تالیف: رامان عشقی، ۰۹۳۵۷۷۰۵۰۶۵ برای فیلم های آموزش پایتون به http://www.blue-python.tk مراجعه کنید.

آموزش برنامه نویسی به زبان پایتون، از مقدماتی تا پیشرفته

نام کتاب: پایتون آبی، نسخه ۱.۱

برای دریافت ویدئوهای آموزشی پایتون، کلاس ها و مقالات دیگر

به وبسایت رسمی آموزش برنامه نویسی پایتون یعنی

http://www.blue-python.tk مراجعه نمایید

** مولف: رامان عشقی **

** ایمیل مولف: raman_diyar@yahoo.com **

** شماره تماس مولف: ۰۹۳۵۷۷۰۵۰۶۵ **

** وبسایت مولف: http://www.blue-python.tk **

توجه: برای ارتباط و یا همکاری با مولف این کتاب (رامان عشقی)

می توانید با همراه شخصی ایشان به شماره ۰۹۳۵۷۷۰۵۰۶۵

تماس گرفته یا از طریق وبسایت پایتون آبی به نشانی

http://www.blue-python.tk اقدام نمایید.

تالیف: رامان عشقی، ۰۹۳۵۷۷۰۵۰۶۵ برای فیلم های آموزش پایتون به http://www.blue-python.tk مراجعه کنید.

به نام خداوندی که ذهن مان را روشن نمود

پیشگفتار

در دنیایی که نمی توان آینده ئ آن را به نفع خود تغییر داد لذا باید جزوی از آینده شد. آینده ئ دنیای برنامه نویسی چیست؟ آینده در دست کدام زبان برنامه نویسی یا کدام فناوری است؟ جواب این سوالات هر چه که باشد شما به آن نیاز ندارید! آینده هر چه که باشد به نفع افراد حرفه ای خواهد بود و نه ضعیف ها، این قانون بوده و خواهد بود. نمی توانم بگویم در آینده کدام زبان برنامه نویسی یا کدام فناوری بیشتر استفاده خواهد شد ولی می توانم با صراحت بگویم کدام افراد بیشتر استفاده خواهند شد: "حرفه ای ها". پس اگر به سراغ پایتون آمده اید و به این زبان علاقه دارید باید بگویم احتمالا این زبان به زودی به زبان فکر شما تبدیل خواهد شد و احتمالا در یک سال آینده به طرز باور نکردنی ای در این زبان پیشرفت خواهید کرد طوری که در هیچ وهله ای از عمرتان همچین سرعتی را در یادگیری مشاهده نکرده باشید. البته باید تلاش خودتان را هم بکار بگیرید. پس حرفه ای شوید و صاحب آینده باشید.

درباره ئ خودم

اولین کدی از پایتون که مشاهده کردم در دوره ئ دوم دبیرستانم بود، از آن زمان تا به امروز که امروز که این کتاب را منتشر کرده ام بیشتر از شش سال می گذرد. من و پایتون هر دو تغییر کرده ایم، من شش سال پیرتر شدم و پایتون شش سال پویاتر و قدرتمند تر. اما هنوز همان زبانی است که کدهایش در اولین نگاه مرا شیفته

خود کرد. متاسفانه در آن زمان مطلقا هیچ منبعی برای یادگیری پایتون به زبان فارسی وجود نداشت و صد البته که من دسترسی به کتب کاغذی زبان اصلی را هم نداشتم. تنها یاری کننده ام مستند سازی های زبان پایتون بود که توسط سایت رسمی آن متنشر می گردند. در ابتدا این مستند سازی ها را بسیار خشک و بی روح دیدم اما با گذشت زمان به آن ها علاقمند شدم. بالاخره من پایتون را از روی همین مستندات آموختم. در چند ماه اخیر به فکر نوشتن سری مقالاتی درباره زبان پایتون افتادم که آنها را در وبسایتم قرار دهم اما به زودی متوجه شدم می توان این مقالات را سازماندهی و منتشر کرد و تالیف کتاب پایتون آبی را آغاز نمودم. این نسخه ای که در دست شماست احتمالا دارای اشتباهات سهوی متعددی از جمله اشتباهات املایی و البته احتمالا چند اشتباه فنی است که در حال اصلاح آنها هستم به هر حال شما همواره می توانید به روز ترین و آخرین ویرایش کتاب پایتون آبی را از سایت پایتون آبی (که آدرس آن در footer صفحه هست) دریافت کنید. لطفا اشتباهاتی را که در کتاب پایتون آبی مشاهده می کنید را با من توسط ایمیلم raman_diyar@yahoo.com در میان بگذارید، در ضمن اگر نمی دانید از کجا شروع کنید و به هر حال مشاوره ای درباره ئ برنامه نویسی پایتون نیاز دارید می توانید از طریق شماره ئ ۰۹۳۵۷۷۰۵۰۶۵ با من تماس بگیرید و پاسختان را بگیرید.

لازم به ذکر می دانم این کتاب اولین و آخیرن اثر آموزشی من در زمینه ئ پایتون نبوده بلکه آثار متعددی از جمله فیلم های آموزشی و مقالات آموزشی در وبسایت پایتون آبی قرار داده ام که می توانید آنها را دریافت کنید. در حال حاضر در وبسایت پایتون آبی مشغول خدمت کردن به شما دوستان عزیز هستم. پس از سر زدن به این سایت دریغ نکنید. یک توصیه به وب مسترهایی که احتمالا این کتاب را برای دانلود در سایت خود قرار داده اند می کنم، لطفا pdf این کتاب را دستکاری نکنید، من برای تالیف این کتاب حدود یک سال تلاش کردم و آن را برای هم وطنان خود به صورت رایگان در وب قرار داده ام پس لطفا به نیت و تلاش من در این زمینه احترام بگذارید.

در انتها این کتاب را به عزیزترین شخص زندگیم تقدیم می کنم.

رامان عشقی

تالیف: رامان عشقی، ۰۹۳۵۷۷۰۵۰۶۵ برای فیلم های آموزش پایتون به http://www.blue-python.tk مراجعه کنید.

فهرست مطالب

تالیف: رامان عشقی، ۰۹۳۵۷۷۰۵۰۶۵ برای فیلم های آموزش پایتون به http://www.blue-python.tk مراجعه کنید.

فصل اول:

مقدمات پایتون

مرجع آموزش پایتون در ایران

http://www.blue-python.tk

مولف: رامان عشقی

بی شک برای شروع برنامه نویسی به هر زبان و روش و متدی باید از یک مثال ساده آغاز کرد. دراین کتاب از مثال معروف hello world بهره گرفته ایم. از منوی استارت پوسته فرمان پایتون را باز کرده و عبارت زیر را تایپ کنید:

```
print('hello world')
```

و سپس Enter را بزنید. چنانچه دستور را درست تایپ کرده باشید و راهنمایی های این کتاب را در قسمت های قبل درست انجام داده باشید عبارت hello world در یک خط پایین تر از دستوری که نوشته اید برای شما چاپ می شود.

توجه: اگر نسخه پایتونی که نصب کرده اید به جای 3x، 2x باشد خطا خواهید گرفت. اگر چنین کرده اید قبل از ادامه کتاب نسخه درست پایتون را نصب کنید.

همان طور که متوجه شده اید کار print چاپ کردن است. اما چاپ کردن چه چیزی؟ برای دانستن این مطلب ابتدا به دانشی درباره انواع متغیرها و اشیا و دقیقتر بگوییم چیزهایی که در پایتون قابل تغییرند باید اشاره کرد.

توابع برنامه نویسی در نگاه اول

تابع در برنامه نویسی به قطعه ای از کد گفته می شود که برای وظیفه خاصی نوشته می شود، فرق آن با بقیه بدنه برنامه ما این است که هر گاه بخواهیم آن را فراخوانی می کنیم تا اجرا گردد. نه اینکه هر زمان که اینترپرتر به آن برخورد کرد منتظر یک دستور شرطی یا چیز دیگری برای تصمیم گیری اجرای آن باشد. print() یک تابع داخلی پایتون است. منظور از توابع داخلی پایتون بیشتر توابعی است که بدون استفاده از دستور import قابل استفاده و در دسترس هستند. تابع print() وظیفه چاپ کردن چیزی را دارد که داخل پرانتز به آن داده می شود. در قسمت های بعدی بیشتر درباره آن صحبت خواهیم کرد.

متغیرها در پایتون

در یک کلام متغیرها مکانهایی در حافظه هستند که اطلاعات در آنها ذخیره می شوند. اما درپایتون اوضاع کمی متفاوت از زبان سی(C) است و به متغیرها مرجع شئ هم گفته می شود. در آینده خواهید فهمید که همه چیز در پایتون یک شئ است از جمله متغیرهای پایتون (درباره شئ گرایی بعدا توضیح داده می شود).

متغیرها در پایتون همان مفهمومی دارند که در سی دارند با فرق اینکه لازم نیست تعریف کنیم که این متغیر از چه کلاسی است اینترپرتر پایتون خودش خواهد فهمید که هر کدام از چه نوع هستند. در پوسته پایتون تایپ کنید: myName='ali' و اینتر را بزنید. شما با اینکار خود یک متغیر به نام myName تعریف کرده اید که مقدار 'ali' را در خود دارد. به دو علامت نقل قول اطراف آن توجه کنید. این علامتها نشان دهنده آن هستند که این یک متغیر string یا رشته ای است. متغیرهای رشته ای از کاراکتها تشکیل شده اند. در سیستم اسکی ۲۵۶ کاراکتر داریم که هر کدام یک شماره یا اندیس دارند که اندیس آنها از ۰-۲۵۵ است. این سیستم استاندارد جهانی است اما اگر شما بخواهید به زبان خودتان یعنی فارسی برنامه نویسی کنید مثلا به جالی 'ali' بنویسید 'علی' باید از نوعی سیستم دیگر مانند utf-8 استفاده کنید.

string یکی از انواع اصلی است به جز آن integer هم وجود دارد: myAge=23 اگر این عبارت را در پوسته پایتون استفاده کنید یک متغیر اینتجر یا رقم صحیح ساخته اید که مقدار آن ۲۳ است. اینتجرهای پایتون می توانند تا هر چقدر که سیستم شما ظرفیت داشته باشد رقم بیشتر در خود ذخیره کنند مثلا اگر سیستم بتواند ۹۹۹۹۹۹۹۹۹۹۹۹۹۹۹۹۹۹ را نگه دارد متغیر مذکور آن را می پذیرد و نگه می دارد. اما اگر داده ما عدد صحیح نباشد میتوانیم آن را شناور یا float بنامیم. شناور ها اعداد اعشاری را ذخیره می کنند. عملا کار شما فرقی نمی کند چون باید فقط مقدارتان را بنویسید و ایترپرتر پایتون خود متوجه می شود که از کدام نوع (type) استفاده کند. برای مثال شما می توانید عبارت newVar=45.33 را در پوسته استفاده کنید که نتیجه آن ساخته شدن یک متغیر جدید است از نوع شناور.

این انواع اصلی قابل تبدیل شدن به یکدیگر هستند. به مثال های زیر دقت کنید:

```
>>> a='2354.4'
>>> b=int(float(a))
>>> c=float(b)
```

خط اول a یک مقدار اعشاری گرفته، خط دوم از دو تابع int() و float() استفاده شده است. اینها جزو توابع داخلی پایتون هستند. تابع float() یک مقدار اعشاری، صحیح یا رشته ای قابل تبدیل به اعشاری را می گیرد و یک عدد اعشاری بر می گرداند. و تابع int() هم یک مقدار صحیح، اعشاری یا رشته ای قابل تبدیل به صحیح را گرفته و صحیح برمیگرداند. خط سوم هم c مقدار اعشاری b را گرفته است.

برای فهمیدن اینکه هر کدام از متغیرهایی که تعریف کرده ایم از چه نوعی هستند می توانیم از تابع type() استفاده کنیم:

```
>>> type(a)
<class 'str'>
>>> type(b)
<class 'int'>
>>> type(c)
<class 'float'>
```

همان طور که می بینید a از کلاس(نوع) str یا همان رشته ای است و b و c هم به همین ترتیب نوعشان نشان داده شده است. درضمن از تابع str() می توان برای تبدیل هر نوع اعشاری یا صحیح به استرینگ(رشته ای) استفاده کرد مانند مثال زیر:

```
>>> myString=str(23.4)
>>> type(myString)
<class 'str'>
```

از تابع print() هم می توانید برای چاپ کردن این مقادیر و متغیرها در خروجی استفاده کنید. مانند:

```
>>> print(a)
2354.4
>>> print(a, b, c, myString)
2354.4 2354 2354.0 23.4
```

همان طور که می بینید در خط سوم مثال بالا ما هر چهار متغیری که تا بحال تعریف کرده بودیم را با هم و پشت سر هم چاپ کردیم.

توجه: شما می توانید برای ساختن رشته ها یا همان استرینگ ها از علامت ' یا " استفاده کنید، به این معنا که پایتون بین این دو هیچ فرقی قائل نمی شود.

کدنویسی

تا به حال هر مثالی را که دیده اید در پوسته پایتون انجام داده ایم، این یک عمل متقابل است یعنی شما یک دستور را در پوسته پایتون تایپ می کنید و سپس چیزی که دستور برمی گرداند را مشاهده می کنید. اما این فقط جنبه آموزشی دارد، برای برنامه نویسی شما باید دستورات خود را در یک فایل جداگانه ذخیره کرده و سپس بوسیله اینترپرتر (نرم افزار) پایتون آنها را اجرا کرده و نتیجه را ببینید. مثال زیر گرچه هنوز مفاهیمی دارد که ممکن است

```
>>> type(a)
<class 'str'>
>>> type(b)
<class 'int'>
>>> type(c)
<class 'float'>
```

از آنها سر در نیاورید اما به شما کمک می کند تا درک کنید که چگونه یک قطعه کد نقش یک برنامه را بازی می کند.

ابتدا یک نرم افزار ویرایش متن را باز کنید، در این مقطع توصیه ما نرم افزاری مانند programmer's notepad یا notepad++ است. کدهای زیر را در آن تایپ کنید:

توجه: استفاده از notepad معمولی ویندوز توصیه نمی شود، زیرا امکانات syntax highlighting را ندارد. نکته دیگر اینکه از نرم افزار های واژه پرداز مانند، wordpad و یا Microsoft word هم نباید استفاده کنید زیرا این نرم افزارها کاراکترها و علامتهایی را به متن شما هنگام ذخیره کردن اضافه می کنند و باعث می شوند کدهای شما غیر قابل اجرا شوند. IDLE پایتون هم انتخاب خوبی است، در محیط پوسته فرمان پایتون به منوی فایل رفته و new window را بزنید، یک صفحه ویرایشگر کد برای شما باز می شود که امکانات اولیه برنامه نویسی همچون syntax highlighting را دارد.

```
who='I '
verb='did '
what='it'
end='.'
print(who, verb, what, end)
print('this is our job')
```

سپس آن را در یک فایل به نام first.py ذخیره کنید، توجه کنید که حتما باید با پسوند py. ذخیره شود. برای ادامه کار آن را باید در مسیر c:\newfiles ذخیره کنید. اگر از نرم افزار notepad معمولی ویندوز استفاده می کنید توجه کنید که از بخش save as type ، در پنجره ذخیره سازی all files را انتخاب کنید. حال از منوی استارت cmd را باز کنید. مسیر آن به صورت :

Start Menu >> Programs >> Accessories >> command prompt

است. دستورات زیر را در آن تایپ کنید:

```
cd c:\newfiles

c:\newfiles>python first.py
I  did  it .
this is our job
```

همانطور که مشاهده می کنید. در این کد ما چند متغیر را با هم چاپ کرده ام و یک استرینگ معمولی را هم در خط بعد چاپ کردیم. این اولین برنامه پایتون بود که شما خودتان نوشتید و اجرا کردید. نوع ساختن و اجرای قالب برنامه های پایتون همین گونه است پس در مثال های بعدی هم می توانید اینگونه از کدهای خود به عنوان برنامه های واقعی استفاده کنید.

توجه: یک خط دیگر هم هست که بدون استثنا باید در تمامی کدهای شما که قرار است در سیستم عامل لینوکس هم اجرا شوند وجود داشته باشد. این خط کد را باید بالاتر از همه دستورات در خط اول بنویسید.

```
#!/usr/bin/python32
```

چند نکته برنامه نویسی:

۱- بهترین روش برنامه نویسی پایتون این است که هر دستور را در یک خط جداگانه بنویسیم ولی می توانیم دستورات پایتون را پشت سر هم نیز بنویسیم، مانند مثال زیر:

```
>>> print('hello ');print('world...');print('bye')
hello
world...
bye
```

به این صورت که باید هر دستور را از دستور بعدی با سِمی کالن (علامت ;) جدا کنیم، بعد از دستور آخر هم نباید سِمی کالن بگذاریم.

۲- هرگز (تکرار می کنم) هرگز از فاصله های بیجا در پایتون استفاده نکنید، چون در پایتون هر فاصله ای کم یا زیاد معنای خاص خودش را دارد که در قسمتهای بعدی با آنها آشنا می شوید.

توابع پایتون را بشناسید

در اینجا قصد داریم از ابتدا به آموزش توابع بپردازیم. توابع از یک نظر به دو دسته کلی تقسیم بندی می گردند:

۱- توابعی که مقدار خاصی را برمی گردانند یا به اصطلاح return می کنند.(در برخی مراجع از این توابع به نام fruitful یا سودمند یاد شده است)

۲- توابعی که چیزی را بر نمی گردانند یا مقدار return ندارند.(یعنی مقدار برگشتی آنها None) است.

توجه: None در پایتون چیزی است که وجود ندارد، مانند متغیری که هنوز مقدار نگرفته یا مقدار تهی دارد. به مثال زیر در پوسته پایتون توجه کنید:

```
>>> d=None
>>> type(d)
<class 'NoneType'>
```

همان طور که می بینید به متغیر d مقدار تهی داده ایم و نوع آن را هم خواسته ایم، و نوع آن به ما NoneType یا از کلاس NoneType اعلام شده است، پس همان طور که می بینید متغیری که تهی هم باشد، یک شئ از یک کلاس محسوب می شود.

یک مثال از توابع نوع اول که مقدار را بر می گردانند می تواند همین تابع type() باشد که به ما نوع پارامتر وارد شده خود را بر می گرداند.

آرگومان چیست؟

آرگومان یا همان argument به مقادیر و متغیرهایی گفته می شود که بین دو پرانتز تابع نوشته می شوند گفته می شود و برای اعمال خاصی که قرار است درون تابع انجام شوند به آن پاس داده می شود. این همان تعریفی است که برای پارامتر یا parameter ارائه می شود، با این تفاوت که پارامتر برای وقتی است که می خواهیم از تابع ساخته شده و کدنویسی شده در برنامه مان استفاده کنیم و یک متغیر یا مقدار را اصطلاحا به آن بفرستیم، در این موارد به چیزی که درون پرانتزهای تابع نوشته می شود پارامتر گفته می شود به این کار یعنی استفاده از تابع قبلا کدنویسی شده فراخوانی تابع گفته می شود. اما آرگومان برای وقتی است که می خواهیم تابعی را بسازیم و کدنویسی کنیم (اصطلاحا گفته می شود تابع را تعریف کنیم). تعریف تابع همیشه قبل از فراخوانی آن صورت می گیرد، چون اگر تابعی نساخته باشیم چطور می توانیم آن را استفاده کنیم. در مثال اخیر d پارامتر تابع ما است نه آرگومان.

تابع خود را بسازید

در تعریف یک تابع از کلمه رزرو شده def و همین طور فاصله گذاری ها استفاده می شود. به مثال زیر دقت کنید:

```
def helpme():
    print('I am here to help you')
    return 'I helped you'
```

ما این مثال را در پوسته فرمان پایتون می نویسیم و Enter را می زنیم، در این مقطع ما یک تابع معمولی را ساخته ایم.

توجه: کلمات رزور شده (reserved word) کلماتی هستند که برای یک زبان برنامه نویسی معنای خاصی دارند و شما نمی توانید از آنها به عنوان اسامی متغیرها یا نام توابع یا هر چیز دیگری به جز کاربردشان استفاده کنید، در اینجا def فقط کاربرد تعریف کردن یک تابع، متد یا ... را دارد و نمی تواند جاهای دیگر مورد استفاده قرار گیرد.

در توضیح مثال بالا باید بگویم که ما باید از def برای تعریف یک تابع استفاده کنیم. پس از آن بلافاصله باید نام تابع و بعد هم پرانتزهای باز و بسته و اگر هم قرار است برای تابع آرگومانی استفاده کنیم آرگومان ها را داخل پرانتز وگرنه نه پرانتزهای خالی(مانند مثال بالا) و سپس هم(علامت :)گذاشته می شوند. سپس به خط بعدی می رویم و به اندازه یک کاراکتر tab یا سه فاصله جلو می رویم و دستور خود را می نویسیم. این عمل را در تمامی خطوط بدنه تابع انجام می دهیم. (اگر از IDLE پایتون استفاده کنید، فاصله ها به طور خودکار ایجاد می شوند).

توجه: شما مجبور نیستید که حتما از سه یا چهار فاصله استفاده کنید، حتی می توانید تنها از یک کاراکتر فاصله یا از ده فاصله استفاده کنید، اما تعداد استاندارد سه (یا به اندازه یک کاراکتر tab است) کاراکتر است. دقت کنید که گرچه در انتخاب اندازه دندانه گزاری آزاد هستید اما حتما باید دندانه گزاری را انجام دهید، زیرا دندانه ها نشان می دهند که تعریف بدنه تابع شما کجا به پایان می رسد و بقیه کدها و دستورات برنامه از کجا آغاز می گردند.

در مثال زیر به آرگومان و نحوه استفاده از آن دقت کنید:

```
>>> def helpme(p):
        print('this is my parameter: ', p)

>>> helpme(4)
this is my parameter:  4
```

در ابتدا تابع را تعریف کرده ایم و برای آن یک آرگومان در نظر گرفته ایم، سپس پارامتری را که به عنوان آرگومان وارد می شود را چاپ کرده ایم. اگر همین دستور را در یک ماژول پایتون (درباره ماژول ها در صفحه بعد توضیح داده شده) هم بکار بریم باز همین نتیجه را پس از اجرای ماژول یا فایل پایتون خواهیم داشت. اما به تعریف زیر دقت کنید.

```
>>> def helpme(p):
        return p

>>> helpme(4)
4
```

در محیط interactive پایتون اگر تابع بالا را با هر پارامتری (هر پارامتری مانند استرینگ، اعشاری یا صحیح) فراخوانی کنیم پارامتر ورودی در خروجی پوسته فرمان پایتون نشان داده خواهد شد. اما در ماژول های (فایلهای) پایتون اینگونه نیست، اگر دقت کنید، ما در بدنه تابع اخیر از دستور تابع print() استفاده نکرده ایم اما چطور مقداری که توسط return برگرداندیم برایمان چاپ شد؟ این ویژگی منحصر به پوسته فرمان پایتون است و در دنیای واقعی خبری از این یکی نیست. یعنی اگر تعریف تابع و فراخوانی تابع اخیر را در یک فایل پایتون ذخیره کنید خواهید دید که چیزی برای شما چاپ نخواهد شد. زیرا مقدار پارامتر تابع فقط بازگردانده شده است. به

صورت خلاصه باید گفت هر چیزی که در دستورات شما باز گردانده شود پوسته فرمان پایتون آن را خود به خود print() هم می کند ولی در ماژولهای پایتون قرار نیست همیچن اتفاقی بیفتد.

توجه: اگر تابعی که در تعریف آن برایش یک یا تعداد بیشتری آرگومان قائل شده و تعیین کرده اید بدون پارامتر فراخوانی کنید خطای TypeError رخ می دهد. اگر تابعی را با تعداد پارامتر بیشتر از آنچه در تعریف آن تعیین شده است هم فراخوانی کنید باز هم این خطا رخ می دهد. به مثال زیر توجه کنید:

```
>>> helpme()
Traceback (most recent call last):
  File "<pyshell#49>", line 1, in <module>
    helpme()
TypeError: helpme() takes exactly 1 argument (0 given)
```

توجه: شما می توانید با تایپ دستور exit() از محیط پوسته فرمان پایتون خارج شوید.

کامنت گذاری

کامنت ها معمولا برای زمانی هستند که snippet (قطعه کد) شما آنقدر پیچیده شده است که نمی توانید بخاطر بسپارید چه اتفاقی در کدام قسمت خواهد افتاد، یا اینکه شک دارید که در آینده آن را فراموش کنید (که حتما هم همینطور خواهد بود)، یا شاید بخواهید که دیگران کد شما را بهتر درک کنند. کاراکتر کامنت گذاری (یادداشت گذاری) #، (بخوانید شارپ یا number sign) است. وقتی اینترپرتر پایتون به این کاراکتر می رسد، خطی که بعد از این کاراکتر قرار دارد را نادیده می گیرد و دستورات آن را اجرا نمی کند. مانند:

```
>>> print('Ali') #prints Ali
Ali
```

اگر می خواهید کامنتهای چند خطی بنویسید مجبور خواهید بود که در ابتدای هر خط # را استفاده کنید.

ماژولهای پایتون

ماژولهای پایتون چیزی نیستند به جز همان فایلهای پایتون که در قسمت های گذشته هم نحوه استفاده از آن ها را مرور کردیم. ماژولهای پایتون چیزهایی مشابه کلاسهای جاوا هستند، البته ماژولهای (module) پایتون یک نوع شئ هستند و دارای خاصیت های خود می باشند. مانند __name__ که یکی از خاصیت های ماژولهای پایتون است.

تالیف: رامان عشقی، ۰۹۳۵۷۷۰۵۰۶۵ برای فیلم های آموزش پایتون به http://www.blue-python.tk مراجعه کنید.

فصل دوم:

ساختارهای معمول پایتون

مرجع آموزش پایتون در ایران

http://www.blue-python.tk

مولف: رامان عشقی

تالیف: رامان عشقی، ۰۹۳۵۷۷۰۵۰۶۵ برای فیلم های آموزش پایتون به http://www.blue-python.tk مراجعه کنید.

باز هم متغیرها

در قسمتهای قبل کمی با متغیرها آشنا شدید، اینجا قصد داریم بیشتر به آنها بپردازیم.

قوانین نام گذاری متغیرها:

۱- نام یک متغیر نباید نام اسامی رزرو شده پایتون مانند: class, def, if, else, lambda, elif, finally باشد.

۲- نام متغیر نباید در بردارنده ی کارکترهایی مانند ٪ $ # @ ! ^ باشد.

۳- نام متغیرها هر چه که باشد حتما باید با یک حرف شروع شود.

اگر نام متغیرها را نابجا و غلط انتخاب کنید SyntaxError رخ خواهد داد.

توجه : نام متغیرها case sensitive هستند یعنی به بزرگ و کوچک بودن حروف حساسند، برای مثال ali با Ali فرق دارد.

در این قسمت اسامی رزور شده توسط پایتون را می بینید، این اسامی نمی توانند به عنوان نام متغیر (identifier) مورد استفاده قرار گیرند.

```
False      class      finally    is         return
None       continue   for        lambda     try
True       def        from       nonlocal   while
```

```
and       del       global    not       with
as        elif      if        or        yield
assert    else      import    pass
break     except    in        raise
```

بودن یا نبود؟

این قسمت به boolean ها می پردازیم. این ها نیز انواعی از داده های پایتون هستند که فقط دو حالت یا بهتر است بگوییم دو مقدار برای آنها قابل دسترسی است که هر کدام فقط یک حالت را می توانند داشته باشند، True یا False.

logical operators (عملگرهای منطقی)

در پایتون سه عملگر منطقی and, or, not داریم که نکته جالب درباره and و or این است که هر دوی اینها اعمال منتطقی ساده ای را که به عهده دارند انجام می دهند و البته مقداری که برمی گردانند Boolean نیست (تنها در صورتی Boolean برمی گردانند که عملوند های آنها هم بولین باشند). در واقع در حالت عادی عملوندی را باز می گردانند (به عنوان return value) که جواب را مشخص می کند. به هر حال این عملگرها برای اعمال عادی منطقی بدون اشکال کار می کنند و لازم نیست گیج شوید.(don't panic)

در زیر مثال هایی را می بینید:

```
>>> '2' and '3'
'3'
>>> '2' and '2'
'2'
>>> '0' and '1'
'1'
>>> a=True
>>> b=False
>>> if a and b:
        print('Python Ocean\n\t')


>>> if a or b:
        print('python ocean\n\t')


python ocean
```

در پایتون تمام اعداد صحیح به جز صفر True به حساب می آیند و فقط صفر است که False می باشد، از همین رو در پایتون ساختارهایی مانند بالا را مشاهده می کنید که در آن به جای Boolean مقدار صحیح بازگردانده می شود.

عبارات کنترل جریان

در پایتون وقتی یک ماژول را اجرا می کنید، تک تک خطهای ماژول به ترتیب پشت سر هم اجرا شده و نتیجه می دهند، اما اگر شما بخواهید یک خط یا یک دستور یا مجموعه ای از دستورات تحت شرایط خاصی اجرا شوند چه؟ در این صورت باید از control flow statements یا همان عبارات کنترل جریان استفاده کنیم. این ساختارها در برخی زبان های دیگر مانند سی پلاس پلاس شامل ساختارهای شرطی با if و case ها می باشند. ما در اینجا هم ساختار شرطی if را به شما معرفی می کنیم.

در ابتدا آگاه باشید که یک ساختار شرطی مانند if فقط یک راه برای اینکه بداند باید خط یا دستور بعد را اجرا کند یا اینکه آن کار خاص را انجام ندهد را دارد و آن هم مفهموم های Boolean یا همان مفاهیم منطقی هستند. به این معنی که if در صورتی دستور بلوک خود را اجرا می کند که عبارت منطقی مربوط به همان if ارزش درستی یا همان True داشته باشد.

یک عبارت منطقی در پایتون به هر چیزی گفته می شود که بتوان از آن برای تولید (بازگرداندن) یکی از دو مفهموم True یا False استفاده کرد. همان طور که در مثال اخیر مشاهده کردید.

به جز اعداد صحیح از رشته ها، کالکشن ها و دنباله ها و انواع داده های شمارشی هم می توان نتیجه ای را گرفت که در ساختار های کنترلی به جای (هم ارزش یا هم ارز با) True , False , None استفاده شوند. برای مثال وقتی از یک رشته خالی استفاده می کنیم به عنوان یک عبارت شرطی (مثال آن را در زیر می بینید) مقداری که بر می گرداند باعث می شود که ساختار if بلوک خود را اجرا نکند و نادیده بگیرد زیرا یک constant خالی به شمار می رود. در این حالت رشته ها یا کالکشن ها و یا ... که خالی نباشند و یک عضو داشته باشند که حداقل اندیس صفر آنها را در اشغال خود داشته باشد، می توانند با True هم ارز باشند.

توجه: در پایتون وقتی یک بلوک جدید ایجاد می کنید برای مثال وقتی یک ساختار شرطی با if می نویسید، اجازه ندارید بلوک را خالی رها کنید، اگر هم می خواهید این کار را برای هدف خاصی انجام دهید باید از pass استفاده کنید. اینگونه که در بلوکی که می خواهید خالی رها شود مثلا یک بلوک متعلق به ساختار if تنها همین دستور را بنویسید. برای مثال به کد زیر دقت کنید:

```
if ' ':
    pass
```

اگردر این حالت از pass استفاده نکنیدوبلوک if را خالی رها کرده وماژول را اجرا کنید،یک IndentationError خواهید گرفت.

ساختار کلی برای if در پایتون به شکل زیر است:

```
if boolean_expression1:
    suite1
elif boolean_expression2:
    suite2
...
else:
    else_suite
```

که در آن Boolean_expression همان عبارت منطقی است که درباره آن گفتیم. و suite ها دستوراتی از پایتون هستند.

در هر ساختار کنترلی پایتون میتواند از صفر تا بینهایت عبارت elif داشته باشد اما بوضوح فقط می تواند یک else داشته باشد. همان طور که گفته شد اگر می خواهیم که در قسمت هیچ اتفاقی نیفتد می توانیم به جای Boolean_expression مخصوص آن قسمت از pass استفاده کنیم.

