آنها را به کلاس **Matrix** اضافه کنید. آیا می‌توانید متدهای دیگری برای انجام کارهای مختلف با ماتریس‌ها طراحی کنید؟

۲۱-۱۱- کاربرد **else** با حلقه‌ها

تا به حال امکاناتی را در زبان پایتون ملاحظه کرده‌ایم که وجود یا عدم وجود آنها، تأثیر چندانی بر توانایی ما در برنامه‌نویسی نداشته‌اند.

اشتمال لیست‌ها، توابع با فرم **lambda**، استفاده از سمی‌کلن و حتی استفاده از حلقه‌های **for** تنها برای ساده‌تر ساختن برنامه‌نویسی و خوانا کردن کد ساخته شده‌اند. ما می‌توانیم بدون استفاده از این قابلیت‌ها هم برنامه‌هایمان را بنویسیم.

در این بخش یکی از امکانات پایتون را بررسی می‌کنیم که وجود آن با اینکه شاید آنچنان هم ضروری نیست، اما در بعضی کاربردها می‌تواند مفید باشد.

در بسیاری از برنامه‌ها می‌خواهیم مقادیر یکسانی را بر اساس یک خصوصیت مرتب کنیم. در بخش ۱۹-۶ مثالی را دیدید که بازیکنان بازی گلف را بر اساس امتیاز آنها از کم به زیاد مرتب می‌کرد.

در برخی از کاربردها می‌خواهیم لیستی از رشته‌ها را بر اساس طول رشته مرتب کنیم.

در اینجا تابعی می‌نویسیم که هدف آن برآوردن این منظور است. روش‌های متنوعی برای انجام این کار وجود دارد. روشی که ما در پیش می‌گیریم بسیار انعطاف‌پذیر است و می‌توانیم انواع مرتب‌سازی‌ها از جمله این نوع مرتب‌سازی را در آن انجام دهیم.

ما لیستی که به‌عنوان آرگومان به تابع ارسال می‌کنیم را `listOfStrings` می‌نامیم و می‌خواهیم لیست دیگری با همین اعضاء، اما با یک ترتیب ویژه بسازیم. آن را `sortedList` می‌نامیم. ابتدا یک عضو را از `listOfStrings` برمی‌داریم و در `sortedList` قرار می‌دهیم. حال در هر مرتبه یک عضو را برمی‌داریم و اندازه آن را با اندازه عضوهای `sortedList` مقایسه می‌کنیم. اگر بزرگ‌تر باشد، عضو بعدی `sortedList` را بررسی می‌کنیم و اگر کوچک‌تر بود قبل از آن عضو قرار می‌دهیم. در صورتی که این رشته از همه رشته‌های `sortedList` بزرگ‌تر بود آن را به آخر این لیست اضافه می‌کنیم:

```
def sort(listOfStrings):
    sortedList = []
    sortedList.append(listOfStrings[0])
    for i in range(1, len(listOfStrings)):
        for j in range(len(sortedList)):
            if len(listOfStrings[i]) < len(sortedList[j]):
                sortedList[j:j] = listOfStrings[i:i+1]
                break
        if listOfStrings[i] not in sortedList:
            sortedList.append(listOfStrings[i])
    return sortedList
```

به شرطی که پس از حلقه `for` دوم گذاشته‌ایم، دقت کنید. اگر اندازه رشته مورد بررسی ما از اندازه همه رشته‌های درون `sortedList` بزرگ‌تر بوده باشد، پس هیچگاه به این لیست اضافه نشده است. در اینجا ما آن را به آخر لیست اضافه می‌کنیم.

جالب است بدانید که می‌توانید به جای استفاده از این شرط از کلمه `else` استفاده کنید. در صورتی که کلمه `else` را در یک کنگره‌گذاری یکسان با حلقه‌ها استفاده کنید، می‌توانید از خاصیت جدیدی بهره بگیرید. اگر تکرار حلقه با موفقیت به اتمام برسد، دستور `else` هم اجرا می‌شود، اما اگر

توسط یک دستور داخلی مانند **break** یا **return** از حلقه خارج شویم، دستور **else** اجرا نمی‌گردد.

در اینجا شما می‌توانید به جای دستور شرطی دوم از کلمه **else** استفاده کنید:

```
def sort(listOfStrings):
    sortedList = []
    sortedList.append(listOfStrings[0])
    for i in range(1, len(listOfStrings)):
        for j in range(len(sortedList)):
            if len(listOfStrings[i]) < len(sortedList[j]):
                sortedList[j:j] = listOfStrings[i:i+1]
                break
        else:
            sortedList.append(listOfStrings[i])
    return sortedList
```

بهترین زمان برای به‌کار بردن این نحوه نگارش، زمانی است که دستورات شرطی زیادی در حلقه وجود دارد که در صورت اجرا شدن دستورات درونی آنها، روند تکرار حلقه متوقف می‌شود. اگر بخواهیم دستور خاصی را تنها زمانی که حلقه به‌طور کامل اجرا شده باشد، اجرا کنیم می‌توانیم از این نحوه نگارش استفاده کنیم.

۲۱-۱۲- واژه‌نامه

Array (آرایه)

گروهی از متغیرها که نام مشترکی دارند و نوع یکسانی از اطلاعات را نگهداری می‌کنند.

single-dimensional sequence (دنباله تک‌بعدی)

دنباله‌ای که هر عضو آن یک مقدار تجزیه‌ناپذیر باشد، مانند یک عدد اعشاری، یک عدد صحیح و یک رشته.

multi-dimensional sequence (دنباله چندبعدی)

