

پایتون از روی مستندات در اینترنت

آموزش مقدماتی پایتون



فهرست

6.....	مقدمه
7.....	فصل 1
7.....	معرفی و ساختار
7.....	روشهای اجرای کد پایتون
8.....	نصب پایتون
9.....	نصب پایتون در ویندوز:
10.....	نصب پایتون در MAC OS
11.....	نصب پایتون در سیستم‌های خانواده لینوکس
12.....	دستور نحو زبان Python
12.....	اولین برنامه پایتون
17.....	عملگرها در python
17.....	عملگرهاي محاسباتي
17.....	عملگرهاي انتسابي
17.....	عملگرهاي منطقی
18.....	عملگرهاي بيتی
18.....	عملگرهاي مقایسه
18.....	متغیرها در python
19.....	قواعد نام‌گذاري متغیرها در پایتون
20.....	داده‌های استاندارد در پایتون
20.....	دیکشنری (Dictionaries)
22.....	لیست (List)
24.....	لیست به عنوان پشته (stack)
24.....	لیست به عنوان صف (queue)



25.....	تایپ (Tuples)
26.....	رشته (String)
27.....	Sets
28.....	ساختارهای کنترلی
28.....	ساختار شرطی
29.....	ساختار تکرار (حلقه)
32.....	توابع
32.....	تعريف تابع
33.....	آزمودن مقادیر ورودی توابع
34.....	توابع درون تابع
34.....	تعريف متغیرها در توابع
37.....	توابع از قبل طراحی شده (Built-in function)
59.....	توابع lambada
62.....	دکوراتورها
64.....	ماژول‌ها
65.....	اطلاعات بیشتر در مورد ماژول‌ها
70.....	ورودی‌ها و خروجی‌ها در پایتون
73.....	کار کردن با فایل‌ها
77.....	Json
77.....	Json چیست؟
77.....	نوع‌های داده‌ای، دستور زبان و نمونه
78.....	Json در پایتون
81.....	قالب‌بندی خروجی
84.....	PIP



85.....	نصب PIP
86.....	دستورات PIP
90.....	فصل 2
90.....	شیء‌گرایی Object oriented
90.....	ارثبری (Inheritance)
90.....	مخفى سازی (Encapsulation)
91.....	چندريختى (Polymorphism)
91.....	شیء‌گرایی در پایتون
92.....	کلاس
93.....	ارثبری در پایتون
96.....	تابع و متغیرهای خصوصی
99.....	تکرارشوندها iterators
101.....	ژنراتور generator
104.....	متدهای داندر یا متدهای جادویی و بازنویسی اپراتورها
108.....	Customizing instance and subclass checks
108.....	Emulating container types
110.....	فصل 3
110.....	مدیریت استثناءها و خطاهای
110.....	خطاهای نحوی (syntax error)
110.....	خطای زمان اجرا (run time error)
110.....	خطای معنایی (semantic error)
111.....	مدیریت خطاهای
117.....	فصل 4
117.....	کار با پایگاه داده



117.....	تعريف
118.....	Mysql
123.....	MongoDb
135.....	فصل 5
135.....	عبارت باقاعدہ Regular Expression
135.....	کاراکترهای انطباقی (Matching Characters)
146.....	فصل 6
146.....	چند نخی Multi Threading
150.....	شیء Thread
151.....	شیء Lock
152.....	شیء Rlock
152.....	شیء Condition
152.....	شیء Semaphore
153.....	شیء Event
153.....	شیء Timer
155.....	فصل 7
155.....	برنامه‌نویسی شبکه Network Programming
156.....	مدل OSI
158.....	مدل TCP/IP
161.....	تفاوت‌های بین لایه‌های OSI و TCP/IP
163.....	سوکت در پایتون
163.....	سوکتهای رشته‌ای (Stram Socket)
164.....	سوکت دیتاگرام (Datagram)
168.....	شیء سوکت (Socket)



173.....	برنامه‌نویسی در اینترنت با پایتون.....
176.....	فصل 8.....
176.....	الگوهای طراحی در پایتون.....
176.....	الگوهای ایجادی.....
177.....	Singleton.....
179.....	Abstract factory.....
184.....	Builder.....
188.....	Factory.....
192.....	Prototype.....
194.....	الگوهای ساختاری.....
194.....	Adapter.....
196.....	Bridge.....
199.....	Decorator.....
202.....	Facade.....
205.....	Proxy.....
206.....	الگوهای رفتاری.....
207.....	Chain of responsibility.....
209.....	Command.....
213.....	Interpreter.....
214.....	Iterator.....
216.....	Mediator.....
218.....	Memento.....
220.....	Observer.....
223.....	State.....
226.....	Strategy.....
228.....	Template method.....
230.....	Visitor.....



مقدمه

کتابی که در مقابل شماست حاصل تلاش چندماهه من در سال 1394 برای یادگیری زبان برنامه‌نویسی پایتون از روی مستندات سایت اصلی پایتون است که هرگز منتشر نگردید. در سال 1399 تصادفاً دوباره این کتاب را بین فایل‌های قدیمی خودم پیدا کردم و تصمیم گرفتم تا آن را برای پایتون ۳ بروزرسانی کرده و منتشر کنم. شایان ذکر است که کتاب فوق ترجمه انحصاری هیچ سایت یا کتابی بهصورت مستقل نیست و محتویات آن از بخش‌های مختلفی از اینترنت جمع‌آوری گردیده است.



فصل 1

معرفی و ساختار

پایتون یک زبان همه منظوره سطح بالاست که باهدف هرچه خواناتر کردن کدهای نوشته شده توسط برنامه نویسان طراحی گردیده. در طراحی پایتون دو خصیصه قدرت و خوانایی کد کاملاً در نظر گرفته شده است به این صورت که در عین خوانایی شما به کتابخانه های بزرگ و فراگیری دسترسی خواهد داشت. همچنین پایتون تمام مدل های برنامه نویسی از قبیل شئ گرایی برنامه نویسی دستوری و برنامه نویسی تابع محور را پشتیبانی می کند، کامپایلر پایتون برای معرفی متغیرها از یک سیستم پویا پشتیبانی می نماید و مدیریت حافظه را نیز به صورت پویا بر عهده گرفته، به این صورت که در چرخه هایی مشخص اقدام به زباله روبی (Garbage Collection) می نماید. یکی دیگر از ویژگی های مهم پایتون تفکیک پذیری نام متغیرها به صورت پویا است (dynamic name) که به برنامه این اجازه را می دهد که متدها را به نام متغیرها در طول اجرای برنامه مرتبط نماید. در حقیقت این زبان طوری طراحی گردیده تا هر برنامه نویس تازه کار یا باتجربه ای بتواند آن را درک کند، از قابلیت های آن استفاده کند و از کد زدن با آن لذت ببرد. همچنین این زبان به صورتی طراحی گردیده که وابسته به سکو (Platform) خاصی نیست؛ بنابراین برنامه نویس می تواند برنامه خود را در سیستم عامل یونیکس بنویسد و با اندکی تغییرات آن را بر روی سیستم عامل های دیگری از قبیل ویندوز، لینوکس یا مکینتاش اجرا نماید. زبان پایتون یک زبان Case Sensitive است این بدان معناست که مفسر پایتون به کوچکی و بزرگی حروف حساس است همچنین برای نمایش توضیحات (Comment) از علامت # برای هر سطر استفاده می شود.

پایتون یک پروژه باز متن قابل توسعه است که توسط بنیاد نرم افزارهای باز متن مدیریت می گردد.

روش های اجرای کد پایتون

در حالت کلی یک برنامه به زبان پایتون به چهار طریق قابل اجرا است.

1. حالت محاوره ای
2. یک ماژول از پایتون
3. یک فایل اسکریپتی
4. متدهای خاص یک سیستم عامل

حالت محاوره ای

این روش ساده ترین روش برای اجرای کدهای پایتون است تنها کافی است تا مفسر خود را اجرا کرده و اقدام به نوشتن کدهای خود کنید و با زدن دکمه Enter نتیجه را مشاهده نمایید اشکال این روش در این است که در



صورت بروز خطا برنامه‌نویس مجبور خواهد بود تا تمام کد را بازنویسی نماید همچنین بعد از اجرا کردن کد عملأً شما کد را از دست می‌دهید و کد شما جایی ذخیره نمی‌شود.

یک مازول از پایتون

هدف ما از نوشتن یک نرمافزار استفاده مجدد از آن روند و برنامه است، پس ما نیاز داریم تا کدهای نوشته شده خود را بارها اجرا نماییم. برای این کار کافی ست تا کدهای خود را در یک فایل با پسوند py ذخیره نمایید و اقدام به اجرای آن از طریق خط فرمان کنید در ادامه بیشتر در مورد اجرای این فایل‌ها توضیح خواهیم داد.

یک فایل اسکریپتی

این نوع فایل حالت اجرایی دارد و مشخصه اصلی آن خط یک این فایل است که شامل آدرس ادامه کدهای این برنامه است توجه کنید که خط یک این برنامه را با توضیحات (Comment) اشتباه نگیرید این طریقه اجرای برنامه در سیستم‌عامل‌های خانواده یونیکس قابل اجرا است و متدالوئرین روشن اجرای کدهای پایتون است.

متدهای خاص یک سیستم‌عامل

سیستم‌عامل‌های مختلف رفتارهای مختلفی در رودررویی با سیستم‌های مختلف دارند برای مثال در سیستم‌عامل ویندوز شما با دو بار کلیک کردن بر روی فایل‌هایی با پسوند py می‌توانید این فایل‌ها را اجرا نمایید البته باید مفسر پایتون از قبل روی سیستم شما نصب شده باشد.

نصب پایتون

اگر شما یکی از استفاده‌کنندگان از اکثر توزیع‌های لینوکس یا سیستم‌های مکینتاش هستید به‌احتمال زیاد مفسر پایتون به صورت پیش‌فرض بر روی سیستم‌عامل شما نصب خواهد بود و شما احتیاجی به نصب و کامپایل سورس آن ندارید اما اگر شما از سیستم‌عامل ویندوز استفاده می‌کنید باید پایتون، ابزار و مازول‌های مربوطه را دانلود و نصب نمایید.

همان‌طور که اشاره شد پایتون قابلیت اجرا بر روی محدوده وسیعی از سکوها (Platforms) از قبیل ویندوز، سیستم‌عامل‌های خانواده مک و تمام سیستم‌عامل‌های خانواده یونیکس و لینوکس را دارا است. با توجه به این خصیصه برنامه نوشته شده با زبان پایتون این توانایی را خواهد داشت که بر روی طیف وسیعی از سیستم‌های عامل اجرا شود؛ بنابراین می‌توان گفت که این زبان یک‌زبان قابل حمل یا portable است (تعریفی که شخصاً علاوه‌ای به آن ندارم).

چه چیزی بیشتر از این احتیاج دارد؟ یک‌زبان سطح بالا با یک کتابخانه قوی که در اکثر سیستم‌های عامل قابل اجراست.



نصب پایتون در ویندوز:

شما در ویندوز دو راه برای نصب مفسر پایتون پیش رو دارید:

شرکت ActiveState یک بسته نرمافزاری شامل مفسر پایتون و همچنین تمام بسته‌هایی که به پایتون این اجرازه را می‌دهد تا به سرویس‌ها و API‌ها و همچنین رجیستری و... ویندوز دسترسی داشته باشد و یک IDE که به یک ویرایشگر کد مجهز است را با نام ActivePython ارائه داده است. شما می‌توانید این بسته نرمافزاری را به صورت رایگان از وبسایت این شرکت با آدرس (<http://www.activestate.com>) دانلود نمایید، اگرچه این بسته نرمافزاری تحت قانون متن باز نیست دانلود و نصب آن رایگان است.

این بسته نرمافزاری دارای IDE است که من شخصاً از آن استفاده می‌کنم و همچنین آن را برای توسعه برنامه‌های طراحی شده با python توصیه می‌نمایم، همچنین تمام مثال‌های موجود در این نوشتار را در ویراستار این بسته نرمافزاری اجرا می‌نمایم. تنها این نکته را در نظر داشته باشید که معمولاً شرکت ActiveState آخرین نسخه از پایتون را ارائه نمی‌دهد و همیشه زمانی بسته خود را به نسخه جدیدتری بروز می‌نماید که نسخه جدیدتری از پایتون ارائه شده باشد بنابراین اگر شما می‌خواهید که از آخرین نسخه این زبان برنامه‌نویسی استفاده کنید باید از روش دوم برای نصب استفاده نمایید.

روش دیگر مفسر رسمی سایت پایتون است که توسط توسعه‌دهنده‌های اصلی آن ایجاد شده و در دسترس همگان قرار گرفته شما می‌توانید آن را از سایت رسمی این مجموعه (<http://www.python.org/download>) دریافت و نصب نمایید.

نصب بسته نرمافزاری ActivePython

1- ابتدا بسته نرمافزاری ActivePython را از آدرس <http://www.activestate.com/Products/ActivePython>

دریافت و اقدام به نصب آن نمایید.

2- اگر از ویندوز‌های 98، 95 یا ME استفاده می‌کنید ابتدا باید Windows Installer 2.0 را بر روی سیستم‌عامل خود نصب نمایید.

3- مراحل نصب را دنبال نمایید تا نصب خاتمه پیدا کند.

4- بعد از خاتمه نصب شما می‌توانید برنامه را از منوی Start و در این آدرس پیدا کنید.

Start->Programs->ActiveState ActivePython x.x.x->PythonWin IDE

خروجی که شما مشاهده می‌نمایید احتمالاً به صورت زیر است

```
ActivePython x.x.x.x (ActiveState Software Inc.) based on Python 2.7
(r27:82500, Aug 23 2010, 17:18:21) [MSC v.1500 32 bit (Intel)] on win32
>>>
```

در زمان این نوشتار (سال 1394) شخصاً از نسخه ActivePython-2.7.0.2 استفاده می‌کنم. (البته توجه داشته باشید که در بازنویسی صورت گرفته کدها را برای اجرا با پایتون 3 تغییر داده ام)



نصب مفسر پایتون از سایت (<http://www.python.org/>)

- آخرین نسخه مفسر پایتون را از آدرس <http://www.python.org/ftp/python/> دریافت کرده.
- فایل Python-2.x.yyy.exe را در کامپیوتر خود اجرا نمایید و مراحل نصب را ادامه دهید.
- اگر شما بر روی سیستم خودتان دسترسی Administrator ندارید می‌توانید در قسمت Advance گزینه Non-Admin Install را انتخاب نمایید.
- بعد از پایان نصب شما می‌توانید در منوی Start از آدرس Start→Programs→Python x.x→IDLE به مفسر پایتون خود دسترسی داشته باشید.

خروجی که مشاهده می‌نمایید احتمالاً به صورت زیر است.

```
Python x.x.x (#49, Oct 2 2003, 20:02:00) [MSC v.1200 32 bit (Intel)] on
win32
Type "copyright", "credits" or "license()" for more information.
*****
Personal firewall software may warn about the connection IDLE
makes to its subprocess using this computer's internal loopback
interface. This connection is not visible on any external
interface and no data is sent to or received from the Internet.
*****
IDLE 1.0
>>>
```

نصب پایتون در MAC OS

در سیستم‌عامل‌های خانواده MAC نیز شما دو گزینه برای نصب پایتون پیش رو دارید. اول اینکه پایتون را نصب کنید و روش دیگر آن است که پایتون را نصب نکنید. در حقیقت سیستم‌عامل‌های این خانواده به طور پیش‌فرض نسخه command line پایتون را بر روی خود نصب دارند. این بدان معناست که اگر شما مشکلی با نسخه نصب شده در سیستم‌عامل خود ندارید می‌توانید از نصب مجدد پایتون صرف‌نظر کنید، البته نسخه مذکور محدودیت‌هایی نیز دارد به عنوان مثال از parser ها در xml پشتیبانی نمی‌کند بنابراین برای استفاده از این توانایی باید نسخه کامل از مفسر را نصب نمایید.

در هر صورت شما همیشه می‌توانید تا آخرین نسخه پایتون را خودتان بر روی کامپیوترا نصب نمایید.

نصب مجدد پایتون

برای این کار شما باید مراحل زیر را دنبال نمایید:

1. دیسک ایمیج MacPython-OSX را از [این آدرس](#) دانلود نمایید.
2. بر روی فایل نصب MacPython-OSX.pkg دو بار کلیک کنید تا اجرا شود.
3. فایل نصب نام کاربری و کلمه عبور شمارا درخواست می‌کند آن‌ها را وارد نمایید و مراحل نصب را ادامه دهید.



4. بعدازینکه مراحل نصب پایان یافت فایل نصب را ببندید به فایل Applications/MacPython-2.3 بروید و فایل PythonIDE را اجرا نمایید.

توجه داشته باشید که زمانی که شما آخرین نسخه پایتون را نصب می‌نمایید نسخه قبلی همچنان بر روی کامپیوتر شما فعال است بنابراین وقتی می‌خواهید کد پایتون را در خط فرمان اجرا نمایید باید نسخه مفسری که کد شمارا تفسیر می‌کند را مشخص نمایید.

نصب پایتون در سیستم‌های خانواده لینوکس همیشه به یاد داشته باشید که در اکثر نسخه‌های لینوکس پایتون به صورت پیش‌فرض نصب است پس دقت کنید تا دوباره کاری نکنید.

اگر شما از سیستم‌عامل‌های خانواده Debian استفاده می‌کنید به سادگی می‌توانید پایتون را بر روی سیستم‌عامل خود نصب کنید تنها کافی است به اینترنت وصل شده، خود را در حالت مدیر قرار دهید و دستور زیر را در خط فرمان اجرا کنید.

```
localhost:~$ su -
Password: [enter your root password]
localhost:~# apt-get install python
```

ولی اگر شما از لینوکس‌های خانواده (POSIX) استفاده می‌نمایید باید از دستورات زیر برای نصب پایتون استفاده نمایید

```
localhost:~$ su -
Password: [enter your root password]
[root@localhost ~]# wget http://python.org/ftp/python/2.3/rpms/redhat-9/python2.3-2.3-5pydotorg.i386
```

با این دستور شما می‌توانید نسخه موردنظر خود را از اینترنت دانلود نمایید برای نصب باید دستور زیر را اجرا نمایید.

```
[root@localhost ~]# rpm -Uvh python2.3-2.3-5pydotorg.i386.rpm
```

دستور بالا نسخه پایتونی را که دانلود نموده‌اید بر روی سیستم شما نصب می‌نماید.
حال می‌توانید در خط فرمان دستور زیر را اجرا نمایید تا از صحت بارگذاری خود اطمینان حاصل نمایید

```
localhost:~$ python
```



دستور نحو^۱ زبان Python

زبان پایتون شباهت‌های زیادی با پرل، سی و جاوا دارد. با این حال، تفاوت‌های مشخصی بین این زبان‌ها وجود دارد.

اولین برنامه پایتون بگذارید یک برنامه را در حالت‌های مختلف برنامه‌نویسی اجرا کنیم.

برنامه‌نویسی تعاملی

استفاده از مترجم برای اجرای دستورات بدون ارسال یک فایل اسکریپت پایتون مورد استفاده قرار می‌گیرد

```
>>> print("Hello, Python!")
```

متن بالا را در مترجم Python تایپ کنید و کلید Enter را فشار دهید، نتیجه به صورت زیر خواهد شد
Hello, Python!

برنامه‌نویسی حالت اسکریپت

در این روش یک پارامتر به عنوان یک پرونده اسکریپتی برای مترجم ارسال می‌شود. مترجم شروع به اجرای دستورات به صورت خط به خط می‌نماید و تا زمانی که دستورات به پایان برسند ادامه پیدا می‌کند و با پایان دستورات کار مترجم به پایان می‌رسد. برای مثال کد زیر را در یک فایل متنی بنویسید و سپس با نام test.py ذخیره نمایید.

```
Print("Hello, Python!")
```

همچنین فرض می‌کنیم که شما مفسر پایتون را در متغیر PATH تنظیم کرده‌اید. اکنون سعی کنید تا برنامه را به روش زیر اجرا کنید

```
$ python test.py
```

نتیجه زیر حاصل می‌شود

Hello, Python!

بگذارید یک روش دیگر برای اجرای یک اسکریپت Python را امتحان کنیم، در اینجا فایل test.py را به صورت زیر می‌نویسیم.

```
#!/usr/bin/python
```

```
Print("Hello, Python!")
```

اکنون سعی کنید این برنامه را به صورت زیر اجرا کنید

```
$ chmod +x test.py      # به فایل دسترسی قابل اجرا شدن میدهد
$ ./test.py
```

syntax ^۱



این نتیجه حاصل می‌شود

Hello, Python!

شناسه‌ها در پایتون

شناسه در پایتون نامی است که برای شناسایی متغیر، عملگر، کلاس، ماژول یا اشیا دیگر استفاده می‌شود. شناسه در پایتون فقط می‌تواند شامل حروف A تا Z یا a تا z، اعداد و زیرخط^۱ باشد. هرچند حرف اول شناسه را نمی‌تواند با یک عدد آغاز گردد.

در اینجا قراردادهایی که برای نام‌گذاری شناسه‌های پایتون وجود دارد آورده شده است.

نام کلاس با یک حرف بزرگ شروع می‌شود. همه شناسه‌های دیگر با یک حرف کوچک شروع می‌شوند. شروع یک شناسه با یک آندرلاین نشان می‌دهد که شناسه خصوصی^۲ است.

شروع یک شناسه با دو آندرلاین نشان می‌دهد که یک هویت خصوصی (در فصول آینده در این مورد بیشتر توضیح خواهیم داد) است.

اگر شناسه‌ای با دو آندرلاین به پایان برسد، شناسه یک نام ویژه تعریف شده در زبان پایتون است.

کلمات رزرو شده در پایتون

لیست زیر کلمات کلیدی رزرو شده در پایتون را نشان می‌دهد. شما نمی‌توانید از این کلمات به عنوان ثابت یا متغیر یا هر نام شناسه دیگر استفاده کنید. تمام کلمات کلیدی پایتون فقط دارای حروف کوچک است.

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

underline²
Private³



خطوط و تورفتگی‌ها

پایتون هیچ نشانه‌ای برای نشان دادن یک بلوک کد، تعریف کلاس و تعریف عملکرد یا کنترل جریان ندارد و بلوک‌های کد توسط تورفتگی خط مشخص شده است که به طور دقیق اجرا می‌شود. تعداد فاصله‌های داخل دندانه متغیر است، اما کلیه عبارات موجود در این بلوک باید به همان اندازه غوطه‌ور شوند به عنوان مثال

```
if True:  
    print("True")  
else:  
    print("False")
```

باین حال، بلوک زیر خطایی ایجاد می‌کند

```
if True:  
print("Answer")  
print("True")  
else:  
print("Answer")  
print("False")
```

بنابراین، در پایتون تمام خطوط مداوم با تعداد یکسان از فضاهای مسدود می‌شوند. مثال زیر دارای چندین عبارت مختلف است



```
#!/usr/bin/python

import sys

file_name = ""
file_finish = ""
file_text = ""

try:
    # open file stream
    file = open(file_name, "w")
except IOError:
    print ("There was an error writing to", file_name)
    sys.exit()
print ("Enter ''", file_finish)
print ("' When finished")
while file_text != file_finish:
    file_text = input("Enter text: ")
    if file_text == file_finish:
        # close the file
        file.close
        break
    file.write(file_text)
    file.write("\n")
file.close()
file_name = input("Enter filename: ")
if len(file_name) == 0:
    print ("Next time please enter something")
    sys.exit()
try:
    file = open(file_name, "r")
except IOError:
    print ("There was an error reading file")
    sys.exit()
file_text = file.read()
file.close()
print(file_text)
```

دستورات چندخطی

دستورات در پایتون به طور معمول با یک خط جدید پایان می‌یابد. با این وجود، پایتون اجازه می‌دهد تا با استفاده از کاراکتر ادامه خط (.) مشخص می‌کند که خط باید ادامه یابد. به عنوان مثال:

```
total = item_one + \
        item_two + \
        item_three
```

بیانیه‌های موجود در برackets های [] ، { } یا () نیازی به استفاده از کاراکتر ادامه خط ندارند. به عنوان مثال:

```
days = ['Monday', 'Tuesday', 'Wednesday',
        'Thursday', 'Friday']
```

توضیحات در پایتون

علامت هشتگ (#) که داخل یک مقدار رشته‌ای نیست، یک توضیح را شروع می‌کند. همه عبارات بعد از # و تا انتهای خط فیزیکی بخشی از توضیحات هستند و مترجم پایتون آن‌ها را نادیده می‌گیرد.



```
#!/usr/bin/python
# یک خط توضیحات
print ("Hello, Python!") # دومین خط توضیحات
```

نتیجه اجرای کد بالا به صورت زیر خواهد بود

Hello, Python!

همچنین بعد از دستورات پایتون نیز می‌توانید توضیحات را در همان خط تایپ کنید

```
name = "Madisetti" # This is again comment
```

شما همچنین می‌توانید چندین خط را به صورت زیر به نوشتن توضیحات اختصاص دهید

```
# This is a comment.
# This is a comment, too.
# This is a comment, too.
# I said that already.
```

همچنین نوشه‌هایی که در بین کاراکترهای

''' یا ''''

قرار می‌گیرند به عنوان یک توضیح چندخطی در نظر گرفته می‌شود و توسط مفسر پایتون نادیده گرفته می‌شود:

```
'''
This is a multiline
comment.
'''
```

چندین دستور در یک خط

کاراکتر (:) اجازه می‌دهد تا چندین دستور در یک خط با توجه به اینکه هیچ دستوری یک بلاک کد جدید را شروع نکند نوشته شوند. در اینجا یک مثال را مشاهده می‌نمایید

```
import sys; x = 'foo'; sys.stdout.write(x + '\n')
```

آرگومان‌های خط فرمان

بسیاری از برنامه‌ها بعد از اجرا اطلاعات اساسی در مورد نحوه اجرا را به کاربر ارائه می‌دهند. پایتون نیز این امکان را با استفاده از آرگومان h- به کاربرانش ارائه می‌دهد

```
$ python -h
usage: python [option] ... [-c cmd | -m mod | file | -] [arg] ...
Options and arguments (and corresponding environment variables):
-c cmd : program passed in as string (terminates option list)
-d      : debug output from parser (also PYTHONDEBUG=x)
-E      : ignore environment variables (such as PYTHONPATH)
-h      : print this help message and exit
```

[etc.]

همچنین می‌توانید اسکریپت خود را به گونه‌ای برنامه‌ریزی کنید که آرگومان‌های مختلفی را به صورت ورودی پپذیرد (به این مورد در فصل‌های آینده خواهیم پرداخت).



عملگرها در python

عملگر در حقیقت یک نماد برای انجام عملی بر روی یک یا چند متغیر دیگر است که اصطلاحاً آن‌ها را عملوند می‌نامند در حقیقت عملگرها کاراکترهای ویژه‌ای هستند که برای مفسر شان از قبل تعریف شده‌اند مانند عملگر جمع یا تفریق.

عملگرهای محاسباتی

عملگرهای محاسباتی پایتون عبارت‌اند از

توضیح	عملگر
جمع دو عملوند	+
تفریق دو عملوند	-
ضرب دو عملوند	*
تقسیم دو عملوند	/
عملوند اول را به توان عملوند دوم می‌رساند	**
باقیمانده حاصل از تقسیم دو عملوند	%

عملگرهای انتسابی

عملگرهای انتسابی پایتون عبارت‌اند از

مثال	توضیح	عملگر
	عملگر انتساب	=
$a = a + b$	$a += b$	$+=$
$a = a - b$	$a -= b$	$-=$
$a = a * b$	$a *= b$	$*=$
$a = a / b$	$a /= b$	$/=$
$a = a ** b$	$a **= b$	$**=$
$a = a \% b$	$a \% = b$	$\% =$
	عملگر شیفت به چپ	<<
	عملگر شیفت به راست	>>

عملگرهای منطقی

عملگرهای منطقی پایتون عبارت‌اند از



and	Or	not
-----	----	-----

عملگرهای بیتی
عملگرهای بیتی پایتون عبارت اند از

عملگر	توضیح
<<	شیفت به چپ بیتی
>>	شیفت به راست بیتی
&	and بیتی
	or بیتی
^	Xor بیتی
~	Not بیتی

عملگرهای مقایسه
عملگرهای مقایسه‌ای پایتون عبارت اند از

\neq	$=!$	$=$	\geq	\leq	$>$	$<$
--------	------	-----	--------	--------	-----	-----

دو عملگر $=!$ و \neq هر دو به معنی نامساوی بوده و تفاوت منطقی باهم ندارند.
در حالت عادی عملگرهای محاسباتی برای رشته‌ها معنی خاصی ندارند این به این معناست که برنامه‌نویس نمی‌توانید دو رشته را در هم ضرب یا دو رشته را از هم تفیریق نماید ولی بعضی از عملگرهای دارای معنی خاصی درباره رشته‌ها می‌باشند برای مثال عملگر جمع در بین دو رشته به معنی الحق آن دو به یکدیگر است یا عملگر ضرب بعد از یک رشته به معنی تکرار آن رشته به تعداد دفعات عملوند سمت راست که از نوع عددی است به مثال زیر توجه نمایید

```
>>> Fpart = "First Part"
>>> Spart = "Second Part"
>>> Fpart + Spart
'First Part Second Part'
>>> "TEST ! " * 4
'TEST ! TEST ! TEST ! TEST !'
```

متغیرها در python

متغیر نامی است که به یک مقدار داده می‌شود، پایتون نیز مانند زبان‌های دیگر دارای انواع داده‌ای از قبیل عددی، رشته‌ای و... است.



ولی یک تفاوت اساسی بین تعریف متغیر در این زبان با تعریف متغیر در زبان‌هایی مانند C یا JAVA وجود دارد و آن این است که در زبان‌هایی مانند C برای تعریف یک متغیر شما نیاز دارید تا نوع متغیر را مشخص کنید ولی در پایتون شما نیازی به تعریف نوع متغیر ندارید و مفسر پایتون به صورت خودکار نوع متغیر را تشخیص می‌دهد در حقیقت متغیرها در پایتون تنها نامی برای یک شیء هستند و مانند C به یک قسمت از حافظه اشاره نمی‌کنند این خاصیت به شما این امکان را می‌دهد تا نوع متغیر خود را در طول برنامه به کرات تغییر دهید.

تعریف یک متغیر در پایتون با مقداردهی آن آغاز می‌شود برای مثال:

```
>>>message = "This is a Test"
>>>number = 12
>>>pi = 3.14
```

برای مشاهده مقادیر این متغیرها شما می‌توانید از دستور Print استفاده کنید

```
>>> print (message)
This is a Test
>>> print (pi)
3.14
```

همچنین برای تشخیص نوع متغیرها می‌توانید از دستور type استفاده کنید

```
>>>type(pi)
<type 'float'>
>>>type(number)
<type 'int'>
```

همچنین شما می‌توانید با دستور input() یک متغیر را مقداردهی کنید

```
>>>number = input("Enter The Number: ")
```

که با اجرای این دستور مفسر منتظر می‌ماند تا شما یک مقدار را وارد کنید و آن را در متغیر number قرار می‌دهد.

قواعد نام‌گذاری متغیرها در پایتون

در نام‌گذاری متغیرها در پایتون شما می‌توانید از اعداد و حروف استفاده کنید به این شرط که نام متغیر شما با عدد آغاز نشود همچنین استفاده از کarakترهای خاصی مانند:

\$ % ^ / ~! - @ # ...



نیز غیرمجاز است، شما می‌توانید از کاراکتر (_) در هر کجای نام متغیر خود استفاده کنید در ضمن نام متغیر شما نباید یکی از 28 کلمه رزرو شده زبان پایتون باشد.

کلمات رزرو شده در پایتون

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

برای مثال

```
>>>variable = 123
>>>_variable = "This is a Test"
>>>variable1 = 3.14
>>>var_2 = 123456789
```

تمام نام‌گذاری‌های بالا صحیح است ولی نام‌گذاری‌های زیر صحیح نمی‌باشند.

```
>>>1variable = "This is not True"
>>>v@riable = "This is not True"
syntaxError: can't assigne to operator
```

داده‌های استاندارد در پایتون

داده‌های استاندارد در پایتون به ۵ دسته کلی تقسیم می‌شوند که در ادامه به معرفی آنها خواهیم پرداخت.

- دیکشنری (Dictionaries)
 - لیست (List)
 - تاپل (Tuples)
 - رشته (String)
 - sets
- (Dictionaries)

یکی از داده‌های استاندارد در زبان پایتون دیکشنری است که ارتباط بین کلیدها و مقادیر را به صورت نظری به نظری مشخص می‌نماید یک دیکشنری با علامت {} آغاز می‌شود و با علامت {} خاتمه می‌یابد، داده‌ها نیز با علامت (,) از هم جدا می‌شوند همچنین کلیدها در دیکشنری از هر نوعی می‌توانند باشند و با علامت (:) از مقادیر جدا می‌شوند. اگر با ساختار Hashtable در زبان JAVA آشنایی داشته باشید دیکشنری نیز دقیقاً مانند آن عمل می‌کند به مثال‌های زیر توجه نمایید:



```
Info = { 'name':'hassan', 'age':18, 'country':'Iran', 12: 15 }
tel = {'jack': 4098, 'sape': 4139}
```

همچنین برنامهنویس از طریق کلیدها می‌تواند به مقادیر دیکشنری دسترسی داشته باشد.

```
Info = { 'name':'hassan', 'age':18, 'country':'Iran', 12: 15 }
print (Info['name'])
print (Info[12])
print (Info)

>>>
hassan
15
{'country': 'Iran', 'age': 18, 12: 15, 'name': 'hassan'}
```

توجه داشته باشید که اگر یک کلید را چند بار تکرار کنید سمت راستی‌ترین کلید در نظر گرفته می‌شود و کلیدهای دیگر حذف می‌گردند به مثال زیر توجه نمایید:

```
Info = { 'name':'hassan', 'name':18, 'Name':'Shiraz', 'name':'Iran', 12: 15 }
print (Info['name'])
print (Info[12])
print (Info)

>>>
Iran
15
{12:15, 'name': 'Iran', 'Name':'Shiraz'}
```

همچنین برنامهنویس به راحتی می‌تواند یک مقدار را به یک دیکشنری اضافه نماید کافی است کلید موردنظر را در دیکشنری مقداردهی نماید برای حذف یک مقدار از دیکشنری نیز می‌توان از عبارت رزرو شده `del` و برای خالی کردن یک دیکشنری از عبارت رزرو شده `clear` استفاده نمود به مثال‌های زیر توجه نمایید:

```
Info = { 'name': 'hassan ', 'age':18 }
Info['Family'] = 'Hassani'
Info[100] = 'one hundred'
print (Info)
del (Info['age'])
print (Info)
Info.clear()
print (Info)

{'name': 'hassan ', 'Family': 'Hassani', 'age': 18, 100: 'one hundred'}
{'name': 'hassan ', 'Family': 'Hassani', 100: 'one hundred'}
{}
```

در مثال بالا به‌طور کامل با روند حذف و اضافه کردن یک مقدار به دیکشنری کاملاً آشنا شدید دیگر متدهای مهم دیکشنری عبارت‌اند از



مقدار کلیدها را لیست می کند	keys()
مقادیر را لیست می کند	values()
اگر کلید key در دیکشنری موجود باشد مقدار TRUE برمی گرداند	has_key(key)
یک کپی از دیکشنری را برمی گرداند	copy()

به مثال‌های زیر توجه کنید:

```
Info = { 'name': 'hassan ', 'age':18, 'country': 'Iran', 12: 15 }
print (Info.keys())
print (Info.values())
if(Info.has_key('name')):
    print ('This key is exsist')

>>>
['country', 'age', 12, 'name']
['Iran', 18, 15, 'hassan ']
This key is exsist
```

همان‌طور که مشاهده کردید کارکرد توابع بالا در این تکه کد کاملاً ملموس است.

(List) لیست

لیست یکی از پرکاربردترین انواع داده‌ای در پایتون است که می‌توان در آن انواع مختلفی از داده‌ها را ذخیره کرد. لیست‌ها در پایتون دقیقاً مانند آرایه‌ها در PERL هستند همچنین برنامه‌نویس می‌تواند از طریق اندیس‌ها به مقدارهای ذخیره شده در لیست نیز دسترسی داشته باشد به مثال‌های زیر توجه نمایید:

```
List = ['Iran', 333, 333, 1234.5]
print (List)
print (List[3] )
print (List[0] )

>>>
['Iran', 333, 333, 1234.5]
1234.5
Iran
```

همچنین در پایتون لیست‌ها این قابلیت را دارند که به صورت تودر تو تعریف شوند به مثال‌های زیر توجه نمایید:



```
List = ['Iran', [12,13,14,15], [16,17], 1234.5]
print (List)
print (List[2] )
print (List[1][3] )

>>>
['Iran', [12, 13, 14, 15], [16, 17], 1234.5]
[16, 17]
15
```

همان‌طور که مشاهده نمودید ساختار لیست‌ها بسیار ساده است و دستیابی به لیست‌های درون لیست‌ها نیز کاری آسان بود. همچنین برنامه‌نویس این امکان را دارد تا به محدوده‌ای از مقادیر در یک لیست دسترسی داشته باشد به مثال زیر توجه نمایید:

```
List = ['Iran', [12,13,14,15], [16,17], 1234.5]
print (List[:3] )
print (List[2:] )
print (List[1:3] )
print (List[2:1] )

>>>
['Iran', [12, 13, 14, 15], [16, 17]]
[[16, 17], 1234.5]
[[12, 13, 14, 15], [16, 17]]
[]
```

همان‌طور که مشاهده نمودید برای نشان دادن محدوده‌ای از یک لیست کافی است داخل [] ابتدا و انتهای محدوده خود را مشخص و آن‌ها را با علامت (:) از هم جدا نمایید لازم به ذکر است که حالی گذاشتن هر کدام از پارامترهای ابتدا و انتهای به معنای ابتدا یا انتهای لیست است.

همچنین برنامه‌نویس به سادگی می‌تواند با استفاده از متدهای لیست یک مقدار را اضافه یا حذف نماید یا آنکه مقدار یک اندیس را تغییر دهد به مثال‌های زیر توجه نمایید:

```
List = ['Iran', [12,13,14,15], [16,17], 1234.5]
List.append(12)
List.insert(0,'Shiraz')
List.remove([12,13,14,15])
print (List)

>>>
['Shiraz', 'Iran', [16, 17], 1234.5, 12]
```

متدهای مهم لیست در پایتون عبارت‌اند از:

مقدار x را به انتهای لیست اضافه می‌نماید.	append(x)
لیست L را به انتهای لیست اضافه می‌نماید.	extend(L)



مقدار x را در اندیس i جایگذاری نموده و بقیه مقادیر را به راست شیفت می‌دهد.	insert(i, x)
اولین مقدار x را از لیست حذف می‌نماید.	remove(x)
اندیس i را از لیست حذف می‌نماید و مقدار آن را بر می‌گرداند همچنانین اگر این متدهای پارامتر ورودی باشد این عمل را برای آخرین خانه انجام می‌دهد.	pop([i])
اولین اندیسی را که دارای مقدار x است را بر می‌گرداند.	index(x)
تعداد اندیس‌هایی را که دارای مقدار x می‌باشند را بر می‌گرداند.	count(x)
یک لیست را مرتب می‌نماید.	sort()
یک لیست را معکوس می‌نماید.	reverse()
طول لیست L را به ما نشان می‌دهد.	Len(L)

لیست به عنوان پشته (stack)

همان‌طور که می‌دانید ساختار پشته به صورت (last-in, first-out) است بنابراین پیاده‌سازی لیست به عنوان پشته بسیار آسان است.

```
stack = [2, 3, 4, 5]
stack.append(6)
stack.append(7)
print(stack.pop())
print(stack)

>>>
7
[2, 3, 4, 5, 6]
```

لیست به عنوان صف (queue)

همان‌طور که می‌دانید ساختار صف به صورت (first-in first-out) است پس اندیس صفر اولین ورودی در نظر گرفته می‌شود ولی لیست‌ها برای پیاده‌سازی صف مناسب نیستند زیرا عمل حذف از اول لیست هزینه بالایی برای برنامه خواهد داشت (بعد از هر بار حذف از ابتدای لیست باید تمامی مقادیر شیفت داده شوند) بنابراین بهتر است تا برای پیاده‌سازی صف از collections.deque که برای این کار بهینه‌سازی شده استفاده نمایید به مثال زیر توجه نمایید:



```

from collections import deque
queue = deque(["Ali", "Milad", "Mohsen"])
queue.append("Reza")
queue.append("Sadegh")
print(queue.popleft())
print(queue.popleft())
print(queue)

>>>
Ali
Milad
deque(['Mohsen', 'Reza', 'Sadegh'])

```

یکی از نکات جالب در لیست استفاده از اندیس با علامت منفی است این برای مفسر پایتون به معنی اندیس دهی از انتهای لیست است یعنی وقتی برنامهنویس اندیس را `-1` قرار می‌دهد به آخرین مقدار از لیست خود اشاره نموده است به مثال‌های زیر توجه نمایید:

```

List = ['Iran', [12, 13, 14, 15], [16, 17], 1234.5]
print(List[-1])
print(List[-2])
print(List[-3][-2])

>>>
1234.5
[16, 17]
14

```

تاپل (Tuples)

تاپل‌ها نیز مانند لیست و دیکشنری داده‌هایی چندقسمتی هستند با این تفاوت که برخلاف لیست تاپل حالتی تغییرناپذیر دارد و به طور مستقیم نمی‌توان یک تاپل را تغییر داد به این معنا که بعد از ساخته شدن امکان تغییر در تاپل‌ها وجود ندارد.

برای تعریف یک تاپل کافی است برنامهنویس مقادیر را بین `()` قرار داده و آن‌ها را با علامت `,` از هم جدا کند، این مانند آن است که برنامهنویس چند مقدار را به یک متغیر نسبت دهد آنگاه تنها باید حواسitan باشد که متغیر شما از نوع تاپل است به مثال‌های زیر توجه نمایید:



```

tuple1 = 12345, 54321, 'hello'
tuple2 = (12345, 54321, 'hello')
print (tuple1)
print (tuple2)
print (tuple2[0] )

>>>
(12345, 54321, 'hello')
(12345, 54321, 'hello')
12345

```

نکته قابل توجه در تاپل‌ها این است که هیچ متدهای برای آن‌ها وجود ندارد همچنین تاپل‌ها این توانایی را دارند که به صورت تودرتو تعریف شوند به مثال‌های زیر توجه نمایید:

```

tuple1 = 1, 2, 3
tuple2 = tuple1, (4, 5, 6)
tuple3 = tuple2, 7, 8, 9
print (tuple1)
print (tuple2)
print (tuple3)
print (tuple3[0][1][-1] )

>>>
(1, 2, 3)
((1, 2, 3), (4, 5, 6))
(((1, 2, 3), (4, 5, 6)), 7, 8, 9)
6

```

(String) رشته

رشته مجموعه‌ای از کاراکترها است که در کنار هم تشکیل متن را می‌دهند. برای مقداردهی در داخل رشته‌ها می‌توان از %s استفاده کرد همچنین برای رشته‌های چندخطی می‌توانید از کاراکترهای """ استفاده کرد کافی سنت در ابتدا و انتهای متن موردنظرتان از آن‌ها استفاده نمایید به مثال‌های زیر توجه فرمایید:

```

print("""Hi dear
How are you??
Iam Fine Thanks""")
Name = 'Mahmod'
Family = 'Hesabi'
print('My Name Is%s My Famil is%s' % (Name,Family) )

>>>
Hi dear
How are you??
Iam Fine Thanks
My Name Is Mahmod My Famil is Hesabi

```

توجه داشته باشید در مثال بالا Name,Family یک تاپل است.



Sets

نوع داده‌ای sets یک پیاده‌سازی برای مجموعه‌ای از مجموعه‌های است به این صورت که با استفاده از این نوع داده‌ای برنامه‌نویس این توانایی را خواهد داشت که مثلاً اجتماع دو مجموعه را حساب کند به مثال‌های زیر توجه نمایید:

```
basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
fruit = set(basket)
print(fruit)
a = set('abracadabra')
b = set('alacazam')
print(a)
print(b)
print(a - b) # in a but not in b
print(a | b) # in either a or b
print(a & b) # in both a and b
print(a ^ b) # in a or b but not both

>>>
set(['orange', 'pear', 'apple', 'banana'])
set(['a', 'r', 'b', 'c', 'd'])
set(['a', 'c', 'z', 'm', 'l'])
set(['r', 'b', 'd'])
set(['a', 'c', 'b', 'd', 'm', 'l', 'r', 'z'])
set(['a', 'c'])
set(['b', 'd', 'm', 'l', 'r', 'z'])
```

همان‌طور که مشاهده می‌کنید می‌توان مقدار یک لیست را در sets ذخیره کرد همچنین وقتی یک‌رشته را در sets ذخیره می‌کنید مقدارهای تکراری به صورت خودکار حذف شده‌اند و تمامی عملیاتی که روی دو مجموعه قابل انجام است برای این نوع داده‌ای قابل اجراست.

متدهای متداول sets عبارت‌اند از

برای اجتماع دو مجموعه از این متده استفاده می‌شود.	union()
برای اشتراک دو مجموعه از این متده استفاده می‌شود.	intersection()
برای به دست آوردن تفاوت دو مجموعه از این متده استفاده می‌شود.	symmetric_difference()
برای بررسی اینکه آیا یک مجموعه زیرمجموعه دیگری است از این متده استفاده می‌شود.	issubset()
برای بررسی اینکه آیا این مجموعه مرجع مجموعه دیگر است از این متده استفاده می‌شود.	issuperset()
برای اضافه کردن یک مقدار به مجموعه مورد استفاده قرار می‌گیرد.	add()
برای حذف یک مقدار در مجموعه مورد استفاده قرار می‌گیرد.	dell



ساختارهای کنترلی

در حقیقت یک برنامه مجموعه‌ای از دستورالعمل‌های است که توسط برنامه‌نویس در برنامه اعمال می‌شود در حالت کلی برنامه‌نویس با دو ساختار می‌تواند نتایج و روند برنامه خود را کنترل نماید.

1. ساختار شرطی

2. ساختار تکرار (حلقه)

اگر با ساختارهای برنامه‌نویسی در زبان‌هایی مانند C آشنایی داشته باشید به راحتی می‌توانید این قسمت را متوجه شوید تنها فرق پایتون با زبانی مانند C این است که در زبانی مانند C بدن حلقه‌ها با {} مشخص می‌گردد ولی در زبان پایتون از فاصله‌ها برای مشخص کردن بدن حلقه استفاده می‌شود.

ساختار شرطی

عبارت شرطی if

پایتون به رویی بسیار ساده این امکان را به برنامه‌نویس می‌دهد تا ساختار خود را کنترل نماید برای این کار کافی است تا از کلمه رزرو شده if استفاده کرده و بعدازآن عبارت شرطی نوشته شود که این عبارت در هنگام اجرا بررسی شده و در صورت صحت شرط اقدام به اجرای قطعه کد موردنظر برنامه‌نویس می‌نماید ساختار کلی آن به صورت زیر است

```
if: شرط
    قطعه کد موردنظر
else:
    قطعه کد موردنظر
```

برای مثال قطعه کد زیر بررسی می‌نماید که آیا مقدار متغیر Val بر عدد 7 بخش‌پذیر است یا خیر

```
Val = 14
Remaining = Val% 7
if Remaining == 0:
    print ("value is Divisible!")
if Remaining!= 0:
    print ("value isn't Divisible!")
```

همچنین شما می‌توانید بحای شرط دوم از کلمه کلیدی else به معنی در غیر این صورت نیز استفاده کنید



```

Val = 14
Remaining = Val% 7
if Remaining == 0:
    print ("value is Divisible!")
else:
    print ("value isn't Divisible!")

```

عبارت if و elif

در این ساختار که به صورت شرط و زیر شرط پیاده‌سازی شده (شرط و زیر شرط در اینجا بدان معناست که اگر شرط درست نبود به سراغ زیر شرط می‌رود و همین‌طور تا آخرین زیر شرط) برنامه در صورت درست نبودن هر شرط باید به سراغ زیر شرط آن شرط برود که تعداد زیر شرط‌ها می‌تواند صفرتاً بی‌نهایت زیر شرط باشد برای مثال در تکه کد زیر کاربرد این عبارت به روشنی نمایش داده شده است.

```

Val = 42
if x < 0:
    x = 0
    print ('Negative changed to zero')
elif x == 0:
    print ('Zero')
elif x == 1:
    print ('Single')
else:
    print ('More')

```

ساختار تکرار (حلقه)

حلقه while

یکی از ساختارهای تکرار در زبان پایتون حلقه while است ساختار این حلقه به این صورت است که تا زمانی که شرط ابتدای حلقه صحیح باشد حلقه تکرار خواهد شد و تنها زمانی برنامه از حلقه خارج می‌شود که مقدار شرط صحیح نباشد. ساختار کلی این حلقه به این صورت است که ابتدا کلمه کلیدی while پس از آن شرط حلقه نمایش داده می‌شود و بعد از آن بدنه حلقه تعریف می‌شود در انتهای بدنه حلقه بخش دیگری وجود دارد else که در صورت پایان حلقه مفسر اقدام به اجرای این بخش می‌نماید لازم به ذکر است که این بخش از حلقه while اختیاری بوده و برنامه‌نویس بنا به ضرورت می‌تواند از آن استفاده نماید یا از آن استفاده ننماید ساختار کلی این حلقه به صورت زیر است.

```

while : شرط:
    قطعه کد موردنظر
else:
    قطعه کد موردنظر

```

برای مثال به تکه کد زیر توجه نمایید



```
a = 0
while a < 10:
    print (a)
    a = a+1
```

این تکه کد اعداد 0 تا 9 را برای شما چاپ می‌نماید.

حلقه for

ساختار حلقه for در زبان پایتون کمی با زبان‌هایی مانند C و Pascal متفاوت است به این صورت که در این زبان‌ها برنامه‌نویس تعداد دفعات اجرای حلقه را با یک شمارنده کنترل می‌نماید ولی در پایتون این دستور به عنوان یک پیمایش گر آرایه مورد استفاده قرار می‌گیرد به این صورت که در هر دور پیمایش حلقه، مقدار یکی از عناصر آرایه را دریافت کرده و در طول حلقه از آن استفاده می‌نماید ساختار کلی حلقه for به صورت زیر است.

```
:شیء قابل پیمایش متغیر پیمایش گر
for قطعه کد موردنظر
else:
    قطعه کد موردنظر
```

همان‌طور که مشاهده می‌نمایید برنامه‌نویس در اینجا نیز مانند حلقه while بعد از حلقه for به یک دسترسی دارد که مفسر پایتون در صورت به پایان رساندن حلقه اقدام به اجرای دستورات این قطعه کد می‌نماید استفاده از else در اینجا نیز مانند حلقه while اختیاری است و بستگی به نظر برنامه‌نویس دارد. برای مثال به تکه کد زیر توجه نمایید

```
a = ['cat', 'window', 'defenestrate']
for x in a:
    print (x, len(x))
```

تکه کد بالا عناصر آرایه a را با طول هر عنصر برای شما چاپ می‌نماید.

استفاده از حلقه for در پایتون شاید کمی مبهم به نظر برسد ولی با کمی کار عملی متوجه می‌شوید که ساختار آن بسیار انعطاف‌پذیر و قدرتمند است همچنین برای پیمایش محدوده‌ای از یک آرایه برنامه‌نویس می‌تواند از تابع range() استفاده نماید.

تابع range()

اگر این تابع را با یک آرگومان عددی صدا زده شود خروجی این تابع به صورت یک آرایه از اعداد بین صفرتا آرگومان ورودی است و اگر این تابع با دو آرگومان عددی صدا زده شود خروجی این تابع به صورت آرایه‌ای از اعداد بین آرگومان اول تا آرگومان دوم است همچنین می‌توان این تابع را با سه آرگومان نیز صدا زد که آرگومان سوم تبدیل به طول گام بین هر عضو است به مثال‌های زیر توجه کنید



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

>>> range(5, 10)
[5, 6, 7, 8, 9]

>>> range(0, 10, 3)
[0, 3, 6, 9]
```

در حقیقت این تابع برای استفاده از محدوده‌ای از یک آرایه است برای درک بهتر این تابع به مثال‌های زیر توجه نمایید:

```
for x in range(7):
    print(a[x])
```

خروجی تکه کد بالا 7 خانه اول آرایه a است، به تکه کد زیر توجه کنید:

```
a =[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
for x in range(5,len(a)):
    print(a[x])
```

خروجی این کد از خانه شماره 5 تا آخرین خانه آن آرایه a است.

دستورات *continue* و *break*

رفتار دستورات continue و break در زبان پایتون مانند همانند آن‌ها در زبان برنامه‌نویسی C است به این صورت که دستور break باعث می‌شود تا برنامه بدون آزمودن مجدد شرط از داخلی‌ترین حلقه در حال اجرا بیرون بپردازد و دستور continue باعث می‌شود تا برنامه به خط اول حلقه رفته و شرط را دوباره چک نماید در این صورت اگر شرط صحیح باشد دوباره حلقه تکرار می‌شود و در غیر این صورت اجرای حلقه پایان می‌پذیرد به مثال‌های زیر توجه نمایید:

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            break
        else:
            print(n, 'is a prime number')
```

تکه کد بالا اعداد اول بین 2 تا 10 را برای شما نمایش می‌دهد.



pass

این دستور معادل هیچی یا همان مقدار `NULL` است به این معنی که با صدازدن این دستور هیچ اتفاقی نخواهد افتاد برای مثال تکه کد زیر یک حلقه بی‌نهایت خواهد بود که تنها با زدن کلیدهای ترکیبی `Ctrl+C` خاتمه می‌یابد

```
while True:  
    pass
```

مقایسه دو حلقه `while` و `for`

این دو حلقه ازنظر معنایی فرق عمده‌ای باهم ندارند و در هر جا می‌توان یکی را به جای دیگری استفاده نمود.