توجه: حتما باید در نوشتن ساختارهای کنترلی indentation یا دندانه گذاری را برای هر قسمت از دستور استفاده کنید. اگر از یک IDE برای برنامه نویسی استفاده می کنید، خودش این دندانه گذاری ها را در بیشتر موارد انجام خواهد داد.

به یک مثال ساده از این ساختار شرطی توجه نمایید:

```
if a>b:
    print('bigger')
elif a<b:
    print('smaller')
elif a==b:
    print('equal')
else:
    print('!!!')
```

مثال بالا با مقایسه بین مقدار ریخته شده داخل دو متغیر a , b نتیجه را ارزیابی و گزارش می کند.

حلقه (loop)

اکنون که با ساختار های کنترلی آشنا شده اید. وقت آن رسیده که بدانید چطور می توان به پایتون گفت یک قطعه کد را به دفعات خاصی تکرار کند. ما در برنامه نویسی این کار را با حلقه ها انجام می دهیم. منظور ما از دفعات خاص هر تعداد باری است که شما می خواهید، مثلا یک بار یا ۱۰۰۰۰۰ بار یا حتی صفر بار.

اما در این قسمت می خواهیم به حلقه while بپردازیم. ساختار این حلقه مانند زیر است:

```
while boolean_expression:
    suite
```

کار این حلقه این گونه است که با توجه به ارزش درستی عبارت منطقی که در مکان Boolean_expression انجام گرفته کدهایی که در قسمت suite وجود دارند را اجرا کند. یعنی تا جایی که عبارت منطقی ما ارزش True را برگرداند کدهای بلوک while تکرار می شوند. اما چگونه می خواهید به این کار پایان بدهید؟ یعنی چطور

می شود while را متوقف کرد؟ برای انجام این کار بیشک آنقدر راه پیش پا دارید که معمولا انتخاب یکی بین آنها برای شما دردسر درست می کند. یکی از راهها (رایج ترین راه) این است که با یک دستور که در داخل حلقه قرار دارد Boolean_expression را False کنید. برای انجام اینکار می توانید از دو دستور break و continue هم استفاده کنید. دستور break حلقه را به طور کلی متوقف می کند. اما دستور continue اجرا جاری حلقه را متوقف و کنترل برنامه را به اجرای بعدی حلقه می برد، به این معنی که دستورات بعد از continue دیگر اجرا نخواهند شد و حلقه به بار بعدیِ اجرا می پردازد.

در مثال زیر شما یک نمونه استفاده از این حلقه را به همراه ساختار شرطی مشاهده می کنید.

```
while True:
    item = get_next_item()
    if not item:
        break
    process_item(item)
```

در این مثال همان طور که می بینید شرط حلقه یک نوع ثابت Boolean است که True هم هست، در این صورت حلقه احتمالا تا ابد ادامه خواهد داشت اما شما می بینید که در ساختار شرطی if گفته شده که تا زمانی که متغیری به نام item وجود دارد، حلقه تکرار شود و گرنه حلقه را با دستور break به طور کامل متوقف می کند. در اینجا دو تابع فراخوانی شده get_next_item() و process_item() را می بینید، فرض شده است که اینها وجود دارند و قبلا تعریف شده اند.

حلقه for.... in

این ساختار هم یک حلقه است که عملکردی درست مشابه با حلقه while دارد با این فرق که برای اجرای حلقه نیاز دارد که از واژه کلیدی in استفاده کند.به این کلمه کلیدی و کاربردهایش در بخش دیگری پرداخته می شود. در زیر ساختار آن را مشاهده می کنید.

```
for variable in iterable:
    suite
```

همانند حلقه while حلقه for هم هر دو دستور(کلمه کلیدی) break و continue را پشتیبانی می کند و همچنین یک ساختار else اختیاری را هم پشتیبانی می کند. متغیر variable ست شده است (اگر قبلا وجود نداشته ساخته شده است) تا به هر شیئ از iterable به ترتیب ارجاعی باشد.

یک iterable (تکرار شدنی) هر نوع داده ای است که می تواند تکرار شود (اعضایش تکه تکه شوند). و شامل string ها (که در آنها تکرار یا تکه تکه شوی کاراکتر به کاراکتر است)، list ها ، tuple ها (چند تایی ها) و دیگر انواع داده کالکشن پایتون می گردد.

به مثال زیر توجه نمایید:

```
for country in ["Denmark", "Finland", "Norway", "Sweden"]:
    print(country)
```

ما در مثال بالا اعضای یک لیست را دانه به دانه در خروجی چاپ کرده ایم.

در مثالی دیگری که در زیر مشاهده می کنید، ابتدا letter که در هر اجرای حلقه یکی از کاراکترهای درون string را می گیرد و سپس توسط یک ساختار شرطی if تصمیم گرفته می شود که آیا کاراکتر صدادار است یا نه، سپس نتیجه چاپ می شود.

```
for letter in "ABCDEFGHIJKLMNOPQRSTUVWXYZ":
    if letter in "AEIOU":
        print(letter, "is a vowel")
    else:
        print(letter, "is a consonant")
```

مدیریت استثناها (exception handling)

بسیاری از متدها و توابع پایتون error ها یا دیگر اتفاقات مهم را بوسیله به راه انداختن (بالا آوردن) یک exception مشخص می کنند. یک اسپشن هم یک شئ است مانند تمام اشیاء پایتون و وقتی که برای شما در خروجی چاپ می شود و نشان داده می شود در واقع با یک پیام متنی به شما می فهماند چه اتفاقی افتاده است.

ساختار معمول برای یک مدیریت کننده استثنا به نحو زیر است:

```
try:
    try_suite
except exception1 as variable1:
    exception_suite1
...
except exceptionN as variableN:
    exception_suiteN
```

توجه داشته باشید که as variable(n) به صورت اختیاری است یعنی می توانید از آن استفاده کرده یا نکنید. کاربرد قسمت مذکور این است که پیغامی که استثنا ایجاد می کند (و به صورت متنی است) را داخل یک متغیر معمولی مانند variable1 می ریزد. ما اجازه داریم این قسمت را نادیده بگیریم و تنها به استثنایی که برخاسته شده است بپردازیم و به پیغام متنی آن بی اعتنا باشیم به همین جهت این بخش قابل حذف است.

ترکیب کلی این دستور خیلی پیچیده تر از این است که می بینید ، برای مثال هر دستور except چندین استثا را می تواند کنترل کند، و حتی یک دستور else اختیاری هم وجود دارد. به این قابلیت ها در بخشهای بعدی پرداخته خواهد شد.

منطق این دستور اینگونه است که اگر دستورات بلوک try همگی بدون ایجاد و بلند کردن استثنا درست اجرا شوند، در این صورت بلوک های except به کلی نادیده گرفته می شوند و کنترل برنامه (اینترپرتر پایتون) از آنها بدون اجرا کردنشان عبور می کند. اگر یک استثنا در داخل بلوک try ایجاد و برخاسته شود کنترل برنامه سریعا به اولین exception داده می شود که با استثنای برخاسته شده در بلوک try مطابقت داشته باشد. همان طور که شاید حدس زده باشید در داخل بلوک try بعد از ایجاد و برخاستن استثنا دیگر دستورات بعد از دستوری که باعث ایجاد استثنا شده است (یعنی ادامه بلوک try) اجرا نخواهند شد زیرا کنترل بلافاصله به بلوکهای except انتقال می یابد.

اگر یک استثنا اتفاق بیفتد و البته شما بخش اختیاری as variable را رعایت کرده باشید، مسلما در داخل بلوک except ، variable مرجعی از شئ استثنایی است که اتفاق افتاده ، به یاد آورید که در پایتون همه چیز شئ است.

اما اگر در هر کدام از بلوکهای except هم یک استثنا رخ دهد، کنترل به بلوک except بعدی انتقال می یابد که با استثنای برخاسته شده مطابقت دارد. جست و جو برای پیدا کردن مدیریت کننده مناسب برای استثناها به همین صورت ادامه پیدا می کند اگر در آخر هیچ مدیریت کننده استثنای مناسبی یافت نشود برنامه بلافاصله با یک استثنای مدیریت نشده خاتمه می یابد.

در این صورت پایتون برای اینکه به ما بگوید چه اتفاقی افتاده است، یک traceback را به همراه پیام استثنایی که باعث خاتمه برنامه شده است را چاپ می کند. در پایین یک مثال را مشاهده می کنید:

```
s = input("enter an integer: ")
try:
    i = int(s)
    print("valid integer entered:", i)
except ValueError as err:
    print(err)
```

در این مثال اگر کاربر ۳.۵ را وارد کند، خروجی به صورت زیر خواهد بود:

invalid literal for int() with base 10: '3.5'

اما در صورتی که عدد ۱۳ را وارد کند، خروجی زیر چاپ خواهد شد:

valid integer entered: 13

در اینجا تنها به مقدمات مدیریت استثنا پرداختیم، به این علت که دانستن مقدمات آن هم برای شما مفید است تا بدانید که در پایتون چه می گذرد و با ماژول شما چگونه برخورد می کند. به مدیریت استثناها به صورت مفصل در بخشهای بعدی پرداخته خواهد شد.

عملگرهای حسابی (Arithmetic Operators)

پایتون یک مجموعه کامل از عملگرهای حسابی را فراهم می کند، که شامل عمگرهای دودویی هم می شود که چهار عمل اصلی محاسباتی را انجام می دهند. + جمع، - تفریق، * ضرب، / تقسیم، هستند. به علاوه بسیاری از انواع داده های پایتون با عملگرهای افزایشی =* و =+ می توانند کار کنند. عملگرهای - + * / همان طور که انتظار می رود به صورت عادی کار خود را انجام می دهند.

```
>>> 3-2
1
>>> 3+2
5
>>> 3*2
6
```

توجه کنید که – را می توانید هم به عنوان عملگر تفریق و هم به عنوان عملگر منفی کننده استفاده کنید، همان طور که در بسیاری از زبانهای برنامه نویسی هم مشترک است. عملگرهای پایتون وقتی تفاوت خود را با دیگر زبانهای برنامه نویسی نشان می دهند که با / کار می کنید. به مثال های زیر توجه کنید:

```
>>> 12 / 3
4.0
>>> 12 / 2.5
4.8
>>> type( 12 / 2.5 )
<class 'float'>
```

همانطور که می بینید در پایتون / (عملگر تقسیم) جوابهای float تولید می کند، نه یک integer . بسیاری از زبانهای برنامه نویسی integer (اعداد صحیح) تولید می کنند و تمام قسمت اعشاری را از بین می برند. ما برای اینکه مانند این کار را انجام دهیم یعنی از یک تقسیم نتیجه integer به جای float بگیریم می توانیم از تابع int() استفاده کنیم یا اینکه از // استفاده کنیم. این عملگر، عملگر کوتاه کننده تقسیم گفته می شود که در بخشهای بعد درباره آن صحبت می شود.

```
>>> a = 5
>>> a
5
>>> a += 8
>>> a
13
```

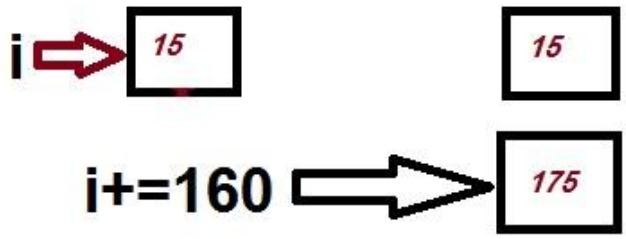
با یک نگاه به مثال بالا متوجه می شوید که مثالها مانند آنچه هستند که در زبان سی مشاهده می شود. در زبانهایی مانند زبان سی مقدار دهی افزایشی (=* و =+) خلاصه ای برای مقدار دهی به یک متغیر خاص توسط پاسخ یک عملیات است. به عنوان نمونه a+=8 همان مقداری را داخل a می ریزد که a=a+8 می ریزد. اما دو نکته بسیار ریز برنامه نویسی در این میان وجود دارند. یکی مخصوص پایتون و دیگری مربوط به مقدار دهی افزایشی در هر زبان برنامه نویسی است.

اولین نکته این است که باید به یاد داشته باشید که انواع داده int ، غیر قابل تغییر یا همان immutable هستند این به این معنی است که تنها یک بار مقدارشان مشخص می شوند، و در این حالت دیگر مقدار int غیر قابل تغییر خواهد ماند. بنابراین چیزی که در پشت پرده اتفاق افتاده است این است که عملگر مقدار دهی افزایشی روی یک

شئ غیر قابل تغییر عمل می کند، که عملیات با آن انجام می شود و سپس یک شئ هم نتیجه عملیات را در خود نگه می دارد. و سر آخر هم شئ ای مانند a که قرار است نتیجه داخل آن ریخته شود، طوری منعکس می شود که مرجعی برای همان object باشد که پاسخ عملیات را دارد.

به این ترتیب در مورد پیشین وقتی اینترپرتر پایتون با عبارت a+=8 مواجه می شود، پایتون a+8 را محاسبه کرده و نتیجه را داخل یک شئ جدید int ذخیره می کند و سپس a را دستکاری می کند به نحوی که این بار a مرجعی از این شئ جدید int باشد نه شئ قبلی int . در اینجا اگر شئ اصلی (مقدار قبلی int که داخل a وجود داشت) دیگر هیچ مرجع شئ (یک مثال از مرجع شئ در اینجا همان a است) ندارد که به آن ارجاع شوند، پس برای garbage collection شدن آماده می شود. احتمالا مفاهیمی که در اینجا به کار رفت کمی شما را سردر گم کرده باشد اما نگران نباشید، به تمام سوالاتی که برای شما پیش آمده پاسخ داده خواهد شد.

نکته ظریف دوم این است که ساختار a operator = b کاملا یکسان با ساختار a = a operator b نیست. زیرا اولین ساختار یا همان عملگرهای مقدار دهی اینترپرتر پایتون مجبور است تنها یک بار مقدار a را در حافظه جستجو کند لذا به صورت بالقوه عملکرد سریعتری را از آن می توان انتظار داشت. همچنین اگر a یک نوع داده با درجه پیچیده ای از فشرده سازی باشد، (برای مثال یک عنصر از یک لیست با یک نوع محاسبات اجباری برای جایگاه اندیس) استفاده از عملگرروش اول باعث می شود تله های کمتری برای ایجاد Error ها وجود داشته باشد، به این معنا که اگر قرار باشد که محاسبات خاصی در هر بار مقدار دهی انجام گیرد، در روش استفاده از عملگرهای افزایشی، این کار فقط یک بار انجام می گیرد نه دو بار و بنابراین احتمال ایجاد خطاها و باگ ها کمتر می شود. به شکل زیر توجه نمایید:



در این شکل شئ غیر قابل تغییری که ۱۵ را نگه می دارد، بعد از انجام عملیات دیگر هیچ مرجع شی ندارد.

در پایتون شما می توانید از عملگرهای + و =+ برای لیست ها و string ها هم استفاده نمایید. در بسیاری از زبان های برنامه نویسی به این ویژگی الحاق یا concatenation گفته می شود، در اینجا ما هم میتوانیم از این واژه استفاده کنیم اما چون در پایتون نام متدها در هر نوع داده متنوع هستند از پیچیده تر کردن آن می پرهیزیم و بهتر

است به سادگی از نام متدها استفاده نماییم. این عمل در string ها (رشته ها)، append و برای لیستها extend خوانده می شود. و برای انجام اینکار از متدهایی با همین نام ها استفاده می شود. به مثال زیر توجه نمایید:

```
>>> first_name='Raman'
>>> last_name='Eshghi'
>>> space=' '
>>> full_name=first_name + space + last_name
>>> print(full_name)
Raman Eshghi
```

درست مانند integer ها، string ها هم immutable یا غیر قابل تغییرند، معنای اینکه غیر قابل تغییر هستند این است مقدار آنها در حافظه دستکاری نمی شود بلکه وقتی قرار است مرجع شئ ای آنها تغییر کند این مقادیر کنار گذاشته شده (اگر دیگر مرجعی نداشته باشند به سوی garbage collection می روند)، و یک مقدار دیگر در حافظه ایجاد و مرجع شئ این بار به شئ ای که آن مقدار را دارد ارجاع داده می شود.

اگر string ها هم مانند integer ها, immutable هستند، همان اتفاقی که برای مقدار string در حافظه می افتد که نمونه ای از آن را پیشتر در مورد integer ها توضیح دادیم. مثال زیر بیان گر این موضوع است:

```
>>> a='foot'
>>> a+='ball'
>>> a
'football'
```

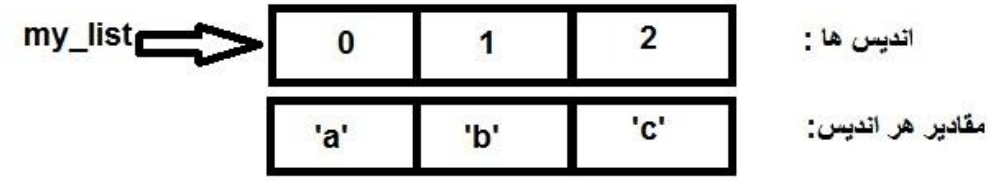
برخلاف آنچه تا بحال دیدیم، لیست ها mutable یا قابل تغییرند، یعنی وقتی عملگر =+ استفاده می شود، شئ اصلی لیست دستکاری و اصلاح می گردد نه اینکه مانند integer ها و string ها شئ اصلی کنار گذاشته شده و شئ جدید برای مقدار جدید ساخته شود. لذا دیگر نیازی نیست که اینترپرتر پایتون دوباره مرجع شئ را به شئ مرتبط سازد. درباره لیست ها در قسمت های بعدی سخن گفته خواهد شد. به مثال زیر توجه نمایید:

```
>>> my_list=['a', 'b']
>>> my_list+='c'
>>> my_list
['a', 'b', 'c']
```

شاید شکل زیر کمی به نظرتان مبهم آید اما درباره لیست در بخش آتی مطالب زیادی گفته شده، فعلا کافی است بدانید که لیست ها قابل تغییرند!

my_list=['a', 'b']

my_list+='c'



شاید این سوال برایتان پیش آید که چون ترکیب زیبای پایتون دیگر شما را از دانستن این که کدام نوع داده قابل تغییر است و کدام غیرقابل تغییر بی نیاز کرده است، چرا باید باز هم به قابل تغییر بودن یا نبودن انواع داده اهمیت داد؟ جواب این سوال در یک چیز است انواع داده غیرقابل تغییر به صورت بالقوه توانایی و فایده های بیشتری را دارند، زیرا یک بار وقتی ساخته می شوند دیگر تغییر نمی کنند. گذشته از این بعضی از انواع داده collection مانند set ها تنها می توانند با انواع غیر قابل تغییر (immutable) کار کنند. از سوی دیگر انواع قابل تغییر معمولا سادگی بیشتری برای استفاده دارند. در آینده باز هم این بحث مطرح خواهد شد.

یک بار دیگر به مثال صفحه ی ۲۴ که برای لیست ها نوشتیم نگاهی بیندازید.

این یک مثال دیگر است، به دقت توجه کنید:

```
>>> my_list+=[4]
>>> my_list
['a', 'b', 'c', 4]
```

و آن را با مثال زیر مقایسه نمایید:

```
>>> my_list+=3
Traceback (most recent call last):
  File "<pyshell#31>", line 1, in <module>
    my_list+=3
TypeError: 'int' object is not iterable
```

اما چرا در شکل بالا دومی مانند شکل اولی نتوانستیم عدد سه را به عنوان یک عضو از لیست به آن اضافه کنیم؟ جواب در یک چیز است، وقتی از عملگرهای حسابی سریع برای لیست ها استفاده می کنیم طرف راست باید iterable باشد.

منظور از iterable تکرار شدنی یا تکرار کردنی است. مانند string ها و خود لیست ها. به همین دلیل است که در مثال صفحه ی ۲۴ توانستیم مقدار رشته ای 'c' را به لیست خود extend کنیم. ولی نتوانستیم مقدار عدد صحیح ۳ را در شکل بالا دومی به my_list اضافه کنیم. زیرا integer ها، iterable نمی باشند.

نکته: به مثال زیر توجه نمایید.

```
>>> a=['a']
>>> a=a+3
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    a=a+3
TypeError: can only concatenate list (not "int") to list
>>> a+=3
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    a+=3
TypeError: 'int' object is not iterable
```

همانطور که در پاراگراف پیشین گفته شد نمی توان به این صورت اعداد صحیح را به لیستها اضافه کرد زیرا iterable نیستند. تنها در صورتی می توانید اینکار را انجام دهید که عملی مانند شکل بالا اولی را انجام دهید. اما سوال اینجاست چرا استثناهای برخواسته شده در دو مثال بالا با هم فرق می کنند؟ این به این دلیل است که وقتی از ساختار a=a+3 استفاده می کنید شما در حال انجام concatenation هستید اما وقتی دارید از ساختار a+=3 (عملگرهای حسابی سریع) استفاده می کنید عمل شما extend است. extend یک متد از همین کلاس list ها است که هر شئ ای که از لیستها ارث بری داشته باشد آن را خواهد داشت. اما توصیه این است که همیشه از ساختاری مانند مثال زیر یا در رتبه بعد از مثال آخر صفحه ی قبل استفاده کنید. سعی کنید تا آنجا که می توانید از پیچیده کردن کدها بپرهیزید و از متدهای در دسترس به جای ساختارهای دوردست و بعید کمک بگیرید:

```
>>> a.extend(3)
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    a.extend(3)
TypeError: 'int' object is not iterable
```

تالیف: رامان عشقی، ۰۹۳۵۷۷۰۵۰۶۵ برای فیلم های آموزش پایتون به http://www.blue-python.tk مراجعه کنید.

ورودی و خروجی (input and output)

برای اینکه قادر باشید برنامه هایی بی همتا بنویسید، باید قادر باشید ورودی را بخوانید، این ورودی می تواند از کیبورد، از فایل یا هر چیز دیگری باشد. منظور از ورودی کیبورد بیشتر ورودی در محیط console یا همان خط فرمان است. همینطور باید بتوانید خروجی را هم تولید کنید چه در همان محیط console و چه در یک فایل. تا به اینجا ما از فرمان print() برای چاپ خورجی در محیط خط فرمان بسیار استفاده کرده ایم.

پایتون تابع built-in (داخلی) input() را برای گرفتن ورودی از کاربر در خط فرمان دارد. این تابع آرگومانهای دلخواهی هم دارد که می توانید وارد کنید، در هنگام گرفتن ورودی متنی که به عنوان آرگومان به آن داده اید برای کاربر چاپ خواهد شد.

تابع input() برای پایان کارش و برداشتن ورودی نیاز دارد که کاربر دکمه Enter(return) را فشار دهد. اگر کاربر چیزی را وارد نکرده و سپس Enter را بزند، تابع input() یک رشته خالی را بر می گرداند. ولی اگر کاربر چیزی را تایپ کرده باشد ورودی کاربر را به صورت یک مقدار (شئ) string برمی گرداند. به مثال های زیر توجه کنید:

```
>>> input('press Enter to continue')
press Enter to continue
''
>>> i=int(input('please enter an integer value\t'))
please enter an integer value	23
>>> print(i)
23
```

در مثال دوم، ما می خواستیم یک مقدار صحیح را از ورودی دریافت کنیم لذا مقدار ورودی را توسط تابع int() به صحیح تبدیل نمودیم.

نکته: رشته های خالی None نیستند، بلکه رشته هستند. اما از لحاظ منطقی ارزش False را دارند.

قطعه کد زیر مثال جالبی از بکار گیری هر آنچه که تا بحال آموخته اید است:

```
print("Type integers, each followed by Enter; or just Enter to finish")
total = 0
count = 0
while True:
    line = input("integer: ")
    if line:
        try:
            number = int(line)
        except ValueError as err:
            print(err)
            continue
        total += number
        count += 1
    else:
        break
if count:
    print("count =", count, "total =", total, "mean =", total / count)
```

به مثال زیر هم توجه کنید:

```
while True:
    try:
        line = input()
        if line:
            try:
                number = int(line)
            except ValueError as err:
                print(err)
                continue
            total += number
            count += 1
    except EOFError:
        break
```

در این قطعه کد، شما ابتدا با یک حلقه به ظاهر –بی پایان برخورد می کنید، سپس در داخل حلقه با یک ساختار مدیریت استثنا مواجه می شوید، این ساختار به شما اجازه می دهد جلوی این بی پایانی را بگیرید، چون همانطور که می بینید در صورت بروز خطای EOFError حلقه بسته می شود. سپس در داخل بلوک try شما می بینید که برنامه از شما ورودی می خواهد. اگر ورودی یک رشته تهی نباشد، یعنی اگر کابر مقادیری را تایپ کند، برنامه به یک مدیریت کننده استثنای دیگر می رسد در این قسمت اگر پایتون توانست مقدار ورودی را به عدد صحیح تبدیل کند،

عدد صحیح را به total اضافه کرده و شمارنده یا همان count را یک واحد اضافه می کند تا متوجه باشد چه تعداد عدد را با هم جمع کرده. اگر هم نتواند به عدد صحیح تبدیل کند(این در صورتی است که کاربر حروف الفبا یا کاراکترهای خاص را هم وارد کرده باشد یعنی هر چیزی به جز ۰۱۲۳۴۵۶۷۸۹)، در آن صورت متن استثنا را در خروجی چاپ کرده و کنترل برنامه را به دور بعدی اجرای حلقه انتقال می دهد.

توابع

در گذشته کمی راجع به توابع گفتیم. حالا وقت آن رسیده است که کمی بیشتر درباره آنها بدانید، شما اکنون تا آنجا با پایتون آشنا شده اید که بتوانید یک تابع واقعی بنویسید. به مثال زیر توجه نمایید:

```
def get_int(msg):
    while True:
        try:
            i = int(input(msg))
            return i
        except ValueError as err:
            print(err)
```

این تعریف یک تابع است که در آن ما یک آرگومان برای تابع تعریف کردیم، این آرگومان msg است. در داخل آن یک حلقه را می بینیم که قرار است از داخل بلوک خود بسته شود. در ساختار مدیریت استثنایی که می بینید اگر کاربر هر چیزی را تایپ کند و Enter را بزند، همان مقدار به عنوان مقدار برگشتی تابع بازگردانده می شود و تابع کارش در همین جا به اتمام می رسد اما اگر کاربر در وارد کردن اطلاعات مشکلی داشته باشد یک ValueError ایجاد کرده که باعث می شود به اجرای بعدی حلقه برود. به این ترتیب تا زمانی که کاربر یک مقدار را تایپ نکند حلقه ادامه پیدا می کند. در ضمن آرگومان msg هم به عنوان پیام در هنگام وارد کردن اطلاعات به کاربرد نشان داده می شود. برای مثال می توانید تابع را به شکل کد زیر فراخوانی کنید:

```
age = get_int("enter your age: ")
```

ماژولها

تالیف: رامان عشقی، ۰۹۳۵۷۷۰۵۰۶۵ برای فیلم های آموزش پایتون به http://www.blue-python.tk مراجعه کنید.

با ماژول ها در قسمتهای قبل آشنا شدید . ماژول های پایتون هم یک نوع شئ هستند . ماژول ها مانند توابع که می توانند برای خود آرگومان ورودی داشته باشند می توانند هنگام اجرا و فراخوانی با آرگومان فراخوانی شوند. برای دسترسی به آرگومان های یک ماژول باید ابتدا یک ماژول دیگر پایتون را وارد کنید:

```
import sys
```

بعد از وارد کردن ماژول بالا می توانید از آرگومان هایی که بِهنگام اجرا به برنامه به آن پاس داده می شوند استفاده کنید، برای مثال به کد زیر دقت کنید:

```
import sys
print(sys.argv)
print('file name is: ',sys.argv[0])
print('arguments are: ', sys.argv[1])
```

sys.argv یک لیست است که آرگومان های برنامه به ترتیب در آن قرار می گیرند، اما باید توجه کنید که آرگومان های برنامه در اندیس های ۱ به بعد قرار خواهند گرفت و اندیس ۰ مخصوص نام کامل ماژول است و نام خود ماژول به همراه مسیر آن در آن ذخیره می شود.

به مثال زیر توجه کنید که از ماژول random استفاده کرده ایم:

```
import random
x = random.randint(1, 6)
y = random.choice(["apple", "banana", "cherry", "durian"])
```

از این ماژول برای اِعمال انتخاب های تصادفی استفاده می شود. برای مثال در خط اول ما از متد randint() برای انتخاب یک عدد صحیح تصادفی بین دو عدد ۱ تا ۶ (که شامل خود این دو عدد هم می شود) استفاده کرده ایم. نحوه استفاده از این متد به همین صورت است که گفته شد. در خط دوم ما از متد choice() استفاده کردیم، این متد هم برای انتخاب تصادفی است اما انتخاب تصادفی بین یک iterable برای مثال یک لیست را دریافت می کند و بین عناصر لیست یک عنصر را به صورت تصادفی انتخاب می کند.

تالیف: رامان عشقی، ۰۹۳۵۷۷۰۵۰۶۵ برای فیلم های آموزش پایتون به http://www.blue-python.tk مراجعه کنید.

فصل سوم:

انواع داده

مرجع آموزش پایتون در ایران

http://www.blue-python.tk

مولف: رامان عشقی

تالیف: رامان عشقی، ۰۹۳۵۷۷۰۵۰۶۵ برای فیلم های آموزش پایتون به http://www.blue-python.tk مراجعه کنید.

قرار دادها

در ابتدای فصل اول شما با قرارداد های ساخت شناسه ها آشنا شدید، همچنین درباره کلمات کلیدی پایتون بحث کردیم.

پایتون یک تابع به نام dir() دارد که می تواند attribute ها یا همان خاصیت های یک شئ را برای شما نشان دهد. این تابع می تواند تمام خاصیت ها شامل متدها و متغیرهای کلاس و ... به شما نشان دهد. اما به یاد آورید که همه چیز در پایتون یک شئ است، لذا شما می توانید attribute های تمام اشیایی که نیاز دارید را با همین تابع به راحتی بیابید، به مثال های زیر دقت کنید:

```
>>> import sys
>>> dir(sys.argv)
['__add__', '__class__', '__contains__', '__delattr__', '_
', '__getitem__', '__gt__', '__hash__', '__iadd__', '__imu
ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr_
'__str__', '__subclasshook__', 'append', 'count', 'extend
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'sy
>>> dir('')
['__add__', '__class__', '__contains__', '__delattr__', '_
', '__getnewargs__', '__gt__', '__hash__', '__init__', '__
new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmo
, 'capitalize', 'center', 'count', 'encode', 'endswith',
'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnume
er', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind
s', 'startswith', 'strip', 'swapcase', 'title', 'translate
```

همانطور که می دانید در خط اول ماژول sys را وارد کردیم ، در خط دوم attribute های شئ sys.argv را خواسته ایم که برایمان چاپ کرده است. در نمونه سوم هم می بینید که تابع dir() بدون آرگومان فراخوانی شده است،در این صورت attribute های buit-in پایتون را برمی گرداند.دردستور چهارم می بینیدکه مattribute های یک رشته خالی را گرفته ایم، در این صورت هیچ فرقی نمی کند رشته خالی باشد یا حاوی کاراکتر باشد attribute ها یکسان هستند.