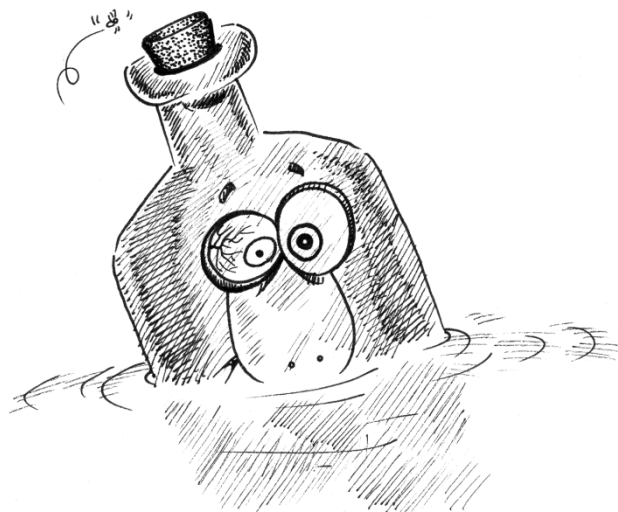
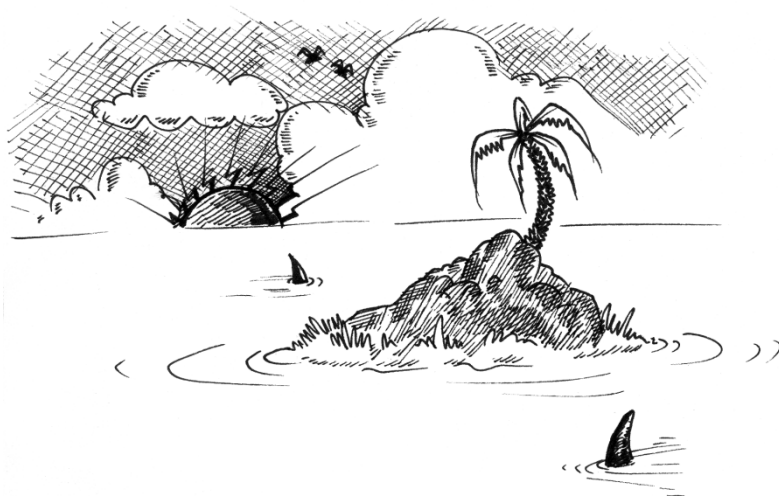
دنباله‌ای که اعضای آن، دنباله (تک‌بعدی یا چندبعدی) هستند.

list comprehension (اشتمال لیست‌ها)

لیست‌هایی که می‌توانند شامل دستوراتی برای تولید یک لیست باشند.

پیوست الف

محیط‌های برنامه‌نویسی پایتون



تاکنون محیط‌های گرافیکی زیادی جهت برنامه‌نویسی پایتون در سیستم عامل‌های گوناگون از سوی شرکت‌های مختلف عرضه شده است، از جمله Tkinter ، wxPython ، win32 ، IDLE و ... هر یک از این محیط‌های برنامه‌نویسی کاربرد خاصی دارند و امکانات لازم جهت ارتباط با مفسر پایتون به عنوان یک زبان همه منظوره را فراهم می‌سازند. در پیوست حاضر به بررسی اجمالی محیط‌های مرسوم و جامع‌تری همچون IDLE و PythonWin می‌پردازیم.

الف-۱- نحوه استفاده از IDLE

IDLE محیط توسعه‌یافته و جالبی است که همراه پایتون نسخه 1.5.2 منتشر شد. این محیط توسط Guido van Rossum ایجاد شده و نام آن برگرفته از عبارت Integrated DeveLopment Environment است (اگرچه ممکن است از نام کم‌دین مشهور انگلیسی که در گروه نمایشی Monty Python فعالیت دارد هم تأثیر پذیرفته باشد). این محیط در حال حاضر بر روی هر دو سیستم عامل‌های ویندوز و یونیکس آزمایش شده است.

IDLE یک Python Shell Window دارد که به شما امکان دسترسی به مُد محاوره‌ای پایتون را می‌دهد. File Editor آن به شما اجازه می‌دهد فایل‌های منبع جدید بسازید و یا فایل‌های موجود را مرور و ویرایش نمایید. در این محیط یک Path Browser وجود دارد که به کمک آن می‌توان در طول یک مسیر ماژول‌های موجود را جستجو کرد.

به‌علاوه یک Class Browser ساده جهت یافتن متدها از درون کلاس‌ها وجود دارد. IDLE در طول پنجره‌های محاوره‌ای Find in Files توانایی جستجوی انعطاف‌پذیری دارد که به شما اجازه می‌دهد فایل‌های خود و یا فایل‌های سیستم را برای یافتن وقوع شناسه‌ها یا هر قطعه متنی دیگری جستجو کنید. در آخر اینکه محیط IDLE یک Debug Control Panel دارد که به‌منظور خطایابی نمادین برنامه‌های پایتون ارائه شده است (اگرچه این محیط در حال تکامل است).

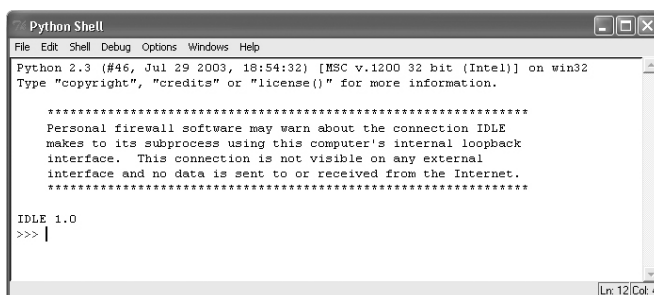
الف-۱-۱- نصب و شروع به کار IDLE

IDLE به‌طور خودکار هنگام سوار کردن Python 1.5.2 و نسخه‌های بالاتر بر روی سیستم شما نصب خواهد شد. کافی است به خاطر داشته باشید در صورتی که در مورد نصب Tcl/Tk سؤالی پرسیده شد، پاسخ مثبت دهید. (چون این کار برای اجرای IDLE ضروری است).

جهت آغاز به کار IDLE در محیط ویندوز از پوشه Python... در منوی Start بر روی IDLE (Python GUI) کلیک کنید. راه دیگر یافتن idle.pyw به‌طور مستقیم و اجرای آن است. برای مثال:

C:\program files\PYTHON23\Tools\idle\idle.pyw

نتیجه، باز شدن پنجره زیر است:



شکل الف-۱

اگر پنجره باز نشد، احتمالاً Tcl/Tk بر روی Path سیستم شما سوار نشده است. برای درست کردن آن در ویندوز 95/98 فایل C:\autoexec.bat را در Notepad باز کنید، خطی که متغیر Path شما را تنظیم می‌کند، پیدا نموده و مسیر C:\PROGRA~1\Tcl\bin را به آن اضافه نمایید. به‌عنوان مثال:

```
PATH=c:\windows;c:\windows\COMMAND;C:\PROGRA~1\Tcl\bin
```

در ویندوز NT و XP می‌توانید Path را از طریق آیکون System در Control Panel تغییر دهید.