توابع

تعريف تابع در علم رایانه، به عنوان حالت خاصی از یک رابطه، به طور گسترده‌تر در منطق و علم تئوری رایانه مطالعه می‌شود. در حقیقت یک تابع نکه کدی است که برنامه‌نویس نامی را به آن اختصاص داده است. از این طریق برنامه‌نویس می‌تواند تا پارامترهایی را به تابع خود ارسال و نتایج آن را به عنوان خروجی دریافت نماید همچنین می‌تواند تا از این تکه کد بارها در طول یک برنامه استفاده نماید که این امر باعث جلوگیری از فرونی کد و همچنین باعث خواناتر شدن برنامه می‌گردد.

تعريف تابع

برای تعريف یک تابع ما ابتدا کلمه `def` و بعد از آن نام تابع را و سپس داخل پرانتز نام متغیرهایی را که به تابع ارسال می‌شود را می‌آوریم سپس با رعایت فاصله‌گذاری می‌توانیم بدنه تابع را تعريف نماییم همچنین توجه نمایید که قواعد نام‌گذاری تابع مانند قواعد نام‌گذاری متغیرها است. همچنین برای بازگرداندن یک مقدار از داخل تابع می‌توان از کلمه کلیدی `return` استفاده کرد به مثال زیر توجه فرمایید:

```
def fib(n):  
    a, b = 0, 1  
    for x in range(n-1):  
        a, b = b, a+b  
    return a  
print (fib(10) )
```

همان‌طور که مشاهده می‌نمایید تابع بالا جمله `n` ام سری فیبوناچی را در خروجی چاپ می‌نماید. در زبان پایتون برنامه‌نویس می‌تواند تا مقدار یک پارامتر ارسالی به تابع را به صورت پیش‌فرض تعیین نماید این بدان معناست که اگر آن پارامتر به تابع ارسال نگردد تابع مقدار پیش‌فرض را برای آن در نظر می‌گیرد تعريف



مقدار پیش‌فرض در پایتون بسیار ساده است کافی است برنامه‌نویس در هنگام تعریف پارامترهای ورودی آن‌ها را مساوی مقدار پیش‌فرض قرار دهد به مثال زیر توجه نمایید:

```
def fib(n = 5):
    a, b = 0, 1
    for x in range(n-1):
        a, b = b, a+b
    return a
print(fib())
```

در مثال بالا خروجی تابع جمله پنجم سری فیبوناچی است. یکی دیگر از توانایی‌های زبان پایتون در تعریف توابع ارسال پارامتر به آن‌ها با استفاده از کلمات کلیدی است این بدان معناست که بنام بردن پارامتر ورودی و مقداردهی به آن هیچ محدودیتی در جابجایی پارامترهای ورودی نخواهید داشت همچنین برنامه‌نویس می‌تواند به صورت ترکیبی نیز از این توانایی استفاده کند اما در ارسال پارامترها باید دقیق شود تا پارامترها بر روی یکدیگر نوشته نشوند به مثال‌های زیر توجه نمایید:

```
def keywordFunction(Family, Age='None age', Country='Iran'):
    print ("-- Mr/Miss", Family, )
    print ("has", Age, "and he/she is from",Country)

keywordFunction('Hosseini')
keywordFunction(Age = '25',Family = 'Hassani')
keywordFunction('Hashemi',Country = 'Canada')

>>> --Mr/Miss Hosseini has None age and he/she is from Iran
>>> --Mr/Miss Hassani has 25 and he/she is from Iran
>>> --Mr/Miss Hashemi has None age and he/she is from Canada
```

همان‌طور که مشاهده نمودید فراخوانی توابع با پارامترهای بالا به درستی کارکرد ولی فراخوانی‌های زیر با مشکل مواجه می‌شود

```
keywordFunction('Hassani',Family = 'Hashemi')
keywordFunction(Family = 'Hoseini','18')
keywordFunction()
keywordFunction(Age = 18)
```

به‌هرحال انتخاب با برنامه‌نویس است که چگونه پارامترها را به توابع ارسال نماید ولی ویراستار این متن استفاده از روش ترکیبی در ارسال پارامترها به توابع را پیشنهاد نمی‌نماید.

آزمودن مقادیر ورودی توابع

متغیرها در زبان پایتون نوع مشخصی ندارند و در طول برنامه ممکن است نوع آن‌ها تغییر نماید که این امر ممکن است در اجرای برنامه ایجاد اشکال نماید لذا شدیداً توصیه می‌شود تا قبل از استفاده از پارامترهای ارسالی



از صحت نوع پارامتر اطمینان حاصل نمایید برای این کار برنامه‌نویس می‌تواند از تابع `type` استفاده نماید به مثال زیر توجه نمایید

```
def fib(n):
    if type(n) != int:
        return type(n)
    a, b = 0, 1
    for x in range(n-1):
        a, b = b, a + b
    return a
print (fib(10))
>>>
34
```

همان‌طور که مشاهده می‌نمایید در تابع جمله چندم سری فیبوناچی بررسی می‌نماییم که اگر پارامتر ارسالی از نوع عددی نبود نوع آن را به ما برمی‌گرداند و در غیر این صورت روال معمول برنامه اجرا خواهد شد.
تابع تو در تو

در زبان پایتون این امکان برای برنامه‌نویس وجود دارد که در داخل تابع خود اقدام به تعریف تابع دیگر نماید این توانایی در تعریف تابع بزرگ و پیچیده بسیار مفید است و برنامه‌نویس را در مدیریت کد خود کمک می‌کند.

تعریف زیر تابع مانند تابع معمولی است و تنها فرق آن‌ها با تابع معمولی این است که این تابع در بدنه اصلی برنامه قابل استفاده نیستند و تنها در داخل تابع خودشان می‌توان از آن‌ها استفاده نمود.

تعریف متغیرها در تابع

متغیرهایی که در یک تابع تعریف می‌شوند تنها درون آن تابع معتبر می‌باشند و هیچ ارتباطی با متغیرهای همنام خارج از تابع نخواهند داشت این بدان معناست که ارسال متغیرها در پایتون به صورت فراخوانی با مقدار (Call by Value) است به مثال زیر توجه نمایید:

```
def func(x):
    print 'x is', x
    x = 2
    x = x + 5
    print ('Changed local x to', x)

x = 50
func(x)
print ('x is still', x)

>>> x is 50
>>> Changed local x to 7
>>> x is still 50
```



همان طور که مشاهده می‌کنید تغییراتی که در طول برنامه بر روی متغیر `x` اعمال نمودیم هیچ تأثیری بر روی متغیر خارجی نداشت و اخلاقی در آن ایجاد نکرد.

همان طور که مشاهده نمودید تعریف پیش‌فرض برای متغیرهای ارسال شده به توابع تنها یکبار صورت می‌گیرد این موضوع در مورد ساختارهای ناپایداری مانند لیست‌ها کمی متفاوت است به مثال زیر توجه فرمایید:

```
def f(a, L= []):
    L.append(a)
    return L

print (f(1))
print (f(2))
print (f(3))

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

همان طور که مشاهده نمودید در هر بار ارسال مقدار به تابع متغیر `L` مقدار قبلی خود را حفظ کرده بود اگر شما تمایلی به حفظ مقدار قبلی متغیر خود در فراخوانی‌های چندباره ندارید می‌توانید کد بالا را به صورت زیر تصحیح کنید و دوباره آن را آزمودن نمایید.

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L

A = [1, 2]
print (f(1))
print (f(2))
print (f(3, A))

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

مشاهده می‌نمایید که مقادیر به اشتراک گذاشته نشده است.

همچنین در پایتون شما می‌توانید تا متغیرهای خود را به صورت سراسری (Global) نیز تعریف نمایید تعریف یک متغیر سراسری بدین معناست که برنامه‌نویس می‌خواهد یک متغیر درون تابع را به مقداری خارج از آن تابع منتبه نماید برای این کار باید از کلمه کلیدی `global` قبل از نام متغیر خود استفاده نماید به مثال زیر توجه نمایید:



```

def func():
    global x
    print ('x is', x)
    x = 2
    print ('Changed global x to', x)

x = 50
func()
print ('Value of x is', x)

>>> x is 50
>>> Changed global x to 2
>>> Value of x is 2

```

همان‌طور که مشاهده نمودید تغییراتی که بر روی متغیر x انجام شد بر روی متغیر متناظر خارجی‌اش نیز اثر گذاشت که این همان معنی تعریف یک متغیر به صورت سراسری است.

یکی دیگر از ویژگی‌های جالب پایتون Documentation string است این ویژگی این امکان را به برنامه‌نویس می‌دهد تا توضیحات کد خود را در میان کد قرار دهد که این امر در برنامه‌های بزرگ و در پروژه‌هایی که به صورت گروهی انجام می‌شود بسیار حائز اهمیت است نکته جالب دیگری که در این مورد می‌توان گفت این مطلب است که این توضیحات در حین اجرای برنامه نیز قابل دسترس می‌باشند. به تکه کد زیر توجه نمایید:

```

def my_function():
    """ Do nothing, but document it.
    No, really, it doesn't do anything.
    """
    pass

print (my_function.__doc__)

>>> Do nothing, but document it.
No, really, it doesn't do anything.

```

همان‌طور که مشاهده کردید توضیحات با ("") آغاز و با همین کاراکترها به پایان می‌رسند. همچنین توجه داشته باشید که توضیحات را باید در ابتدای تابع و قبل از سایر کدها بنویسید در غیر این صورت توضیحات شما برای برنامه قابل دسترس نخواهد بود به مثال زیر توجه نمایید:

```

def my_function():
    print 2+2
    """Do nothing, but document it.
    No, really, it doesn't do anything.
    """

print (my_function.__doc__)
my_function()

>>> None
>>> 4

```



توجه داشته باشید که در پایتون بدنه توابع با فاصله‌گذاری مشخص می‌شود.

توابع از قبل طراحی شده (Built-in function)

در زبان برنامه‌نویسی پایتون توابعی وجود دارد که از قبیل پیاده‌سازی شده‌اند. طراحان پایتون در زمان پیاده‌سازی زبان پایتون این توابع را پیاده‌سازی کرده و همراه با نسخه‌های پایتون منتشر می‌شود در ادامه فهرستی از این توابع را مشاهده می‌نمایید:

Built-in Functions				
set()	memoryview()	hash()	delattr()	abs()
setattr()	min()	help()	dict()	all()
slice()	next()	hex()	dir()	any()
sorted()	object()	id()	divmod()	ascii()
staticmethod()	oct()	input()	enumerate()	bin()
str()	open()	int()	eval()	bool()
sum()	ord()	isinstance()	exec()	breakpoint()
super()	pow()	issubclass()	filter()	bytearray()
tuple()	print()	iter()	float()	bytes()
type()	property()	len()	format()	callable()
vars()	range()	list()	frozenset()	chr()
zip()	repr()	locals()	getattr()	classmethod()
__import__()	reversed()	map()	globals()	compile()
	round()	max()	hasattr()	complex()

بعضی از این توابع برای ما آشنا هستند مثلً `print()` که باعث چاپ یک مقدار در خروجی می‌شود یا `range()` که در حلقه‌ها مورداستفاده قرار می‌گرفت. در این بخش قصد داریم تا تعدادی از پرکاربردترین این توابع را برای شما معرفی کنیم:

`abs(x)`

همان‌طور که مشخص است این تابع یک پارامتر را از ورودی دریافت کرده و مقدار قدر مطلق آن را در خروجی بر می‌گرداند. به مثال زیر توجه کنید:



```

float = -54.26
print('Absolute value of float is:', abs(float))

# An integer
int = -94
print('Absolute value of integer is:', abs(int))

# A complex number
complex = (3 - 4j)
print('Absolute value or Magnitude of complex is:', abs(complex))
>>>
Absolute value of float is: 54.26
Absolute value of integer is: 94
Absolute value or Magnitude of complex is: 5.0

```

ascii(object)

این متدها مقدار اسکی قابل نمایش از یک شیء را برمی‌گردانند. به مثال‌های زیر توجه کنید:

```

>>> ascii("¥")
"'\\"xa5'"
>>> ascii("µ")
"'\\"xb5'"
>>> ascii("Ё")
"'\\"xcb'"

```

bin(x)

این تابع معادل دو دویی مقدار ورودی را به عنوان خروجی برمی‌گرداند. به مثال‌های زیر توجه کنید:

```

>>> bin(2)
'0b10'
>>> bin(10)
'0b1010'
>>> bin(16)
'0b10000'

```

dir([object])

همان‌طور که مشخص است این تابع یک شیء را به عنوان پارامتر ورودی دریافت می‌کند و فهرستی از ویژگی‌ها و متدهای آن شیء را به عنوان خروجی برمی‌گرداند. به مثال زیر توجه کنید:



```

import random
print("The contents of random library are ::")
print(dir(random))
>>>
The contents of random library are ::

['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF', 'Random', 'SG_MAGICCONST',
 'SystemRandom', 'TWOPI', '_BuiltinMethodType', '_MethodType', '_Sequence',
 '_Set', '__all__', '__builtins__', '__cached__', '__doc__', '__file__',
 '__loader__', '__name__', '__package__', '__spec__', '_acos', '_ceil', '_cos', '_e',
 '_exp', '_inst', '_log', '_pi', '_random', '_sha512', '_sin', '_sqrt', '_test',
 '_test_generator',
 '_urandom', '_warn', 'betavariate', 'choice', 'expovariate', 'gammavariate',
 'gauss',
 'getrandbits', 'getstate', 'lognormvariate', 'normalvariate',
 'paretovariate', 'randint',
 'random', 'randrange', 'sample', 'seed', 'setstate', 'shuffle', 'triangular',
 'uniform',
 'vonmisesvariate', 'weibullvariate']

```

divmod(a, b)

این تابع دو مقدار عددی را از ورودی دریافت کرده و یک مقدار جفت عددی از مقدار تقسیم و باقیمانده تقسیم دو عدد را به یکدیگر برمی‌گرداند. به مثال‌های زیر توجه کنید:

```

# divmod() with int
print('(5, 4) = ', divmod(5, 4))
print('(10, 16) = ', divmod(10, 16))
print('(11, 11) = ', divmod(11, 11))
print('(15, 13) = ', divmod(15, 13))

# divmod() with int and Floats
print('(6.0, 5) = ', divmod(8.0, 3))
print('(3, 9.0) = ', divmod(3, 8.0))
print('(13.5, 6.2) = ', divmod(7.5, 2.5))
print('(1.6, 10.7) = ', divmod(2.6, 0.5))

>>>
(5, 4) = (1, 1)
(10, 16) = (0, 10)
(11, 11) = (1, 0)
(15, 13) = (1, 2)
(6.0, 5) = (2.0, 2.0)
(3, 9.0) = (0.0, 3.0)
(13.5, 6.2) = (3.0, 0.0)
(1.6, 10.7) = (5.0, 0.10000000000000009)

```



`eval(expression[, globals[, locals]])`

این تابع یک عبارت را ارزیابی و اجرا می‌کند. به مثال‌های زیر توجه کنید:

```
>>> eval("print(\"this is a test\")")
this is a test
```

همان‌طور که در مثال بالا مشاهده می‌کنید یک دستور پایتون را به صورت متنی به این تابع ارسال کردیم و متده `eval` متن را تفسیر و اجرا نمود. به مثال دیگری در این زمینه توجه کنید:

```
eval = eval(input("Enter any number of your choice"))
print(eval)
print(type(eval))

>>>
Enter any number of your choice 10 * 10
100
<class 'int'>
```

همان‌طور که در این مثال مشاهده می‌کنید، مقدار $10 * 10$ را به عنوان ورودی به تابع دادیم و توسط متده `eval` تفسیر و خروجی آن برگردانده شد.

`exec(object[, globals[, locals]])`

یک مقدار را به عنوان ورودی دریافت کرده و آن را به عنوان یک کد اجرا می‌نماید به مثال زیر توجه نمایید:

```
prog = 'print("The sum of 5 and 10 is", (5+10))'
exec(prog)

>>>
The sum of 5 and 10 is 15
```

همان‌طور که مشاهده می‌کنید مقدار ورودی یک کد پایتون است که توسط تابع `exec` اجرا شده و خروجی اش بازگردانده می‌شود.

تابع `exec` و `eval` دارای دو تفاوت اساسی هستند در ادامه به توضیح این دو تفاوت خواهیم پرداخت:
 eval تنها یک عبارت واحد را می‌پذیرد درحالی که متده `exec` می‌تواند یک بلاک از کد که شامل حلقه مدیریت خطای کلاس و یا تابع باشد را دریافت و اجرا نماید. (منظور از یک عبارت واحد مقدار کدی است که شما می‌توانید مقدار آن را برابر با یک متغیر قرار دهید است)

تابع `eval` مقدار عبارت داده شده را بعد از اجرا برمی‌گرداند، این در حالی است که متده `exec` مقدار برگردانده شده از کد را نادیده می‌گیرد و همیشه برای آن مقدار `None` برمی‌گرداند.



به مثال‌های زیر توجه نمایید:

```
>>> program = """
for i in range(3):
    print("Python is cool")
"""

>>> exec(program)
Python is cool
Python is cool
Python is cool
>>> eval(program)
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    eval(program)
  File "<string>", line 2
    for i in range(3):
      ^
SyntaxError: invalid syntax
```

همان‌طور که مشاهده کردید متدهای eval امکان اجرای عبارت فوق را ندارد درحالی‌که متدهای exec بدون هیچ مشکلی آن را اجرا نمود. به مثال‌های بیشتری در این زمینه توجه کنید:

```
>>> a = 2
>>> my_calculation = '42 * a'
>>> exec(my_calculation)
>>> eval(my_calculation)
```

84

همان‌طور که مشاهده می‌کنید تفاوت دیگر این دو متدهای eval و exec در این مثال نمایان است. همیچین می‌توانید برای درک بهتر مثال‌های زیر را نیز بررسی کنید:

```
x = eval('5')           # x <- 5
x = eval('%d + 6' % x)  # x <- 11
x = eval('abs(%d)' % -100) # x <- 100
x = eval('x = 5')        # INVALID; assignment is not an expression.
x = eval('if 1: x = 4')  # INVALID; if is a statement, not an expression.

exec('print(5)')         # prints 5.
exec('print(5)\nprint(6)') # prints 5{newline}6.
exec('if True: print(6)') # prints 6.
exec('5')                # does nothing and returns nothing.
```

filter(function, iterable)

این متدهای تابعی و یک لیست را به عنوان ورودی دریافت می‌کند. از این تابع برای حذف بعضی مقادیر در یک مقدار تکرارشونده استفاده می‌شود. به مثال‌های زیر توجه نمایید:



```
# a list contains both even and odd numbers.
seq = [0, 1, 2, 3, 5, 8, 13]

# result contains odd numbers of the list
result = filter(lambda x: x % 2 != 0, seq)
print(list(result))

# result contains even numbers of the list
result = filter(lambda x: x % 2 == 0, seq)
print(list(result))

>>>
[1, 3, 5, 13]
[0, 2, 8]
```

format(value[, format_spec])

این متده برای تبدیل یک نوع به انواع دیگر در پایتون مورداستفاده قرار می‌گیرد. به مثال‌های زیر توجه نمایید:

```
>>> format(255, 'x')
ff
```

همان‌طور که مشاهده می‌کنید مقدار 255 با این متده فرمت هگزادسیمال تبدیل شده است.

getattr(object, name[, default])

این متده مقدار یک ویژگی مشخص را بر می‌گرداند. به مثال زیر توجه نمایید:

```
class Person:
    name = "John"
    age = 36
    country = "Norway"

x = getattr(Person, 'age')
print(x)
>>>
36
```

توجه داشته باشید که روش مرسوم فراخوانی ویژگی‌ها نسبت به متده فوق در زمان کمتری فراخوانی می‌شود ولی زمانی که بخواهیم مقدار یک ویژگی که نامش در یک متغیر رشته‌ای ذخیره شده را بگیریم یا زمانی که مقدار پیش‌فرضی برای یک ویژگی وجود ندارد این متده گزینه مناسبی است به مثال زیر توجه نمایید:



```

# performance analysis of getattr()
import time

# declaring class
class Pbod :
    name = "Python based on documentation"
    age = 24

# initializing object
obj = Pbod()

# use of getattr to print name
start_getattr = time.time()
print("The name is " + getattr(obj, 'name'))
print("Time to execute getattr " + str(time.time() - start_getattr))

# use of conventional method to print name
start_obj = time.time()
print("The name is " + obj.name)
print("Time to execute conventional method " + str(time.time() - start_obj))
>>>
The name is Python based on documentation
Time to execute getattr 0.05714058876037598
The name is Python based on documentation
Time to execute conventional method 0.012974262237548828

```

همان‌طور که مشاهده می‌کنید فراخوانی به روش مرسوم بازمان بسیار کمتری کار می‌کند.

setattr(object, name, value)

این متده مقدار یک ویژگی را دریکشی تنظیم می‌کند. به مثال زیر توجه نمایید:

```

class Person:
    name = "John"
    age = 36
    country = "Norway"

setattr(Person, 'age', 40)

print(Person.age)
>>>
40

```

همچنین به مثال دیگری در این زمینه توجه نمایید:



```

# properties of setattr()

# initializing class
class Pbod:
    name = 'Python based on documentation'

# initializing object
obj = Pbod()

# printing object before setattr
print("Before setattr name : ", str(obj.name))

# using setattr to assign None to name
setattr(obj, 'name', None)

# using setattr to initialize new attribute
setattr(obj, 'description', 'CS portal')

# printing object after setattr
print("After setattr name : " + str(obj.name))
print("After setattr description : ", str(obj.description))
>>>
Before setattr name : Python based on documentantion
After setattr name : None
After setattr description : CS portal

```

delattr(object, name)

این متد یک ویژگی مشخص را در یک شیء مشخص حذف می‌کند. به مثال زیر توجه نمایید:



```

# Python code to illustrate delattr()
class Python:
    stu1 = "Henry"
    stu2 = "Zack"
    stu3 = "Stephen"

names = Python()

print('Students before delattr()--')
print('First = ', names.stu1)
print('Second = ', names.stu2)
print('Third = ', names.stu3)

# implementing the method
delattr(Python, 'stu3')

print('After deleting fifth student--')
print('First = ', names.stu1)
print('Second = ', names.stu2)
# this statement raises an error
print('Third = ', names.stu3)
>>>
Students before delattr()--
First = Henry
Second = Zack
Third = Stephen
After deleting fifth student--
First = Henry
Second = Zack
Traceback (most recent call last):
  File "C:/Users/pbod/Desktop/test.py", line 23, in <module>
    print('Third = ', names.stu3)
AttributeError: 'Python' object has no attribute 'stu3'

```

hasattr(object, name)

این متده بررسی می کند که آیا یک ویژگی خاص دریکشی خاص وجود دارد یا خیر، در صورت وجود داشتن مقدار True و در غیر این صورت مقدار False را برمی گرداند، به مثال زیر توجه نمایید:



```

class PBOD :
    name = "Python based on documentation"
    age = 24

# initializing object
obj = PBOD()

# using hasattr() to check name
print ("Does name exist ? " + str(hasattr(obj, 'name')))

# using hasattr() to check motto
print ("Does motto exist ? " + str(hasattr(obj, 'motto')))

>>>
Does name exist ? True
Does motto exist ? False

```

hash(object)

این متدهش یک شیء مشخص را در صورت دارا بودن هش بر می گرداند. به مثال های زیر توجه نمایید:



```

# hash for integer unchanged
print('Hash for 181 is:', hash(181))

# hash for decimal
print('Hash for 181.23 is:', hash(181.23))

# hash for string
print('Hash for Python is:', hash('Python'))

# tuple of vowels
vowels = ('a', 'e', 'i', 'o', 'u')

print('The hash is:', hash(vowels))

class Person:
    def __init__(self, age, name):
        self.age = age
        self.name = name

    def __eq__(self, other):
        return self.age == other.age and self.name == other.name

    def __hash__(self):
        print('The hash is:')
        return hash((self.age, self.name))

person = Person(23, 'Adam')
print(hash(person))

>>>
Hash for 181 is: 181
Hash for 181.23 is: 530343892119126197
Hash for Python is: 4500159601682222495
The hash is: 411280729035788803
The hash is:
614277537308993031

```

همان‌طور که مشاهده می‌کنید مقدار هش برای انواع مختلف از داده‌ها متفاوت است.

`help([object])`

همان‌طور که مشخص است این تابع یک پارامتر از نوع شیء را به عنوان ورودی دریافت می‌کند و برای نمایش مستندات تابع، کلاس، مازول، کلمات کلیدی یا ... استفاده می‌شود. همچنین اگر این تابع بدون ورودی فراخوانی شود ابزار تعاملی خط فرمان این تابع فراخوانی می‌شود. به مثال‌های زیر توجه نمایید:



```
>>> help(print)
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

Prints the values to a stream, or to sys.stdout by default.
Optional keyword arguments:
file: a file-like object (stream); defaults to the current sys.stdout.
sep: string inserted between values, default a space.
end: string appended after the last value, default a newline.
flush: whether to forcibly flush the stream.
```

همان‌طور که مشاهده می‌کنید مستندات تابع از پیش طراحی شده print در خروجی برگردانده شد.



```

class Helper:
    def __init__(self):
        '''The helper class is initialized'''

    def print_help(self):
        '''Returns the help description'''
        print('helper description')

help(Helper)
help(Helper.print_help)

>>>
Help on class Helper in module __main__:

class Helper(builtins.object)
| Methods defined here:
|
|   __init__(self)
|       The helper class is initialized
|
|   print_help(self)
|       Returns the help description
|
| -----
| Data descriptors defined here:
|
|   __dict__
|       dictionary for instance variables (if defined)
|
|   __weakref__
|       list of weak references to the object (if defined)

Help on function print_help in module __main__:

print_help(self)
    Returns the help description

```

همان‌طور که مشاهده می‌کنید یک کلاس و یک تابع را ایجاد و سپس با استفاده از `help` مستندات آن را در خروجی برگرداندیم. حال ببینیم اگر `help` را بدون وردی صدا بزنیم چه اتفاقی رخ می‌دهد.



```
>>> help()
```

Welcome to Python 3.6's help utility!

If this **is** your first time using Python, you should definitely check out the tutorial on the Internet at <https://docs.python.org/3.6/tutorial/>.

Enter the name of any module, keyword, **or** topic to get help on writing Python programs **and** using Python modules. To quit this help utility **and return** to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, **or** topics, type "modules", "keywords", "symbols", **or** "topics". Each module also comes **with** a one-line summary of what it does; to list the modules whose name **or** summary contain a given string such **as** "spam", type "modules spam".

```
help>
```

همان‌طور که مشاهده می‌کنید با فراخوانی بدون متغیر ورودی ابزار تعاملی خط فرمان پایتون شروع به کار کرد، حال اگر مقداری را به عنوان ورودی برایش ارسال کنیم مستندات آن را برای ما برمی‌گرداند.

id(object)

همان‌طور که مشخص است این تابع یک پارامتر را به عنوان ورودی دریافت می‌کند و شناسه آن پارامتر را به عنوان خروجی برمی‌گرداند. این شناسه در طول عمر شیء ورودی منحصر به فرد و ثابت است.



```
# This program shows various identities
str1 = "geek"
print(id(str1))

str2 = "geek"
print(id(str2))

# This will return True
print(id(str1) == id(str2))

# Use in Lists
list1 = ["akash", "priya", "abdul"]
print(id(list1[0]))
print(id(list1[2]))

# This returns false
print(id(list1[0]) == id(list1[2]))
>>>
140252505691448
140252505691448
True
140252505691840
140252505739928
False
```

input([prompt])

از این تابع برای دریافت ورودی از خط فرمان در زبان پایتون استفاده می‌شود. به مثال زیر توجه نمایید:

```
num = input ("Enter number :")
print(num)
name1 = input("Enter name : ")
print(name1)

# Printing type of input value
print ("type of number", type(num))
print ("type of name", type(name1))

>>>
Enter number : 12
12
Enter name : python Lover
python Lover
type of number <class 'str'>
type of name <class 'str'>
```



isinstance(object, classinfo)

این متده که شیء را از ورودی دریافت کرده و نوع آن را با نوعی که در ورودی دریافت کرده مقایسه می‌نماید و در صورت که بودن مقدار True و در صورت مشکل داشتن مقدار False را بر می‌گرداند. به مثال‌های زیر توجه نمایید:

```
test_int = 5
test_str = "Python based on documentation"
test_list = [1, 2, 3]

# testing with isinstance
print ("Is test_int integer? : " + str(isinstance(test_int, int)))
print ("Is test_int string? : " + str(isinstance(test_int, str)))
print ("Is test_str string? : " + str(isinstance(test_str, str)))
print ("Is test_list integer? : " + str(isinstance(test_list, int)))
print ("Is test_list list? : " + str(isinstance(test_list, list)))

# testing with tuple
print ("Is test_int integer or list or string? : "
      + str(isinstance(test_int, (list, str, int))))
print ("Is test_list string or tuple? : "
      + str(isinstance(test_list, (str, tuple)))))

>>>
Is test_int integer? : True
Is test_int string? : False
Is test_str string? : True
Is test_list integer? : False
Is test_list list? : True
Is test_int integer or list or string? : True
Is test_list string or tuple? : False
```

issubclass(class, classinfo)

این متده که آیا یک کلاس فرزند کلاس دیگری است یا خیر، به مثال زیر توجه نمایید:

```
class myAge:
    age = 36

class myObj (myAge):
    name = "John"
    age = myAge

print(str(issubclass(myObj, myAge)))
>>>
True
```

به مثال دیگری که در زیر آمده نیز توجه نمایید:



iter(object[, sentinel])

در زبان پایتون متد فوق یک شیء قابل تکرار را به یک شیء از نوع پیمایش شونده^۴ تبدیل می‌کند. به مثال زیر توجه نمایید:

```
x = iter(["apple", "banana", "cherry"])
print(next(x))
print(next(x))
print(next(x))
>>>
apple
banana
cherry
```

map(function, iterable, ...)

همان‌طور که مشخص است این تابع دو یا بیشتر پارامتر را به عنوان ورودی دریافت کرده که به ترتیب عبارت‌اند از

: این پارامتر شامل تابعی است که مقداری را به عنوان ورودی دریافت می‌کند.
function : یک متغیر از نوع قابل تکرار^۵ است.

این تابع ورودی اول خود را یک تابع است را برای تک‌به‌تک ورودی‌های بعدی که متغیرهایی از نوع قابل تکرار هستند اجرا کرده و یک متغیر از نوع قابل تکرار را بر می‌گرداند. به مثال زیر توجه کنید:

```
def addition(n):
    return n + n

numbers = (1, 2, 3, 4)
result = map(addition, numbers)
print(list(result))
>>>
[2, 4, 6, 8]
```

next(iterator[, default])

این متد برای پیمایش یک شیء قابل پیمایش مورد استفاده قرار می‌گیرد، به مثال زیر توجه نمایید:

iterator⁴
iterable⁵



```

import time

# initializing list
list1 = [1, 2, 3, 4, 5]

# keeping list2
list2 = list1

# converting list to iterator
list1 = iter(list1)

print ("The contents of list are : ")

# printing using next()
# using default
start_next = time.time()
while (1) :
    val = next(list1,'end')
    if val == 'end':
        break
    else :
        print (val)
print ("Time taken for next() is : " + str(time.time() - start_next))

# printing using for loop
start_for = time.time()
for i in list2 :
    print (i)
print ("Time taken for loop is : " + str(time.time() - start_for))
>>>
The contents of list are :
1
2
3
4
5
Time taken for next() is : 0.013008356094360352
1
2
3
4
5
Time taken for loop is : 0.010007619857788086

```

همان‌طور که مشاهده می‌کنید برای print کردن محتوای یک لیست حلقه for بازدهی بهتری نسبت به حلقه while دارد.

pow(base, exp[, mod])

به وسیله این متده می‌توان چندم یک عدد را محاسبه نمود. به مثال‌های زیر توجه نمایید:



```

print("Positive x and positive y : ",end="")
print(pow(4, 3))

print("Negative x and positive y : ",end="")
# negative x, positive y (-x**y)
print(pow(-4, 3))

print("Positive x and negative y : ",end="")
# positive x, negative y (x**-y)
print(pow(4, -3))

print("Negative x and negative y : ",end="")
# negative x, negative y (-x**-y)
print(pow(-4, -3))
>>>
Positive x and positive y : 64
Negative x and positive y : -64
Positive x and negative y : 0.015625
Negative x and negative y : -0.015625

```

توجه داشته باشید که این تابع امکان فراخوانی با 3 متغیر را نیز دارد و درصورتی که با 3 متغیر فراخوانی شود به صورت زیر کار خواهد کرد. به مثال زیر توجه نمایید:

```

# Returns The value of (3**4) % 10
>>> print (pow(3,4,10))
1

```

همان طور که مشاهده می کنید ورودی اول را به توان ورودی دوم می رساند سپس باقیمانده آن را نسبت به ورودی سوم در خروجی برمی گرداند.

slice(stop)

این تابع این امکان را به برنامه نویس می دهد که دنباله از اشیا از هر نوع (رشته، لیست، ...) را قطعه قطعه کرده و در خروجی برگرداند. به مثال های زیر توجه نمایید:



```

String ='Python based on documentation'
s1 = slice(3)
s2 = slice(1, 5, 2)
print("String slicing")
print(String[s1])
print(String[s2])

# List slicing
L = [1, 2, 3, 4, 5]
s1 = slice(3)
s2 = slice(1, 5, 2)
print("\nList slicing")
print(L[s1])
print(L[s2])

# Tuple slicing
T = (1, 2, 3, 4, 5)
s1 = slice(3)
s2 = slice(1, 5, 2)
print("\nTuple slicing")
print(T[s1])
print(T[s2])
>>>
String slicing
Pyt
yh

List slicing
[1, 2, 3]
[2, 4]

Tuple slicing
(1, 2, 3)
(2, 4)

```

type(object)

این متده کلاس مقدار ورودی خود را برمی‌گرداند. به مثالهای زیر توجه کنید:



```
# the type() function
print(type([]) is list)
print(type([]) is not list)
print(type(()) is tuple)
print(type({}) is dict)
print(type({}) is not list)

>>>
True
False
True
True
True
```

همان‌طور که در مثال بالا می‌بینید انواع مختلف را به‌وسیله تابع فوق بررسی کردیم. به مثال‌های بیشتری در این زمینه توجه کنید:



```

# Class of type dict
class DictType:
    DictNumber = {1:'John', 2:'Wick',
                  3:'Barry', 4:'Allen'}

    # Will print the object type
    # of existing class
    print(type(DictNumber))

# Class of type list
class ListType:
    ListNumber = [1, 2, 3, 4, 5]

    # Will print the object type
    # of existing class
    print(type(ListNumber))

# Class of type tuple
class TupleType:
    TupleNumber = ('Python', 'based on', 'documentation')

    # Will print the object type
    # of existing class
    print(type(TupleNumber))

# Creating object of each class
d = DictType()
l = ListType()
t = TupleType()
>>>
<class 'dict'>
<class 'list'>
<class 'tuple'>

```

در مثال بعد دو کلاس که خودمان تعریف کرده‌ایم را با این متدها مقایسه می‌کنیم، به این مثال توجه کنید:



```

# Class of type dict
class DictType:
    DictNumber = {1:'John', 2:'Wick', 3:'Barry', 4:'Allen'}

# Class of type list
class ListType:
    ListNumber = [1, 2, 3, 4, 5]

# Creating object of each class
d = DictType()
l = ListType()

# Will print accordingly whether both
# the objects are of same type or not
if type(d) is not type(l):
    print("Both class have different object type.")
else:
    print("Same Object type")
>>>
Both class have different object type.

```

تابع lambda

در پایتون تابع لامبدا روشی ساده برای ایجاد یک تابع بینام^۶ است. ساختار نگارشی کلی یک تابع لامبدا به صورت زیر است

`lambda arguments : expression`

که در آن عبارت اجراسده و نتیجه آن برگردانده می‌شود به مثال زیر توجه کنید:

>>> `lambda x: x + 1`

در بالا یک تابع لامبدا ساده را تعریف کردیم که یک مقدار را به عنوان ورودی دریافت کرده و آن را با یک جمع می‌کند و نتیجه را به عنوان خروجی بازمی‌گرداند به مثال زیر توجه کنید:

>>> `(lambda x: x + 1)(2)`
3

همان‌طور که مشاهده می‌کنید مقدار 2 را به تابع خود ارسال کردیم و مقدار 3 در خروجی بازگردانده شد، ولی روشی که معمولاً برای فراخوانی یک تابع لامبدا استفاده می‌شود به این صورت نیست، برای استفاده از یک تابع لامبدا معمولاً آن را برابر با یک متغیر قرار داده و سپس آن متغیر را مانند یک تابع معمولی فراخوانی می‌کنیم، به مثال زیر توجه نمایید:

⁶ anonymous



```
>>> x= lambda a : a + 10
>>> print(x(5))
15
```

توجه داشته باشید که یک تابع لامبدا به صورت خودکار نتیجه را return می‌کند، همچنین یک تابع لامبدا می‌تواند به تعداد دلخواه آرگومان رودی داشته باشد، به مثال زیر توجه کنید:

```
>>> x = lambda a, b : a * b
>>> print(x(5, 6))
30
```

در بالا تابعی را پیاده‌سازی کردیم که در آن آرگومان a و آرگومان b در یکدیگر ضرب می‌شوند و نتیجه آن برگردانده می‌شود، همچنین می‌توانیم تابعی مشابه برای جمع سه عدد را نیز به راحتی پیاده‌سازی نماییم به مثال زیر توجه نمایید:

```
>>> x = lambda a, b, c : a + b + c
>>> print(x(5, 6, 2))
13
```

چرا از توابع لامبدا استفاده می‌کنیم؟

قدرت توابع لامبدا زمانی بهتر نشان داده می‌شود که از آن‌ها به عنوان یک تابع ناشناس در یک عملکرد دیگر استفاده می‌شود. به مثال زیر توجه نمایید:

```
>>> def myfunc(n):
...     return lambda a : a * n
>>> mydoubler = myfunc(2)
>>> print(mydoubler(11))
22
```

معمولًاً از توابع لامبدا به عنوان آرگومانی در توابع رده‌بالاتر استفاده می‌شود، برای مثال توابع لامبدا با توابعی مانند filter و map استفاده می‌شوند؛ بنابراین ابتدا این دو تابع را با اختصار تعریف می‌کنیم و سپس به سراغ مثال‌های پیچیده‌تر خواهیم رفت.

map

کار اصلی این تابع اجرای یک لامبدا بر روی یک iterator است برای مثال یک لیست را در نظر بگیرید که می‌خواهیم تمام مؤلفه‌های آن را در دو ضرب کنیم، روش کار به این صورت خواهد بود:

```
>>> numbers=(1,2,3,4,5)
>>> double = lambda x : x * 2
>>> result=list(map(double, numbers))
>>> print(result)
[2, 4, 6, 8, 10]
```



همان‌طور که مشاهده می‌کنیم تابع لامبدا بر روی تمامی مؤلفه‌های لیست با موفقیت اجرا گردید و در خروجی بازگردانده شد، به مثال دیگر توجه کنید:

```
>>> list_1 = [1,2,3,4]
>>> list_2 = [17,12,11,10]
>>> list_3 = [-1,-4,5,9]
>>> list(map(lambda x, y: x+y, list_1, list_2))
[18, 14, 14, 14]
>>> list(map(lambda x, y, z: x+y+z, list_1, list_2, list_3))
[17, 10, 19, 23]
>>> list(map(lambda x, y, z: x+y-z, list_1, list_2, list_3))
[19, 18, 9, 5]
```

همان‌طور که در مثال بالا مشاهده می‌کنید برای توابع لامبایی که چندین ورودی دارند می‌توان چندین لیست را به عنوان ورودی ارسال کرد، البته توجه داشته باشید که اجرای تابع لامبدا به تعداد مؤلفه‌های فهرستی که کمترین تعداد را دارند، محاسبه و بررسی می‌شود.

filter

تابع filter نیز مانند تابع map عمل می‌کند با این تفاوت که امکان بررسی یک شرط را بر روی تمامی مؤلفه‌ها می‌دهد در واقع مقادیری که بازگردانده می‌شود شامل مؤلفه‌هایی است که برای آن‌ها مقدار تابع برابر با «درست» (True) ارزیابی شود، برای مثال می‌خواهیم با استفاده از تابع filter تمامی اعداد بزرگ‌تر از 3 را در یک لیست برگردانیم برای این کار به صورت زیر برنامه خود را می‌نویسیم.

```
>>> numbers=(1,2,3,4,5)
>>> double = lambda x : x > 3
>>> result=list(filter(double, numbers))
>>> print(result)
[4, 5]
```

یا به مثال زیر توجه کنید:

```
>>> numbers = [5, 7, 22, 97, 54, 62, 77, 23, 73, 61]
>>> final_list = list(filter(lambda x: (x%2 != 0), numbers))
>>> print(final_list)
[5, 7, 97, 77, 23, 73, 61]
```

در این مثال همان‌طور که مشاهده می‌کنید اعدادی که بر دو بخش پذیر هستند برگردانده می‌شوند.



decorators

دکوراتورها^۷ دستورالعمل ساده‌ای برای فراخوانی تابع مرتبه بالا را برای ما فراهم می‌کنند؛ به عبارت دیگر دکوراتور تابعی است که تابع دیگر را دریافت می‌کند و رفتار آن را تغییر می‌دهد (بدون آن که ماهیت خود تابع تغییر کند) و آن را برمی‌گرداند. به مثال زیر توجه کنید:

```
def hello_decorator(func):
    def inner1():
        print("Hello, this is before function execution")
        func()
        print("This is after function execution")
    return inner1

def function_to_be_used():
    print("This is inside the function !!")
function_to_be_used = hello_decorator(function_to_be_used)
function_to_be_used()
>>>
Hello, this is before function execution
This is inside the function !!
This is after function execution
```

همان‌طور که مشاهده می‌کنید ما تابع `function_to_be_used` را به عنوان ورودی به تابع `hello_decorator` ارسال کردیم و در تابع دوم اقدام به اجرای آن نمودیم، خروجی به ما نشان می‌دهد که تابع اول با موفقیت اجرا شده است.

برای کار کردن با دکوراتورها روش ساده‌تری نیز وجود دارد، برای این کار می‌توان از کاراکتر `@` استفاده کرد. تابعی که در خط بعد از `@` باید، به عنوان آرگومان به تابعی که در کنار علامت `@` تعریف شده، فرستاده و در آن تغییراتی ایجاد می‌شود. در نظر داشته باشید که در هنگام فراخوانی تابع، تغییرات را مشاهده خواهید کرد. به مثال زیر توجه کنید:



```

def hello_decorator(func):
    def inner1():
        print("Hello, this is before function execution")
        func()
        print("This is after function execution")
    return inner1

@hello_decorator
def function_to_be_used():
    print("This is inside the function !!")

function_to_be_used()
>>>
Hello, this is before function execution
This is inside the function !!
This is after function execution

```

همان‌طور که مشاهده می‌کنید با استفاده از کاراکتر @ توانستیم با موفقیت از دکوراتوری که خودمان نوشته بودیم استفاده کنیم.

حال باید یک مثال کاربردی‌تر بنویسیم و با استفاده از یک دکوراتور زمان اجرای یک برنامه را محاسبه کنیم به کد زیر توجه نمایید:

```

import time
import math

def calculate_time(func):
    def inner1(*args, **kwargs):
        begin = time.time()

        func(*args, **kwargs)
        end = time.time()
        print("Total time taken in : ", func.__name__, end - begin)
    return inner1

@calculate_time
def factorial(num):
    time.sleep(2)
    print(math.factorial(num))

factorial(10)
>>>
628800
Total time taken in : factorial 2.0007894039154053

```



ماژول‌ها

اگر از مفسر پایتون خارج شوید و دوباره وارد آن شوید، تعاریفی که شما انجام داده‌اید (توابع و متغیرها) از بین می‌روند؛ بنابراین، اگر می‌خواهید برنامه‌ای طولانی بنویسید، بهتر است از یک ویرایشگر متن استفاده کنید تا بتوانید آن را به جای ورودی برای مفسر ارسال و اجرا کنید. این کار به اسکریپت نویسی معروف است. با طولانی شدن برنامه شما، ممکن است بخواهید آن را به چندین فایل برای تعمیر و نگهداری آسان‌تر تقسیم کنید. همچنین ممکن است بخواهید بدون کپی کردن توابع آن در هر برنامه، از آن‌ها در برنامه‌های دیگری که نوشته‌اید استفاده کنید.

برای رسیدن به این منظور، پایتون راهی برای قرار دادن تعاریف در یک پرونده و استفاده از آن‌ها در یک اسکریپت یا در یک نمونه تعاملی از مفسر را دارد چنان فایلی یک ماژول نامیده می‌شود. تعاریف از یک ماژول می‌تواند در ماژول‌های دیگر یا در ماژول اصلی استفاده شود. ماژول یک فایل حاوی تعاریف و عبارات پایتون است. نام فایل نام ماژول که پسوند .py به آن اضافه شده است. در درون ماژول، نام ماژول (به عنوان یک رشته) به عنوان متغیر جهانی در name در دسترس است. به عنوان مثال، از ویرایشگر متن مورد علاقه خود برای ایجاد فایلی به نام fibo.py در دایرکتوری فعلی با محتویات زیر استفاده کنید:

```
# Fibonacci numbers module

def fib(n):      # write Fibonacci series up to n
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

def fib2(n):     # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b
    return result
```

حالا وارد مترجم پایتون شوید و این ماژول را با دستور زیر فراخوانی کنید:

```
>>> import fibo
```

این کار ماژول فیبو را در مفسر شما وارد می‌کند و با استفاده از نام ماژول شما می‌توانید به توابعی که در آن ماژول تعریف شده به صورت زیر دسترسی پیدا کنید



```
>>> fibo.fib(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

اگر قصد استفاده از یک تابع را دارید، همچنین می‌توانید آن را به یک نام محلی اختصاص دهید به مثال زیر توجه نمایید

```
>>> fib = fibo.fib
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

اطلاعات بیشتر در مورد ماژول‌ها

یک ماژول می‌تواند شامل دستورات اجرایی و همچنین تعاریف تابع باشد. این عبارات برای شروع اولیه ماژول در نظر گرفته شده است. آن‌ها فقط اولین باری که نام ماژول با یک عبارت import فراخوانی می‌شود اجرا می‌شوند. هر ماژول دارای جدول نماد خصوصی خود است که به عنوان جدول نماد جهانی توسط تمام توابع تعریف شده در ماژول استفاده می‌شود؛ بنابراین، نویسنده یک ماژول می‌تواند متغیرهای جهانی در ماژول بدون نگرانی در مورد درگیری‌های تصادفی با متغیرهای جهانی کاربر استفاده کند. از طرف دیگر، اگر می‌دانید چه کاری انجام می‌دهید، می‌توانید متغیرهای جهانی یک ماژول را به اهمان علامت‌گذاری استفاده شده برای مراجعه به توابع آن، modname.itemname استفاده کنید.

ماژول‌ها می‌توانند ماژول‌های دیگر را فراخوانی کنند. این مرسم است اما لازم نیست که تمام بیانیه‌های import را در ابتدای ماژول (اسکریپت، برای آن موضوع) قرار دهید. نام ماژول وارد شده در جدول نماد جهانی وارد ماژول قرار می‌گیرد.

نوعی از عبارت import وجود دارد که نام‌ها را از ماژول، مستقیماً وارد جدول نماد ماژول وارد کننده می‌کند. مثلاً:

```
>>> from fibo import fib, fib2
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

همچنین روش دیگری نیز برای فراخوانی محتویات یک ماژول وجود دارد که در ادامه به آن‌ها اشاره خواهیم کرد

```
>>> from fibo import *
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```



در این حالت تمامی دستورات شامل کلاس‌ها و توابع درون مازول فراخوانی می‌شوند به جز آن‌هایی که با آندرلاین (–) شروع می‌شوند. در بیشتر موارد، برنامه‌نویسان Python از این روش استفاده نمی‌کنند زیرا مجموعه‌ای از شناسه‌های ناشناخته را در مفسر وارد می‌کند و احتمال این وجود دارد که مواردی را که از قبل تعریف کرده‌اید را بازنویسی کند.

توجه داشته باشید که به‌طورکلی، عمل فراخوانی با استفاده از * از یک مازول یا بسته روشن درستی نیست، زیرا اغلب باعث ناخوانا شدن کد می‌شود. با این حال، استفاده از آن برای صرفه‌جویی در تایپ در جلسات تعاملی اشکالی ندارد.

در صورت دنبال کردن نام مازول با کلمه کلیدی as، مازول فراخوانی شده را به کلمه‌ای که بعد از آن آمده است محدود نماید.

```
>>> import fibo as fib
>>> fib.fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

همچنین می‌تواند در هنگام استفاده از کلمه کلیدی from نیز به صورت مشابه از آن استفاده کرد:

```
>>> from fibo import fib as fibonacci
>>> fibonacci(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

توجه داشته باشید

برای بهره‌وری بهتر، هر یک از مازول تنها یکبار در هر جلسه از مترجم وارد می‌شود؛ بنابراین، اگر شما مازول خود را تغییر دهید، شما باید مفسر را راهاندازی مجدد نمایید، یا اگر می‌خواهید در یک اجرای تعاملی از آن استفاده نمایید می‌توانید از importlib.reload() استفاده نمایید، به عنوان مثال:

```
import importlib; importlib.reload(modulename)
```

اجرای یک مازول به صورت اسکریپت

همچنین می‌توانید مازول پایتون را به صورت یک فایل اسکریپتی اجرا نمایید

```
python fibo.py <arguments>
```

کدهای درون مازول تنها زمانی اجرا می‌شوند که شما آن‌ها را فراخوانی کنید ولی زمانی که متغیر ویژه __name__ برابر با __main__ باشد به این معناست که این کد با فراخوانی مستقیم نیز قابل اجراست، بنابراین بهتر است که کدی شبیه زیر را در انتهای مازول خود اضافه کنید



```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

شما می‌توانید فایل را هم به عنوان یک اسکریپت و هم به عنوان یک ماژول قابل فراخوانی استفاده کنید، زیرا کدی که خط فرمان را تجزیه می‌کند تنها در صورت اجرای ماژول به عنوان پرونده اصلی اجرا می‌شود:

```
$ python fibo.py 50
0 1 1 2 3 5 8 13 21 34
```

تنها به ازای فراخوانی یک ماژول کدی اجرا نخواهد شد

```
>>> import fibo
>>>
```

مسیر جستجوی ماژول

هنگامی که ماژول به نام spam فراخوانی می‌شود، مفسر ابتدا در میان ماژول‌های داخلی آن نام را جستجو می‌کند. در صورتی که یافت نشد، سپس به دنبال فایلی به نام spam.py در فهرستی از دایرکتوری‌هایی توسط متغیر sys.path برگردانده می‌شود جستجو می‌کند. sys.path از این مسیرها آغاز شده است:

- دایرکتوری حاوی اسکریپت ورودی (پوشه فعلی در صورت مشخص نبودن پرونده).
- فهرستی از مسیرها که در متغیر عمومی PYTHONPATH موجود است.
- مسیر پیش‌فرض نصب وابستگی‌ها

ماژول‌های استاندارد

پایتون با کتابخانه‌ای از ماژول‌های استاندارد همراه است که در یک سند جداگانه، مرجع کتابخانه پایتون (مرجع کتابخانه آخرت) توضیح داده شده است. بعضی از ماژول‌ها در مفسر ساخته می‌شوند. این‌ها امکان دسترسی به کارهایی را می‌دهند که جزئی از هسته اصلی زبان نیستند، اما با این وجود برای کارایی ساخته شده‌اند و یا به ابتدایی سیستم‌عامل مانند تماس‌های سیستم دسترسی دارند. مجموعه‌ای از این ماژول‌ها یک گزینه پیکربندی است که همچنین بر روی پلت فرم اصلی بستگی دارد. به عنوان مثال، ماژول winreg فقط در سیستم‌های ویندوز ارائه می‌شود. یک ماژول خاص سزاوار توجه است: sys که در هر مترجم پایتون ساخته شده است. متغیرهای sys.ps1 و sys.ps2 رشته‌های مورد استفاده را به عنوان نوعی اولیه و ثانویه تعریف می‌کنند:



```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'...
>>> sys.ps1 = 'C> '
C> print('Yuck!')
Yuck!
C>
```

این دو متغیر فقط در صورتی که تعریف می‌شوند که مترجم در حالت تعاملی باشد. متغیر sys.path فهرستی از رشته‌هایی است که مسیر جستجوی مترجم را برای ماژول‌ها تعیین می‌کند. در صورتی که PYTHONPATH تنظیم‌نشده باشد، از یک مسیر پیش‌فرض که از متغیر محیط PYTHONPATH گرفته می‌شود، یا از پیش‌فرض داخلی ساخته می‌شود. با استفاده از عملیات لیست استاندارد می‌توانید آن را تغییر دهید:

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```

تابع dir()

دستور داخلی dir() برای یافتن اینکه کدام‌یک از ماژول‌ها را تعریف می‌کند، استفاده می‌شود. این یک لیست مرتب‌شده از رشته‌ها را بر می‌گرداند:



```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__loader__', '__name__',
 '__package__', '__stderr__', '__stdin__', '__stdout__',
 '__clear_type_cache__', '__current_frames__', '_debugmallocstats', '_getframe',
 '__home__', '_mercurial', '_xoptions', 'abiflags', 'api_version', 'argv',
 'base_exec_prefix', 'base_prefix', 'builtin_module_names', 'byteorder',
 'call_tracing', 'callstats', 'copyright', 'displayhook',
 'dont_write_bytecode', 'exc_info', 'excepthook', 'exec_prefix',
 'executable', 'exit', 'flags', 'float_info', 'float_repr_style',
 'getcheckinterval', 'getdefaultencoding', 'getdlopenflags',
 'getfilesystemencoding', 'getobjects', 'getprofile', 'getrecursionlimit',
 'getrefcount', 'getsizeof', 'getswitchinterval', 'getttotalrefcount',
 'gettrace', 'hash_info', 'hexversion', 'implementation', 'int_info',
 'intern', 'maxsize', 'maxunicode', 'meta_path', 'modules', 'path',
 'path_hooks', 'path_importer_cache', 'platform', 'prefix', 'ps1',
 'setcheckinterval', 'setdlopenflags', 'setprofile', 'setrecursionlimit',
 'setswitchinterval', 'setattr', 'stderr', 'stdin', 'stdout',
 'thread_info', 'version', 'version_info', 'warnoptions']
```

بدون آرگومان ورودی تابع `dir()` شناسه‌هایی را که شما در حال حاضر تعریف کرده یا به آن‌ها دسترسی دارید را لیست می‌کند:

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fib = fibo.fib
>>> dir()
['__builtins__', '__name__', 'a', 'fib', 'fibo', 'sys']
```

توجه داشته باشید که انواع مختلفی از شناسه‌ها برگردانده می‌شوند: متغیرها، مازول‌ها، عملکردها و ... تابع `dir()` نام توابع و متغیرهای داخلی را درج نمی‌کند. اگر فهرستی از آن‌ها را می‌خواهید، آن‌ها در مازول‌های استاندارد `builtins` تعریف شده‌اند:



```
>>> import builtins
>>> dir(builtins)
['ArithmetError', 'AssertionError', 'AttributeError', 'BaseException',
 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning',
 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError',
 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning',
 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False',
 'FileExistsError', 'FileNotFoundException', 'FloatingPointError',
 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError',
 'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError',
 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError',
 'MemoryError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented',
 'NotImplementedError', 'OSError', 'OverflowError',
 'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError',
 'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning',
 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError',
 'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError',
 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError',
 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning',
 'ValueError', 'Warning', 'ZeroDivisionError', '...', '_build_class_',
 '__debug__', '__doc__', '__import__', '__name__', '__package__', 'abs',
 'all', 'any', 'ascii', 'bin', 'bool', 'bytearray', 'bytes', 'callable',
 'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits',
 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit',
 'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr',
 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass',
 'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview',
 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property',
 'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice',
 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars',
 'zip']
```

ورودی‌ها و خروجی‌ها در پایتون

برنامه در حقیقت اجرای یکروند منطقی است که در آن، برنامه ورودی‌ها را با توجه به منطق برنامه پردازش کرده و آن‌ها را به عنوان خروجی بازمی‌گرداند پس عملیات ورود و خروج اطلاعات جزئی جداناپذیر از یک برنامه کارآمد است.

در زیر با توابعی که برای این کار در زبان پایتون در نظر گرفته شده‌اند بیشتر آشنا خواهیم شد.
تابع حال ما از دو روش برای نمایش خروجی‌های خود استفاده می‌کردیم یکی استفاده از تابع print,write و دیگری مراجعه به متغیر.

برای تبدیل کردن متغیرها به نوع داده رشته‌ای و نمایش آن‌ها در پایتون دو تابع وجود دارد str و reper فرق این دو تابع در این است که تابع str برمی‌گرداند به صورت قابل فهم برای انسان است این در حالی است که مقداری که تابع reper برمی‌گرداند مقداری است که برای مفسر قابل فهم است. در بسیاری از متغیرها از



قبيل متغيرهای عددی یا ساختارهایی مانند لیستها یا دیکشنری‌ها هر دو تابع مقادیر یکسانی را برمی‌گردانند ولی در بعضی متغیرها از قبیل متغیرهای اعشاری ممکن است مقادیر برگردانده شده باهم فرق کنند به مثال‌های زیر توجه نمایید.