اگر __builtins__ را به عنوان آرگومان این تابع وارد کنیم، تمام attribute های داخلی پایتون را بر می گرداند. __builtins__ یک ماژول پایتون است که حاولی تمامی attribute های پایتون می شود.

```
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseExcep
or', 'Ellipsis', 'EnvironmentError', 'Exception', 'False', 'Floati
Error', 'ImportWarning', 'IndentationError', 'IndexError', 'KeyErr
, 'None', 'NotImplemented', 'NotImplementedError', 'OSError', 'Ove
ceWarning', 'RuntimeError', 'RuntimeWarning', 'StopIteration', 'Sy
', 'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError',
icodeWarning', 'UserWarning', 'ValueError', 'Warning', 'WindowsErr
oc__', '__import__', '__name__', '__package__', 'abs', 'all', 'any
'classmethod', 'compile', 'complex', 'copyright', 'credits', 'dela
'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'h
'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'map', '
'pow', 'print', 'property', 'quit', 'range', 'repr', 'reversed', '
'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

انواع صحیح

پیشتر از این درباره اعداد صحیح گفته شده است، می خواهیم در اینجا بیشتر به این انواع داده پرداخته و متدها و attribute های آنها را مورد بررسی قرار دهیم، در جدول زیر متدهایی که روی اعداد صحیح عادی عمل می کنند را مشاهده می کنید:

کاربرد	ترکیب
x و y را جمع می کند	x + y
عمل تفریق را انجام می دهد	x - y
عمل ضرب عادی را انجام می دهد	x * y
عمل تقسیم را انجام می دهد و پاسخ را به صورت اعشاری بر می گرداند	x / y
عمل تقسیم را انجام می دهد و پاسخ را به صورت صحیح بر می گرداند	x // y
باقی مانده تقسیم را بر می گرداند	x % y
x را به توان y می رساند	x ** y
x را منفی می کند	-x
کار خاصی انجام نمی دهد جز نشان دادن اینکه x مثبت است	+x
قدر مطلق x را برمی گرداند	abs(x)
خارج قسمت و باقی مانده تقسیم x بر y را به صورت یک tuple از دو عدد صحیح باز می گرداند	divmod(x, y)
همان کاری را انجام می دهد که x ** y می کند	pow(x, y)
یک نوع راه سریع برای بدست آورد عبارت x ** y % z است	pow(x, y, z)

در جدول پایین هم چند متد دیگر را می بینید که برای کار با اعداد صحیح استفاده می شوند:

کاربرد	ترکیب
شکل باینری عدد صحیح i را بر می گرداند	bin(i)
شکل هگزادسیمال عدد صحیح آرگومان را برمی گرداند	hex(i)
شکل صحیح آرگومان خود را برمی گرداند، برای مثال از string و float به integer تبدیل خواهد کرد	int(x)
شکل اکتال عدد صحیح وارد شده را بر می گرداند	oct(i)

integer literal ها یا همان انواع داده صحیح، همانطور که می دانید از مبنای ده دهی استفاده می کنند. اما شما می توانید این اعداد صحیح را به هر مبنایی که دوست دارید برده و از آنها استفاده کنید. اگر تجربه برنامه نویسی داشته باشید می دانید که انجام اینکار بدون متدهایی که در جدول شماره چهار شرح داده شده اند هم میسر است و

می توان با الگوریتم های خاصی این کار را انجام داد. اما یک نکته مهم را هیچ وقت در برنامه نویسی فراموش نکنید، هرگز به دنبال ساختن چیزی که از قبل وجود داشته نروید. به جز برای یادگیری. در زیر مثالهایی را مشاهده می کنید:

```
>>> 453463467 #decimal
453463467
>>> 0b1010110001   #مبناي دو
689
>>> 0o34334723   # اُكتال  همان  مبناي هشت يا
7453139
>>> 0xDAE759042   # هگزادسيمال
58761515074
```

در دستور اول شما یک عدد صحیح معمولی (با مبنای ده) را تایپ کردید، همان طور که می بینید عددی که برگردانده است تبدیل شده ی عدد به حالت مبنای ده است . دستورات دوم ، سوم و چهارم هم به ترتیب اعدادی صحیح با مبنا های دودویی ، اُکتال و هگزا دسیمال را در اینترپرتر نوشته ایم همانگونه که می بینید ، مقدار برگشتی حالت ده دهی اعداد وارد شده است،فرقی نمی کند درچه مبنایی باشند،مبنای ده آنهادر شِل پایتون به شما باز می گردد.

نکته: برای نوشتن اعداد باینری یا دو دویی در ابتدای عدد باید عبارت 0b ، برای نوشتن اعداد مبنای هشت یا اُکتال در ابتدای عدد باید عبارت 0o (صفر+اُ انگلیسی) و برای هگزا دسیمال باید در ابتدا عبارت 0x را بنویسیم. در تمام این سه حالت می توان از حروف بزرگ به جای حروف کوچک استفاده نمود.

توجه داشته باشید که از توابع و عملگرهایی که در جدول ۳ معرفی شده اند روی اعداد صحیح اثر می گذارند و اهمیتی ندارند که عدد صحیح چه مبنایی دارد. به مثال زیر توجه کنید:

```
>>> 0o4353 / 0o1
2283.0
>>> type( 0o4353 / 0o1)
<class 'float'>
```

این توابع و عملگرها کارشان را انجام داده و در آخر پاسخ را به صورت اعداد صحیح یا اعشاری با مبنای ده باز می گردانند.

تمامی عملگرهایی که در جدول شکل ۳ مشاهده نمودید، دارای یک نسخه از عملگرهای سریع محاسباتی هستند، مانند: =٪ و =– و =// و =/ و =** ، درباره این عملگرها و چگونگی کاربرد آنها صحبت شده.

اشیا می توانند با نسبت دادن مقادیر داده صحیح یا هر نوع دیگری به متغیر ها ساخته شوند. برای مثال بوسیله ی x=17 یا با فراخوانی نوع داده مورد نظر به عنوان تابعش مانند: x=int(17) . برخی از اشیا (مانند آنهایی که از نوع

decimal.Decimal هستند) تنها می توانند با استفاده از نوع داده شان ایجاد شوند، زیرا برای آنها هیچ نوع نمایش لفظی وجود ندارد. وقتی یک شئ تنها با استفاده از نوع داده خود آفریده می شود، سه مورد دیگر برای استفاده وجود دارد.

اولین مورد این است که نوع داده دلخواه ما بدون هیچ آرگومانی فراخوانی شود، در این صورت یک شئ با مقدار پیش فرض آفریده خواهد شد. برای مثال x=int() یک شئ صحیح به مقدار ۰ را خواهد آفرید. تمامی انواع داده های داخلی پایتون می توانند بدون آرگومان فراخوانی شوند مانند آنچه در سطر بالا مشاهده کردید.

دومین مورد این است که نوع داده ما با یک آرگومان فراخوانی شود. اگر آرگومان ما از همان نوعی باشد که تابعی که برای ایجاد شئ داده است (مانند int(23) که ۲۳ از همان نوع صحیح یا int است)، در این صورت یک کپی سطحی (shallow copy) از شئ اصلی (آرگومان وارد شده) آفریده خواهد شد. با کپی سطحی در آینده آشنا خواهید شد. اما اگر آرگومان داده شده به تابع نوع ما از نوع متفاوتی باشد (برای مثال int(23.4) که آرگومان ما float ولی تابع ما برای ساخت شئ int است) در آن صورت یک تلاش برای یک تبدیل اتفاق خواهد افتاد. در این حالت اگر تابع ما بتواند تبدیل نوع یا conversion را انجام دهد که شئ جدید ساخته می شود و در هنگام تبدیل نوع مشکلی پیش آید یک استثنای ValueError را ایجاد می کند. اگر بتواند شئ را با موفقیت بسازد. شئ را باز می گرداند. اگر هم نوع قابل تبدیل نباشد در آن صورت یک استثنای TypeError ایجاد خواهد شد. توجه داشته باشید که تمامی float ها قابل تبدیل به صحیح و str هایی که در خود تنها digit یا یکی از ۰۱۲۳۴۵۶۷۸۹ را داشته باشند قابل تبدیل به صحیح هستند.

سومین مورد این است که دو یا حتی تعداد بیشتری آرگومان به تابع داده شوند. البته تمای انواع داده این مورد و حالت را پشتیبانی نمی کنند. و حتی آن انواع داده ای که تابعشان این گونه فراخوانی را پشتیبانی می کند، انواع داده و معانی آنها متفاوت هستند. مثلاً برای تابع int() دو آرگومان اجازه داده می شود که در این صورت اولی می تواند یک str باشد که حاوی عدد صحیح است و آرگومان دوم مبنای عددیِ عدد صحیحی است که در رشته اول موجود است. برای مثال int("A4", 16) یک شئ صحیح به مقدار ۱۶۴ را می آفریند.

عملگرهای بیتی

عملگرهای بیتی عملگرهایی هستند که بر روی بیتها اثرگذارند. معمولا در پایتون خیلی کم پیش می آید که با آنها سرو کار داشته باشید.

عملگرهای بیتی در جدول شماره ۸ نشان داده شده اند. از پایتون ۳.۱ به بعد یک متد int.bit_length() قابل دسترسی و استفاده است. این تابع تعداد بیتهایی را که برای نمایش عدد صحیح نیاز است را بر می گرداند. به مثال زیر توجه کنید:

```
>>> (1391).bit_length()
11
```

ترکیب	کاربرد
i | j	OR بیتی از دو عدد i و j را بر می گرداند
i ^ j	XOR از دو عدد i و j را بر می گرداند
i & j	AND بیتی از دو عدد i و j را بر می گرداند
i << j	i به سمت چپ شیفت می کند(حرکت می دهد) به اندازه j بیت
i >> j	i به سمت راست شیفت می کند(حرکت می دهد) به اندازه j بیت
~i	بیت های i را برعکس می کند

Booleans

در پایتون دو نوع شئ داخلی (built-in) بولین وجود دارند: True و False . مانند انواع داده های داخلی دیگر پایتون، نوع داده ئ bool هم می تواند به عنوان یک تابع بدون آرگومان یا با آرگومان فرخوانی شود. بدون آرگومان False برمی گرداند. با یک آرگومان از نوع bool ، کپی آرگومان را باز می گرداند، و با هر نوع آرگومان دیگری سعی می کند که یک مقدار bool یا همان یکی از True یا False را تولید کند. تمامی انواع داده اصلی و استاندارد کتابخانه ئ پایتون این قابلیت را دارند که یک پاسخ bool را پدید آورند. به مثال زیر توجه کنید:

```
>>> type(False)
<class 'bool'>
>>> bool()
False
>>> bool(3)
True
>>> bool(-1)
True
>>> bool(0)
False
```

همچنین در مثال زیر می بینید که چگونه از عملگرهای منطقی استفاده می شود.

```
>>> t = True
>>> f = False
>>> t and f
False
>>> t and True
True
```

انواع داده ممیز شناور (floating point)

پایتون سه نوع داده ممیز شناور در خود دارد: یکی float که پیشتر با آن آشنا شده ایم، دیگری complex و آخری هم decimal.Decimal است. هر سه این سه نوع داده immutable یا غیر قابل تغییرند. نوع داده float اعداد اعشاریِ دقت مضاعف را در خود نگه می دارد که ظرفیت این اعداد اعشاری به کامپایلر c یا C# یا جاوایی بستگی دارد که اینترپرتر پایتون با آن کامپایل شده است. آنها دقت های محدودی دارند و نمی توان گفت که در تمام آنها به صورت قابل اطمینانی دقت یکسان خواهد بود. اعداد اعشاری به دو صورت نوشته می شوند یکی با استفاده از نقطه ی ده دهی، و دیگری با استفاده از نمادگذاری تشریحی ریاضی، برای مثال: ۲.۳ و ۲e-۴ هر دو اعداد اعشاری هستند.

تمامی توابع و عملگرهایی که در جدول شکل ۱۱ مشاهده می کنید برای هر نوع داده (عدد) از نوع float قابل استفاده هستند. نوع داده float هم این قابلیت را دارد که به صورت یک تابع فراخوانی شودمانند float(). اگر بدون هیچ آرگومان فراخوانی شود ۰.۰ را برمی گرداند ، با یک آرگومان از نوع اعشاری کپی آن آرگومان را بر می گرداند. با هر گونه آرگومان دیگری تلاش می کند تا آرگومان وارد شده را به float تبدیل کند. اگر بخواهد از انواع دیگر به float تبدیل کند بهترین گزینه بی شک str ها هستند، منظور در اینجا رشته هایی مانند '14.3' و '6e-2' است.

به هر حال خیلی از چیزها به نوع کامپایلری که نرم افزار اینترپرتر پایتون را با استفاده از آن برای سیستم شما کامپایل کرده اند بستگی دارد. در پایین یک جدول کوچک از تنها تعدادی از توابعی که برای انجام اعمال معمولی ریاضی می توانند مورد استفاده قرار گیرند موجود است. برای استفاده از این توابع و متدها ابتدا باید ماژول math را وارد (import) کنید.

توجه: اگر می خواهید بیشتر در این باره بدانید به منابع پایتون مراجعه کنید، در این کتاب به دلیل اینکه هدف آن آموزش کاربردی برنامه نویسی است کمتر به مباحث کاربرد ریاضیات پرداخته شده است.

ترکیب	کاربرد
`math.acos(x)`	arc cos آرگومان ورودی خود را بر می گرداند
`math.acosh(x)`	برای محاسبه arc cos هایپربولیک آرگومان خود استفاده می شود.
`math.log10(x)`	لگاریتم عدد x را در پایه ۱۰ محاسبه می کند
`math.pi`	مقدار تقریبی عدد پی را بر می گرداند
`math.sqrt(x)`	جذر x را بر می گرداند
`math.tan(x)`	تانژانت x را بر می گرداند

نکته: وقتی در یک ماژول پایتون از رشته هایی با محتویات غیر استاندارد استفاده می کنید، برای مثال رشته ای مانند: 'سلام دنیا' یک رشته است که محتویات آن تابع ASCII نباشند و شامل کاراکترهایی مانند کاراکترهای غیر انگلیسی و از یک زبان دیگر باشند مانند زبان فارسی، در این صورت می توانید با اضافه کردن قسمت زیر به ابتدای برنامه خود کاری کنید به راحتی رشته هایی حاوی کلمات و جملات فارسی را هم به خوبی جملات انگلیسی بخواند.

```
# -*- coding: utf-8 -*-
```

توجه داشته باشید که این قطعه از کد را تنها باید در بالاترین قسمت (اول هر ماژول) بنویسید، درست قبل از نوشتن هر دستور یا چیز دیگری.

رشته ها

رشته ها در پایتون توسط نوع داده غیر قابل تغییر str که شامل یک سری از کاراکترهای Unicode است ساخته می شوند. نوع داده str مانند انواع داده دیگر می تواند به صورت یک تابع فراخوانی گردد تا اشیای str (رشته ای) را بسازد. در این صورت بدون هیچ نوع آرگومانی این تابع یک رشته خالی ('') و با یک آرگومان به جز نوع str شکل رشته ایِ آرگومان را باز خواهد گرداند. و با یک آرگومان از نوع رشته ای یک کپی از آرگومان خود را باز می گرداند. تابع str() همچنین می تواند به عنوان یک تابع برای تبدیل (conversion) استفاده شود. در این صورت آرگومان اول این تابع باید از نوع رشته ای باشد یا اینکه از نوعی قابل تغییر به نوع رشته ای باشد، تا دو آرگومان رشته ایِ دیگر هم می تواند داشته باشد که یکی (دومین آرگومان) encoding که برای رشته استفاده می شود را مشخص می کند و دیگری هم مشخص می کند که با خطاها و استثناهایی که درباره encoding رشته اتفاق می افتد چه باید کرد.

تالیف: رامان عشقی، ۰۹۳۵۷۷۰۵۰۶۵ برای فیلم های آموزش پایتون به http://www.blue-python.tk مراجعه کنید.

در قسمتهای پیشین کتاب درباره ایجاد انواع داده ایه رشته ای گفته ایم که برای ساختن آنها آزاد هستید که از علامت نقل قول دوتایی یا تک نقل قول استفاده نمایید، اما در پایتون می توانید حتی با علامتهای نقل قول سه تایی هم رشته بسازید، نام زبان اصلی این نوع رشته ها triple quoted string است. فرقی نمی کند که شما از علامت نقل قول ' یا از علامت " استفاده کنید، مانند زیر:

```
>>> text='''I am a\n triple quoted string'''
>>> text="""me too, \n I am a triple quoted string also"""
```

در جدول آتی شما کدهای فرار (scape) در رشته های پایتون را مشاهده می کنید:

تالیف: رامان عشقی، ۰۹۳۵۷۷۰۵۰۶۵ برای فیلم های آموزش پایتون به http://www.blue-python.tk مراجعه کنید.

کد فرار	معنا
\newline	نادیده گرفتن خط جدید
\\	برای گذاشتن \ در نوشته بکار می رود
\'	برای گذاشتن علامت نقل قول تکی در نوشته بکار می رود
\"	برای گذاشتن علامت نقل قول دوتایی بکار می رود
\a	بل در سیستم اسکی
\b	بک اسپیس در سیستم اسکی
\f	فرم فید در اسکی
\n	کاراکتر پایان خط در اسکی
\N{name}	کاراکتر یونیکد با نام داده شده داخل آکلاد
\ooo	کاراکتر با کد اکتال داده شده
\r	کارکتر برگشت در اسکی
\t	کاراکتر تب در اسکی

به مثال زیر توجه کنید:

```
>>> print('\' I am in single quotes \'')
' I am in single quotes '
```

ما در این مثال از کاراکتر escape (فرار) برای گذاشتن یک علامت نقل قول تکی در داخل رشته استفاده کرده ایم، این به این خاطر است که اگر به صورت عادی کاراکتر ' را در رشته بگذاریم، در آن صورت، اینترپرتر پایتون آن را با پایان رشته اشتباه می گیرد و یک استثنای SyntaxError ایجاد می شود. مانند مثال زیر:

```
>>> print(' 'I am in single quetes' ')
SyntaxError: invalid syntax
```

برای دانستن کد Unicode هر کاراکتر می توانید از تابع ord() استفاده نمایید. می توانید مانند مثال زیر عمل کنید:

```
>>> ord('d')
100
>>> ord('ي')
1610
```

همان طور که می بینید کدی که برای کاراکتر 'd' برگردانده شده ۱۰۰ و آن که برای کاراکتر 'ی' برگردانده شده است، ۱۶۱۰ است، اگر به صفحه کلید خود نگاهی بیندازید می بینید که هر دوی این حروف یک دکمه از صفحه کلید شما را دارند پس چرا کد کاراکتری آنها متمایز است؟ ساده است چون کاراکتر انگلیسی 'd' از سیستم ASCII

که سیستم استاندارد کاراکترهای کامپیوتری است پیروی می کند، اما کاراکتر 'ی' که یک کاراکتر فارسی است از Unicode پیروی می کند. ascii یک سیستم متشکل از ۲۵۶ کاراکتر است که این کاراکتر ها از ۰ تا ۲۵۵ شماره گذاری شده اند، همان طور که می بینید شماره ای که به کاراکتر 'd' اختصاص داده شده است، عدد ۱۰۰ است. این ۲۵۶ کاراکتر شامل حروف انگلیسی بزرگ و کوچک و کاراکترهای ویژه ای همچون & ^ $ #@` ~ ' ; : و همچنین کاراکترهای فضای خالی و کاراکترهای خط جدید و ... هستند. اما Unicode سیستمی دیگر و جدا از ascii است که برای شماره گذاری و مشخص کردن کاراکترهای غیر استاندارد (غیر انگلیسی) استفاده می شود استفاده از این سیستم کاراکتری به صورت کلی باعث می شود که حجم بیشتری از داده ها تشکیل شوند، به خصوص زمانی که شما از متون با حجم بالایی از کارکترهای Unicode استفاده می کنید حجم برنامه یا فایل متنی یا هر چیز دیگری که در آن از سیستم Unicode استفاده می شود بالا رود. این اتفاق به این دلیل است که کامپیوتر اطلاعات را به صورت بایتی می بیند نه به صورتی مانند str حتی وقتی درحال بارگذاری یک صفحه وب هستید هم همین موضوع صادق است یعنی شما دارید مقادیر بایت را دریافت می کنید. bytes هم یک نوع از داده ها در پایتون است. بایت ها در واقع لایه زیرین هر چیزی هستند که شما در حال مشاهده آن هستید، به مثال زیر توجه کنید:

```
>>> 'd'.encode()
b'd'
>>> 'ي'.encode()
b'\xd9\x8a'
```

```
>>> type(b'd')
<class 'bytes'>
```

متدی که ما در این جا از آن استفاده کردیم، متد ()encode است. این متد سعی می کند که شئ کاراکتر یا رشته را به صورت بایت هایش برای شما نمایش دهد. علامت b قبل از 'd' به این معناست که این یک مقدارbytes است نه یک رشته زیرا رشته ها هم با ' ' احاطه شده اند، این برای این است که اینترپرتر و خود شما آنها را اشتباه نگیرید. در دستور دوم همان طور که می بینید مقادیر بایتی کاراکتر 'ی' که از سیستم Unicode استفاده می کند چقدر بیشتر و حجیم تر از کاکتر 'd' است که از سیستم ascii بهره می برد، توجه کنید که متد encode دو آرگومان ورودی دارد که اولی باید encoding را مشخص کند و دیگری باید نحوه برخورد با خطاهایی که در encode کردن اتفادق می افتد. اگر می بینید که هیچ کدام از آرگومان های این متد مقدار دهی نشده اند این به این خاطر است که هر دوی این آرگومان های پیش بینی شده، از نوع keyword argument هستند و همچنین دارای مقداردهی پیشین (یعنی در حین تعریف تابع می باشند). این خود یکی از ویژگی های جالب پایتون است که در آینده به آن خواهیم پرداخت.

بهتر است قبل از اینکه جلوتر برویم با چند تابع دیگر نیز آشنا شوید:

تابع ascii() : این تابع سعی می کند که هر مقداری که از آرگومانش باز گردانده می شود را به صورت یک str در سیستم ascii در آورده و بازگرداند.

تابع chr() : این تابع یک عدد را گرفته و کارکتر معادل آن را در سیستم کدگذاری Unicode باز خواهد گرداند.

مقایسه در str ها

انواع رشته ای در پایتون از عملگرهای عادی مانند: <, <=, !=, >= به صورت کامل پشتیبانی می کنند. این عملگرها انواع رشته ای را در حافظه به صورت بایت به بایت مورد مقایسه و بررسی قرار می دهند. متاسفانه معمولا دو مشکل اساسی به هنگام مقایسه انواع رشته ای به وجود می آید مانند هنگامی که لیستهایی شامل انواع رشته ای را مرتب می کنید . هر دوی این مشکلات مربوط به تمام زبان های برنامه نویسی که سیستم Unicode را پشتیبانی می کنند است و پایتون هم از این قاعده مستثنی نیست.

اولین مسئله این است که برخی از کاراکترهای Unicode این قابلیت را دارند که با استفاده از دو یا حتی چند شکل بایتی خود معرفی شوند. برای مثال کاراکتر 'Å' (که مثال آن را در زیر می بینید،) با 0x00C5 در سیستم Unicode،

```
>>> chr(0x00C5)
'Å'
```

می تواند در سیستم اِنکدینگِ utf-8 به سه شکل مختلف نشان داده شود که در زیر می بینید:

```
[0xE2, 0x84, 0xAB]
[0xC3, 0x85]
[0x41, 0xCC, 0x8A]
```

خوش بختانه می توانیم این مشکل را حل نماییم. اگر ما ماژول UnicodeData را در برنامه خود وارد کنیم، و تابع unicodedata.normalize() را به همراه 'NFKC' به عنوان اولین آرگومان و دویمن آرگومان یک مقدار رشته ای که شامل کاراکتر مورد نظر ما باشد (کاراکتری که مشکل ایجاد می کند برای مثال در اینجا کاراکترِ 'Å') فراخوانی کنیم، این تابع یک مقدار رشته ای به ما بر خواهد گرداند، که این مقدار رشته ای دارای یک مقدار bytes استاندارد شده است که دیگر مشکلاتی که ذکر شد را ایجاد نخواهد کرد.

مشکل دوم این است که مرتب سازی بسیاری از کاراکترها در مقادیر رشته ای در هر زبان خاص همان زبان است، برای مثال در زبان سوئدی ä بعد از z می آید، اما در آلمانی این گونه نیست. یا برای مثال در انگلیسی کاراکتر Ø به گونه ای مرتب می شود که انگار همان کارکتر o (اُ انگلیسی) است اما در زبان نروژی این گونه نمی باشد. همان گونه که می دانید گاهی هم اتفاق می افتد که مقادیر رشته ای ترکیبی از زبان های مختلف هستند! این یعنی مشکل روی مشکل.

پایتون در این گونه موارد برای اینکه خطاها و اشکلات ریز اینچنینی را به بار نیاورد مسئله را پیچیده تر نمی کند. پایتون در هنگام مقایسه مقادیر رشته ای مقادیر بایتی همان رشته را که در حافظه (RAM) قرار دارند را مقایسه می کند. این یک نوع ترتیب بندی شدن بر اساس موقعیت کدهای رشته در سیستم Unicode است. در این صورت وقتی بخواهد رشته ای که حاوی کاراکترهای زبان انگلیسی است را مرتب کند آنها را بر اساس ascii مرتب می کند.

تکه تکه سازی str ها

ما می دانیم که آیتم ها (item)ی تکی (تکه ها) در یک دنباله یا هر چیزی که به صورت یک دنباله بیاید، مانند کاراکترهای داخل یک رشته که می توانند به وسیله عملگر دسترسی به آیتم (item access operator) یا همان [] بیرون کشیده شوند. شما با این عملگر (عملگر دسترسی به آیتم ها) آشنایی ندارید، پس بدانید این عملگر کارش بیرون کشیدن است، منظور از بیرون کشیدن در این جا بدست آوردن و دسترسی است نه حذف کردن. به مثال زیر توجه کنید که چگونه با استفاده از این عملگر یک تکه از رشته خود را بیرون کشیده ایم:

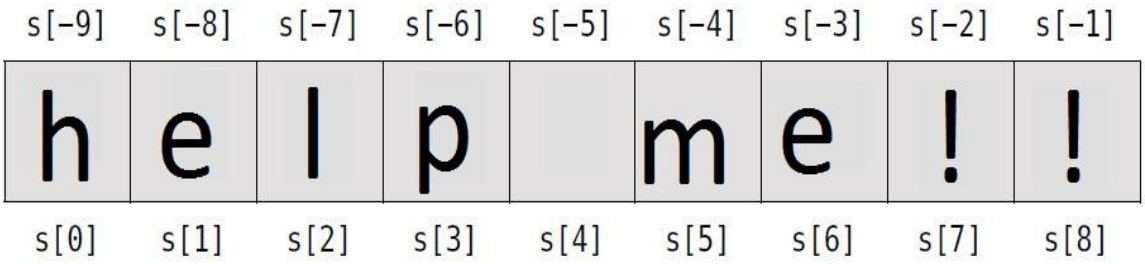
```
>>> a='help me!'
>>> print(a[6])
e
```

در مثال بالا ما شئ ای که در مکانی با اندیس۶ در متغیر رشته ایِ a بوده را درخواست نمودیم وآن هم به ما برگردانده شده است. در واقع این عملگر بسیار فراگیر است نه تنها برای دسترسی به یک آیتم یا کاراکتر می توانید از آن استفاده کنید بلکه می توانید به تکه هایی با اندازه بیش از یک آیتم یا کاراکتر در هر قسمتی که باشند دسترسی داشته باشید، به خاطر همین به این عملگر، عملگرِ تکه سازی هم گفته می شود زیرا به جز تَک آیتم ها و تَک کاراکترها می تواند به تِکه هایی شامل چندین آیتم یا کاراکتر دسترسی پیدا کند.

در ابتدا ما کارمان را با بیرون کشیدن تَک کاراکترها (individual characters) آغاز می کنیم. اندیس ها در یک رشته و یا یک لیست یا هر چیز دیگری در پایتون از صفر شروع شده و بالا می روند تا تمام اشیای داخل متغیر

(مرجع شئ) دارای اندیس شوند. اما به جز این روش اندیس گذاری شما می توانید برای دسترسی به یک آیتم یا چند آیتم خاص از روش اندیس گذاری منفی هم استفاده کنید این روش به گونه ای است که از آخرین کاراکتر شروع به شمارش می کند و به آخرین کاراکتر یا آیتم اندیسی برابر با ۱- (منفیِ یک) داده می شود و به همین صورت به آیتم یکی مانده به آخر اندیس ۲- (منفیِ دو) داده شده تا به اولین آیتم برسد. باز هم مشاهده می کنید که در هر دو روش اولین آیتم کوچکترین اندیس صحیح را می گیرد (در روش اندیس گذاریِ منفی اولین آیتم کوچکترین عدد منفی را دریافت می کند). در مثال زیر به اندیس گذاری هر کدام از کاراکترها یا آیتمهای داخل متغیر رشته ایِ s توجه نمایید.