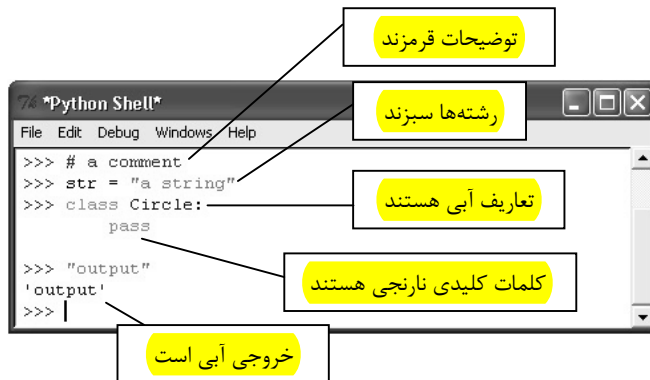
الف-۱-۲- استفاده از پنجره Python Shell

این پنجره هنگامی که شما IDLE را اجرا می‌کنید، ظاهر می‌شود. مانند مد محاوره‌ای مستقیم، شما یک دستور پایتون را در خط فرمان (مقابل >>>) تایپ می‌کنید و کلید Enter را فشار می‌دهید تا آن را به مفسر پایتون ارسال نمایید. برخلاف مد محاوره‌ای باقاعده، هنگامی که وسط یک دستور مرکب چندخطی هستیم، خط فرمان ثانوی (...) نمایش داده نمی‌شود.

اگر خود را در موقعیتی یافتید که به نظر می‌رسید پایتون قفل کرده و نمی‌توانید خط فرمان جدیدی دریافت کنید، احتمالاً مفسر منتظر است تا شما چیز بخصوصی را وارد کنید. فشردن کلیدهای **CTRL + C** یک پیام وقفه صفحه کلید ارسال می‌کند و شما را به خط فرمان برمی‌گرداند. این کلید می‌تواند جهت قطع کردن هر فرمان در حال اجرای دیگر نیز استفاده شود.

الف-۱-۳- رنگ آمیزی

کد شما همچنان که در IDLE تایپ می‌کنید بر اساس انواع نحوی پایتون رنگ آمیزی می‌شود. توضیحات به رنگ قرمزند. رشته‌ها سبز رنگ، تعاریف و خروجی‌های مفسر آبی و کلمات کلیدی پایتون هم نارنجی هستند:

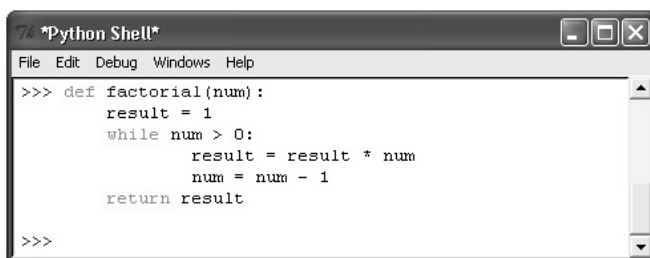


شکل الف-۲

این اکثر آن چیزی است که شما معمولاً نگران آن هستید. هر چند تفاوت‌های دیگری هم میان انواع مختلف خروجی وجود دارد. خروجی **console** (پس‌یابی‌ها) قهوه‌ای و خطاهای استاندارد به رنگ سبز تیره هستند.

الف-۱-۴- کنگره‌گذاری

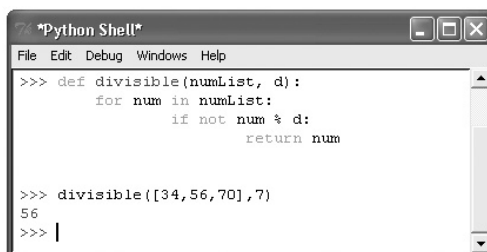
پشتیبانی خودکار برای کنگره‌گذاری در محیط IDLE تدارک دیده شده است. در صورتی که مثال شکل الف-۳ را تایپ کنید، در می‌یابید هرگاه کلید Enter را پس از def یا While بفشارید، خط بعد به‌طور خودکار برای شما کنگره‌گذاری می‌شود تا بلوک جدیدی را وارد کنید. تا زمانی که در یک بلوک هستید شما به‌طور خودکار در تورفتگی برابر با دستور قبلی قرار می‌گیرید. در مثال ما، این اتفاق برای دستور numa=anuma-a1 رخ داده است. سرانجام، با هر بار فشردن کلید BackSpace یک کنگره (تراز) به عقب باز می‌گردید. در مثال زیر، قبل از دستور return از این کلید استفاده شده است. بعد از دستورات pass، return، break، continue و raise شما به‌طور خودکار به اندازه یک کنگره (تراز) به عقب برمی‌گردید.



شکل الف-۳

الف-۱-۵- تکمیل کلمه

تایپ کردن `Alt + /` یک مکانیزم تکمیل لغت را فراخوانی می‌کند. در مثال شکل الف-۴ لازم نیست کلمه `divisible` را در خط آخر به‌طور کامل تایپ کنید. ما می‌توانیم تنها حرف `d` و `Alt + /` را تایپ کنیم تا این کلمه خود به خود کامل شود.