```
f1 = (1.0/7.0)
print str(f1)
print repr(f1)

>>> 0.142857142857
>>> 0.14285714285714285
```

همان‌طور که مشاهده نمودید مقدار برگردانده شده توسط دو تابع مقداری متفاوت است.

```
x = 10 * 3.25
y = 200 * 200
print 'The value of x is ' + repr(x) + ', and y is ' + repr(y) + '...'
print 'The value of x is ' + str(x) + ', and y is ' + str(y) + '...'

>>> The value of x is 32.5, and y is 40000...
>>> The value of x is 32.5, and y is 40000...
```

```
hello = 'hello, world\n'
print str(hello)
print repr(hello)

>>> hello, world
>>> 'hello, world\n'
```

همان‌طور که در خروجی‌ها نمایان است تابع str با کarakتر \n به عنوان یک کarakتر معنی‌دار (خط جدید) رفتار می‌نماید این در حالی است که تابع repr با آن دقیقاً مانند بقیه کarakترها رفتار می‌نماید. همچنین در نمایش رشته‌ها در پایتون شما می‌توانید آن‌ها را فرمت بندی نمایید به مثال‌های زیر توجه نمایید.



```
for x in range(1,11):
    print repr(x).rjust(2), repr(x*x).rjust(3), repr(x*x*x).rjust(4)
```

```
1   1   1
2   4   8
3   9   27
4  16  64
5  25 125
6  36 216
7  49 343
8  64 512
9  81 729
10 100 1000
```

همان‌طور که مشاهده می‌نمایید تابع `rjust` به تعداد مقداری که در ورودی دریافت می‌کند از سمت راست خانه به رشتہ خود اختصاص می‌دهد همچنین تابعی بنام `ljust` نیز وجود دارد که همان کار تابع قبلی را انجام می‌دهد با این تفاوت که خانه‌ها را از سمت چپ اختصاص می‌دهد. همچنین شما می‌توانید به این صورت نیز مشابه کار بالا را انجام دهید.

```
for x in range(1,11):
    print '{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x)
```

```
1   1   1
2   4   8
3   9   27
4  16  64
5  25 125
6  36 216
7  49 343
8  64 512
9  81 729
10 100 1000
```

همان‌طور که مشاهده می‌نمایید خروجی دقیقاً شبیه خروجی کد قبل است. توجه داشته باشید که فاصله‌ای به اندازه یک `space` بین خروجی‌ها در هر خط وجود دارد که توسط تابع `print` اعمال گردیده این تابع در چاپ متغیرها این فاصله را بین هر دو متغیر در نظر می‌گیرد.

همچنین برنامه‌نویس می‌تواند تا از تابع `format` نیز برای مقداردهی متغیرها در یک رشتہ استفاده نماید به مثال زیر توجه نمایید:

```
print '{0} and {1}'.format('First', 'Second')
>>> First and Second
```



همچنین در این تابع برنامه‌نویس می‌تواند با کلمات کلیدی متغیرهای خود را نام‌گذاری و مقداردهی نماید به مثال زیر توجه نمایید:

```
print 'He is {Name} from {Country}'.format(Name='Ali', Country='Iran')

>>> He is Ali from Iran.
```

همچنین برنامه‌نویس می‌تواند تا به صورت ترکیبی نیز از دو روش بالا برای مقداردهی متغیرهایش استفاده نماید به مثال زیر توجه نمایید:

```
print 'The story of {0}, {1}, and {other}'.format('Bill',
'Manfred', other='Georg')

>>> The story of Bill, Manfred, and Georg.
```

همچنین در این مقداردهی برنامه‌نویس می‌تواند تا نوع داده‌ای خود را فرمت بندی نماید برای مثال دستور `r`!تابع `reper()` و دستور `s!` تابع `str()` را بر روی متغیر شما اعمال می‌نماید همچنین برنامه‌نویس می‌تواند با استفاده از کاراکتر `:` نوع فرمت داده ورودی را مشخص نماید به مثال‌های زیر توجه نمایید

```
import math
print 'str = {!s} and repr = {!r}'.format(math.pi, math.pi)

>>> str = 3.14159265359 and repr = 3.141592653589793

import math
print 'The value of PI is approximately {:.3f}'.format(math.pi)

>>> The value of PI is approximately 3.142
```

همان‌طور که مشاهده می‌نمایید در تکه کد بالا تنها ۳ رقم اعشاری اول عدد پی در خروجی نمایش داده شد.

کار کردن با فایل‌ها

کار کردن با فایل‌ها یکی از توانایی‌های مهم هر زبان است بدون ذخیره و بازیابی اطلاعات نرم‌افزارها چیزی جز ماشین‌های محاسبه‌گر نخواهد بود پایتون نیز مانند هر زبان برنامه‌نویسی دیگری این توانایی را دارد تا محتوای فایل‌ها را بخواند یا در آن‌ها بنویسد برای خواندن یک فایل در زبان پایتون برنامه‌نویس می‌تواند از تابع `open(filename, mode)` استفاده نماید که مدهای خواندن یک فایل عبارت‌اند از:

حالتی که در آن فایل قابل خواندن باشد.	<code>r</code>
حالتی که در آن فایل قابل نوشتن باشد. (در صورت وجود فایل همنام اطلاعات بر روی آن فایل بازنویسی می‌شود.)	<code>w</code>



حالتی که در آن نوشتن در انتهای فایل صورت می‌گیرد و به فایل اضافه می‌شود.	a
در سیستم‌عامل ویندوز از این وضعیت برای نوشتن در انتهای فایل‌های باینری مورداستفاده قرار می‌گیرد.	b
حالتی که در آن فایل هم قابلیت نوشتن و هم قابلیت خواندن را دارد.	r+

همچنین مدهای دیگری از قبیل rb, wb,r+b نیز وجود دارد که کارکردشان واضح است به مثال زیر توجه نمایید:

```
f = open('test.txt', 'w')
print f

>>> <open file 'test.txt', mode 'w' at 0x014D2F40>
```

همان‌طور که مشاهده می‌کنید فایل test.txt را در مد نوشتن باز کردیم حال می‌خواهیم با متدهای کار کردن با فایل‌ها بیشتر آشنا شویم.

تمام فایل را به صورت یکجا می‌خواند.	read
یک خط از یک فایل را برمی‌گرداند و کرسر را به سر خط بعد می‌برد.	readline
تمام فایل را می‌خواند و هر خط را در یک خانه از یک آرایه ذخیره می‌کند.	readlines
مقداری را که به عنوان ورودی دریافت کرده در فایل می‌نویسد.	write
ورودی این تابع یک لیست از رشته‌ها است و تمام محتویات آن را در فایل می‌نویسد.	writelines
بعد از اینکه کار با فایل تمام شد برای اینکه تغییرات اعمال شده بر روی فایل اعمال شود	close
کرسر به شماره خانه‌ای می‌برد که از ورودی دریافت کرده.	seek
وضعیت فعلی موقعیت کرسر را نمایش می‌دهد.	tell

به مثال‌های زیر توجه نمایید:



```

f = open('test.txt', 'w')
f.write('Start Of File\n')
f.write('This is a first line\n')
f.write('This is a Second line\n')
f.write('This is a Third line\n')
f.close()
f= open('test.txt', 'a')
f.write('end of File')
f.close()
f= open('test.txt', 'r')
print f.tell()
print f.read()
print f.tell()

>>> 0
>>> Start Of File
>>> This is a first line
>>> This is a Second line
>>> This is a Third line
>>> end of File
>>> 93

```

همان‌طور که مشاهده نمودید در تکه کد بالا ابتدا یک فایل بنام test.txt ساختیم و چندخطی در آن نوشتیم سپس فایل را دوباره باز کردیم و یک خط به انتهای آن اضافه کردیم سپس تمام محتویات فایل را ز فایل خوانده و در خروجی نمایش دادیم همچنین قبل و بعد از خواندن اطلاعات نیز موقعیت کرسر را در خروجی چاپ نمودیم. به مثال‌های بعدی توجه نمایید:

```

set(['orange', 'pear', 'apple', 'banana'])
set(['a', 'r', 'b', 'c', 'd'])
set(['a', 'c', 'z', 'm', 'l'])
set(['r', 'b', 'd'])
set(['a', 'c', 'b', 'd', 'm', 'l', 'r', 'z'])
set(['a', 'c'])
set(['b', 'd', 'm', 'l', 'r', 'z'])

f= open('test.txt', 'r')
print f.tell()
print f.readline()
print f.tell()
print f.readline()
print f.tell()
f.close()

>>> 0
>>> Start Of File
>>> 15
>>> This is a first line
>>> 37

```



همان‌طور که مشاهده می‌کنید تابع readline در هر بار فراخوانی تنها یک خط از فایل را می‌خواند موقعیت کرسر را نیز قبل و بعد از فراخوانی این تابع نمایش داده شده است.

تابع seek دارای دو نوع overload است یکی اینکه شماره خانه‌ای که می‌خواهید کرسر به آن برود (offset) را در ورودی دریافت کند و دیگری اینکه (from_what) را نسبت به موقعیتی که مشخص می‌کنید پیدا کند (offset).

در حالت کلی متغیر دوم سه حالت می‌تواند داشته باشد

0: به این معنی است که از ابتدای فایل (offset) را محاسبه کند.

1: به این معنی است که از موقعیت فعلی کرسر (offset) را محاسبه کند.

2: به این معنی است که از انتهای فایل (offset) را محاسبه کند.

به مثال زیر توجه فرمایید:

```
f= open('test.txt', 'r')
f.read()
print f.tell()
f.seek(-5,2)
print f.tell()
f.seek(12,0)
print f.tell()
f.seek(20,1)
print f.tell()
f.close()

>>> 93
>>> 88
>>> 12
>>> 32
```

بعضی از خواص (Attribute) های کلاس خواندن فایل به قرار زیر می‌باشند

در صورتی که فایل را نبسته باشید مقدار false بر می‌گرداند در غیر این صورت مقدار true بر می‌گرداند.	closed
وضعیت دسترسی فایل را به شما نشان می‌دهد.	mode
نام فایلی که با آن مشغول به کار هستیم را برای ما بر می‌گرداند.	name

به مثال‌های زیر توجه نمایید:



```

f= open("test.txt",'r+')
if f.closed:
    print 'This is not Connected'
else:
    print f.mode
    print f.name
f.close()
if f.closed:
    print 'File Closed'

>>> r+
>>> test.txt
>>> File Closed

```

Json

چیست؟ Json

نشانه‌گذاری شیء جاوااسکریپت^۸ با کوتنه‌نوشت جی‌سان^۹، یک استاندارد باز متنی سبک برای انتقال داده‌ها است به گونه‌ای که برای انسان نیز خوانا باشد. Json از زبان اسکریپتنویسی جاوااسکریپت در نشاندادن ساختمان داده‌های ساده و آرایه‌های انجمنی مشتق شده است. با وجود ارتباط عمیقی که با جاوااسکریپت دارد، Json مستقل از زبان است و مفسرهایش تقریباً برای هر زبانی موجود هستند.

قالب Jdon در ابتدا توسط داگلاس کرافورد مشخص و در RFC4627 شرح داده شده است. نوع رسانه اینترنتی رسمی آن، application/json و پسوند نام پرونده‌های آن .json است.

Json بیشتر برای سریالایز و انتقال ساختمان داده‌ها از طریق ارتباطی شبکه‌ای به کار گرفته می‌شود. بیشترین استفاده آن برای انتقال داده‌ها بین یک کارساز و یک برنامه وبی به عنوان جایگزینی برای اکسامال است.

از ویژگی‌های JSON می‌توان به موارد زیر اشاره نمود:

- داده‌ها در جفت‌های name/value ذخیره می‌شوند
- داده‌ها با ویرگول انگلیسی از هم جدا شده اند
- اشیاء درون Curly braces (کروشه) قرار می‌گیرند (علامت‌های {})
- آرایه‌ها درون براکت‌ها قرار می‌گیرند (علامت‌های [])

نوع‌های داده‌ای، دستور زبان و نمونه
نوع‌های داده‌ای ساده Json عبارت‌اند از:

JavaScript Object Notation⁸
JSON⁹



- اعداد (صحیح یا حقیقی)
- رشته (یونیکدهایی که با «» محصور شده...)
- مقدار بولی (false یا true، درست یا نادرست)
- آرایه (دنباله دارای ترتیبی از مقدارها، جدا شده با ویرگول (,) و محصور شده با «[» و «]»)
- شیء (مجموعه‌ای از جفت‌های کلید مقداری، جدا شده با ویرگول (,) و محصور شده با «{» و «}»)
- کلید می‌بایست که یک رشته باشد)
- تهی یا null

مثال زیر یک شیء در Json است که یک شخص را شرح می‌دهد. در این شیء نوع داده‌ای متنی برای نام و نام خانوادگی، نوع داده‌ای عددی برای سن، یک شیء برای ذخیره نشانی فرد و یک فهرست (یک آرایه) برای ذخیره شماره‌های تلفن شخص است:

```
{
  "firstName": "John",
  "lastName": "Smith",
  "age": 25,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021"
  },
  "phoneNumber": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "fax",
      "number": "646 555-4567"
    }
  ]
}
```

Json در پایتون

برای کار با کتابخانه json احتیاج است که پکیج آنرا از قبل بر روی پایتون خود نصب کرده باشید، این کار را به کمک مدیر بسته های پایتون PIP می‌توانید به راحتی انجام دهید، (البته توجه داشته باشید که از قبل باید PIP را نیز نصب کرده باشید) برای این کار باید دستور زیر را در خط فرمان اجرا کنید.



```
pip install jason
```

بعد از نصب برنامه نویس می تواند با فراخوانی این کتابخانه از مزایای آن استفاده کند، توجه داشته باشید که ساختار dictionary و json در زبان برنامه نویسی پایتون بسیار به یکدیگر شبیه است و ما با استفاده از دوتابع loads و dumps این دو را به یکدیگر تبدیل می کنیم.

loads

از تابع json.loads() برای تبدیل رشته json به یک دیکشنری در پایتون استفاده می کنیم.

dumps

از تابع json.dumps() برای تبدیل یک شی پایتون به رشته json استفاده می کنیم.

به مثال زیر توجه نمایید:

```
import json
# some JSON:
json_content = '{ "name": "John", "age": 30, "city": "New York" }'
# parse x:
content = json.loads(json_content)
# the result is a Python dictionary:
print(content["age"])
# convert into JSON:
json_content = json.dumps(content)
# the result is a JSON string:
print(json_content)
>>>
30
{"name": "John", "age": 30, "city": "New York"}
```

همچنین در پایتون امکان تبدیل نوع های زیر به فرمت Json نیز وجود دارد

- ✚ dict
- ✚ list
- ✚ tuple
- ✚ string
- ✚ int
- ✚ float
- ✚ True
- ✚ False
- ✚ None

در کد زیر تبدیل تمامی این موارد را آزموده ایم، به مثال زیر توجه نمایید:



```

import json
print(json.dumps({"name": "John", "age": 30}))
print(json.dumps(["apple", "bananas"]))
print(json.dumps(("apple", "bananas")))
print(json.dumps("hello"))
print(json.dumps(42))
print(json.dumps(31.76))
print(json.dumps(True))
print(json.dumps(False))
print(json.dumps(None))
>>>
{"name": "John", "age": 30}
["apple", "bananas"]
["apple", "bananas"]
"hello"
42
31.76
true
false
null

```

همچنین توجه داشته باشید که در هنگام تبدیل پایتون به json، اشیا پایتون به معادل خود در جاوا اسکریپت تبدیل می‌شوند، به جدول زیر توجه نمایید:

JSON	پایتون
Object	dict
Array	list
Array	tuple
String	str
Number	int
Number	float
true	True
false	False
null	None

به مثال زیر توجه نمایید:



```

import json
content = {
    "firstName": "John",
    "lastName": "Smith",
    "age": 25,
    "address": {
        "streetAddress": "21 2nd Street",
        "city": "New York",
        "postalCode": "10021"
    },
    "phoneNumber": [
        {
            "type": "home",
            "number": "212 555-1234"
        },
    ]
}
json_content = json.dumps(content)
print("===== JSON =====")
print(json_content)
dictionary_content = json.loads(json_content)
print("===== Dictionary =====")
print(dictionary_content["firstName"])
print(dictionary_content["address"])
print(dictionary_content["phoneNumber"][0]["number"])
>>>
===== JSON =====
{"firstName": "John", "lastName": "Smith", "age": 25, "address": {"streetAddress": "21 2nd Street", "city": "New York", "postalCode": "10021"}, "phoneNumber": [{"type": "home", "number": "212 555-1234"}, {"type": "fax", "number": "646 555-4567"}]}
===== Dictionary =====
John
{'streetAddress': '21 2nd Street', 'city': 'New York', 'postalCode': '10021'}
212 555-1234

```

همان‌طور که در مثال بالا مشاهده می‌کنید تمامی مقادیر تبدیل به معادل json خود شده‌اند.

قالب بندی خروجی

همان‌طور که در بالا ذکر کردیم برای تبدیل یک شی پایتون به رشتہ از تابع `dumps` استفاده می‌کنیم، این تابع پارامترهای دیگری نیز دارد که کار قالب بندی خروجی این تابع را بر عهده دارد، برای مثال `indent` برای مقدار پله‌های فرورفتگی در خروجی را برای قالب بندی مشخص می‌نماید. به مثال زیر توجه نمایید:



```

import json
content = {
    "firstName": "John",
    "lastName": "Smith",
    "age": 25,
    "address": {
        "streetAddress": "21 2nd Street",
        "city": "New York",
        "postalCode": "10021"
    },
}
print(json.dumps(content, indent=4))
>>>
{
    "firstName": "John",
    "lastName": "Smith",
    "age": 25,
    "address": {
        "streetAddress": "21 2nd Street",
        "city": "New York",
        "postalCode": "10021"
    }
}

```

پارامتر بعدی این تابع separators (جدا کننده ها) است. مقدار پیشفرض آن هم (" : ", ",") است؛ یعنی با استفاده از علامت ، (ویرگول انگلیسی) و یک اسپیس اشیاء را از هم جدا کرده و با استفاده از علامت : و یک اسپیس key ها را از value ها جدا میکند. ما میتوانیم این مقدار را به دلخواه خود تغییر دهیم. به مثال زیر توجه نمایید:



```

import json
content = {
    "firstName": "John",
    "lastName": "Smith",
    "age": 25,
    "address": {
        "streetAddress": "21 2nd Street",
        "city": "New York",
        "postalCode": "10021"
    },
}
print(json.dumps(content, indent=2, separators=(". ", " = ")))
>>>
{
    "firstName" = "John".
    "lastName" = "Smith".
    "age" = 25.
    "address" = {
        "streetAddress" = "21 2nd Street".
        "city" = "New York".
        "postalCode" = "10021"
    }
}

```

همان‌طور که مشاهده می‌کنید ما دو پارامتر جدا کننده را با پارامترهای دیگر جایگزین کردیم که در خروجی مشهود است.

پارامتر دیگر در این تابع sort_keys است که مشخص می‌کند نتایج باید به ترتیب خاصی (بر اساس حروف الفبای انگلیسی) نمایش داده شوند (با استفاده از true یا false). به مثال زیر توجه نمایید:



```

import json
content = {
    "firstName": "John",
    "lastName": "Smith",
    "age": 25,
    "address": {
        "streetAddress": "21 2nd Street",
        "city": "New York",
        "postalCode": "10021"
    },
}
print(json.dumps(content, indent=2, sort_keys=True))
>>>
{
    "address": {
        "city": "New York",
        "postalCode": "10021",
        "streetAddress": "21 2nd Street"
    },
    "age": 25,
    "firstName": "John",
    "lastName": "Smith"
}

```

PIP

پایتون نیز مانند هر زبان مهم برنامه‌نویسی دیگری از کتابخانه‌ها و فریمورک‌های شخص ثالث پشتیبانی می‌کند. این کتابخانه‌ها را می‌توانید بر روی پایتون نصب کنید تا در هر پروژه، چرخ را از نو اختراع نکنید! این فریمورک‌ها در یک مخزن بسته‌های پایتون^{۱۰} (PyPI) قابل دسترسی هستند. درواقع pip یک سیستم مدیر بسته^{۱۱} در زبان پایتون و ابزاری است مبتنی بر خط فرمان که از آن برای نصب، حذف، بروز رسانی و در کل مدیریت بسته‌های (یا کتابخانه‌های شخص ثالث) پایتون استفاده می‌گردد. همانند npm برای زبان node و composer برای زبان maven یا php برای زبان جاوا و... .

اگر از پایتون 2.7.9 (و بالاتر) یا پایتون 3.4 (و بالاتر) استفاده می‌کنید، در این صورت PIP به طور پیش‌فرض به همراه پایتون نصب شده است. اما اگر از نسخه‌های قدیمی‌تر استفاده می‌کنید، در این صورت باید خودتان اقدام به نصب آن کنید.

Python Package Index¹⁰
Package manager¹¹



نصب PIP

برای نصب pip لازم است تا فایل get-pip.py را دانلود نمایید. سپس به وسیله دستوری مشابه python get-pip.py در خط فرمان، با سطح کاربری Administrator (در ویندوز) یا root (در گنولینوکس) می‌توانید اقدام به نصب pip نمایید. فراموش نشود، در زمان نصب نیاز به اتصال اینترنت است. البته در سیستم عامل‌های مختلف راه‌های دیگری نیز برای نصب PIP وجود دارد که در ادامه در مورد هر کدام از آن‌ها به اختصار توضیخ می‌دهیم.

نصب در Windows

برای نصب PIP در این سیستم عامل ابتدا get-pip.py را دانلود کرده و سپس با استفاده از دستور زیر آن را اجرا می‌نماییم.

```
> python get-pip.py
[...]
Successfully installed [...]
> pip --version
pip 7.0.1 [...]
```

نصب در Linux

در لینوکس نیز همانند سیستم عامل ویندوز می‌توان با دانلود و اجرای فایل get-pip.py و اجرای دستور زیر اقدام به نصب PIP کرد.

```
user> sudo python3 get-pip.py
[...]
Successfully installed [...]
user> pip3 --version
pip 7.0.1 [...]
```

همچنین می‌توان با استفاده از ابزار مدیریت بسته خود سیستم، PIP را نصب کرد.

Advanced Package Tool (Python 2.x)

```
sudo apt-get install python-pip
```

Advanced Package Tool (Python 3.x)

```
sudo apt-get install python3-pip
```



pacman Package Manager (Python 2.x)

```
sudo apt-get install python3-pip
```

pacman Package Manager (Python 3.x)

```
sudo pacman -S python-pip
```

Yum Package Manager (Python 2.x)

```
sudo yum upgrade python-setuptools
sudo yum install python-pip python-wheel
```

Yum Package Manager (Python 3.x)

```
sudo yum install python3 python3-wheel
```

Dandified Yum (Python 2.x)

```
sudo dnf upgrade python-setuptools
sudo dnf install python-pip python-wheel
```

Dandified Yum (Python 3.x)

```
sudo dnf install python3 python3-wheel
```

Zypper Package Manager (Python 2.x)

```
sudo zypper install python-pip python-setuptools python-wheel
```

Zypper Package Manager (Python 3.x)

```
sudo zypper install python3-pip python3-setuptools python3-wheel
```

دستورات PIP

شما با استفاده از دستورات مدیر بسته می‌توانید تا نصب، حذف و بهروزرسانی کتابخانه‌های پایتون را مدیریت کنید. در ادامه به توضیح متدائلترین دستورات آن خواهیم پرداخت.



Version

برای مشاهده نسخه PIP که در حال حاضر بر روی سیستم شما نصب شده می‌توانید از این دستور استفاده نمایید.

Install

برای نصب یک پکیج جدید از این دستور استفاده می‌شود، ساختار کلی این دستور به صورت زیر است.

```
# pip install [package name]
root> pip install SomePackage
```

البته این دستور آخرین نسخه از این بسته را نصب می‌کند و اگر شما بخواهید تا نسخه خاصی از یک بسته را نصب کنید می‌توانید از ساختار زیر استفاده نمایید:

```
# pip install [package name]==[version]
root> pip install SomePackage==1.0.4
```

همچنین برای نصب تمامی بسته‌هایی که در یک فایل منتهی ذکر گردیده‌اند می‌توان از ساختار زیر استفاده نمایید:

```
# pip install -r [package file name]
root> pip install -r requirements.txt
```

همچنین برای بهروزرسانی یک بسته می‌توان از کلید --upgrade با ساختار زیر استفاده نمود:

```
# pip install --upgrade [package name]
root> pip install --upgrade SomePackage
```

به جای --upgrade می‌توانید از U- نیز استفاده نمایید. همچنین برای نصب مجدد کامل یک بسته نیز می‌توان از کلید --force-reinstall با ساختار زیر استفاده نمود:

```
# pip install --upgrade [package name] --force-reinstall
root> pip install --upgrade SomePackage --force-reinstall
```

Uninstall

برای حذف یک بسته می‌توان از این دستور استفاده نمود، برای این کار می‌توانید از ساختار زیر استفاده نمایید:

```
# pip uninstall [package name]
root> pip uninstall SomePackage
```



List

برای گرفتن لیست تمامی بسته‌های نصب شده می‌توان از این دستور با ساختار زیر استفاده نمود:

```
# pip list
root> pip list
```

همچنین برای نمایش لیست بسته‌هایی که باید به روزرسانی شوند می‌توان از کلید `--outdated` با ساختار زیر استفاده نمود:

```
# pip list --outdated
root> pip list --outdated
```

Freeze

برای گرفتن لیست تمامی بسته‌هایی که توسط PIP نصب شده‌اند می‌توان از این دستور با ساختار زیر استفاده نمود:

```
# pip freeze
root> pip freeze
```

همچنین اگر یک فایل از بسته‌های نصب شده داشته باشد، با زدن کد زیر در کنار پروژه یک فایل متنی بنام requirements ایجاد می‌شود که شامل فهرستی از بسته‌های نصب شده در پروژه داخل این فایل قرار دارد.

```
# pip freeze > [Some file name]
root> pip freeze > requirement.txt
```

Show

برای مشاهده جزئیات یک بسته از این دستور با ساختار زیر استفاده می‌شود:

```
# pip show [package name]
root> pip show SomePackage
```

Search

برای جستجوی بسته‌ها می‌توان از سایت <https://pypi.org> استفاده کرد یا از دستور search با ساختار زیر در خط فرمان استفاده نمود:



```
# pip search [package name]  
root> pip search SomePackage
```

توجه داشته باشید که تمامی دستورات فوق، با ساختار زیر نیز قابل اجرا هستند.

```
#python -m pip [some PIP command]  
> python -m pip install json
```



فصل 2

Object oriented شیء‌گرایی

درواقع در زبان‌های برنامه‌نویسی شیء‌گرایی داستان تازه‌ای نیست. نوعی تفکر یا شیوه در برنامه‌نویسی که ساختار و بلوک اصلی تشکیل‌دهنده برنامه‌ها در آن اشیاء می‌باشد، این نوع نگرش دارای مزایای زیادی از قبیل مدیریت پیچیدگی و هزینه نگهداری کمتر است. در حقیقت شیء‌گرایی یک خصوصیت در یک زبان برنامه‌نویسی نیست حتی در زبانی مانند C که امکانات شیء‌گرایی وجود ندارد نیز می‌توان برنامه‌ای به صورت شیء‌گرا نوشت. زبان برنامه‌نویسی پایتون از جمله زبان‌هایی است که شیء‌گرایی در بطن آن قرار دارد و برنامه‌نویسان با استفاده از این زبان می‌توانند به صورت قدرتمندی اقدام به تولید برنامه‌های شیء‌گرا نمایند.

در این سیستم هر شیء دارای یک سری خصوصیت و قابلیت است که با اصطلاح به آن‌ها Properties و MethodsId می‌گویند:

مفاهیم عمدۀ شیء‌گرایی عبارت‌اند از:

ارثبری (Inheritance)

ارثبری یا وراثت از مفاهیم اساسی برنامه‌نویسی شیء‌گرایی است هر شیء یک نمونه (Instance) از یک کلاس است و هر کلاس می‌تواند از کلاس‌های دیگر مشتق شده باشد بدین معنی که خواص و قابلیت‌های آن‌ها را به ارث برده باشد بدین ترتیب شیء ما تمام خواص آن کلاس را به ارث خواهد برداشت خوبی که در اینجا می‌توان زد خودرو است، تمام خودروها خواصی مانند چرخ، فرمان، موتور یا قابلیت‌هایی مانند حرکت، افزایش سرعت و... را از کلاس پدر خود یعنی خودرو به ارث می‌برند به این ترتیب شما می‌توانید یکشی از خودرو موردنظر خود بسازید که خصوصیت و قابلیت‌های کلاس خودرو را دارا است و سپس خصوصیت موردنظر خود را به آن اضافه کنید و خودروی جدید را با نام تجاری خود به بازار عرضه نمایید.

مخفي سازی (Encapsulation)

کپسوله سازی یا مخفی سازی به معنی مجزا بودن قسمت‌های مختلف یک برنامه از یکدیگر است، در برنامه‌نویسی هرچه اجزای برنامه شما از یکدیگر مستقل‌تر باشند برنامه شما ساختار بهتری خواهد داشت همچنین خطایابی و توسعه آن نیز آسان‌تر خواهد بود برای مثال شما کلید خودروی خود را برمی‌دارید آن را روشن می‌کنید و به سمت محل کار حرکت می‌کنید در طول این پروسه اتفاقات زیادی در خودروی شما رخ خواهد داد، موتور چرخ‌ها را به حرکت درمی‌آورد پمپ بنزین به موتور بنزین می‌رساند و... این در حالی است که این رویدادها کاملاً از هم مستقل می‌باشند به این معنی که پنچر شدن چرخ خودرو شمارا از حرکت بازمی‌دارد



ولی اخلاقی در کار پمپبنزین خودرو شما ایجاد نخواهد کرد حال فکر کنید که شما کنترل تمام این روندها را خودتان به عهده می‌گرفتید یا هنگامی که لاستیک خودرو پنچر می‌شد مجبور بودید تا موتور را نیز تعمیر کنید...

چندریختی (Polymorphism)

چندشکلی یا چندریختی کمیتی است که به یک رابط امکان می‌دهد تا از عملیات یکسانی در قالب یک کلاس عمومی استفاده کند. عمل خاص کلاس را ذات حقیقی شیء تعیین می‌کند. مثال ساده‌ای از چندریختی، فرمان خودرو است. عمل فرمان اتومبیل برای تمام خودروها بدون توجه به سازوکاری که دارند، یکسان است. فرمان برای خودرو که به طور مکانیکی کار می‌کند یا با نیروی برق یا هر چیز دیگری، عمل یکسانی را انجام می‌دهد؛ بنابراین، پس از اینکه شما عملکرد فرمان را یاد گرفتید، می‌توانید فرمان هر خودرویی را کنترل کنید. همین هدف در برنامه‌نویسی نیز اعمال می‌شود. به طور کلی، مفهوم چندریختی، اغلب با عبارت ((یک رابط، چندین روش)) بیان می‌شود. این بدین معنی است که امکان طراحی رابط عمومی برای گروهی از عملیات مرتبط وجود دارد. چندریختی یا چندشکلی (Polymorphism) به این معنا است که اشیاء می‌توانند در موقعیت‌های مختلف، رفتارهای متفاوتی بروز دهند. مثلاً یک تابع در صورتی که بر روی نمونه‌ای از کلاس آ فراخوانی شود، رفتار ب را بروز دهد در حالی که اگر بر روی کلاس ج (که فرزند کلاس آ است) فراخوانی شود، رفتاری متفاوت انجام دهد.

چنانچه سه اصل ارثبری، مخفی سازی و چندریختی به درستی در کنار هم استفاده شوند محیطی مطمئن و پویا را در برنامه شما به وجود خواهند آورد که هم کارایی بالایی خواهد داشت هم استحکام بیشتری خواهد داشت. در حقیقت از طریق کاربرد اصول شیءگرا، بخش‌های گوناگون یک برنامه^۰ پیچیده را می‌توان ترکیب نمود و موجودیتی یکپارچه، با استحکام و قابل مدیریت تشکیل داد.

شیءگرایی در پایتون

شیءگرایی در برنامه‌نویسی دقیقاً مانند کار با اشیا در دنیای واقعی است. ساختاری که وظیفه تعریف اشیاء در زبان برنامه‌نویسی را دارند کلاس نامیده می‌شوند در حقیقت اشیاء در زبان برنامه‌نویسی به صورت کلاس تعریف می‌شوند بنابراین برنامه‌نویس برای اینکه بتواند به صورت شیءگرا برنامه‌ای را بنویسد باید بر ساختار و مفاهیم کلاس به طور کامل مسلط باشد.

در پایتون تمام اجزای اصلی زبان به صورت اشیا پیاده‌سازی شده‌اند. از متغیرها و توابع و انواع داده‌های اصلی مثل int بگیرید تا خود کلاس‌ها. بله حتی خود کلاس‌ها هم به صورت اشیا تعریف شده‌اند. طبق تصمیماتی که مبنی بر یکپارچه ساختن اشیای زبان و اشیای تعریف شده از طرف برنامه‌نویسان صورت پذیرفت، تمام اشیای اصلی زبان پایتون از یکشی واحد و برتر به نام object ساخته می‌شوند و یا به عبارت دیگر ارث می‌گیرند. این مسئله چند سالی است که در پایتون اتفاق افتاده است اما به علت عادت قدیمی برنامه‌نویسان به کلاس‌های



عادی و همچنین نبود مستندات کافی برای تشریح کلاس‌های جدید، هنوز هم که هنوز است کلاس‌های سبک جدید فراگیر نشده‌اند.

کلاس

کلاس نقشه نوعی و مشترک برای گروهی از اشیاء است که ویژگی‌های مشترکی دارند و رفتارهای مشترکی از خود نشان می‌دهند درواقع کلاس‌ها روند منطقی می‌باشند که برنامه‌نویس برای حل مسائل در دنیای واقعی آن‌ها را طراحی می‌نماید.

تعریف کلاس در زبان پایتون بسیار ساده است تنها کافی است برنامه‌نویس از کلمه کلیدی class استفاده نماید و بعداز آن نام کلاس را با رعایت قواعد نام‌گذاری متغیرها ذکر کند و سپس بدنه کلاس را تعریف نماید از لحاظ فنی این تمام‌کاری است که برنامه‌نویس برای تعریف یک کلاس باید انجام دهد به مثال زیر توجه نمایید:

```
class FirstClass(object):
    def FirstMethod(self):
        return 'First Class Definitions'

x= FirstClass()
print (x.FirstMethod())

>>> First Class Definitions
```

همان‌طور که مشاهده نمودید یک کلاس با نام FirstClass تعریف کردیم و در این کلاس نیز متده با نام FirstMethod تعریف نمودیم توجه نمایید که کلمه self به معنی ارجاع به خود کلاس است، وجود کلمه کلیدی self در تعریف توابع درون کلاس الزامی است به این مثال نیز توجه نمایید:

```
class FirstClass(object):
    def FirstMethod(self,number):
        return 'new number is = {!s} '.format(number)

x= FirstClass()
print (x.FirstMethod(12))

>>> new number is = 12
```

همان‌طور که مشاهده نمودید برای متده که تعریف کردیم یک مقدار ورودی نیز در نظر گرفتیم که آن را برای ما در خروجی برمی‌گرداند اگر به نظرتان این مثال‌ها بسیار ساده است زیاد نگران نباشید در آینده مثال‌های پیچیده‌تری خواهیم زد.

متده int()

در پایتون چیزی به نام سازنده یا (Constructor) وجود ندارد اما تابع فوق تمامی نیازهای برنامه‌نویس را به عنوان یک سازنده تأمین می‌نماید به این صورت که در هنگام تعریف یک نمونه از کلاس این متده به صورت خودکار



فراخوانی می‌شود و برنامه‌نویس می‌تواند تنظیمات مربوط به کلاس خود را در هنگام تعریف یک نمونه از آن کلاس اعمال نماید به مثال زیر توجه فرمایید:

```
class FirstClass(object):
    def __init__(self, variable1, variable2):
        self.f = variable1
        self.s = variable2

x = FirstClass("first", "seconde")
print(x.f, x.s)

>>> first seconde
```

همان‌طور که در مثال‌های قبل مشاهده کردید برنامه‌نویس برای اینکه بتواند از یک کلاس استفاده نماید ابتدا باید یک نمونه از آن را بسازد ساختن یک نمونه از کلاس بسیار ساده است تنها کافی است برنامه‌نویس یک متغیر را با نام کلاس مقداردهی نماید. باید توجه داشت که نمونه‌های مختلف از یک کلاس هیچ اختلالی در کار یکدیگر ایجاد نمی‌کنند، بنابراین برنامه‌نویس می‌تواند تا چندین نمونه از یک کلاس را بسازد و از آن‌ها استفاده نماید بدون اینکه مشکلی در برنامه‌اش به وجود آید.

ارثبری در پایتون

در مورد ارثبری یا وراثت در ابتدای این بخش به صورت کامل صحبت کردیم همان‌طور که حتماً متوجه شده‌اید، وراثت به معنای دارا بودن خصوصیات کلاس والد است همان‌طور که ممکن است یک فرزند خصوصیاتی مانند رنگ مو یا چشم خود را از والدین خود به ارث ببرد.

اعمال وراثت در پایتون بسیار راحت است قبلاً نیز در مثال‌های مربوط به این بخش این کار را کرده‌ایم اگر توجه کرده باشید در مثال‌های قبل هنگام تعریف کلاس از کلمه کلیدی object استفاده می‌کردیم این بدان معناست که کلاس ما فرزند کلاس object است و تمام خصوصیات آن را به ارث می‌برد (البته این روش جدیدی در تعریف کلاس‌های پایتون به حساب می‌آید این بدان معناست که شما می‌توانید از روش قدیمی در تعریف کلاس پایتون استفاده کنید) به مثال زیر توجه نمایید:



```

class Parent(object):
    def ParentSay(self):
        return "Parent Class"

class FirstChild(Parent):
    def ChildSay(self):
        return "Child Class"

child = FirstChild()
print (child.ParentSay())
print (child.ChildSay())

>>> Parent Class
>>> Child Class

```

همان‌طور که مشاهده نمودید در مثال قبل دو کلاس بنام‌های Parent و FirstChild تعریف نمودیم که کلاس دوم فرزند کلاس اول است در ادامه یک نمونه از کلاس FirstChild ساختیم و با استفاده از وراثت با خصوصیات و قابلیت‌های کلاس والد ارتباط برقرار کردیم.

همچنین در زبان پایتون این امکان وجود دارد تا ارثبری به صورت چندگانه بر روی کلاس‌ها اعمال گردد البته باید توجه داشت که نوع تعریف کلاس شما در روند استفاده از خصوصیات و قابلیت‌ها در ارثبری متفاوت است. در نوع قدیمی تعریف کلاس به صورت depth-first از سمت چپ به راست بدین صورت که اگر برنامه‌نویس خاصیت یا قابلیتی را صدا بزند کامپایلر پایتون اولین کلاس والد از سمت چپ را بررسی می‌نماید و همین‌طور به صورت بازگشتی کلاس‌های والد آن کلاس والد را نیز بررسی می‌نماید و تنها زمانی به سراغ والد بعدی می‌رود که خصوصیت یا قابلیت مورد نظر را پیدا نکند و همین‌طور الی آخر...

در کلاس‌های نوع جدید رویکرد تفکیک بر اساس تغییرات به صورت پویا انجام می‌شود تا از فراخوانی چندگانه کلاس () (این متدهایی که به صورت نماینده‌ای از متدهای فراخوانی شده برای والد یا یک کلاس همان‌نگاره است. این متدهای فراخوانی ممکن است یک کلاس لغو (override) شده‌اند بسیار مفید خواهد بود – این متدهای فراخوانی که در یک کلاس از این‌گونه شده‌اند بسیار در زبان‌های دیگری با پشتیبانی از ارثبری چندگانه به عنوان فراخوانی متدهای call-next-method (call-next-method) شناخته می‌شود که بسیار قدرتمندتر از ارثبری منفرد است.

در کلاس‌های نوع جدید مرتب‌سازی پویا الزامی است زیرا تمام اجزا در ارثبری چندگانه به صورت یک یا چند رابطه نمایش داده خواهند شد (جایی که یکی از پایین‌ترین کلاس‌های والد می‌تواند با مسیرهای چندگانه به کلاس‌های بالاتر دسترسی داشته باشد) برای مثال تمام کلاس‌های نوع جدید فرزند کلاس object می‌باشند بنابراین هر یک از اجزاء در ارثبری چندگانه بیشتر از یک مسیر را برای دسترسی به کلاس object تهیه می‌کنند.



لغو متده (override)

چنانچه نام یک متده در کلاس فرزند با نام یک متده در کلاس های والدش یکسان باشد در این صورت می گوییم که کلاس موجود در کلاس فرزند کلاس موجود در کلاس والد را لغو (override) کرده است زمانی که یک متده لغو شده در یک کلاس فراخوانی می شود همیشه از نگارش تعریف شده در کلاس فرزند استفاده می شود. اما گاهی برنامه نویس نیاز دارد تا دقیقاً به کلاس لغو شده در کلاس والد دسترسی داشته باشد برای این کار نیز راه حلی وجود دارد به مثال زیر توجه نمایید:

```
class Parent(object):
    def printParent(self):
        print ("Parent")

class Child(Parent):
    def refrencetoParent(self):
        Parent.printParent(self)

    def printParent(self):
        print ("Child")

child = Child()
child.refrencetoParent()
child.printParent()

>>>
Parent
Child
```

همان طور که در مثال بالا مشاهده نمودید تابع printParent در کلاس Child لغو (override) گردیده ولی با استفاده از تابع refrencetoParent امکان دسترسی به آن را ممکن نمودیم. همچنین در پایتون این امکان وجود دارد تا متده که خارج از کلاس تعریف شده را در داخل یک کلاس استفاده نمود به طوری که تمام خواص آن کلاس را با خود همراه داشته باشد به مثال زیر توجه نمایید:



```

def DefinedOutside (self, x, y):
    return min(x, x+y)

class Class:
    Dinside = DefinedOutside
    def hello(self):
        return 'hello world'
    h = hello

instance = Class ()
print (instance.Dinside(2,-3) )
print (instance.hello ())
print (instance.h())

>>>
-1
hello world
hello world

```

همان‌طور که مشاهده نمودید سه متده hello, Dinside و h زیرمجموعه‌های کلاس Class می‌باشند که به صورت متده از این کلاس قابل‌دستیابی می‌باشند. البته انجام چنین کارهایی تنها کد شمارا گنج و ناخوانا می‌کند ولی این قابلیت در پایتون وجود دارد.

توابع و متغیرهای خصوصی¹²

در زبان پایتون متغیرها و توابع به صورت پیش‌فرض به صورت عمومی¹³ تعریف می‌شوند به این معنی که شما با ساخت یک نمونه از کلاس یا در صورت فراخوانی ایستا¹⁴ این امکان وجود دارد که به توابع و متغیرها دسترسی داشته باشید و اگر برنامه‌نویس بخواهد از دسترسی به توابع یا متغیرها جلوگیری نماید باید از قوانینی که برای این کار در پایتون تعییشده پیروی کند.

در پایتون با اضافه کردن یک زیرخط¹⁵ به ابتدای متغیر یا تابع به دیگران اعلام می‌کنیم که این متغیر یا تابع یک متغیر محافظت‌شده¹⁶ است، البته وقتی متغیری را به این صورت تعریف می‌کنیم مانند زبان‌های برنامه‌نویسی دیگر این اتفاق به صورت واقعی رخ نمی‌دهد و امکان دسترسی از طریق شیء همچنان وجود دارد، در واقع متغیرهای خصوصی که از جایی غیر از درون یک شیء قابل‌دسترس نباشند، در پایتون وجود ندارد، ولی اضافه کردن زیرخط به صورت یک قرارداد به دیگر برنامه‌نویسان اعلام می‌کند که نباید خارج از کلاس از این متغیر یا تابع استفاده شود و باید به عنوان جزئیات پیاده‌سازی در نظر گرفته شود و نباید بدون اطلاع قبلی تغییر کند. به مثال زیر توجه کنید:

private¹²
public¹³
static¹⁴
underline¹⁵
protected¹⁶



```
>>> class Test(object):
...     def __protected_method(self):
...         print("__protected_method called")
...     def public_method(self):
...         return self.__protected_method()
...
>>> x = Test()
>>> x.public_method()
__protected_method called
>>> x.__protected_method()
__protected_method called
```

همان‌طور که مشاهده می‌کنید با اینکه برای تابع `_protected_method` از قانون یک زیرخط در ابتدا استفاده کردیم با این حال توانستیم به سادگی این تابع را فراخوانی کنیم.

تغییرات نام (*name mangling*)

از آنجایی که یک کاربرد اصلی و مهم برای خصوصی کردن متغیرها و توابع وجود دارد و آن جلوگیری از تداخل نام‌ها بانام‌های تعریف شده در زیر کلاس‌ها است لذا مکانیسم تغییر نام^{۱۷} در زبان پایتون تعبیه گردیده به این صورت که اگر قبل از نام یک کلاس از دو زیرخط استفاده شود نام متغیر یا تابع در مفسر به صورت

`_classname__spam`

نام‌گذاری می‌شود که `classname` نام کلاس جاری است که خط زیرین قبل از آن حذف شده است. تغییرات نام برای امکان‌پذیر کردن تغییر تابع توسط زیر کلاس‌ها، بدون شکست در فراخوانی‌های تابع درون-کلاسی مفید هستند. برای مثال:

Name mangling¹⁷



```

>>> class Test_class(object):
...     def __private_method(self):
...         print("__private_method called")
...     def __protected_method(self):
...         print("__protected_method called")
...     def call_protected_method(self):
...         return self.__protected_method()
...     def call_private_method(self):
...         return self.__private_method()
...
>>> x = Test_class()
>>> x.call_protected_method()
__protected_method called
>>> x.__protected_method()
__protected_method called
>>> x.call_private_method()
__private_method called
>>> x.__private_method()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Test_class' object has no attribute '__private_method'

```

همان‌طور که مشاهده می‌کنید تابعی که با دو زیرخط تعریف شده است در هنگام فراخوانی با خطای مواجه می‌شود، قابلیت فراخوانی به این صورت را ندارد. بباید برنامه را به صورت دیگری پیاده‌سازی کنیم

```

>>> class Test_class(object):
...     def __private_method(self):
...         print("__private_method called")
...     def __protected_method(self):
...         print("__protected_method called")
...     def call_protected_method(self):
...         return self.__protected_method()
...     def call_private_method(self):
...         return self.__private_method()
...
>>> x = Test_class()
>>> dir(x)
['__Test_class__private_method', '__class__', '__delattr__', '__dict__',
 '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
 '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__',
 '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
 '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__',
 '__protected_method', 'call_private_method', 'call_protected_method']
>>> x.__Test_class__private_method()
__private_method called

```

همان‌طور که در مثال بالا مشاهده می‌کنید ابتدا با استفاده از متدهای موجود در کلاس را مشاهده کردیم، همان‌طور که در خروجی این تابع مشاهده می‌کنید نام تابع `__private_method` به `__Test_class__private_method` تغییر نام داده است و از طریق این نام قابل فراخوانی است ولی از طریق نامی که در کلاس به آن اختصاص داده شده است امکان فراخوانی وجود ندارد، در واقع در زبان پایتون به



معنای واقعی امکان خصوصی کردن متغیرها و کلاس‌ها وجود ندارد ولی به صورت قراردادی تا حدودی این موضوع شبیه‌سازی شده است.

iterators تکرارشونده‌ها

همان‌طور که در بخش ساختارهای تکرار مشاهده نمودید امکان پیمایش در اکثر داده‌های استاندارد امکان‌پذیر است به مثال‌های زیر توجه کنید:

```
for element in [1, 2, 3]:
    print(element)
for element in (1, 2, 3):
    print(element)
for key in {'one':1, 'two':2}:
    print(key)
for char in "123":
    print(char)
for line in open("myfile.txt"):
    print(line, end='')
```

در تمامی این حالت‌ها برنامه با موفقیت کار می‌کند و تمامی این متغیرها قابل پیمایش هستند. ولی در پشت‌صحنه چه اتفاقی رخ می‌دهد، درواقع تمامی این داده‌ها قابل تکرار^{۱۸} هستند، متغیرهای قابل تکرار را می‌توان به متغیرهای تکرارشونده تبدیل کرد برای این کار از متده استفاده می‌شود، سپس با متده next می‌توان به تمامی عناصر شیء قابل تکرارشونده دسترسی داشت، به مثال زیر توجه نمایید:

```
>>> iterable = 'PYT'
>>> iterator = iter(s)
>>> iterator
<str_iterator object at 0x0000027938070550>
>>> next(iterator)
'P'
>>> next(iterator)
'Y'
>>> next(iterator)
'T'
>>> next(iterator)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

همان‌طور که مشاهده می‌کنید در کد بالا ابتدا ما متغیر قابل تکرار iterable را به یک متغیر تکرارشونده به نام iterator تبدیل کردیم سپس با استفاده از متده next امکان پیمایش در متغیر تکرارشونده تا انتهای آن وجود داشت.

iterable¹⁸



در حلقه for نیز همین اتفاق رخ می‌دهد به این صورت که ابتدا با استفاده از متدهای `next` و `iter` قابل تکرار به متغیر تکرارشونده تغییر پیدا می‌کند سپس با متدهای `next` انتهای متغیر پیمایش می‌شود و زمانی که دیگر عضوی برای پیمایش نباشد خطای `StopIteration` فراخوانی می‌شود. به مثال زیر توجه نمایید:

```
>>> iterable = (1,5,7,9,12,78,56,12)
>>> iterator = iter(iterable)
>>> while True:
...     try:
...         item = next(iterator)
...     except StopIteration:
...         break # Iterator exhausted: stop the loop
...     else:
...         print(item)
...
1
5
7
9
12
78
56
12
```

در مثال بالا ما با استفاده از مکانیسمی که قبل تر توضیح داده بودیم توانستیم به وسیله حلقه `while` حلقه for را شبیه‌سازی کنیم.