```
>>> s='help me!!'
>>> len(s)
9
```



برعکس آن چیزی که شما ممکن است فکر کنید اندیس گذاری منفی بسیار هم مفید است، مخصوصا اندیس منفیِ یک (۱-) که بدون توجه به طول متغیر آخرین آیتم را برمی گرداند، خودتان فکر کنید اگر همچین چیزی نبود شما باید یک خط کد اضافی می نوشتید! شاید با خود اندیشیده باشید که اگر یک اندیس را درخواست کنیم که وجود ندارد چه؟ مثلا در شکل بالا اگر اندیس ۹ یعنی s[9] را بخواهیم چه اتفاقی می افتد؟ در این صورت یک IndexError اتفاق می افتد. به مثالهای زیر که درباره شکل بالا هستند توجه نمایید:

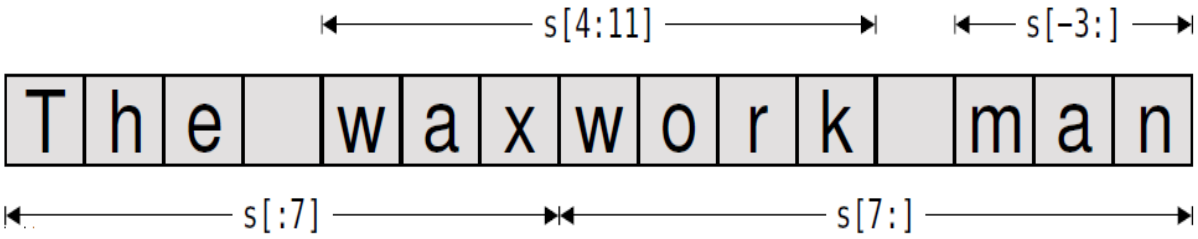
```
>>> s[3]
'p'
>>> s[0]
'h'
>>> s[4]
' '
>>> s[-1]
'!'
>>> s[-4]
'm'
```

عملگر تکه تکه سازی یا [] را به سه ترکیب زیر می توانید استفاده کنید:

```
seq[start]
seq[start:end]
seq[start:end:step]
```

که seq در اینجا هر sequence یا دنباله ای می تواند باشد، شامل لیستها، رشته ها، tuple ها یا چند تایی ها و.... . step, start, end هر سه باید مقادیر int باشند (یا که متغیرهایی از نوع int باشند). ما از ترکیب اوّلمان در مثال های قبل استفاده کرده ایم. این ترکیب (ترکیب اول) آیتمی که دارای اندیس شماره start است را از sequence بیرون بکشد. ترکیب دوم اینگونه عمل می کند که از اندیسی به شماره start شروع کرده و تا اندیس شماره end هر چه مابین این ها وجود دارد را بیرون کشیده و باز می گرداند، توجه کنید که آیتمی که دارای اندیس start است هم بیرون کشیده می شود اما آیتمی که دارای اندیس end است جزو تکه ای که بیرون کشیده می شود نمی باشد. ترکیب سوم را هم به زودی تشریح خواهیم کرد.

اگر ما ترکیب دوم را استفاده کنیم (که دارای یک علامت colon است)، در این صورت ما می توانیم اندیس start را ننویسیم در این صورت شکل دستور به صورت s[:3] خواهد بود. اگر این عمل را انجام دهیم اندیس شروع یا start مقدار پیش فرض صفر را خواهد گرفت. حتی می توانیم end را هم ننویسیم، در این صورت اندیس end مقدار len(seq) را می گیرد، برای مثال s[:] تمام رشته را بر خواهد گرداند. به مثال زیر توجه نمایید:




```
>>> s='The waxwork man'
>>> s[3]
' '
>>> s[4:4]
''
>>> s[4:5]
'w'
>>> s[:5]
'The w'
>>> s[5:]
'axwork man'
>>> s[6:-3]
'xwork '
```

یکی از راههایی که می توان یک قطعه داده str را داخل یک رشته ی دیگر قرار داد، با استفاده از ترکیب الحاق و تکه تکه سازی است:

```
>>> s = s[:12] + "wo" + s[12:]
>>> s
'The waxwork woman'
```

البته ما می توانستیم همین کار را با دستور زیر نیز انجام دهیم:

```
s[:12] + s[7:9] + s[12:]
```

البته در آینده خواهید دید که استفاده از هیچ کدام از این روش ها وقتی که تعداد زیادی داده str و تعداد زیادی دستور از این نوع برای پردازش وجود دارند (مثلا در یک حلقه) مناسب نیست. بلکه بهترین راه استفاده از متد str.join() است.

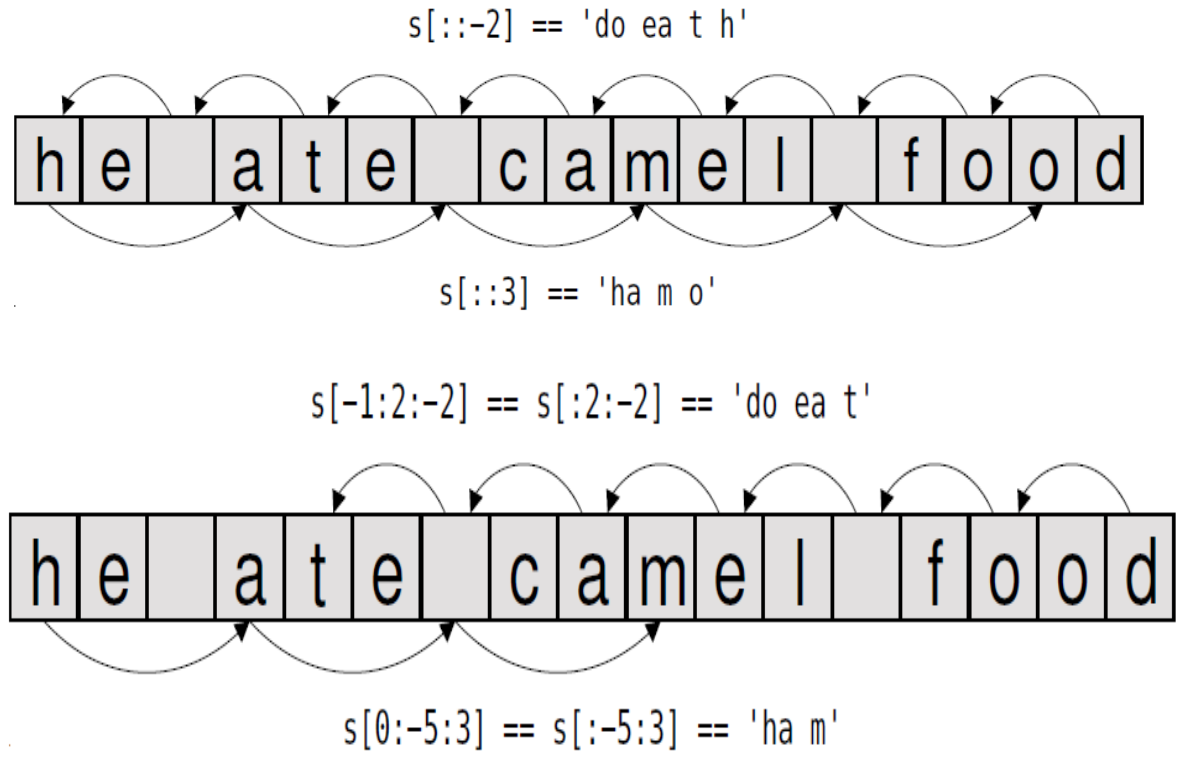
سومین ترکیبی که مشاهده کردید کارش شبیه به ترکیب دوم است با این تفاوت که به جای بیرون کشیدن تک تک کاراکترها (آیتم ها) با گام هایی به اندازه ی step پیش رفته، و آیتم ها را بیرون بکشیم. برای مثال اگر مقدار step را عدد ۲ بگذاریم، هر دو کاراکتر به دو کاراکتر آیتم بیرون کشیده می شود. مانند ترکیب دوم که استفاده کردیم، در این ترکیب هم می توان هر کدام از اندیس های start, end, step را خالی گذاشت. اگر ما اندیس شروع را حذف کنیم مقدار پیش فرض صفر را خواهد گرفت مگر اینکه step را منفی نوشته باشیم. در این صورت start مقدار پیش فرضِ ۱- (منفیِ یک) را خواهد گرفت. اگر ما اندیس end را نادیده بگیریم و ننویسیم در این صورت مقدار پیش فرض آن len(seq) خواهد شد، مگر اینکه برای اندیس step مقداری منفی بنویسیم در این صورت end مقدار پیش فرض اندیسی پیش از شروع رشته را خواهد گرفت. اگر هم ما step را ننویسیم مقدار پیش فرض ۱ را می گیرد. توجه کنید که نمی توانید که به جای step عدد صفر را بنویسید. اگر این کار را انجام دهید یک ValueError اتفاق می افتد. به مثال صفحه ی بعد توجه کنید:

```
>>> s[2:8:0]
Traceback (most recent call last):
  File "<pyshell#21>", line 1, in <module>
    s[2:8:0]
ValueError: slice step cannot be zero
```

به مثال های زیر توجه نمایید:

```
>>> s='he ate camel food'
>>> s[::3]
'ha m o'
>>> s[::-2]
'do ea t h'
>>> s[-1:2:-2]
'do ea t'
>>> s[:2:-2]
'do ea t'
>>> s[0:-5:3]
'ha m'
>>> s[:-5:3]
'ha m'
>>> s[:-5:3]
'ha m'
```

به اشکال زیر دقت کنید که در مثالهای بالا را تشریح می کنند:



متدها و عملگرهای str ها

به دلیل اینکه str ها دنباله هایی غیرقابل تغییر(immutable)هستند،تمام کاربردهایی که با انواع داده های غیر قابل تغییر می توان استفاده کرد را می توان برای رشته ها هم بکار برد. این کاربردها شامل تست کردن عضو بودن با

استفاده از in ، الحاق با + ، append کردن با =+ ، ضرب (چند برابر سازی) با *، همچنین =* می باشند. ما تمامی این کاربردها و عملگرها را در این بخش به زودی تشریح خواهیم کرد. به علاوه بسیاری از متدهایی را که می توان با رشته ها استفاده کرد را هم تشریح می کنیم.

رشته ها اشیای دنباله ای هستند لذا هر کدام از آنها طول خاصی را دارند، بنابراین می توانیم تابع len() را با یک آرگومان از نوع رشته ای فراخوانی کنیم. مقداری که از این تابع برگردانده خواهد شد، تعداد کاراکترهای موجود در رشته ای هستند که به عنوان آرگومان به تابع فرستاده شده. (مقدار صفر برای یک رشته خالی بازگردانده می شود.) در گذشته گفتیم که برای الحاق (concatenate) کردن رشته ها به هم در مواردی که دستورات الحاق زیادی باید انجام شوند بهتر است به جای + از str.join() استفاده کنیم. این متد یک دنباله (sequence) را به عنوان آرگومان خود دریافت می کند ، برای مثال یک لیست یا یک tuple که حاوی آیتم هایی از نوع str باشد را می گیرد و رشته های داخل دنباله های ذکر شده را داخل یک رشته به هم می چسباند. نکته این است که این متد رشته های موجود در آرگومان خود را به وسیله ئ همان رشته ای به هم می چسباند که روی آن فراخوانی شده است. به این معنی که بین هر دو تا از رشته های داخل آرگومان خود یک رشته را قرار خواهد داد، این رشته همان رشته ای است که این متد از روی آن فراخوانی شده است. به مثال های زیر توجه کنید:

```
>>> p=['help', 'me', 'please']
>>> ' '.join(p)
'help me please'
>>> '/'.join(p)
'help/me/please'
>>> '**'.join(p)
'help**me**please'
```

در دومین دستور ما آیتم های موجود در لیست p را که همه از نوع رشته ای بوده اند را به وسیله ی یک کاراکتر فضای خالی در کنار هم قرار داده ایم . شما می توانید آیتم های داخل لیست را (اگر str باشند) به بدون هیچ فاصله ای به هم بچسبانید، برای این کار مانند دستور زیر، اولین رشته را باید رشته ئ تهی بگذارید:

```
>>> ''.join(p)
'helpmeplease'
```

متد str.join() می تواند به همراه تابع built-in (داخلی) reversed() برای برعکس کردن رشته ی مان مورد استفاده قرار گیرد. توجه داشته باشید که آرگومان ورودی تابع reversed() می تواند هر نوع دنباله ای باشد مثل لیست ها، چند تایی ها(tuple) و یا رشته ها. همان جور که در مثال زیر می بینید، کاری را که با تابع داخلیِ reversed() انجام می دهیم را می توان با p[::-1] هم انجام داد.

```
>>> ''.join(reversed(p))
'pleasemehelp'
>>> p[::-1]
['please', 'me', 'help']
>>> ''.join(p[::-1])
'pleasemehelp'
```

اگر در قسمتی از برنامه نیاز دارید که بدانید آیا یک آیتم خاص در داخل یک sequence یا دنباله قرار دارد یا نه چه باید بکنید؟ برای مثال اگر شما بخواهید بدانید که آیا کاراکترِ '؟' داخل یک رشته خاص وجود دارد یا نه چه می کنید؟ در این صورت یک راه استفاده از عملگر عضویت (membership operator) یعنی in است. اگر این عملگر را بر روی یک رشته بکار ببریم، True باز می گرداند اگر که رشته ی رشته یا کاراکتر سمت چپ آن مساوی با بخشی از رشته ی سمت راستش باشد. رشته سمت راست آرگومان راست و رشته سمت چپ در پایتون آرگومان چپ نامیده می شوند.

```
>>> 'a' in 'about'
True
>>> 'k' in 'about'
False
>>> 'about' in 'about
True
>>> '' in 'about'
True
>>> '' in ''
True
```

در صورتی که نیاز داشته باشیم مکان دقیق (اندیس) یک رشته داخل یک رشته دیگر را بفهمیم، ما دو متد داریم که می توانیم ازهر کدام استفاده نماییم.یکی str.index() است این متد اندیس مکان رشته ی آرگومان را داخل رشته ی سمت چپ (رشته ی مادر که در سمت چپ نوشته می شود و متد روی آن فراخوانی می گردد) پیدا می کند و بر می گرداند، اگر که در حین انجام این کار مشکلی به وجود آید یا اینکه رشته ی آرگومان اصلاً داخل رشته ی مادر وجود نداشته باشد یک ValueError ایجاد می گردد. متد دیگری که شما می توانید از آن استفاده کنید، str.find() است. این متد کارایی بهتری دارد به صورتی که اندیس مکانی را که رشته ی آرگومان خود را در رشته مادر (رشته سمت چپ که همان رشته ای است که متد روی آن فراخوانی شده است) پیدا کرده است را بر می گرداند، چنانچه رشته ی آرگومان در رشته ی مادر موجود نباشد، مقدار ۱- را بر خواهد گرداند. البته هر دوی این متدها دو آرگومان دیگر را هم به عنوان آرگومان های دوم و سوم خود می گیرند.که دومین آرگومان اندیس مکان شروع جست وجوی رشته در رشته ی مادر و سومین آرگومان اندیس مکان پایان جست و جو در رشته ی مادر هستند. کاربرد این متدها را در مثال های زیر می بینید:

تالیف: رامان عشقی، ۰۹۳۵۷۷۰۵۰۶۵ برای فیلم های آموزش پایتون به http://www.blue-python.tk مراجعه کنید.

```
>>> 'about'.find('k')
-1
>>> 'about'.index('k')
Traceback (most recent call last):
  File "<pyshell#19>", line 1, in <module>
    'about'.index('k')
ValueError: substring not found
>>> 'about'.index('b')
1
>>> 'about'.find('b')
1
>>> 'about'.find('u')
3
```

در صفحه بعد جدول بلندی از متدهای str ها را مشاهده می کنید. مثال های متدهای جدول صفحات بعد را در زیر مشاهده می کنید:

```
>>> 'a  b'.expandtabs(1)
'a b'

>>> 'I am in \n newline'.isprintable()
False
>>> 'I am in current line'.isprintable()
True

>>> 'R4m4n Eshghi'.replace('4', 'a', 2)
'Raman Eshghi'

'Raman Eshghi'
>>> ' Raman Eshghi '.strip()
'Raman Eshghi'
>>> '^# Raman Eshghi %$*'.strip('&^%$#@* ')
'Raman Eshghi'
```

```
>>> 'raman eshghi'.capitalize()
'Raman eshghi'
>>> 'raman eshghi'.upper()
'RAMAN ESHGHI'
```

```
>>> 'raman eshghi'.title()
'Raman Eshghi'
```

ترکیب	کاربرد
s.capitalize()	یک کپی از رشته ئ s را بر می گرداند البته اولین حرف آن را بزرگ می کند
s.replace(t, u, n)	یک کپی از رشته s را بازمیگرداند که در آن تمام رشته های t با رشته های u عوض شده اند، n از نوع صحیح است و مشخص می کند چه ماکزیمم تعداد t با u در رشته عوض شوند.
s.count(t, start, end)	تعداد اتفاق افتادن (پیدا شدنِ) رشته ئ t را در رشته ئ s از اندیس start تا اندیس end برمی گرداند.
s.encode(encoding, err)	درباره این متد در گذشته گفته ایم، این متد شکل بایتی رشته ئ s را با سیستم encoding بر می گرداند،
s.endswith(x, start, end)	این متد چک می کند که آیا رشته ئ s با زیر رشته ئ x به پایان می رسد یا نه. اگر آری True وگرنه False را بر می گرداند، start, end هم بازه ای در s برای بررسی هستند
s.expandtabs(size)	تمام کاراکترهای tab داخل رشته ئ s را با space یا فضاهای خالی پر می کند. تعداد فضاهای خالی که به جای هر tab قرار می گیرند، همان مقداری است که size مشخص می کند.
s.find(t, start, end)	درباره این متد هم به تفضیل توضیح داده شده است
s.format(...)	این متد برای formatting به کار می رود، این متد را در بخش های بعدی به طور کامل تشریح می کنیم.
s.index(t, start, end)	درباره این متد هم به تفضیل توضیح داده شده است
s.isalnum()	این متد True برمی گرداند اگر رشته s خالی نبوده و تنها شامل کاراکترهای الفبایی و digit ها باشد
s.isalpha()	این متد True بر می گرداند اگر رشته s خالی نبود و تنها شامل کاراکترهای حروف الفبا باشد.
s.isprintable()	True بر می گرداند هنگامی که S تهی باشد یا اینکه تمام کاراکترهایی که در s وجود دارند با تابع print() نمایش داده شوند مانند فضای خالی نه مثل کاراکتر خط جدید (\n)
s.isdigit()	این متد True برمی گرداند اگر رشته s خالی نبوده و فقط شامل کاراکترهای عددی یا digit در سیستم ascii باشد.
s.isidentifier()	True برمی گرداند اگر رشته s خالی نباشد و بتوان از آن به عنوان یک شناسه درست استفاده کرد
s.islower()	True برمی گرداند اگر حداقل یک کاراکتر قابل کوچک شدن (منظور حروف بزرگ و کوچک انگلیسی است) داشته باشد، و بتوان این حروف را به حرف کوچک تبدیل کرد.

s.split(t, n)	یک لیست را باز می گرداند، به صورتی که رشته ئ s را قطعه قطعه کرده و تمام تکه هایی که بوسیله ئ کاراکتر فضای خالی از هم جدا شده اند را در یکی از اندیس های لیست جدید می ریزد. t باید یک رشته باشد. اگر t را بنویسیم به جای فضای خالی تکه های رشته در اطراف رشته ئ t تشکیل می شوند.
s.splitlines(f)	تمامی خطوط موجود در یک رشته را که با کاراکتر \n (خط جدید) مشخص شده اند را قسمت کرده و در لیستی بر می گرداند اگر آرگومانش True باشد، \n را هم در لیست جدید به کار می برد
s.startswith(x, start,	کاربرد آن درست برعکس تابع s.endswith() است که گفته شد.
s.swapcase()	تمام کاراکترهای بزرگ را به کاراکترهای کوچک و کوچکها را به بزرگ تبدیل می کند
s.title()	اولین حرف تمام کلمات یک رشته را بزرگ می کند و رشته را برمی گرداند
s.upper()	تمام کاراکترهای داخل رشته ئ s را به حروف بزرگ تبدیل می کند
s.strip(chars)	یک کپی از رشته s را بر می گرداند که تمام رشته های char از ابتدا و انتهای رشته ئ s حذف شده اند. اگر این متد بدون آرگومان فراخوانی گردد، تنها کاراکترهای فضای خالی از ابتداو انتهای s حذف می شوند.
s.lower()	تمام کاراکترهای داخل رشته ئ s را به حروف کوچک تبدیل کرده و بر می گرداند

```
>>> 'Raman Eshghi has written this book,'.split()
['Raman', 'Eshghi', 'has', 'written', 'this', 'book,']
>>> 'Raman Eshghi has written this book,'.split('a')
['R', 'm', 'n Eshghi h', 's written this book,']
>>> 'Raman Eshghi has written this book,'.split('t')
['Raman Eshghi has wri', '', 'en ', 'his book,']
```

```
>>> 'Raman Eshghi \n has written this book'.splitlines()
['Raman Eshghi ', ' has written this book']
>>> 'Raman Eshghi \n has written this book'.splitlines(True)
['Raman Eshghi \n', ' has written this book']
```

```
>>> 'PYTHON'.lower()
'python'
>>> 'PYTHON'.lower().title()
'Python'
```

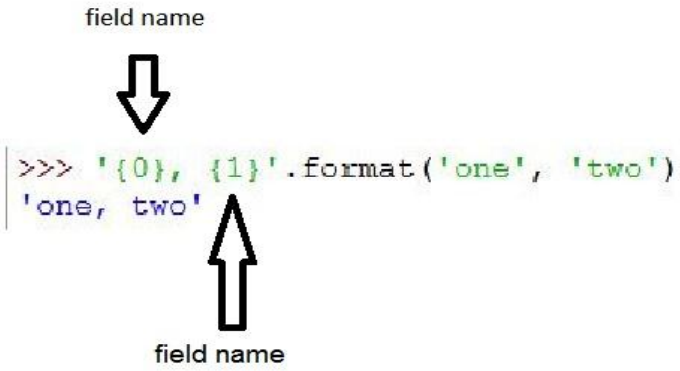
قالب بندی رشته ها با استفاده از متد str.format()

متد str.fomat() یک راه بسیار انعطاف پذیر و قدرتمند را برای ایجاد انواع str فراهم کرده است. استفاده از متد str.format() برای موارد خیلی ساده و معمولی بسیار ساده است. اما برای موارد پیچیده تر باید ترکیب مورد استفاده در این دستور را بیاموزید.

این متد یک رشته ی جدید را با فیلدهای جایگذاری شده باز خواهد گرداند. به صورتی که آرگومان های وارد شده را به صورت درستی در داخل متنی که از روی آن فراخوانی شده است قرار می دهد. به مثال ساده ی زیر توجه کنید، این آسانترین کاربرد این متد است:

```
>>> 'This is a {0} for learning programming in {1} language'.format('book', 'python')
'This is a book for learning programming in python language'
```

هر فیلد جایگذاری با یک نام فیلد در داخل علامت های {} مشخص می شود. اگر نام فیلد (field name) یک عدد (int) ساده باشد، به عنوان اندیس مکان یکی از آرگومان هایی که به متد str.format() پاس داده می شوند بکار گرفته می شود. پس در مورد مثال بالا فیلدی که نامش ۰ است، بوسیله ی اولین آرگومان جایگذاری می شود. و آن فیلدی که نامش ۱ است بوسیله ی آرگومان دوم جایگذاری می شود. به شکل زیر توجه نمایید:



اگر ما بخواهیم که در داخل متنمان با همین روش خودِ علامت های {} را جایگذاری کنیم چه؟ در این صورت باید به جای هر کدام از علامت های آکولاد که قرار است در رشته ی نهایی (رشته ای که از متد برگردانده می شود) برگردانده شود، دو علامت {} بگذاریم. به مثال زیر توجه کنید:

```
>>> "{{{0}}} {1} ;-}}".format("I'm in braces", "I'm not")
"{I'm in braces} I'm not ;-}"
```

یکی از کاربردهای جالب این متد این است که می توان با استفاده از آن بدون کدنویسی اضافی انواع int و str را در هم الحاق کرد، به مثال زیر توجه کنید که چگونه یک رشته را می توان در کنار یک عدد قرار داد.

```
>>> 'number is '+1
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    'number is '+1
TypeError: Can't convert 'int' object to str implicitly
>>> '{0} {1}'.format('number is', 1)
'number is 1'
```

فیلد جایگذاری هر کدام از چهار ترکیب زیر را می تواند داشته باشد.

```
{field_name}
{field_name!conversion}
{field_name:format_specification}
{field_name!conversion:format_specification}
```

نام فیلد (field name)

یک نام فیلد هم می تواند یک عدد صحیح باشد که در این صورت باید با اندیس یکی از آرگومان های متد str.format() همخوانی داشته باشد یا اینکه با یکی از آرگومان های keywork هم خوانی داشته باشد. در مثال زیر به نام فیلدها توجه نمایید:

```
>>> "{who} turned {age} this year".format(who="She", age=88)
'She turned 88 this year'
>>> "The {who} was {0} last week".format(12, who="boy")
'The boy was 12 last week'
```

نکته: keyword arguments یا آرگومان های کیوُرد آرگومان هایی هستند که در زمان فراخوانی شدن می توان برای فراخوانی آنها از یک متغیر مانند variable_name= استفاده کرد. اما آرگومان های مکانی یا positional آرگومان هایی هستند که تا به حال برای توابعی که فراخوانی کرده ایم استفاده کرده ایم، برای مثال در فراخوانیِ str('help us') ، رشته ی 'help us' آرگومانِ مکانی است.

اولین مثال دو آرگومانِ keyword استفاده می کند، who و age .دومین مثال از آرگومان های مکانی (positional) استفاده کرده است و آرگومان دوم آن هم keyword argument است. باید به یک قاعده ی بسیار مهم در استفاده از این دو نوع آرگومان با هم بسیار توجه کنید، این است که همیشه (تکرار می کنیم) همیشه، آرگومان های مکانی بر آرگومان های کیورد مقدّم هستند، به این معنی که اگر قرار است در فراخوانیِ یک تابع از هر دو نوعِ این

آرگومان ها استفاده کنیم، باید ابتدا از آرگومان های مکانی (positional) استفاده کنیم و بعد آرگومان های کیورد را بیاوریم. زیرا آرگومان های مکانی از اندیس هر کدام از آرگومان ها استفاده می کنند و نباید بین آنها اندیسی را اشغال کرد.

نام فیلد ها حتی می توانند به عنوان مرجعی برای انواع داده ی collection مانند لیست ها استفاده شوند. در اینچنین حالت هایی ما می توانیم برای مشخص کردن یک آیتم خاص در داخل آن لیست یا نوع داده ی collection از یک اندیس خاص آن آیتم استفاده کنیم. توجه کنید که در این صورت باید اندیس یک آیتم را استفاده کنیم نه اینکه از یک تکه (که حاویِ چندین آیتم است) در یک آرگومانِ کیورد استفاده کنیم. به مثال زیر توجه کنید:

```
>>> stock = ["paper", "envelopes", "notepads", "pens", "paper clips"]
>>> "We have {0[1]} and {0[2]} in stock".format(stock)
'We have envelopes and notepads in stock'
```

در این مثال ۰ یک نوع آرگومان مکانی (positional) است، بنابراین {۰[۱]} دومین آیتمِ لیستِ stock است و {۰[۲]} سومین آیتمِ لیستِ stock است.

نکته: از پایتون 3.0 به بعد این امکان برای شما وجود دارد که نام فیلد ها را ننویسید، در این صورت به ترتیب به هر کدام از آنها یک اندیس داده می شود (از صفر به بعد). مثال:

```
>>> '{} {} {}.'.format('I', 'am', 'programmer')
'I am programmer.'
```

فراخوانی ها (calls)

یک قطعه کدِ فراخوانی، برای فراخواندنِ یک شئ که اینچنین قابلیتی را داشته باشد مورد استفاده قرار می گیرد، حتی به صورت خالی و بدون آرگومان.در پایتون تمام اشیایی که دارای یک متد()__call__هستندقابل فراخوانی اند. اگر در فراخوانیِ یک تابع یا یک شئِ قابل فراخوانی یا یک متد از اشیای داخلی و ... قرار باشد آرگومان هایی دریافت شوند، ابتدا تمامیِ عباراتی که مربوط به آرگومان ها می باشند قبل از اینکه این فراخوانی انجام گیرد مورد بررسی قرار خواهند گرفت. در فراخوانی ها اگر آرگومان های keyword وجود داشته باشند، ابتدا آنها به آرگومان های positional (مکانی) تبدیل می شوند. به این ترتیب یک لیستی از جایگاههای خالی برای پارامترهای تابع ایجاد می شود. اگر به اندازه ی n تا آرگومان مکانی (positional) وارد شده اند، ابتدا آنها به همان ترتیبی که وارد شده اند داخل n جایگاه موجود قرار می گیرند. در گام بعدی برای هر keyword argument ، شناسه برای

تصمیم گرفتن اینکه کدام جایگاه مناسب تر است استفاده می شود. این کار تا آنجا ادامه پیدا می کند که آرگومانها به پایان برسند، اگر تعداد اسلات ها (جایگاهها) به پایان برسد اما هنوز تعدادی از پارامترها جایگذاری نشده باقی بمانند، در این صورت یک استثنای TypeError ایجاد خواهد شد. توجه نمایید که در این صورت حتی اگر مقدار None هم به عنوان یکی از پارامترها وارد شود، قابل قبول است و آن هم یکی از اسلات ها را برای خود می گیرد. حالا اگر پارامترها تماماً مورد استفاده قرار گرفته و درون اسلات ها جایگذاری شوند، اما هنوز یک سری از اسلات ها خالی باقی مانده باشند بوسیله ی مقادیر پیش فرضی که در تعریف بدنه تابع برایشان در نظر گرفته شده است پر می شوند، برای مثال یک keyword argument که مقدار پیش فرض دارد، اگر مقداری برای آن وارد نشود مقدار پیشفرض آن استفاده می شود. بنابراین یک شئ mutable مانند یک لیست یا یک دیکشنری به عنوان داده های پیشفرض در میان تمامیِ تماس هایی که یک پارامتر خاص را برای آرگومانی خاص ندارند، به اشتراک گذاشته خواهد شد. این روش کدنویسی باید سعی شود مورد استفاده قرار نگیرد. اگر هم حالتی پیش آید که در یک فراخوانی تعدادی اسلات وجود داشته باشند که برخی از آنها پر نشده اند و در عین حال آرگومان های آن مکان ها دارای مقدار پیشفرض نمی باشند یک استثنای TypeError ایجاد خواهد شد.

همانطور که می دانید اگر تعداد پارامترهای positional زیاد از حد باشد، یک TypeError ایجاد می شود، مگر اینکه در وارد کردن پارامتر ها از ترکیبی مانند *identifier استفاده کنیم، که در این صورت پارامتر رسمیه تابع یک تاپل که شامل آرگومان های positional اضافی می شود آفریده خواهد شد (اگر آرگومان های مکانیِ اضافی وجود نداشته باشند، در این صورت یک تاپلِ خالی به تابع فرستاده خواهد شد.

حالا چه اتفاقی می افتد اگر هیچکدارم از آرگومان های keyword با هیچکدام از نام های پارامترهای تابع همخوانی نداشته باشند؟ در این صورت هم یک استثنای TypeError رخ خواهد داد. مگر اینکه یک پارامتر رسمی با استفاده از ترکیب **identifier موجود باشد. در این صورت است که آن پارامتر رسمی، یک دیکشنری که حاویِ آرگومان های keyword اضافی است ایجاد می شود. به این صورت که کیورد های آرگومان ها به عنوان key و مقادیر آرگومان ها به عنوان مقادیر متناظر دیکشنری ها مورد استفاده قرار خواهند گرفت.

اگر ترکیب *expression در فراخوانیِ تابع وجود داشته باشد، expression باید یک iterable (تکرار شدنی) باشد. با هر کدام از عناصر داخل این iterable طوری رفتار می شود که انگار آرگومانهای مکانی (positional argument) پشت سر هم هستند . برای مثال اگر آرگومان های مکانیِ x1, x2, ..., xn در فراخوانی وجود دارند و از ترکیب *expression هم استفاده می کنیم که expression دنباله ای از عناصر y1, y2, ..., ym باشد، این فراخوانی در واقع یک فراخوانی با تعداد m+n آرگومان مکانی (positional argument) است. و آرگومان های آن به صورت x1, x2, ..., xn, y1, y2,..., yn خواهند بود.

پیآمد مسئله ای که درپاراگراف پیشین موردبررسی قرارگرفت (استفاده ازexpression*)این است که حتی اگر شما ازترکیبexpression*بعدازkeyword argumentها استفاده کنید،این ترکیب پیش از keyword argument ها مورد بررسی و پردازش قرار می گیرد (و البته پیش از ترکیب expression**) . به مثال زیر توجه کنید، که این مسئله را روشن می کند:

```
>>> def my_function(a, b):
        print('first argument is: ', a)
        print('second argument is: ', b)


>>> my_function(b=1, *(2,))
first argument is:  2
second argument is:  1
>>> my_function(a=2, *(1,))
Traceback (most recent call last):
  File "<pyshell#24>", line 1, in <module>
    my_function(a=2, *(1,))
TypeError: my_function() got multiple values for keyword argument 'a'
```