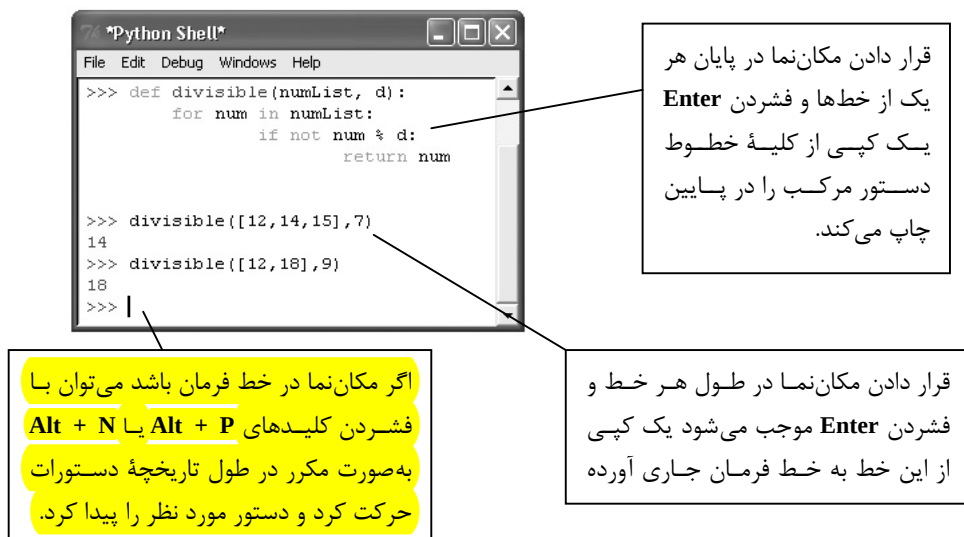


شکل الف-۴

تکمیل کلمه، مبتنی بر کلماتی است که پیش از این وارد بافر شده‌اند. اولین پاسخ، کلمه‌ای است که اخیراً وارد بافر شده و وارد کردن `Alt + /` به‌طور پیاپی، کلمات امکان‌پذیر بعدی را نشان می‌دهد. بنابراین در مثال شکل الف-۴، وارد کردن `Alt + /` برای سه مرتبه کلمات `num`، `not` و سپس `numList` را به‌طور متوالی ظاهر می‌سازد. اگر از `nu` شروع کنید این موضوع به کلمات `num` و `numList` محدود می‌گردد.

الف-۱-۶- تاریخچه دستورات

یکی دیگر از امکانات ارزشمند و پرکاربرد IDLE مکانیزم تاریخچه دستورات است، چرا که شما را از تایپ مجدد دستورات رها می‌سازد:

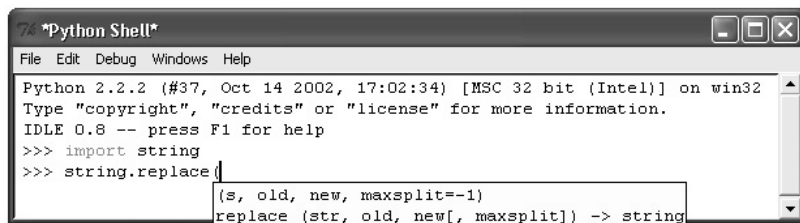


شک

هرگاه یک دستور پیشین را پایین آورید (خواه با استفاده از **Alt+aP** یا **Alt+aN** و خواه به‌صورت مستقیم)، شما می‌توانید آن را به‌طور دلخواه دستکاری و ویرایش نمایید و سپس با فشردن کلید **Enter** آن را مجدداً به مفسر ارسال کنید.

الف-۱-۷- توضیحات در هنگام فراخوانی

به محض اینکه پرانتز را در فراخوانی یک تابع یا متد باز می‌کنید، کادر کوچکی در زیر خط جاری ظاهر می‌شود که اطلاعاتی در باب آرگومان‌های مورد نظر به شما می‌دهد.



شکل الف-۶

این کادر تا زمانی که شما پرانتز را ببندید باقی می‌ماند. این اتفاق برای توابع از پیش ساخته، هر تابع یا متد از ماژول‌های کتابخانه‌ای پایتون (از جمله سازنده‌های کلاس) و همچنین برای هر تابع یا متد کاربر-تعریف رخ می‌دهد.

برای توابع یا متدهایی که شما تعریف کرده‌اید این توضیح، اسامی پارامترهای تعریف شده را نشان می‌دهد. همانطور که در شکل الف-۷ نشان داده شده، خط اول رشته مستندسازی تابع یا متد (خط خلاصه) هم در این توضیح نمایش داده می‌شود. این موضوع می‌تواند به‌منظور دادن اطلاعات در مورد مقدار بازگشتی، مورد استفاده قرار گیرد:

```
Python 2.2.2 (#37, Oct 14 2002, 17:02:34) [MSC 32 bit (Intel)] on win32
Type "copyright", "credits" or "license" for more information.
IDLE 0.8 -- press F1 for help
>>> def divisible(numList, divisor):
    """ divisible(numlist, divisor) -> number"""
    for num in numList:
        if not num % divisor:
            return num

>>> divisible(
    (numList, divisor)
    divisible(numlist, divisor) -> number
```

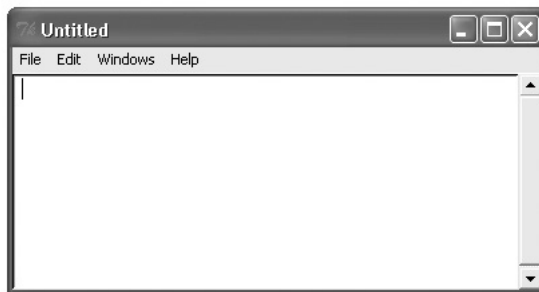
شکل الف-۷

الف-۱-۸- استفاده از ویرایشگر فایل

ویرایشگر فایل IDLE (File Editor) به شما اجازه می‌دهد فایل‌های منبع جدید بسازید و یا فایل موجودی را ویرایش کنید. ویرایشگر فایل می‌تواند از راه‌های گوناگونی باز شود:

کارکرد	کلید در یونیکس	کلید در ویندوز	از طریق منو
باز کردن پنجره جدید	Ctrl+N Ctrl+X	Ctrl+N	File > New...
باز کردن فایل موجود	Ctrl+F Ctrl+X	Ctrl+O	File > Open...
بازکردن ماژول	Ctrl+M Ctrl+X	Ctrl+M	File > Open Module...

گزینه New Window یک پنجره خالی باز می‌کند (مثلاً برای وقتی که می‌خواهید ساختن یک فایل منبع پایتون را از پیش‌نویس آغاز کنید). گزینه Open به شما اجازه می‌دهد هر فایلی را از روی دیسک سخت انتخاب نمایید و آن را باز کنید در صورتی که با استفاده از Open Module شما تنها نیاز به وارد کردن نام فایل ماژول دارید. برای مثال با وارد کردن Shelve یا Shelve.py که بر روی مسیر جستجوی ماژول (sys.path) قرار دارد، این ماژول را پیدا نموده و آن را برای شما باز می‌کند.