همچنین با استفاده از داندر متدهای `__iter__` و `__next__` می‌توانیم خاصیت iterable بودن را به کلاس‌های خود اضافه کنیم، به مثال زیر توجه نمایید:



```

>>> class Reverse:
...     """Iterator for looping over a sequence backwards."""
...     def __init__(self, data):
...         self.data = data
...         self.index = len(data)
...
...     def __iter__(self):
...         return self
...
...     def __next__(self):
...         if self.index == 0:
...             raise StopIteration
...         self.index = self.index - 1
...         return self.data[self.index]
...
>>> rev = Reverse('spam')
>>> print(iter(rev))
<__main__.Reverse object at 0x00000279380706D8>
>>>
>>> for char in rev:
...     print(char)
...
m
a
p
s

```

همان‌طور که مشاهده می‌کنید با استفاده از دو داندر متده فوق موفق شدیم کلاس خود را تبدیل به یک شیء قابل تکرار که در حلقه for قابل پیمایش باشد نماییم.

ژنراتور generator

ژنراتورها^{۱۹} یا سازنده‌ها ابزار ساده و قدرتمندی برای ساخت قابل تکرارها هستند به این صورت که می‌توان آن‌ها را به تکرارشونده‌ها تبدیل کرد و سپس با استفاده از متده next در آن‌ها پیمایش انجام داد. ساختار کلی ژنراتورها مانند ساختار یک متده معمولی است با این تفاوت که در توابع معمولی برای برگرداندن اطلاعات از return استفاده می‌شود ولی در ژنراتورها از yield استفاده می‌شود. (دستور yield به این صورت عمل می‌کند که تابع را متوقف و وضعیت آن را ذخیره می‌کند تا در آینده از همان جایی که متوقف شده بود به کار خود ادامه دهد.) به مثال زیر توجه نمایید:

¹⁹ generator



```

>>> def test_generator(l):
...     total = 0
...     for n in l:
...         total += n
...         yield total
...
>>> generator = test_generator([15,10, 7])
>>> print(generator)
<generator object test_generator at 0x0000027937F3C830>
>>> print(next(generator))
15
>>> print(next(generator))
25
>>> print(next(generator))
32

```

می‌بینیم که به ازای هر بار فراخوانی متده next، مقدار جدیدی به ما نمایش داده می‌شود. اگر دوباره تابع را فراخوانی کنیم، خطای StopIteration دریافت خواهد کرد.

```

>>> print(next(generator))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration

```

همچنین می‌توان ژنراتورها به صورت زیر با نگارشی دیگر نیز پیاده‌سازی کرد:

```

>>> generator = (x for x in ['a', 'b', 'c', 'd'])
>>> print(generator)
<generator object <genexpr> at 0x0000027937F3C9E8>
>>> print(next(generator))
a
>>> print(next(generator))
b
>>> print(next(generator))
c
>>> print(next(generator))
d
>>> print(next(generator))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration

```

همان‌طور که در مثال بالا مشاهده می‌کنید خروجی کد فوق یک ژنراتور است.

مقایسه ژنراتور و لیست

در کد زیر مقایسه‌ای بین حافظه‌ی اشغال شده بین list و generator خواهیم داشت.



```
>>> import sys
>>> temp_generator = (i * 2 for i in range(100000))
>>> temp_list = [i * 2 for i in range(100000)]
>>>
... print("Generator Size : ", sys.getsizeof(temp_generator))
Generator Size : 88
>>> print("List Size : ", sys.getsizeof(temp_list))
List Size : 824464
```

همان‌طور که مشاهده می‌کنید، مصرف حافظه در ژنراتورها بسیار کمتر از لیست است. اتفاقی که در اینجا رخ می‌دهد این است که به جای اینکه تمام داده‌ها در حافظه قرار گیرد، فقط آن قسمتی که نیاز است در حافظه می‌گیرد.

با اینکه ژنراتورها حافظه‌ی کمی مصرف می‌کنند ولی باعث کند شدن اجرای کد نیز می‌شوند. به مثال زیر توجه کنید:

```
>>> import cProfile
>>> cProfile.run("sum((i * 2 for i in range(10000000)))")
    10000005 function calls in 2.283 seconds

Ordered by: standard name

      ncalls  tottime  percall  cumtime  percall filename:lineno(function)
100000001      1.164    0.000      1.164    0.000 <string>:1(<genexpr>)
            1    0.000    0.000      2.283    2.283 <string>:1(<module>)
            1    0.000    0.000      2.283    2.283 {built-in method builtins.exec}
            1    1.119    1.119      2.283    2.283 {built-in method builtins.sum}
            1    0.000    0.000      0.000    0.000 {method 'disable' of
'_lsprof.Profiler' objects}

>>> cProfile.run("sum([i * 2 for i in range(10000000)])")
      5 function calls in 1.405 seconds

Ordered by: standard name

      ncalls  tottime  percall  cumtime  percall filename:lineno(function)
            1    0.958    0.958      0.958    0.958 <string>:1(<listcomp>)
            1    0.137    0.137      1.405    1.405 <string>:1(<module>)
            1    0.000    0.000      1.405    1.405 {built-in method builtins.exec}
            1    0.309    0.309      0.309    0.309 {built-in method builtins.sum}
            1    0.000    0.000      0.000    0.000 {method 'disable' of
'_lsprof.Profiler' objects}
```

همان‌طور که مشاهده می‌کنید 2.283 ثانیه زمان برد تا ژنراتور اجرا شود ولی با لیست به زمان 1.405 ثانیه رسیدیم. (البته توجه داشته باشید که دستگاهی که من کد را بر روی آن اجرا کردم سیستم مناسبی است و ممکن است زمان در سیستم شما عدد متفاوتی باشد.)



متدهای داندر یا متدهای جادویی و بازنویسی اپراتورها

متدهای داندر یا جادویی در پایتون متدهایی هستند که دارای دو پیشوند و پسوند آندرلاین (_) هستند. Dunder در اینجا به معنی "Double Under (Underscores)" است. این‌ها معمولاً برای بازنویسی اپراتورها استفاده می‌شوند. چند نمونه از متدهای جادویی عبارت‌اند از: __init__, __add__, __len__, __repr__ و ... متدهای شوند. چند نمونه از متدهای جادویی فراخوانی وقتی نمونه‌ای از کلاس ایجاد می‌شود __init__ به عنوان سازنده (Constructor) بدون هرگونه فراخوانی وقتی نمونه‌ای از کلاس ایجاد می‌شود اجرا می‌شود، مانند سازنده‌ها در برخی از زبان‌های برنامه‌نویسی دیگر مانند C++, Java, C# و PHP و ... به کمک این متدها است که ما می‌توانیم دو رشته را با استفاده از اپراتور + به یکدیگر الحقق کنیم و یا برای متغیرهای عددی به اهمان اپراتور قبلی دو عدد را باهم جمع کنیم.

در اینجا یک پیاده‌سازی ساده را می‌بینیم:

```
# declare our own string class
class String:

    # magic method to initiate object
    def __init__(self, string):
        self.string = string

    # Driver Code
    if __name__ == '__main__':
        # object creation
        string1 = String('Hello')

        # print object location
        print(string1)

OutPut :
<__main__.String object at 0x7fe992215390>
```

خروجی این کد آدرس حافظه شیء کلاس String را چاپ می‌کند. بباید یک متدهای __repr__ برای نمایش شیء خود اضافه کنیم (در مورد اینکه متدهای __repr__ چه کاری می‌کند و چه زمانی فراخوانی می‌شود بعداً بیشتر توضیح خواهیم داد ولی در حال حاضر تنها کافی است بدانید که این متدهای زمانی فراخوانی می‌شود که شما بخواهید یک نمونه از کلاس را print نمایید).



```
# declare our own string class
class String:

    # magic method to initiate object
    def __init__(self, string):
        self.string = string

    # print our string object
    def __repr__(self):
        return 'Object: {}'.format(self.string)

# Driver Code
if __name__ == '__main__':

    # object creation
    string1 = String('Hello')

    # print object location
    print(string1)

OutPut :
Object: Hello
```

اگر سعی کنیم رشته‌ای به آن اضافه کنیم:

```
# declare our own string class
class String:

    # magic method to initiate object
    def __init__(self, string):
        self.string = string

    # print our string object
    def __repr__(self):
        return 'Object: {}'.format(self.string)

# Driver Code
if __name__ == '__main__':

    # object creation
    string1 = String('Hello')

    # concatenate String object and a string
    print(string1 + ' world')

OutPut :
TypeError: unsupported operand type(s) for +: 'String' and 'str'
```

اکنون متد `__add__` را به کلاس `String` اضافه کنید:



```

# declare our own string class
class String:

    # magic method to initiate object
    def __init__(self, string):
        self.string = string

    # print our string object
    def __repr__(self):
        return 'Object: {}'.format(self.string)

    def __add__(self, other):
        return self.string + other

# Driver Code
if __name__ == '__main__':

    # object creation
    string1 = String('Hello')

    # concatenate String object and a string
    print(string1 + ' Geeks')

```

OutPut :

Hello Geeks

در ادامه تعدادی از متدهای داندر و اپراتورهای معادل آنها به اختصار معرفی شده‌اند.

سفارشی سازهای اولیه

یک متداشت است که در هنگامی که یک نمونه از کلاس ساخته می‌شود اجرا می‌شود (لازم به ذکر است که این متداشت قبل از متداشت `__init__` فراخوانی می‌شود)

`object.__new__(cls[, ...])`

بعدازاینکه یک نمونه از کلاس ساخته شد و قبل از ارسال نمونه به متغیری که نمونه‌سازی را آغاز کرده فراخوانی می‌شود.

`object.__init__(self[, ...])`

هنگامی که نمونه در حال نابودی است فراخوانی می‌شود. به این روش نهایی یا ویرانگر²¹ نیز گفته می‌شود.

static²⁰
Destructor²¹



`object.__del__(self)`

این متده با فراخوانی تابع درونی `repr()` اجرا میشود و کارشن نمایندگی یک شیء بهصورت رسمی درزمانی است که بهصورت یکرسته فراخوانی میشود. به زبان سادهتر زمانی که برنامهنویس با یک نمونه از کلاس بهصورت رشته بخورد میکند این متده فراخوانی میشود.

`object.__repr__(self)`

این متده با فراخوانی تابع `str(object)` یا توابع درونی `format()` و `print()` برای محاسبه غیررسمی و یا به زبان سادهتر نمایندگی یک شیء درزمانی که برنامهنویس اقدام به چاپ شیء مینماید. مقدار برگردانده شده توسط این شیء باید از نوع رشته باشد.

`object.__str__(self)`

`_repr_` و `_str_`
به مثال زیر توجه نمایید

```
>>> import datetime
>>> now = datetime.datetime.now()
>>> str(now)
'2020-04-06 11:17:34.654487'
>>> repr(now)
'datetime.datetime(2020, 4, 6, 11, 17, 34, 654487)'
```

`_bytes_`

`_format_`

`_hash_`

`_bool_`

سفارشی کردن دسترسی ویژگی

- `__getattr__`
- `__getattribute__`
- `__setattr__`
- `__delattr__`
- `__dir__`



۲۲ پیاده‌سازی توصیف گرها

```
__get__
__set__
__delete__
__set_name__
```

Customizing instance and subclass checks

```
__instancecheck__
__subclasscheck__
```

Emulating container types

```
__len__
__length_hint__
__getitem__
__setitem__
__delitem__
__missing__
__iter__
__reversed__
__contains__
```

متدها	اپراتور
object.__add__(self, other)	+
object.__sub__(self, other)	-
object.__mul__(self, other)	*
object.__floordiv__(self, other)	//
object.__truediv__(self, other)	/
object.__mod__(self, other)	%
object.__pow__(self, other[, modulo])	**
object.__lshift__(self, other)	<<
object.__rshift__(self, other)	>>
object.__and__(self, other)	&
object.__xor__(self, other)	^
object.__or__(self, other)	

همچنین Extended Assignments

متدها	اپراتور
object.__iadd__(self, other)	+=
object.__isub__(self, other)	-=
object.__imul__(self, other)	*=



object.__idiv__(self, other)	/=
object.__ifloordiv__(self, other)	//=
object.__imod__(self, other)	%=
object.__ipow__(self, other[, modulo])	**=
object.__ilshift__(self, other)	<<=
object.__irshift__(self, other)	>>=
object.__iand__(self, other)	&=
object.__ixor__(self, other)	^=
object.__ior__(self, other)	=
object.__iadd__(self, other)	+=

همچنین Unary Operators

متدها	اپراتور
object.__neg__(self)	-
object.__pos__(self)	+
object.__abs__(self)	abs()
object.__invert__(self)	~
object.__complex__(self)	complex()
object.__int__(self)	int()
object.__long__(self)	long()
object.__float__(self)	float()
object.__oct__(self)	oct()
object.__hex__(self)	hex()

همچنین Comparison Operators

متدها	اپراتور
object.__lt__(self, other)	<
object.__le__(self, other)	<=
object.__eq__(self, other)	==
object.__ne__(self, other)	!=
object.__ge__(self, other)	>=
object.__gt__(self, other)	>



فصل 3

مدیریت استثناءها و خطاها

به طور کلی مدیریت خطا و استثناء به معنی انجام کار مناسب در هنگام رخ دادن وضعیتی غیرعادی در نرم افزار است. حالات زیادی برای اتفاق افتادن این موارد وجود دارد مثلاً تقسیم بر صفر، باز کردن یا مسیردهی به فایلی که وجود ندارد، افتادن در حلقه بینهایت و... در اینجاست که برنامه نویس باید شرایطی را محیا نماید تا در صورت بروز چنین مشکلات غیرقابل پیش بینی نرم افزار به طور کامل از سرویس دهی بازنماند.

مجموعه این رفتارها که نرم افزار را در مقابل رخدادهای غیرقابل پیش بینی محافظت می کند را مدیریت خطا یا استثناء می گویند. کنترل مناسب خطاها توسط برنامه نویس از عدم سرویس دهی نرم افزار که موجب نارضایتی کاربر می شود جلوگیری می نماید.

در حالت کلی ما با سه نوع خطا در برنامه مواجه خواهیم گشت:

- 1- خطاهاي نحوی (syntax error)
- 2- خطاهاي زمان اجرا (run time error)
- 3- خطاهاي معنایی (semantic error)

خطاهای نحوی (syntax error)

خطاهایی که با رعایت نکردن قواعد زبان برنامه نویسی توسط کامپایلر تشخیص داده می شوند و اجرای برنامه را مختل می کنند. اگر چنین خطایی در برنامه شما رخ دهد برنامه قادر به اجرا نخواهد بود و کامپایلر پایتون با نمایش خطایی سعی می نماید تا محل بروز خطا را به برنامه نویس نشان دهد تا نسبت به رفع خطا اقدام نماید.

خطای زمان اجرا (run time error)

خطاهای زمان اجرا خطاهایی هستند که در هنگام کامپایل تشخیص داده نمی شوند و رخ دادن آنها به هنگام اجرای برنامه یا در شرایطی خاص اتفاق می افتد و باعث از کار افتادن نرم افزار می گردند. این خطاها چون در زمان اجرای برنامه و در شرایط خاص اتفاق می افتد استثنا نامیده می شوند. پیدا و رفع کردن چنین خطاهایی از خطاهای نحوی سخت تر بوده و نیازمند پرسه سنگین و زمان بر خطایی یا آزمایش است.

خطای معنایی (semantic error)

وقتی برنامه کامپایل و اجرا شود اما به دلیل خطا در منطق برنامه باعث عملکرد نادرست برنامه می شود که به آن خطای معنایی می گویند این گونه خطاها یا در اثر نادرست بودن الگوریتم برنامه یا نادرست بودن نحوه کد نویسی



اشتباه رخ می‌دهد. کشف کردن و رفع این گونه خطاهای بسیار دشوار است زیرا ممکن است برنامه در حالتی خاص جواب درست بدهد و در حالتی دیگر جواب نادرست برگرداند.

مدیریت خطاهای

در پایتون این امکان وجود دارد تا برنامه‌نویس برای قسمتی از برنامه استثنایها را کنترل نماید. برای این کار برنامه‌نویس می‌توان از عبارت try استفاده کرد این عبارت به صورت زیر مورداستفاده قرار می‌گیرد

- ابتدا تکه کد موردنظر را بین کلمات کلیدی try و except قرار می‌دهیم بدنه try به عنوان گرداننده خطای می‌شود به این معنی که در صورت بروز خطای این قسمت اجرا خواهد شد ساختار این دستور به شکل زیر است.

try:

کد موردنظر شما

except keyword:

کد جهت بروز خطای

در اینجا نوع استثنایی است که امکان دارد اتفاق بیفتد به این معنی که اگر این خطای اتفاق افتاد این تکه کد را اجرا کن. اگر این قسمت خالی باشد به این معناست که هر خطایی از هر نوعی که باشد توسط این مکانیسم بررسی می‌شود که به آن مکانیسم عمومی نیز می‌گویند. دو کلمه کلیدی else و finally در این ساختار وجود دارد که استفاده از آن‌ها برای برنامه‌نویس اختیاری است. بعد از بدنه اصلی مدیریت خطای می‌آید و اگر خطایی در بدنه برنامه اتفاق نیافتد اجرا می‌شود در غیر این صورت اجرا خواهد شد.

: بعد از بدنه اصلی مدیریت خطای می‌آید و در هر صورت بدنه آن اجرا خواهد شد.

ساختار مدیریت خطای در پایتون به صورت زیر است:

- اگر هیچ استثنایی رخ ندهد بخش except نادیده گرفته می‌شود و از آن عبور می‌کند.
- اگر در بدنه try استثنایی رخ دهد ادامه بدنه در نظر گرفته نمی‌شود و از همان قسمت به سراغ بدنه except می‌رود، اگر نوع خطای رخداده با keyword همخوانی داشته باشد وارد بدنه except می‌شود و سپس ادامه برنامه را اجرا می‌کند.
- اگر چندین مکانیسم مدیریت خطای در کد وجود داشته باشد پایتون اولین مکانیسمی را که بیشترین مطابقت را با نوع خطای رخداده داشته باشد انتخاب می‌نماید پس توجه داشته باشید که از لحاظ منطقی مکانیسم عمومی باید آخرین مکانیسم مدیریت خطای شما باشد.
- در غیر این صورت اجرای برنامه متوقف می‌شود و یک خطای برای کاربر نمایش داده خواهد شد.



در حالت کلی پیامها دارای دو بخش می‌باشند نوع خطا و توضیحی در مورد خطا که این دو با علامت (:) از هم جدا می‌شوند همچنین کامپایلر پایتون محل وقوع خطا را نیز برای برنامه‌نویس در متغیر Traceback مشخص می‌نماید به مثال‌های زیر توجه نمایید:

```
>>> 10*(1/0)
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    10*(1/0)
ZeroDivisionError: integer division or modulo by zero

>>> 4+ spam*3
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    4+ spam*3
NameError: name 'spam' is not defined

>>> '2' +2
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    '2' +2
TypeError: cannot concatenate 'str' and 'int' objects
```

همان‌طور که مشاهده نمودید مثال‌هایی را مطرح نمودیم که ایجاد خطا کردند حال می‌خواهیم با استفاده از مدیریت خطای پایتون آن‌ها را مدیریت نماییم به مثال زیر توجه نمایید:

```
try:
    1*(1/0)
except ZeroDivisionError:
    print "somthing wrong happend!!"
except:
    print "unhandel exception"

>>>
somthing wrong happend!!
```

در مثال قبل دو نوع مکانیسم برای مدیریت خطای گرفته شده ولی چون نوع خطای رخداده بدنی این خطای اجرا می‌شود و دیگر بدنی خطای دیگر بررسی نمی‌شود به مثال بعد توجه نمایید:



```

try:
    '2' + 2
    print "after error"
except ZeroDivisionError:
    print "somthing wrong happend!!"
except:
    print "unhandel exception"
print "after handeling"

>>>
unhandel exception
after handeling

```

همان‌طور که در مثال قبل مشاهده می‌نمایید دستوری که بعد از خطا وجود دارد اجرانشده است اما ادامه برنامه که بعد از محدوده مدیریت خطا وجود دارد اجرانشده است. حال می‌خواهیم همین ساختار را با کلمه کلیدی else امتحان کنیم

```

try:
    '2' + 2
    print "after error"
except ZeroDivisionError:
    print "somthing wrong happend!!"
except:
    print "unhandel exception"
else:
    print "Else Running!!!"
finally:
    print "Finally running!!!"
print "Continue Code"

>>>
unhandel exception
Finally running!!!
Continue Code

```

حال همان مثال قبل را این بار بدون خطا اجرا می‌کنیم به تکه کد زیر توجه نمایید:



```

try:
    print "Code"
except ZeroDivisionError:
    print "somthing wrong happend!!!"
except:
    print "unhandel exception"
else:
    print "Else Running!!!"
finally:
    print "Finally running!!!"
print "Continue Code"

>>>
Code
Else Running!!!
Finally running!!!
Continue Code

```

همان طور که در مثال‌ها مشاهده نمودید قسمت `else` تنها زمانی اجرا می‌شود که هیچ خطایی رخ ندهد ولی قسمت `finally` در هر صورت اجرا خواهد شد.

همچنین در پایتون این امکان وجود دارد که مدیریت خطاها به صورت تودرتو انجام پذیرد به مثال زیر توجه نمایید:

```

import sys
try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except IOError as (errno, strerror):
    print "I/O error({0}): {1}".format(errno, strerror)
    try:
        print 10/0
    except ZeroDivisionError:
        print "ZeroDivisionError happend"
    finally:
        print "try..try"

except ValueError:
    print "Could not convert data to an integer."
except:
    print "Unexpected error:", sys.exc_info()[0]

>>>
I/O error(2): No such file or directory
ZeroDivisionError happend
try..try..try

```



عبارت `raise` در پایتون این امکان را به برنامهنویس می‌دهد تا برنامه را مجبور به اجرا کردن یک استثنای مشخص نماید. متغیر `raise` در ورودی دریافت می‌کند یکی نوع استثنای و دیگری اطلاعاتی در مورد خطای اتفاق افتاده است در حقیقت برنامهنویس می‌تواند یک نوع جدید از استثنای را به صورت شخصی ایجاد نماید به مثال زیر توجه نمایید:

```
try:
    raise NameError('HiThere')
except NameError:
    print "A NameError Happend!"

>>>
A NameError Happend!
```

همچنین به کار بردن عبارت `raise` به تنها ی باعث نمایش خطای رخداده می‌شود به مثال زیر توجه فرمایید:

```
try:
    raise NameError('HiThere')
except NameError:
    print "A NameError Happend!"
    raise

>>>
A NameError Happend!

Traceback (most recent call last):
  File "C:\Users\arCco\Desktop\test1.py", line 2, in <module>
    raise NameError('HiThere')
NameError: HiThere
```

همچنین این امکان در زبان پایتون وجود دارد تا یک کلاس را به صورت یک استثنای `raise` نماید به مثال زیر توجه نمایید:



```
class ShortInputException(Exception):
    def __init__(self, length, atleast):
        Exception.__init__(self)
        self.length = length
        self.atleast = atleast
    try:
        s = raw_input('Enter something --> ')
        if len(s) < 3:
            raise ShortInputException(len(s), 3)
    except EOFError:
        print '\nWhy did you do an EOF on me?'
    except ShortInputException, x:
        print 'ShortInputException: The input was of length%d was expecting at
least%d' % (x.length, x.atleast)
    else:
        print 'No exception was raised.'

>>>
Enter something --> 12
ShortInputException: The input was of length 2 was expecting at least 3
```



فصل 4

کار با پایگاه داده²³

یکی از قابلیت‌هایی که هر زبان برنامه‌نویسی باید داشته باشد، دسترسی به پایگاه‌های داده است. پایتون هم از این قاعده مستثنی نیست. پایتون قابلیت ایجاد ارتباط با تقریباً تمامی پایگاه‌های داده‌ای که امروزه استفاده می‌شود، را دارد. در این فصل قرار است که بررسی توانایی پایتون برای ارتباط با دو نوع پایگاه داده مختلف بپردازیم. توجه کنید که پیش‌فرض این فصل این است که شما با مفاهیم پایگاه داده و زبان SQL آشنایی دارید و قصد ما در این فصل آموزش این مباحث نیست.

تعريف

پایگاه داده، دادگان، بانک داده، بانک اطلاعات، به مجموعه‌ای از داده‌های مرتبط، ساختارمند یا سازمان‌یافته گفته می‌شود که دسترسی به این اطلاعات معمولاً از طریق سیستم مدیریت پایگاه داده یا (DBMS) صورت می‌گیرد. سیستم مدیریت پایگاه داده متشکل از مجموعه‌ی یکپارچه از نرم‌افزارهای رایانه‌ای است که به کاربران اجازه می‌دهد تا با یک یا چند پایگاه داده ارتباط یافته و به اطلاعات موجود در پایگاه داده دسترسی یابند؛ هرچند که این دسترسی کامل بوده و یا در صورت وجود محدودیت به بخشی از اطلاعات دسترسی پیدا کنند. DBMS عملکردهای مختلفی را برای ورود، ذخیره‌سازی و بازیابی مقادیر زیادی از اطلاعات فراهم و راههای متنوعی برای مدیریت چگونگی سازمان‌یابی اطلاعات ارائه می‌کند. از آنجاکه بین پایگاه داده و سیستم مدیریت پایگاه داده قربت بسیاری وجود دارد بعضی اوقات اصطلاح پایگاه داده برای اشاره به هر دو بکار می‌رود.

موجودیت دی.بی.ام.اس. عملکردهای مختلفی را برای مدیریت یک پایگاه داده در اختیار قرار می‌دهد؛ و داده‌های موجود در پایگاه که می‌توانند طبقه‌بندی شوند به چهار دسته اصلی عملیاتی دسته‌بندی می‌شوند:

- تعریف داده‌ها: ایجاد، اصلاح و حذف تعاریفی که سازمان یک داده با آن تعریف می‌شود.

- به روزرسانی: درج، اصلاح و حذف داده‌های واقعی.

- بازیابی: ارائه اطلاعات در یک قالب به‌طوری که به‌طور مستقیم قابل استفاده و یا قابل پردازش برای برنامه‌های کاربردی دیگر باشد. داده‌های بازیابی شده ممکن است در یک قالب که اساساً همانند آن چیزی که در پایگاه داده ذخیره شده است ساخته شود و یا در قالب جدیدی ناشی از تغییر و ترکیب اطلاعات موجود در پایگاه به وجود آید.



- مدیریت: ثبت‌نام و نظارت بر کاربران، اجرای امنیت داده‌ها، نظارت بر عملکرد، حفظ تمامیت داده‌ها، خرید و فروش با کنترل هم‌زمانی، و دوره نقاوت بعد اطلاعات است که توسط برخی از حوادث مانند خطای غیرمنتظره سیستم به وجود آید.

Mysql

مای‌اس‌کیوال یک سامانه مدیریت پایگاه داده‌ها متن‌باز و یک پایگاه داده است، که توسط شرکت اوراکل توسعه، توزیع، و پشتیبانی می‌شود.

سرور مای‌اس‌کیوال به چندین کاربر اجازه استفاده هم‌زمان از داده‌ها را می‌دهد. مای‌اس‌کیوال از مزیت‌های زیر بهره‌مند است:

- مقیاس‌پذیری و قابلیت انعطاف
- عملکرد بالا
- در دسترس بودن بالا
- پشتیبانی از تراکنش‌ها
- محافظت از داده
- آسان بودن مدیریت
- آزاد بودن برنامه
- پشتیبانی شبانه‌روزی

MySQL یک شاخه (Fork) از MariaDB

MariaDB یک شاخه توسعه‌یافته در انجمن‌های برنامه‌نویسی از سیستم مدیریت پایگاه داده رابطه‌ای MySQL است که قصد دارد تحت GNU GPL آزاد بماند.

به دلیل نگرانی در مورد خریداری MySQL توسط اوراکل اصلی MySQL ایجاد شد. MariaDB در نظر دارد سازگاری بالایی با MySQL داشته باشد و از قابلیت جایگزینی MySQL برخوردار باشد و این مهم از طریق مطابقت دقیق با API‌ها و دستورات MySQL به دست‌آمده است. برخی اختلافات و ناسازگاری‌های مستند بین نسخه‌های MySQL و MariaDB وجود دارد، به عنوان مثال برخی از ابزارهای تعامل با MySQL مانند MySQL Workbench کاملاً با MariaDB سازگار نیستند.

پایتون و Mysql

از آنجایی که MySQL مشهورترین پایگاه داده در وب است، ابتدا کار با این پایگاه داده را بررسی می‌کنیم. همچنین برای آنکه بتوانید از MySQL استفاده کنید باید از قبل آن را در سیستم خود نصب کرده باشید. همچنین پایتون به صورت پیش‌فرض توانایی اتصال به MySQL را ندارد بلکه باید بسته ارتباط خاص خودش را نیز نصب کنید. بسته‌های متعددی برای پایتون (برای اتصال به MySQL) نوشته شده‌اند که ما از MySQL



استفاده خواهیم کرد. برای نصب MySQL Connector نیز از PIP استفاده می‌کنیم. نصب Connector از طریق PIP با کد زیر انجام می‌شود:

```
pip install mysql-connector
```

بعد از نصب موفقیت‌آمیز شما می‌توانید با فراخوانی این بسته از آن در برنامه خود استفاده نمایید.

اتصال به پایگاه داده

Connect

بعد از فراخوانی بسته MySQL Connector شما برای اتصال به پایگاه داده خود از متده استفاده می‌کنید، البته توجه داشته باشید که اطلاعات اتصال را باید از طریق ورودی برای این متده ارسال نمایید. به مثال زیر توجه نمایید:

```
import mysql.connector

mydb = mysql.connector.connect(
    host="localhost",
    user="yourusername",
    password="yourpassword"
)
print(mydb)
>>>
<mysql.connector.connection.MySQLConnection object at 0x016645F0>
```

در صورتی که خروجی شما با ما یکسان باشد به این معناست که بقدرتی توانسته‌اید به پایگاه داده خود متصل شوید، همچنین ورودی‌های این متده عبارت‌اند از :

- : شامل آدرس سروری است که پایگاه داده شما در آن در حال اجراست.
- : نام کاربری که در پایگاه داده شما برای ارتباط در نظر گرفته شده است.
- : کلمه عبوری که برای آن نام کاربری در نظر گرفته شده است.
- : نام پایگاه داده‌ای که می‌خواهید به آن متصل شوید.

connected

اگر نمونه ایجادشده متصل باشد مقدار true و در غیر این صورت مقدار false را برمی‌گرداند.

close

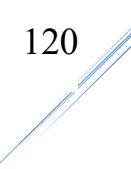
این متده به یک اتصال به پایگاه داده mysql خاتمه می‌دهد.

اجرای دستورات

cursor

های ما به اجازه می‌دهند که با استفاده از یک اتصال پایگاه داده، چندین محیط کاری داشته باشیم.





execute

با استفاده از این متدهای توانیم کد SQL را در محیط کاری یک cursor اجرا نماییم.
به مثال زیر توجه نمایید:

```
import mysql.connector

mydb = mysql.connector.connect(
    host="localhost",
    user="yourusername",
    password="yourpassword"
)

mycursor = mydb.cursor()

mycursor.execute("CREATE DATABASE mydatabase")
```

در کد بالا ما با استفاده از cursor و متدهای execute دستور مربوط به ساختن یک پایگاه داده جدید بنام mydatabase را اجرا نموده‌ایم. حال می‌خواهیم فهرستی از پایگاه داده‌های موجود را نمایش دهیم که ما به صورت زیر خواهد بود:

```
import mysql.connector

mydb = mysql.connector.connect(
    host="localhost",
    user="myusername",
    password="mypassword"
)

mycursor = mydb.cursor()
mycursor.execute("SHOW DATABASES")
for x in mycursor:
    print(x)
>>>
('information_schema',)
('mydatabase',)
('performance_schema',)
('sys',)
```

همان‌طور که مشاهده می‌کنید با استفاده از cursor توانستیم به مقادیر داخل دستور اجرایی توسط متدهای execute دسترسی داشته باشیم. حال می‌خواهیم با استفاده از همین دستورات یک جدول در پایگاه داده‌ای که ساخته‌ایم ایجاد کنیم، به مثال زیر توجه نمایید:



```

import mysql.connector
mydb = mysql.connector.connect(
    host="localhost",
    user="yourusername",
    password="yourpassword",
    database="mydatabase"
)
mycursor = mydb.cursor()
mycursor.execute("CREATE TABLE customers (name VARCHAR(255), address
VARCHAR(255))")

```

همان‌طور که در کد بالا مشاهده می‌کنید این بار این دستور را برای پایگاه داده mydatabase اجرا نمودیم، این اولین بار است که در ما کد sql خود را برای یک پایگاه داده خاص اجرا می‌نماییم.
ما همچنین می‌توانیم تابع execute را با دو مقدار ورودی اجرا کنیم به این صورت که مقادیری را به کد sql خود ارسال کنیم. به مثال زیر توجه نمایید:

```

import mysql.connector

mydb = mysql.connector.connect(
    host="localhost",
    user="yourusername",
    password="yourpassword",
    database="mydatabase"
)

mycursor = mydb.cursor()
sql = "INSERT INTO customers (name, address) VALUES (%s, %s)"
val = ("John", "Highway 21")
mycursor.execute(sql, val)
mydb.commit()
print(mycursor.rowcount, "record inserted.")
>>>
1 record inserted.

```

همان‌طور که مشاهده می‌کنید یک سطر جدید به جدول customers اضافه کردیم. متدهای commit و executemany ذخیره اعمال تغییرات در جداول یک پایگاه داده مورد استفاده قرار می‌گیرد.
از این متدهای اجرای چندین دستور پشت سرهم استفاده می‌کنیم به مثال زیر توجه نمایید:



```

import mysql.connector

mydb = mysql.connector.connect(
    host="localhost",
    user="yourusername",
    password="yourpassword",
    database="mydatabase"
)
mycursor = mydb.cursor()
sql = "INSERT INTO customers (name, address) VALUES (%s, %s)"
val = [
    ('Peter', 'Lowstreet 4'),
    ('Amy', 'Apple st 652'),
    ('Hannah', 'Mountain 21'),
    ('Michael', 'Valley 345'),
    ('Sandy', 'Ocean blvd 2')
]
mycursor.executemany(sql, val)
mydb.commit()
print(mycursor.rowcount, "was inserted.")
>>>
5 record was inserted.

```

در مثال در متدهای executemany پارامتر اول آن همان دستور SQL ما و پارامتر دومش نیز فهرستی از tuple هاست که همان دادهایی هستند که میخواهیم در جدول وارد کنیم.

نکته دیگر در مثال فوق خاصیت rowcount است که در این مثال تعداد سطرهایی که از کد sql ما تأثیر گرفته-اند را نشان میدهد.

بازیابی نتایج fetchall

کار این متدها برگرداندن تمامی خروجی یک دستور sql است. به مثال زیر توجه نمایید:



```

import mysql.connector

mydb = mysql.connector.connect(
    host="localhost",
    user="yourusername",
    password="yourpassword",
    database="mydatabase"
)
mycursor = mydb.cursor()
mycursor.execute("SELECT * FROM customers")
myresult = mycursor.fetchall()
for x in myresult:
    print(x)
>>>
(1, 'John', 'Highway 21')
(2, 'Peter', 'Lowstreet 27')
(3, 'Amy', 'Apple st 652')
(4, 'Hannah', 'Mountain 21')
(5, 'Michael', 'Valley 345')

```

fetchone

این متده بدون توجه به تعداد سطرهای برگردانده شده تنها سطر اول را به عنوان خروجی برمی‌گرداند، به مثال زیر توجه نمایید:

```

import mysql.connector

mydb = mysql.connector.connect(
    host="localhost",
    user="yourusername",
    password="yourpassword",
    database="mydatabase"
)

mycursor = mydb.cursor()
mycursor.execute("SELECT * FROM customers")
myresult = mycursor.fetchone()
print(myresult)
>>>
(1, 'John', 'Highway 21')

```

MongoDb

مانگودی‌بی یک پایگاه داده‌های سند-گرای متن‌باز، کلارا، مقیاس‌پذیر، بدون نیاز به طرح‌بندی اولیه نوشته شده در زبان برنامه‌نویسی سی⁺⁺ است.

هدف مانگودی‌بی پر کردن فاصله ذخیره بندهای کلید/مقداری که سریع و مقیاس‌پذیر هستند و سامانه‌های سنتی مدیریت پایگاه داده رابطه‌ای که در خواسته‌های غنی و عملکرد عمیقی دارند— بوده است. مانگودی‌بی



برای رفع مشکلاتی طراحی شده که با پایگاه داده‌های رابطه‌ای بهسادگی رفع نمی‌شوند؛ برای مثال اگر پایگاه داده کارسازهای زیادی را در برگیرد.

مانگودی‌بی به جای اینک همانند پایگاههای داده‌های رابطه‌ای کلاسیک داده‌ها را در جداول ذخیره کند، داده‌های ساختاریافته را در اسنادی با قالبی شبیه به جی‌سون (مانگودی‌بی این قالب را بی‌سون(BSON) می‌نامد) ذخیره‌سازی می‌کند، و بدین ترتیب یکپارچه‌سازی داده‌ها را در برخی اقسام برنامه‌های کاربردی آسان‌تر و سریع‌تر می‌کند.

پایتون و MongoDB

همان‌طور که گفتیم پایتون می‌تواند با برنامه‌های پایگاه داده مختلفی کار کند اما ممکن است نخواهد از برنامه‌هایی مانند MySQL استفاده کنید، بلکه برنامه‌هایی مانند MongoDB مدنظر شما باشند. MongoDB یکی از مشهورترین برنامه‌های پایگاه داده‌ی NoSQL است. درواقع پایگاههای داده‌ی SQL همگی دارای جدول‌هایی هستند و این جدول‌ها با یکدیگر روابط مشخص دارند اما پایگاههای داده‌ی NoSQL مدل‌های دیگری به‌غیراز مدل جدولی SQL برای دریافت و ذخیره‌ی داده‌ها دارند. می‌توان از مشهورترین پایگاههای داده‌ی NoSQL به موارد زیر اشاره کرد:

- MongoDB
- Redis
- DynamoDB
- Azure Cosmos DB
- Couchbase
- HBase

پایگاه داده‌ی MongoDB داده‌ها را در قالب‌هایی شبیه قالب JSON ذخیره می‌کند، به همین دلیل بسیار منعطف و مقیاس‌پذیر است. البته توجه داشته باشید که برای اجرای کدهای خود باید از قبل این پایگاه داده را در کامپیوتر خود نصب داشته باشید.

همچنین پایتون به صورت پیش‌فرض توانایی اتصال به MongoDB را ندارد بلکه باید بسته ارتباط خاص خودش را نیز نصب کنید. بسته‌های متعددی برای پایتون (برای اتصال به MongoDB) نوشته شده‌اند که ما از PyMongo استفاده خواهیم کرد. برای نصب PyMongo نیز از PIP استفاده می‌کنیم. نصب PyMongo از طریق PIP با کد زیر انجام می‌شود:

```
pip install pymongo
```

بعد از نصب موفقیت‌آمیز شما می‌توانید با فراخوانی این بسته از آن در برنامه خود استفاده نمایید.



اتصال به پایگاه داده

برای اتصال به یک پایگاه داده در MongoDB باید ابتدا شیء MongoClient را ایجاد کنید، سپس URL اتصال را به همراه ip و نام پایگاه داده به آن ارسال کنید. اگر چنین پایگاه داده‌ای از قبل وجود نداشته باشد آن را ساخته و یک اتصال با آن برقرار می‌کند.

```
import pymongo
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
```

البته شما همچنین می‌توانید با ساختار دیگری نیز آدرس، پورت و نام پایگاه داده خود را برای MongoClient ارسال کنید به مثال زیر توجه نمایید:

```
import pymongo
myclient = pymongo.MongoClient("localhost", 27017)
mydb = myclient.mydatabase
```

list_database_names

برای مشاهده لیست پایگاه‌های داده‌ی موجود در سیستم می‌توان از این متده استفاده کرد به مثال زیر توجه نمایید:

```
import pymongo

myclient = pymongo.MongoClient('mongodb://localhost:27017/')
print(myclient.list_database_names())
>>>
['admin', 'local', 'mydatabase']
```

طفاً توجه داشته باشید که MongoDB صبر می‌کند تا document در collection شما دریافت شود و سپس آن پایگاه داده را می‌سازد. به عبارت دیگر در MongoDB تا زمانی که محتوایی دریافت نشود، هیچ collection یا پایگاه داده‌ای ساخته نخواهد شد و collection خالی هیچ معنایی ندارد. در MongoDB مفهوم معادل جدول و document معادل ردیف (record) است.



اجرای دستورات

Collection

کالکشن^{۲۴} معادل جدول در پایگاه‌های داده‌ی SQL است. برای ساخت collection ها باید از شیء پایگاه داده استفاده کرده و نام کالکشن را انتخاب کنید. در صورتی که collection موردنظر شما از قبل وجود نداشته باشد توسط MongoDB ساخته خواهد شد. به مثال زیر توجه نمایید:

```
import pymongo

myclient = pymongo.MongoClient('mongodb://localhost:27017/')
mydb = myclient['mydatabase']
mycol = mydb["customers"]
```

list_collection_names

این متده فهرستی از collection های موجود در پایگاه داده را در خروجی برای ما برمی‌گرداند، به مثال زیر توجه کنید:

```
import pymongo

myclient = pymongo.MongoClient('mongodb://localhost:27017/')
mydb = myclient['mydatabase']
mycol = mydb["customers"]
print(mydb.list_collection_names())
>>>
['customers']
```

توجه داشته باشید؛ تا زمانی که داده‌ای (document) در این collection ها ثبت نشود، collection موردنظر ایجاد نخواهد شد. بنابراین در لیست collection ها (مثال‌های بالا) نیز وجود نخواهد داشت.

drop

این متده برای حذف یک collection مورداستفاده قرار می‌گیرد، به مثال زیر توجه نمایید:

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]
mycol.drop()
```

همان‌طور که مشاهده می‌کنید در کد بالا اقدام به حذف یک collection با نام customers کردۀ‌ایم. (البته



توجه داشته باشید که چون ما به این collection در ادامه نیاز داریم برای انجام تمرینات بعدی دوباره آن را ایجاد نمودیم.)

insert_one

در مثال‌های قبل دیدیم که چگونه یک collection به نام customers ساخته شد حال می‌خواهیم مقادیری را در آن ذخیره کنیم، برای این کار از می‌توانیم از متده استفاده کنیم. به مثال زیر توجه نمایید:

```
import pymongo
```

```
myclient = pymongo.MongoClient('mongodb://localhost:27017/')
mydb = myclient['mydatabase']
mycol = mydb["customers"]
mydict = { "name": "John", "address": "Highway 37" }
x = mycol.insert_one(mydict)
print(x)
>>>
<pymongo.results.InsertOneResult object at 0x03D62918>
```

insert_many

اگر بخواهیم چندین داده (document) را به صورت همزمان وارد collection خود کنیم، استفاده از متده insert_one() کار سختی خواهد بود. روش بهتر استفاده از متده insert_many() است. آرگومان اول این متده یک list است که شامل dictionary‌هایی است که حاوی تمام موارد موردنظر برای ثبت در collection است. به

مثال زیر توجه نمایید:

```
import pymongo
```

```
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

mylist = [
    { "name": "Amy", "address": "Apple st 652" },
    { "name": "Hannah", "address": "Mountain 21" },
    { "name": "Michael", "address": "Valley 345" },
    { "name": "Sandy", "address": "Ocean blvd 2" }
]

x = mycol.insert_many(mylist)
# print list of the _id values of the inserted documents:
print(x.inserted_ids)

>>>
[ObjectId('5b19112f2ddb101964065487'), ObjectId('5b19112f2ddb101964065488'),
 ObjectId('5b19112f2ddb101964065489'), ObjectId('5b19112f2ddb10196406548a')]
```



همان‌طور که مشاهده می‌کنید در خط آخر این کد با استفاده از `inserted_ids` توانستیم فهرستی از `objectId` هایی که به ازای هر `insert` اضافه شده بود را نمایش دهیم.

۲۵ شیء کوئری

قبل از ورود به بخش‌های `update_many`, `delete_many` و `update_one` لازم است تا در مورد مفهوم شیء کوئری توضیح کوتاهی ارائه دهیم، درواقع شیء کوئری یک شیء ساده است (علامت‌های `{}`) که درون خود کوئری خاصی دارد. منظور ما از کوئری استفاده از زبان برقراری ارتباط با پایگاه داده‌ی MongoDB است. بعداً در بخش مربوط به آن توضیحات بیشتری در این مورد خواهیم داد.

`update_one`

این متدهایی برای بهروزرسانی یک `document` مورداستفاده قرار می‌گیرد به این صورت که در پارامتر اول یک شیء کوئری و در پارامتر دوم نیز یک شیء که مقدار جدید ردیف را در خود دارد دریافت می‌کند و بر اساس آن کوئری، یک `document` را بروز رسانی خواهد کرد. به مثال زیر توجه نمایید:

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

myquery = { "address": "Valley 345" }
newvalues = { "$set": { "address": "Canyon 123" } }

mycol.update_one(myquery, newvalues)

#print "customers" after the update:
for x in mycol.find():
    print(x)
>>>
{'_id': 1, 'name': 'John', 'address': 'Highway37'}
{'_id': 2, 'name': 'Peter', 'address': 'Lowstreet 27'}
{'_id': 3, 'name': 'Amy', 'address': 'Apple st 652'}
{'_id': 4, 'name': 'Hannah', 'address': 'Mountain 21'}
{'_id': 5, 'name': 'Michael', 'address': 'Canyon 123'}
```

اگر از در مثال‌ها از متدهایی استفاده می‌شود که درباره آن‌ها توضیحی داده نشده (برای مثال متدهای `find`) نگران نباشید در بخش‌های بعد در مورد تمام آن‌ها صحبت خواهیم کرد.



update_many

اگر بخواهیم بیشتر از یک document را ویرایش کنیم باید از این تابع استفاده نماییم. این تابع برخلاف تابع update_one() که تنها نتیجه‌ی اول را ویرایش می‌کرد، هر نتیجه‌ای که توسط شیء کوئری برگردانده شود را ویرایش خواهد کرد. به مثال زیر توجه نمایید:

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

myquery = { "address": { "$regex": "^S" } }
newvalues = { "$set": { "name": "Minnie" } }

x = mycol.update_many(myquery, newvalues)
print(x.modified_count, "documents updated.")
>>>
2 documents updated.
```

همان‌طور که مشاهده می‌کنید در خط آخر این کد با استفاده از modified_count تعداد ردیف‌های بهروزرسانی شده را نمایش دادیم.

delete_one

این متده برای حذف یک document مورداستفاده قرار می‌گیرد. به مثال زیر توجه نمایید:

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

myquery = { "address": "Mountain 21" }
mycol.delete_one(myquery)
for x in mycol.find():
    print(x)
>>>
{'_id': 1, 'name': 'John', 'address': 'Highway37'}
{'_id': 2, 'name': 'Peter', 'address': 'Lowstreet 27'}
{'_id': 3, 'name': 'Amy', 'address': 'Apple st 652'}
{'_id': 5, 'name': 'Michael', 'address': 'Valley 345'}
```

همان‌طور که مشاهده می‌کنید در مثال بالا یکی از document ها را حذف کردیم.

delete_many

این متده برای حذف چندین document مورداستفاده قرار می‌گیرد. اولین ورودی این متده یک شیء کوئری است که مشخص می‌کند کدام document ها را باید پاک کنیم. به مثال زیر توجه نمایید:



```

import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

myquery = { "address": { "$regex": "^S" } }
x = mycol.delete_many(myquery)
print(x.deleted_count, "documents deleted")
>>>
2 documents deleted.

```

همان‌طور که مشاهده می‌کنید در مثال قبل اقدام به حذف تمامی document هایی که نام آن‌ها با S شروع می‌شود کردیم، برای این کار از عبارات باقاعدۀ استفاده کردیم. (اگر با عبارات باقاعدۀ آشنایی ندارید نگران نباشد در فصل ۵ به تفصیل در مورد آن صحبت خواهیم کرد).

همان‌طور که مشاهده می‌کنید در خط آخر این کد با استفاده از deleted_count تعداد ردیف‌های حذف شده را نمایش دادیم.

همچنین اگر بخواهیم تمام document ها را در یک collection حذف نماییم، می‌توانیم یک شیء کوئری خالی را به عنوان پارامتر ورودی برای این متده ارسال کنیم، تا کوئری شامل تمام موارد شود. به مثال زیر توجه نمایید:

```

import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]
x = mycol.delete_many({})
print(x.deleted_count, "documents deleted")
>>>
4 documents deleted.

```

بازیابی نتایج

find_one

این متده معادل دستور SELECT است و برای پیدا کردن داده‌ها درون collection مورده استفاده قرار می‌گیرد. تنها مورد آن این است که تنها یک document را در خروجی برمی‌گرداند. به مثال زیر توجه نمایید:



```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]
x = mycol.find_one()
print(x)
>>>
{'_id': 1, 'name': 'John', 'address': 'Highway37'}
```

find

این متدهایی برای واکشی اطلاعات از پایگاه داده است اما تفاوت آن با متدهای قبلی این است که میتواند هر تعداد document که به عنوان خروجی برگردانده شود را برگرداند. اولین پارامتر ورودی این متدهای کوئری است. اگر ما یک شیء کوئری خالی را به عنوان ورودی به ارسال نماییم تمام documents را درون collection میگرداند. برگردانده خواهد شد. به مثال زیر توجه نمایید:

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

for x in mycol.find():
    print(x)
>>>
{'_id': 1, 'name': 'John', 'address': 'Highway37'}
{'_id': 2, 'name': 'Peter', 'address': 'Lowstreet 27'}
{'_id': 3, 'name': 'Amy', 'address': 'Apple st 652'}
{'_id': 4, 'name': 'Hannah', 'address': 'Mountain 21'}
{'_id': 5, 'name': 'Michael', 'address': 'Valley 345'}
```

پارامتر دوم که یک پارامتر اختیاری است، تعیین میکند چه فیلد هایی در نتایج برگشتی قرار بگیرند. این پارامتر اختیاری است و اگر چیزی به آن نداده باشد تمام فیلد ها برگردانده خواهند شد. به مثال زیر توجه نمایید:



```

import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

for x in mycol.find({}, { "_id": 0, "name": 1, "address": 1 }):
    print(x)
>>>
{'name': 'John', 'address': 'Highway37'}
{'name': 'Peter', 'address': 'Lowstreet 27'}
{'name': 'Amy', 'address': 'Apple st 652'}
{'name': 'Hannah', 'address': 'Mountain 21'}
{'name': 'Michael', 'address': 'Valley 345'}

```

همان‌طور که مشاهده می‌کنید در مثال قبل در ورودی تعیین کردیم که دو فیلد نام و آدرس نمایش داده شوند و فیلد `_id` برگردانده نشود.

توجه داشته باشید که شما اجازه ندارید که در یک شیء مقادیر 0 و 1 را مشخص کنید؛ درواقع اگر به یک فیلد مقدار 0 بدهید، بقیه‌ی فیلدها مقدار 1 می‌گیرند. در این مورد تنها یک استثنا وجود دارد و آن این است که یکی از فیلدها `_id` باشد در این صورت می‌توانیم هر دو مقدار 0 و 1 را مشخص کنیم. به مثال زیر توجه نمایید:

```

import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

for x in mycol.find({}, { "address": 0 }):
    print(x)
>>>
{'_id': 1, 'name': 'John'}
{'_id': 2, 'name': 'Peter'}
{'_id': 3, 'name': 'Amy'}
{'_id': 4, 'name': 'Hannah'}
{'_id': 5, 'name': 'Michael'}

```

آشنایی با شیء کوئری

ما می‌توانیم با استفاده از شیء کوئری نتایج خود را فیلتر کنیم. همان‌طور که می‌دانید شیء کوئری پارامتر اول `find()` است. به طور مثال اگر بخواهیم `document` هایی را برگردانیم که آدرس آن‌ها Park Lane 38 باشد باید چیزی شبیه به `{ "address": "Park Lane 38" }` را در پارامتر ورودی ارسال کنیم همچنین ما می‌توانیم `document` با استفاده از `modifier` ها شیء کوئری خود را بسیار پیشرفته‌تر کنیم؛ به طور مثال اگر بخواهیم



هایی را برگردانیم که در آن‌ها address با حرف S یا حروف بعد از S (در الفبای انگلیسی) شروع شده باشد می‌توانیم از `{"$gt": "S"}` استفاده کنیم.

شما حتی می‌توانید از عبارات باقاعدۀ نیز برای ساخت شئ کوئری پیچیده‌تر استفاده کنید. به‌طور مثال اگر بخواهیم document هایی را برگردانیم که آن‌ها با حرف S شروع شده باشد می‌گوییم `regex": "$^S"`. البته دنیای regular expression ها دنیای بسیار بزرگی است که در فصل مربوط به خودش به صورت کامل به آن پرداخته‌ایم، اگر شما به درستی از regular expression استفاده کنید، به‌راحتی می‌توانید هر مقداری را در متدهای مربوطه جستجو کنید.

Sort

این متد برای مرتب‌سازی نتایج برگشتی از پایگاه داده استفاده می‌شود و نتایج برگشتی را به صورت صعودی یا نزولی مرتب می‌کند (حالت پیش‌فرض این مرتب‌سازی صعودی است). این متد دو پارامتر را به عنوان ورودی دریافت می‌کند که به ترتیب `fieldname` (نام فیلد) و `direction` (به معنی «جهت») – همان نزولی یا صعودی بودن که با دو عدد 1 و -1 مشخص می‌شوند) نام دارند. به مثال زیر توجه کنید:

```
import pymongo
```

```
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

mydoc = mycol.find().sort("name", -1)
for x in mydoc:
    print(x)
>>>
{'_id': 12, 'name': 'William', 'address': 'Central st 954'}
{'_id': 14, 'name': 'Viola', 'address': 'Sideway 1633'}
{'_id': 10, 'name': 'Vicky', 'address': 'Yellow Garden 2'}
{'_id': 9, 'name': 'Susan', 'address': 'One way 98'}
{'_id': 6, 'name': 'Sandy', 'address': 'Ocean blvd 2'}
{'_id': 8, 'name': 'Richard', 'address': 'Sky st 331'}
{'_id': 2, 'name': 'Peter', 'address': 'Lowstreet 27'}
{'_id': 5, 'name': 'Michael', 'address': 'Valley 345'}
{'_id': 1, 'name': 'John', 'address': 'Highway37'}
{'_id': 4, 'name': 'Hannah', 'address': 'Mountain 21'}
{'_id': 13, 'name': 'Chuck', 'address': 'Main Road 989'}
{'_id': 7, 'name': 'Betty', 'address': 'Green Grass 1'}
{'_id': 11, 'name': 'Ben', 'address': 'Park Lane 38'}
{'_id': 3, 'name': 'Amy', 'address': 'Apple st 652'}
```

همان‌طور که مشاهده می‌کنید ما خروجی این درخواست را بر اساس `name` و به صورت نزولی مرتب نمودیم.



limit

این متدها نتایجی برگشتی از یک درخواست مشخص را محدود می‌کند. این تابع یک پارامتر در ورودی دریافت می‌کند که آن پارامتر مشخص می‌کند که تعداد ردیف (document) های برگشتی چند تعداد باشد. به مثال زیر توجه نمایید:

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

myresult = mycol.find().limit(3)

#print the result:
for x in myresult:
    print(x)
>>>
{'_id': 1, 'name': 'John', 'address': 'Highway37'}
{'_id': 2, 'name': 'Peter', 'address': 'Lowstreet 27'}
{'_id': 3, 'name': 'Amy', 'address': 'Apple st 652'}
```



فصل 5

عبارت باقاعدۀ Regular Expression

در علم رایانه، عبارت باقاعدۀ regular expressions (regular expressions) که آن را تحت عنوان regexp یا regex نیز می‌برند به معنی تطبیق رشته در متن است که از قبیل نویسه‌های خاص، کلمات و الگوهایی از نویسه‌ها است. در حقیقت عبارات باقاعدۀ دارای ساختارهای مخصوصی می‌باشند که به‌وسیله آن‌ها می‌توان یک‌رشته بزرگ خطی را جستجو پیمایش نمود.