```
>>> my_function(1, *(3,))
first argument is:  1
second argument is:  3
```

در اولین فراخوانی، آرگومان b به صورت ابتدا به صورت keyword argument فراخوانی شده اما همان طور که گفته شد چون ترکیب (2,)* در این جا موجود است، با وجود اینکه حتی پس از keyword argument آمده است اما اول آن مورد پردازش قرار می گیرد، لذا اول (2,)* پردازش می شود، این یعنی اینکه اولین آیتم داخل این تاپل یعنی ۲ به a داده می شود ، سپس که آیتم دیگر داخل تاپل نیستند به سراغ آرگومان های بعدی تابع رفته می شود و در این مرحله b=1 عمل می کند و مقدار b را یک قرار می دهد. اما در فراخوانی دوم چرا دچار مشکل شده ایم؟ به این دلیل است که در اینجا ابتدا (1,)* پردازش می شود و مقدار آرگومان a برابر با 1 قرار می گیرد و آرگومان بعدی که پردازش می شود هم a=1 ولی نمی توان چند مقدار را به یک آرگومان کیورد داد، همان طور که در خطای چاپ شده هم می بینید. در فراخوانیِ سوم هم چون تمامی آرگومان ها positional argument هستند، پس دیگر لازم نیست که ابتدا (3,)* مورد پردازش قرارگیرد(اگر این کار انجام شودآرگومان ها اشتباها مقدار دهی می شوند). پس اولین آرگومان مکانیِ ما یعنی ۱ به a داده می شود و بعدی که پردازش می شود یعنی ۳ به b داده می شود؛در برنامه نویسی با پایتون بهتر است از به همراه هم استفاده کردن ترکیب expression* و keyword argument ها خودداری کنید.

اگر ترکیب expression** مورد استفاده قرار گیرد، expression باید یک نوع داده mapping باشد. که محتویات آن به صورت آرگومان های کیورد (keyword arguments) مورد استفاده قرار خواهند گرفت. که در

این حالت محتویات این نوع داده ی mapping به عنوان keyword argument های اضافی تلقی می شوند. در صورت استفاده از یک keyword که هم در expression ما وجود داشته باشد و هم در میان آرگومان های کیورد (که مستقیما وارد شده اند) یک TypeError ایجاد می شود.

توجه نمایید که پارامتر هایی که از ترکیب expression* یا expression**استفاده می کنند، نمی توانند به عنوان جایگاه (slot) برای آرگومان های مکانی و یا کیورد برای آرگومان های کیورد استفاده شوند.

هر فراخوانی همیشه یک مقدار باز خواهد گرداند، شاید هم مقدار بازگردانده شده None باشد اگر یک فراخوانی حتی None هم برنگرداند پس حتما یک استثنا بر می گرداند. اینکه با مقدار بازگردانده شده چگونه برخورد می شود به نوع شئِ بازگردانده شده بستگی دارد. که در این جا تمام حالات را بررسی می کنیم:

اگر شئ فراخوانی شده یک تابع ساخته شده بدست کاربر باشد. در این صورت بلوک کدهای تابع اجرا می شود (در حالی که لیست آرگومان ها را به بدنه کدها داده می شود). اولین چیزی که بلوک کدهای تابع انجام خواهد داد این است که پارامترهای داده شده را در جای خاص خود در جایگاه آرگومان های تابع قرار می دهد. وقتی اجرا بلوک کد تمام شد یک عبارت return اجرا می شود این مقدار برگشتی تابع را مشخص می کند.

اگر شئ فراخوانی شده یک تابع یا متدِ built-in (داخلیِ پایتون) باشد . مقدار بازگردانده شده به اینترپرتر داده می شود.

اگر شئ فراخوانی شده یک شیئ کلاس باشد، یک نمونه ئ جدید از کلاس بازگردانده می شود.

اگر شئ فراخوانی شده یک متد نمونه ی کلاس باشد، تابع تعریف شده متناظرِ تعریف شده توسط کاربر توسط لیستی از آرگومان ها فراخوانی شده که این لیست به اندازه یک عضو طولانی تر از لیست آرگومان های فراخوانی است به این صورت که نمونه ی مذکور اولین آرگومان خواهد شد.

اگر شئ فراخوانی شده یک نمونه ی کلاس باشد آن کلاس باید یک متد ()__call__ را تعریف کند و نتیجه حاصل همان خواهد بود که از فراخوانیِ یک متد می گیریم.

تالیف: رامان عشقی، ۰۹۳۵۷۷۰۵۰۶۵ برای فیلم های آموزش پایتون به http://www.blue-python.tk مراجعه کنید.

فصل چهارم:

انواع داده collection

مرجع آموزش پایتون در ایران

http://www.blue-python.tk

مولف: رامان عشقی

در فصل گذشته اغلب انواع داده های مهمِ پایتون را آموخته اید. در این فصل شما یاد می گیرد که چگونه شرایط برنامه نویسی خود را با استفاده از انواع داده ی collection و مدیریت کردن داده هایتان در قالب این انواع بهتر نمایید.

نگهداری و استفاده از داده ها در قالب انواع داده ی collection کارهایی مانند انجام اعمال تکراری برای تمام آیتم ها را بسیار تسهیل می بخشد. همچنین کارِ خواندن اطلاعات از یک فایل را بسیار آسان می کند (مقادیر داخل فایل را داخل یک نوع داده ی collection می ریزیم). با خواندن و نوشتن فایل های متنی نیز در آینده آشنا خواهید شد.

انواع دنباله ای (sequence types)

یک نوع داده ی دنباله ای نوعی از داده ای است که دارای تمام ویژگی های زیر باشد:

۱- از عملگر عضویت (membership) یا همان in پشتیبانی کند.
۲- از تابع گرفتن طول یا len() پشتیبانی کند.
۳- از عملگر تکه تکه سازی یا همان [] پشتیبانی کند.

زبان پایتون پنج نوع داده ی دنباله ایِ داخلی را دارد: bytearray و bytes و list و str و tuple .

چندتایی ها (tuples)

یک تاپل (چند تایی) دنباله ی مرتبی از صفر یا تعداد بیشتری مرجع شئ است. tuple ها همان تکه تکه سازی و استخراج آیتم هایی را که str ها پشتیبانی می کنند را پشتیبانی می نمایند. این ویژگی کارِ بیرون کشیدن آیتم ها را از داخل یک tuple بسیار آسان می کند. مانند str ها که immutable یا غیر قابل تغییر بودند، tuple ها نیز غیرقابل تغییر هستند. لذا ما نمی توانیم یک یا تعدادی از آیتم های درون آنها را حذف یا عوض کنیم. اگر بخواهیم قادر به انجام اینچنین کارهایی باشیم راه حل ساده ای وجود دارد، به جای tuple ها از لیست ها استفاده می کنیم. البته اگر هم یک tuple داریم و می خواهیم آن را مستقیما به لیست تبدیل کنیم، می توانیم از متد list() استفاده کرده و تاپل را به عنوان آرگومان به آن پاس دهیم.

نوع داده ی tuple می تواند به عنوان یک تابع فراخوانی شود، مانند: tuple() . اگر این تابع را بدون آرگومان فراخوانی کنیم یک تاپل خالی را باز می گرداند. با یک آرگومان از نوع تاپل یک کپیِ سطحی از آرگومان خود را باز می گرداند. و با هر نوع آرگومان دیگری سعی می کند که آن را به تاپل تبدیل کند و بر گرداند. این تابع تنها یک آرگومان قبول می کند. تاپل ها همچنین می توانند بدون تابعِ tuple() آفریده شوند. یک تاپلِ خالی با استفاده از یک جفت پرانتز باز و بسته آفریده خواهد شد. و یک تاپل که شامل یک یا چندین آیتم باشد با کمک علامت کاما (,) که بین آیتم ها قرار می گیرد آفریده می شود. اگر بخواهید یک تاپل را به یک تابع دیگر به عنوان آرگومان پاس دهید، بهتر است که آیتم های تاپل را حتما در بین پرانتز های آن بنویسید. مانند دستور زیر:

```
>>> print(('I', 'am', 'in', 'a', 'tuple!'))
('I', 'am', 'in', 'a', 'tuple!')
```

همان طورکه می بینید،نتیجه ی چاپ شده یک تاپل است زیرا خودمان خواستیم تا یک تاپل چاپ گردد. به مثال های زیر درباره تاپل ها توجه کنید.

```
>>> t=tuple()
>>> type(t)
<class 'tuple'>
>>> len(t)
0
>>> t=tuple('hello world')
>>> t
('h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd')
>>> len(t)
11
```

در مثال زیر هم نحوه اندیس گذاریِ مرجع شئ هایی که داخل تاپل هستند را مشاهده می کنید، توجه کنید که اندیس گذاریِ عناصر یک چند تایی (تاپل) مانند str ها است با این فرق که در رشته ها ما در هر مکان فقط یک کاراکتر را داریم اما در تاپل ها ما در هر مکان یک مرجع شئ را داریم که می تواند به اشیایی از انواع مختلف در حافظه اشاره داشته باشد مثلا به اشایی از انواع int و float و ... .

```
>>> my_tuple=('one', 1, 12.4, '3', b'01')
>>> len(my_tuple)
5
>>> my_tuple[3]
'3'
>>> my_tuple[3:]
('3', b'01')
>>> my_tuple[::2]
('one', 12.4, b'01')
```

تاپل ها تنها دو متد را برایمان فراهم می کنند، یکی t.count(x) که تعداد اتفاق افتادن (وجود) شئ x را در تاپلِ t را باز می گرداند. و متد دیگر t.index(x) که اندیس اتفاق افتادن (وجود داشتن) شئ x را در چپ گرا ترین جایگاه در تاپل t باز می گرداند، اگر هم در تاپل t شئ x وجود نداشته باشد یک ValueError ایجاد می کند.

(اینها همان دو متدی هستند که برای لیست ها هم مورد استفاده قرار می گیرند). تاپل ها همچنین عملگرهایی مانند + , -, [ ], in, not in, *=, += پشتیبانی می کنند، نحوه استفاده و کارکرد این عملگرها در تاپل ها درست مانند انواع داده ی str است زیرا هر دوی آنها immutable می باشند. هنگامی که می خواهید تاپل ها را با هم مقایسه کنید می توانید از عملگرهای مقایسه ای استاندارد زبان های برنامه نویسی به راحتی استفاده نمایید: >=, <=, ==, !=, >, < . در استفاده از این عملگرهای مقایسه ای دقت داشته باشید که هنگام استفاده از آنها آیتم به آیتم تاپل ها با یکدیگر مقایسه می شوند، همچنین تاپل های تو در تو (آشیانه ای یا همان nested) نیز به تبع مورد بررسی و مقایسه با هم قرار خواهند گرفت. به مثال های زیر توجه نمایید:

```
>>> names='Raman', 'Mahammad', 'Mahdi', 'Mahan'
>>> type(names)
<class 'tuple'>
>>> nested_names = names[2:], names[0:1], names
>>> type(names)
<class 'tuple'>
>>> names
('Raman', 'Mahammad', 'Mahdi', 'Mahan')
>>> type(nested_names)
<class 'tuple'>
>>> nested_names
(('Mahdi', 'Mahan'), ('Raman',), ('Raman', 'Mahammad', 'Mahdi', 'Mahan'))
>>> #as you see nested_names is a nested tuple
```

به دستور زیر دقت کنید، ما در این خط کد یک تاپل تو در تو را با استفاده از الحاق سه تاپل دیگر ساخته ایم، اگر توجه کنید تاپل دوم که می خواهیم الحاق کنیم به شکل خاصی نوشته شده است، به این صورت که جدا از اینکه آن را داخل پرانتز ها گذاشته ایم، داخل همان پرانتز یک علامت جدا کننده ی کاما هم استفاده کردیم، اگر این کاما را نمی گذاشتیم چه؟ در این صورت چیزی که داخل پرانتز نوشته شده است یک نوع str به حساب می آمد که دیگر نمی توانستیم آن را با تاپل ها الحاق کنیم، در صورتی که شما بخواهید یک تاپل را با یک نوع str الحاق کنید یک TypeError دریافت خواهید نمود. به کدهای زیر توجه کنید که نشان گر این موضوع هستند:

```
>>> new_tuple = nested_names + ('characters',) +names
```

```
>>> a=('s')
>>> type(a)
<class 'str'>
>>> ('a', )+'s'
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    ('a', )+'s'
TypeError: can only concatenate tuple (not "str") to tuple
```

نکته دیگر اینکه اگر در این حالت پرانتزهای احاطه کننده ی آیتم را هم فراموش کنید باز هم TypeError دریافت خواهید کرد:

```
>>> ('a', )+'s', +(2,)
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    ('a', )+'s', +(2,)
TypeError: can only concatenate tuple (not "str") to tuple
```

ویژگی جالب تاپل ها این است که خیلی اوقات شما از آنها استفاده می کنید اما ممکن است به آن واقف نباشید ، همان طور که دیدید تاپل ها را می توان با استفاده از پرانتز آفرید اما در جایگاه های خاصی هم می توانید از گذاشتن پرانتزهای احاطه کننده ی آیتم ها صرف نظر کنید، به مثال زیر توجه نمایید:

```
>>> k=2,
>>> type(k)
<class 'tuple'>
```

یک روش کلی در برنامه نویسی پایتون وجود دارد که در آن پرانتزهای احاطه کننده ی آیتم های تاپل ها را در دو موقعیت خاص نمی نویسند یکی "سمت چپ عملگرهای دودویی (مانند + - =)" **و دیگری** "سمت راست عبارت یکانی است" **است. به مثال های زیر توجه نمایید:**

```
>>> a, b = 1, 2
>>> a, b
(1, 2)
>>> a
1
>>> b
2
>>> k=(a, b)
>>> type(k)
<class 'tuple'>
```

همان طور که گفته شد بهتر است برای جلوگیری از خطاهای احتمالی از پرانتزهای احاطه کننده استفاده شود نه اینکه صرف نظر شوند. در مثال زیر ما یک تاپل ایجاد کرده ایم، درون این تاپل در اندیس ۱ ما آیتمی از نوع list **داریم، سپس در داخل همین لیست در اندیس ۰ ما آیتمی از نوع** str **داریم و در داخل همین رشته ما تعدادی کاراکتر داریم که می توانیم با استفاده از عملگر تکه تکه سازی هر کدارم را که بخواهیم فراخوانی کنیم:**

```
>>> l=['I am a python programmer']
>>> tup='hello', l, ('hi there')
>>> type(tup)
<class 'tuple'>
>>> tup
('hello', ['I am a python programmer'], 'hi there')
>>> tup[1][::2]
['I am a python programmer']
>>> tup[1][0][::2]
'Ia  yhnporme'
```

نکته: آیا می دانید id() چیست؟ این یک تابع built-in در پایتون است که آرگومان آن هر شئ می تواند باشد(به یاد آورید که گفته شده همه چیز در پایتون یک شئ است)، و مقداری که بر می گرداند یک عدد صحیح (int) است. این عدد صحیح در واقع identity آن شئ (شئ ای که به عنوان پارامتر وارد شد) است و برای هر شئ یک عدد خاص و بی همتا است که با id هیچ کدام از اشیای دیگر برابری نمی کند، البته بعد از نابود شدن آن شئ id آن شئ می تواند به شئ جدیدی تعلق گیرد. به مثال زیر توجه کنید:

```
>>> id(34.300)
36286008
>>> id(type(34.300))
505297976
```

توجه نمایید که در CPython این تابع آدرس شیئ را در حافظه برمی گرداند.

تاپل های نام گذاری شده(named tuple)

یک تاپل نامگذاری شده درست مثل یک تاپل معمولی است از این جهت که ویژگی های ذاتیِ تاپل های معمولی را با خود دارد. اما چیزی که باعث تمایز آن با تاپلها می شود این است که به آیتم های آن نتنها بوسیله ی اندیس (مشابه تاپل های معمولی) بلکه بوسیله ی نام آیتم ها هم مراجعه کرد. این قابلیت مجموعه سازیِ انواع داده را به ما می دهد. قسمت جالب ماجرا راجع به تاپل های نام گذاری شده این است که اینها را نوع داده ی خاصی نمی توان دانست حتی در واقع می توان گفت اینها نوعی تاپل هم نیستند که آفریده می شوند، بلکه نوع داده ای هستند که شما آن را می آفرینید و شباهت آنها با تاپل ها است که باعث می شود namedtuple خوانده شوند. در حقیقت عناصر یک namedtuple توسط خصیصه ی نامگذاری شده می توانند مراجعه شوند، در کتابخانه و ماژولهای استاندارد پایتون هم از این انواع داده استفاده شده است، برای مثال در ماژول time ، تابع time.localtime() وجود دارد که با فراخوانیِ آن شما یک شئ دریافت می کنید که سال فعلی را می توانید با ترکیبی مانند t[0] یا t.tm_year بدست آورید. یک namedtuple می تواند یک نوع داده ی داخلیِ پایتون باشد (مانند time.struct_time) یا اینکه بوسیله ی تعریف یک کلاس عادی آفریده شود (شئ گرایی در بخشهای بعد مورد بررسی قرار می گیرد). یک namedtuple خوب را می توانید با استفاده از تابع collections.namedtuple() بیافرینید،

ماژول collections تابع namedtuple() را در خود دارد، این تابع برای آفرینش namedtuple ها بکار می رود. ما در زیر یک namedtuple را با همین تابع آفریده ایم:

```
>>> import collections
>>> NewClothings=collections.namedtuple('NewClothings', 'Pants Shirts')
>>> Pants=collections.namedtuple('Pants', 'jeans tonokeh')
>>> Shirts=collections.namedtuple('Shirts', 'Tshirt udershirt')
```

در مثال بالا اولین namedtuple که ساخته ایم، NewColthings است. بعدی ها هم Path و Shirts می باشند. به دستور زیر توجه کنید:

```
>>> type(NewClothings)
<class 'type'>
```

این برای شما جای سوال نیست؟ چون وقتی نوع NewClothings را خواستیم، نوع آن type گفته شد، چرا؟ چرا از نوع namedtuple نیست؟ جواب سوال این است که namedtuple تنها یک اسم است و وجود خارجی ندارد ، یعنی در مثال بالا شما یک *نوع* داده ی جدید ساخته اید نه یک instance از یک کلاس دیگر یعنی NewClothings خودش یک کلاس جدید است و نه یک شئ از یک کلاس دیگر، برای مثال str ، list و... نوع هستند و NewClothing s هم یک نوع داده ی جدید است. به دستور زیر توجه کنید:

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

در همین مثال مشاهده می کنید که داخل NewClothings از namedtuple های Pants و Shirts استفاده کردیم، این عمل مشابه تاپل های تو در تو است. در تابع collections.namedtuple() اولین آرگومان ما اسم نوع داده مان را مشخص می کند، نام انواع داده مانند نام کلاس ها باید با حروف بزرگ شروع شود، پس این قاعده را رعایت کنید، آرگومان دوم در واقع یک رشته است که کلمات داخل آن رشته با فضاهای خالی از هم جدا شده اند، هر کلمه در واقع یک property برای آن namedtuple محسوب می شود. اگر توجه کنید خواهید دید که این property ها را هم حرف اولشان با حرف بزرگ نوشتیم، این به این دلیل است که این ها خود قرار است یک namedtuple دیگر باشند.

```
>>> type(NewClothings.Pants)
<class 'property'>
>>> type(NewClothings)
<class 'type'>
```

اما چگونه می توان از این namedtuple هایی که ساخته ایم استفاده کنیم؟ به مثال زیر توجه نمایید:

```
>>> my_clothes=NewClothings(Pants('yes', 'no'), Shirts('no', 'yes'))
```

این مثال یک نمونه از ایجاد یک شیئ از این namedtuple است. به دستور زیر توجه کنید:

```
>>> type(my_clothes)
<class '__main__.NewClothings'>
```

برای مثال به این صورت می توانید از شئ my_clothes استفاده کنید:

```
>>> print('Is the pants that you were tonokeh?\n\t', my_clothes.Pants.tonokeh)
Is the pants that you were tonokeh?
        no

>>> print('{0.Pants.jeans}, {0.Shirts.Tshirt}'.format(my_clothes))
yes, no
```

namedtuple ها در پایتون تعداد کمی متدهای private دارند که این نوع متدها نامشان با یک علامت زیر خط (underscore) _ آغاز می شود. یکی از این نوع متدها namedtuple._asdict() است. موارد استفاده از این متد توضیح داده خواهد شد. پیش از اینکه به توضیح این متد بپردازم خوب است یاد آور می شوم هنگامی که از تابع collections.namedtuple() استفاده می کنید در واقع شما از این تابع خواسته اید که یک کلاس برای شما بسازد، خوب اگر واقعا کلاسی ساخته می شود چگونه می توان کدهایی که در ساختن این کلاس بکار می رود را مشاهده کرد؟ برای دیدن کدها تنها کافی است آرگومان سومی هم به تابع collections.namedtuple() بدهید، این یک keyword argument است و مقدار آن را برابر با True بگذارید، در این صورت بعد از اجرای دستور مشاهده می کنید که پس از ساخته شدن namedtuple درخواست شده، کدهایی که کلاس جدید ما را ساخته اند هم بازگردانده می شود. برای مثال:

```
NewClothings=collections.namedtuple('NewClothings', 'Pants Shirts', verbose=True)
```

حالا برگردیم به به بحث متدهای private اگر به کدهای بدنه ی تعریف کلاستان دقت کنید، تابع _asdict() را می بینید که تعریف شده است.

متد namedtuple._asdict() (که به صورت private است) یک شئ mapping شامل جفت هایی از کلید-مقدار (key-value) را باز می گرداند، به صورتی که هر key یکی از عناصر تاپل و هر value هم مقدار متناظر آن کلید می باشد.

تالیف: رامان عشقی، ۰۹۳۵۷۷۰۵۰۶۵ برای فیلم های آموزش پایتون به http://www.blue-python.tk مراجعه کنید.

لیست ها (lists)

یک لیست دنباله ای از صفر یا تعداد بیشتری مرجع شئ است. آیتم های درون لیست ها را همانطور که در تاپل ها و در رشته ها می توانستیم استخراج کنیم می توانیم با استفاده از عملگر تکه سازی [ ] بیرون بکشیم. برعکس رشته ها و تاپل ها لیست ها انواع داده هایی mutable هستند لذا ما می توانیم هر کدام از آیتم ها یا تکه هایی از آیتم های درون آنها را تعویض یا حذف کنیم یا اینکه تکه یا آیتمی جدید را به لیست خود اضافه کنیم.

نوع داده ی لیست هم می تواند به صورت یک تابع فراخوانی گردد. تابع list() بدون هیچ نوع آرگومانی یک لیست خالی را باز می گرداند با آرگومان هایی از نوع لیست یک کپیِ سطحی از آرگومان خود را باز می گرداند، و با آرگومانی از هر نوع دیگر سعی می کند که آرگومان را به لیست تبدیل کرده و لیست حاصل را باز گرداند، اگر در اینکار موفق نباشد یک TypeError باز می گرداند. مانند زیر:

```
>>> list(34)
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    list(34)
TypeError: 'int' object is not iterable
```

این تابع تنها یک آرگومان می پذیرد. لیست ها همچنین می توانند بدون استفاده از تابع list() آفریده شوند، یک لیست خالی با استفاده از [ ] آفریده خواهد شد. و یک لیست شامل تعداد بیشتری آیتم هم با استفاده از عناصری از هر نوع که کنار هم قرار گرفته باشند و با علامت کاما (,) از یکدیگر جدا شده باشند و توسط یک جفت براکت احاطه شده باشند آفریده می شود. لیست ها را می توان با استفاده از عملگرهای استاندارد مقایسه ای با هم مقایسه نمود، نحوه مقایسه ئ آن ها با هم مانند تاپل ها است. همچنین دسترسی به عناصر و آیتم های یک لیست با استفاده از عملگر تکه سازی دقیقا به همان صورتی است که در تاپل ها هم انجام می گرفت، لذا از اضافه گویی می پرهیزیم.

در اینجا به یک مثال ساده توجه کنید:

```
>>> my_list=["my name's Raman", 'date', ['$', 46], (8,), 3.4]
>>> type(my_list), my_list.__len__()
(<class 'list'>, 5)
```

همانطور که مشاهده کردید، ما برای بدست آوردن طول لیست خود می توانیم به جای استفاده از تابع built-in ، len() از متد __len__() بهره بگیریم.

ما برای دسترسی به آیتم های درون لیست می توانیم از عملگر تکه سازی (slice operator) یا همان [ ] استفاده کنیم. اما با این وجود اگر بخواهیم بدون دردسر بخواهیم دو یا تعداد بیشتری از آیتم های درون لیست را بیرون

کشیده و بازگردانیم، می توانیم از عملگر بازکننده ی دنباله (sequence unpacking) استفاده نماییم. هر نوع داده ای که iterable باشد (تکرار شدنی باشد) مانند تاپل ها، لیست ها و... می تواند به وسیله ی عملگر unpacking sequence باز شود، این عملگر همان علامت * یا ستاره است (star/ asterisk). وقتی این عملگر را با دو یا تعداد بیشتری متغیر در طرف چپ یک مساوی قرار می دهید، که یکی از این متغیر ها در کنار (در کنار سمت چپ خود) علامت ستاره را دارد، آیتم ها به متغیرها داده خواهند شد به ترتیبی که تمام اشیایی که (مراجع شئ که) در سمت راست مساوی باقی می مانند به متغیرِ پس از ستاره داده می شوند. به مثال ها توجه کنید:

```
>>> a, b, *c = my_list
>>> a
"my name's Raman"
>>> b
'date'
>>> c
[['$', 46], (8,), 3.4]
```

در مثال بالا می بینید که متغیر های a و b به ترتیب صفرمین و اولین آیتم های درون لیست را گرفته اند و متغیر c هم چون از عملگر ستاره استفاده کرده ایم باقی مانده ی لیست را به خود اختصاص داده است. وقتی عملگر ستاره (sequence unpacking operator) این گونه مورد استفاده قرار می گیرد، عبارت c* و عبارات مشابه آن را عبارات ستاره دار (stared expressions) می گویند.

پایتون همچنین یک مفهوم مرتبطی با این قضیه دارد که به آن آرگومان های ستاره دار (stared arguments) می گویند، در اینجا یک مثال مطرح می کنیم، فرض کنیم که تابعی مانند تابع زیر داریم:

```
>>> def fun(i, j, k):
        return None
```

همانطور که می بینید این تابع چیز خاصی را باز نمی گرداند اما برای اجرای آن دقیقا به سه آرگومان نیاز داریم، این آرگومان ها را هم می توانیم به صورت مستقیم و ساده وارد نماییم و هم با استفاده از آرگومان های ستاره دار. به مثال ها دقت کنید:

```
>>> fun(2, 3, 4)
>>> my_list=[2, 3, 4]
>>> fun(*my_list)
>>> fun(*[2, 3, 4])
>>> fun(*(2, 3, 4))
>>> fun(2, *my_list[1:])
```

در دومین فراخوانی به بعد ما از آرگومان های ستاره دار استفاده نموده ایم، اما توجه کنید که در فراخوانیِ سوم و چهارم یکی به ترتیب ازعملگر ستاره برروی لیست و تاپل استفاده نموده ایم. چیزی که درفراخوانیِ دوم رخ می دهد این است که لیست سه آیتمیِ ما با استفاده از عملگر * باز می شود و هر کدام از آیتم ها درون یکی از آرگومان ها

قرار می گیرند. توجه نمایید که اگر تعداد آیتم های لیست باز شده از تعداد آرگومان های تابع بیشتر باشد یک TypeError ایجاد خواهد شد. درباره این عملگر مفصلاً در بخشی به نام "**فراخوانی ها**" توضیح خواهیم داد.

آیا می دانید چگونه می توان تمامیِ آیتم های درون یک لیست را دانه به دانه بیرون کشید؟ به مثال زیر توجه نمایید:

```
>>> for i in range(len(my_list)):
        print(my_list[i])
```

آیا می دانید چگونه می توان تمام یا حداقل بخش وسیعی از آیتم های داخل لیست ها را تغییر داد؟ به مثال زیر توجه کنید که چگونه این کار را انجام داده است:

```
>>> for i in range(len(my_list)):
        my_list[i]=change(my_list[i])
```

تابع built-in ، range() یک تکرار شونده (iterator) را بر می گرداند که برای ما تعدادی عدد صحیح فراهم می کند. وقتی تابع range() تنها یک آرگومان (از نوع int) داشته باشد برای مثال مانند: range(n) در این صورت iterator ی که این تابع باز می گرداند می تواند اعداد صحیحِ $0, 1, 2, \ldots, n-1$ را برای ما تولید کند که میتوانیم در برخی جاها مانند حلقه های for از آنها استفاده کنیم.

pythonic چیست؟

این یک اصطلاح به معنای استفاده از امکانات خاص پایتون است، امکاناتی که زبان های دیگر از داشتن آن عاجزند، پایتون از ابتدا هم برای ساده کردن کدها و افزایش سرعت گسترش نرم افزار ها آفریده شده است، سراسر زبان پایتون و کتابخانه استاندارد آن، امکاناتی را برایتان فراهم می کند تا بتوانید به ساده ترین نحو ممکن کدنویسی کنید تا در کمترین زمان ممکن به نتیجه دلخواه برسید. آیا آنچه در دو مثال بالا دیده اید همان چیزی است که اصول زبان پایتون می گوید؟ نه! دو مثالی که در بالا مشاهده کردید قطعا خرابکاریِ یک برنامه نویس C# است! برای گرفتن آیتم های درون یک لیست در پایتون می توانید از کدی به مراتب ساده تر استفاده کنید، مانند زیر:

```
>>> for elements in my_list:
        print(elements)
```

یکی از اصول اساسیِ برنامه نویسیِ پایتون سادگی و تمیز بودن کد است که در مثال بالا آن را می بینید، می توانید اصول پایتون را که هنگام برنامه نویسی به آن پایبند باشید را با دستور import this مشاهده کنید:

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
```

عقاید پایتون (توسط تیم پیترز):

زیبا بهتر از زشت است *** مستقیم (صریح) بهتر از غیر مستقیم است *** ساده بهتر از پیچیده است *** پیچیده بهتر از فوق العاده پیچیده است *** مسطح بهتر از تو در تو (آشیانه ای) است *** پراکنده (رقیق) بهتر از متراکم (غلیظ) است *** قابلیت خوانایی کد یک اصل است *** موارد خاص، آنقدر خاص نیستند که مقررات را بشکنند اما بهر حال به خلوص کدها ضربه می زنند *** Error ها هرگز نباید به سادگی نادیده گرفته شوند مگر اینکه مستقیما ساکت شوند! *** هنگامی که از کدها گیج شده اید، وسوسه ئ حدس زدن را قبول نکنید *** حتما و حتما یک راه کاملا آشکار برای انجام آن وجود دارد گرچه این راه ممکن است در ابتدا خیلی روشن نباشد *** حالا خیلی بهتر از هرگز است *** گرچه که هرگز از همین حالا بهتر است *** اگر تشریح پیاده سازی ها مشکل است، حتما کد شما روش خوبی نبوده *** اگر تشریح پیاده سازی ها آسان است کد شما می تواند چیز خوبی باشد *** فضای نام ها (namespaces) نظر خوبی هستند بیاید بیشتر از آنها استفاده کنیم***

تالیف: رامان عشقی، ۰۹۳۵۷۷۰۵۰۶۵ برای فیلم های آموزش پایتون به http://www.blue-python.tk مراجعه کنید.

حالا بیاید به لیست ها باز گردیم، در زیر یک جدول را می بینید که در آن متدهای قابل دسترسی توسط اشیای لیست با کمی توضیح آورده شده است:

ترکیب	کاربرد
L.append(x)	آیتم x را به انتهای لیست L اضافه می کند
L.count(x)	تعداد آیتم x داخل لیست L را می شمارد و باز می گرداند
L.extend(m) L += m	تمامیه آیتم های قابل تکرار داخل iterable آرگومان خود یعنی m را به L اضافه می کند، مثلا تمام کارکترهای داخل رشته را به تک تک به عنوان آیتم به L اضافه می کند، مشابه این کار را ترکیب L+=m هم انجام می دهد.
L.index(x, *start*, *end*)	اندیس مکان شیئ x را داخل لیست L بر می گرداند (اولین مکان وجود x از چپ) آرگومان های start وend به ترتیب نقاط شروع و پایان جستو جو را مشخص می کنند
L.insert(i, x)	آیتم x را در مکان اندیس i به لیست اضافه می کند، i باید حتما int باشد.
L.pop()	آیتمی که در راست ترین جایگاه (اندیس -1) لیست قرار دارد را حذف کرده و برمی گرداند.
L.pop(i)	آیتمی که در اندیس i قرار دارد را برگردانده و حذف می کند
L.remove(x)	اولین اتفاق افتادن x را در L از سمت چپ پیدا کرده و آن آیتم را حذف می کند، جستو جو همیشه از سمت چپ صورت می گیرد.
L.reverse()	لیست را درجا می چرخاند و برعکس می کند. عملا اندیس هر آیتم را i-1- می کند
L.sort(...)	لیست را مرتب می کند (بر اساس ترتیب الفبایی و ...) آرگومان های دیگری که می گیرد، کاری مشابه کار تابع داخلی sorted() انجام می دهند.

در زیر مثال هایی از متدهای موجود در جدول را می بینید:

```
>>> l=['a', 'b', 'c']
>>> l.extend('0123456789')
>>> l.sort()
>>> l
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c']
>>> l.append('help me!')
>>> l
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c', 'help me!']
>>> l.pop()
'help me!'
>>> l
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c']
>>> l.reverse()
>>> l
['c', 'b', 'a', '9', '8', '7', '6', '5', '4', '3', '2', '1', '0']
>>> l.index('3')
9
```

نکته : تا بحال از کلماتی مانند statement یا بلوک کد یا سوییت (suite) یا expression یا ... را استفاده کرده ایم ، اما بهتر است بدانید که فرق اینها دقیقا با یکدیگر چیست. یک statement بخشی از یک بلوک کد است و بلوک کد همان suite است. یک بلوک کد یا سوییت قسمتی از کد است که با یک واحد خاص دندانه گذاری از بقیه کد جدا شده است. یک statement همان expression است. یک statement می تواند بخشی از کد باشد که با کلمات کلیدی ای همچون if, while, for می آید.