شکل الف-۸

File Editor ممکن است از طریق **Path Browser** (مرورگر مسیر) یا هنگام استفاده از **Debugger** هم بالا بیاید (ظاهر شود). در تمام حالات یک ویرایش‌گر فایل جدید، نام فایل و مسیر آن را بر روی نوار عنوان نشان می‌دهد و یا در صورتی که فایلی ذخیره نشده و جدید باشد، **Untitled** خوانده می‌شود.

الف-۱-۸-۱- ویرایش یک فایل

شما می‌توانید در این پنجره تایپ کنید و کدی که تایپ می‌کنید مانند **Shell Window** رنگ‌آمیزی می‌شود. کنگره‌گذاری خودکار نیز به‌طور مشابهی رخ می‌دهد. هر کنگره به اندازه چهار کاراکتر فاصله است که این اندازه، فاصله استاندارد پایتون برای فایل‌های منبع است. در حالی که این فاصله در **Shell Window** به اندازه یک **Tab** است. کلیه اعمال، کلیدها، امکانات و ویرایش از جمله **Paste, Copy, Cut** و ... در این محیط نیز همچون **Shell Window** کار می‌کنند. تنها تفاوت‌ها نبود توضیحات فراخوانی و البته تاریخچه دستورات است. به علاوه امکانات ویرایشی دیگری هم تدارک دیده شده که در این پنجره کاربرد زیادی دارند و در جدول صفحه بعد آورده شده‌اند:

عملیات	کلید	از طریق منو
کنگره‌گذاری یک منطقه	Ctrl +]	Edit > Indent region
برگرداندن کنگره یک منطقه	Ctrl + [	Edit > Dindent region
تبدیل یک منطقه به توضیح	Alt + 3	Edit > Comment out region
خارج کردن منطقه از حالت توضیح	Alt + 4	Edit > Uncomment region
تبدیل منطقه به Tab	Alt + 5	Edit > Tabify region
خارج کردن منطقه از حالت Tab	Alt + 6	Edit > Untabify region

امکانات کنگره‌گذاری هنگام اضافه کردن یا حذف الگوهای تو در تو به‌طور آشکاری سودمند است. گزینه Tabify region فواصل عمده هر خط را به Tab تبدیل می‌کند (هر هشت فاصله به یک Tab تبدیل می‌گردد). Untabify region تمام Tab‌های درون منطقه انتخاب شده را به تعداد فواصل صحیح تبدیل می‌کند، یعنی عکس عمل قبل.

ممکن است کد شما از منابع گوناگونی مانند فایل‌های متنی یا صفحات وب به ویرایشگر فایل کپی شده باشد. بنابراین این ابزار به همراه امکانات کنگره‌گذاری، جهت تبدیل کد به یک قالب کد منبع استاندارد بسیار مفیدند.

الف-۱-۸-۲- ذخیره‌سازی یک فایل

اگر عنوان یک بافر (نشان داده شده بر روی نوار عنوان) بوسیله علامت‌های * محصور شده باشد، یعنی نسبت به آخرین زمانی که باز شده تغییر کرده است مثلاً اگر فایلی را باز کنید و بخواهید در آن تغییراتی ایجاد نمایید به محض اینکه اولین کلید را بفشارید عنوان این فایل بر روی نوار عنوان با علامت‌های * در ابتدا و انتهای آن نشان داده می‌شود. در این پنجره سه دستور برای ذخیره‌سازی فایل وجود دارد. گزینه Save فایل را با نام کنونی ذخیره می‌سازد. Save as این امکان را می‌دهد که نامی جدید را وارد کنید و راه سوم (Save Copy As) مانند Save As عمل می‌کند با این تفاوت که نام کنونی فایل را تغییر نمی‌دهد.

کارکرد	کلید در یونیکس	کلید در ویندوز	از طریق منو
Save	Ctrl+S Ctrl+X	Ctrl+S	File > Save
Save As	Ctrl+W Ctrl+X	Alt+S	File > Save As ...
Save Copy As	Ctrl+XW	Alt+Shift+S	File > Save Copy As ...

الف-۱-۸-۳- اجرای یک فایل

شما می‌توانید یک ماژول را به این صورت توسعه دهید که ابتدا آن را import نموده و سپس جهت آزمایش تغییرات ایجاد شده، آن را به درون Shell Window مجدداً بارگذاری (Reload) کنید. این کار می‌تواند با استفاده از فرمان Import Module از خود File Editor و یا با استفاده از دستورات import و reload از درون Shell Window انجام شود.

عملیات	کلید	از طریق منو
Import یا بارگیری مجدد	F5	Edit > Import Module
اجرای یک اسکریپت	Ctrl + F5	Edit > Run Script

عمل Import module یک ماژول را import می‌کند و یا آن را مجدداً بارگذاری می‌کند (Reload)؛ سپس آن را به Shell Window می‌فرستد یا در صورت لزوم باز می‌کند. چاره دیگر برای توسعه، این است که کدتان را به تابعی مجهز کنید تا آزمایشی را که می‌خواهید انجام دهد و از فرمان Run Script استفاده کنید. خروجی باز هم به Shell Window ارسال می‌شود. این به شما امکان می‌دهد در یک چرخه متناوباً «تغییرات را در کدتان اعمال کنید»، «کلیدهای Ctrl + S را بفشارید» تا تغییرات ذخیره شود و سپس «کلیدهای Ctrl + F5 را بفشارید» تا نتایج را مشاهده کنید.

الف-۱-۹- استفاده از پنجره‌های محاوره‌ای Find/Replace

گزینه Find... در منوی Edit پنجره محاوره‌ای شکل الف-۹ را باز می‌کند:



شکل الف-۹

جستجو در حالت Wrap around و Dawn (حالت پیش‌فرض) باعث می‌شود که بافر از محل قرار گرفتن مکان نما به سمت پایین (تا انتهای بافر) جستجو شود و در صورت پیدا نشدن عبارت،

جستجو از بالای فایل ادامه می‌یابد. اگر گزینه `regularaexpression` انتخاب شده باشد، عبارت شما به‌عنوان یک مبین منظم در نظر گرفته می‌شود.

`MatchaCase` باعث می‌شود کلمه شما از لحاظ حروف بزرگ یا کوچک دقیقاً همانطوری که نوشته‌اید جستجو شود و انتخاب `WholeaWord` موجب می‌شود عبارت شما به‌عنوان یک کلمه کامل جستجو شود، نه قسمتی از کلمات.