عبارات باقاعدۀ محدود و مربوط به زبان خاصی نمی‌باشند و در بسیاری از زبان‌های برنامه‌نویسی این امکان از قبل طراحی گردیده و امکانات مناسبی نیز وجود دارد.

ماژولی که در زبان پایتون برای این کار در نظر گرفته شده است ماژول re است. برنامه‌نویس می‌تواند با استفاده از یک‌زبان ساده و کوچک اقدام به تولید قواعدی نماید. این قواعد می‌تواند در مورد هر چیزی مانند آدرس پست الکترونیکی یا دستورات متنه یا هر چیز دیگری که برنامه‌نویس بخواهد باشد.

در پایتون عبارات باقاعدۀ تبدیل به کدهای بایتی می‌شوند که به‌وسیله یک موتور انطباق‌دهنده که به زبان برنامه‌نویسی C نوشته شده اجرا می‌شوند. توجه داشته باشید که برای کارهای پیچیده حتماً باید در مورد نحوه اجرای این کدها دقیق‌تر از مذکور شده باشید تا بتوانید بهترین راه را برای نوشتن عبارت خود انتخاب نمایید که هم سریع‌تر باشد و هم نتیجه دقیق‌تری داشته باشد.

زبان عبارات باقاعدۀ بسیار کوچک و محدود است بنابراین شما این توانایی را نخواهید داشت که هر کاری را به‌سادگی با این زبان انجام دهید؛ بنابراین در قسمت‌هایی که قوانین شما پیچیده می‌شوند استفاده از توابع پایتون بیشتر توصیه می‌شود.

کاراکترهای انطباقی (Matching Characters)

در عبارات باقاعدۀ بیشتر کاراکترها معادل خودشان می‌باشند برای مثال شما اگر بخواهید کلمه test را با استفاده از عبارات باقاعدۀ در یک متن جستجو نمایید دقیقاً باید خود این کلمه را جستجو نمایید. (همچنین می‌توانید تا حساسیت به بزرگی و کوچکی حروف را نیز فعال نمایید)

اما در این میان کاراکترهای ویژه‌ای نیز وجود دارند که با خودشان انطباق ندارند و یا تکرار و ترکیب آن‌ها با کاراکترهای دیگر نیز معانی ویژه‌ای را شامل می‌شود. به این کاراکترها اصطلاحاً فرا کاراکتر (MetaCharacter) گفته می‌شود در زیر فهرستی از این کاراکترها را مشاهده می‌نمایید.

. ^ \$ * +? { [] \ | ()



در ادامه به توضیح این کاراکترها خواهیم پرداخت:

 این کاراکترها برای مشخص نمودن محدوده‌ای از کاراکترها که برنامه‌نویس تمایل دارد انطباق دهد (که اصطلاحاً به آن کلاس می‌گوییم) مورد استفاده قرار می‌گیرد. کاراکترها می‌توانند به صورت انفرادی لیست شوند یا به صورت یک محدوده از کاراکترها مشخص گردند که با کاراکتر – از هم جدا می‌شوند برای مثال [a-c] که معادل عبارت [abc] است. همچنین اکثر فرا کاراکترها در داخل این کلاس فعال نمی‌باشند این بدان معناست که عبارت [abc\$] تمام کاراکترهایی که شامل کاراکترهای a,b,c,\$ می‌باشند را انطباق خواهد داد.

همچنین برنامه‌نویس می‌تواند عباراتی را انطباق دهد که با عبارت موردنظرش مطابقت نداشته باشد این کار را به راحتی با متمم نمودن عبارت داخل کلاس می‌تواند انجام دهد برای مثال [^a5]، این عبارت به این معناست که هر چیزی را غیر از ۵ مطابقت بدء.

(\) backslash

بدون شک مهم‌ترین فرا کاراکتر در عبارات باقاعده این کاراکتر است که به آن backslash می‌گویند. یکی از کاربردهای این کاراکتر از بین بردن خاصیت فرا کاراکتری است بدین معنا که برای مثال اگر برنامه‌نویس می‌خواهد تا کاراکتر [را انطباق دهد با استفاده از این کاراکتر می‌تواند خاصیت فرا کاراکتری آن را از بین برداشته باشد.] یا []

همچنین بعضی از عبارات ترکیبی معنادار نیز با این کاراکتر آغاز می‌شوند در زیر تعدادی از این عبارات را معرفی می‌نماییم

\d	اعداد ددهی را انطباق می‌دهد؛ که معادل کلاس [0-9] است.
\D	کاراکترهایی که شامل اعداد غیر ددهی می‌شوند را انطباق می‌دهد؛ که معادل کلاس [^0-9] است.
\s	کاراکترهایی که شامل whitespace می‌شوند را انطباق می‌دهد؛ که معادل کلاس [\t\n\r\f\v] است.
\S	کاراکترهایی که شامل whitespace نمی‌شوند را انطباق می‌دهد؛ که معادل کلاس [^ \t\n\r\f\v] است.
\w	کاراکترهایی الفبایی عددی (alphanumeric) را انطباق می‌دهد؛ که معادل کلاس [a-zA-Z0-9_] است.
\W	کاراکترهایی الفبایی عددی (alphanumeric) نباشند را انطباق می‌دهد؛ که معادل کلاس [^a-zA-Z0-9_] است.

همچنین برنامه‌نویس می‌تواند از این عبارات ترکیبی در داخل یک کلاس از عبارات باقاعده نیز استفاده نماید.



(.) Dot

این کاراکتر به معنای انطباق هر عبارتی بهغیراز عبارت خط جدید است این فرا کاراکتر تنها زمانی مورداستفاده قرار می‌گیرد که برنامهنویس نوع کاراکتر برایش اهمیت نداشته باشد برای مثال اگر شما به دنبال کلمات دوحرفی باشید که با ۰ خاتمه می‌یابند می‌توانید عبارت باقاعده خود را بهصورت ۰ بنویسید این عبارت کلماتی مانند do, so, go... را برای شما انطباق می‌دهد.

کاراکترهای تکرار

پیدا کردن عبارات با ساختار مشخص و محتوای تغییرپذیر یکی از بارزترین برتری‌های عبارات باقاعده نسبت به متدهای موجود برای پیمایش رشته‌ها است. برای مثال از فرا کاراکتر * برای نشان دادن تکرار استفاده می‌شود. نحوه استفاده از این فرا کاراکتر به این صورت است که بعد از یک کاراکتر می‌آید و نشان می‌دهد که در این ساختار این کاراکتر صفر یا بی‌نهایت بار تکرار داشته باشد برای مثال اگر عبارت باقاعده ما بهصورت bo*k باشد این عبارت مقادیر bk یا bok یا boook را با این ساختار مطابقت می‌دهد. البته موتور مطابقت دهنده پایتون برای تکرار کاراکترها محدودیت ایجاد نموده که معادل محدودیت یک مقدار عددی در C است. (که البته مقدار بزرگی است و احتمالاً شما نمی‌خواهید چنین حجمی از حافظه را به این کار اختصاص دهید) توجه نمایید که استفاده از ساختار تکرار * هزینه زیادی برای برنامه شما خواهد داشت به این صورت که موتور تطبیق‌دهنده سعی می‌نماید تا تمام حالت‌های ممکن را امتحان نماید. به مثال زیر توجه نمایید: فرض کنید که عبارت باقاعده a^{bcd} داریم حال می‌خواهیم عبارت abc_b را با این ساختار بررسی نماییم:

- ترجمه تحتاللفظی عبارت باقاعده به این صورت است که جمله موردنظر ما باید با کاراکتر a آغاز و با کاراکتر b خاتمه بیابد همچنین کلاس [bcd] نیز می‌تواند بین این دو کاراکتر صفتاً بی‌نهایت بار تکرار شود
- 1- کاراکتر a با ابتدای عبارت ما مطابقت دارد.
 - 2- به سراغ کلاس * [bcd] می‌رود و تمامی حالت‌هایی را که می‌تواند برای آن امتحان می‌نماید.
 - 3- موتور انطباق‌دهنده سعی می‌نماید تا کاراکتر b را انطباق دهد اما کاراکتر b به موقعیتش بهعنوان آخرین کاراکتر ثبت گردیده ولی جمله همچنان ادامه دارد پس این عبارت قابل قبول نیست.
 - 4- یک نسخه پشتیبان از آن تهیه می‌کند،

مقداری که مطابقت داده می‌شود abc_b است؛ که نشان می‌دهد موتور انطباق‌دهنده چگونه دورترین حالت‌های ممکن را بررسی می‌نماید و اگر مطابقتی پیدا نکرد این کار را تا انتهای رشته تکرار می‌نماید. دیگر فرا کاراکتری که برای تکرار موردادستفاده قرار می‌گیرد کاراکتر + است این کاراکتر بدان معناست که یک یا بی‌نهایت بار تکرار می‌تواند وجود داشته باشد. زمانی از این فرا کاراکتر استفاده می‌شود که برنامهنویس بخواهد



عمل تکرار حداقل یکبار اتفاق افتاده باشد. برای مثال اگر عبارت باقاعده ما به صورت $bo+k$ باشد این عبارت مقادیر bok یا $book$ را با این ساختار مطابقت می‌دهد.

? علامت سؤال یکی دیگر از قواعد تکرار در عبارات باقاعده است این کاراکتر بدان معناست که عبارتی که بر آن اعمال شده می‌تواند وجود داشته باشد یا وجود نداشته باشد برای مثال اگر عبارت باقاعده ما $bo?k$ باشد عبارات bok و bk با این ساختار مطابقت دارد.

پیچیده‌ترین ساختار تکرار در عبارات باقاعده ساختار $\{m,n\}$ است که m و n دو عدد طبیعی مثبت می‌باشند و بدان معناست که ساختار موردنظر حداقل m و حداکثر n بار می‌تواند اجرا شود همچنین اگر مقدار n خالی باشد بدان معناست که محدودیتی ازلحاظ تعداد وجود ندارد. بهترین مثال برای فهمیدن کارکرد این ساختار پیاده‌سازی ساختارهای قبلی تکرار است به جدول زیر توجه نمایید

?	+	*
[0,1]	[1,]	[0,]

با دقت به مثال بالا در ک مناسبی از این ساختار پیدا خواهد کرد.

| این فرا کاراکتر به معنی یا است و ساختار آن به صورت کامل شبیه or است. به این صورت که در عبارت $A | B$ مقدارهایی که معادل A یا B باشند مطابقت داده می‌شوند.

^ این فرا کاراکتر نشان‌دهنده ابتدای یک خط است. مگر اینکه پرچم چندخطی (MULTILINE) مقداردهی شده باشد. در این صورت این فرا کاراکتر تنها مشخص کننده ابتدای رشته خواهد بود. اگر این فرا کاراکتر را در ابتدای ساختار عبارت باقاعده خود قرار دهیم بدان معناست که ساختار ما حتماً باید در ابتدای خط قرار داشته باشد این بدان معناست که مثلاً ساختار $From | Eternity$ در عبارت

From Here to Eternity

دارای نتیجه است ولی همان ساختار در عبارت

Here From to Eternity

هیچ مقداری را در برنمی‌گیرد.

| این ساختار ازلحاظ رفتاری کاملاً مانند فرا کاراکتر ^ عمل می‌کند به این معنا که نشان‌دهنده ابتدای خط است.

\$ این فرا کاراکتر نشان‌دهنده انتهای یک خط است. اگر این فرا کاراکتر را در انتهای عبارت باقاعده خود قرار دهیم بدان معناست که عبارت مذکور حتماً باید در انتهای خط قرار گرفته باشد این بدان معناست که مثلاً ساختار $\$$ در عبارت:

' {block}'



دارای نتیجه است ولی در عبارت

```
' {block}'
```

هیچ مقداری را برنمی‌گرداند ولی در عبارت

```
' {block} \n'
```

دارای نتیجه است.

- Z این ساختار از لحاظ رفتاری مانند فرا کاراکتر \$ عمل می‌کند به این معنا که نشان‌دهنده انتهای خط است.
- b این ساختار نمایان کننده یک محدوده برای یک کلاس است به این صورت که ابتدا و انتهای یک عبارت را مشخص می‌نماید برای مثال عبارت b\bclass در جمله 'no class at all' دارای انطباق است ولی در جمله‌های one subclass is' یا 'the declassified algorithm' نمی‌باشد.
- B این ساختار متمم ساختار قبل است به این معنی که تنها زمانی تطابق صورت می‌پذیرد که عبارت موردنظر تطابق نداشته باشد.

حال که با ساختارهای عبارات باقاعده آشنا شدید می‌خواهیم با معرفی توابع پایتون به سراغ مثال‌های عملی برویم.

در حقیقت عبارات باقاعده در پایتون بهوسیله کلاس re.compile() به صورت الگویی از یک شیء ترجمه می‌شود که متدهایی برای کار با آن شیء در پایتون در نظر گرفته شده است. در ادامه این بحث به معرفی این متدها خواهیم پرداخت به مثال زیر توجه نمایید:

```
import re
p = re.compile('^\d+')
print p
>>>
<_sre.SRE_Pattern object at 0x7efd68113880>
```

همان‌طور که مشاهده می‌نمایید عبارت موردنظر به صورت یک شیء ترجمه شده و مقدار P در مثال بالا یک شیء است حال شما با استفاده از متدهای این شیء می‌توانید با آن کار کنید.

match اولین متدهی که در اینجا معرفی می‌شود این متده است این متده مشخص می‌نماید که آیا انطباقی در ابتدای رشته موردنظر وجود دارد یا نه.

search این متده یک رشتہ را بررسی می‌نماید و محل نقاطی را که تطابق رخداده مشخص می‌نماید.



Findall

این متدهایی را بررسی کرده و تمام عباراتی که در آنها انتظامی رخداده را به صورت یک لیست برمی‌گرداند به مثال زیر توجه نمایید:

```
import re
p = re.compile('d+ ')
print p.findall('12 drummers drumming, 1 piper piping, 100 lords a-leaping')

>>>
['12', '1', '100']
```

همان‌طور که مشاهده نمودید یک لیست از عبارات انتظامی داده شده برای ما چاپ گردید.

finditer

این متدهایی را پیمایش نموده و تمام عباراتی که در آنها انتظامی رخداده را پیدا و آنها را به صورت یک شیء که مجموعه‌ای از داده‌ها است (iterator) برمی‌گرداند.

متدهای search و match اگر هیچ انتظامی را پیدا نکنند مقدار None را برمی‌گردانند ولی اگر موفق به پیدا کردن انتظامی شوند یک نمونه از شیء MatchObject برمی‌گردانند که شامل اطلاعاتی نظیر آدرس ابتداء و انتهای عبارت انتظامی داده شده و... است.

```
import re
p = re.compile('d+')
m = p.match('string goes here')
if m:
    print 'Match found: ', m.group()
else:
    print 'No match'

>>>
No match
```

همان‌طور که در مثال بالا مشاهده می‌نمایید به دلیل اینکه هیچ انتظامی رخداده وارد قسمت else می‌شود و آن قسمت را اجرا می‌نماید.

شیء MatchObject نیز خود دارای متدهایی است که به برنامه‌نویس اجازه می‌دهد تا با مقادیر آن کار نماید متدهای این شیء عبارت‌اند از:

group
مقداری که انتظامی داده شده را برمی‌گرداند به مثال زیر توجه نمایید:



```

import re
p = re.compile('\d+ ')
match = p. match('12 drummers drumming')
print p
print match.group()

>>>
<_sre.SRE_Pattern object at 0x7fac868d0030>
12

```

start

موقعیت ابتدای زیرشته انطباق داده شده را برمی‌گرداند.

end

موقعیت انتهای زیرشته انطباق داده شده را برمی‌گرداند.

span

یک تاپل به صورت (انتهای، ابتدای) از موقعیت زیرشته انطباق داده شده را برمی‌گرداند.

به مثال زیر توجه نمایید:

```

import re
p = re.compile('\d+ ')
match = p.findall('12 drummers drumming')
print match.start()
print match.end()
print match.span()

>>>
0
2
(0, 2)

```

همان‌طور که مشاهده نمودید در این مثال کاربرد متدها کاملاً مشخص است.

حال که با متدهای شیء MatchObject آشنا شدیم یک مثال برای `finditer` را بررسی می‌نماییم:



```

import re
p = re.compile('\d+ ')
iterator = p.finditer('12 drummers drumming, 1 pipers piping, 100 lords a-
leaping')
for match in iterator:
    print match.group()
    print match.span()

>>>
12
(0, 2)
1
(22, 23)
100
(39, 42)

```

همان‌طور که مشاهده می‌نمایید با استفاده از حلقه for به تمام زیرشته‌های انطباق داده شده دسترسی پیدا نمودیم.

توجه داشته باشید که در زبان پایتون این امکان فراهم گردیده که بدون نیاز با ترجمه الگو اقدام با استفاده از توابع تطبیق الگو در ای زبان نمایید. برای این کار برنامه‌نویس می‌تواند از ماثولهای match(), search(), استفاده نماید به مثال زیر توجه نمایید: findall(), sub()

```

import re
print re.match(r'From\s+', 'Fromage amk')
print re.match(r'From\s+', 'From amk Thu May 14 19:12:10 1998').span()

>>>
None
(0, 5)

```

همان‌طور که مشاهده می‌نمایید بدون اینکه الگوی تطبیق خود را ترجمه نماییم اقدام به تطبیق الگو نمودیم. البته اتفاقی که در پشت پرده این ماثولهای رخ می‌دهد دقیقاً همان ترجمه الگو است که آن را در کامپیوتر شما انجام می‌دهد. این که برنامه‌نویس کدامیک از این روش‌ها را برای تطبیق الگوی خود استفاده نماید بستگی به نوع برنامه و عملی که انجام می‌دهد دارد اگر برنامه تنها در مواردی محدود احتیاج به تطبیق الگو داشته باشد استفاده از روش دوم مناسب‌تر است ولی اگر برنامه شما احتیاج دارد که الگوهای بسیاری را در طول برنامه بررسی نماید یا یک الگو چندین بار در نقاط مختلف برنامه بررسی نماید روش اول روش مناسب‌تری است.

پرچم‌های هم گردان (compilation flags)

این پرچم‌ها به برنامه‌نویس این امکان را می‌دهد تا بعضی از جنبه‌های توابع تطبیق الگوی خود را تغییر دهد. این پرچم‌ها در ماثول re قابل دستیابی می‌باشند و دارای دو نوع آدرس کوتاه و بلند می‌باشند به این معنی که مثلاً



برای غیرفعال سازی حساسیت به حروف کوچک و بزرگ شما می‌توانید از آدرس کوتاه re.I یا آدرس طولانی ترش re.IGNORECASE استفاده نماید.

در جدول زیر لیست این پرچم‌ها را مشاهده می‌نمایید

شامل تمام کاراکترها و همچنین خط جدید می‌شود.	DOTALL, S
حساس بودن به حروف کوچک و بزرگ را غیرفعال می‌نماید.	IGNORECASE, I
انطباق را برای زبان محلی سیستم شما فعال می‌سازد.	LOCALE, L
انطباق چندخطی را فعال می‌نماید.	MULTILINE, M
Verbose را فعال می‌کند که می‌تواند الگوهای را به صورت واضح‌تر و قابل فهم‌تری سازمان‌دهی نماید.	VERBOSE, X
پایگاه داده Unicode خود را برای مازول RE فعال می‌نماید.	UNICODE, U

به مثال زیر توجه نمایید

```
charref = re.compile(r"""
&#[#] # Start of a numeric entity reference
(
  0[0-7]+ # Octal form
  | [0-9]+ # Decimal form
  | x[0-9a-fA-F]+ # Hexadecimal form
)
; # Trailing semicolon
""", re.VERBOSE)
```

همان‌طور که مشاهده می‌نمایید با استفاده از پرچم VERBOSE توانستیم توضیحات خود را در میان کد قرار دهیم که این امر به هرچه واضح‌تر شدن الگوی ما بسیار کمک می‌نماید.

بیشتر اوقات ما از تطبیق الگو چیزی بیشتر از اینکه متوجه شود آیا الگوی خاصی در رشتہ موردنظرش وجود دارد انتظار داریم. ما معمولاً احتیاج داریم تا زیرشته‌هایی مطابق با الگویمان را به دست آوریم یا آن‌ها را از رشتہ خود حذف نماییم یا اینکه آن‌ها را تغییر دهیم. تطبیق الگو معمولاً با تجزیه الگوی خود به زیرگروه‌ها برای تطبیق دادن اجزای مورد علاقه شما مورداً استفاده قرار می‌گیرد و رشتہ موردنظر را کالبدشکافی می‌نماید. گروه‌ها با فرا کاراکترهای ” و ” مشخص می‌شوند. معنی و کاربرد این دو کاراکتر بسیار به معنی و کاربرد آن‌ها در ریاضیات نزدیک است. آن‌ها می‌توانند به صورت تودر تو تعریف شوند و همچنین با استفاده از فرا کاراکترهای تکرار آن‌ها را مدیریت نمود.

در پایتون زیرگروه‌ها به صورت از چپ به راست و از بیرونی‌ترین به داخلی‌ترین مقداردهی می‌شوند و با متدهای group() در شیء MatchObject قابل دسترسی می‌باشند همچنین در صورتی که برنامه‌نویس از گروه‌بندی



استفاده نموده باشد با استفاده از متدها groups() میتواند به یک تاپل که شامل اجزای گروهش است دسترسی داشته باشد. به مثالهای زیر توجه نمایید:

```
import re
p = re.compile(' (a(b)c)d')
match = p.match('abcd')
if match:
    print match.groups()
    print match.group(2,1,2)
else:
    print 'Nothing Found'

>>>
('abc', 'b')
('b', 'abc', 'b')
```

همان‌طور که در مثال قبل مشاهده نمودید شما میتوانید با استفاده از خروجی متدها group به تعدادی از حالت‌های انطباق به وسیله اندیس‌ها دسترسی داشته باشید اصلاح رشته‌ها در عبارات باقاعدۀ

تا اینجا آموختید که چگونه عبارات را در رشته‌ها جستجو نمایید ولی گاهی اتفاق می‌افتد که بخواهید عبارات انطباق داده شده را اصلاح نمایید یا اینکه آن‌ها حذف نمایید. در پایتون این امکان وجود دارد شما با استفاده از متدهای این زبان میتوانید تغییرات موردنظر خود را بر روی رشته‌ها اعمال نمایید در ادامه به معرفی این متدها خواهیم پرداخت

به وسیله این متدها میتوان رشته را به اجزای موردنظر خود را تفکیک نمایید.	split()
پیدا کردن تمام زیررشته‌هایی که با الگوی شما انطباق دارند و جایگزین نمودن آن‌ها با یک زیررشته دیگر.	sub()
یک نمونه از متدهای sub به اهمان توانایی است با این فرق که مقداری که برمی‌گرداند یک تاپل شامل دو عضو است که یکی رشته جدید و دیگری تعداد زیررشته‌های جایگزین شده است.	subn()

به مثالهایی که در رابطه با این متدها در ادامه آمده است توجه نمایید:

```
import re
p = re.compile(r'\W+ ')
print p.split('This is a test.')

>>>
['This', 'is', 'a', 'test', '']
```



همان‌طور که مشاهده نمودید تمام کاراکترهای الفبایی عددی که طول آن‌ها بیشتر از صفر بود را تطبیق و مجزا نمود.

```
import re
p = re.compile('blue|white|red('
print p.sub('colour', 'blue socks and red shoes')

>>>
colour socks and colour shoes
```

در این مثال نیز زیرشته‌های موردنظر خود را با رشته موردنظرمان جایگزین نمودیم:

```
import re
p = re.compile('blue|white|red('
print p.subn('colour', 'blue socks and red shoes')

>>>
('colour socks and colour shoes', 2)
```

همان‌طور که مشاهده نمودید خروجی ما داری دو مقدار است که یکی رشته جدید و دیگری تعداد حالت‌هایی که انطباق در آن‌ها رخداده است.

در زبان پایتون عبارات باقاعدۀ بسیار مفید و قدرتمند می‌باشند ولی در پاره‌ای از موارد رفتار آن‌ها دور از انتظار برنامه‌نویس است از این‌رو باید در استفاده از آن‌ها بسیار دقت نمود.



فصل 6

چند نخی Multi Threading

چند نخی یا multi threading به معنای توانایی یک برنامه در تقسیم شدن به چند تار (زیر برنامه) است که می‌توانند جداگانه و در عین حال همزمان توسط رایانه اجرا شوند به این ترتیب نرمافزار تعامل بهتری با کاربر خواهد داشت. نخ‌ها در حقیقت تکه کدهایی هستند که در برنامه به صورت موازی اجرا خواهند شد و به برنامه‌نویس این امکان را می‌دهند چندین کار را به صورت همزمان در برنامه خود انجام دهد.

نکته قابل توجه در این نوع از برنامه‌نویسی این است که چند نخی ارتباط مستقیم با سیستم‌عامل شما دارد به این معنا که سیستم‌عامل شما باید قابلیت اجرای برنامه‌های چند نخی را داشته باشد برای مثال ویندوز و اکثر سیستم‌عامل‌های خانواده لینوکس و... این توانایی را دارند.

اجرا کردن یک نخ در پایتون مانند اجرا کردن یک برنامه به وسیله کد است با این تفاوت که مزیت‌های زیر را نیز خواهد داشت:

- چندین نخ در داخل یک نخ می‌توانند داده‌ها و ارتباطات خود را خیلی راحت‌تر از زمانی که هر کدام از آن‌ها به تنهایی یک پردازش محسوب می‌شوند بین یکدیگر به اشتراک بگذارند.
- گاهی اوقات نخ‌ها به صورت پردازش‌های light-weight فراخوانی می‌شوند بنابراین آن‌ها به حافظه سربار (memory overhead) زیادی احتیاج نخواهند داشت که این امر باعث بالا رفتن کیفیت کد برنامه‌نویس می‌شود.

اولین مازولی که برنامه‌نویس برای تعریف یک نخ باید با آن آشنا باشد مازول thread است که امکانات سطح پائینی را برای کار کردن با نخ‌ها در اختیار برنامه‌نویس قرار می‌دهد برای اینکه برنامه‌نویس یک نخ را بسازد لازم است تا از متدهای start_new_thread به شکل زیر استفاده نماید:

```
thread.start_new_thread (function, args[, kwargs])
```

که ورودی‌های آن نام تابع موردنظر برنامه‌نویس که می‌خواهد در نخ قرار بگیرد و آرگومان‌های ورودی آن که به صورت یک تاپل می‌باشند خواهد بود به مثال زیر توجه نمایید:



```

import thread
import time

# Define a function for the thread
def print_time(threadName, delay):
    count = 0
    while count < 5:
        time.sleep(delay)
        count += 1
        print "%s: %s" % (threadName, time.ctime(time.time()))

# Create two threads as follows
try:
    thread.start_new_thread(print_time, ("Thread-1", 2,))
    thread.start_new_thread(print_time, ("Thread-2", 4,))
except:
    print "Error: unable to start thread"

while 1:
    pass

>>>
Thread-1: Mon Jan 24 13:14:50 2011
Thread-2: Mon Jan 24 13:14:52 2011
Thread-1: Mon Jan 24 13:14:52 2011
Thread-1: Mon Jan 24 13:14:54 2011
Thread-2: Mon Jan 24 13:14:56 2011
Thread-1: Mon Jan 24 13:14:56 2011
Thread-1: Mon Jan 24 13:14:58 2011
Thread-2: Mon Jan 24 13:15:00 2011
Thread-2: Mon Jan 24 13:15:04 2011
Thread-2: Mon Jan 24 13:15:08 2011

```

همان‌طور که در مثال قبل مشاهده نمودید دو نخ با نام‌های thread-1 و thread-2 ایجاد کردیم که اولی بعد از 2 ثانیه و دومی بعد از 4 ثانیه تأخیر اقدام به چاپ تاریخ می‌کنند و هر کدام این کار را 5 بار تکرار می‌کنند البته ماژول thread مازول بسیار خوبی برای کار با نخ‌هاست ولی در تعداد نخ‌هایی که می‌تواند با آن‌ها کار کند محدودیت دارد در زیر چند تابع از این ماژول را معرفی می‌نماییم

خطاهای مربوط به نخ را برمی‌گرداند.	Error()
یک نوع از شیء lock است.	LockType()
یک وقفه در نخ اصلی ایجاد می‌نماید تا زیر نخ‌ها بتوانند از یک تابع خاص استفاده نمایند.	interrupt_main()
باعث ایجاد خطای systemexit می‌شود و اگر نتواند آن را پیدا کند نخ را متوقف می‌سازد.	exit()
یکشی جدید از نوع lock را برمی‌گرداند.	allocate_lock()



هویت نخهایی که در حال حاضر در حال اجرا هستند را برمی‌گرداند که به صورت یک عدد مثبت است.	get_ident()
---	-------------

ماژول دیگری که در این زمینه وجود دارد ماژول `threading` است که دسترسی سطح بالاتری را نسبت به ماژول `thread` در اختیار برنامه‌نویس قرار می‌دهد همچنین تابع این ماژول `thread` توانایی بیشتری در مقابل ماژول `thread` برای کنترل نخها در اختیار برنامه‌نویس قرار می‌دهد. در ادامه با تعدادی از توابع این ماژول بیشتر آشنا خواهیم شد

تعداد نخهایی را که در حال حاضر فعال می‌باشند را برمی‌گردانند که این عدد برابر است با تعداد فهرستی که تابع <code>enumerate()</code> برمی‌گرداند.	<code>active_count()</code> <code>activeCount()</code>
یک تابع تولیدکننده که یک متغیر شیء از نوع <code>condition</code> را برمی‌گرداند. یک متغیر <code>condition</code> به یک یا چندین نخ این امکان را می‌دهد تا منتظر بمانند تا زمانی که توسط نخ دیگری فراخوانی شوند.	<code>Condition()</code>
یکشی از <code>Thread</code> فعلی را برمی‌گرداند که به نخی که این تابع را صدا کرده است. اگر نخ فراخوانی کننده به صورت مستقیم با ماژول <code>threading</code> ساخته نشده باشد یکشی مجازی از آن نخ (<code>dummy thread</code>) که از نظر دسترسی به توابع محدود گشته بازگردانده می‌شود.	<code>current_thread()</code> <code>currentThread()</code>
یک لیست از تمامی اشیائی از نوع <code>thread</code> که در حال حاضر فعال هستند را برمی‌گرداند. این لیست شامل نخهای کمکی نخهای مجازی که توسط <code>current_thread</code> ساخته شده‌اند و همچنین نخهای اصلی است ولی شامل نخهایی که کارشان به پایان رسیده یا هنوز آغاز به کار نکرده‌اند است.	<code>Enumerate()</code>
یک تابع تولیدکننده که یکشی جدید از نوع <code>event</code> را به ما برمی‌گرداند. هر رویداد یک پرچم را مدیریت می‌کند که می‌تواند با استفاده از متدهای <code>set()</code> و <code>clear()</code> مقدارش <code>true</code> شود و با متدهای <code>wait()</code> و <code>clear()</code> مقدارش را <code>false</code> کرد همچنین متدهای <code>true()</code> و <code>false()</code> نخ را متوقف می‌کند و صبر می‌کند تا مقدار پرچم <code>true</code> شود.	<code>Event()</code>
یک تابع تولیدکننده که یکشی ابتدایی از نوع <code>lock</code> را برمی‌گرداند. وقتی یک نخ آن را در دست می‌گیرد متعاقباً متوقف خواهد شد تا زمانی که دوباره آزاد شود و ای آزادسازی توسط هر نخ دیگری می‌تواند محقق شود.	<code>Lock()</code>
یک تابع تولیدکننده که یکشی جدید از نوع <code>lock</code> را برمی‌گرداند. این	<code>RLock()</code>



<p>شیء باید توسط نخی که آن را ایجاد کرده است آزاد شود. زمانی که یک نخ چنین شیئی را ایجاد می‌کند شاید نخ مشابهی همین شیء را ایجاد کند بدون اینکه متوقف شده باشد بنابراین هر نخ به ازای هر بار ایجاد این شیء باید یکبار آن را آزاد نماید.</p>	
<p>یک تابع تولیدکننده که یک شیء جدید از نوع semaphore را بر می‌گرداند. یک semaphore عددی را مدیریت می‌کند که نشان‌دهنده تعداد فراخوانی کلاس () است. همچنین با هر بار فراخوانی() acquire() این عدد کاهش پیدا می‌نماید و با تولید هر مقدار جدیدی این عدد افزایش می‌یابد. همچنین اگر لازم باشد() acquire() متوقف می‌شود تا زمانی که عدد semaphore منفی نباشد. مقدار پیش‌فرض value عدد یک است. (در علوم رایانه سمافور به متغیری اطلاق می‌شود که در محیط‌های همروند برای کنترل دسترسی فرایندها به منابع مشترک به کار می‌رود. سمافور می‌تواند به دو صورت دودویی (که تنها دو مقدار صحیح و غلط را دارا است (و یا شمارنده اعداد صحیح باشد. از سمافور برای جلوگیری از ایجاد مسئلهٔ مسابقه میان فرایندها استفاده می‌گردد. به این ترتیب، اطمینان حاصل می‌شود که در هر لحظه تنها یک فرایند به منبع مشترک دسترسی داشته و می‌تواند از آن بخواهد یا بنویسد).</p>	Semaphore([value])
<p>یک تابع تولیدکننده که یک شیء جدید از نوع bounded semaphore را بر می‌گرداند. یک bounded semaphore عدد سمافور را بررسی می‌کند تا اطمینان حاصل کند که از مقدار اصلی‌اش کمتر نیست. اگر این اتفاق بیفتد آنگاه خطای ValueError رخ خواهد داد. بیشتر اوقات سمافورها برای محافظت کردن از منابعی با ظرفیت محدود دسترسی استفاده می‌شوند. تعدد استفاده از سمافور در برنامه‌ها ممکن است در برنامه ایجاد مشکل نماید. مقدار پیش‌فرض value عدد یک است.</p>	BoundedSemaphore([value])
<p>یک تابع را به عنوان تعییب‌کننده برای تمامی نخ‌هایی که از آن مازول threading ساخته شده‌اند مستقر می‌نماید. در حقیقت این تابع هر نخ را قبل از اینکه متدها run برایش اجرا شود به تابع() sys.settrace() ارجاع می‌دهد.</p>	settrace(func)
<p>یک تابع را به عنوان profile function برای تمامی نخ‌هایی که از آن مازول threading ساخته شده‌اند مستقر می‌نماید. در حقیقت این تابع هر نخ را قبل</p>	setprofile(func)



از اینکه متدهای run برایش اجرا شود به تابع sys.setprofile() ارجاع می‌دهد.	stack_size([size])
---	--------------------

در ادامه به بررسی کلاس‌های این مژول خواهیم پرداخت

وظیفه این کلاس نمایش داده نخ محلی (thread-local data) است. داده نخ محلی داده‌ای است که مقادیر آن مخصوص به آن نخ است. برای مدیریت داده نخ محلی کافیست یک نمونه از این کلاس بسازید و مقادیر موردنظر خود را در آن‌ها مقداردهی نمایید.	local
این کلاس یک نخ از کنترل را نمایش می‌دهد. این کلاس می‌تواند به طور امن در حالتی محدود به زیر کلاس مشتق شود.	Thread
نخی که یک تابع را بعد از یک بازه زمانی مشخص اجرا می‌کند.	Timer

شیء Thread

این کلاس فعالیتهایی را که در حال اجرا در یک نخ کنترلی مجزا هستند را نمایش می‌دهد. دو راه برای تشخیص فعالیتها وجود دارد، شناسایی اشیاء قابل صدا شدن در سازنده (constructor) و یا لغو (override) کردن متدهای run در یک زیر کلاس. هیچ متدهایی (سازنده‌ها در این مورد استثنای می‌باشند) نباید در زیر کلاس‌ها لغو شوند. به زبان دیگر تنها سازنده و متدهای run را می‌توان در این کلاس لغو (override) نمود. زمانی که یک شیء از این کلاس ساخته شد، با فراخوانی متدهای start و alive در خود را آغاز می‌نماید. از زمانی که نخ‌ها فعالیت خود را آغاز نمودند وضعیتشان به صورت alive در می‌آید؛ و زمانی که کار متدهای run به پایان می‌رسد این وضعیت را از دست می‌دهند. از متدهای is_alive() می‌توان برای بررسی وضعیت alive بودن یک نخ استفاده نمود.

همچنین نخ‌های دیگر می‌توانند متدهای join و run را فراخوانی نمایند. این کار باعث می‌شود تا نخی که متدهای فراخوانی شده منتظر بماند تا زمانی که نخ فراخوانی کننده به پایان برسد.

هر نخ دارای نامی است که برنامه‌نویس می‌تواند آن را به سازنده ارسال نماید و از طریق صفت name به آن دسترسی داشته باشد یا آن را تغییر دهد. همچنین یک نخ می‌تواند به عنوان یک نخ کمکی (daemon thread) علامت‌گذاری شود این بدان معناست که برنامه پایتون زمانی که به پایان می‌رسد تمام نخ‌های کمکی به صورت خودکار پایان می‌یابد. با استفاده از خاصیت daemon می‌توان یک نخ را به عنوان نخ کمکی علامت‌گذاری نمود. به طور کلی متدهای این کلاس را در جدول زیر مشاهده می‌نمایید:

با عرض آغاز فعالیت نخ می‌شود. این متدهای در صورتی که بیش از یکبار در هر شیء فراخوانی شود خطای RuntimeError را ایجاد می‌نمایند.	start()
--	---------



این متد فعالیت نخ را نمایش می‌دهد.	run()
صبر می‌نماید تا کار نخ به پایان برسد.	join([timeout])
یک مقدار از نوع رشته‌ای که تنها هدف یک نخ را مشخص می‌نماید و از لحاظ معنایی هیچ کاربردی ندارد. همچنین این امکان وجود دارد که چندین نخ مختلف از یک نام استفاده نمایند.	getName() setName() name
شناسه هر نخ (thread identifier) یک مقدار عددی بزرگ‌تر از صفر است که تا زمانی که با متد start آغاز به کار نکرده باشد مقدارش None است.	ident
وضعیت یک نخ را در رابطه با زندگی‌بودنش نشان می‌دهد.	is_alive() isAlive()
یک مقدار بولی که نخ را به عنوان یک نخ کمکی علامت‌گذاری می‌نماید.	isDaemon() setDaemon() daemon

شیء Lock

یک Lock اصلی یک هماهنگ ساز (synchronization) اولیه است و این بدان معناست که در زمان قفل شدن یک نخ، قابل مدیریت با نخ منفرد دیگری نخواهد بود. در حال حاضر این شیء در پایتون پایین‌ترین سطح در هماهنگ‌سازی اولیه که به‌طور مستقیم توسط مازول توسعه یافته thread اجرا می‌شود را اجرا می‌نماید. این شیء دو متد اصلی دارد که در زیر درباره آن‌ها توضیح خواهیم داد

یک نخ را متوقف می‌سازد یا به‌اصطلاح قفل می‌نماید. زمانی که این متد بدون آرگومان ورودی صدا زده می‌شود نخ متوقف می‌شود تا زمانی که نخ بار دیگر راهاندازی شود و مقدار true را برمی‌گرداند. زمانی که این متد با آرگومان ورودی با مقدار true صدا زده شود نخ متوقف می‌شود و مقدار true برگردانده می‌شود. زمانی که این متد با آرگومانی با مقدار false صدا زده شود نخ متوقف نمی‌شود. اگر این متد بدون آرگومان ورودی صدا زده شود مقدار false را برمی‌گرداند در غیر این صورت نخ را متوقف نموده و مقدار true را برمی‌گرداند.	Lock.acquire()
یک نخ متوقف شده را آزاد می‌نماید. اگر نخ شما در حالت lock نباشد این متد هیچ کاری انجام نمی‌دهد. این متد هیچ مقداری را برنمی‌گرداند.	Lock.release()



شیء Lock

یک قفل بازگشتی (reentrant) که این قابلیت را دارد که چندین بار فراخوانی شود و ازنظر عملکرد تفاوتی با قفل معمولی ندارد. اگر بخواهیم دقیق‌تر شویم، این شیء برای رساندن مفهوم مالکیت داشتن بر نخ‌ها در سطوح بازگشتی مورداستفاده قرار می‌گیرد.

شیء Condition

مقدار این شیء همیشه در ارتباط با بعضی از انواع lock‌ها می‌باشد؛ که می‌توان آن را برایش ارسال یا به صورت پیش‌فرض برایش تنظیم نمود. این شیء دو متدهای متدهای منتظرشان در شیء lock صدا زده می‌شوند. همچنین متدهای دیگری نیز دارند که در ادامه به آن‌ها اشاره خواهیم نمود:

زیربنای یک قفل را مشخص می‌نماید. این متدهای متد منتظرش در زیربنای یک قفل فراخوانی می‌شود. مقداری که این متغیر بر می‌گرداند همان مقداری است که متد منتظرش بر می‌گرداند.	acquire()
زیربنای یک قفل را آزاد می‌نماید. این متدهای متد منتظرش در زیربنای یک قفل فراخوانی می‌شود. این متدهای مقداری را برنمی‌گردانند.	release()
نخ به حالت انتظار می‌رود تا زمانی که خطای رخ دهد یا زمان در نظر گرفته شده به پایان برسد. اگر نخی که این متدهای فراخوانی نموده در زمان فراخوانی در حالت قفل نباشد خطای runtimeerror به وقوع خواهد پیوست.	wait([timeout])
یک نخ را از حالت انتظار بیدار می‌نماید. اگر نخی که این متدهای فراخوانی نموده در زمان فراخوانی در حالت قفل نباشد خطای runtimeerror به وقوع خواهد پیوست.	notify()
تمامی نخ‌هایی را که در حالت انتظار به سر برند را بیدار می‌نماید. عملکرد این متدهای مانند متدهای notify است با این تفاوت که برای تمامی نخ‌هایی که در وضعیت انتظار هستند اعمال می‌شود.	notify_all()
	notifyAll()

شیء Semaphore

این شیء یکی از قدیمی‌ترین شکل‌های هندسی در هماهنگ‌سازی تاریخچه علم کامپیوتر است. کار اصلی یک سمافور مدیریت شمارندهای داخلی می‌باشد که با هر بار فراخوانی متدهای acquire کاهش و با هر بار فراخوانی متدهای release افزایش می‌یابند. مقدار این شمارنده هرگز نباید کمتر از صفر شود زمانی که مقدار یک شمارنده صفر باشد و متدهای acquire فراخوانی شود سمافور اجازه start شدن را به نخ نمی‌دهد و به اصطلاح آن را قفل



می‌نماید و منتظر می‌ماند تا متدهای release فراخوانی شود. در حقیقت سمافورها برای کنترل دسترسی‌ها مورد استفاده قرار می‌گیرند.

آرگومان ورودی این کلاس عددی است که به شمارنده سمافور اختصاص داده می‌شود. به صورت پیش‌فرض این عدد برابر با ۱ است.	<i>class threading.Semaphore()</i>
یک سمافور را ایجاد می‌نماید. در صورتی که مقدار شمارنده سمافور بزرگ‌تر از صفر باشد تنها یک مقدار از آن کم می‌نماید در غیر این صورت پردازش را به حالت انتظار می‌برد تا زمانی که پردازه دیگری کارش به پایان برسد و مقدار شمارنده را افزایش دهد.	<i>acquire()</i>
مقدار شمارنده داخلی سمافور را یک واحد افزایش می‌دهد. اگر مقدار شمارنده سمافور صفر باشد و پردازشی در حالت انتظار باشد آن پردازش را نیز از حال انتظار به حالت اجرا درمی‌آورد.	<i>release()</i>

شیء Event

یکی از ساده‌ترین راه‌های ارتباط برقرار کردن بین نخ‌ها است.

این شیء پرچم‌های (flag) داخلی را مدیریت می‌نماید به این صورت که با فراخوانی متدهای set و clear مقدار آنها را با فراخوانی متدهای is_set و isSet مقداردهی می‌نماید. در ادامه به بررسی متدهای این شیء خواهیم پرداخت:

یک پرچم داخلی را ایجاد می‌نماید.	<i>class threading.Event</i>
اگر مقدار پرچم موردنظر true باشد این متدهای set و clear را به شما برمی‌گرداند.	<i>is_set()</i> <i>isSet()</i>
مقدار یک پرچم داخلی را معادل true قرار می‌دهد.	<i>set()</i>
مقدار یک پرچم داخلی را معادل false قرار می‌دهد.	<i>clear()</i>
یک نخ را متوقف می‌نماید تا زمانی که مقدار پرچم با true مقداردهی شود یا زمانی که در آرگومان ورودی مقداردهی گردیده به پایان برسد.	<i>wait([timeout])</i>

شیء Timer

این کلاس یک عمل مشخص را بعد از مدت‌زمانی مشخص شده انجام می‌دهد. در حقیقت این کلاس یک زیر کلاس (subclass) از کلاس thread است. این کلاس با فراخوانی متدهای start و cancel شروع به کار می‌نماید و با فراخوانی متدهای cancel به کار خود پایان می‌دهد. به مثال زیر توجه نمایید:



```
import threading

def hello():
    print "hello, world"

t = threading.Timer(5.0, hello)
t.start() # after 5 seconds, "hello, world" will be printed

>>>hello, world
```

همان‌طور که در مثال بالا مشاهده نمودید تکه کد نوشته شده بعد از ۵ ثانیه اقدام به اجرای تابع hello می‌نماید.



7 فصل

برنامه‌نویسی شبکه Network Programming

برای کسب توانایی در ساخت برنامه‌هایی که بتوانند تحت شبکه و استانداردهای موجود برای ساخت شبکه کار کنند نیاز است اطلاعات مقدماتی در مورد شبکه‌های کامپیوتری داشته باشید در ابتدا ما به معرفی کوتاه در این زمینه می‌پردازیم.

سرور (server) چیست؟

سرور که در برخی متون فارسی کارساز یا خادم هم نامیده می‌شود، به برنامه‌ای رایانه‌ای گفته می‌شود که خدمات خود را به دیگر برنامه‌های رایانه‌ای (و کاربران آن‌ها) در همان رایانه یا در رایانه‌های دیگر ارائه می‌کند. به رایانه‌ای که چنین برنامه‌ای روی آن اجرا شود نیز کارساز گفته می‌شود.

کلاینت (client) چیست؟

کارخواه، یک نرمافزار کاربردی یا سامانه است که از طریق یک شبکه به خدمات یک سامانهٔ رایانه‌ای دیگر به نام سرور یا کارساز دسترسی دارد. این عبارت نخستین بار برای افزارهایی که قابلیت اجرای برنامه‌های مستقل خودشان را نداشتند اما می‌توانستند با رایانه‌های دور از طریق شبکه برهم‌کنش داشته باشند، به کار رفت. مدل کارخواه-کارساز امروزه نیز در اینترنت به کار می‌رود. مرورگرهای وب، کارخواههایی هستند که به کارسازهای وب وصل می‌شوند و صفحات وب را برای نمایش بازیابی می‌کنند.

میزبان (Host) چیست؟

در شبکه‌های رایانه‌ای، رایانه‌ای است که به اینترنت - یا به طور کلی تر - به هر شبکهٔ داده‌ای، متصل است. یک میزبان شبکه می‌تواند اطلاعات و نیز نرمافزار کارخواه client و یا کارساز server (server) را میزبانی کند. هر میزبان اینترنتی یک نشانی آی‌پی یکتا دارد که بخش نشانی میزبان را نیز شامل می‌شود. نشانی میزبان، یا به طور دستی توسط مدیر رایانه وارد می‌شود و یا به طور خودکار در آغاز به‌وسیلهٔ پروتکل پیکربندی پویای میزبان (DHCP) اختصاص می‌یابد.

هر میزبانی، یک گرهٔ شبکهٔ فیزیکی (دستگاه شبکه) است، ولی هر گره شبکه فیزیکی، یک میزبان نیست. به گره‌های شبکه مانند مودم‌ها و سوئیچ‌های شبکه نشانی‌های میزبان اختصاص نمی‌یابد و به عنوان میزبان در نظر گرفته نمی‌شوند. به دستگاههایی چون چاپگرهای شبکه و مسیریاب‌های سخت‌افزاری، نشانی آی‌پی میزبان



اختصاص می‌یابد، اما چون آن‌ها رایانه‌های همه‌منظوره نیستند، به‌طور عمومی گاهی به عنوان میزبان در نظر گرفته نمی‌شوند.

سوکت (socket) چیست؟

اصلی‌ترین عامل در یک ارتباط شبکه‌ای Socket است که اعمال شبکه را به صورت خواندن و نوشتمن در یک فایل شبیه‌سازی نموده است. سوکت در اصل مانند یک کانال ارتباطی است که میان دونقطه ایجاد شده و ارتباط را برقرار می‌سازد. برای داشتن یک ارتباط شبکه‌ای باید یک سوکت ایجاد کنیم که لازمه این کار داشتن آدرس IP، نوع پروتکل ارتباط (TCP / UDP) و شماره Port مقصده است.

پروتکل (protocol) چیست؟

در شبکه‌های کامپیوتر به مجموعه قوانینی اطلاق می‌گردد که نحوه ارتباط را قانونمند می‌نماید. نقش پروتکل در کامپیوتر نظیر نقش زبان برای انسان است. برای مطالعه یک کتاب نوشته شده به فارسی می‌بایست خواننده شناخت مناسبی از زبان فارسی را داشته باشد. به‌منظور ارتباط موفقیت‌آمیز دو دستگاه در شبکه نیز باید هر دو دستگاه از یک پروتکل مشابه استفاده کنند.

در علوم کامپیوتر و ارتباطات، پروتکل عبارت است از استاندارد یا قراردادی که برای ارتباط میان دونقطه برقرار می‌شود. پروتکل اتصال بین دو نود، انتقال داده بین آن دو تبادلات میان را ممکن کرده و آن را کنترل می‌کند. پروتکل در ساده‌ترین حالت می‌تواند به عنوان قوانین ادارهٔ منطق، ترکیب و همزمانی ارتباطات در نظر گرفته شود. پروتکل‌ها ممکن در سخت‌افزار یا نرم‌افزار یا ترکیبی از این دو پیاده‌سازی شوند. پروتکل در پایین‌ترین سطح رفتار اتصال سخت‌افزاری را تعریف می‌کند. معنی لغوی پروتکل مجموعه قوانین است.

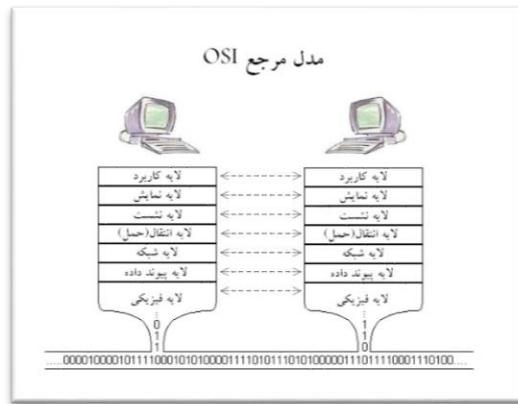
مدل OSI

مدل مرجع اتصال داخلی سیستم‌های باز که گاه «مدل هفت لایه ASI» نیز خوانده می‌شود، توصیفی مفهومی و مجرد از لایه‌هایی است که دو یا چند سیستم مخابراتی یا شبکه‌های کامپیوتری از طریق آن به یکدیگر متصل می‌شوند. این مدل خود یک معماری شبکه نیست چون هیچ سرویس یا پروتکلی در آن تعریف نمی‌شود.

اگر صفرها و یک‌ها همین‌طور پشت سر هم قرار بگیرند اطلاعات منتقل نمی‌شود. بلکه باید درباره نحوه ارسال و شکل اطلاعات توافق شود. برای این منظور در شبکه پروتکل‌هایی به وجود آمد OSI. یکی از مدل‌های استاندارد و پذیرفته شده است که برای استفاده پروتکل‌ها در شبکه به کار می‌رود.