استفاده از del statement (عبارت del)

اگرچه نام del یادآور کلمه ی delete به معنای حذف کردن است، این عبارت لزوما خود اشیا (داده ها) را حذف نمی کند. وقتی آن را بر روی یک مرجع شئ که به یک آیتم غیر collection اشاره می کند استفاده می کنیم، این عبارت مرجع شئ را از آن شئ جدا کرده و مرجع شئ را حذف می کند (مرجعِ شئ نه شئ) برای مثال:

```
>>> a='I am a computer'
>>> a
'I am a computer'
>>> del a
```

حالا به نتیجه فراخوانیِ مرجع شئ پس از نابود سازیِ آن توجه کنید:

```
>>> a
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    a
NameError: name 'a' is not defined
```

همانطور که می بینید شما یک استثنای NameError دریافت کردید، به این دلیل که نام a تعریف نشده است، البته که این نام قبلا تعریف شده است اما در حال حاضر ما این مرجع شئ را حذف کرده ایم. خوب مرجع شئ حذف شده است حالا چه اتفاقی برای خود شئ (داده های ما که در اینجا همان رشته ی 'I am a computer' می باشد) خواهد افتاد؟ ابتدا چک می شود که آیا هیچ مرجع شئ دیگری برای این شئ وجود دارد یا نه؟ اگر وجود داشت که شئ پابرجا می ماند و گرنه این شئ برای garbage collection آماده سازی می شود. اما اینکه واقعا garbage collection کی اتفاق می افتد یا اینکه حتی اتفاق می افتد یا نه غیر قابل پیش بینی است و به شدت به پلتفرم و نوع توزیع پایتون بستگی دارد. لذا برای اینکه مطمئن شویم که اشیای اضافی در حافظه باقی نمی مانند پایتون برای ما دو راه گذاشته است، یکی استفاده از بلوک های try… finally که در بخش های پیشین درباره آنها گفته شد و دیگر استفاده از عبارت with یا with statement است. درباره این مباحث در آینده بحث خواهیم نمود.

وقتی از del بر روی یک نوع داده ی collection استفاده شود، تنها مرجع شئ به آن collection از بین خواهد رفت.خودcollection و آیتم های درون آن (و آیتم های درون آن که خودهمcollection هستند به صورت تو درتو) تماما برای garbage–collection آماده سازی می شوند در صورتی که مرجع شئ دیگری به آنها اشاره نداشته باشد.

برای انواع داده ی collection که mutable هم هستند مانند list ها del را حتی روی تک آیتم ها و یا تکه ای از چندین آیتم استفاده کرد. در هر دو حالت باید از عملگر تکه سازی [ ] استفاده کنیم. مانند دیگر مرجع اشیایی که تا به حال گفتیم اگر آیتم های داخل لیست ها (یا انواع داده ی دیگر mutable collection) اگر مرجع شئ دیگری به داده هایی که در جایگاه آن آیتم بودند (اشیا) اشاره نداشته باشد این اشیا هم برای garbage– collection آماده سازی می شوند.

نکته: garbage collection پروسه ی آزاد سازیِ بخشی از حافظه است که دیگر از آن استفاده نمی شود. پایتون عمل garbage collection را با کمک شمارش مراجع شئ و cyclic garbage collector انجام می دهد.

list comprehentions (مفاهیم لیست)

بار دیگر به مثال هایی که پیرامون لیست ها بیان کرده ایم باز گردید، و مرور کنید. به آن مثالی که در آن با استفاده از یک حلقه ی for آیتم هایی را به لیست اضافه نموده ایم، توجه کنید، به نظر می رسد روش خوبی باشد اما شاید بهتر باشد از کدهای دیگری برای انجام همین کار استفاده کنیم، به دستور زیر توجه کنید:

```
>>> l=[]
>>> l.extend('0123456789')
>>> l
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

با دستوراتی که می بینید ما یک کلاس آفریده و سپس با استفاده از قطعه قطعه کردن آیتم های کاراکتریِ داخل رشته که قابلیت iterate شدن را دارد (iterator است) سعی کرده ایم آنها را داخل یک لیست ذخیره کنیم و لیست جدید را بسازیم.

همین کارا را با کد زیر هم می توانستیم انجام دهیم:

```
>>> l=[i for i in '0123456789']
>>> l
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

در این کد ما از list comprehention برای آفرینش لیست استفاده کرده ایم . ترکیب استفاده از list comprehention به صورت زیر است:

```
[item for item in iterable]
```

این نوع استفاده از list comprehention دقیقا کاربرد یکسانی با کدهای توضیح داده شده ئ پیشین دارد، اما دو چیز که سبب می شوند list comprehention ها جالب ترو قدرتمند تر باشند این است که ما هم می توانیم از expression ها استفاده کنیم و هم می توانیم یک شرط را به list comprehention متصل کنیم، این قابلیت ها دو ترکیب کلی دیگر به ما می دهند که می توانیم از آنها هم استفاده کنیم:

```
[expression for item in iterable]
```

```
[expression for item in iterable if condition]
```

برای این ترکیب ها مثال هایی آورده شده است به این مثال ها توجه نمایید:

```
>>> years=[y for y in range(1362, 1366)]
>>> leap_years=[y for y in range(1362, 1366) if y % 4 == 0]
>>> leap_years
[1364]
>>> other_years=[y for y in range(1900, 1940)
            if (y % 4==0 and y % 100!=0) or (y % 4==0)]
>>> other_years
[1900, 1904, 1908, 1912, 1916, 1920, 1924, 1928, 1932, 1936]
```

اولین مثال کاربرد ساده ی list comprehention است و مثال دوم کاربرد ترکیب دوم و مثال سوم هم کاربردی از ترکیب سوم است.

hashable

یک شئ زمانی hashable است که دارای یک مقدار hash باشد که هرگز در طول حیات آن شئ تغییر نمی کند. این شئ اگر hashable باشد حتما یک متدِ __hash__() را دارد و می تواند با اشیای دیگر مقایسه شود (بوسیله ی متد __eq__() با اشیای دیگر مقایسه می شود). اشیایی که اینگونه مقایسه می شوند باید حتما دارای یک مقدار یکسانِ hash باشند.

قابلیتِ hashable بودن باعث می گردد که یک شیئ بتواند به عنوان key در دیکشنری ها و یا یکی از اعضای ستها استفاده شود این به این خاطر است که این ساختار های داده در درون خود از مقدار hash یا همان hash value استفاده می کنند.

تمامیِ اشیای built-in پایتون که immutable هستند، همچنین hashable نیز می باشند و این در صورتی است که هیچکدام از اشیای mutable داخلی پایتون مانند دیکشنری ها و لیست ها hashable نیستند. تمامی اشیایی که نمونه هایی از کلاسهای تعریف شده توسط کاربر هستند به صورت پیشفرض hashable هستند و مقدار hash آنها هم مقداری است که از تابع id() روی آن شئ باز می گردد.

immutable

یک شئ immutable شئ است که دارای مقداری ثابت است، اشیای immutable شامل اعداد (تمامی انواع اعداد) و رشته ها و تاپل ها می باشند. این نوع از اشیا نمی توانند تغییر پیدا کنند. در صورتی که شما بخواهید تغییرات و اصلاحاتی روی این اشیا انجام دهید باید یک شئ جدید ایجاد شود که مقدار جدید را در خود دارد و مرجع شئ به آن شئ جدید اشاره می کند. این نوع اشیا در موقعیت هایی که یک مقدار ثابت hash مورد نیاز است نقش مهمی را ایفا می کنند. برای مثال به عنوان یک key در دیکشنری ها.

iterable چیست؟

iterable شئ است که قادر به بازگردانیِ هر کدام از اعضای خود به صورت یک به یک است. برای مثال تمامی انواع داده ی دنباله ای (sequence type) مانند str ها، لیست ها و تاپل ها و برخی انواع داده ی غیر دنباله ای (non-sequence) دیکشنری و فایل ها (در هنگام خوانده شدن) و همچنین اشیایی از هر کلاسی که دارای متد های iter__() و getitem__() است. مانند:

```
>>> l=['s', 3, 5.6, (45, )]
>>> l.__getitem__(1)
3
```

iterable ها (اشیای iterable) می توانند در یک حلقه ی for یا در خیلی جاهای دیگر که یک دنباله نیاز است (مانند zip() و map()) مورد استفاده قرار گیرند. وقتی که یک شئِ iterable به تابع داخلی iter() به عنوان آرگومان داده می شود، این تابع iterator مربوط به آن شئِ iterable را باز می گرداند. برای استفاده از اشیای iterable لازم نیست که حتما از تابع iter() استفاده کنیم، زیرا حلقه ی for در بیشتر موارد کار آن را انجام می دهد.

Iterator چیست؟

یک شئ که نماینده ی جریانی از داده ها است را iterator گویند، در بخشهای پیشین گفته شد که با استفاده از تابع iter() که آرگومان آن یک شئ iterable است می توان یک شئ iterator گرفت. فراخوانی های پی در پی به متد

__next__() روی iterator یا استفاده از تابع next() با آرگومانی از شئ iterator آیتم های پی در پیِ جریان را باز می گرداند. به مثال توجه کنید:

```
>>> l.__iter__().__next__()
's'
>>> l
['s', 3, 5.6, (45,)]
```

```
>>> i=iter(l)
>>> next(i)
's'
>>> next(i)
3
>>> next(i)
5.6
```

هنگامی که این فراخوانی ها به حدی می رسد که دیگر آیتمی در دسترس نیست (همه آیتم ها برگردانده شده اند)، یک استثنای StopIteration بازگردانده می شود. در این حالت اصطلاحاً می گوییم شئ iterable ما تحلیل رفته و مصرف شده است و دیگر هر فرخوانی از متد __iter__() یا تابع iter() همین استثنا را باز خواهد گرداند. پس iterator ها اشیای خوبی برای گرفتن آیتم ها ی درون یک شئ iterator هستند اما برای یک بار.

```
>>> next(i)
5.6
>>> next(i)
(45,)
>>> next(i)
Traceback (most recent call last):
  File "<pyshell#17>", line 1, in <module>
    next(i)
StopIteration
```

همانطور که گفته شد هر شیئ که دارای یک متد __iter__() باشد یک iterable است. لذا می توانیم نتیجه بگیریم که هر iterator (که خود از یک شئ iterable ایجاد شده) خودش یک نوع شئ iterable هم هست و می تواند در ایجاد iterator های بعدی مورد استفاده قرار گیرد. توجه کنید که اگر از یک شئ iterator تحلیل رفته برای ساختن iterator بعدی استفاده کنید، در واقع شما از یک iterable خالی استفاده نموده اید و نتیجه ی آن این است که iterator بعدی که از این iterable ساخته می شود خود یک iterator تحلیل رفته است.

نکته: در IDLE پایتون (python shell) برای اینکه بتوانید از documentation یا مستند سازی های یک کلاس، متد یا ... را ببینید و آن را مطالعه کنید، باید از تابع داخلی help() استفاده نمایید. ترکیب زیر را مشاهده کنید:

تالیف: رامان عشقی، ۰۹۳۵۷۷۰۵۰۶۵ برای فیلم های آموزش پایتون به http://www.blue-python.tk مراجعه کنید.

```
>>> help(str)
Help on class str in module builtins:

class str(object)
 |  str(string[, encoding[, errors]])
```

انواع set

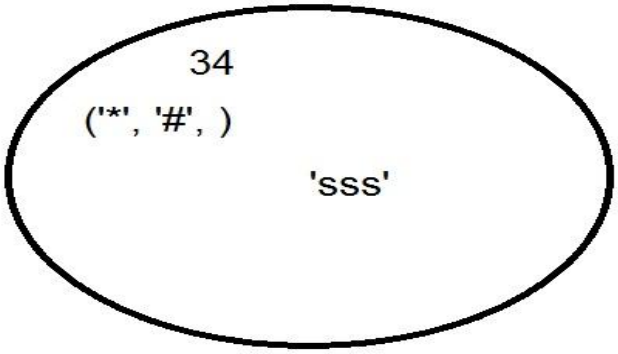
یک نوع داده ی set ، یک نوع داده ی collection است که از عملگر عضویت (membership) یا همان in و تابع گرفتن سایز یا همان len() و در ضمن iterable نیز هست. به علاوه انواع set حداقل یک متدِ set.isdisjoint() را فراهم می کنند و قابلیت مقایسه شدن با یکدیگر را هم دارند. در پایتون دو نوع داده ی set وجود دارند یکی از آنها mutable است که آن نوع set است و دیگری immutable است که به آن frozenset می گویند. زمانی که انواع داده ی set را iterate می کنیم، آیتم های این نوع داده ها در یک ترتیب دلخواه (نامرتب و بی نظم) به ما می دهند. تنها اشیای hashble می توانند به یک set اضافه شوند. اشیای hashable اشیایی هستند که دارای یک متد خاصِ __hash__() هستند که این متد مقداری که باز می گرداند همواره در طول وجود داشتن شئ ما یکسان است. تمامی انواع داده ی داخلی پایتون که immutable هستند همچینین hashable هم هستند و لذا می توانند به set ها اضافه گردند این شاملِ forzenset هم می گردد. اما انواع داده ای که mutable هستند مانند set, list, dict ها نمی توانند به set ها اضافه شوند زیرا این انواع hashable نیستند. پس set ها ی تو در تو (آشیانه ای) عملا وجود ندارند.

عمل مقایسه در set ها همانگونه انجام می گیرد که در انواع داده هایی مانند تاپل ها صورت می گیرد. به عنوان مثال در مورد set هایی که انواع تاپل یا forzonset را در داخل خود دارند عمل مقایسه به آیتم های درون این آیتم ها هم کشیده می شود.

set

یک set یک collection نامرتب از صفر یا تعداد بیشتری مراجع شئ است که به اشیای hashable رجوع دارند. set ها انواع داده ی mutable هستند لذا به راحتی می توانیم آیتم ها را از set ها اضافه یا کم کنیم. اما چون نامنظم هستند پس هیچ ترتیب اندیسی ندارند بنابراین نمی توان آنها را مانند str ها تکه تکه کرد. تصویر زیر نمایی است از آنچه توسط set نگهداری می شود:

```
>>> my_set={34, ('*', '#',), 'ssss'}
```



نوع داده ی set می تواند به عنوان یک تابع فراخوانی گردد، بدون هیچ آرگومانی یک set خالی باز می گرداند با یک آرگومان از نوع set یک کپیِ سطحی از آرگومان را برمی گرداند و با هر گونه آرگومان دیگری سعی می کند که آن را به set تبدیل کرده و باز گرداند. این تابع تنها یک آرگومان می پذیرد. set های غیر خالی را هم می توان بدون تابع set() آفرید اما یک set خالی حتما باید توسط تابع set() ایجاد گردد و نمی توان ستهای خالی را با استفاده از آکلاد های خالی {} آفرید زیرا آکلاد های خالی برای آفریدن دیکشنری ها استفاده می شوند (این انواع داده هم در آینده توضیح داده می شوند). یک set غیر خالی که شامل یک یا تعداد بیشتری آیتم باشد را می توان به وسیله ی دنباله ای از آیتم ها که با کاما از هم جدا شده اند و توسط یک جفت علامت { } احاطه شده اند ایجاد نمود. راه دیگر ایجاد set ها استفاده از مفاهیم set یا set comprehention است.

شما در استفاده از آیتم های تکراری در set ها آزاد هستید اما همانطور که در مثال زیر نشان داده شده است set ها آیتم های تکراری را حذف می نمایند. در واقع باید گفت یکی از موارد استفاده ی set ها جایی است که می خواهید آیتم های تکراری را حذف نمایید.

```
>>> my_set
{'ssss', ('*', '#'), 34}
>>> my_set.add(34)
>>> my_set
{'ssss', ('*', '#'), 34}
```

تالیف: رامان عشقی، ۰۹۳۵۷۷۰۵۰۶۵ برای فیلم های آموزش پایتون به http://www.blue-python.tk مراجعه کنید.

set ها عملگرهای in و همچنین not in را پشتیبانی می کنند . برای گرفتن تعداد آیتم ها هم می توان از تابع داخلیِ len() استفاده نمود. در مثال زیر نحوه ی ایجاد یک ست را می بینید:

```
>>> set('help us')
{' ', 'e', 'h', 'l', 'p', 's', 'u'}
```

لیستی از متدهایی که می توان برای set ها به کار برد در جدول زیر آورده شده است،

ترکیب	کاربرد
s.add(x)	آیتم x را به ست s اضافه می کند
s.clear()	تمامی آیتم های ست s را پاک می کند
s.copy()	یک کپی سطحی از ست S باز می گرداند
s.difference(t) s - t	یک ست از آیتمهایی را برمی گرداند که در s هستند ولی در ست t نه
s.discard(x)	اگر آیتم x را در ست s پیدا کند آن را حذف خواهد نمود

به مثال های زیر درباره ی set ها توجه نمایید:

```
>>> a_set={1}
>>> a_set
{1}
>>> type(a_set)
<class 'set'>
>>> a_set.add('I am another element')
>>> a_set.add(dir())
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    a_set.add(dir())
TypeError: unhashable type: 'list'
>>> type(dir())
<class 'list'>
>>> a_set.discard(453)
>>> a_set
{'I am another element', 1}
>>> a_set.clear()
```

همانطور که می بینید در دستور پنجم سعی کرده ایم که شئ ای که از تابع داخلیِ dir() بازگردانده می شود را به عنوان یکی از آیتم های set به آن اضافه کنیم اما TypeError دریافت کرده ایم، این به این دلیل است که مقدار بازگردانده شده توسط تابع dir() یک لیست است، لیست ها انواع داده ای mutable هستند و mutable ها

hashable نیستند و لذا چون آیتم های یک set باید حتما hashable باشند هرگز نمی توان یک لیست را به عنوان یک آیتم به set اضافه نمود. در آخرین دستور استفاده شده ما تمامی آیتم های داخل ست را پاک کرده ایم.

همانطور که در مثال زیر می بینید نمی توان با استفاده از یک جفت { } خالی یک سِت ایجاد کرد زیرا در این حالت شما یک دیکشنری ایجاد خواهید نمود اما میتوانید برای ایجاد set های با تعداد یک یا بیشتر آیتم می توان از { } استفاده نمود. در عوض برای آفریدن یک سِت خالی می توانید از تابع set() استفاده کنید:

```
>>> new_obj={}
>>> type(new_obj)
<class 'dict'>
>>> empty_set=set()
>>> empty_set
set()
>>> type(empty_set)
<class 'set'>
```

همانطور که مشاهده می کنید از سِت ها نمی توان به همان اندازه و گستردگی که از تاپل ها و از list ها استفاده می کردیم استفاده نماییم، اما با این حال موارد استفاده خاصی هم دارند. سِت ها برا ی مقایسه کردن دو دنباله از داده ها می توانند مورد استفاده قرار گیرند و سپس بعد از انجام عملیات لازم می توان سِتی را که نیاز داریم تا آیتم های آن را فراخوانی یا حذف یا اعمال دیگری را روی آن انجام دهیم را به یک لیست یا تاپل تبدیل کنیم انجام اینکار به سادگی پاس دادن یک آرگومان از نوع سِت می باشد. به مثال زیر توجه کنید:

```
>>> a_set.add('4')
>>> list(a_set)
['4']
>>> tuple(a_set)
('4',)
```

یکی از مهمترین کاربرد سِتها هم همانطور که گفته شد این است که آیتم های تکراری در آنها نگه داری نمی شوند و از هر آیتم فقط و فقط یکی وجود دارد. به یاد داشته باشید که یک سِت خود نمی تواند داخل سِت دیگری قرار بگیرد زیرا یک سِت خودش mutable است پس نمی تواند hashable باشد.

نکته: حدس می زنید None چه باشد؟ آیا None صفر است؟ آیا یک است؟ آیا هر چیز تهی None است؟ مثلا آیا یک لیست خالی None است؟ پاسخ منفی است. در پایتون None شئ ای از کلاس NoneType است. هنگامی که None را با هر شئ دیگری در پایتون مقایسه کنید متوجه می شوید که با هیچکدام برابر نیست.

تالیف: رامان عشقی، ۰۹۳۵۷۷۰۵۰۶۵ برای فیلم های آموزش پایتون به http://www.blue-python.tk مراجعه کنید.

مفاهیم سِت (set comprehention)

همان طوری که گفته شد علاوه بر آفریدن سِت ها به وسیله ی تابع set() و { } ما همچنین می توانیم از set comprehention استفاده کنیم، این ترکیب یک expression و یک حلقه است به علاوه ی شروط دیگری همراه آن نوشته می شوند . ترکیب استفاده از آنها به صورت زیر می باشد:

```
{expression for item in iterable}
{expression for item in iterable if condition}
```

```
>>> {x for x in empty_set if True}
set()
>>> {x for x in a_set if True}
{'4'}
```

سِتهای یخ زده(frozensets)

یک frozenset ستی است که وقتی آفریده شد دیگر نمی تواند تغییر کند، به هر حال این به این معنا نیست که این نوع داده را نمی توان دستکاری کرد بلکه به این معناست که frozenset ها immutable هستند. frozenset ها تنها از یک راه می توانند آفریده شوند و آن هم از طریق تابع frozenset() است. بدون هر گونه آرگومانی یک frozenset خالی ایجاد و بر می گرداندو با آرگومانی از نوعfrozensetیک کپی سطحی از آرگومان را برمی گرداند و با هر گونه دیگر از آرگومان سعی می کند که آرگومان را به frozenset تبدیل کند و برگرداند. این تابع تنها یک آرگومان قبول می کند.

به این دلیل که frozenset ها immutable هستند، لذا تنها متدهایی را پشتیبانی می کنند که اثری روی خود آن frozenset که از روی آن فراخوانی شده اند نداشته باشند.

بازکردن و نوشتن فایلهای ساده

برای ذخیره و استفاده مجدد اطلاعات باید یک زبان برنامه نویسی دارای قابلیت های پیشرفته و high-level برای خواندن و نوشتن انواع فایل ها باشد . در این بخش به کار با فایل های متنی که شما آنها را بیشتر با پسوند txt می شناسید می پردازیم. فلسفه ی کلی کار با فایل ها در پایتون این است: باز کن، استفاده کن، ببند. بستن فایلها بعد از استفاده از آنها از اهمیت بالایی برخوردار است، به این دلیل که ممکن است اگر یک فایل به صورت باز رها شود، اطلاعات ذخیره شده در آن دچار مشکل شوند. برای باز کردن فایل ها از تابع open() استفاده می شود. توجه کنید:

```
open(file, mode='r', buffering=-1, encoding=None, errors=None, closefd=True)
```

اولین آرگومان این تابع، آدرس کامل و نام فایلی است که باید برای خواندن یا نوشتن باز شود. همانطور که می بینید به جز آرگومان اول تمامی آرگومان ها مقدار دهی اولیه دارند، لذا می توانید در صورتی که نیازی را به مقدار دهی مجدد آن آرگومانها احساس نمی کنید، هنگام فراخوانی آنها را مقدار دهی نکنید. به این معنا که می توانید در فراخوانی این تابع تنها یک پارامتر را بکار ببرید. به مثال زیر توجه کنید:

```
>>> newfile=open('users.txt')
```

مطلب ذکر شده در مثال بالا به کار برده شده است. آرگومان دوم یعنی mode ، حالتی است که می خواهیم فایل را آنگونه باز کنیم، برای مثال شما گاهی اوقات فایل را فقط برای خواندن و گاهی اوقات فقط برای نوشتن و... باز می کنید. در این جا، اگر آرگومان mode را 'r' قرار دهید فایل برای خواندن باز می شود و نمی توانید در آن بنویسید، اگر این آرگومان را 'w' قرار دهید فایل تنها برای نوشتن باز می شود و نمی توانید محتویات داخل آن را بخوانید. اگر هم آن را 'a' قرار دهید می توانید بدون دستکاریِ داده های قبلی در فایل شروع به نوشتن در فایل کنید. و نیز اگر این آرگومان 'b' را بگیرد در آن صورت برای حالت باینری آماده می شود. برای باز کردن و خواندن فایل باینری از 'wb' و برای باز کردن و نوشتن فایل باینری از 'rb' می توان استفاده کرد.

اما آرگومان کیوُرد encoding چکار کنیم؟ اگر بخواهید برای خواندن یا نوشتن فایلهایتان از سیستمی به غیر از ansi استفاده کنید (مثلا برای خواندن فایلهای متنی که تمام یا قسمتی از متن حاوی متون فارسی یا غیر انگلیسی است) باید این آرگومان را مقداری به جز None بدهید. برای مثال برای متون فارسی و عربی و ... از مقدار 'utf-8' استفاده کنید. به مثال توجه نمایید:

```
>>> newfile=open('names.txt', 'w', encoding='utf-8')
```

توجه کنید که اگر حالت فایل را نوشتن بگذارید و اگر در مسیر معلوم شده چنین فایلی وجود نداشته باشد فایلی با نامِ مشخص شده ایجاد می شود ولی اگر فایلی را بخواهید برای خواندن باز کنید و آن فایل در مسیری که مشخص می کنید وجود نداشته باشد یک استثنای IOError ایجاد می شود. حالا برای نوشتن داخل همین فایلی که در مثال بالا باز کرده ایم باید اینگونه عمل کنیم:

```
>>> newfile=open('names.txt', 'w', encoding='utf-8')
>>> newfile.write('نام این کتاب پایتونیک است')
25
>>> newfile.write('\n')
1
>>> newfile.write('The name of this book is Pythonic')
33
>>> newfile.close()
>>> newfile=open('names.txt', 'r')
>>> newfile.read()
'ظ‍ط³ط§ط ©ع†ظ†ظ^ظ‍ظ†ظ‍ط§ظ¾ظ ¨ط§ظ‍ظ‍ط©ع †ظ†ظ§ط …ط§ظ†\nThe name of
```

در مثال بالا در دستور پنجم ما بعد از اینکه کارمان با فایل تمام شده و از آن استفاده کرده ایم، فایل را بستیم. سپس یک بار هم برای خواندن فایل را باز کردیم. همانگونه که می بینید متن فارسی که در فایل ذخیره شده است به خوبی نشان داده نشده این به این دلیل است که ما هنگام باز کردن فایل encoding را مشخص نکردیم. شکل صحیح به صورت زیر خواهد بود:

```
>>> newfile=open('names.txt', 'r', encoding='utf-8')
>>> newfile.read()
'نام این کتاب پایتونیک است\nThe name of this book is Pythonic'
```

در اینجا برای اینکه یک مثال کامل تر داشته باشیم از فایلی استفاده خواهیم کرد که در سیستم شما هم موجود است. به پوشه پایتون بروید همان پوشه ای که پایتون در آن نصب شده است. برای مثال اگر شما از پایتون ۳.۳ استفاده می کنید پایتون شما در پوشه python33 و معمولا در درایو ویندوزتان نصب شده است. در این پوشه فایل license.txt را مشاهده می کنید. در پایتون شل آن را برای خواندن باز کنید:

```
>>> newfile=open('license.txt', 'r')
>>> newfile.read()
```

اگر از متد read() روی این فایل استفاده کنید خواهید دید که به یکباره تمام محتویات فایل چاپ می شود. اما بجای اینکار می توانید از متد readlines() استفاده کنید، این تابع خط به خط محتویات فایل را می خواند و بعد از خواندن هر خط به خط بعدی می رود.

```
>>> newfile.readline()
'A. HISTORY OF THE SOFTWARE\n'
>>> newfile.readline()
'==========================\n'
>>> newfile.tell()
56
```

اما متد دیگری هم وجود دارد که فایل را خط به خط می خواند و محتویات آن خط به خط داخل یک لیست ریخته و بر می گرداند به صورتی که هر خط یکی از عناصر لیست باشد. این متد readlines() است. از این متد می توانید در حلقه ها استفاده کنید برای مثال:

```
>>> for lines in newfile.readlines():
        pass
```

این تنها روش برای گرفتن خط به خط محتویاتِ داخل یک فایل نیست بلکه قطعه کد زیر نیز کارساز است:

```
>>> for line in open('license.txt', 'r'):
        print(line)
```

دلیل اینکه این کد کار می کند این است که یک شئ فایل می تواند مانند یک دنباله (برای مثال یک لیست) iterate گردد، به این صورت که هر آیتم یک رشته است که حاوی خط بعدی که در متن است می باشد. توجه نمایید که پایتون با رشته ی '\n' است که انتهای خط را می فهمد، اگر این رشته write() کردن داخل فایل را در متن بنویسید پایتون متوجه می شود که باید در آن نقطه خاص پایان خط به متن اضافه کند، به این صورت دیگر فرقی نخواهد کرد که این قطعه کد را در سیستم عامل لینوکس استفاده کنید یا ویندوز نتیجه ای که هنگام خواندن فایل متنی می بینید یکسان است.

انواع Mapping

یک نوع داده ی mapping نوعی است که از عملگر in و تابع len() پشتیبانی می کند و همچنین iterable است. انواع داده ی mapping در واقع collection هایی از آیتم هایی به صورت key-value یا همان کلید- مقدار می باشند. هنگامی که این انواع داده iterate می شوند (برای مثال بوسیله ئ تابع داخلی iter()) انواع داده ی

mapping که به صورت نامنظم هستند برای مثال dict آیتم های خود را در یک ترتیب دلخواه آماده می کنند که قابل پیش بینی نیست.

تنها اشیای hashable (و immutable) می توانند به عنوان کلید ها (keys) در یک دیکشنری استفاده گردند، این به این معنی است که انواع داده ای مانند float, int, str, frozenset, tuple می توانند به عنوان کلید ها انتخاب شوند در حالی که انواعی مانند list, set, dict نمیتوانند به عنوان key مورد استفاده قرار گیرند. تمام این قضایا مربوط به کلید های دیکشنری است در صورتی که value ها می توانند دارای هر مقداری از هر نوع داده ای باشند و محدودیتی وجود ندارد. مقایسه نمودن دیکشنری ها با یکدیگر مانند مقایسه ی دیگر انواع داده است که تا به حال گفته شد.

دیکشنری ها (dict)

یک دیکشنری نوعی داده ی collection نامنظم است که شامل صفر یا تعداد بیشتری جفت های key–value می باشد که در آن key ها مرجع های شئ هستند و که به اشیایی اشاره دارند که hashable هستند و value ها مراجع شئ ای از اشیایی از هر نوع داده می توانند باشند. دیکشنری ها یا همان نوع داده ی dict، mutable هستند، لذا به راحتی می توانیم آیتم های آنها را از داخل آنها حذف یا به آن اضافه کنیم اما چون به صورت نا منظم هستند در استفاده از آیتم های آنها ما از اندیس نمی توانیم بهره ببریم. لذا نمی توانیم آنها را تکه تکه کنیم (با استفاده از عملگر تکه سازی یا همان [ ]) .

نکته: API مخفف عبارت Application Programming Interface به معنای رابط برنامه نویسی کاربردی است.

نوع داده ی dict می تواند به عنوان یک تابع فراخوانی گردد، بدون هیچ گونه آرگومانی یک دیکشرنی خالی را باز می گرداند با هر نوع داده ی mapping یک دیکشنری بر اساس آرگومان خود برخواهد گرداند. و مانند دیگر انواع داده با آرگومانی از همان نوع یعنی از نوع دیکشنری یک کپی سطحی از آگومان خود را باز می گرداند. در ضمن می توان از یک نوع دنباله ای به عنوان آرگومان این تابع استفاده نمود به صورتی که هر کدام ا ز آیتم های داخل این دنباله خود یک دنباله ی دیگر باشد که در آن دنباله ی داخلی دو شئ موجود باشند که یکی key و دیگری value ما در دیکشنری خواهد بود. برای روشن شدن چگونگیِ این مطلب به مثال زیر دقت کنید:

```
>>> d=dict([['s', 34], [(45,), 'help me']])
>>> d
{'s': 34, (45,): 'help me'}
```

همانطور که می بینید در داخل دنباله ی لیستیِ ما دو آیتم وجود دارند که هر کدام از آنها خود نیز یک دنباله (از نوع لیست) محسوب می شوند. در این لیستهای داخلی اولین آیتم یعنی آیتمی که اندیس شماره صفر را دارد به عنوان کلید یا key و آیتم دوم به عنوان value انتخاب می گردد. همانطور که گفته شد در یک دیکشنری value ها می توانند از هر نوعی باشند اما کلید ها باید از انواع hashable یا immutable باشند. اما روش دیگری هم برای ایجاد دیکشنری ها وجود دارد و آن هم استفاده از { } است. برای آفریدن دیکشنری ها با این روش شما راههای مختلفی پیش رو دارید که در همه آنها باید جفت های key-value را با علامت کلن (,) از یکدیگر جدا نمایید. در زیر برخی از روش های ایجاد دیکشنری ها آورده شده است:

```
>>> d1={'firstname': 'Raman', 'lastname': 'Eshghi', 'age': None}
>>> d1
{'lastname': 'Eshghi', 'age': None, 'firstname': 'Raman'}
>>> d2=dict(firstname='Raman', lastname='Eshghi', age='None')
>>> d2
{'lastname': 'Eshghi', 'age': 'None', 'firstname': 'Raman'}
```

هر دوی این روش ها قابل قبول هستند و توسط برنامه نویسان حرفه ای به وفور استفاده می گردند. توجه نمایید که d2 با استفاده از آرگومان های کیوُرد (keyword argements) آفریده شده است و d1 با کمک گرفتن از { } ایجاد شده.