گزینه `Replace...` از منوی `Edit` پنجره شکل الف-۱۰ را باز می‌کند. این پنجره شبیه به پنجره `Search Dialog` است؛ به‌علاوه جعبه متنی برای وارد کردن عبارت جایگزین و چهار دکمه دارد که دارای کارکردهای واضحی هستند.

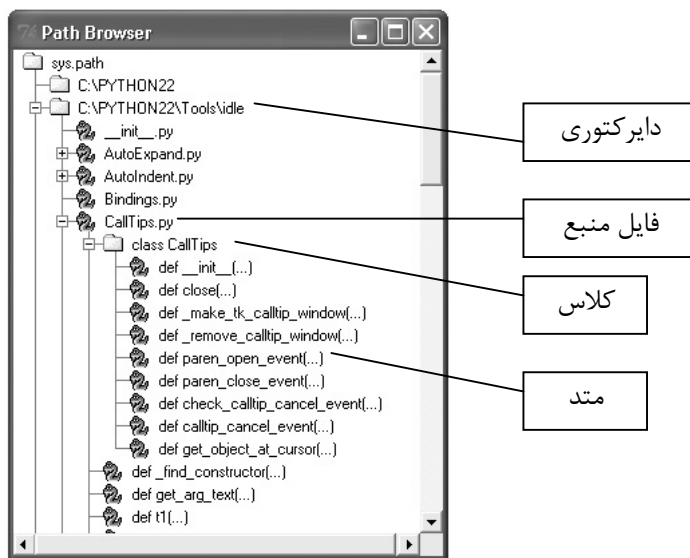


شکل الف-۱۰

گزینه `Find in Files...` از منوی `Edit` پنجره زیر را باز می‌کند. اگر این گزینه را از درون `IDLE` انتخاب کنید، در پنجره‌ای که باز می‌شود کلیه فایل‌هایی که روی مسیر جستجوی پایتون قرار دارد (`*.py`) به‌طور پیش‌فرض جستجو خواهد شد. در صورتی که این گزینه را از درون پنجره ویرایشگر فایل مسیر جستجوی پیش‌فرض پوشه‌ای است که فایل حاضر در آن ذخیره شده است. انتخاب گزینه `Recurse down subdirectories` باعث می‌شود تمام فایل‌های موجود در زیردایرکتوری‌های مسیر داده شده نیز جستجو گردند.

الف-۱-۱- استفاده از مرورگر مسیر (Path Browser)

مرورگر مسیر (`Path Browser`) به شما امکان می‌دهد دایرکتوری‌های مسیر سیستم پایتون را جهت یافتن ماژول‌ها جستجو کنید. همان‌طور که در شکل زیر نشان داده شده دایرکتوری‌ها، فایل‌های منبع، کلاس‌ها و متدها در این پنجره به صورت درختی نمایش داده شده است (شکل الف-۱۱).



شکل الف-۱۱

الف-۲- PythonWin

PythonWin محیطی است که بر خلاف IDLE تنها در سیستم عامل ویندوز قابل استفاده است. این رابط گرافیکی کاربر دارای یک اشکال‌یاب مجتمع و یک محیط ویرایش توانمند پایتون است.

الف-۲-۱- پنجره تعاملی

پنجره تعاملی (Interactive) یک برنامه کوچک پایتون است که مفسر از پیش ساخته پایتون را شبیه‌سازی می‌کند. در این پنجره هرگاه کلید Enter را می‌فشارید، متن شما بررسی می‌شود و در مورد اینکه چه کاری انجام شود، تصمیم گرفته می‌شود.

اگر خط جاری به عنوان یک بلاک شناخته شود (خط اول با ">>>" آغاز می‌شود و بقیه خطوط با "...") به انتهای بافر کپی می‌شود اما چیزی اجرا نمی‌گردد. با فشردن مجدد کلید Enter بلاک اجرا می‌شود. اگر در انتهای بافر باشیم Enter همیشه موجب اجرای خط می‌شود. هرگاه پایتون تشخیص دهد بلاک تمام نشده فشردن Enter باعث نمایش "... می‌شود.

هرگاه خط وارد شده دارای خطا باشد، فایل و خط حاوی خطا مشخص می‌گردد. شما می‌توانید با استفاده از گزینه Undo از منوی Edit، دکمه Undo بر روی نوار ابزار و یا کلید میانبر Ctrl+Z آخرین عمل از جمله ارسال خط به مفسر را لغو کنید.

الف-۲-۲- منوها و نوار ابزار

عملکرد گزینه‌های Open ، Close ، Save و Print واضح است. Locate File به شما امکان می‌دهد اسکریپت‌های پایتون را به سادگی بیابید. با تایپ کردن نام یک فایل می‌توانید تمام دایرکتوری‌های موجود بر روی مسیر sys.path را جستجو کنید.

گزینه Import یک اسکریپت را Import می‌کند و یا آن را مجدداً بارگذاری می‌کند. PythonWin تعیین می‌کند که در مورد این فایل عمل Import اجرا شود یا Reload .

بر روی نوار ابزار، دکمه‌هایی به منظور انجام اعمال Open ، Save ، Import/Reload ، Undo ، Redo ، Run ، Cut ، Copy ، Paste ، Print و ... در نظر گرفته شده که دسترسی به گزینه‌ها را سرعت می‌بخشند.

پیوست ب

فطاهای برنامه نویسی



در مدت یادگیری پایتون احتمالاً با هر سه نوع خطاهای برنامه‌نویسی آشنا می‌شوید، اما دو نوع از این خطاها بیشتر قابل تشخیص هستند.