در مدل OSI شبکه را به هفت لایه تقسیم می‌کنند که این لایه‌ها مستقل از هم هستند و هر لایه کاری را انجام می‌دهد وظیفه دارد که به لایه‌های بالاتر سرویس بدهد. ارتباط بین دو لایه Interface نام دارد و لایه‌های متناظر پروتکل متناظر دارند. این هفت لایه به صورت زیر می‌باشند:





لایه‌ها در مدل OSI (Physical)

این لایه با مشخصات فیزیکی محیط‌های انتقال مانند کابل‌ها و کانکتورها در ارتباط است در این لایه مشخصات مکانیکی، فیزیکی و عملیاتی محیط انتقال همچون ولتاژ الکتریکی، سرعت و فاصله مجاز تعریف شده است.

لایه دو (Data)

این لایه فراهم‌کننده ارتباط با لایه فیزیکی و محیط انتقال است. این لایه با آدرس‌های فیزیکی کار می‌کند و بر اساس آن سخت‌افزارها یا لایه Data Link را پیدا می‌کنند این لایه وظیفه کشف خطا و تشخیص صحت داده رسیده را نیز بر عهده دارد. از پروتکل‌های این لایه می‌توان FDDI, Ethernet نام برد.

لایه سه (Network)

وظیفه اصلی لایه شبکه مسیریابی و هدایت بسته بین مسیرهای مختلف تا رسیدن به مقصد است. این لایه برخلاف لایه دوم که با آدرس‌های فیزیکی کار می‌کند از IP Address جهت شناخت مقصد استفاده می‌کند Router که یکی از ابزارهای مسیریاب است در این لایه عمل می‌کند. پروتکل‌های IP,IPX در این لایه عمل می‌کنند.

لایه چهار (Transport)

از وظایف لایه انتقال می‌توان، اطمینان از رسیدن داده‌ها به مقصد فرستادن پیغام پس از دریافت بسته کنترل و ارسال بسته‌ها و ارسال مجدد بسته‌های خراب و تقسیم بسته‌های بزرگ به قطعات کوچک قابل انتقال نام برد.



لایه پنجم (Session)

لایه جلسه وظیفه برقراری تماس بین مبدأ و مقصد و قطع آن پس از اتمام ارتباط را بر عهده دارد. همچنین در صورتی که حین انتقال ارتباط قطع شود این لایه وظیفه دارد مجدداً ارتباط را برقرار کند تا ارسال داده‌ها از سرگرفته شود.

لایه ششم (Presentation)

وظیفه این لایه ترجمه کدهای رسیده از لایه‌های پایین‌تر به Format قابل استفاده توسط برنامه‌های کاربردی مانند Mail است. در این لایه عملیات رمزگذاری و فشرده‌سازی انجام می‌شود.

لایه هفتم (Application)

در این لایه برنامه‌های کاربردی قرار دارند که کاربران با آن‌ها در تعامل هستند مانند انواع Browser‌ها، برنامه‌های ارسال Mail و غیره.

TCP/IP

مدل TCP/IP یا مدل مرجع اینترنتی که گاهی به مدل DOD (وزارت دفاع)، مدل مرجع ARPANET نامیده می‌شود، یک توصیف خلاصه لایه TCP/IP برای ارتباطات و طراحی پروتکل شبکه کامپیوتر است. در سال ۱۹۷۰ به وسیله DARPA ساخته شده که برای پروتکل‌های اینترنت در حال توسعه مورد استفاده قرار گرفته است، ساختار اینترنت دقیقاً به وسیله مدل TCP/IP منعکس شده است.

مدل اصلی TCP/IP از ۴ لایه تشکیل شده است. هرچند که سازمان IETF استانداردی که یک مدل ۵ لایه‌ای است را قبول نکرده است. به‌حال پروتکل‌های لایه فیزیکی و لایه پیوند داده‌ها به وسیله IETF استاندارد نشده‌اند. سازمان IETF تمام مدل‌های لایه فیزیکی را تائید نکرده است. با پذیرفتن مدل ۵ لایه‌ای در بحث اصلی با مسئولیت فنی برای نمایش پروتکل است عجیب نیست که نمایش ۵ لایه‌ای را در آموزش بیاوریم و این امکان را می‌دهد که راجع به پروتکل‌های غیر IETF در لایه فیزیکی صحبت کنیم. این مدل قبل از مدل مرجع OSI گسترش یافته واحد و ظایف مهندسی اینترنت (IETF)، برای مدل و پروتکل‌های گسترش یافته تحت آن پاسخ‌گو است، هیچ‌گاه خود را ملزم ندانست که توسط OSI تسلیم شود. در حالی که مدل بیسیک OSI کاملاً در آموزش استفاده شده است و OSI به یک مدل ۷ لایه‌ای معرفی شده است، معماری یک پروتکل واقعی RFC ۱۱۲۲ مورداً استفاده در محیط اصلی اینترنت خیلی منعکس نشده است. حتی یک مدرک معماری IETF که اخیراً منتشر شده یک مطلب با این عنوان دارد: "لایه‌بندی مضر است." تأکید روی لایه‌بندی به عنوان محرک کلیدی معماری یک ویژگی از مدل TCP/IP نیست، اما نسبت به OSI بیشتر است.



لایه کاربرد (Application)

لایه کاربردی بیشتر توسط برنامه‌ها برای ارتباطات شبکه استفاده می‌شود. داده‌ها از برنامه در یک قالب خاص برنامه عبور می‌کنند سپس در یک پروتکل لایه انتقال جاگیری می‌شوند. ازانجایی که پشته IP بین لایه‌های Application کاربردی) و (انتقال Transport (هیچ لایه دیگری ندارد، لایه کاربردی Application می‌باشد هر پروتکلی را مانند پروتکل لایه نشست (session) و نمایش (presentation) در OSI عمل می‌کنند در بگیرد. داده‌های ارسال شده روی شبکه درون لایه کاربردی هنگامی که در پروتکل لایه کاربردی جاگیری شدن عبور می‌کنند. ازانجا داده‌ها به سمت لایه‌های پایین‌تر پروتکل لایه انتقال می‌روند. دو نوع از رایج‌ترین پروتکل‌های لایه پایینی TCP و UDP هستند. سرورهای عمومی پورت‌های مخصوصی به این‌ها دارند (HTTP) پورت ۸۰ و FTP پورت ۲۱ (رادارند...) درحالی که کلاینت‌ها از پورت‌های روزانه بی‌دوام استفاده می‌کنند. روت‌ها و سوئیچ‌ها این لایه را بکار نمی‌گیرند اما برنامه‌های کاربردی بین راه در پهنای باند این کار را می‌کنند، همان‌طور که پروتکل RSVP (پروتکل ذخیره منابع) انجام می‌دهد.

لایه انتقال (transport)

مسئولیت‌های لایه انتقال، قابلیت انتقال پیام را END-TO-END و مستقل از شبکه، به اضافه کنترل خط، قطعه‌قطعه کردن و کنترل حریان را شامل می‌شود. ارسال پیام END-TO-END یا کاربردهای ارتباطی در لایه انتقال می‌توانند جور دیگری نیز گروه‌بندی شوند:: ۱ اتصال گرا مانند TCP ۲ بدون اتصال مانند UDP لایه انتقال می‌تواند کلمه به کلمه به عنوان یک مکانیسم انتقال مانند یک وسیله نقلیه که مسئول امن کردن محتويات خود (مانند مسافران و اشیاء) است که آن‌ها را صحیح و سالم به مقصد برساند، بدون اینکه یک لایه پایین‌تر یا بالاتر مسئول بازگشت درست باشند. لایه انتقال این سرویس ارتباط برنامه‌های کاربردی به یک دیگر را در حین استفاده از پورت‌ها فراهم آورده است. ازانجایی که IP delivery یک فقط یک IP می‌تواند روی یک پروتکل اولین لایه پشته TCP/IP برای ارائه امنیت و اطمینان است. توجه داشته باشید که IP می‌تواند روی یک پروتکل ارتباط داده مطمئن امن مانند کنترل ارتباط داده سطح بالا (HDLC) اجرا شود. پروتکل‌های بالای انتقال مانند RPC نیز می‌توانند اطمینان را فراهم آورند. به طور مثال TCP یک پروتکل اتصال گر است که موضوع‌های مطمئن بی‌شماری را برای فراهم آوردن یک رشته باشد مطمئن و این آدرس دهی می‌کند: داده in order می‌رسند. داده‌ها حداقل خطاهای را برای تکراری دور ریخته می‌شوند. بسته‌های گم شده و از بین رفته دوباره ارسال می‌شوند. دارای کنترل تراکم ترافیک است SCTP. جدیدتر نیز یک مکانیسم انتقالی مطمئن و امن و اتصال گر است - رشته پیام گر است نه رشته باشد گرا مانند - TCP و جریان‌های چندگانه‌ای را روی یک ارتباط



منفرد تسهیم می‌کند؛ و همچنین پشتیبانی چند فضا را (multi-homing) نیز در مواردی که یک پایانه ارتباطی می‌تواند توسط چندین آدرس IP بیان شود. (اینترفیس‌های فیزیکی چندگانه) را فراهم می‌آورد تا اینکه اگر یکی از آن‌ها دچار مشکل شود ارتباط دچار وقفه نشود. در ابتدا برای کاربردهای تلفنی (برای انتقال VSS روی استفاده می‌شود اما می‌تواند برای دیگر کاربردها نیز مورداستفاده قرار بگیرد.

لایه شبکه (Network)

لایه شبکه مشکل گرفتن بسته‌های سرتاسر شبکه منفرد را حل کرده است. نمونه‌هایی از چنین پروتکل‌هایی ۲۵X. و پروتکل HOST/IMP مربوط به ARPANET است. با ورود مفهوم درون‌شبکه‌ای کارهای اضافی به این لایه اضافه می‌شوند از جمله گرفتن از شبکه منبع به شبکه مقصد و عموماً routing کردن و تعیین مسیر بسته‌های میان یک شبکه از شبکه‌ها را که به عنوان شبکه داخلی یا اینترنت شناخته می‌شوند را شامل می‌شود. در همه پروتکل‌های شبکه IP وظیفه اساسی گرفتن بسته‌های داده‌ای را از منبع به مقصد انجام می‌دهد IP. می‌تواند داده‌ها را از تعدادی از پروتکل‌های مختلف لایه بالاتر حمل کند. این پروتکل‌ها هر کدام توسط یک شماره پروتکل واحد و منحصر به فرد شناسایی می‌شوند: ICMP و IGMP. به ترتیب پروتکل‌های ۱ و ۲ هستند. برخی از پروتکل‌های حمل شده توسط IP (مانند ICMP) مورداستفاده برای اطلاعات تشخیص انتقال راجع به انتقالات (IP)، IGMP، IGNP (IP در بالای multicast)، RPT و OSPT. تمام پروتکل‌های مسیریابی مانند IP و OSI را ایجاد کرده‌اند. تمام پروتکل‌های مسیریابی مانند RPT نیز بخشی از لایه شبکه هستند. آنچه آن‌ها را بخشی از لایه شبکه کرده است این است که هزینه load آن‌ها (play load) درمجموع با مدیریت لایه شبکه در ارتباط است. کپسول بندی و جاگیری خاص آن به اهداف لایه‌بندی بی‌ارتباط است.

لایه ارتباط داده‌ها (Data link)

لایه ارتباط داده از متدهای برای حرکت بسته‌ها از لایه شبکه روی دو میزبان مختلف که درواقع واقعاً بخشی از پروتکل‌های شبکه نیستند، استفاده می‌کند، چون IP می‌تواند روی یک گستره از لایه‌های ارتباطی مختلف اجرا شود. پردازش‌های بسته‌های انتقال داده شده روی یک لایه ارتباطی داده شده می‌تواند در راه انداز وسایل نرم‌افزاری برای کارت شبکه به خوبی میان افزارها یا چیپ‌های ویژه کار صورت گیرد. این امر می‌تواند توابع ارتباط داده‌ها را مانند اضافه کردن یک header به منظور آماده کردن آن برای انتقال انجام دهد سپس واقعاً فرم را روی واسطه فیزیکی منتقل کند. برای دسترسی اینترنت روی یک مودم dial-up معمولاً بسته‌های IP با استفاده از PPP منتقل می‌شوند. برای دسترسی به اینترنت با پهنای باند بالا مانند ADSL یا مودم‌های کابلی PPPOE غالباً استفاده می‌شود. در یک شبکه کابلی محلی معمولاً اینترنت استفاده می‌شود و دو شبکه‌های بی‌سیم محلی



۱۱ IEEE ۸۰۲.۱۱ استفاده می‌شود. برای شبکه‌های خیلی بزرگ هردو روش PPP یعنی خطوط T-Carrier یا E-Carrier تقویت‌کننده فرم، ATM یا بسته روی SONET/SDM (POS) اغلب استفاده می‌شوند. لایه ارتباطی همچنین می‌تواند جایی که بسته‌ها برای ارسال روی یک شبکه خصوصی مجازی گرفته می‌شوند نیز باشد. هنگامی که این کار انجام می‌شود داده‌های لایه ارتباطی داده‌های کاربردی را مطرح می‌کنند و نتایج به پشت IP برای انتقال واقعی بازمی‌گردند. در پایانه دریافتی داده‌ها دوباره به پشته stack می‌آیند (یکبار برای مسیریابی و بار دوم برای VPN). لایه ارتباط می‌تواند ابتدای لایه فیزیکی که مشتمل از اجزای شبکه فیزیکی واقعی هستند نیز مرتبط شود. اجزایی مانند هاب‌ها، تکرارکننده‌ها، کابل فیبر نوری، کابل کواکیسال، کارت‌های شبکه، کارت‌های ورق دهنده host و ارتباط‌دهنده‌های شبکه مرتبط : ۴۵- (R,BNC,...) و مشخصات سطح پایینی برای سیگنال‌ها (سطح ولتاژ، فرکانس‌ها و ...)

لایه فیزیکی (Physical)

لایه فیزیکی مسئول کد کردن و ارسال داده‌ها روی واسط ارتباطی شبکه است و با داده‌ها در فرم بیت‌هایی که از لایه فیزیکی وسیله ارسال کننده (منبع) هستند و در لایه فیزیکی و دستگاه مقصد دریافت می‌شوند کار می‌کند. اترنت، SCSI، Token ring، هاب‌ها، تکرارکننده‌ها، کابل‌ها و ارتباط‌دهنده‌ها وسائل اینترنتی استانداردی هستند که روی لایه فیزیکی تابع بندی شده‌اند. لایه فیزیکی همچنین دامنه بسیاری از شبکه سخت‌افزاری مانند LAN و توبولوزی WAN و فناوری بی‌سیم (Wireless) را نیز در بر می‌گیرد.

تفاوت‌های بین لایه‌های TCP/IP و OSI

سه لایه بالایی در مدل - OSI لایه کاربردی، لایه نمایش و لایه اجلاس معمولاً درون یک لایه در مدل TCP/IP یکجا جمع شده‌اند. در حالی که بعضی از برنامه‌های کاربردی پروتکل OSI مانند X.۴۰۰ نیز باهم دیگر جمع شده‌اند، نیاز نیست که یک پشته پروتکل TCP/IP برای هماهنگ کردن آن‌ها بالای لایه انتقال باشد. برای مثال پروتکل کاربردی سیستم نائل شبکه (NFS) روی پروتکل نمایش داده خارجی (XDR) اجرا می‌شود و روی یک پروتکل با لایه اجلاس کار می‌کند و فراخوان رویه راه دور (RPC) را صدا می‌زند. مخابرات را به طور مطمئن ذخیره می‌کند، پس می‌تواند با امنیت روی پروتکل UDP اجرا شود. لایه اجلاس تقریباً به پایانه مجازی Telnet که بخشی از متن بر اساس پروتکل‌هایی مانند پروتکل‌های کاربردی مدل HTTP و SMTP است هستند مرتبط می‌شود؛ و نیز با شمارش پورت UDP و TCP که بخشی از لایه انتقال در مدل TCP/IP است مطرح می‌شود. لایه نمایش شبکه استاندارد MIME است که در HTTP و SMTP نیز استفاده می‌شود. از آنجایی که سعی برای پیشرفت پروتکل IETF به لایه‌بندی محض ربطی ندارد، بعضی از پروتکل‌های آن ممکن است برای مدل OSI متناسب باشند. این ناسازگاری‌ها هنگامی که فقط به مدل اصلی ISO ۷۴۹۸ نگاه



کنیم بیشتر تکرار می‌شوند، بدون نگاه کردن به ضمایم این مدل (مانند چارچوب مدیریتی ISO یا سازمان درونی ISO ۸۶۴۸ لایه شبکه (IONL) هنگامی که IONL و استناد چهارچوب مدیریتی مطرح می‌شوند، ICMP و IGMP، به طور مرتب به عنوان پروتکل‌های مدیریت لایه برای لایه شبکه تعریف می‌شوند. در روشی مشابه، IONL یک ساختمان برای "قابلیت‌های همگرایی" وابسته به زیر شبکه "مانند ARP و RARP را فراهم آورده است. پروتکل‌های IETF می‌توانند پشت سر هم کاربرد داشته باشند چون توسط تونل زدن پروتکل‌های مانند GRE توضیح داده می‌شوند در حالی که استناد بیسیک OSI با تونل زدن ارتباطی ندارند بعضی مفاهیم تونل زدن هنوز هم در توسعه‌های معماری OSI وجود دارند. مخصوصاً دروازه‌های لایه انتقال بدون چهارچوب پروفایل TCP/IP بین‌المللی استاندارد شده است. تلاش‌های پیشرفت دهنده مرتبط با OSI، به خاطر استفاده پروتکل‌های TCP/IP در جهان واقعی رهاسده‌اند. لایه‌ها در ادامه توضیح از هر لایه در پشت‌هه رشته IP آمده است.

لایه کاربردی لایه کاربردی بیشتر توسط برنامه‌ها برای ارتباطات شبکه استفاده می‌شود. داده‌ها از برنامه در یک قالب خاص برنامه عبور می‌کنند سپس در یک پروتکل لایه انتقال جاگیری می‌کنند.

از آنجایی که پشت‌هه IP بین لایه‌های کاربردی و انتقال هیچ لایه دیگری ندارد، لایه کاربردی باید هر پروتکلی را مانند پروتکل لایه اجلاس و نمایش در OSI عمل می‌کنند در بگیرد. داده‌های ارسال شده روی شبکه درون لایه کاربردی هنگامی که در پروتکل لایه کاربردی جاگیری شدند عبور می‌کنند. از آنجا داده‌ها به سمت لایه‌های پایین‌تر پروتکل لایه انتقال می‌روند. دو نوع از رایج‌ترین پروتکل‌های لایه پایینی TCP و UDP هستند. سرورهای عمومی پورت‌های مخصوصی به این‌ها دارند (HTTP پورت ۸۰ و FTP پورت ۲۳ را درارند و (...در حالی که کلاینت‌ها از پورت‌های روزانه بی‌دوم استفاده می‌کنند. روتراها و سوئیچ‌ها این لایه را بکار نمی‌گیرند اما برنامه‌های کاربردی بین راه در پهنه‌ای باند این کار را می‌کنند، همان‌طور که پروتکل RSVP پروتکل ذخیره منابع) انجام می‌دهد.

۳ لایه بالایی در مدل - OSI لایه کاربردی، لایه نمایش و لایه نشست معمولاً درون یک لایه در مدل TCP/IP مجتمع می‌شوند. در حالی که برخی از برنامه‌های کاربردی پروتکل OSI مانند X ۴۰۰ نیز با یکدیگر جمع شده‌اند، نیاز نیست که یک پشت‌هه پروتکل TCP/IP برای یکپارچه کردن آن‌ها بالای لایه انتقال باشد. برای نمونه پروتکل کاربردی سیستم نائل شبکه (NFS) روی پروتکل نمایش داده خارجی (XDR) اجرا می‌شود و روی یک پروتکل با لایه نشست کار می‌کند و فراخوان رویه راه دور (RPC) را صدا می‌زنند. RPC را مخفی می‌نمایند. مخابرات را به طور مطمئن ذخیره می‌کند، پس می‌تواند با امنیت روی پروتکل UDP اجرا شود. لایه نشست تقریباً به پایانه مجازی Telnet که بخشی از متن بر اساس پروتکل‌هایی مانند پروتکل‌های کاربردی مدل HTTP و SMTP TCP/IP هستند مرتبط می‌شود؛ و نیز با شمارش پورت UDP و TCP که بخشی از لایه انتقال در مدل TCP/IP است مطرح می‌شود. لایه نمایش شبیه استاندارد MIME که در HTTP و SMTP نیز استفاده می‌شود است. از آنجایی که تلاش برای پیشرفت پروتکل IETF به لایه‌بندی محض ربطی ندارد، برخی از پروتکل‌های آن ممکن است برای مدل OSI متناسب باشند. این ناسازگاری‌ها هنگامی که فقط به مدل اصلی OSI، ISO ۷۴۹۸ می‌شوند.



نگاه کنیم بیشتر تکرار می‌شوند، بدون نگاه کردن به ضمایم این مدل (مانند چارچوب مدیریتی ISO ۴|۷۴۹۸) یا سازمان درونی ISO ۸۶۴۸ لایه شبکه (IONL) هنگامی که IONL و مستندات چهارچوب مدیریتی مطرح می‌شوند، ICMP و IGMP، به طور مرتب به عنوان پروتکل‌های مدیریت لایه شبکه تعريف می‌شوند. در روشی مشابه، IONL یک ساختمان برای «قابلیت‌های همگرایی وابسته به زیر شبکه» مانند ARP و RARP را فراهم آورده است. پروتکل‌های IETF می‌توانند پشت سر هم کاربرد داشته باشند چون توسط تونل زدن پروتکل‌هایی مانند GRE (Generic Routing Encapsulation) شرح داده می‌شوند در حالی که مستندات پایه‌ای با تونل زدن ارتباطی ندارند برخی مفاهیم تونل زدن هنوز هم در توسعه‌های معماری OSI وجود دارند. مخصوصاً دروازه‌های لایه انتقال بدون چهارچوب پروفایل استانداردشده بین‌المللی. تلاش‌های پیشرفت دهنده مرتبط با OSI، به خاطر استفاده پروتکل‌های TCP/IP در دنیای واقعی رها شده‌اند.

سوکت در پایتون

تا اینجا اطلاعات مقدماتی در رابطه با سوکت و انواع مدل‌های شبکه‌ای پیدا نمودیم حال به سراغ برنامه‌نویسی در شبکه می‌رویم. یکی از راههایی که برنامه‌نویس می‌تواند یک ارتباط را در شبکه ایجاد نماید بازنمودن یک سوکت بر روی مقصد مورد نظر است برای این کار در پایتون کلاس socket طراحی گردیده، برای اینکه بتوانید به سراغ این توابع بروید ابتدا باید مقداری اطلاعات کلی در مورد سوکت در پایتون داشته باشید: پایتون از دو حوزه ارتباطی در شبکه استفاده می‌کند که به ترتیب عبارت‌اند از حوزه اینترنت AF_INET و حوزه یونیکس AF_UNIX که به عنوان خانواده آدرس حوزه‌ها مورد استفاده قرار می‌گیرند. آدرس‌های حوزه یونیکس به صورت یک‌رشته متنی می‌باشند که به آن‌ها مسیر محلی می‌گویند ولی در حوزه اینترنت آدرس‌ها به صورت میزبان و پورت (host, port) می‌باشند؛ که میزبان می‌تواند به صورت یک‌رشته متنی باشد که نشان‌دهنده یک آدرس معتبر برای میزبان است همچنین می‌تواند شامل یک آدرس IP که با نقطه از هم جدا شده‌اند نیز باشد. پورت نیز یک مقدار عددی از 1 تا 65535 است و نشان‌دهنده درگاهی است که سوکت برای اتصال از آن استفاده می‌نماید.

از میان انواع سوکت‌ها در پایتون دو نمونه از آن‌ها کاربرد بیشتری نسبت به سایر نمونه‌ها دارد تا جایی که گاهی از این دو نمونه به عنوان تنها نمونه‌های سوکت در زبان برنامه‌نویسی یاد می‌شود.

سوکت‌های رشته‌ای (Stram Socket)

این سوکت‌ها از نوع اتصال گرا (Connection Oriented) است که یک نوع ارتباط دوطرفه و قابل اطمینان را با رعایت ترتیب و نظارت بر خطاهای احتمالی ایجاد می‌نماید. شایان ذکر است که این ارتباط توسط پروتکل tcp پشتیبانی می‌شود.



سوکت دیتاگرام (Datagram)

این سوکت از نوع غیر اتصالی (Connectionless) است که یک نوع ارتباط دوطرفه غیر قابل اطمینان است، در این ارتباط هیچ تضمینی برای ترتیب و ارسال داده‌ها به طور کامل وجود ندارد. شایان ذکر است این ارتباط توسط پروتکل udp پشتیبانی می‌شود.

نمونه سوکت‌های بالا پرکاربردترین انواع سوکت‌ها در برنامه‌نویسی شبکه است ولی تنها نمونه‌های سوکت در پایتون نیستند. برای مثال نوع دیگری از سوکت وجود دارد به نام سوکت raw، این نوع سوکت‌ها اطلاعات را به صورت مرتب و قابل اطمینان منتقل می‌کنند. همچنین این نوع ارتباط به شما امکان ایجاد یک packet خام را می‌دهد. با استفاده از این امکان شما این اجازه را دارید که packet دلخواه خود را بسازید و ارسال نمایید.

حال زمان آن رسیده که به معرفی کلاس‌ها و توابع مربوط به کار کردن در پایتون برویم نحوه استفاده از کلاس سوکت در حالت کلی به صورت زیر است:

```
socket(IpVersion, socketType[, protocol])
```

ورودی اول نوع آدرس آی‌پی را در شبکه مشخص می‌نماید که برای مثال آی‌پی نسخه 4 یا 6 باشد ورودی دوم نوع ارتباط در یک ارتباط سوکت را (UDP یا TCP) بودن را مشخص می‌نماید ورودی سوم شماره پروتکلی را که شما برای ارتباط خود مدنظر دارید را مشخص می‌نماید این عدد می‌تواند 0 یا شماره موردنظر شما باشد توجه دارید که این مقدار یک مقدار اختیاری است و مقدار پیش‌فرض آن صفر است.

به مثال زیر توجه نمایید:

```
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

همان‌طور که مشاهده می‌نمایید یک سوکت ایجاد نمودیم حال می‌توانیم با استفاده از متدهای کلاس سوکت اقدام به ارتباط و تبادل اطلاعات نماییم برای مثال تکه کد زیر یک ارتباط با آدرس hostname و پورت portNumber برقرار می‌نماید:

```
sock.connect((hostname, portNumber))
```

در ادامه به توضیح توابع ماثول سوکت خواهیم پرداخت.

```
socket.create_connection(address[, timeout])
```



یک ارتباط به مقدار آدرس (که یک تاپل متشکل از (ip,port)) است برقرار می نماید و یک شیء از نوع سوکت را برمی گرداند ارسال مقدار متغیر timeout که یک مقدار اختیاری است باعث می شود تا در صورت برقرار نشدن ارتباط در زمان مشخص شده ارتباط fail شود.

`socket.getaddrinfo(host, port[, family[, socktype[, proto[, flags]]]])`

این تابع یک لیست با ساختار (family, socktype, proto, canonname, sockaddr) را برمی گرداند که سه آیتم اول به صورت عددی می باشند و مقادیری می باشند که از قبل به مأذول ما ارسال گردیده اند canonname یک رشته است که نشان دهنده نام رسمی آن host است همچنین این مقدار می تواند به صورت عددی از نوع ورژن های مختلف از آدرس IP نیز باشد. Sockaddr نیز یک تاپل است که اطلاعاتی را در مورد آدرس سوکت دارد است.

`socket.getfqdn([name])`

تمامی اطلاعات لازم برای نام دامنه وارد شده را برمی گرداند، همچنین اگر متغیر ورودی را به این تابع ارسال ننماییم مقداری که این تابع برمی گرداند نام دامنه برای localhost است.

`socket.gethostname(hostname)`

مقدار hostname را به معادل Ipv4 اش تبدیل می نماید که یک رشته به صورت 100.100.100.100 است همچنین اگر مقدار ورودی این تابع آدرس Ip باشد مقداری که برمی گرداند معادل همان آدرس آی پی است. این تابع از Ipv6 پشتیبانی نمی نماید.

`socket.gethostbyname_ex(hostname)`

مقدار hostname را به متناظر Ipv4 تبدیل می نماید و یک لیست به صورت (hostname, aliaslist, ipaddrlist) را برمی گرداند.

`socket.gethostname()`

این تابع نام ماشینی را که host شما بر روی آن قرار دارد را به شما برمی گرداند.

`socket.gethostbyaddr(ip_address)`

این تابع یک لیست با ساختار (hostname, aliaslist, ipaddrlist) را برمی گرداند.

`socket.getnameinfo(sockaddr, flags)`



این تابع یک آدرس سوکت را به یک تاپل دو عضوی (host,port) تبدیل می‌نماید که وابسته به تنظیمات پرچم‌ها (flags) است.

`socket.getprotobyname(protocolname)`

وظیفه این تابع تبدیل نام یک پروتکل اینترنتی به یک مقدار ثابت مناسب برای ارسال به آرگومان‌های تابع socket است. این کار زمانی کاربرد خواهد داشت که سوکت در حالت raw (SOCK_RAW) بازشده باشد. برای حالت‌های معمولی اگر پروتکل موردنظر را از قلم انداخته باشید این کار به صورت خودکار انجام می‌پذیرد.

`socket.getservbyname(servicename[, protocolname])`

وظیفه این تابع تبدیل نام یک پروتکل یا سرویس اینترنتی را به پورت متناظرش برای آن سرویس خاص است.

`socket.getservbyport(port[, protocolname])`

وظیفه این تابع تبدیل شماره یک پورت یا یک پروتکل اینترنتی به نام سرویس متناظرش برای آن سرویس خاص است.

`socket.socket([family[, type[, proto]]])`

وظیفه این تابع ایجاد یک سوکت جدید است که در ورودی خانواده آدرس سوکت، نوع و پروتکل مورداستفاده در سوکت را دریافت می‌نماید. خانواده آدرس‌ها در سوکت می‌تواند AF_INET (به صورت پیشفرض)، AF_UNIX یا AF_INET6 باشد. نوع سوکت نیز می‌تواند SOCKET_STREAM (به صورت پیشفرض) یا SOCK_DGRAM یا دیگر ثابت‌هایی که برای سوکت وجود دارد تعریف شود.

`socket.socketpair([family[, type[, proto]]])`

وظیفه این تابع ایجاد یک جفت ارتباط با دریافت آدرس و نوع ارتباط و شماره پورت است.

`socket.fromfd(fd, family, type[, proto])`

تکثیر فایل توصیفگر و ساخت یک ارتباط سوکت از روی نتایج حاصل. فامیلی آدرس و نوع سوکت و پروتکل نیز از سوکت گرفته می‌شود.

`socket.ntohl(x)`

یک نوع integer 32 بیتی را به یک میزبان بایتی سفارشی تبدیل می‌نماید. در ماشین‌ها جایی که میزبان بایتی سفارشی با شبکه بایتی سفارشی معادل هم باشند یک (no-op) رخ می‌دهد.

`socket.ntohs(x)`



مانند تابع قبلی است با این تفاوت که یک مقدار integer 16 بیتی را به یک میزبان بایتی سفارشی تبدیل می‌نماید.

`socket htonl(x)`

یک نوع integer 32 بیتی را از میزبان به شبکه بایتی سفارشی تبدیل می‌نماید. در ماشین‌ها جایی که میزبان بایتی سفارشی با شبکه بایتی سفارشی معادل هم باشند یک (no-op) رخ می‌دهد.

`socket htons(x)`

یک نوع integer 16 بیتی را از میزبان به شبکه بایتی سفارشی تبدیل می‌نماید. در ماشین‌ها جایی که میزبان بایتی سفارشی با شبکه بایتی سفارشی معادل هم باشند یک (no-op) رخ می‌دهد.

`socket.inet_aton(ip_string)`

یک مقدار رشته‌ای از آی‌پی نسخه چهار را از فرمت نقطه‌ای (برای مثال 192.168.0.1) به یک بسته 32 بیتی در فرمت باینری تبدیل می‌کند. این توانایی زمانی مفید است که برنامه شما بخواهد با یک نرمافزار که از کتابخانه استاندارد C استفاده می‌نماید و احتیاج دارد که نوع آدرسش struct باشد گفتگو نماید. اگر ساختار آی‌پی که به این تابع به عنوان آرگومان ارسال می‌شود نادرست باشد خطای socket.error ایجاد می‌شود.

`socket.inet_ntoa(packed_ip)`

یک بسته 32 بیتی از آی‌پی نسخه چهار را در فرمت باینری به یک مقدار رشته‌ای در فرمت نقطه‌ای (برای مثال 192.168.0.1) تبدیل می‌نماید. این توانایی زمانی مفید است که برنامه شما بخواهد با یک نرمافزار که از کتابخانه استاندارد C استفاده می‌نماید و احتیاج دارد که نوع آدرسش struct باشد گفتگو نماید.

`socket.inet_pton(address_family, ip_string)`

یک آدرس آی‌بی را از فرمت آدرس فامیلی خاص خود به یک بسته در فرمت باینری تبدیل می‌نماید. این توانایی زمانی مفید است که یک کتابخانه یا پروتکل شبکه برای یک شیء از نوع struct صدا زده شود. آدرس‌های فامیلی که در حال حاضر برای این کار پشتیبانی می‌شوند AF_INET و AF_INET6 می‌باشند. اگر ساختار آی‌پی که به این تابع به عنوان آرگومان ارسال می‌شود نادرست باشد خطای socket.error ایجاد می‌شود.

`socket.inet_ntop(address_family, packed_ip)`

یک بسته از آی‌پی آدرس را در فرمت باینری به استاندارد یک فامیلی آدرس تبدیل می‌نماید. این توانایی زمانی مفید است که یک کتابخانه یا پروتکل شبکه برای یک شیء از نوع struct صدا زده شود. آدرس‌های فامیلی که در حال حاضر برای این کار پشتیبانی می‌شوند AF_INET و AF_INET6 می‌باشند. اگر ساختار آی‌پی که به این تابع به عنوان آرگومان ارسال می‌شود نادرست باشد خطای socket.error ایجاد می‌شود.



`socket.setdefaulttimeout()`

مقدار پیش‌فرض `timeout` را برای یک سوکت جدید برمی‌گرداند. اگر مقدار برگردانده شده `None` باشد این بدان معناست که برای این سوکت هیچ مقداردهی نشده است. زمانی که مازول سوکت برای اولین بار ایجاد می‌شود این مقدار برابر `None` است.

`socket.setdefaulttimeout(timeout)`

مقدار `timeout` را برای یک سوکت جدید مقداردهی می‌نماید.

`socket.SocketType`

این یک نوع در پایتون است که نوع شیء سوکت را مشخص می‌نماید.

شیء سوکت (`Socket`)

شیء سوکت دارای متدهای زیر می‌باشند که در ادامه به معرفی آنها خواهیم پرداخت.

`socket.accept()`

وظیفه این متده است که در شبکه ایجاد شده در مراز یک آدرس گوشبزنگ می‌ماند تا یک ارتباط در شبکه برقرار شود.

`socket.bind(address)`

وظیفه این متده کردن یک سوکت با یک آدرس است. توجه داشته باشید که سوکت از قبل نباید مرزبندی شده باشد. به مفهوم ساده‌تر سوکت برای اینکه قابل‌شناسایی باشد احتیاج دارد تا یک نام مخصوص خود داشته باشد که این تابع این کار را برای سوکت انجام می‌دهد. (ساختار آدرس بستگی به خانواده آدرسی دارد که شما برای سوکت خود معرفی نموده‌اید).

`socket.close()`

وظیفه این متده کامل قطع ارتباطاتی است که یک سوکت ایجاد نموده است. همچنین سوکت‌ها در زباله روبی (`garbage collected`) به صورت خودکار بسته می‌شوند.

`socket.connect(address)`

یک ارتباط از طریق سوکت را به آدرس موردنظر ایجاد می‌نماید. (ساختار آدرس بستگی به خانواده آدرسی دارد که شما برای سوکت خود معرفی نموده‌اید).



`socket.connect_ex(address)`

مانند تابع قبل یک ارتباط از طریق سوکت ایجاد می‌نماید با این تفاوت که در هنگام بروز خطاها یی که تابع `connect()` در سطح-سی (C-Level) بر می‌گرداند این متدهای به جای اینکه یک خطا را ایجاد نماید یک شاخص خطا را بر می‌گرداند.

`socket.fileno()`

این متدهای فایل مفسر سوکت را بر می‌گرداند (که به صورت یک مقدار عددی کوچک است).

`socket.getpeername()`

این متدهای آدرس‌های سوکتی را که با آن در ارتباط هستند را بر می‌گرداند. کاربرد این متدهای برای پیدا کردن شماره پورت‌هایی است که سوکت به آنها در ارتباط است. توجه داشته باشید که بعضی از دستگاه‌ها از این تابع پشتیبانی نمی‌کنند.

`socket.getsockname()`

این متدهای آدرس یک سوکت را بر می‌گرداند. این توانایی مفید است که شما به شماره پورت یک سوکت نیاز دارید با استفاده از این سوکت می‌توانید به این مقدار دسترسی پیدا نمایید.

`socket.listen(backlog)`

وظیفه این متدهای منتظر ماندن برای گوش سپردن به ایجاد یک ارتباط یا ارسال اطلاعات در یک ارتباط است. آرگومان `backlog` طول صفی را که این سوکت می‌تواند با آنها ارتباط برقرار کند را مشخص می‌نماید که حداقل 1 و حداقل آن بستگی به نیاز سیستم دارد (معمولًاً 5 است).

`socket.makefile([mode[, bufsize]])`

این متدهای فایلی که به سوکت وابسته شده را بر می‌گرداند.

`socket.recv(bufsize[, flags])`

وظیفه این متدهای دریافت اطلاعات از یک سوکت است. مقداری که این تابع بر می‌گرداند یک مقدار از نوع `string` است که شامل اطلاعات دریافت شده است. آرگومان `bufsize` حداکثر سایز اطلاعاتی که در یک ارتباط قابل دریافت است را مشخص می‌نماید.

`socket.recvfrom(bufsize[, flags])`

وظیفه این متدهای مانند تابع قبل است با این تفاوت که مقداری که توسط این متدهای بزرگ‌دانده می‌شود، یک جفت اطلاعات است که به صورت (string, address) است که مقدار آرگومان اول مقداری است که توسط سوکت دریافت شده و مقدار آرگومان دوم آدرس سوکتی است که اطلاعات را ارسال نموده است.

socket.recvfrom_into(buffer[, nbytes][, flags])

وظیفه این متدهای دریافت اطلاعات از یک سوکت است با این تفاوت نسبت به متدهای قبل که به جای اینکه یک مقدار رشته‌ای جدید تولید کند و اطلاعات را در آن بریزد اطلاعات را مستقیماً در بافر ذخیره می‌نماید. خروجی این متدهای یک جفت به صورت (nbytes, address) است که آرگومان اول تعداد بایت‌های اطلاعات دریافت شده را نمایش می‌دهد و آرگومان دوم آدرس سوکتی است که اطلاعات را ارسال نموده است.

socket.recv_into(buffer[, nbytes][, flags])

وظیفه این متدهای دریافت اطلاعات در یک سایز مشخص از یک سوکت است. همچنین اطلاعات دریافت شده را در بافر ذخیره می‌نماید که سریع‌تر از ذخیره آن‌ها در یک رشته است. اگر مقدار nbytes مشخص نشده باشد یا برابر صفر باشد دریافت اطلاعات به اندازه سایز بافری که Available است انجام می‌پذیرد.

socket.send(string[, flags])

وظیفه این متدهای ارسال اطلاعات به یک متدهای دیگر است. سوکت مورد نظر باید با سوکتی که می‌خواهد اطلاعات را برایش ارسال نماید در ارتباط باشد. خروجی این متدهایی است که آن‌ها را ارسال نموده است. این تابع هیچ تعهدی در قبال ارسال تمامی اطلاعات ندارد و برنامه خودش باید این وظیفه را به عهده بگیرد.

socket.sendall(string[, flags])

وظیفه این متدهای ارسال اطلاعات به یک متدهای دیگر است. سوکت مورد نظر باید با سوکتی که می‌خواهد اطلاعات را برایش ارسال نماید در ارتباط باشد. برخلاف متدهای قبلی این متدهای وظیفه ارسال تمامی اطلاعات را به عهده می‌گیرد تا زمانی که خطایی رخ دهد. در صورتی که همه‌چیز با موفقیت انجام شود این متدهای هیچ مقداری را برنامه‌گراند و در زمانی که خطایی رخ دهد خروجی این متدهای معادل خطایی است که به موقع پیوسته است. شایان ذکر است زمانی که خطایی رخ می‌دهد روند ارسال اطلاعات متوقف شده و اینکه چه مقدار از اطلاعات با موفقیت ارسال شده قابل تشخیص نیست.

socket.sendto(string[, flags], address)



وظیفه این متدهای ارسال اطلاعات به یک متدهای دیگر است با این فرق که لزوماً نباید با سوکتی که می‌خواهد اطلاعات را برایش ارسال نماید در ارتباط باشد. تنها کافیست آدرس سوکت مقصد را برایش مشخص نمایید تا اطلاعات را برای سوکت موردنظر ارسال نماید.

`socket.setblocking(flag)`

وظیفه این متدهای مسدود کردن یا رفع مسدودیت از یک سوکت است. اگر مقدار پرچم برابر صفر باشد سوکت رفع مسدودیت می‌شود در غیر این صورت اگر پرچم برابر یک شود سوکت مسدود می‌شود. تمام سوکت‌ها در ابتدا در وضعیت مسدود بودن به سر می‌برند. اگر برنامه‌نویس `recv()` را زمانی صدا نماید که سوکت در وضعیت رفع مسدودیت به سر می‌برد این متدهای هیچ مقداری را دریافت نخواهد نمود، همچنین اگر در این وضعیت `send()` نیز صدا زده شود این تابع نمی‌تواند اطلاعات را به صورت منظم دسته‌بندی نماید و خطای خطا رخ می‌دهد. در وضعیت مسدود بودن فراخوانی سوکت مسدود مس شود تا متدهای در حال اجرا کارش را به پایان برساند.

`socket.settimeout(value)`

وظیفه این متدهای تنظیم یک `timeout` برای مسدود بودن یک سوکت در حین انجام یک عملیات است. مقدار آرگومان `value` یک عدد اعشاری غیر منفی است همچنین برنامه‌نویس می‌تواند مقدار آن را با `None` نیز مقداردهی نماید. اگر قبل از اینکه عملیات سوکت به پایان برسد این زمان تمام شود سوکت اقدام به ایجاد خطای `timeout` می‌نماید. همچنین اگر مقدار آرگومان `value` را با مقدار `null` مقداردهی نماید بدان معناست که هیچ محدودیت زمانی وجود ندارد همچنین اگر مقدار این آرگومان را با صفر مقداردهی نمایند بدان معناست که سوکت در وضعیت رفع مسدودیت قرار دارد.

`socket.gettimeout()`

مقداری که این متدهای گرداند مدت زمانی است که به عنوان `timeout` برای یک سوکت خاص ثبت گردیده است، که یک مقدار اعشاری است. همچنین اگر مقداری ثبت نشده باشد مقدار `None` برگردانده می‌شود.

`socket.shutdown(how)`

وظیفه این متدهای خاتمه دادن به یک یا دو طرفه در یک ارتباط است. اگر مقدار آرگومان `how` برابر با `SHUT_RD` باشد دریافت اطلاعات غیرفعال می‌شود، اگر مقدار آرگومان `how` برابر `SHUT_WR` باشد ارسال اطلاعات غیرفعال می‌شود و اگر مقدار آرگومان `how` برابر با `SHUT_RDWR` باشد هم دریافت و هم ارسال اطلاعات هر دو غیرفعال می‌شوند.

حال که متدهای سوکت را به صورت کامل توضیح دادیم در ادامه چند مثال عملی را بررسی می‌نماییم:



ابتدا قسمت Server برنامه را می‌نویسیم

```
HOST = 'Server' # Symbolic name meaning all available interfaces
PORT = 50007 # Arbitrary non-privileged port
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen(1)
conn, addr = s.accept()
print 'Connected by', addr
while 1:
    data = conn.recv(1024)
    if not data: break
    conn.send(data)
conn.close()
```

حال به قسمت Client می‌پردازیم

```
# Echo client program
import socket

HOST = 'ServerAddress' # The remote host
PORT = 50007 # The same port as used by the server
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))
s.send('Hello, world')
data = s.recv(1024)
s.close()
print 'Received', repr(data)
```

همان‌طور که مشاهده می‌نمایید ابتدا یک برنامه به عنوان سرویس‌دهنده ساخته‌ایم که بر روی پورت شماره 50007 به صورت گوش‌به‌زنگ برای ایجاد یک ارتباط باقی می‌ماند زمانی که ارتباطی برقرار می‌شود نام ارتباط گیرنده را نمایش می‌دهد همچنین اطلاعاتی را که از طریق سر. یس گیرنده برایش ارسال می‌گردد را در یک متغیر ذخیره و بعد از پایان ارسال آن را دوباره برای سرویس‌گیرنده ارسال می‌نماید.

قسمت دوم کد مربوط به سرویس‌گیرنده است در این قسمت یک اتصال با سرور ایجاد می‌شود و اقدام به ارسال اطلاعات برای آن می‌نماید (توجه داشته باشید که مقدار متغیر Host باید آدرس برنامه سرویس‌دهنده شما باشد) همچنین مقداری را که از طرف سرویس‌دهنده به عنوان پاسخ دریافت می‌کند را نیز چاپ می‌نماید.

همان‌طور که مشاهده نمودید برای یک ارتباط ساده تنها از تعداد محدودی از توابع و متدهای نامبرده شده در بالا استفاده می‌شود حال یک مثال پیچیده‌تر را بررسی می‌نماییم:



```

import socket

# the public network interface
HOST = socket.gethostbyname(socket.gethostname())

# create a raw socket and bind it to the public interface
s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_IP)
s.bind((HOST, 0))

# Include IP headers
s.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)

# receive all packages
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)

# receive a package
print s.recvfrom(65565)

# disabled promiscuous mode
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)

```

در مثال بالا یک sniffer بسیار ساده با استفاده از سوکت raw را در سیستم‌عامل ویندوز طراحی نموده‌ایم. تنها توجه داشته باشید برای اینکه اسکریپت بالا کار کند احتیاج دارد تا در وضعیت مدیر اصلی باشد.

برنامه‌نویسی در اینترنت با پایتون

یکی دیگر از توانائی‌های پایتون برنامه‌نویسی در محیط اینترنت و کار کردن با پروتکل‌های معروف اینترنت از قبیل FTP، POP3، HTTP و... است در ادامه با توانائی‌های پایتون در این زمینه بیشتر آشنا می‌شویم.

پروتکل FTP که مخفف (File Transfer Protocol) است به معنای قرارداد انتقال پرونده است که در شبکه‌های رایانه‌ای برای انتقال پرونده‌ها درون شبکه مورد استفاده قرار می‌گیرد.

در میان رایانه‌های میزبان، FTP به‌طور ویژه یک قرارداد متداول برای دادوستد فرمان‌ها و پرونده‌ها در هر شبکه پشتیبان از قرارداد اینترنت و قرارداد هدایت انتقال (TCP/IP) (مانند اینترنت و اینترانت) است. درگاه (Port) پیش‌فرض برای خدمات FTP، درگاه 21 و برای انتقال داده از درگاه 20 استفاده می‌شود.

در یک انتقال FTP دو رایانه دخیل است، یک کارساز (Server) و یک کاربر (Client). برنامه‌های کارساز FTP را اجرا می‌کند و درخواست پذیرش در شبکه را با رایانه دیگر (کاربر) مطرح می‌کند. رایانه کاربر برنامه‌های کاربری FTP را اجرا و یک ارتباط با کارساز برقرار می‌نماید.



هنگامی که یک ارتباط برقرار می‌شود کاربر می‌تواند تعدادی از برنامه‌ها را تغییر دهد (دست‌کاری محدود)، مانند بارگذاری پرونده در کارساز و بارگیری پرونده از آن، یا باز نامیدن یا حذف پرونده‌ها در کارساز... درواقع همهٔ بسترهای رایانه‌ای از FTP پشتیبانی می‌کنند و به هر ارتباط رایانه‌ای که بر اساس قرارداد هدایت انتقال/قرارداد اینترنت باشد صرف‌نظر از این‌که از چه سامانهٔ عاملی استفاده می‌شود، اگر رایانه‌ها اجازهٔ دسترسی به FTP را داشته باشند، این اجازه را می‌دهد که در پرونده‌های رایانهٔ دیگر در این شبکه تغییراتی ایجاد نماید.

برای اینکه یک کاربر بتواند از این پروتکل استفاده نماید ابتدا باید به یک کارساز ارتباط برقرار نماید و سپس با استفاده از نام کاربری و کلمه عبور به آن سامانه وارد شود، البته بدون با توجه به تنظیمات کارساز این امکان نیز وجود دارد که بدون وارد شدن به سامانه کارساز نیز بتوان از آن پرونده دریافت نمود. در زبان پایتون تمام امکاناتی که یک کاربر برای استفاده از یک سامانه احتیاج دارد طراحی و تعبیه گردیده است که در ادامه با آن‌ها بیشتر آشنا خواهید شد.

برای استفاده از این پروتکل در پایتون ماژول `ftplib` طراحی گردیده که تمامی نیازهای برنامه‌نویس را برای استفاده از این پروتکل فراهم نموده در حالت کلی یک ارتباط با پروتکل FTP در زبان پایتون به صورت زیر است:

```
from ftplib import FTP
f=FTP('ftp.mozilla.org')
print f.login('anonymous', 'guess@who.org')
f.retrlines('LIST')

>>>
230-
230-ftp.mozilla.org / archive.mozilla.org - files are in /pub.mozilla.org
230-
230-Notice: This server is the only place to obtain nightly builds and
needs to
230-remain available to developers and testers. High bandwidth servers that
230-contain the public release files are available at
ftp://releases.mozilla.org/
230-If you need to link to a public release, please link to the release
server,
230-not here. Thanks!
230-
230-Attempts to download high traffic release files from this server will
get a
230- "550 Permission denied." response.
230 Login successful.
-rw-r--r-- 1 ftp ftp 528 Nov 01 2007 README
-rw-r--r-- 1 ftp ftp 560 Sep 28 2007 index.html
drwxr-xr-x 35 ftp ftp 4096 Nov 10 17:07 pub
```



همان‌طور که در مثال قبل مشاهده نمودید کار کردن با این کتابخانه بسیار راحت است به‌راحتی به یک کارساز وصل شدیم و لیست دایرکتوری‌های موجود در کارساز را دریافت نمودیم حال که یک دید کلی پیدا نمودید به معنی توابع و مازول‌های پایتون خواهیم پرداخت

```
class ftplib.FTP([host[, user[, passwd[, acct[, timeout]]]]])
```

یک نمونه از کلاس Ftp را بر می‌گرداند زمانی که آرگومان host را ارسال نموده باشید متدهای connect(host) به ساخته می‌شود. زمانی که آرگومان user ارسال شود متدهای login(user,passwd,acct) می‌شود.

```
class ftplib.FTP_TLS([host[, user[, passwd[, acct[, keyfile[, certfile[, timeout]]]]]]])
```

یک زیر کلاس برای کلاس Ftp که از Tls پشتیبانی می‌نماید در RFC 4217 به صورت کامل شرح داده شده است. به صورت عادی با درگاه شماره 21 ارتباط برقرار می‌نماید. برای اینکه مد امنیتی برای این ارتباط برقرار شود باید متدهای prot_p فراخوانی شود.

شیء FTP

متدهای شیء ftp کاربردهای متفاوتی دارند بعضی از آن‌ها برای کار کردن با فایل‌های متنی و گروهی نیز برای کار کردن با فایل‌های باینری می‌باشند.

```
FTP.set_debuglevel(level)
```

وظیفه این متدهای تعريف یک سطح برای debuglevel است که میزان خروجی قابل‌نمایش را کنترل می‌نماید. مقدار آن به صورت پیش‌فرض 0 است که معنای این است که هیچ خروجی وجود ندارد. اگر این مقدار یک باشد بدان معناست که به ازای هر درخواست تنها یک خط خروجی وجود خواهد داشت و اگر مقدار این

```
FTP.connect(host[, port[, timeout]])
```

به آدرس میزبان و درگاهی که معرفی شده است وصل می‌شود. شماره درگاه پیش‌فرض برای این متدهای 21 است که درگاه شناخته شده برای پروتکل FTP است. این تابع برای هر نمونه نباید بیشتر از یکبار صدا زده شود. بقیه متدها تنها زمانی قابل استفاده می‌باشند که بعد از این تابع صدا زده شوند بدین معنا که اول باید اتصال برقرار شود.