کلیدها یا همان key ها در دیکشنری ها یکتا هستند، این به این معنا است که اگر بخواهید به یک دیکشنری جفتی از key-value را اضافه کنید که key قبلا در دیکشنری وجود داشته است یک جفت دیگر را به دیکشنری اضافه نمی کند بلکه مقدار جدید value را به کلید قبلی که در دیکشنری بوده است می دهد و در واقع value را با مقدار جدید replace می کند.

```
>>> d
{'s': 34, (45,): 'help me'}
>>> d['s']
34
>>> d[s]
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    d[s]
NameError: name 's' is not defined
>>> d[(45,)]
'help me'
>>> d[(45,)]='do not touch this'
>>> d
{'s': 34, (45,): 'do not touch this'}
```

در دومین دستور مثال های بالا ما مقداری که با کلید 's' جفت است را خواسته ایم که به ما برگردانده شده. در سومین دستور چه؟ در سومین دستور ما از یک مرجع شئ استفاده کرده ایم یعنی این مرجع شئ باید خود به شئ

اشاره داشته باشد که یکی از کلید های دیکشنری است اما می بینید که طبق استثنای NameError هیچ مرجع شئ به این نام تعریف و ساخته نشده است. در دستور پنجم ما می خوام مقدار متناظر با کلید (45,) را تغییر دهیم که به راحتی میسر است. اما می توانیم با دستوری مانند زیر هر کدام از جفت های کلید- مقدار را که بخواهیم از دیکشنری حذف کنیم.

```
>>> del d['s']
>>> d
{(45,): 'do not touch this'}
```

باز هم یاد آوری می کنیم که دیکشنری ها مانند لیست ها منظم نیستند بلکه نامنظم هستند و شما تنها می توانید با استفاده از کلید ها به مقادیر دسترسی پیدا کنید. آیتم ها را همینطور می توانید با استفاده از dict.pop() از دیکشنری حذف کنید. مانند:

```
>>> d1
{'lastname': 'Eshghi', 'age': None, 'firstname': 'Raman'}
>>> d1.pop('lastname')
'Eshghi'
```

همانطور که درمثال دیدید این متد پس از حذف کردن کلیدو مقدار، مقدار متناظر با کلید آرگومان را باز می گرداند. دیکشنری ها تابع داخلیِ len() را نیز پشتیبانی می کنند. همچنین برای کلیدهای موجود در دیکشنری ها می توانید از in و not in به صورت صریح استفاده کنید. شما ممکن است بخواهید کلیدها یا مقادیر موجود در دیکشنری ها را دانه به دانه به صورت جفت یا جدا از هم استخراج کنید در زیر مثال هایی را مشاهده می کنید.

```
>>> for key, value in d2.items():
        print('this is key: ', key)
        print('this is corresponding value: ', value)


this is key:  lastname
this is corresponding value:  Eshghi
this is key:  age
this is corresponding value:  None
this is key:  firstname
this is corresponding value:  Raman
```

این یکی از مثال هایی است که در آن کلید ها و مقادیر را توسط متد dict.items() استخراج کرده ایم. در مثال زیر تنها کلید ها و در مثال بعدی آن تنها مقادیر داخل دیکشنری را استخراج نموده ایم.

```
>>> for key in d2.keys():
        print('current key is: ', key)


current key is:  lastname
current key is:  age
current key is:  firstname
>>> for value in d2.values():
        print('current value is: ', value)


current value is:  Eshghi
current value is:  None
current value is:  Raman
```

به مثال زیر توجه کنید که چگونه می توان آیتم های key–value داخل یک دیکشنری را به یک لیست فرستاد (برای مقاصد احتمالی).

```
>>> my_list=list(d2.items())
>>> my_list
[('lastname', 'Eshghi'), ('age', 'None'), ('firstname', 'Raman')]
>>> list(d2.keys())
['lastname', 'age', 'firstname']
>>> list(d2.values())
['Eshghi', 'None', 'Raman']
```

جدولی که در زیر مشاهده می کنید شامل متدهای موجود در انواع dict در صفحات آتی مثالهایی هم آورده شده اند.

تالیف: رامان عشقی، ۰۹۳۵۷۷۰۵۰۶۵ برای فیلم های آموزش پایتون به http://www.blue-python.tk مراجعه کنید.

ترکیب	کاربرد
d.clear()	تمامی آیتم های موجود در دیکشنری d را پاک می کند (از بین می برد)
d.copy()	یک کپی سطحی از دیکشنری d باز می گرداند.
d.fromkeys(s, v)	یک دیکشنری را برمیگرداند که شامل جفتهایی است که کلیدهای آنها آیتم های داخل دنباله s باشند و مقادیر آنها هم None باشد. یا اگر v داده شده است مقادیرشان v باشد.
d.get(k)	مقدار متناظر با کلید k را بازمیگرداند، اگر k در دیکشنری نباشد None برمی گرداند
d.get(k, v)	مقدار متناظر با کلید k را بازمیگرداند، اگر k در دیکشنری نباشد v را بازمی گرداند.
d.items()	یک view را که شامل جفتهای key-value از دیکشنری d است باز می گرداند
d.keys()	یک view که شامل تمامی کلیدهای موجود در دیکشنری d است را باز می گرداند
d.pop(k)	مقدار متناظر با کلید k را بازمیگرداند و جفت آیتم k را از دیکشنری حذف می کند اگر کلید k در دیکشنری وجود نداشته باشد، یک استثنای KeyError ایجاد می کند
d.pop(k, v)	مقدار متناظر با کلید k را باز می گرداند و جفت آیتم k را از دیکشنری حذف می کند، اگر کلید k در دیکشنری وجود نداشتبه باشد، v را باز می گرداند.
d.popitem()	یک جفت key-value دلبخواهی را از داخل دیکشنری حذف می کند و اگر دیکشنری خالی باشد KeyError ایجاد می کند.
d.update(a)	تمام جفتهای کلید مقدار موجود در a را به داخل دیکشنری d وارد می کند، اگر کلیدی هم در a و هم در d وجود داشته باشد، مقدار موجود در a را به آن در دیکشنری d می دهد. a هم می تواند دیکشنری باشد، هم یک iterable شامل جفتهای key-value و هم آرگومان های کیورد (keyword arguments)
d.values()	یک view از تمامی مقادیر موجود در d را باز می گرداند

به مثال های زیر توجه نمایید:

```
>>> d
{'s': 's'}
>>> d.clear()
>>> d
{}
>>> d.update(a=1, b=2, c=[9, '*', '#', 34e-1])
>>> d
{'a': 1, 'c': [9, '*', '#', 3.4], 'b': 2}
>>> new_d = d.fromkeys(d)
>>> new_d
{'a': None, 'c': None, 'b': None}
>>> new_d=d.fromkeys(d, 'silent')
>>> new_d
{'a': 'silent', 'c': 'silent', 'b': 'silent'}
```

```
>>> new_d.get('a')
'silent'
>>> new_d.get(45)
>>> new_d.get('232')
>>> type(new_d.get('sas'))
<class 'NoneType'>
```

```
>>> a=iter(d.items())
>>> a
<dict_itemiterator object at 0x023E4180>
```

```
>>> next(a) #or we could use a.__next__()
('a', 1)
>>> next(a)
('c', [9, '*', '#', 3.4])
>>> next(a)
('b', 2)
>>> next(a)
Traceback (most recent call last):
  File "<pyshell#37>", line 1, in <module>
    next(a)
StopIteration
>>> #you saw that iterator is exhausted
```

همانطور که در جدول هم گفته شده است، متدهای dict.items() و dict.values() و dict.keys() مقادیر view دیکشنری را باز می گردانند. توجه نمایید:

```
>>> d
{'s': 's'}
>>> type(d.values())
<class 'dict_values'>
```

یک dict view در واقع یک iterable با قابلیت فقط خواندنی است. در حالت کلی ما همانگونه که با iterable ها رفتار می کنیم می توانیم با view ها هم رفتار کنیم. اما با این حال تفاوت هایی وجود دارند که یک view را از یک iterable معمولی متمایز می کنند. اولین و شاید مهمترین تفاوت که یکی از ویژگی های جالب view های دیکشنریها است این است که هنگامی که دیکشنری تغییر کند، view های گرفته شده از آن دیکشنری هم تغییر پیدا خواهند کرد. و تفاوت دیگر این است که view های مربوط به کلید ها و مقادیر دیکشنری ها دارای قابلیت هایی هستند که می توان چند نمونه از اعمالی که روی set ها انجام می شد را روی آنها نیز انجام داد.

تالیف: رامان عشقی، ۰۹۳۵۷۷۰۵۰۶۵ برای فیلم های آموزش پایتون به http://www.blue-python.tk مراجعه کنید.

فصل پنجم:

آنچه باید بدانید

مرجع آموزش پایتون در ایران

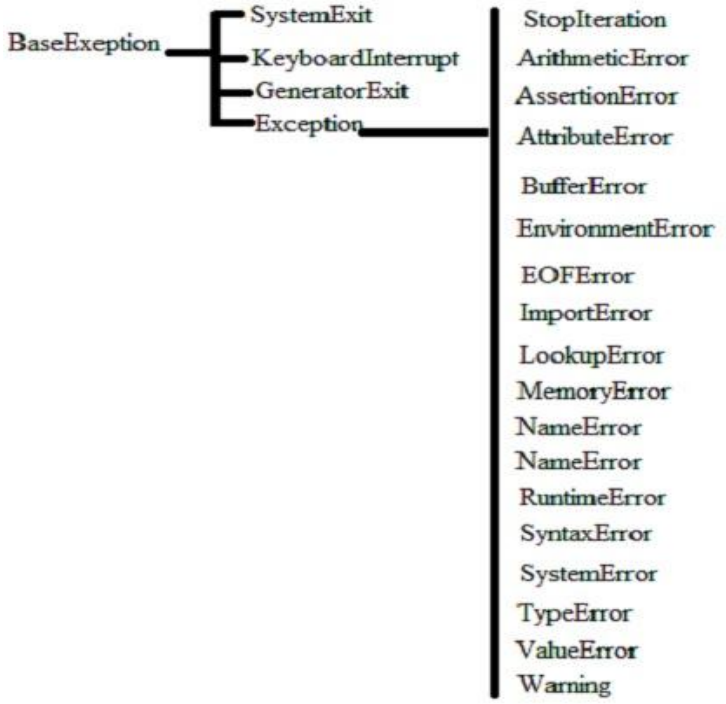
http://www.blue-python.tk

مولف: رامان عشقی

در اولین بخش از این فصل درباره exception های پایتون توضیح خواهیم داد. شما در هر ساعتی از عمر خود که به برنامه نویسی توسط زبان پایتون مشغول هستید امکان دارد ده ها exception متفاوت را مشاهده کنید که هر

کدام می تواند به دلیلی متفاوت از دیگری رخ داده باشد. این حالات خاص می توانند بر روی مسیر برنامه شما اثر گذار باشند و شما در آینده خواهید دید نتنها دست و پاگیر نیستند بلکه بسیار سودمند هم هستند.

به صورت کلی تمامیه exception ها در پایتون instance هایی از کلاس هایی هستند که از BaseException اشتقاق یافته اند. در شکل زیر استثناهای داخلیِ پایتون را مشاهده می کنید.



در اینجا سعی می کنیم تا آنجا که ممکن است این استثناها را برایتان تشریح کنیم.

Exception

تمامیِ استثناهایی که مربوط به خروج غیر سیستمی باشند از این کلاس اشتقاق یافته اند. البته تمامیِ استثناهای تعریف شده بوسیله ی کاربر نیز باید از این کلاس ارث بری داشته باشند.

ArithmeticError

کلاس پایه برای تمامیِ استثناهایی که به هنگام رخ دادن خطاهای مختلف حسابی احضار می شوند. استثناهای فرزند این استثنا در python شامل سه تای زیر هستند:

۱. OverFlowError
۲. ZeroDivisionError
۳. FloatingPointError

در زیر نیز به اختصار درباره ی هر کدام از این استثناها توضیحی داده شده است،

OverflowError

این استثنا هنگامی برمی خیزد که پاسخ یک محاسبه (ریاضی) آنقدر بزرگ باشد که نمی توان آن را نشان داد. این اتفاق برای int ها نخواهد افتاد (اگرچه در مورد آنها معمولاً MemoryError احضار می گردد). این استثنا بیشتر اوقات دامن گیر float ها می گردد به دلیل محدودیت های زبان C (این خود به این دلیل است که پایتون استاندارد به وسیله زبان سی نوشته شده است).

ZeroDivisionError

همانطور که از نام آن پیداست هنگامی رخ می دهد که عددی بر صفر تقسیم گردد.

```
>>> a=3/0
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    a=3/0
ZeroDivisionError: division by zero
```

BufferError

هنگامی که یک عملیات مربوط به بافر با شکست مواجه می شود.

LookupError

هنگامی رخ می دهد که اندیس یا کلیدی که در عملیات مربوط به انواع mapping یا sequence قابل قبول و صحیح نباشد. استثناهای داخلی که از آن ارث بری دارند شامل دو استثنای زیر هستند:

۱. IndexError
۲. KeyError

IndexError

هنگامی که اندیس وارد شده در یک دنباله فاقد اعتبار است، اغلب اوقات به معنی این که اندیس مذکور خارج از محدوده اندیس های دنباله است. توجه کنید که اگر قرار باشد به جای اندیس های عددی در یک نوع داده از نوع دیگری استفاده شود، در آن صورت در شرایط خاص `TypeError` اتفاق خواهد افتاد.

```
>>> a[5]
Traceback (most recent call last):
  File "<pyshell#15>", line 1, in <module>
    a[5]
IndexError: list index out of range
```

KeyError

وقتی که کلید مشخص شده در یک دیکشنری وجود نداشته باشد این استثنا احضار می گردد.

```
>>> my_dict['name']
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    my_dict['name']
KeyError: 'name'
```

EnvironmentError

کلاس پایه برای تمام استثناهایی که می توانند در خارج از سیستم پایتون رخ دهند.

AssertionError

هنگامی رخ می دهد که یک عبارت `assert` با شکست روبرو گردد.

EOFError

هنگامی که با استفاده از تابع داخلیِ `input()` سعی در خواندن یک فایل را دارید اما به پایان فایل برخورد می کنید بدون آنکه هیچ داده ای خوانده شده باشد، این استثنا فراخوانی می گردد.

FloatingPointError

هنگامی که عملیاتی مربوط به `float` ها با شکست مواجه می شود این استثنا فراخوانی می گردد.

تالیف: رامان عشقی، ۰۹۳۵۷۷۰۵۰۶۵ برای فیلم های آموزش پایتون به http://www.blue-python.tk مراجعه کنید.

IOError

هنگامی که یک عملیات ورودی یا خروجی با شکست مواجه می شود شما با این استثنا مواجه خواهید شد.

ImportError

وقتی یک عبارت `import` در هنگام پیدا کردن ماژولی که شما مشخص نموده اید با شکست مواجه می شود.

```
>>> import mybook
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    import mybook
ImportError: No module named mybook
>>> from sys import name
Traceback (most recent call last):
  File "<pyshell#24>", line 1, in <module>
    from sys import name
ImportError: cannot import name name
```

KeyboardInterrupt

هنگامی که کاربر در حین اجرای برنامه (اسکریپت) با استفاده از کلید های وقفه یا همان `Control-C` و یا `Delete` اجرا را متوقف کند همچین استثنایی احضار می شود.

```
>>> import time
>>> time.sleep(3)
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    time.sleep(3)
KeyboardInterrupt
```

MemoryError

در هنگامی که یک عملیات در حال انجام مقدار حافظه اختصاص یافته به خود را از دست می دهد (به پایان می رساند) و برای ادامه اجرا دچار مشکل می شود.

NameError

هنگامی که یک اسم در حوضه `global` یا `local` یافت نشد این اشتثنا ایجاد می شود.

```
>>> print('my name is: ', name)
Traceback (most recent call last):
  File "<pyshell#27>", line 1, in <module>
    print('my name is: ', name)
NameError: name 'name' is not defined
```

TypeError

هنگامی رخ می دهد که یک تابع یا عملیات روی انواع غیر معتبر و نا مناسب اعمال می شود.

```
>>> my_dict[[2, 3]]
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    my_dict[[2, 3]]
TypeError: unhashable type: 'list'
```

ValueError

هنگامی که یک تابع یا عملگر یک مقدار را دریافت می کند که از لحاظ نوع مشکلی ندارد ولی از لحاظ مقداری دچار مشکل است این استثنا احضار می شود.

```
>>> f=float(4)
>>> f
4.0
>>> f2=float('help him')
Traceback (most recent call last):
  File "<pyshell#30>", line 1, in <module>
    f2=float('help him')
ValueError: could not convert string to float: 'help him'
```

بیشتر درباره استثناها

تا به حال با ساختار استفاده ا ز استثناها در برنامه تان و تعدادی از استثناهای built–in پایتون آشنا شدید. اکنون وقت آن رسیده که به دو بحث مهم نیز رسیدگی کنیم که یکی فراخوانی دستیِ استثناها و دیگری ساخت و استفاده از استثناهای دلخواه است.

در ابتدا به فراخوانی استثناهای دلخواه داخلی پایتون می پردازیم، با استفاده از کلمه کلیدی raise می توان استثنای دلخواه را فراخوانی کرد، مانند مثال زیر:

```
try:
    print(int(input('please enter a valid number')))
except ValueError:
    raise ValueError('please enter a valid integer or float')
```

در مثال آتی نیز نحوه تعریف کردن استثناهای دلخواه خودتان آورده شده است. برای درک این مثال باید بخش شئ گرایی را مطالعه نمایید. فلسفه ی تعریف این نوع استثناها در واقع یک کلاس بسیار ساده است که ارث بری مستقیم از یکی از استثناهای داخلی یا کلاس Exception داشته باشد. به مثال توجه نمایید:

```
>>> class MyException(Exception):
        def __init__(self, value):
                self.value = value
        def __str__(self):
                return repr(self.value)
```

در زیر نیز نحوه بکار گیری این استثنای تعریف شده توسط کاربر را می توانید با استفاده از کلمه رزرو شده ی raise مشاهده می کنید:

```
>>> raise MyException(4 ** 3)
Traceback (most recent call last):
  File "<pyshell#50>", line 1, in <module>
    raise MyException(4 ** 3)
MyException: 64
```

توابع (۲)

در بخشهای قبل آموختید چگونه می توان یک تابع خاص خود را تعریف نمود و از آن استفاده کرد. در این قسمت می خواهیم به یک نکته دیگر که بی ربط با توابع هم نیست اشاره کنیم. برای شروع به مثال زیر توجه کنید:

```
>>> a='342'
>>> def func():
        print(a)


>>> func()
342
>>> def func2():
        b='r3s8'
        print(b)


>>> func2()
r3s8
>>> print(b)
Traceback (most recent call last):
  File "<pyshell#41>", line 1, in <module>
    print(b)
NameError: name 'b' is not defined
```

در مثال بالا همانطور که می بینید، با اینکه در درون بدنه ی تابع func2 متغیر b را تعریف کرده بودیم با این حال هنگامی که خواستیم به طور مستقیم از متغیر استفاده کنیم استثنای NameError اتفاق می افتد که نشان می دهد

این نام در این حوضه تعریف نشده است و همچین مرجع شئ در این حوضه موجود نمی باشد. این به این دلیل است که متغیر b در حوضه ی بدنه ی تابع func2() تعریف شده است نه در حوضه بدنه ی ماژول ما.پس نتیجه می گیریم که متغیر هایی که در درون بدنه تابع ها تعریف می شوند محلی هستند یعنی از داخل بدنه ی اصلی ماژول (اسکریپت) قابل فراخوانی نیستند ولی متغیرهایی که در بدنه ی ماژول تعریف شده اند به صورت عمومی و global هستند و می توان هر جایی آنها را استفاده کرد. حالا به مثال زیر توجه نمایید:

```
>>> my_var=34
>>> def k():
        my_var=50


>>> my_var=34
>>> def k():
        my_var=50
        return my_var

>>> k()
50
>>> my_var
34
```

در مثال بالا توجه می کنید با اینکه متغیر my_var را در بدنه ی تابع k() تغییر داده ایم اما در بدنه اصلی برنامه هنوز مقدار پیشین خود را دارد. اگر بخواهید متغیرهایی که در بدنه اصلی تعریف و استفاده می شوند وقتی در بدنه یک تابع تغییر می کنند تغییراتشان به صورت عمومی روی تمام حوضه ها اعمال شود می توانید از کلمه کلیدی global استفاده کنید. نحوه استفاده از این کلمه کلیدی به صورت زیر است:

```
>>> my_var
34
>>> my_var=34
>>> def k():
        global my_var
        my_var=50
        return my_var

>>> k()
50
>>> my_var
50
```

همانطور که مشاهده می کنید متغیر my_var نیز دچار تغییر شده است.

lambda چیست؟

پیش از اینکه به معنای این کلمه رزرو شده ی پایتون بپردازیم و موارد استفاده آن را یاد آور شویم ذکر این نکته ضروری است که گرچه پایتون زبانی است بی نهایت شیئ گرا اما می توان با آن به اسکریپت نویسیِ روال گرا یا تابع گرا نیز پرداخت به همین دلیل برخی از ویژگی های زبان های برنامه نویسیِ تابع گرا نیز در آن وجود دارند که کلمه رزرو شده ئ lambda نیز یکی از همین ویژگی هاست. با استفاده از این کلمه کلیدی شما قادر خواهید بود توابع کوچک و بی نامی را بیافرینید که همانند توابع واقعی قادر به دریافت آرگومان هستند. توابعی که به این روش ساخته می شوند به توابع لامدا معروف هستند. در واقع این توابع از لحاظ ترکیب زبان پایتون تنها عبارات کوچکی هستند که برای رسیدن به هدف خاصی استفاده می شوند. این توابع را معمولا در مکان هایی از برنامه نویسی استفاده خواهید کرد که نیاز به اشیای تابع دارید، اشیای تابع اشیایی هستند که خود از یک تابع دیگر حاصل شده و در خود هم یک عملکرد تابعی را دارند و می توانند آرگومان هایی را دریافت کنند. این اشیا را با توابع معمول اشتباه نگیرید زیرا هر چیزی در پایتون یک شئ است و لذا توابع ساخته شده ئ معمول نیز یک نوع شیئ هستند. به مثال آتی توجه کنید،

```
>>> def make_incrementor(k):
        return lambda x: x + k

>>> f = make_incrementor(20)
>>> f(3)
23
```

در مثال بالا نام تابع make_incrementor به معنای "افزایش دهنده بساز" است. و کار این تابع همین است یعنی مقداری که این تابع بر می گرداند در واقع یک شئ تابعی است که می تواند یک پارامتر دریافت کند و مقدار پارامتر را با مقدار وارد شده به عنوان پارامتر در تابع اصلی جمع بسته و سپس آن را بر گرداند. در خط دوم همانطور که می بینید مقدار return تابع اصلی همان مقداری است که از تابع lambda باز خواهد گشت، در عبارت x : x+k اولین x از سمت چپ نقش آرگومان ورودی را دارد و عبارت x + k نقش بدنه تابع لامدا و در عین حال نقش مقدار return شده را هم دارد. به این صورت هر گاه که تابع اصلی فراخوانده شود شئ را باز می گرداند که خود آن شئ یک نوع تابع ساده (شئ تابعی) است و می تواند یک آرگومان را بگیرد. در دستور دوم پارامتر تابع اصلی ۲۰ است و این باعث می گردد مقدار k بیست شود. و در دستور سوم پارامتر شئ تابعی ۳ بوده و سبب می شود مقدار x سه گردد. در نهایت حاصل جمع این دو مقدار باز گردانده خواهد شد. همانند توابع معمول که باید در فراخوانی آنها تعداد و تقدم نوع آرگومان ها رعایت شود، در فراخوانی اشیای تابعی که از لامدا ساخته شده اند باید این موضوع رعایت گردد. به مثال زیر توجه کنید که می بایست در آن یک آرگومان را به شئ تابعی پاس می دادیم اما آن را به اشتباه بدون آرگومان فرخوانی کردیم و استثنای TypeError رخ داد:

```
>>> f()
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    f()
TypeError: <lambda>() takes exactly 1 argument (0 given)
```

باید این نکته را یاد آور شد که شما برای ساخت اشیای تابعی لامدا محدود به استفاده از آن در درون توابع دیگر نیستید به مثال زیر توجه نمایید:

```
>>> l = lambda name : 'my name is ' + name
>>> l('Raman')
'my name is Raman'
```

استفاده از Docstring

docstring در واقع چیزی نیست جز یک رشته که در ابتدای تعریف تابع نوشته می شود. هدف آن هم ساختن مستندات برای آن تابع است. می توانید در این رشته توضیحاتی را درباره تابع خود ذکر کنید. IDE ها معمولا docstring های داخل توابع را بیرون کشیده و هنگام برنامه نویسی برای راحت تر نمودن دسترسی آنها را هنگام استفاده به نمایش در می آورند. یک نمونه از این مثال را مشاهده می کنید:

```
>>> def return_it():
        """ this will return something,
        but I don't know what!!!"""
        return

>>> return_it()
>>> type(return_it())
<class 'NoneType'>
```

Context Manager (اداره کننده ی کد)

مفهوم context menager از عبارت و بلوک with برای کنترل ورود به یک بلوک خاص و خروج از آن بلوک کد استفاده می کند. ذکر این نکته ضروری است که همچنین می توان از بلوک try نیز برای رسیدن به این هدف استفاده نمود زیرا یکی دیگر از کاربردهای بلوک try به غیر از کنترل به وقوع پیوستن استثناها این است. در واقع از کلمه رزرو شده ی with به این منظور استفاده می شود که اگر در یک قسمت خاص نیاز به بستن تمامی عملیات ها در قسمتی از کد داشته باشیم بتوانیم به راحتی این کار را انجام دهیم. برای مثال به قطعه کد زیر توجه نمایید:

```
>>> myfile=open('dictionary.txt', mode='w')
>>> myfile.write('Blue-Python')
11
>>> myfile.close()
```

همانطور که مشاهده می کنید هدف از این چند دستور ساده ایجاد یک فایل و ذخیره یک سری متن در درون آن است. حال فرض کنید قبل از بستن فایل که در خط آخر این مثال دستور آن را مشاهده می کنید بخواهیم ۵۰۰۰۰۰ (پانصد هزار) متن دیگر را هم در این فایل بنویسیم. روشن است که این کار را باید با استفاده از یک حلقه انجام دهیم مانند زیر:

```
>>> for i in range(0, 500000):
        myfile.write(str(i))
```

همانطور که حدس می زنید ریختن ۵۰۰۰۰۰ متن داخل فایل مدتی طول می کشد. اما اگر در طول ریختن این اطلاعات داخل فایل و در درون این حلقه یک استثنا رخ دهد چه اتفاقی خواهد افتاد؟ مسلما اگر استثنا را مدیریت نکرده باشید باعث توقف و خروج برنامه خواهد شد، به این ترتیب پیش از اینکه فایل در حال نوشته شدن را با استفاده از متد close() ببندید برنامه خاتمه میابد و فرصت بستن فایل از بین خواهد رفت به این ترتیب تمامیِ متونی که تا لحظه بسته شدن برنامه در فایل ریخته شده بودند از دست خواهند رفت زیرا چیزی ذخیره نشده بود و تنها شئ myfile در حافظه بود که متون داخل آن ریخته می شدند که آن هم با بسته شدن برنامه از دسترس ما خارج شده است. یکی از راهکار ها همان طور که احتمالاً حدس زده اید استفاده از بلوک try است تا در هنگام رخ دادن استثناها فایل را ببندیم. اما راهکار بعدی استفاده از عبارت with است. این عبارت به این صورت عمل می کند که در صورت بروز مشکل یا اتمام کار بلوکش فایل را می بندد. برای اینکه بتوانیم از عبارت with بر روی فایل دلخواه مان استفاده کنیم باید از قاعده زیر پیروی کنیم:

```
>>> with open('dictionary.txt', mode='w') as myfile:
        for i in range(0, 500000):
            myfile.write(str(i))
```

در مثال بالا اگر در حین نوشتن فایل استثنایی رخ دهد پیش از بسته شدن برنامه ابتدا فایل بسته می شود تا تغییرات در فایل ذخیره شوند. از این عبارت روی توابع و اشیای پایتون هم می توان استفاده نمود به مثال صفحه ی بعد توجه کنید:

```
>>> def A():
        return 1

>>> def B():
        return 2

>>> with A() as a, B() as b:
        print('yeah!!!')
```

مثال بالا را می توان به صورت زیر نیز نوشت:

```
>>> with A as a:
        with B as b:
                print('yeah!!!')
```

فصل ششم:

ماژول ها و نکات مربوط به آنها

مرجع آموزش پایتون در ایران

تالیف: رامان عشقی، ۰۹۳۵۷۷۰۵۰۶۵ برای فیلم های آموزش پایتون به http://www.blue-python.tk مراجعه کنید.

http://www.blue-python.tk

مولف: رامان عشقی

در بخش های پیشین نیز گفته شده که ماژول های پایتون چیزی به جز فایل هایی با پسوند py. نمی باشند که خود شامل یک سری خواصی هم هستند. در این ماژول ها (module) کلاس ها، توابع، متغیر ها و دیگر عناصر کدنویسی پایتون قرار می گیرند. اگر در گذشته با زبانی مانند زبان جاوا کار کرده باشید حتما می دانید که در جاوا مفهوم دیگری به نام پکیج (package) وجود دارد. در پایتون نیز پکیج ها وجود دارند. در پایتون پکیج یک پوشه ی معمولی است که در آن حتما یک فایل __init__.py قرار گرفته باشد. این فایل در واقع نقش سازنده (constructor) را برای آن پکیج دارد به این ترتیب که هر گاه آن پکیج در کدنویسی وارد می گردد این فایل بلافاصله اجرا می شود. در داخل پوشه ی پکیج به علاوه ماژول __init__.py هر ماژول دلخواه دیگری هم می تواند قرار بگیرد. همین که ماژول در داخل پوشه ی پکیج باشد به طور خود به خود عضو آن پکیج است.

برای درک مفهوم پکیج می توانید وارد دایرکتوری ای که پایتون در آنجا نصب شده است بروید، برای مثال اگر شما از پایتون ۳.۴ استفاده می کنید به فرض اینکه شما از سیستم عامل ویندوز استفاده کنید و ویندوزتان در درایو C: نصب شده باشد، احتمالا پایتون در دایرکتوریِ C:\python34 نصب شده است. به این دایرکتوری رفته به پوشه ی

lib بروید. این پوشه ای است که کتابخانه ی گسترده ی استاندارد و خارجی پایتون در آن قرار دارد. در این پوشه تعداد بسیار زیادی ماژول و پوشه قرار گرفته است، بیشتر این پوشه ها پکیج های پایتون هستند برای مثال به پکیج http وارد شوید که در آن تعدادی ماژول قرار دارد. در این پکیج ماژول های __init__.py, client.py , server.py, cookies.py, cookiejar.py قرار دارند. در کدنویسی می توانید این ماژول ها را به روش زیر وارد کنید:

```
>>> import http.client
>>> from http import client
```

اگر از روش اول برای وارد کردن ماژول دلخواه استفاده کنید، در حین کدنویسی و استفاده از کلاس ها یا توابع یا ... که درون ماژول client قرار گرفته اند مجبور خواهید بود که در هر بار فراخوانی آن را به صورت نقطه گذاری از نام پکیج استفاده کنید:

```
>>> http.client.error()
HTTPException()
```

برای اینکه از شر این نوع فراخوانی خلاص شوید می توانید از دستور دوم استفاده کنید، در این صورت مثال بالا به شکل زیر خواهد شد:

```
>>> client.error()
HTTPException()
```

البته می توانید هر کدام از کلاس ها را نیز به همین روش وارد کنید:

```
>>> from http.client import error
>>> error()
HTTPException()
```

همین طور برای وارد کردن تمامیِ کلاس های موجود در یک ماژول یا تمامیِ ماژول های یک پکیج می توانید از astrisk (علامت ستاره) استفاده نمایید:

```
>>> from http import *
>>> from http.client import *
```

حتی المقدور بهتر است از این روش پرهیز کنید زیرا می تواند باعث name confliction یا در هم ریختگی نام ها در ماژولتان گردد. زیرا ممکن است نام ها و کلاس ها و ماژولهایی که وارد می کنید با متغیر هایی که در ماژول تان ساخته اید هم نام باشند در این صورت به مشکل بر خواهید خورد.