ب-۱- خطاهای نحوی

خطاهای نحوی که به عنوان خطاهای زمان تجزیه هم شناخته می‌شوند، شاید متداول‌ترین خطاهایی باشند که شما با آن برخورد می‌کنید:

```
>>> while 1 print "Hello world!"
Traceback ( File "<interactive input>", line 1
      while 1 print "Hello world!"
              ^
SyntaxError: invalid syntax
```

تجزیه‌گر (که قسمتی از مفسر است) خط نادرست را تکرار می‌کند و یک پیکان کوچک در نزدیک‌ترین نقطه به محل تشخیص خطا نمایش می‌دهد. توکنی که قبل از علامت پیکان آمده است منجر به خطا شده است (یا لاقلاً خطا در آن شناخته شده است). در مثال فوق، خطا در کلمه کلیدی **print** تشخیص داده شده، چرا که علامت کولن (:) بعد از دستور **print** از قلم افتاده است. همچنین نام فایل و شماره خط در پیغام خطا چاپ شده تا در صورتی که ورودی از یک اسکریپت می‌آید، شما بدانید در کجا به دنبال خطا بگردید.

ب-۲- اعتراض‌ها

حتی اگر دستور (یا عبارت) از لحاظ نحوی درست باشد، ممکن است وقتی قصد اجرای آن را دارید منجر به خطا شود. خطاهایی که در طول اجرای برنامه رخ می‌دهند، اعتراض نامیده می‌شوند و مطلقاً خطاهای مهلکی نیستند. شما به زودی می‌آموزید که چگونه آنها را کنترل کنید. اغلب اعتراضات توسط برنامه‌ها مدیریت نمی‌شوند (به هر دلیل) و پیغام خطایی را به این صورت نتیجه می‌دهند:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<interactive input>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero

>>> 4 + Spam*3
Traceback (most recent call last):
  File "<interactive input>", line 1, in ?
NameError: name 'Spam' is not defined
```

```
>>> '2' + 2
```

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

```
File "<interactive input>", line 1, in ?
```

```
TypeError: cannot concatenate 'str' and 'int' objects
```

خط آخر پیغام خطا مشخص می‌کند که چه اتفاقی افتاده است.

اعتراض‌ها انواع گوناگونی دارند که نوع آنها در قسمتی از پیغام چاپ می‌شود. انواع ذکر شده در مثال فوق عبارتند از **ZeroDivisionError**، **NameError** و **TypeError**. رشته‌ای که به عنوان نوع اعتراض چاپ شده، اسم پیش‌ساخته‌ای برای اعتراضی که رخ داده است، می‌باشد. این نحوه در مورد تمام اعتراض‌های پیش‌ساخته صحیح است، اما صحت آن در مورد اعتراض‌های کاربر-تعریف لزومی ندارد (اگرچه قرارداد مفیدی است). اسامی اعتراض‌های استاندارد شناسه‌های پیش‌ساخته‌ای هستند که نوع و هویت خطا را مشخص می‌کنند (و نه کلمات کلیدی رزرو شده در زبان پایتون).

بقیه خط، جزئیاتی است که تفسیر و معنای آن بستگی به نوع اعتراض دارد.

قسمت قبلی پیغام خطا، زمینه‌ای در مورد محل وقوع خطا به فرم پس‌یابی یک پشته ارائه می‌دهد. به‌طور کلی این پس‌یابی پشته‌ای خطوط مبدأ برنامه را لیست، اگرچه خطوط خوانده شده از ورودی استاندارد نمایش داده شده نمی‌شوند.

مرجع کتابخانه پایتون، تمام اعتراض‌های پیش‌ساخته و معانی آنها را فهرست کرده است.

ب-۳- کنترل اعتراض‌ها

در پایتون، نوشتن برنامه‌هایی که برخی از اعتراض‌ها را کنترل کند، امکان‌پذیر است. به این مثال نگاه کنید:

```
>>> while 1:
...     try:
...         x = int(raw_input("Please enter a number: "))
...         break
...     except ValueError:
...         print "Oops! That was no valid number. Try again..."
... 
```

در این مثال تا زمانی که یک ورودی معتبر وارد شود، وارد کردن یک عدد از کاربر درخواست می‌شود، اما اجازه می‌دهد که او (با استفاده از **Ctrl+C** یا هر چیز دیگری که سیستم عامل آن را پشتیبانی می‌کند) برنامه را قطع کند. توجه کنید که یک وقفه تولید شده توسط کاربر به‌وسیله تولید یک اعتراض **KeyboardInterrupt** علامت داده می‌شود.

الگوی استفاده از دستور **try** به‌صورت زیر است:

- ابتدا بند **try** اجرا می‌شود (دستور(های) میان دو کلمهٔ کلیدی **try** و **except**)
- اگر هیچ اعتراضی رخ ندهد بند **except** نادیده می‌شود و اجرای دستور **try** پایان می‌یابد.
- اگر در طول اجرای بند **try** اعتراضی رخ ندهد مابقی بند رها می‌شود، آنگاه در صورتی که نوع آن با نام اعتراضی که پس از کلمهٔ کلیدی **except** آمده مطابقت کند، باقی بند **except** اجرا می‌شود و سپس اجرا از دستور بعدی **try** ادامه می‌یابد.
- اگر اعتراضی رخ دهد که با اعتراض نام برده شده در بند **except** مطابقت نکند، کنترل به دستورات **try** بیرونی‌تر انتقال می‌یابد و در صورتی که کنترل‌کنندهٔ دیگری وجود نداشته باشد یک اعتراض کنترل نشده رخ داده و اجرا با پیغامی (که نمونهٔ آن را در بالا دیدید) متوقف می‌گردد.

یک دستور **try** ممکن است به منظور کنترل و مدیریت اعتراضات گوناگون، بیش از یک بند **except** داشته باشد اما تنها یکی از چند بند موجود اجرا شده خواهد شد. کنترل‌کننده‌ها تنها اعتراضی را کنترل می‌کنند که در بند **try** متناظرشان اتفاق افتاده باشد، نه در کنترل‌کنندگان دیگری که در همان دستور **try** قرار دارند.

یک بند **except** ممکن است چندین اعتراض را در قالب یک لیست پراانتزگذاری شده عنوان کند. برای مثال:

```
... except ValueError:
...     pass
```

اگر نام‌های اعتراض در مقابل دستور **except** حذف شوند، این بند **except** برای تمامی اعتراض‌ها عمل می‌کند. بنابراین از این خصوصیت با احتیاط بسیار زیادی استفاده کنید. زیرا این روش به راحتی یک خطای واقعی برنامه‌نویسی را پنهان می‌کند. این دستور می‌تواند به منظور چاپ یک پیغام خطا و تولید مجدد اعتراض نیز به کار برده شود:

```
import string, sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(string.strip(s))
except IOError, (errno, strerror):
    print "I/O error(%s): %s" % (errno, strerror)
except ValueError:
```

```
print "Could not convert data to an integer."
except:
    print "Unexpected error:", sys.exc_info()[0]
    raise
```

دستور **try...except** می‌تواند یک بند **else** اختیاری هم داشته باشد که در صورت استفاده از آن باید پس از تمام بندهای **except** قرار گیرد. این امکان، برای کدهایی که در عدم وقوع اعتراض باید اجرا شوند، مفید است. برای مثال:

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print 'cannot open', arg
    else:
        print arg, 'has', len(f.readlines()), 'lines'
        f.close()
```

استفاده از **else** بهتر از افزودن کدهای اضافی به قسمت **try** است، زیرا به‌طور تصادفی از تولید یک اعتراض جلوگیری می‌کند. اعتراضی که توسط کدِ محافظت شده به‌وسیلهٔ دستور **try...except** تولید نشده است.

هنگامی که یک اعتراض رخ می‌دهد، ممکن است مقدار وابسته‌ای هم داشته باشد که این مقدار با نام آرگومان اعتراض هم شناخته می‌شود. وقوع و نوع آرگومان به نوع اعتراض بستگی دارد. برای انواع اعتراضاتی که یک آرگومان دارند، ممکن است عبارت **except** متغیری پس از نام (یا لیست) اعتراض داشته باشد تا مقدار آرگومان را دریافت کند. مانند:

```
>>> try:
...     spam()
... except NameError, x:
...     print x
...
name 'spam' is not defined
```

اگر اعتراضی یک آرگومان داشت، این آرگومان به عنوان قسمت آخر (جزئیات) پیغام اجرای اعتراض کنترل نشده چاپ می‌شود. کنترل‌کننده‌های اعتراض تنها اعتراضاتی را که در قسمت **try** رخ داده‌اند کنترل نمی‌کنند، بلکه اگر اعتراضاتی درون توابع فراخوانده شده توسط دستور **try** هم رخ داده دهد کنترل می‌نمایند. برای مثال:

```
>>> def this_fails():
...     x = 1/0
...
>>> try:
...     this_fails()
... except ZeroDivisionError, detail:
...     print 'Handling run-time error:', detail
...
Handling run-time error: integer division or modulo by zero
```

ب-ع- تولید اعتراض‌ها

دستور **raise** به برنامه‌نویسان اجازه می‌دهد تا اعتراض معینی را به اجبار تولید کنند:

```
>>> raise NameError, 'Hi there'
Traceback (most recent call last):
  File "<interactive input>", line 1, in ?
NameError: Hi there
```

اولین آرگومان **raise** نام اعتراضی را که باید تولید شود مشخص می‌کند و آرگومان دوم اختیاری است، آرگومان اعتراض را تعیین می‌کند. اگر نیاز داشته باشید که تعیین کنید اعتراضی تولید شود اما نخواهید آن اعتراض کنترل گردد، فرم ساده‌ای از دستور **raise** شما را قادر می‌سازد که آن را دوباره تولید کنید:

```
>>> try:
...     raise NameError, 'Hi there'
... except NameError:
...     print 'An exception flew by!'
...     raise
...
An exception flew by!
Traceback (most recent call last):
  File "<interactive input>", line 2, in ?
NameError: Hi there
```

ب-د- اعتراض‌های کاربر-تعریف

برنامه می‌تواند اعتراضات خودشان را با نسبت‌دهی یک رشته به یک متغیر و یا ساختن یک کلاس اعتراض جدید، نام گذاری کنند. برای نمونه:

```
>>> class MyError(Exception):
...     def __init__(self, value):
...         self.value = value
...     def __str__(self):
...         return `self.value`
...
>>> try:
...     raise MyError(2*2)
... except MyError, e:
...     print 'My exception occurred, value:', e.value
...
My exception occurred, value: 4
>>> raise MyError, 'oops!'
Traceback (most recent call last):
  File "<interactive input>", line 1, in ?
MyError: 'oops!'
```

ماژول‌های استاندارد زیادی از این مطلب جهت گزارش خطاهایی که درون توابعشان رخ می‌دهد، استفاده می‌کنند.

ب-۶- تعریف اعمال پایانی

دستور **try** یک عبارت اختیاری دیگر هم دارد که به‌منظور تعریف اعمال نهایی در نظر گرفته شده است. منظور از این اعمال، دستوراتی است که در هر شرایطی باید اجرا شوند. برای مثال:

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print 'Goodbye, world!'
...
Goodbye, world!
Traceback (most recent call last):
  File "<interactive input>", line 2, in ?
KeyboardInterrupt
```

بند **finally**، چه در حالتی که اعتراضی در بند **try** رخ دهد و چه در حالتی که هیچ اعتراضی رخ ندهد، اجرا می‌شود. هنگامی که یک اعتراض رخ می‌دهد این اعتراض پس از اجرای بند **finally** مجدداً تولید می‌شود.

همچنین در صورت رها شدن دستور **try** از طریق دستور **break** یا **return**، بند **finally** در هنگام خروج اجرا می‌شود.

کدی که در بند **finally** قرار دارد برای انتشار منابع خارجی (مانند فایل‌ها و اتصالات شبکه) مفید است، صرف‌نظر از اینکه استفاده از منابع موفقیت آمیز بوده است یا نه. یک دستور **try** باید یک یا چند بند **except** داشته باشد و یا یک بند **finally**، اما نمی‌تواند هر دو را در بر گیرد.

فهرست منابع

Downey Allen B., Elkner Jeffrey, Meyers Chris; Wellesley, Massachusetts; April 2002 ; *How to Think Like a Computer Scientist: Learning with Python*

Pilgrim, Mark ; January 2002 ; *Dive Into Python*

v.Rossum Guido, Drake Fred L. ; December 2001 ; *Python Tutorial*

<http://www.python.org>

<http://www.ibiblio.org/obp/thinkCSPy>

<http://www.orgmf.com.ar/condor/pyshelf22.zip>

<http://www.DiveIntoPython.org>

<http://www.BruceEckle.com>



<http://www.NowPython.com>



A Powerful, High-level,
Object-oriented, Easy-to-learn
Programming Language