```
FTP.getwelcome()
```



فصل 8

الگوهای طراحی در پایتون

در مهندسی نرم افزار، الگوی طراحی^{۲۶} یک راه حل عمومی قابل تکرار برای مشکلات متداول در زمینه طراحی نرم افزار است. الگوی طراحی، یک طراحی تمام شده نیست که به صورت مستقیم بتواند تبدیل به کد منبع یا ماشین شود؛ بلکه، یک توضیح یا قالب برای حل یک مسئله در شرایط مختلف است. الگوها در واقع بهترین روش ممکن هستند که یک برنامه نویس می‌تواند در هنگام طراحی یک برنامه برای حل مشکلات از آن‌ها استفاده کند. الگوهای طراحی شیء‌گرا نوعاً نشان‌دهنده روابط و تعامل‌ها بین کلاس‌ها و شیء‌ها هستند، بدون این‌که کلاس‌ها یا اشیا نهایی برنامه را مشخص کند. الگوی طراحی که در خود وضعیت‌های تغییرپذیر دارد، شاید مناسب زبان‌های برنامه نویسی تابعی نباشد. هم‌چنان، در بعضی از زبان‌ها که برای حل یک مسئله راه حل‌های آماده از پیش تعریف شده وجود دارد، استفاده از بعضی الگوهای طراحی ممکن است برای زبان‌های غیر شیء‌گرا مناسب نباشد. به همین ترتیب، الگوهای طراحی شیء‌گرا ممکن است برای زبان‌های غیر شیء‌گرا مناسب نباشد.

الگوهای طراحی با توجه به کاربردها و اهدافی که به آن منظور ارائه شده‌اند، به سه دسته عمومی تقسیم می‌شوند

 الگوهای ایجادی²⁷

 الگوهای ساختاری²⁸

 الگوهای رفتاری²⁹

که در هریک از آن‌ها سعی می‌کنیم مهم‌ترین‌شان را به اختصار توضیح دهیم.

الگوهای ایجادی

به الگوهای طراحی که برای حل مشکلات مربوط به ایجاد اشیا در نرم افزار ارائه شده‌اند، الگوهای ایجادی یا می‌گویند. متداول‌ترین الگوهای طراحی سازنده عبارت‌اند از: Creational

Design Pattern²⁶

Creational Design Patterns²⁷

Structural Design Patterns²⁸

Behavioral Design Patterns²⁹

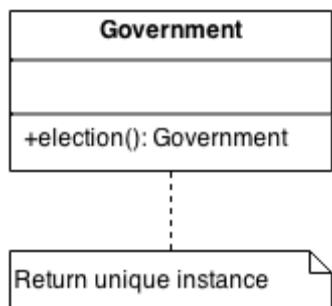


Singleton

این الگو ایجاد شیء از یک کلاس را محدود می‌سازد. این الگو زمانی مفید است که در سرتاسر سیستم تنها به یک نمونه از آن کلاس نیاز باشد. این مفهوم عموماً به سیستم‌هایی که با یک یا تعداد محدودی نمونه بهینه‌تر کار می‌کنند، نیز تعمیم داده می‌شود. واژه آن از مفهوم ریاضی یگانه (Singleton) برگرفته شده است.

مثال در دنیای واقعی

الگوی Singleton تضمین می‌کند که از هر کلاس یک کلاس فقط یک نمونه داشته باشد و یک نقطه دسترسی جهانی به آن نمونه را فراهم می‌کند. دفتر رئیس‌جمهور یک Singleton است. قانون اساسی روش‌های انتخاب رئیس‌جمهور را مشخص می‌کند، مدت تصدی مقام را محدود می‌کند و ترتیب جانشینی را مشخص می‌کند. درنتیجه، حداکثر یک رئیس‌جمهور فعال در هر زمان مشخص وجود دارد. صرف‌نظر از هویت شخصی رئیس‌جمهور فعال، عنوان رئیس‌جمهور» یک نقطه دسترسی جهانی است که یک فرد را در دفتر ریاست جمهوری مشخص می‌کند.



روش پیاده سازی الگوی طراحی singleton

نحوه پیاده‌سازی این الگو به صورت کلی به این گونه است که کلاس باید یک تابع داشته باشد تا یک شی از آن کلاس را در صورتی که قبل از ساخته نشده است، بسازد. برای اطمینان از اینکه نمونه دیگری از این کلاس قابل ایجاد نباشد باید دسترسی به Constructor کلاس را به صورت Private تعریف کنیم. مراحل پیاده سازی سینگلتون عبارت‌اند از :

1. اضافه کردن یک متغیر Static از نوع Private به کلاس مورد نظر به منظور ذخیره‌سازی نمونه Singleton.
2. نوشتتن یک تابع Static از نوع Public برای دریافت نمونه ساخته شده Singleton (این تابع ورودی نخواهد داشت).



3. پیاده‌سازی "مقداردهی اولیه" متغیر در داخل تابع Static نوشته شده است. در این مرحله باید یک شی جدید در اولین فراخوانی ایجاد شود و مقدار آن را در متغیر Static قرار دهد. این تابع در فراخوانی‌های بعدی باید شی ایجادشده را فراخوانی کند.

4. باید Constructor مربوط به کلاس و کلاسی که از آن ارث بری شده است را Private کنیم. پس از این کار دیگر قادر نخواهیم بود در خارج از کلاس، از آن شی نمونه‌ای بسازیم و این کار فقط در همان کلاس امکان‌پذیر است.

5. در نهایت باید در کدهای خود به جای فراخوانی مستقیم Constructor سینگلتون (Direct calls) از تابع که نوشته‌ایم به این منظور استفاده کنیم. حالا به سراغ پیاده‌سازی این الگو می‌رویم. به مثال زیر دقت کنید.

```
"""
Ensure a class only has one instance, and provide a global point of
access to it.
"""

class Singleton(type):
    """
    Define an Instance operation that lets clients access its unique
    instance.
    """
    def __init__(cls, name, bases, attrs, **kwargs):
        super().__init__(name, bases, attrs)
        cls._instance = None

    def __call__(cls, *args, **kwargs):
        if cls._instance is None:
            cls._instance = super().__call__(*args, **kwargs)
        return cls._instance

class MyClass(metaclass=Singleton):
    """
    Example class.
    """
    pass

def main():
    m1 = MyClass()
    m2 = MyClass()
    assert m1 is m2

if __name__ == "__main__":
    main()
```



Abstract factory

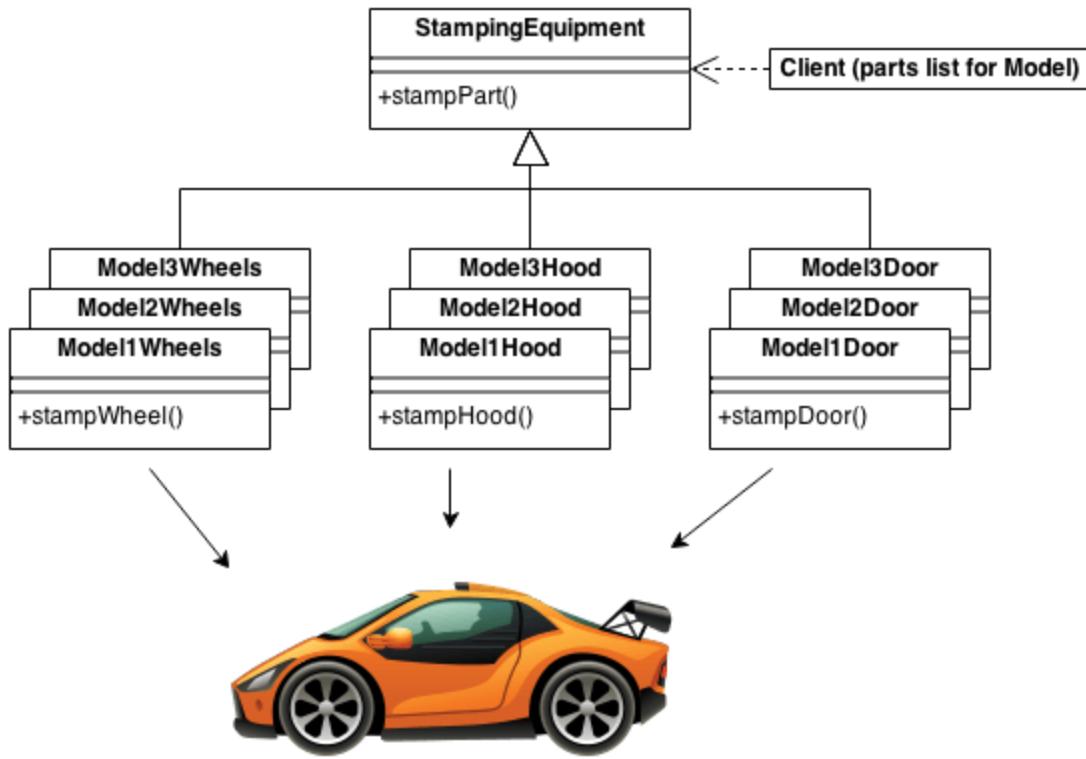
الگوی کارخانه انتزاعی^{۳۰}، در الگوهای نرمافزاری روشی برای جمع‌بندی گروهی از کارخانه‌های مجازی است که ساختار مشابهی دارند ولی از کلاس‌های مختلفی تشکیل شده‌اند.

در حالت عادی، برنامه زیرشاخه یک ساختار یکپارچه از کارخانه انتزاعی را می‌سازد و سپس از واسط کاربری می‌خواهد که شیء‌های مختلفی در آن (دارای شباهت در تعدادی از ویژگی‌ها) بسازد. برنامه زیرشاخه نمی‌داند (یا اهمیت نمی‌دهد) که چه شیء‌ای را از کتابخانه گرفته است. چون تنها از شیء ساخته‌شده استفاده می‌کند. این الگو جزیيات اجرا و استفاده از گروهی از اشیاء را، از نحوه پیاده‌سازی آن‌ها جدا می‌کند. چراکه ساخت اشیاء در کارخانه صورت می‌گیرد. ماهیت کارخانه انتزاعی را می‌توان این‌طور تعریف کرد که رابطی برای ساخت خانواده‌ای از اشیاء مرتبط یا وابسته بدون نیاز به مشخص کردن نوع کلاس آن‌ها.

مثال در دنیای واقعی

هدف از کارخانه انتزاعی، ایجاد واسط بین خانواده‌های اشیاء مرتبط بدون تعیین کلاس‌های غیرقابل تغییر است. این الگو در تجهیزات ورق فلزی مورداستفاده در ساخت خودروهای ژاپنی یافت می‌شود. به این صورت که دستگاه‌هایی که کار پرس ورق را انجام می‌دهند بهصورت یک کارخانه انتزاعی کار می‌کنند که باعث ایجاد قطعات بدنخودرو می‌شود. از همین ماشین‌آلات برای ساخت درب‌های سمت راست، درب‌های چپ، گلگیرهای جلوی راست، گلگیرهای جلو سمت چپ و غیره برای مدل‌های مختلف اتومبیل استفاده می‌شود. از طریق استفاده از غلتک برای تغییر شکل و سایز پرس ورق، کلاس‌های تولیدشده توسط ماشین‌آلات می‌توانند ظرف سه دقیقه تغییر کنند.





روش پیاده سازی الگوی طراحی *Abstract Factory*

در این بخش به بررسی مراحل پیاده سازی الگوی طراحی کارخانه انتزاعی می پردازیم. مراحل پیاده سازی این الگو عبارت اند از :

1. ابتدا باید فهرستی از انواع اشیای موردنظر در نرم افزار تهیه شود.
2. باید برای هر یک از انواع آنها یک Interface تعریف شود. سپس تمام کلاس‌های این اشیا باید از این Interface پیروی کنند.
3. در این گام Interface کارخانه انتزاعی باید ایجاد شود. در این Interface مجموعه‌ای از روش‌های ایجاد اشیا انتزاعی تعریف می‌شود.
4. باید برای هر یک از اشیا که سبک مشابهی دارند کلاس‌های به عنوان کارخانه سازنده آنها ایجاد شود. این کارخانه‌ها که اشیا مرتبط را تولید می‌کنند باید از Interface کارخانه انتزاعی پیروی کنند.
5. کد مربوط به راه اندازی کارخانه را در نرم افزار باید نوشت. این کد باید یک کلاس کارخانه مرتبط را بسته باشد، بسته به پیکربندی برنامه یا محیط فعلی. این کارخانه را به کلیه کلاس‌هایی که محصولات را طراحی می‌کنند انتقال دهید.



6. هر جایی در کدهای نرم افزار که اشیا به صورت مستقیم ایجاد شده اند، باید شناسایی شود. سپس آن‌ها با توابع ساخت مناسب با توجه به کارخانه مربوطه جایگزین شوند.
حالا به سراغ پیاده سازی این الگو میرویم. به مثال زیر دقت کنید.



```

"""
Provide an interface for creating families of related or dependent
objects without specifying their concrete classes.
"""
import abc

class AbstractFactory(metaclass=abc.ABCMeta):
    """
    Declare an interface for operations that create abstract product
    objects.
    """
    @abc.abstractmethod
    def create_product_a(self):
        pass

    @abc.abstractmethod
    def create_product_b(self):
        pass

class ConcreteFactory1(AbstractFactory):
    """
    Implement the operations to create concrete product objects.
    """
    def create_product_a(self):
        return ConcreteProductA1()

    def create_product_b(self):
        return ConcreteProductB1()

class ConcreteFactory2(AbstractFactory):
    """
    Implement the operations to create concrete product objects.
    """
    def create_product_a(self):
        return ConcreteProductA2()

    def create_product_b(self):
        return ConcreteProductB2()

class AbstractProductA(metaclass=abc.ABCMeta):
    """
    Declare an interface for a type of product object.
    """
    @abc.abstractmethod
    def interface_a(self):
        pass

class ConcreteProductA1(AbstractProductA):
    """
    Define a product object to be created by the corresponding concrete
    factory.
    Implement the AbstractProduct interface.
    """
    def interface_a(self):
        pass

```



```

class ConcreteProductA2(AbstractProductA):
    """
    Define a product object to be created by the corresponding concrete
    factory.
    Implement the AbstractProduct interface.
    """
    def interface_a(self):
        pass

class AbstractProductB(metaclass=abc.ABCMeta):
    """
    Declare an interface for a type of product object.
    """
    @abc.abstractmethod
    def interface_b(self):
        pass

class ConcreteProductB1(AbstractProductB):
    """
    Define a product object to be created by the corresponding concrete
    factory.
    Implement the AbstractProduct interface.
    """
    def interface_b(self):
        pass

class ConcreteProductB2(AbstractProductB):
    """
    Define a product object to be created by the corresponding concrete
    factory.
    Implement the AbstractProduct interface.
    """
    def interface_b(self):
        pass

def main():
    for factory in (ConcreteFactory1(), ConcreteFactory2()):
        product_a = factory.create_product_a()
        product_b = factory.create_product_b()
        product_a.interface_a()
        product_b.interface_b()

if __name__ == "__main__":
    main()

```



Builder

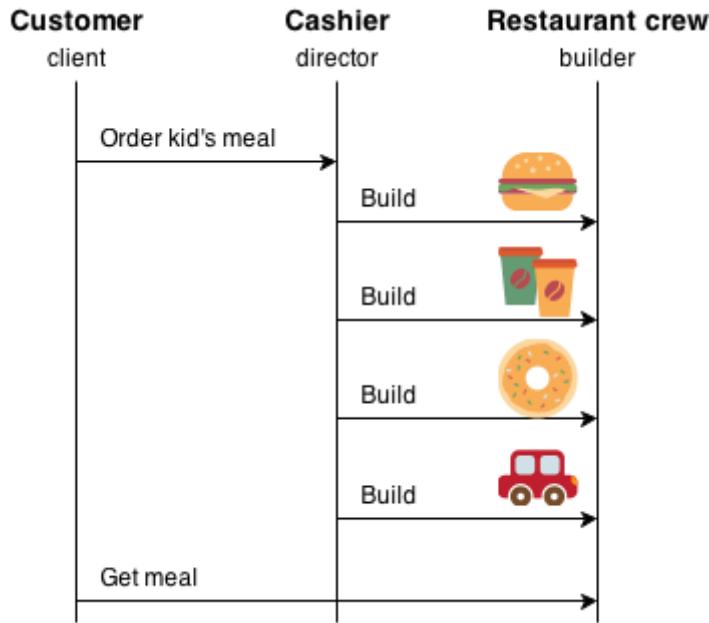
الگوی سازنده^{۳۱} یک الگوی مهندسی نرم‌افزار به منظور ایجاد اشیا است. این الگو برخلاف الگوی کارخانه انتزاعی (abstract factory pattern) و الگوی روش کارخانه (factory method pattern) که به منظور رعایت پدیده چندریختی (polymorphism) ایجاد شده‌اند به منظور حل وجهه دیگری از مشکل ساخت و تنظیم اشیا در برنامه‌نویسی معرفی شده است. مشکل بدین شرح است که گاهی نیاز است هنگام ساخت یک شیء تعداد زیادی پارامتر را به سازنده (constructor) آن تحويل دهیم و این کار خوانایی برنامه را کم می‌کند. به منظور حل این مشکل از الگوی سازنده استفاده می‌کنیم. در این الگو به جای طراحی تعدادی سازنده (constructor) با تعداد زیادی پارامتر، از یک شیء دیگر استفاده می‌کنیم که کار پارامتر دهی را به صورت مرحله‌به‌مرحله و خواناتر انجام می‌دهد و درنهایت از نوع شیء موردنظر یک نمونه با تنظیمات خواسته شده به ما تحويل می‌دهد.

معمولًاً یک طراح در سیر طراحی ابتدا با الگوی روش کارخانه (factory method) شروع می‌کند سپس به الگوی کارخانه انتزاعی (abstract factory) یا الگوی نمونه اولیه (prototype) یا سازنده (builder) متousel می‌شود. معمولًاً هنگامی به الگوی سازنده متousel می‌شویم که در فرایند طراحی به انعطاف‌پذیری بیشتر نیاز پیدا می‌کنیم. نیت استفاده از این الگو این است که فرایند ساخت (construction) یک شیء پیچیده را از کد آن شیء (object representation) جدا کنیم.

مثال در دنیای واقعی

الگوی Builder ساخت یک شیء پیچیده را از نمایش آن جدا می‌کند تا همین روند ساخت‌وساز مشابه بتواند بازنمایی‌های مختلفی ایجاد کند. این الگوی توسط رستوران‌های فست‌فود برای ساخت وعده‌های غذایی کودکان استفاده می‌شود. وعده‌های غذایی کودکان به‌طور معمول از یک کالای اصلی، یک ماده جانبی، یک نوشیدنی و یک اسباب‌بازی تشکیل می‌شود (به عنوان مثال یک همبرگر، سیب‌زمینی سرخ‌کرده، کیک و دایناسور اسباب‌بازی). توجه داشته باشید که در محتوای وعده‌های غذایی کودکان می‌توان تنوع ایجاد کرد اما روند ساخت‌وساز همان است. چه مشتری همبرگر، چیزبرگر یا مرغ سوخاری سفارش دهد، روند مشابه است. کارمند در پیشخوان خدمه را به سمت جمع‌آوری یک کالای اصلی، آیتم جانبی و اسباب‌بازی هدایت می‌کند. این موارد سپس در یک کیسه قرار می‌گیرند. نوشیدنی را در یک فنجان قرار داده و خارج از کیسه می‌ماند. همین روند در رستوران‌های رقیب نیز قابل استفاده است.





روش پیاده سازی الگوی طراحی *Builder*

برای پیاده سازی الگوی طراحی سازنده نیازمند طی کردن چندین گام خواهیم بود. پس از آشنایی با مراحل پیاده سازی این الگوی طراحی، به بررسی مثالی از الگوی طراحی سازنده با استفاده زبان برنامه نویسی PHP خواهیم پرداخت. در این مثال مراحل ساخت یک خانه با استفاده از الگوی طراحی سازنده را بررسی خواهیم کرد. مراحل پیاده سازی این الگو عبارت اند از :

1. ابتدا باید از این موضوع اطمینان حاصل شود که به طور واضح و دقیق تمام اقدامات و مراحل لازم برای ساخت تمامی نمونه های شی موردنظر (محصول) تعریف شده باشند. در غیر این صورت پیاده سازی این الگوی طراحی امکان پذیر نخواهد بود.
2. سپس این مراحل باید در یک Interface *Builder* تعریف شوند.
3. در سومین گام باید یک کلاس *Builder* برای هر یک از نمونه های شی موردنظر ایجاد شود. این کلاس ها باید Interface *Builder* ساخته شده در گام قبلی را Implement کنند. به عبارتی دیگر همه آن ها باید از مراحل ساخت و ساز تعریف شده پیروی کنند.
4. همچنین می توان از کلاس دیگری به نام *Director* برای مدیریت و پیاده سازی مراحل قبل استفاده کرد. این کلاس می تواند از راه های مختلفی به ساخت اشیا با استفاده از همان *Builder* ها پردازد.
5. بنابراین ابتدا باید کلاس های *Director* و *Builder* ایجاد شوند. سپس باید قبل از اینکه عملیات ساخت شی آغاز شود، کلاس *Builder* به کلاس *Director* مرتبط شود. این ارتباط یک بار به وسیله ارسال



پارامترهای کلاس Director در Constructor کلاس Director ایجاد می‌شود. با این کار کلاس Director در ساخت و سازهای بعدی از Builder استفاده می‌کند. البته برای این منظور یک راه حل جایگزین نیز وجود دارد. در راه حل دوم می‌توان Builderها را به صورت مستقیم در تابعی (Method) که وظیفه ساخت در کلاس Director را دارد، وارد کنید.

6. در نهایت اگر تمام اشیا از همان Interface پیروی کنند می‌توان نتایج ساخت و ساز را به صورت مستقیم از کلاس Director دریافت کرد. در غیر این صورت باید نتایج از همان کلاس Builder دریافت شوند. حالا به سراغ پیاده سازی این الگو می‌رویم. به مثال زیر دقت کنید.



```

"""
Separate the construction of a complex object from its representation so
that the same construction process can create different representations.
"""

import abc

class Director:
    """
    Construct an object using the Builder interface.
    """

    def __init__(self):
        self._builder = None

    def construct(self, builder):
        self._builder = builder
        self._builder._build_part_a()
        self._builder._build_part_b()
        self._builder._build_part_c()

class Builder(metaclass=abc.ABCMeta):
    """
    Specify an abstract interface for creating parts of a Product
    object.
    """

    def __init__(self):
        self.product = Product()

    @abc.abstractmethod
    def _build_part_a(self):
        pass

    @abc.abstractmethod
    def _build_part_b(self):
        pass

    @abc.abstractmethod
    def _build_part_c(self):
        pass

class ConcreteBuilder(Builder):
    """
    Construct and assemble parts of the product by implementing the
    Builder interface.
    Define and keep track of the representation it creates.
    Provide an interface for retrieving the product.
    """

    def _build_part_a(self):
        pass

    def _build_part_b(self):
        pass

    def _build_part_c(self):
        pass

```



```

class Product:
    """
        Represent the complex object under construction.
    """
    pass

def main():
    concrete_builder = ConcreteBuilder()
    director = Director()
    director.construct(concrete_builder)
    product = concrete_builder.product

if __name__ == "__main__":
    main()

```

Factory

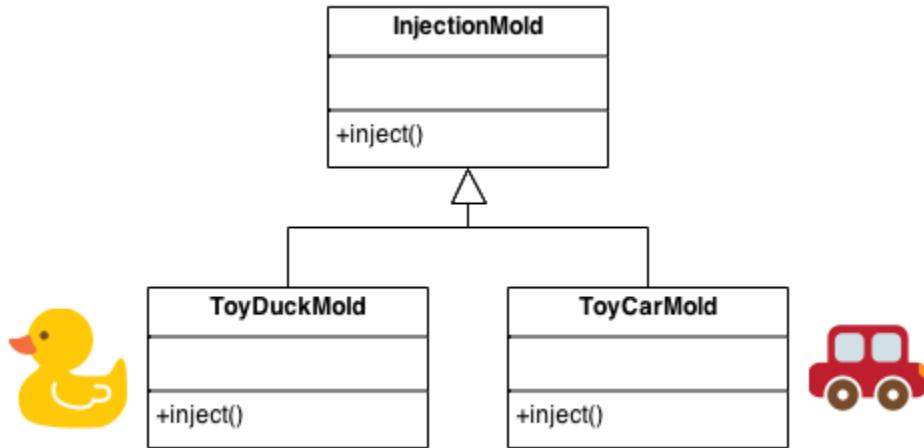
الگوی کارخانه^{۳۲} شئ یکی از الگوهای طراحی برنامه‌نویسی شئ‌گرا برای پیاده‌سازی مفهوم کارخانه‌ها است؛ مانند سایر الگوهای سازندگی، کارخانه شئ با مشکل ایجاد اشیاء (محصولات) بدون مشخص کردن کلاس اشیاء خاصی است که قرار است ساخته شوند. اساس الگوی متده کارخانه «تعریف یک رابط برای ایجاد اشیاء که اجازه می‌دهد اشیایی که آن رابط را پیاده‌سازی می‌کنند در رابطه با یکنکه کدام کلاس باید ایجاد شود تصمیم بگیرند. متده الگوی کارخانه اجازه می‌دهد که یک کلاس تصمیم در رابطه با ایجاد اشیاء را به زیرکلاس‌ها واگذار نماید.» است.

اساساً کاربرد الگوی کارخانه برای شرایطی است که چندین کلاس با ریشه مشترک داریم (یعنی چندین کلاس یک کلاس فوقانی را پیاده‌سازی می‌کنند) و غالب استفاده نیز با شئ سازی (نمونه‌سازی) از کلاس فوقانی صورت می‌گیرد.

مثال در دنیای واقعی

الگوی طراحی کارخانه رابطی را برای ایجاد اشیاء تعریف می‌کند، اما این اجازه را نیز می‌دهد تا زیرکلاس‌ها تصمیم بگیرند که کدام کلاس‌ها بلافصله اجرا شوند. پرس‌های قالب تزریقی مثال خوبی برای این الگو هستند. تولیدکنندگان اسباب‌بازی‌های پلاستیکی پودر قالب‌گیری پلاستیک را پردازش می‌کنند و پلاستیک را به قالب‌های موردنظر تزریق می‌کنند. کلاس اسباب‌بازی (ماشین، شکل رفتار و غیره) توسط قالب مشخص می‌شود.



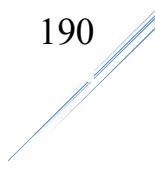


مراحل پیاده سازی الگوی طراحی Factory

در این بخش به بررسی مراحل پیاده سازی الگوی طراحی Factory می پردازیم، مراحل پیاده سازی این الگو عبارت اند از :

1. ابتدا باید یک Interface تعریف کنیم که در آن توابعی تعریف شود که در همه اشیا مشترک و کاربردی است. سپس تمام اشیا باید از یک Interface پیروی کنند.
 2. باید یک تابع به عنوان کارخانه سازنده (Factory) آن شی در کلاسشن اضافه شود. نوع متغیری که این تابع Return می کند باید با Interface که در بخش قبل ساخته شد، مطابقت داشته باشد.
 3. در کدهای نوشته شده هرجایی که اشیا توسط روش سنتی (به وسیله کیورد new) ایجاد شده اند باید پیدا شوند و برای ساخت آنها از تابع کارخانه سازنده آنها استفاده شود.
 4. باید مجموعه ای از کلاس های فرزند سازنده (creator) برای انواع اشیایی که کارخانه می سازد، تعریف شوند. سپس در این کلاس ها باید تابع کارخانه بازنویسی (Override) شود تا اصولی که در این تابع تعریف شده اند، به شکل موردنیاز تغییر داده شوند. مثلا اگر در نحوه ساخت یک خودروی سواری و خودروی باربری تفاوتی وجود دارد، در کلاس های مربوط به خود این موضوع در نظر گرفته شود.
 5. اگر انواع مختلف و زیادی از اشیا وجود دارند که از نظر شما ساختن کلاس های فرزند برای همه آنها منطقی به نظر نمی رسند و دارای ویژگی های مشترکی هستند، می توانید آن ویژگی را در کلاس والد وارد کنید و از تعریف مکرر آن در کلاس های فرزند خودداری کنید.
 6. اگر مشاهده کردید میان اشیای ساخته شده تابع کارخانه هیچ ویژگی مشترکی وجود ندارد و تابع کارخانه خالی مانده است، می توانید آن را به صورت abstract تعریف کنید. همچنین اگر خلاف این امر صادق بود و ویژگی در این تابع باقی ماند، آن ویژگی را می توان، یک ویژگی پیشفرض برای اشیا در نظر گرفت.
- حالا به سراغ پیاده سازی این الگو میرویم. به مثال زیر دقت کنید.





```

"""
Define an interface for creating an object, but let subclasses decide
which class to instantiate. Factory Method lets a class defer
instantiation to subclasses.
"""
import abc

class Creator(metaclass=abc.ABCMeta):
    """
    Declare the factory method, which returns an object of type Product.
    Creator may also define a default implementation of the factory
    method that returns a default ConcreteProduct object.
    Call the factory method to create a Product object.
    """
    def __init__(self):
        self.product = self._factory_method()

    @abc.abstractmethod
    def _factory_method(self):
        pass

    def some_operation(self):
        self.product.interface()

class ConcreteCreator1(Creator):
    """
    Override the factory method to return an instance of a ConcreteProduct1.
    """
    def _factory_method(self):
        return ConcreteProduct1()

class ConcreteCreator2(Creator):
    """
    Override the factory method to return an instance of a
    ConcreteProduct2.
    """
    def _factory_method(self):
        return ConcreteProduct2()

class Product(metaclass=abc.ABCMeta):
    """
    Define the interface of objects the factory method creates.
    """
    @abc.abstractmethod
    def interface(self):
        pass

class ConcreteProduct1(Product):
    """
    Implement the Product interface.
    """
    def interface(self):
        pass

```



```

class ConcreteProduct2(Product):
    """
    Implement the Product interface.
    """
    def interface(self):
        pass

def main():
    concrete_creator = ConcreteCreator1()
    concrete_creator.product.interface()
    concrete_creator.some_operation()

if __name__ == "__main__":
    main()

```

Prototype

الگوی نمونه اولیه یکی از الگوهای طراحی در توسعه نرمافزار است. وقتی نوع اشیاء ساخته شونده با یک نمونه اولیه مشخص شود، از این الگو استفاده شده است؛ که درواقع یک مورد مشابه از خودساخته است یا به اصطلاح خودسازی کرده است. موارد استفاده از این الگو:

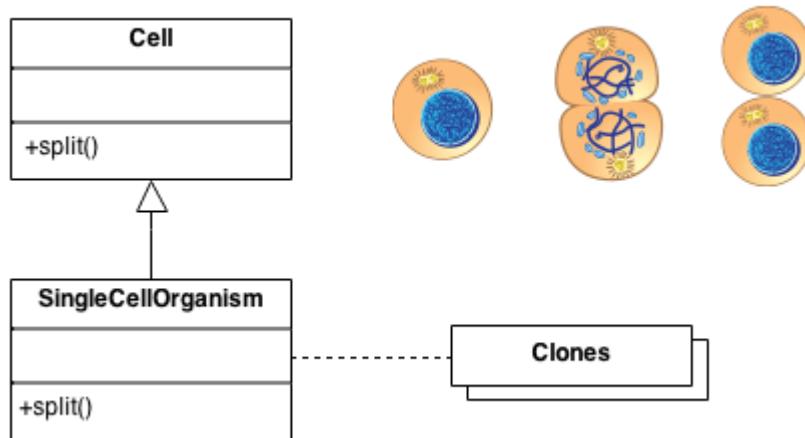
▪ جلوگیری از وجود کلاس فرزند از یک کلاس سازنده اشیاء در نرمافزار مشتری همان‌طوری که الگوی کارخانه انتزاعی عمل می‌کند.

▪ جلوگیری از هزینه وراثتی ساخت شیء جدید درروش معمول آن (به عنوان مثال با استفاده از دستور 'new' چراکه در برنامه‌ها این کار از نظر پردازشی پرهزینه خواهد بود.)

مثال در دنیای واقعی

الگوی نمونه اولیه نوع اشیاء را با استفاده از نمونه اولیه برای ایجاد مشخص می‌کند. نمونه‌های اولیه محصولات جدید اغلب قبل از تولید کامل ساخته می‌شوند، اما در این مثال نمونه اولیه منفعل است و در کپی کردن خود شرکت نمی‌کند. تقسیم میتوزیک یک سلول - که منجر به دو سلول یکسان می‌شود - نمونه‌ای از نمونه اولیه است که در کپی کردن خود نقش فعالی دارد و بنابراین، الگوی نمونه اولیه را نشان می‌دهد. هنگامی که یک سلول شکافته می‌شود، دو سلول از ژنتیپ یکسان حاصل می‌شوند. به عبارت دیگر، سلول خود کلون می‌شود.





پیاده سازی الگوی طراحی Prototype

در این بخش به بررسی مراحل پیاده سازی الگوی طراحی نمونه اولیه می پردازیم، مراحل پیاده سازی این الگو عبارت‌اند از :

1. ابتدا باید یک Interface تعریف شود. سپس در آن یک تابع تحت عنوان Clone ایجاد گردد. سپس کلاس‌های موردنظر باید از این Interface پیروی کنند و تابع Clone را پیاده سازی کنند. این گام برای قانونمند ساختن نرم افزار استفاده می‌شود و در صورت تشخیص می‌توان از آن صرف نظر کرد.
 2. سپس در کلاس نمونه اولیه باید از یک متده Constructor جدید به نام Clone تعریف شود. این وظیفه دارد مقادیر پارامترهای تعریف شده در کلاس را از شی اصلی دریافت کند و به نمونه جدید ساخته شده، منتقل سازد.
- حالا به سراغ پیاده سازی این الگو می‌رویم. به مثال زیر دقت کنید.



```

"""
Specify the kinds of objects to create using a prototypical instance,
and create new objects by copying this prototype.
"""

import copy

class Prototype:
    """
    Example class to be copied.
    """

    pass


def main():
    prototype = Prototype()
    prototype_copy = copy.deepcopy(prototype)

    if __name__ == "__main__":
        main()

```

الگوهای ساختاری

الگوهای طراحی ساختاری یا Structural، مجموعه‌ای از راه حل‌هایی هستند که برای حل مشکلات توسعه پذیری ساختار نرم افزارها، به کمک برنامه نویسان می‌آیند. این الگوهای طراحی برای مدیریت ارتباط میان کلاس‌ها و شیوه‌ها با یکدیگر استفاده می‌شوند. این الگوهای طراحی عبارت‌اند از:

Adapter

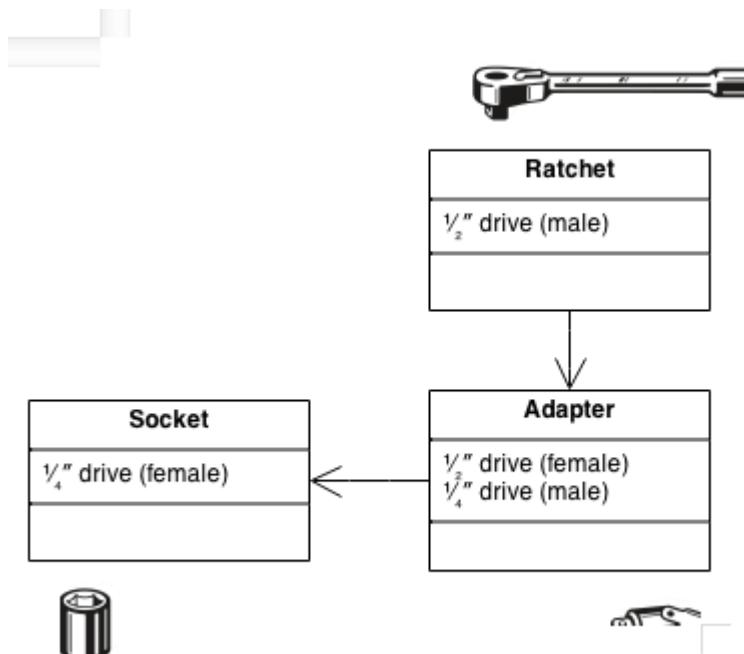
در مهندسی نرم افزار، الگوی آداتور^{۳۳} (الگوی وفق دهنده) یک الگوی طراحی نرم افزار است که به رابط یک کلاس اجازه می‌دهد تا توسط رابط دیگری مورد استفاده قرار گیرد. عموماً با این هدف مورد استفاده قرار می‌گیرد که بدون تغییر در کد منبع، بتوان استفاده از کلاس‌های فعلی را مقدور ساخت. یک آداتور به دو رابط ناسازگار اجازه می‌دهد تا بتوانند باهم کار کنند. این یک تعریف کلی از مفهوم آداتور است. ممکن است رابط‌ها ناسازگار باشند ولی قابلیت درونی آن‌ها باید سازگار با نیاز باشد. الگوی طراحی آداتور از طریق تبدیل رابط یک کلاس به رابط مورد انتظار توسط کلاینت، به کلاس‌های ناسازگار اجازه می‌دهد تا بتوانند از قابلیت‌های هم‌دیگر استفاده کنند.

Adapter^{۳۳}



مثال در دنیای واقعی

الگوی آدپتور اجازه می‌دهد تا با تبدیل واسط یک کلاس به یک رابط مورد انتظار مشتری، کلاس‌های ناسازگار با یکدیگر کار کنند. آچار سوکت نمونه‌ای از آدپتور را ارائه می‌دهد. یک سوکت به یک چرخ دستی وصل می‌شود به شرط آنکه اندازه درایو یکسان باشد. اندازه معمولی درایو $2/1$ و $4/1$ است. بدیهی است، یک گیربکس درایو $4/1$ اینچی در سوکت درایو $4/1$ اینچ جای نمی‌گیرد مگر اینکه از آدپتور استفاده شود. یک آدپتور $2/1$ تا $1/4$ اینچی دارای یک اتصال زن $1/2$ اینچی برای قرار گرفتن در قسمت درایو $1/2$ اینچی و یک اتصال نری $1/4$ اینچی برای قرار گرفتن در سوکت درایو $1/4$ است.



حالا به سراغ پیاده سازی این الگو میرویم. به مثال زیر دقت کنید.



```

"""
Convert the interface of a class into another interface clients expect.
Adapter lets classes work together that couldn't otherwise because of
incompatible interfaces.
"""

import abc

class Target(metaclass=abc.ABCMeta):
    """
    Define the domain-specific interface that Client uses.
    """
    def __init__(self):
        self._adaptee = Adaptee()

    @abc.abstractmethod
    def request(self):
        pass

class Adapter(Target):
    """
    Adapt the interface of Adaptee to the Target interface.
    """
    def request(self):
        self._adaptee.specific_request()

class Adaptee:
    """
    Define an existing interface that needs adapting.
    """
    def specific_request(self):
        pass

def main():
    adapter = Adapter()
    adapter.request()

if __name__ == "__main__":
    main()

```

Bridge

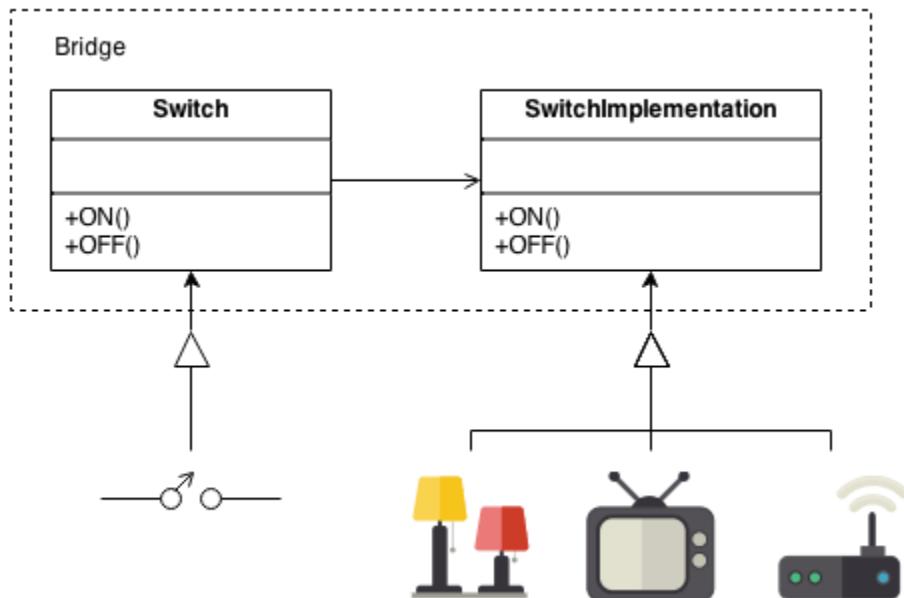
الگوی پل^{۳۴}، یک الگوی طراحی در مهندسی نرم افزار است که به معنای جداسازی یک انتزاع از اجرای آن به طوری که این دو بتوانند به صورت مستقل تغییرپذیر باشند. است. این الگو توسط باند چهار نفره (GoF) معرفی شده است. الگوی پل از گروههای قالب دار و گروههای تجمعی و بعضی از ثباتی برای جداسازی مسئولیت‌ها در طبقات مختلف استفاده می‌کند.

Bridge³⁴



مثال در دنیای واقعی

الگوی پل یک انتزاع از اجرای آن را جدا می‌کند، به طوری که این دو می‌توانند به‌طور مستقل متفاوت باشند. سیستم مرکزی کنترل لوازم برقی یک خانه که دارای چراغ‌های کنترل‌کننده، پنکه سقفی و غیره است، نمونه‌ای از الگوی طراحی پل است. هدف از سوئیچ روشن یا خاموش کردن دستگاه است. سوئیچ واقعی را می‌توان به عنوان یک زنجیره چندحالته، سوئیچ دوحالته ساده یا انواع سوئیچ‌های کم‌نور استفاده کرد.



حالا به سراغ پیاده سازی این الگو میرویم. به مثال زیر دقت کنید.



```

"""
Decouple an abstraction from its implementation so that the two can vary
independently.
"""
import abc

class Abstraction:
    """
    Define the abstraction's interface.
    Maintain a reference to an object of type Implementor.
    """
    def __init__(self, imp):
        self._imp = imp

    def operation(self):
        self._imp.operation_imp()

class Implementor(metaclass=abc.ABCMeta):
    """
    Define the interface for implementation classes. This interface
    doesn't have to correspond exactly to Abstraction's interface; in
    fact the two interfaces can be quite different. Typically the
    Implementor interface provides only primitive operations, and
    Abstraction defines higher-level operations based on these
    primitives.
    """
    @abc.abstractmethod
    def operation_imp(self):
        pass

class ConcreteImplementorA(Implementor):
    """
    Implement the Implementor interface and define its concrete
    implementation.
    """
    def operation_imp(self):
        pass

class ConcreteImplementorB(Implementor):
    """
    Implement the Implementor interface and define its concrete
    implementation.
    """
    def operation_imp(self):
        pass

def main():
    concrete_implementor_a = ConcreteImplementorA()
    abstraction = Abstraction(concrete_implementor_a)
    abstraction.operation()

if __name__ == "__main__":
    main()

```



Decorator

در برنامه‌نویسی شئ گرا، الگوی آذین گر یا دکوراتر^{۳۵} یک الگوی طراحی است که امکان افزودن رفتار (behavior) به یک شئ را به‌طور پویا^{۳۶} یا ایستا^{۳۷} فراهم می‌سازد بی‌آنکه رفتار اشیاء دیگر از همان کلاس (که شئ موردهبحث از آن ساختهشده) دستخوش تغییر شوند. الگوی طراحی آذین گر معمولاً برای پایندی بهقاعدۀ تکوظیفه‌ای مورداستفاده قرار می‌گیرد چراکه این الگوی طراحی، امکان تقسیم عملکردها (functionality) بین کلاس‌های مختلف که هرکدام دغدغه‌های (concern) خاص را پوشش می‌دهند، فراهم می‌سازد.

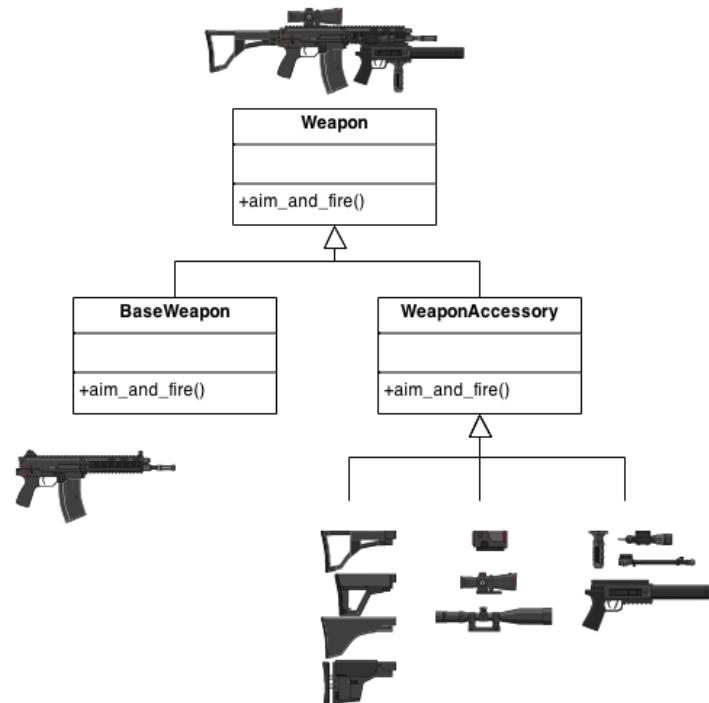
مثال در دنیای واقعی

دکوراتور به‌طور پویا مسئولیت‌های اضافی را به یک شئ متصل می‌کند. تزئیناتی که به درختان کاج یا صنوبر اضافه‌شده نمونه‌هایی از دکوراتورها هستند. چراغ‌ها، گلدان‌ها، نی‌های شیرینی، تزئینات شیشه‌ای و غیره را می‌توان به درخت اضافه کرد تا جلوه‌ای تزئینی به آن بخشیده باشد. تزئینات خود درخت را تغییر نمی‌دهند و بدون توجه به تزئینات خاص مورداستفاده، به عنوان یک درخت کریسمس قابل تشخیص است. به عنوان نمونه‌ای از قابلیت‌های اضافی، اضافه کردن چراغ‌ها به شخص اجازه می‌دهد تا یک درخت کریسمس را روشن کند.

مثال دیگر: تفنگ حمله به‌خودی خود یک اسلحه کشنده است؛ اما می‌توانید از "تزئینات" خاصی استفاده کنید تا آن را دقیق‌تر، ساکت و ویرانگر کنید.

Decorator^{۳۵}
Dynamic^{۳۶}
Static^{۳۷}





حالا به سراغ پیاده سازی این الگو میرویم. به مثال زیر دقت کنید.



```

"""
Attach additional responsibilities to an object dynamically. Decorators
provide a flexible alternative to subclassing for extending
functionality.
"""
import abc

class Component(metaclass=abc.ABCMeta):
    """
    Define the interface for objects that can have responsibilities
    added to them dynamically.
    """
    @abc.abstractmethod
    def operation(self):
        pass

class Decorator(Component, metaclass=abc.ABCMeta):
    """
    Maintain a reference to a Component object and define an interface
    that conforms to Component's interface.
    """
    def __init__(self, component):
        self._component = component

    @abc.abstractmethod
    def operation(self):
        pass

class ConcreteDecoratorA(Decorator):
    """
    Add responsibilities to the component.
    """
    def operation(self):
        # ...
        self._component.operation()
        # ...

class ConcreteDecoratorB(Decorator):
    """
    Add responsibilities to the component.
    """
    def operation(self):
        # ...
        self._component.operation()
        # ...

class ConcreteComponent(Component):
    """
    Define an object to which additional responsibilities can be
    attached.
    """
    def operation(self):
        pass

```



```

def main():
    concrete_component = ConcreteComponent()
    concrete_decorator_a = ConcreteDecoratorA(concrete_component)
    concrete_decorator_b = ConcreteDecoratorB(concrete_decorator_a)
    concrete_decorator_b.operation()

if __name__ == "__main__":
    main()

```

Facade

الگوی نما^{۳۸}، یک الگوی ساختاری در الگوهای طراحی نرمافزار است که معمولاً در برنامه‌نویسی شیء‌گرا از آن استفاده می‌شود. نام آن برگرفته از شباهت آن به مشابه آن در معماری ساختمان نما (ساختمان) است.

نما، شیء‌ای است که یک رابط راحت برای دسترسی به قسمت بزرگ و پیچیده‌ای از کد است. مثل کتابخانه کلاس‌ها. فساد می‌تواند:

- + استفاده از یک کتابخانه نرمافزاری را آسان‌تر کند، بفهمد و آن را تست کند. چراکه توابع راحتی برای عملیات عادی دارد.

- + به دلایل مشابهی، خواندن از کتابخانه را راحت‌تر و امکان‌پذیر‌تر می‌کند.
- + کاهش وابستگی به کدهای خارجی مهم‌ترین وظیفه یک کتابخانه داخلی است، چراکه بیشتر قسمت‌های کد از نما استفاده می‌کنند، باعث تغییرپذیری بیشتر در طراحی سیستم می‌شود.

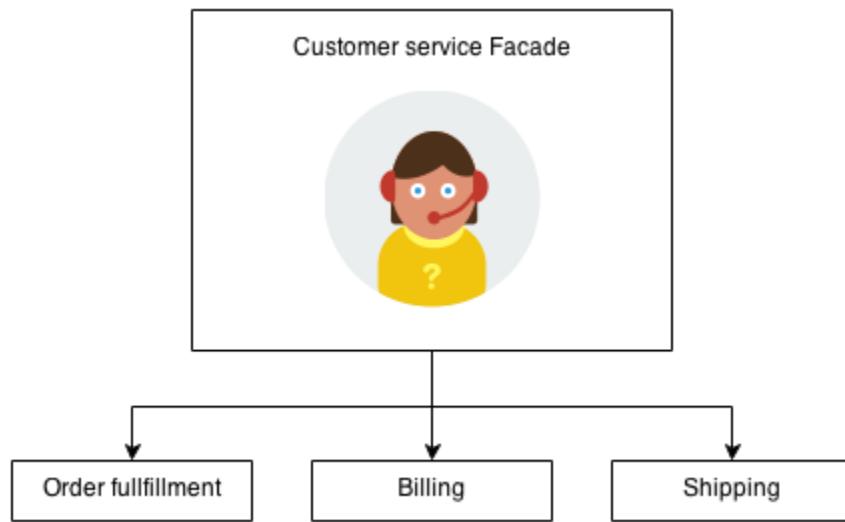
- + بسته مجموعه‌ای از رابطه‌ای برنامه کاربری با طراحی ضعیف، تنها با یک رابط برنامه کاربری (API) که از طراحی خوبی برخوردار است.

معمولًاً وقتی از الگوی طراحی نما استفاده می‌شود که سامانه از پیچیدگی زیادی برخوردار است یا فهمیدن آن دشوار است، به خاطر آن‌که تعداد زیادی از کلاس‌های دارای وابستگی داخلی یا کلاس‌هایی که کد آن‌ها در دسترس نباشد وجود داشته باشند. این الگو پیچیدگی یک سامانه بزرگ را مخفی کرده و یک رابط ساده برای مشتری فراهم می‌کند. معمولًاً دارای یک کلاس پوشه بندی ساده است که مجموعه‌ای از عضوهای موردنیاز مشتری در آن وجود دارند. این اعضا به جای مشتری نما، به سامانه دسترسی دارند و نحوه پیاده‌سازی را مخفی می‌کنند.



مثال در دنیای واقعی

یک رابط سطح بالاتر و یکپارچه را برای یک سیستم فرعی تعریف می‌کند که استفاده از آن را آسان‌تر می‌کند. مصرف‌کنندگان هنگام خرید تلفنی از فروشگاه با الگوی طراحی نما روبرو می‌شوند. مصرف‌کننده با یک شماره تماس می‌گیرد و با یک نماینده خدمات مشتری صحبت می‌کند. نماینده خدمات مشتری به عنوان یک نمایه عمل می‌کند و رابط کاربری را برای بخش تحقق سفارش، بخش صدور صورتحساب و بخش حمل و نقل فراهم می‌کند.



حالا به سراغ پیاده سازی این الگو میرویم. به مثال زیر دقت کنید.



```

"""
Provide a unified interface to a set of interfaces in a subsystem.
Facade defines a higher-level interface that makes the subsystem easier
to use.
"""

class Facade:
    """
    Know which subsystem classes are responsible for a request.
    Delegate client requests to appropriate subsystem objects.
    """

    def __init__(self):
        self._subsystem_1 = Subsystem1()
        self._subsystem_2 = Subsystem2()

    def operation(self):
        self._subsystem_1.operation1()
        self._subsystem_1.operation2()
        self._subsystem_2.operation1()
        self._subsystem_2.operation2()

class Subsystem1:
    """
    Implement subsystem functionality.
    Handle work assigned by the Facade object.
    Have no knowledge of the facade; that is, they keep no references to
    it.
    """

    def operation1(self):
        pass

    def operation2(self):
        pass

class Subsystem2:
    """
    Implement subsystem functionality.
    Handle work assigned by the Facade object.
    Have no knowledge of the facade; that is, they keep no references to
    it.
    """

    def operation1(self):
        pass

    def operation2(self):
        pass

def main():
    facade = Facade()
    facade.operation()

if __name__ == "__main__":
    main()

```

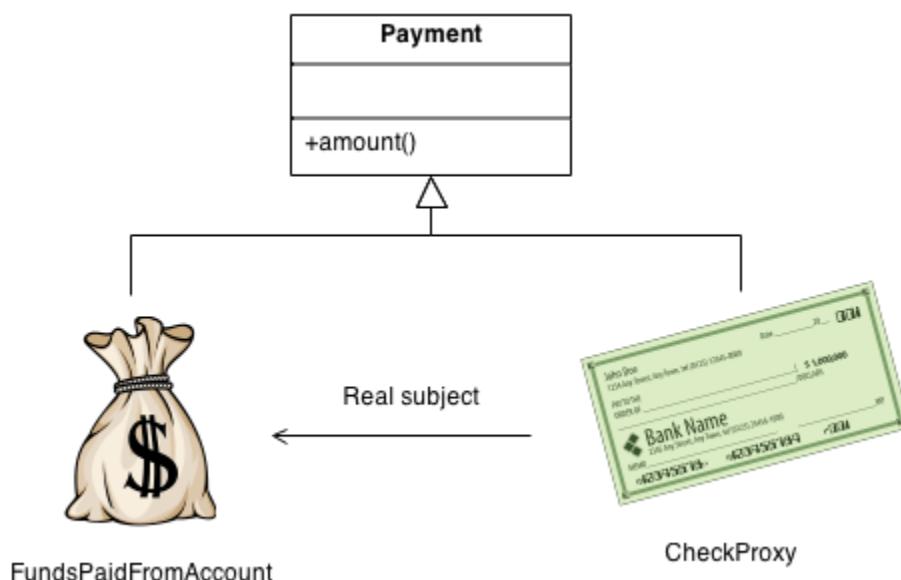


Proxy

الگوی وکالت^{۳۹} در برنامه‌نویسی، از جمله الگوهای طراحی نرم‌افزار است. وکیل، در کلی‌ترین حالت، کلاس واسطه‌ای برای انجام کار دیگری است. پروکسی می‌تواند رابط هر چیزی باشد: یک ارتباط تحت شبکه، شیء بزرگی در داخل حافظه، یک فایل، یا یک سری از منابع دیگر که نمونه‌سازی از آن‌ها بسیار سخت یا غیرممکن است. به صورت خلاصه، وکیل پوشه بند یا نیروی اجرایی‌ای است که توسط مشتری صدا زده می‌شود تا به شیء اصلی در پشت پرده دسترسی پیدا کند. استفاده از وکیل می‌تواند انتقال به شیء واقعی را راحت‌تر کند یا روش‌های دیگری برای آن به وجود بیاورد. در الگوی وکالت قابلیت‌های اضافه‌تر می‌تواند به وجود بیاید، به‌طور مثال، استفاده از حافظه موقت، برای وقتی که استفاده از شیء اصلی هزینه پردازشی و هزینه دسترسی به حافظه زیادی داشته باشد، یا در چک کردن شرایط اولیه قبل از آن که عملیات بر روی شیء اصلی اعمال شود. برای مشتری با استفاده از وکیل مانند استفاده از شیء اصلی است، چراکه رابط هر دو مشابه است.

مثال در دنیای واقعی

الگوی وکالت یک نگهدارنده یا یک مکان را برای دسترسی به یک شیء فراهم می‌کند. چک یا پیش‌نویس بانکی وکالت برای بودجه در یک حساب است. از چک می‌توان به‌جای پول برای انجام خرید استفاده کرد و درنهایت دسترسی به پول نقد را در حساب صادرکننده کنترل کرد.



حالا به سراغ پیاده‌سازی این الگو می‌رومیم. به مثال زیر دقت کنید.