تالیف: رامان عشقی، ۰۹۳۵۷۷۰۵۰۶۵ برای فیلم های آموزش پایتون به http://www.blue-python.tk مراجعه کنید.

اکنون بد نیست که به تشریح برخی از ماژول های کتابخانه استاندارد پایتون بپردازیم. اما بهتر است که شما پیش از این که به کار حرفه ای با ماژول ها و کتابخانه داخلی بپردازید حداقل شئ گرایی را تا سطح متوسط آموخته باشید. لذا توصیه می شود پیش از ادامه این مبحث به فصل آتی (شئ گرایی در پایتون) رفته و آن را مطالعه کنید، سپس بازگشته و مطالعه ادامه این فصل بپردازید.

اگر به توصیه بالا عمل کرده باشید، اکنون وقتی است که باید تعدادی از ماژول های کتابخانه ی استاندارد پایتون بپردازید. کتابخانه استاندارد پایتون یا همان python standard library مجموعه ای عظیم و پرکاربرد از ماژولهایی است که در خیلی از زمینه های برنامه نویسی نیاز های شما را تا حد زیادی برطرف می کنند. لازم به ذکر است اگر دریافتید که کتابخانه استاندارد پایتون قادر به پاسخگویی نیاز شما در زمینه ی خاص نیست با یک جست و جوی ساده در اینترنت احتمالا چندین ماژول پرکاربرد و خوب برای رسیدن به هدف خود پیدا خواهید نمود. تجربه نشاده داده است پایتون می تواند پاسخگوی تمامی نیازهای شما باشد.

یادآوری می کنم که در این بخش قرار نیست تمام نکات مربوط به ماژول ها گفته شود ، بلکه نکات پرکاربرد برنامه نویسی که شما نیاز زیادی برای دانستن آن دارید در اختیار شما قرار می گیرد.

ماژول sys

برخی از محتویات درون ماژول را به صورت همراه مثال تشریح خواهم نمود:

sys.argv یک لیست است، که شامل تمام پارامتر هایی می شود که در هنگام اجرای برنامه به آن پاس داده می شوند. توجه نمایید که اولین اندیس از این لیست حاوی آدرس کاملِ ماژول ما است. (در مثال زیر ما دستورات را از طریق محیط پوسته فرمان اجرا می کنیم، به همین دلیل ماژولی را نمی سازیم و لذا آرگومانی هم به این ماژول پاس داده نشده است، همچنین آدرسی هم در لیست sys.argv قرار نگرفته است).

```
>>> import sys
>>> sys.argv
['']
```

sys.byteorder یک مقدار str است که می تواند یکی از دو مقدار little یا big را داشته باشد که نشانگر byte order سیستم است.

sys.executable یک مقدار str است که آدرس کامل اینترپرتر پایتون که ماژول شما در حال اجرا شدن بوسیله ی آن است را در خود دارد.

```
>>> sys.byteorder
'little'
>>> sys.executable
'C:\\Python32\\pythonw.exe'
```

sys.exit() اجرای این تابع باعث بسته شدن ماژول و اتمام کار آن می شود. انجام این کار به همراه احضار یک استثنای SystemExit انجام می گیرد.

```
>>> sys.exit()
Traceback (most recent call last):
  File "<pyshell#14>", line 1, in <module>
    sys.exit()
SystemExit
```

sys.getwindowsversion() تابعی است که نسخه ی دقیق سیستم عامل ویندوز را باز می گرداند.

```
>>> sys.getwindowsversion()
sys.getwindowsversion(major=6, minor=1, b
uild=7601, platform=2, service_pack='Serv
ice Pack 1')
```

sys.path لیستی از مکانهایی را نشان می دهد که هنگامی وارد کردن ماژول ها این مکانها مورد جست و جو قرار می گیرند. (شما می توانید با قرار دادن ماژول خود در یکی از این مکان ها آن را بوسیله ی اینترپرتر پایتون وارد کنید).

```
>>> sys.path
['C:\\Python32\\Lib\\idlelib', 'C:\\Windo
ws\\system32\\python32.zip', 'C:\\Python3
2\\DLLs', 'C:\\Python32\\lib', 'C:\\Pytho
n32', 'C:\\Python32\\lib\\site-packages',
'C:\\Python32\\lib\\site-packages\\win32'
, 'C:\\Python32\\lib\\site-packages\\win3
2\\lib', 'C:\\Python32\\lib\\site-package
s\\Pythonwin']
```

sys.platform رشته ای است که پلتفرم را برمی گرداند.

```
>>> sys.platform
'win32'
```

sys.version رشته ای است که حاوی اطلاعات کاملی درباره نسخه ی پایتون باز می گرداند.

```
>>> sys.version
'3.2.2 (default, Sep  4 2011, 09:51:08) [
MSC v.1500 32 bit (Intel)]'
```

تالیف: رامان عشقی، ۰۹۳۵۷۷۰۵۰۶۵ برای فیلم های آموزش پایتون به http://www.blue-python.tk مراجعه کنید.

ماژول os

برخی از اعمالی که وابسته به نوع سیستم عامل شما هستند را می توانید از طریق این ماژول انجام دهید.

os.environ[' '] به جای رشته ی تهی ای که در درون علامت های براکت نوشته شده باید متغیر محیطیِ مورد نظر را بنویسید (درون همان رشته). مقداری که برگردانده می شود، همان مقداری است که متغیر محیطی مورد نظر دارد.

```
>>> os.environ['home']
'C:\\Users\\Chita Co'
```

os.getcwd() مسیر دایرکتوری فعلی را باز می گرداند.

os.chdir('dir full path') دایرکتوری فعلی را به مسیری که در رشته ی آرگومان هست تغییر می دهد، توجه کنید که در پایتون به جای \ (بَکسلش خالی) در رشته ها باید از دو علامت بَکسلش در کنار یکدیگر استفاده کنید تا یک بَکسلش به حساب بیایند. همین قاعده را باید در نوشتن آدرس ها هم رعایت نمود.

os.listdir() لیستی را که حاوی کلیه ی محتویات درون دایرکتوری جاری است را باز می گرداند. میتوانید به عنوان آرگومان مسیر دلخواهی را این تابع پاس دهید تا به جای لیست محتویات دایرکتوری جاری، محتویات آن مسیر را بازگرداند. به مثال ها توجه کنید:

```
>>> import os
>>> os.getcwd()
'C:\\'
>>> os.chdir('d:\\wxpython programming')
>>> os.getcwd()
'd:\\wxpython programming'
>>> os.listdir()
['boa-constructor-0.6.1.src.win32.exe', 'Ma
nning.Publications.wxPython.in.Action.Mar.2
006.pdf', 'wxPython2.8-win32-docs-demos-2.8
.12.1.exe', 'wxPython2.8-win32-unicode-2.8.
12.1-py27.exe']
```

os.system('command') می توان دستورات خط فرمان خاص سیستم عامل را اجرا نماید. به جای رشته ی command رشته ای حاویِ دستور مورد نظرتان بنویسید. اگر این تابع صفر بازگرداند یعنی دستور با موفقیت اجرا شده و اگر هر چیزی به جز صفر باز گرداند به معنای عدم موفقیت در اجرای دستور است (ممکن است دستورتان از لحاظ ترکیب مشکل داشته باشد). می توانید مثال زیر را امتحان کنید:

```
>>> os.system('dir && pause')
0
```

تالیف: رامان عشقی، ۰۹۳۵۷۷۰۵۰۶۵ برای فیلم های آموزش پایتون به http://www.blue-python.tk مراجعه کنید.

ماژول time

این ماژول می تواند بخشی از کاربردهای زمانی منطقه ای را برایتان برطرف کند.

time.localtime() زمان محلی را به صورت کامل باز می گرداند.

time.sleep(seconds) به اندازه ی تعداد ثانیه های seconds برنامه را به تعویق انداخته و پس از پایان این مدت اجرا را ادامه می دهد.

```
>>> import time
>>> time.localtime()
time.struct_time(tm_year=2002, tm_mon=1, tm
_mday=7, tm_hour=0, tm_min=24, tm_sec=18, t
m_wday=0, tm_yday=7, tm_isdst=0)
>>> time.timezone
-12600
>>> time.sleep(5)
```

ماژول subprocess

این ماژول کاربردهای زیادی دارد که در اینجا به یکی از آنها اشاره می کنیم.

subprocess.call() تابعی است که برای اجرای فرمان های محلی سیستم عامل می توانید از آن استفاده کنید. برای مثال در ویندوز فرمان های cmd. در فراخوانی آن آرگومان کیبوردِ shell را برابر با True قرار دهید و در جایگاه اولین آرگومان دستور مورد نظر را بنویسید.

```
>>> subprocess.call('call update.exe', shell=True)
1
```

عدد یک بازگردانده شده به این معنی است که فرمان داده شده به درستی اجرا نشده است. اگر فرمان درست اجرا شود عدد صفر باز گردانده خواهد شد.

ماژول zlib

تالیف: رامان عشقی، ۰۹۳۵۷۷۰۵۰۶۵ برای فیلم های آموزش پایتون به http://www.blue-python.tk مراجعه کنید.

از این ماژول می توان برای فشرده سازی داده ها استفاده کرد. از این کاربرد در برنامه نویسی پروتکل های شبکه یا سوکت نویسی استفاده زیادی می شود.

zlib.compress() این تابع یک مقدار رشته ای را که به صورت دیتا است را می گیرد و یک مقدار رشته ای دیتا که فشرده شده است هم باز می گرداند. برای مقادیر دیتا باید در ابتدای آنها b را قرار داد.

zlib.decompress() از این تابع برای خارج کردن همان مقادیر فشرده شده از حالت فشرده استفاده می شود. به مثال توجه کنید:

```
ImportError: No module named tcl
>>> import zlib
>>> compressed = zlib.compress(b'nuqw34vrn2
304nu23n-ufjspeonvt73-4598n23tjwoerjvsmdfjs
;adla;sjdf')
>>> import zlib
>>> raw_data = b'nuqw34vrn2304nu23n-ufjspeo
nvt73-4598n23tjwoerjvsmdfjs;adla;sjdf'
>>> com_data = zlib.compress(raw_data)
>>> decomp_data = zlib.decompress(com_data)
>>> raw_data == decomp_data
True
```

تالیف: رامان عشقی، ۰۹۳۵۷۷۰۵۰۶۵ برای فیلم های آموزش پایتون به http://www.blue-python.tk مراجعه کنید.

فصل هفتم:

شیئ گرایی در پایتون

مرجع آموزش پایتون در ایران

http://www.blue-python.tk

مولف: رامان عشقی

در بسیاری از منابع از شئ گرایی و مفاهیم آن با عنوان پیچیده ترین و دشوارترین حیطه ی برنامه نویسی اشاره شده است و این اغراق تا حدی زیاد بوده که علاقمندان به یادگیری برنامه نویسی را تا حد زیادی دلسرد می کند و از ادامه راه بازمی دارد. همچنین در بسیاری از منابع ذکر شده است که برای درک شئ گرایی در یک زبان خاص مانند سی شارپ شما نیازمند داشتن سطح بالایی از آگاهی در سی پلاس پلاس هستید. این در حالی است که شئ گرایی هم فقط بخشی از فرآیند برنامه نویسی است، گرچه مهمترین بخش آن هم هست. اما یادگیری و بکارگیری آن تا حدی که توصیف شده مشکل نیست حداقل در زبان پایتون که اینطور است. در واقع آن چیز که مهم و گاه دشوار

است بکار بردن صحیح و بجای تمام عناصر کدنویسی در کنار هم در فاصله زمانیِ مناسب برای یک بخش از پروژه است.

به جرأت می توان گفت پایتون در مقایسه با زبان هایی همچون سی پلاس پلاس، سی شارپ و حتی برادرش جاوا، مفاهیم شئ گرایی و راهکارهای object-oriented را با سادگی هر چه تمام تر در اختیار برنامه نویسان قرار داده است. در واقع یکی از دلایل گرایش به این زبان همین سادگی راهکارهای شئ گرا در پایتون است. البته باید این نکته را متذکر شد که سادگی دلیلی بر کارایی کم یا ضعف این زبان نیست. بلکه این سادگی سبب سهولت باور نکردنی در نگه داری و ارتقاء و همینطور توسعه سریع نرم افزارهای نوشته شده به این زبان می گردد، علاوه بر موارد ذکر شده امنیت یکی دیگر از مزیت هایی است که پایتون برای برنامه نویسان علاقمند فراهم کرده است.

اگر تا به این قسمت از کتاب را مطالعه نموده باشید پس با تعداد زیادی از کلاس ها و اشیای پایتون رو برو شده اید. اما اگر تجربه برنامه نویسی شئ گرا نداشته باشید احتمالا حتی نمی دانید تفاوت این دو مفهوم یعنی شئ و کلاس در چه چیزی است. شئ گرایی را می توان راهکاری برای بسته بندی داده ها و متد ها و محدود کردن دسترسی به آنها و همچین گسترش این داده ها دانست.

فرق کلاس با شئ چیست؟ برای روشن نمودن این مطلب می توان از مثال ساختمان استفاده کرد. برای ساختن یک ساختمان شما اول از هر چیز نیاز به یک نقشه دارید. و سپس شروع به ساختن آن خواهید نمود. در این مثال می توان نقشه ساختمان را همان کلاس و خود ساختمان که از روی آن نقشه ساخته شده را شئ دانست. کلاس طرح و نقشه ی اولیه ای است که شئ از روی آن ساخته می شود.

ساده ترین ترکیب برای ساختن کلاس ترکیبی مانند زیر است:

```
class ClassName:
        statement 1
        .
        .
        .
        statement n
```

که در آن به جای ClassName باید نام کلاس دلخواهتان را بنویسید. توجه داشته باشید که نام کلاس را باید با یک حرف بزرگ شروع کنید. اگر نام کلاس باید حاوی چند کلمه باشد برای مثال FileDownloader باید حرف اول تمام کلمات را به صورت uppercase (حروف بزرگ) بنویسید. مثال:

```
>>> class MyClass:
        pass

>>> a = MyClass()
>>> a
<__main__.MyClass object at 0x01FF8CB0>
>>> id(a)
33524912
```

در عبارت اول ما کلاسی به نام MyClass تعریف کردیم، البته در این کلاس فعلا هیچ متغیر کلاس یا متغیر شیئ یا حتی متد خاصی را هم جای ندادیم. در دستور عبارت دوم یک شئ از کلاسمان به نام a ساختیم .

مسلماً برای اینکه بتوانید از یک کلاس استفاده کنید ابتدا باید آن را تعریف کرده باشید یا اینکه اگر در ماژول دیگری قرار دارد آن را وارد کرده باشید. در مثال زیر ما ابتدا ماژول subprocess را که یکی از ماژول های کتابخانه استاندارد پایتون است را وارد نموده ایم و سپس از کلاس Popen که در آن قرار دارد استفاده کرده و شئِ p را آفریده ایم. توجه کنید که حرف اول اسامی اشیا را باید به صورت کوچک یا lowercase بنویسید و اگر از چند کلمه تشکیل شده است از چنین الگویی استفاده کنیم: wordWordWord

```
>>> import subprocess
>>> p = subprocess.Popen('dir /a', shell=True)
>>> p
<subprocess.Popen object at 0x020222B0>
```

ذکر این نکته خالی از فایده نیست که شما می توانید تعریف یک کلاس را داخل بدنه ی یک عبارت if یا حتی داخل بدنه ی یک تابع قرار دهید، به مثال آتی توجه نمایید. در مثال پیش رو ی شما یک تابع تعریف کرده ایم که دارای یک آرگومان از نوع keyword argument است و دارای مقدار دهی اولیه ی True می باشد. در بدنه ی کلاس و در جریان کنترلیِ if اگر پارامتری که به تابع پاس می دهید True باشد (یا ارزش آن هم وزن True باشد)، کلاس ساخته می شود. در داخل کلاس هم یک متغیر ساخته شده، به این نوع متغیرها که به این شکل داخل کلاس تعریف می شوند، "شئ کلاس" گفته می شود. سپس از کاربر خواسته شده تا پاسخ دهد، اگر پاسخ yes بود از همان کلاس ساخته شده یک شئ به نام my_object ساخته می شود و از طریق این شئ به متغیر کلاس name دسترسی می شود.

```
>>> def classMaker(flag=True):
        if flag:
                print('making the class')
                class MyName:
                        name='Raman'
        print('do you want to make an object of the class?')
        answer=input()
        if answer=='yes':
                my_object=MyName()
                print('object made')
                print('your name is', my_object.name)


>>> classMaker()
making the class
do you want to make an object of the class?
yes
object made
your name is Raman
```

اکنون که با مفهموم پایه ایِ کلاس آشنا شدید بهتر است به سراغ متد سازنده ی کلاس برویم. در گذشته گفته شد که در هر پکیج یک ماژول با نام __init__ موجود است که به هنگام وارد کردن پکیج ابتدا محتویات درون این ماژول اجرا می شود. مشابه همین امر در کلاس ها هم دنبال می شود یعنی در هر کلاس هم یک متد __init__ وجود دارد که به آن متد سازنده گفته می شود. دلیل این امر این است که هر کدی که در درون این متد قرار گیرد در هنگامی که می خواهیم شئ ای را از این کلاس بسازیم به صورت خود به خود محتویات درون آن اجرا می شود.

به مثال زیر توجه کنید:

```
>>> class NewC():
        def __init__(self):
                name='author'
                print(name)


>>> first_obj=NewC()
author
```

مشاهده می گردد که بلا فاصله در زمانی که شئ را از کلاس ساختیم دستور print که در بدنه ی متد __init__ قرار دارد اجرا شده است و author را چاپ کرده است. علت اینکه اینجا ما __init__ را متد می نامیم نه تابع در واقع یک تفاوت کوچک است ، این فرق این است که توابع مستقل هستند و در کلاس دیگر قرار نگرفته اند اما متدها آن دسته توابع هستند که به کلاسی دیگر تعلق دارند و در حقیقت از روی یک شئ دیگری فراخوانده می شوند.

نکته : متدهایی که در ابتدا و انتهای آنها__ (double underscore) قرار گرفته است را متدهای خاص (special methods) گویند. تعدادی از این متدها در تمامی کلاسهای پایتون وجود دارند.

اشیای کلاس(class objects)

ممکن است این عبارت کمی گمراه کننده باشد اما در حقیقت به دسته ای از متغیر ها در کلاس گفته می شود که خارج از بدنه ی متدهای دیگر کلاس و به صورت مستقل (و با شکل معمول) در بدنه ی اصلی کلاس تعریف می شوند. برای مثال در کلاس زیر متغیر job یک شئ کلاس است.

```
>>> class MyJob():
        job='python programmer'
```

اشیای کلاس را می توان به صورت بکار برد و استفاده نمود:

۱. instantiation یا همان نمونه سازی
۲. attribute references یا همان مرجع صفت

روش اول یا نمونه سازی روشی است که برای دسترسی و استفاده و حتی تغییر آن شئ کلاس نیازی به ساختن یک شئ از آن کلاس نداریم. تنها کافی است شئ کلاس را از طریق نقطه گذاری پس از نام کلاس استفاده کنید، مانند زیر:

```
>>> class MyJob():
        job='python programmer'


>>> MyJob.job
'python programmer'
>>> #حالا مي خواهيم اين شيئ كلاس را تغيير دهيم
>>> MyJob.job = 'newbie'
>>> #حالا نتيجه كار را امتحان مي كنيم
>>> MyJob.job
'newbie'
```

توجه کنید که اگر در این روش شئ کلاس را تغییر دهید، شئ کلاس دیگر مقدار قبلی خودش را ندارد و مقدار جدید را می گیرد و حتی هنگامی که از کلاس نمونه سازی می کنیم و یک instance از آن می آفرینیم یا به زبان دیگر یک شئ از روی آن می سازیم، شئ کلاس باز هم مقدار جدید خود را دارد. چون در روش نمونه سازی شما با خود کلاس کار می کنید و بر روی آن اعمال تغییر می کنید نه بر روی یک مثال از آن کلاس. درست مثل این است

که شما نقشه ی یک ساختمان را تغییر بدهید به این ترتیب هر ساختمانی که از این پس از روی این نقشه ساخته شود با توجه به نقشه جدید ساخته می شود. دلیل نام گذاری این نوع متغیر ها (شئ کلاس) هم همین است که می توان آنها را بدون نمونه سازی و ساختن شئ از کلاس استفاده کرد. درست مثل اینکه آنها خود شئ مستقل هستند (که هستند، زیرا در پایتون همه چیز شئ است!!!). مثال زیر گویای این نکته است که با دستکاری شئِ کلاس از روش مرجعِ صفت خود کلاس دچار تغییر شده است و دیگر تمام نمونه هایی که از این کلاس ساخته می شوند از این مقدار جدید را روی شئِ کلاسِ job اعمال می کنند.

```
>>> new_obj=MyJob()
>>> new_obj.job
'newbie'
```

روش دوم دستیابی و استفاده از شئ کلاس، روش مرجعِ صفت یا attribute reference است. در این روش شما ابتدا باید یک شئ یا همان instance (مثال) از روی کلاس بسازید و سپس شروع به استفاده و دستکاری شئ کلاسها از روی آن شئ یا instance کنید. در این روش هر اتفاقی هم که برای شئ کلاس بیفتد بر کلاس اصلی بی اثر خواهد بود زیرا شئ کلاس در این روش از روی شئ یا نمونه استفاده می شود نه از خود کلاس بنابراین شئ یا نمونه ای که از کلاس ساخته ایم تغییر می کند ولی کلاس بدون تغییر می ماند. درست مثل اینکه ساختمانی را از روی یک نقشه بسازیم و سپس بدون اینکه نقشه را دستکاری کنیم به اعمال تغییرات در ساختمانمان بپردازیم در این حالت تغییرات ما روی ساختمان های بعدی که از این نقشه ساخته می شوند اعمال نخواهد شد و تنها ساختمان حاضر را شامل می شود. به مثال توجه کنید:

```
>>> second_obj=MyJob()
>>> second_obj.job
'python programmer'
>>> second_obj.job = second_obj.job.upper()
>>> second_obj.job
'PYTHON PROGRAMMER'
>>> MyJob.job
'python programmer'
```

همانطور که در خط آخر می بینید شئِ کلاس که متعلق به کلاس اصلی است و نه شئ (نمونه ی) ساخته شده دچار تغییر نشده است.

اشیای نمونه (instance objects)

یکی دیگر از انواع متغیر هایی که در درون کلاس ها استفاده می شود، اشیای نمونه یا instance objects هستند. دلیل نام گذاری این نوع از متغیر ها این است که آنها را فقط می توان از روی شئ یا نمونه ی ساخته شده از روی

کلاس استفاده و دستکاری کرد، نه به صورت مستقیم توسط خود کلاس. به عبارت دیگر تنها از یک روش می توان به آنها دسترسی داشت و آن هم روش مرجعِ صفت (attribute reference) است. این نوع از متغیرها را باید داخل متد های تعریف شده در داخل کلاسمان تعریف کنیم و در تعریف و استفاده از آن ها در داخل کلاس از کلمه ی رزرو شده ی self باید استفاده نمود. در سازنده ی کلاس باید کلمه ی self را به عنوان اولین آرگومان بکار برد. به مثال توجه کنید:

```
>>> class MyHair():
        def __init__(self):
                self.hair = 'brown'
```

self.hair در مثال بالا یک شئ نمونه است . همانطور که گفته شد شئ نمونه را نمی توان از طریق instantiation استفاده کرد بلکه تنها باید از روش attribute reference استفاده نمود. مانند زیر:

```
>>> class MyHair():
        def __init__(self):
                self.hair = 'brown'


>>> a_hair = MyHair()
>>> a_hair.hair
'brown'
```

اکنون در مثال های زیر مقایسه ای بین دو شئ کلاس و شئ نمونه را خواهیم داشت:

```
>>> class MyClass(object):
        name = 'Raman Eshghi'
        def __init__(self):
                self.myName='Raman Eshghi'


>>> MyClass.name
'Raman Eshghi'
>>> MyClass.myName
Traceback (most recent call last):
  File "<pyshell#42>", line 1, in <module>
    MyClass.myName
AttributeError: type object 'MyClass' has no attribute 'myName'
>>> my_obj=MyClass()
>>> my_obj.myName
'Raman Eshghi'
>>> my_obj.name
'Raman Eshghi'
```

همانگونه که مشاهده می کنید در مثال بالا هنگامی که خواستیم به شئِ نمونه ی myName به روش instantiation از کلاس MyClass دسترسی پیدا کنیم، با AttributeError مواجه شدیم. چون نمی توان به اشیای نمونه به این صورت دسترسی پیدا کرد.

تالیف: رامان عشقی، ۰۹۳۵۷۷۰۵۰۶۵ برای فیلم های آموزش پایتون به http://www.blue-python.tk مراجعه کنید.

متدها

متدها را در شئ گرایی می توان از جهات مختلف مورد تجزیه و تحلیل قرار داد. در این قسمت به تحلیل روش های دسترسی به متدها خواهیم پرداخت. در این باره باید گفت که متدهایی که در کلاس با self تعریف می شوند (مشابه متغیرهایی که آنها را شئ نمونه می نامیدیم) تنها با روشی مشابه روش attribute reference قابل دسترسی هستند. این در صورتی است که متدهایی که بدون self تعریف می گردند، فقط و فقط می توانند با روشی مشابه روش instantiation فراخوانی و استفاده شوند. به منظور درک مطالب گفته شده به مثال ها توجه کنید:

```
>>> class MyNewClass():
        def print_it():
                print('Blue-Python Corporation')
                print('Raman Eshghi')
        def __init__(self):
                self.name = 'Raman Eshghi'
                self.language = 'farsi'
        def getLanguage(self):
                return self.language


>>> MyNewClass.print_it()
Blue-Python Corporation
Raman Eshghi
>>> MyNewClass.getLanguage()
Traceback (most recent call last):
  File "<pyshell#25>", line 1, in <module>
    MyNewClass.getLanguage()
TypeError: getLanguage() takes exactly 1 argument (0 given)
>>>
>>> #حالا می خواهیم یک شیئ از کلاس بسازیم و آن را بیازماییم
```

```
>>> third_obj = MyNewClass()
>>> third_obj.getLanguage()
'farsi'
>>> third_obj.print_it()
Traceback (most recent call last):
  File "<pyshell#30>", line 1, in <module>
    third_obj.print_it()
TypeError: print_it() takes no arguments (1 given)
```

وراثت (inheritance)

وراثت را می توان مهمترین بخش از شئ گرایی ،و شئ گرایی را مهمترین بخش برنامه نویسی دانست. با توجه به عبارت اخیر می توانید اهمیت وراثت را درک کنید. وراثت قابلیت استفاده از کلاس های دیگر (که از پیش ساخته و تعریف شده اند) در کلاس جدید است. به این شکل که بتوانیم از متدها و کلیه ی صفات کلاس دیگری در کلاس خود استفاده کنیم. در عین حال که خود کلاس اصلی را دستخوش تغییر نکنیم بتوانیم کلاسی تعدادی از صفات و متدها و ... که از کلاس اصلی برای کلاس ما به ارث رسیده است را دچار تغییر کنیم تا بتوانیم آنچه می خواهیم را بدست آوریم. به این ویژگی ها وراثت گفته می شود. به کلاسی که می خواهیم از اجزای تشکیل دهنده ی آن در کلاسمان استفاده کنیم کلاس مادر یا پایه و یا والد (parent) یا همان سوپر (super) گفته می شود. و آن کلاسی که در حال اجرای وراثت بر روی آن هستیم یعنی خصیصه های کلاس والد را ارث می برد، کلاس فرزند (child) یا کلاس مشتق شده گفته می شود. اگر کلاسها را نقشه های ساختمان بدانیم، می توان هر تعداد از آنها را که بخواهیم در یک نقشه ی جدید به ارث بریم به این معنا که نقشه ی جدید یک فرزند (child) باشد که از تمامیِ نقشه های پیشین ما ارث می برد و تمامی خصیصه ها و متدهای آنها را می تواند استفاده کند.

وراثت باعث می شود شما بتوانید بدون دستکاری کلاس والد از تمامی امکانات درون آن استفاده کنید و در عین حال می توانید کدهای آن را در داخل فرزند بازنویسی (override) کنید تا به آن چیزی که دلتان می خواهد برسید. شما در ساختن یک کلاس می توانید از هر تعداد کلاس دیگر ارث ببرید. همچینین کلاس فرزند شما می تواند برای یک کلاس دیگر کلاس جدیدتر کلاس والد باشد یعنی محدودیتی برای کلاسهای فرزند وجود ندارد و خود هم می توانند کلاس والد یک کلاس دیگر بشوند.

در زیر ترکیبی برای ساخت کلاس مشتق شده از کلاس دیگر را می بینید:

```
class YourClassName(BaseClassName):
        statement 1
        .
        .
        .
        statement n
```

به مثال زیر که نمونه هایی از ارث بری را نشان داده است توجه کنید:

```
>>> class MyNewClass():
        def print_it():
                print('Blue-Python Corporation')
                print('Raman Eshghi')
        def __init__(self):
                self.name = 'Raman Eshghi'
                self.language = 'farsi'
        def getLanguage(self):
                return self.language

>>> class DrivedClass(MyNewClass):
        def setLanguage(self):
                self.language = 'English'


>>> a = DrivedClass()
>>> a.getLanguage()
'farsi'
>>> a.setLanguage()
>>> a.getLanguage()
'English'
```

فرض کنید بخواهید بدانید که آیا یک کلاس از کلاس دیگری ارث بری دارد یا نه، یا اینکه بدانید که یک شئ خاص نمونه ای از کلاسی خاص هست یا نه؟ برای حل این مشکلات و جواب دادن به این سوالات شما باید از تابع داخلی isinstance( ) استفاده نمایید. این تابع دو پارامتر می پذیرد که اولین پارامتر باید یک کلاس یا شئ و پارامتر دوم باید یک کلاس (و یا در شرایط خاصی شئ باشد، که توصیه نمی شود) . اگر شئ یا کلاسی که در آرگومان اول وارد شده فرزندی از کلاسی که در آرگومان دوم است باشد، تابع مقدار True و گرنه مقدار False را باز خواهد گرداند. به مثال ها توجه کنید:

```
>>> a = DrivedClass()
>>> isinstance(a, DrivedClass)
True
>>> isinstance(a, MyNewClass)
True
>>> isinstance(MyNewClass, DrivedClass)
False
>>> isinstance(MyNewClass, object)
True
>>> isinstance(DrivedClass, MyNewClass)
False
>>> isinstance(int, object)
True
>>> isinstance(bool, int)
False
>>> isinstance(a, str)
False
>>> isinstance(list, str)
False
```

این یک شروع برای شما و یک پایان برای کتاب پایتون آبی است....

برای دریافت راهنمایی و مشاوره درباره برنامه نویسی پایتون می توانید با من (۰۹۳۵۷۷۰۵۰۶۵) تماس بگیرید. از کمک کردن دریغ نخواهم نمود.