```

"""
Provide a surrogate or placeholder for another object to control access
to it or add other responsibilities.
"""

import abc

class Subject(metaclass=abc.ABCMeta):
    """
    Define the common interface for RealSubject and Proxy so that a
    Proxy can be used anywhere a RealSubject is expected.
    """

    @abc.abstractmethod
    def request(self):
        pass

class Proxy(Subject):
    """
    Maintain a reference that lets the proxy access the real subject.
    Provide an interface identical to Subject's.
    """

    def __init__(self, real_subject):
        self._real_subject = real_subject

    def request(self):
        # ...
        self._real_subject.request()
        # ...

class RealSubject(Subject):
    """
    Define the real object that the proxy represents.
    """

    def request(self):
        pass

def main():
    real_subject = RealSubject()
    proxy = Proxy(real_subject)
    proxy.request()

if __name__ == "__main__":
    main()

```

الگوهای رفتاری

الگوهای طراحی رفتاری یا Behavioral مجموعه‌ای از راهکارهای کد نویسی مربوط به تعامل و ارتباط اشیا هستند. این الگوهای طراحی عبارت‌اند از:



Chain of responsibility

کار این الگوی طراحی به این صورت است که وقتی یک request دریافت می‌شود می‌تواند بر اساس نوع آن سناریوهای مختلفی جهت پردازش آن request وجود داشته باشد.

تعریفی که Gang of Four از این الگوی طراحی کرده است به صورت زیر است: "اجتناب از وابستگی بین فرستنده Request و دریافت‌کننده آن با ایجاد قابلیت رسیدگی به Request توسط شیوه‌ها و سناریوهای های مختلف".

در این الگو زنجیره request را دریافت کرده و سپس این Request را بین Chain (حلقه) های مختلف ارسال می‌کند تا زمانی که یک chain این درخواست را پردازش کند و هنگامی که این درخواست توسط یکی از Chain ها پردازش شود ارسال متوقف می‌شود. در حقیقت Request وارد Chain های مختلف می‌شود و در زنجیره حرکت می‌کند. زمانی که یک زنجیره قابلیت پردازش این Request را داشته باشد حرکت Request بین زنجیره‌های مختلف متوقف می‌شود.

نقش‌های شرکت‌کننده در این الگوی طراحی موارد زیر است:

1. یک Handler Interface است که signature متد handleRequest را تعريف می‌کند. متد در شامل نام متد وردي و خروجي متد است. دقیقاً اتفاقی که در یک interface یا یک متد Abstract می‌افتد) و همچنین امكان set کردن successor را نیز در این کلاس دیده شده است.

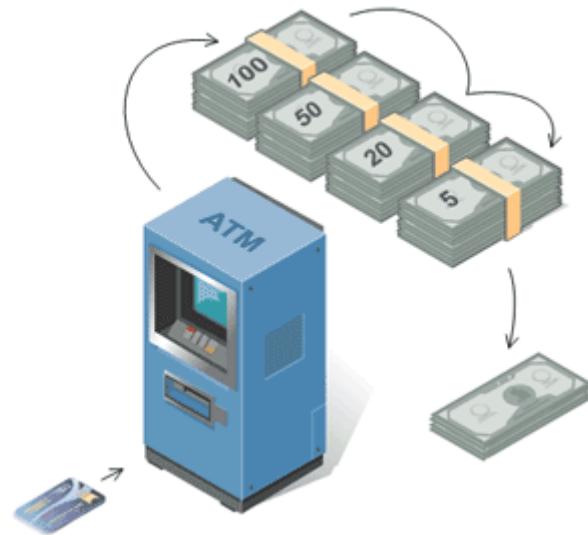
2. که از کلاس Handler ارث می‌برد و وظیفه پیاده‌سازی متد handleRequest را دارد و همچنین به successor نیز دسترسی دارد. اگر یک Concrete Handler نتواند یک request را کند آن request را از طریق Successor به Handler دیگری ارسال می‌کند.

3. Client: وظیفه ارسال request به یک Concrete handler را دارد.

مثال در دنیای واقعی

الگوی زنجیره‌ای از مسئولیت با قرار دادن بیش از یک موضوع برای رسیدگی به درخواست، از اتصال فرستنده درخواست به گیرنده جلوگیری می‌کند. دستگاه خودپرداز از مکانیسم دادن پول از زنجیره مسئولیت استفاده می‌کند.





حالا به سراغ پیاده سازی این الگو میرویم. به مثال زیر دقت کنید.



```

"""
Avoid coupling the sender of a request to its receiver by giving
more than one object a chance to handle the request. Chain the receiving
objects and pass the request along the chain until an object handles it.
"""

import abc

class Handler(metaclass=abc.ABCMeta):
    """
    Define an interface for handling requests.
    Implement the successor link.
    """

    def __init__(self, successor=None):
        self._successor = successor

    @abc.abstractmethod
    def handle_request(self):
        pass

class ConcreteHandler1(Handler):
    """
    Handle request, otherwise forward it to the successor.
    """

    def handle_request(self):
        if True: # if can_handle:
            pass
        elif self._successor is not None:
            self._successor.handle_request()

class ConcreteHandler2(Handler):
    """
    Handle request, otherwise forward it to the successor.
    """

    def handle_request(self):
        if False: # if can_handle:
            pass
        elif self._successor is not None:
            self._successor.handle_request()

def main():
    concrete_handler_1 = ConcreteHandler1()
    concrete_handler_2 = ConcreteHandler2(concrete_handler_1)
    concrete_handler_2.handle_request()

if __name__ == "__main__":
    main()

```

Command

با استفاده از این الگوی طراحی می‌توانیم فرآخوانی یا Call کردن یک متده را بسته‌بندی (Encapsulate) کنیم که این کار فواید زیادی دارد.



قبل از شروع توضیح این الگو نیاز داریم که کمی راجع به Encapsulation یا بسته‌بندی سازی در برنامه‌نویسی شیء‌گرا باهم صحبت کنیم.

به زبان ساده Encapsulation یا بسته‌بندی سازی به معنی از دسترس خارج کردن جزئیات داخلی که موردنیاز نیست از کاربر است. به طور مثال در دنیای واقعی وقتی ما یک دارویی که به شکل کپسول است را خریداری می‌کنیم ما به جزئیات داخلی آن دسترسی نداریم و ما فقط با قورت می‌توانیم از آن استفاده کنیم! در دنیای برنامه‌نویسی هم همین قضیه وجود دارد. فرض کنیم کاربر به یک سری اطلاعات نیاز دارد که کلاس A وظیفه m1 را اختیار گذاشتن این داده را به کاربر بر عهده دارد. در کلاس A دو متدهد m1 و m2 وجود دارد. وظیفه متدهد m1 تولید داده است و وظیفه متدهد m2 بازگرداندن داده تولیدشده توسط متدهد m1 است. برای کاربر ما مهم این است که داده در اختیارش قرار بگیرد. پس اینکه به متدهد m1 دسترسی داشته باشد بی‌معنی است. یکی از راههای بسته‌بندی سازی یا Encapsulation در زبان‌های برنامه‌نویسی استفاده از Access Specifire ها (عمومی^{۴۰} و خصوصی^{۴۱} و...) هست. سطوح‌های مختلفی برای بسته‌بندی یا Encapsulation وجود دارد. می‌توان دسترسی به یک کلاس را مدیریت کرد یا دسترسی به متدهای یک کلاس را مدیریت کرد و یا Property ها و فیلدۀای یک کلاس رو مدیریت کرد.

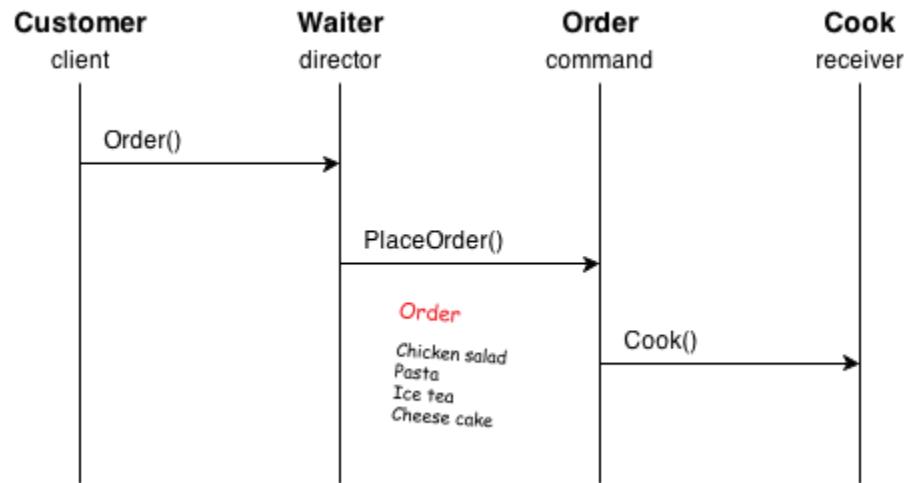
حالا به بحث خودمان یعنی الگوی طراحی Command برمی‌گردیم. ما با استفاده از این الگوی طراحی فراخوانی یا call کردن یک متدهد را Encapsulate می‌کنیم و با استفاده از این کپسوله کردن ویژگی‌های زیادی را در اختیار ما می‌گذارد از جمله می‌توانیم آن را ذخیره کنیم و دوباره فراوانی کنیم و یا امکان redo یا undo کردن یک متدهد ایجاد کنیم.

مثال در دنیای واقعی

الگوی Command اجازه می‌دهد تا درخواست‌ها به عنوان اشیاء محصور شوند و از این طریق به مشتری اجازه می‌دهد تا با درخواست‌های مختلف پارامتر داده شود. "برگه سفارش خوراک" در یک رستوران نمونه‌ای از الگوی Command است. پیشخدمت از مشتری سفارش یا دستور می‌گیرد و با نوشتن آن در برگه سفارش خوراک، آن سفارش را محصور می‌کند. سفارش سپس برای یک آشپز در سفارش کوتاه صفت می‌شود. توجه داشته باشید که پد "برگه‌های سفارش خوراک" استفاده شده توسط هر پیشخدمت به منو بستگی ندارد و بنابراین می‌توانند از دستورات مربوط به طبخ موارد مختلف استفاده کنند.

Public^{۴۰}
Private^{۴۱}





حالا به سراغ پیاده سازی این الگو میرویم. به مثال زیر دقت کنید.



```

"""
Encapsulate a request as an object, thereby letting you parameterize
clients with different requests, queue or log requests, and support
undoable operations.
"""
import abc

class Invoker:
    """
        Ask the command to carry out the request.
    """
    def __init__(self):
        self._commands = []

    def store_command(self, command):
        self._commands.append(command)

    def execute_commands(self):
        for command in self._commands:
            command.execute()

class Command(metaclass=abc.ABCMeta):
    """
        Declare an interface for executing an operation.
    """
    def __init__(self, receiver):
        self._receiver = receiver

    @abc.abstractmethod
    def execute(self):
        pass

class ConcreteCommand(Command):
    """
        Define a binding between a Receiver object and an action.
        Implement Execute by invoking the corresponding operation(s) on
        Receiver.
    """
    def execute(self):
        self._receiver.action()

class Receiver:
    """
        Know how to perform the operations associated with carrying out a
        request. Any class may serve as a Receiver.
    """
    def action(self):
        pass

def main():
    receiver = Receiver()
    concrete_command = ConcreteCommand(receiver)
    invoker = Invoker()
    invoker.store_command(concrete_command)
    invoker.execute_commands()

if __name__ == "__main__":
    main()

```

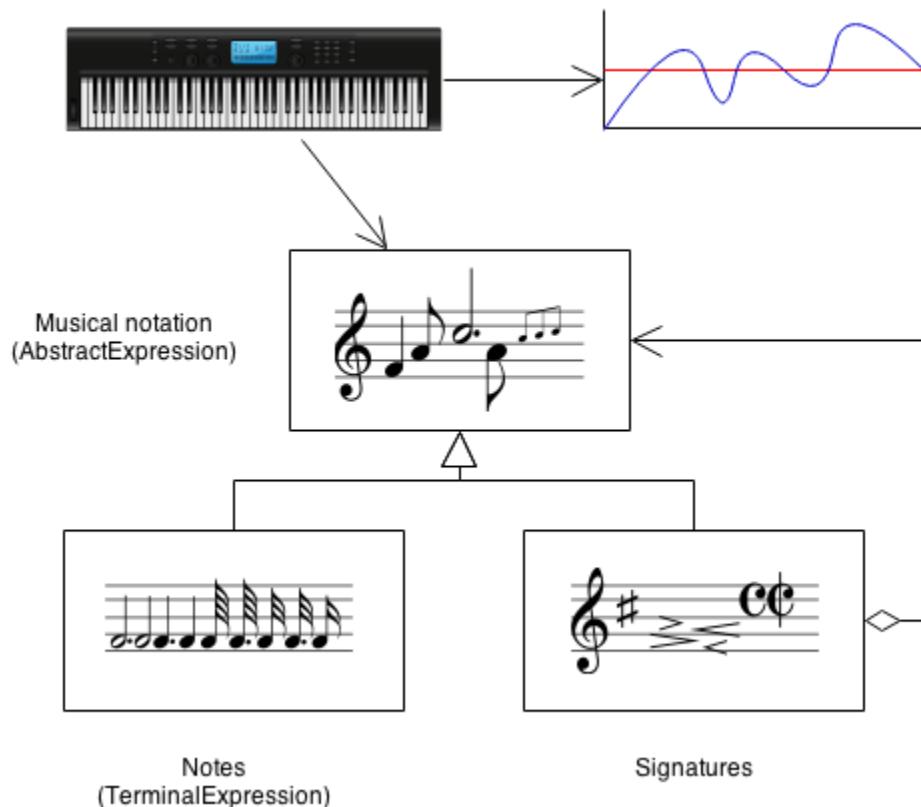


Interpreter

همان‌طور که اسم این الگوی طراحی پیداست این الگو روش و راه حل مناسبی جهت پیاده‌سازی یک مفسر یا مترجم در اختیار ما قرار می‌دهد. با استفاده از این الگو یک زبان ساده را می‌توانیم پیاده‌سازی کنیم.

مثال در دنیای واقعی

الگوی مترجم یک گرامر را برای یک زبان و یک مترجم برای تفسیر گرامر تعریف می‌کند. نوازنده‌گان نمونه‌ای از مترجمان هستند. گام صدا و مدت‌زمان آن می‌تواند در نمادهای موسیقی روی دفتر نت نشان داده شود. این نمادها زبان موسیقی را فراهم می‌کند. نوازنده‌گان موسیقی از این امتیاز استفاده می‌کنند و قادر به تکثیر صدای اصلی و مدت‌زمان هر صدا هستند.



حالا به سراغ پیاده‌سازی این الگو می‌رویم. به مثال زیر دقت کنید.



```

"""
Define a representation for a grammar of the given language along with an
interpreter that uses the representation to interpret sentences in the
language.
"""
import abc

class AbstractExpression(metaclass=abc.ABCMeta):
    """
    Declare an abstract Interpret operation that is common to all nodes
    in the abstract syntax tree.
    """
    @abc.abstractmethod
    def interpret(self):
        pass

class NonterminalExpression(AbstractExpression):
    """
    Implement an Interpret operation for nonterminal symbols in the grammar.
    """
    def __init__(self, expression):
        self._expression = expression

    def interpret(self):
        self._expression.interpret()

class TerminalExpression(AbstractExpression):
    """
    Implement an Interpret operation associated with terminal symbols in
    the grammar.
    """
    def interpret(self):
        pass

def main():
    abstract_syntax_tree = NonterminalExpression(TerminalExpression())
    abstract_syntax_tree.interpret()

if __name__ == "__main__":
    main()

```

Iterator

در همه زبان‌های برنامه‌نویسی داشتن مجموعه و فهرستی از اشیا (Collection) اتفاقی است که بسیار رخداده. زمانی هم که ما یک مجموعه‌ای از اشیا داریم پیمایش این مجموعه برای دسترسی به اشیا یکی از مسائلی هست که ما با آن روبرو می‌شویم. در اکثر زبان‌های برنامه‌نویسی تکنیک‌هایی برای پیمایش انواع وجود دارد. (در زبان برنامه‌نویسی C# List و ArrayList دو نوع Collection می‌باشند)

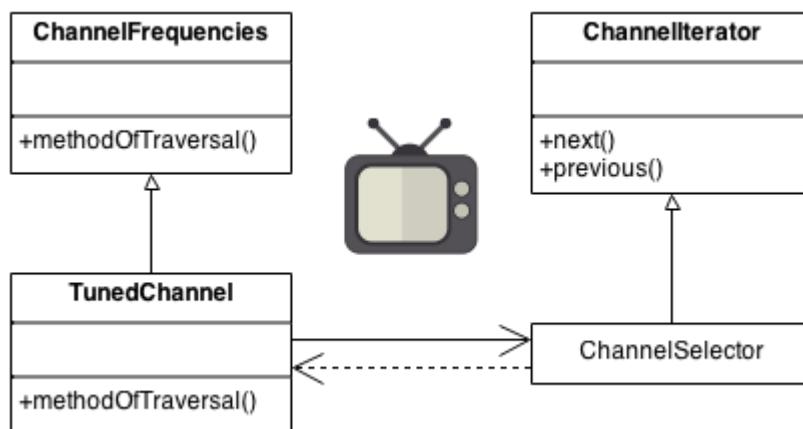


فرض کنید ما یک Collection داریم که آن را بهوسیله یک حلقه for پیمایش کنیم. این پیمایش به صورت زیر می‌شود.

مثال در دنیای واقعی

روش‌هایی برای دستیابی به عناصر یک جسم به صورت متوالی و بدون افشاگری ساختار زیرین جسم ارائه می‌دهد. پروندهای اشیاء کل هستند. در تنظیمات اداری که دسترسی به پروندهای از طریق کارمندان اداری یا دبیرخانه انجام می‌شود، الگوی Iterator با منشی که به عنوان Iterator عمل می‌کند نشان داده می‌شود.

در تلویزیون‌های اولیه، از شماره‌گیری برای تغییر شبکه‌ها استفاده می‌شد. هنگام گشت‌وگذار در بین شبکه‌ها، بیننده ملزم به انتقال شماره‌گیری شبکه مورد نظر، صرف نظر از این که آیا آن شبکه را دریافت کرده است یا خیر است. در تلویزیون‌های مدرن از دکمه بعدی و قبلی استفاده می‌شود. وقتی بیننده دکمه "بعدی" را انتخاب کند، شبکه تنظیم شده بعدی نمایش داده می‌شود. در نظر بگیرید در یک اتاق هتل در یک شهر عجیب تلویزیون را تماشا کنید. هنگام گشت‌وگذار از طریق شبکه‌های تلویزیونی، شماره شبکه اهمیتی ندارد، اما محتوای آن مهم است. اگر بیننده به محتوای آن شبکه علاقه‌ای نداشته باشد، بیننده می‌تواند بدون اطلاع از شماره آن، شبکه بعدی را درخواست کند.



حالا به سراغ پیاده سازی این الگو می‌رویم. به مثال زیر دقت کنید.



```

"""
Provide a way to access the elements of an aggregate objects sequentially
without exposing its underlying representation.
"""

import collections.abc

class ConcreteAggregate(collections.abc.Iterable):
    """
    Implement the Iterator creation interface to return an instance of
    the proper ConcreteIterator.
    """

    def __init__(self):
        self._data = None

    def __iter__(self):
        return ConcreteIterator(self)

class ConcreteIterator(collections.abc.Iterator):
    """
    Implement the Iterator interface.
    """

    def __init__(self, concrete_aggregate):
        self._concrete_aggregate = concrete_aggregate

    def __next__(self):
        if True: # if no_elements_to_traverse:
            raise StopIteration
        return None # return element

def main():
    concrete_aggregate = ConcreteAggregate()
    for _ in concrete_aggregate:
        pass

if __name__ == "__main__":
    main()

```

Mediator

الگوی طراحی Mediator همان طور که اسم از آن نیز می‌توان حدس زد وظیفه‌ی یک واسط و مدیریت کردن ارتباطات بین اشیا را بر عهده دارد.

نقش‌های شرکت‌کننده در این الگوی طراحی موارد زیر هستند:

Mediator: یک اینترفیس است برای ارتباط بین اشیایی که باهم کار می‌کنند.
ConcreteMediator: ارتباط بین اشیا در این کلاس تعریف می‌شود و همچنین اشیایی که قرار است باهم در ارتباط باشند را می‌شناسد.

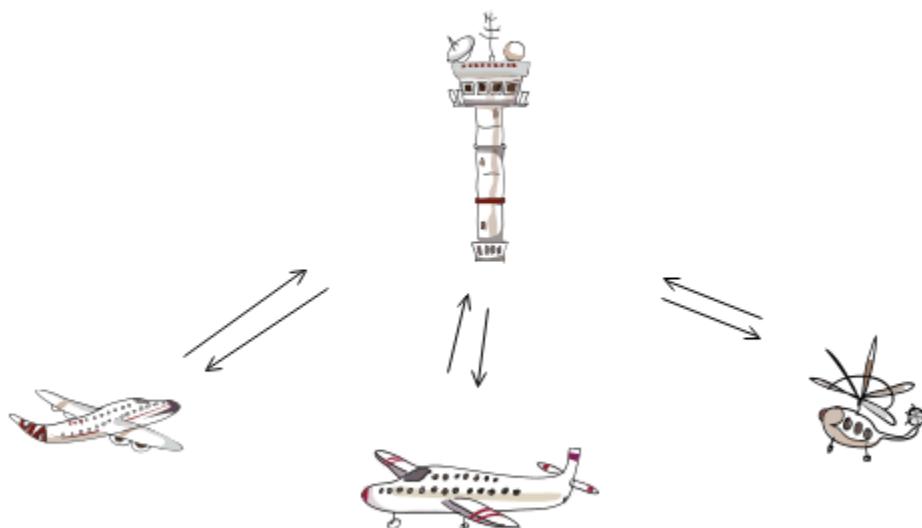


همان اشیایی می‌باشند که باهم در ارتباط هستند و **Mediator** را می‌شناسند.

مثال در دنیای واقعی

واسطه‌گر شیئی را تعریف می‌کند که چگونگی تعامل مجموعه‌ای از اشیاء را کنترل می‌کند. برج مراقبت در یک فرودگاه نحوه استفاده از این الگو را به خوبی نشان می‌دهد. خلبانان هواپیماهایی که به منطقه فرودگاه نزدیک می‌شوند یا در حال ترک آن هستند، با برج مراقبت ارتباط برقرار می‌کنند نه اینکه صریحاً با یکدیگر ارتباط برقرار کنند. محدودیتهایی که می‌توانند از زمین برخیزند یا در فرودگاه فرود بیایند توسط برج اعمال می‌شود. ذکر این نکته حائز اهمیت است که برج مراقبت تمام پرواز را کنترل نمی‌کند. این فقط برای اعمال محدودیتها در منطقه فرودگاه وجود دارد.

ATC Mediator



حالا به سراغ پیاده سازی این الگو میرویم. به مثال زیر دقت کنید.



```

"""
Define an object that encapsulates how a set of objects interact.
Mediator promotes loose coupling by keeping objects from referring to
each other explicitly, and it lets you vary their interaction
independently.
"""

class Mediator:
    """
    Implement cooperative behavior by coordinating Colleague objects.
    Know and maintains its colleagues.
    """
    def __init__(self):
        self._colleague_1 = Colleague1(self)
        self._colleague_2 = Colleague2(self)

class Colleague1:
    """
    Know its Mediator object.
    Communicate with its mediator whenever it would have otherwise
    communicated with another colleague.
    """
    def __init__(self, mediator):
        self._mediator = mediator

class Colleague2:
    """
    Know its Mediator object.
    Communicate with its mediator whenever it would have otherwise
    communicated with another colleague.
    """
    def __init__(self, mediator):
        self._mediator = mediator

def main():
    mediator = Mediator()

    if __name__ == "__main__":
        main()

```

Memento

کاربرد این الگوی طراحی زمانی است که ما نیاز داریم یک حالت خاص از یک شیء را ذخیره کنیم تا در صورت نیاز شیء را بعد از تغییر، به حالت ذخیره شده برگردانیم. همه ما کم و بیش درگیر بازی های کامپیوتری بوده ایم. Save کردن یک بازی دقیقاً حالتی است که می شود از الگوی طراحی Memento استفاده کرد. فقط نکته ای که در Memento باید مورد توجه قرار بگیرد این است که فقط همان Object میتواند Memento را ایجاد کرده است می تواند به Memento خودش دسترسی داشته باشد.



نقش‌های شرکت‌کننده در این الگو موارد زیر هستند:

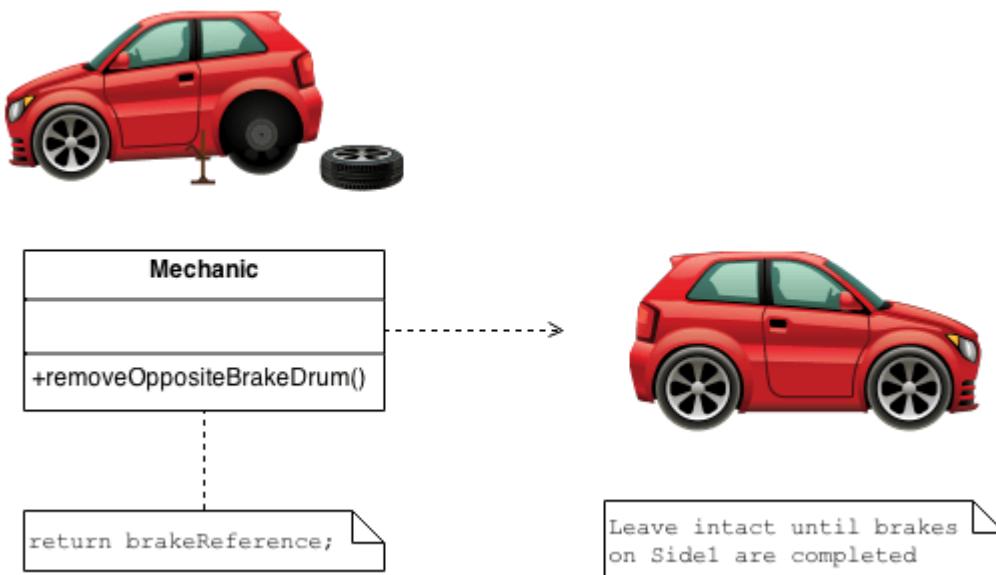
1. Memento: حالت شیء را ذخیره می‌کند. Memento بر حساب نیاز می‌تواند همه یا قسمتی از حالت داخلی یک شیء را ذخیره کند. همچنین در برابر دسترسی‌های اشیاء غیر از شیء اصلی محافظت می‌کند.

2. Originator: Memento اشامل یک Snapshot از حالت فعلی را ایجاد می‌کند. همچنین از برای Restore کردن استفاده می‌کند.

3. Caretaker: مسئول حفاظت و نگهداری از Memento است. همچنین هیچ عملیاتی رو محتوای انجام نمی‌دهد.

مثال در دنیای واقعی

Memento وضعیت داخلی یک شیء را ضبط و بیرونی می‌کند تا بعداً شیء به آن حالت بازگردد. این الگوی در بین مکانیک‌ها که لنت را روی اتومبیل‌های خود ترمیم می‌کنند متداول است. تایرها از هر دو طرف برداشته می‌شوند و ترمزهای راست و چپ در دسترس قرار می‌گیرند. فقط یک طرف جدا می‌شود و طرف دیگر به عنوان بررسی چگونگی عملکرد قطعات ترمز در کنار هم مورد استفاده قرار می‌گیرد. فقط پس از اتمام کار از یک طرف، طرف دیگر نیز هم جدا می‌شود. هنگامی که طرف دوم جدا شود، طرف اول به عنوان Memento عمل می‌کند.



حالا به سراغ پیاده سازی این الگو می‌رویم. به مثال زیر دقت کنید.



```

"""
Capture and externalize an object's internal state so that the object
can be restored to this state later, without violating encapsulation.
"""

import pickle

class Originator:
    """
    Create a memento containing a snapshot of its current internal
    state.
    Use the memento to restore its internal state.
    """

    def __init__(self):
        self._state = None

    def set_memento(self, memento):
        previous_state = pickle.loads(memento)
        vars(self).clear()
        vars(self).update(previous_state)

    def create_memento(self):
        return pickle.dumps(vars(self))

    def main():
        originator = Originator()
        memento = originator.create_memento()
        originator._state = True
        originator.set_memento(memento)

    if __name__ == "__main__":
        main()

```

Observer

این الگو زمانی مور استفاده قرار می‌گیرد که ما چندین شیء داریم (Observers) که به یک شیء (Subject) وابسته هستند و زمانی که Subject ما تغییر حالت می‌دهد نیاز است شیء‌های Observer ما از این تغییر حالت آگاه شوند.

نقش‌های شرکت‌کننده در این الگو موارد زیر هستند:

Subject: Observers .1 باشند. همچنین یک Interface جهت اضافه یا کم کردن Observer در اختیار قرار می‌دهد.



2. ConcreteSubject: حالتهای مختلف یک Subject را در خود دارد. زمانی که تغییر حالت می‌دهد به Observer هایش اعلام می‌کند.

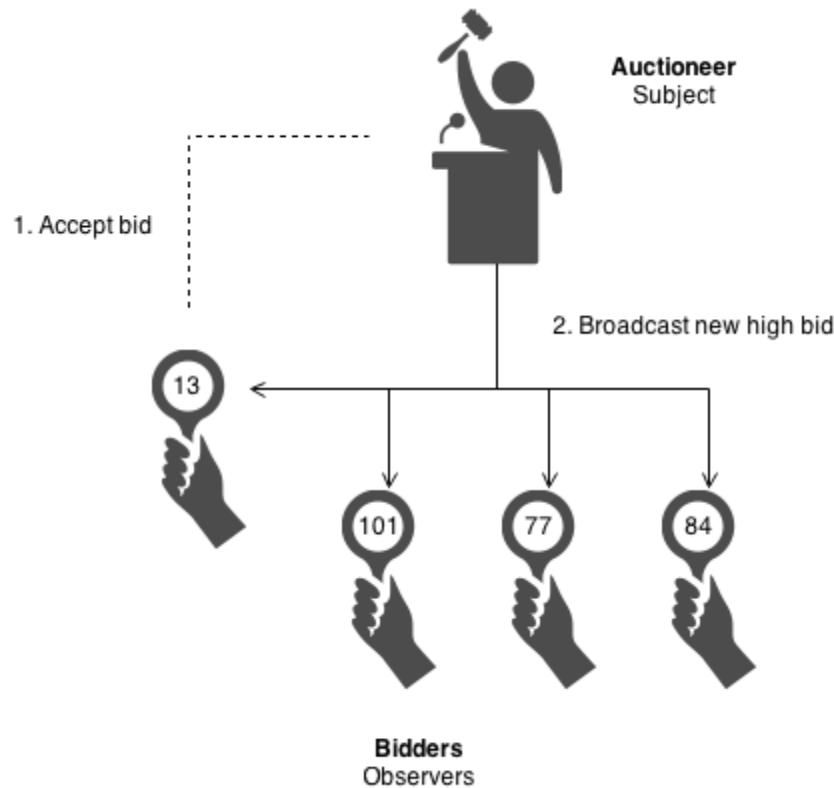
3. Observer: یک Subject دریافت تغییرات Notification جهت Updating Interface در اختیار قرار می‌دهد.

4. ConcreteObserver: یک Reference به شیء ConcreteSubject در خود دارد. حالتی را که باید توسط Subject ها پایدار بماند را در خود دارد. پیاده‌سازی می‌کند Updating Interface جهت دریافت حالت Subject بهروز شده ها.

مثال در دنیای واقعی

Observer یک رابطه یک به چند را تعریف می‌کند به‌طوری‌که وقتی حالت یک شیء تغییر کند، به دیگران اطلاع داده می‌شود و به صورت خودکار بروز می‌شوند. برخی از حراج‌ها این الگوی را نشان می‌دهند. هر پیشنهاد‌دهنده دارای یک تابلوی شماره‌گذاری شده است که برای نشان دادن پیشنهاد استفاده می‌شود. برگزار‌کننده مزایده را شروع می‌کند و هنگامی که یک تابلو برای قبول پیشنهاد بالا می‌رود، "مشاهده" می‌کند. پذیرش پیشنهاد قیمت، قیمت پیشنهادی را که به صورت پیشنهاد قیمت جدید برای همه داوطلبان پخش می‌شود، تغییر می‌دهد.





حالا به سراغ پیاده سازی این الگو میرویم. به مثال زیر دقت کنید.



```

"""
Provide a way to access the elements of an aggregate objects sequentially
without exposing its underlying representation.
"""

import collections.abc

class ConcreteAggregate(collections.abc.Iterable):
    """
    Implement the Iterator creation interface to return an instance of
    the proper ConcreteIterator.
    """

    def __init__(self):
        self._data = None

    def __iter__(self):
        return ConcreteIterator(self)

class ConcreteIterator(collections.abc.Iterator):
    """
    Implement the Iterator interface.
    """

    def __init__(self, concrete_aggregate):
        self._concrete_aggregate = concrete_aggregate

    def __next__(self):
        if True: # if no_elements_to_traverse:
            raise StopIteration
        return None # return element

def main():
    concrete_aggregate = ConcreteAggregate()
    for _ in concrete_aggregate:
        pass

if __name__ == "__main__":
    main()

```

State

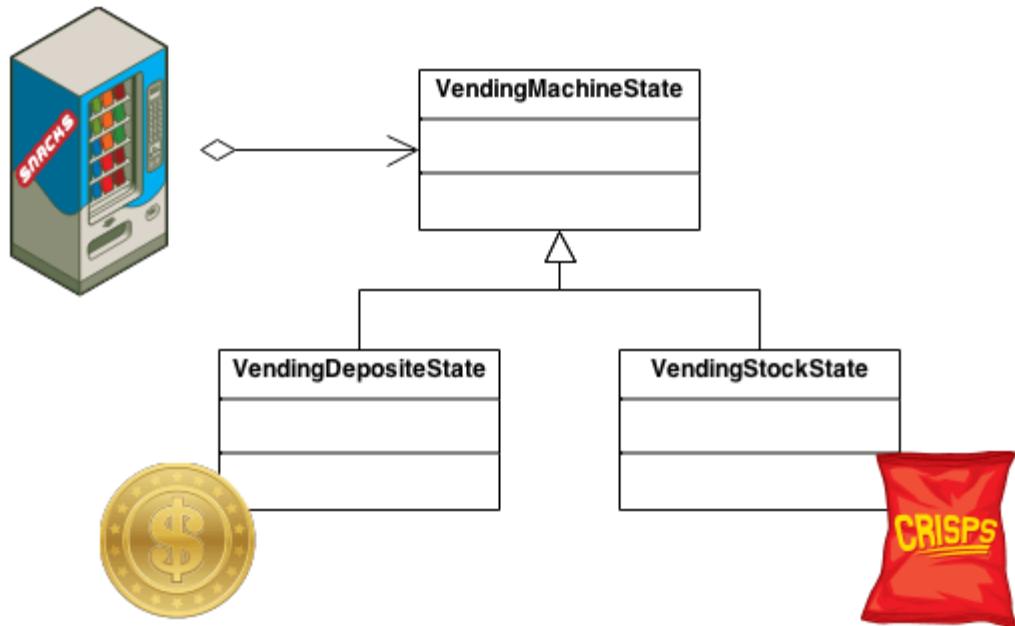
این Desgin Pattern این قابلیت رو به ما می‌دهد که ما بتوانیم بر اساس تغییر حالت داخلی یک شیء عملکردهای متفاوتی برای متدهای آن شیء داشته باشیم.

مثال در دنیای واقعی

الگوی State به یک شیء اجازه می‌دهد تا هنگام تغییر وضعیت داخلی، رفتار خود را تغییر دهد. این الگوی را می‌توان در یک ماشین فروش مشاهده کرد. ماشین‌های واریز حالت‌هایی دارند که بر اساس موجودی، میزان ارز سپرده‌گذاری، امکان ایجاد تغییر، کالای انتخاب شده و غیره وجود دارد. وقتی پول واریز می‌شود و انتخابی صورت



می‌گیرد، یک ماشین فروش کالا را تحویل می‌دهد و هیچ تغییری ندارد، محصول را تحویل می‌دهد و تغییر می‌کند، به دلیل کمبود پول در سپرده، کالایی را تحویل نمی‌دهد و یا به دلیل کاهش موجودی هیچ محصولی را تحویل نمی‌دهد.



حالا به سراغ پیاده سازی این الگو میرویم. به مثال زیر دقت کنید.



```

"""
Allow an object to alter its behavior when its internal state changes.
The object will appear to change its class.
"""
import abc

class Context:
    """
    Define the interface of interest to clients.
    Maintain an instance of a ConcreteState subclass that defines the
    current state.
    """
    def __init__(self, state):
        self._state = state

    def request(self):
        self._state.handle()

class State(metaclass=abc.ABCMeta):
    """
    Define an interface for encapsulating the behavior associated with a
    particular state of the Context.
    """
    @abc.abstractmethod
    def handle(self):
        pass

class ConcreteStateA(State):
    """
    Implement a behavior associated with a state of the Context.
    """
    def handle(self):
        pass

class ConcreteStateB(State):
    """
    Implement a behavior associated with a state of the Context.
    """
    def handle(self):
        pass

def main():
    concrete_state_a = ConcreteStateA()
    context = Context(concrete_state_a)
    context.request()

if __name__ == "__main__":
    main()

```

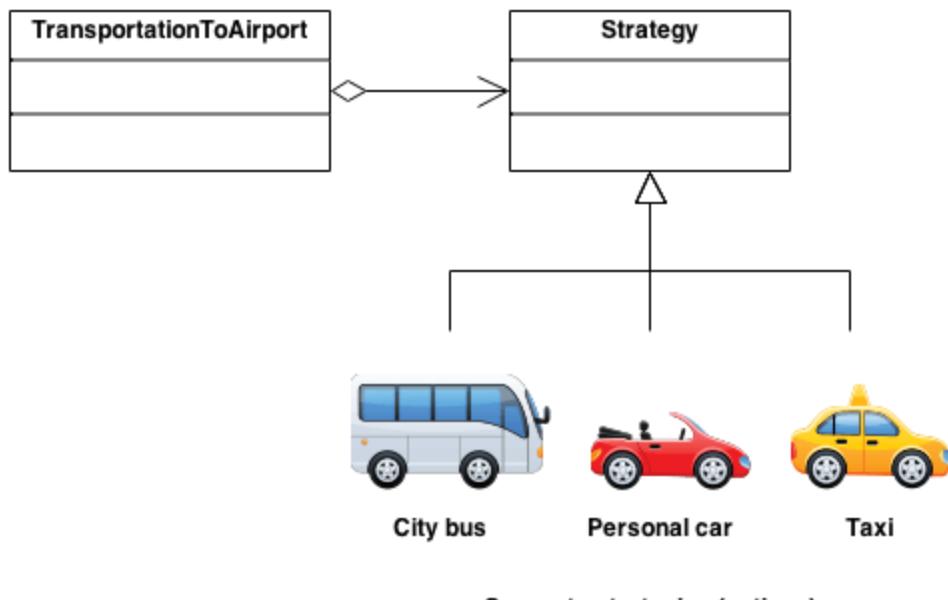


Strategy

این الگوی زمانی مورداستفاده قرار می‌گیرد که ما قصد داریم الگوریتم‌های مختلف که از راههای متفاوت دارای نتایج یکسان هستند را در یک کلاس داشته باشیم و آنها را کپسوله سازی کنیم و در صورت نیاز هر کدام از آنها را استفاده کنیم.

مثال در دنیای واقعی

استراتژی مجموعه‌ای از الگوریتم‌ها را تعریف می‌کند که می‌توانند به صورت متقابل مورداستفاده قرار گیرند. نحوه حمل و نقل به فرودگاه نمونه‌ای از یک الگوی طراحی استراتژی است. گزینه‌های مختلفی از جمله رانندگی اتومبیل شخصی، گرفتن تاکسی، اتوبوس فرودگاهی، اتوبوس شهری یا سرویس لیموزین وجود دارد. برای برخی از فرودگاه‌ها، متروها و بالگردها نیز به عنوان شیوه حمل و نقل به فرودگاه در دسترس هستند. هر یک از این روش‌های حمل و نقل مسافر را به فرودگاه می‌رساند و از آنها می‌توان به صورت متناوب استفاده کرد. مسافر باید استراتژی را بر اساس مبادلات بین هزینه، راحتی و زمان انتخاب کند.



حالا به سراغ پیاده سازی این الگو می‌رویم. به مثال زیر دقت کنید.



```

"""
Define a family of algorithms, encapsulate each one, and make them
interchangeable. Strategy lets the algorithm vary independently from
clients that use it.
"""
import abc

class Context:
    """
    Define the interface of interest to clients.
    Maintain a reference to a Strategy object.
    """
    def __init__(self, strategy):
        self._strategy = strategy

    def context_interface(self):
        self._strategy.algorithm_interface()

class Strategy(metaclass=abc.ABCMeta):
    """
    Declare an interface common to all supported algorithms. Context
    uses this interface to call the algorithm defined by a
    ConcreteStrategy.
    """
    @abc.abstractmethod
    def algorithm_interface(self):
        pass

class ConcreteStrategyA(Strategy):
    """
    Implement the algorithm using the Strategy interface.
    """
    def algorithm_interface(self):
        pass

class ConcreteStrategyB(Strategy):
    """
    Implement the algorithm using the Strategy interface.
    """
    def algorithm_interface(self):
        pass

def main():
    concrete_strategy_a = ConcreteStrategyA()
    context = Context(concrete_strategy_a)
    context.context_interface()

if __name__ == "__main__":
    main()

```



Template method

الگوی Template Method (متد قلابدار-متد قالب) در دسته الگوهای رفتاری قرار دارد.

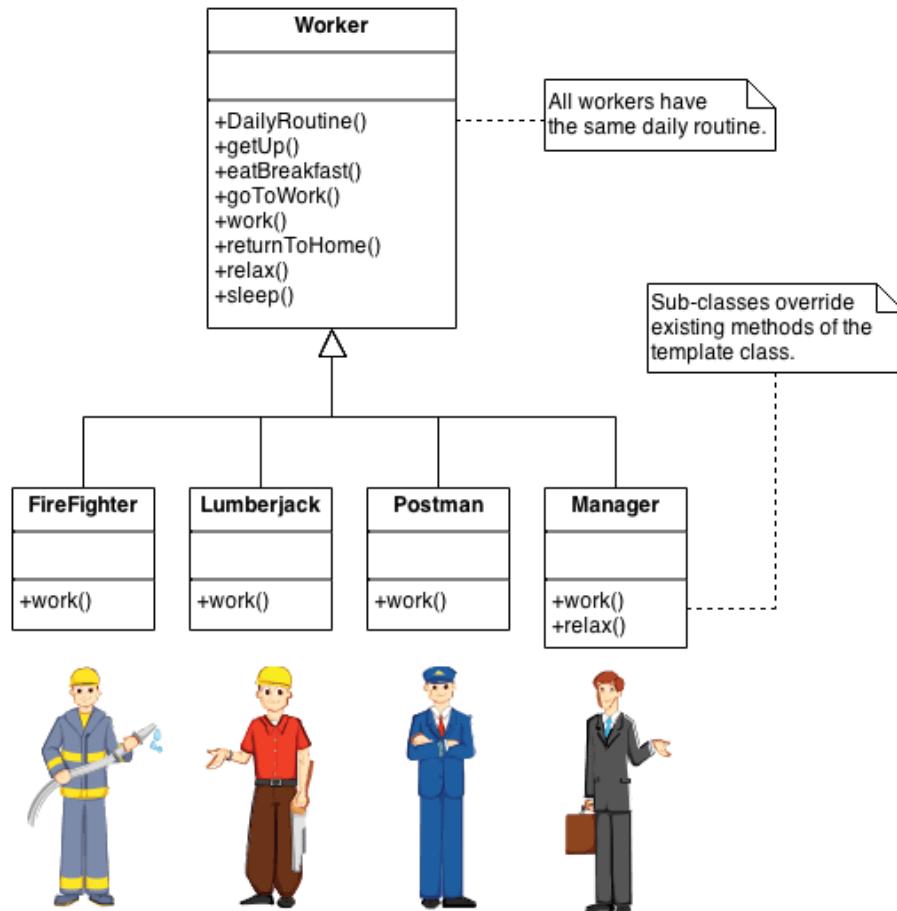
اسکلت یک الگوریتم در یک عملیات را تعریف می‌کند، بعضی مراحل را به زیر کلاس‌ها منتقل می‌کند. Template Method اجازه می‌دهد که زیر کلاس‌ها گام‌های خاصی از یک الگوریتم را دوباره تغییر دهند بدون تغییر ساختار الگوریتم. به عبارت دیگر اسکلت کلی الگوریتم را تعریف کرده و این امکان را فراهم می‌کند که بدون تغییر ساختار کلی الگوریتم بتوان برخی از مراحل آن را به زیر کلاس‌ها سپرد.

مثال در دنیای واقعی

روش قالب یا اسکلت یک الگوریتم را در یک عمل تعریف می‌کند و برخی مراحل را به زیر کلاس‌ها واگذار می‌کند. سازندگان خانه هنگام تهیه زیرمجموعه جدید از روش الگو استفاده می‌کنند. یک زیرمجموعه معمولی شامل تعداد محدودی از برنامه‌های کف با انواع مختلف در دسترس برای هر یک است. در یک طرح طبقه، پایه، فریم، لوله‌کشی و سیم‌کشی برای هر خانه یکسان خواهد بود. تنوع در مراحل بعدی ساخت معرفی شده است تا انواع گسترده‌تری از مدل‌ها را تولید کند.

مثال دیگر: کارهای روزمره یک کارگر





حالا به سراغ پیاده سازی این الگو میرویم. به مثال زیر دقت کنید.



```

"""
Define the skeleton of an algorithm in an operation, deferring some
steps to subclasses. Template Method lets subclasses redefine certain
steps of an algorithm without changing the algorithm's structure.
"""

import abc

class AbstractClass(metaclass=abc.ABCMeta):
    """
    Define abstract primitive operations that concrete subclasses define
    to implement steps of an algorithm.

    Implement a template method defining the skeleton of an algorithm.
    The template method calls primitive operations as well as operations
    defined in AbstractClass or those of other objects.
    """

    def template_method(self):
        self._primitive_operation_1()
        self._primitive_operation_2()

    @abc.abstractmethod
    def _primitive_operation_1(self):
        pass

    @abc.abstractmethod
    def _primitive_operation_2(self):
        pass

class ConcreteClass(AbstractClass):
    """
    Implement the primitive operations to carry out
    subclass-specific steps of the algorithm.
    """

    def _primitive_operation_1(self):
        pass

    def _primitive_operation_2(self):
        pass

def main():
    concrete_class = ConcreteClass()
    concrete_class.template_method()

if __name__ == "__main__":
    main()

```

Visitor

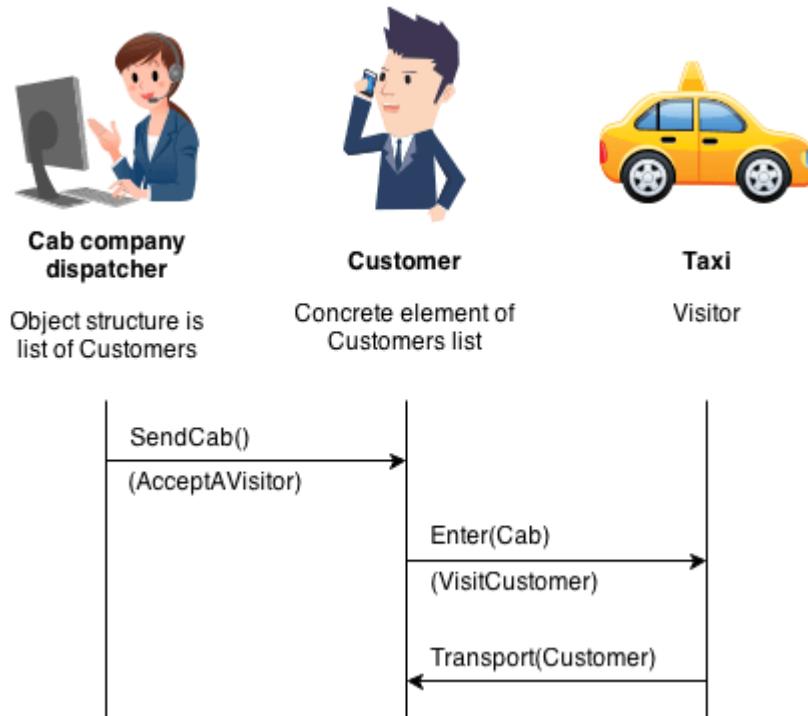
اگر بخواهیم به کلاسی بدون اینکه به ساختار آن دست بزنیم متدهایی را اضافه کنیم از این الگو استفاده می‌کنیم. این الگو تلاش می‌کند تا بین لایه Business logic (برای آشنایی با مفهوم Business Logic و



برنامه‌نویسی سه لایه به مقاله معماری سه لایه در ASP.NET مراجعه کنید) و الگوریتم جدایی ایجاد کند. ما می‌توانیم منطق جدیدی به برنامه اضافه کنیم بدون اینکه به ساختار دیتای خود دست بزنیم.

مثال در دنیای واقعی

الگوی Visitor عملیاتی است که باید روی عناصر ساختار یک شیء انجام شود بدون اینکه کلاس‌هایی را که در آن کار می‌کند تغییر دهید. این الگو را می‌توان در عملکرد یک شرکت تاکسی مشاهده کرد. وقتی شخصی با یک شرکت تاکسی (پذیرش بازدیدکننده) تماس می‌گیرد، این شرکت یک ماشین را برای مشتری اعزام می‌کند. پس از ورود به تاکسی، مشتری، ویزیتور دیگر کنترل حمل و نقل خود را ندارد و کنترل حمل و نقل با تاکسی (راننده) است.



حالا به سراغ پیاده سازی این الگو می‌رویم. به مثال زیر دقت کنید.



```

"""
Represent an operation to be performed on the elements of an object
structure. Visitor lets you define a new operation without changing the
classes of the elements on which it operates.
"""
import abc

class Element(metaclass=abc.ABCMeta):
    """
    Define an Accept operation that takes a visitor as an argument.
    """
    @abc.abstractmethod
    def accept(self, visitor):
        pass

class ConcreteElementA(Element):
    """
    Implement an Accept operation that takes a visitor as an argument.
    """
    def accept(self, visitor):
        visitor.visit_concrete_element_a(self)

class ConcreteElementB(Element):
    """
    Implement an Accept operation that takes a visitor as an argument.
    """
    def accept(self, visitor):
        visitor.visit_concrete_element_b(self)

class Visitor(metaclass=abc.ABCMeta):
    """
    Declare a Visit operation for each class of ConcreteElement in the
    object structure. The operation's name and signature identifies the
    class that sends the Visit request to the visitor. That lets the
    visitor determine the concrete class of the element being visited.
    Then the visitor can access the element directly through its
    particular interface.
    """
    @abc.abstractmethod
    def visit_concrete_element_a(self, concrete_element_a):
        pass

    @abc.abstractmethod
    def visit_concrete_element_b(self, concrete_element_b):
        pass

```



```

class ConcreteVisitor1(Visitor):
    """
    Implement each operation declared by Visitor. Each operation
    implements a fragment of the algorithm defined for the corresponding
    class of object in the structure. ConcreteVisitor provides the
    context for the algorithm and stores its local state. This state
    often accumulates results during the traversal of the structure.
    """
    def visit_concrete_element_a(self, concrete_element_a):
        pass

    def visit_concrete_element_b(self, concrete_element_b):
        pass

class ConcreteVisitor2(Visitor):
    """
    Implement each operation declared by Visitor. Each operation
    implements a fragment of the algorithm defined for the corresponding
    class of object in the structure. ConcreteVisitor provides the
    context for the algorithm and stores its local state. This state
    often accumulates results during the traversal of the structure.
    """
    def visit_concrete_element_a(self, concrete_element_a):
        pass

    def visit_concrete_element_b(self, concrete_element_b):
        pass

def main():
    concrete_visitor_1 = ConcreteVisitor1()
    concrete_element_a = ConcreteElementA()
    concrete_element_a.accept(concrete_visitor_1)

if __name__ == "__main__":
    main()

```

