ماژول ها و پکیج ها

برنامه نویسی ماژولار به فرآیند تقسیم یک وظیفه برنامه نویسی بزرگ و سخت به زیر وظایف یا ماژول‌های کوچکتر، قابل مدیریت‌تر و جداگانه اشاره دارد. سپس ماژول‌های جداگانه می‌توانند مانند بلوک‌های ساختمانی با هم ترکیب شوند تا یک برنامه بزرگتر ایجاد شود.

چندین مزیت برای ماژولار کردن کد در یک برنامه بزرگ وجود دارد:

- سادگی: به جای تمرکز بر روی کل مسئله، یک ماژول به طور معمول بر روی یک قسمت نسبتاً کوچکتر از مسئله تمرکز دارد. اگر شما در حال کار روی یک ماژول واحد هستید، دامنه مسئله کوچکتر است. این باعث می‌شود توسعه آسان‌تر و کم‌خطای‌تر باشد.
- قابلیت نگهداری: ماژول‌ها به طور معمول طوری طراحی شده‌اند که محدوده منطقی بین دامنه‌های مختلف مسئله را اعمال کنند. اگر ماژول‌ها به گونه‌ای نوشته شده باشند که حداقل وابستگی را داشته باشند، احتمال کمتری وجود دارد که تغییرات در یک ماژول تأثیرات آن را بر سایر قسمت‌های برنامه داشته باشد. (حتی شاید بتوانید تغییرات را در یک ماژول اعمال کنید، بدون این که هیچ دانش خارج از آن ماژول را داشته باشید.) این باعث می‌شود توانایی همکاری تعداد زیادی برنامه نویس در یک برنامه بزرگ را فراهم کند.
- قابل استفاده مجدد: قابل استفاده بودن عملکردهای تعریف شده در یک ماژول، به راحتی (با استفاده از گسترده مناسب) توسط سایر قسمت‌های برنامه استفاده شده است. این باعث حذف نیاز به تکثیر کدهای تکثیر شده خواهد شد.
- Scoping: معمولًا، ماژول‌ها فضای نام جدید را تعریف می‌کنند، که به جلوگیری از تصادف نام هویت‌ها در مناطق مختلف برنامه کمک می‌کند. (یکی از اصول در Zen of Python فضای نام‌ها یک ایده بزرگ و عالی هستند - بیایید بیشتر از آنها استفاده کنیم!)

توابع، ماژول‌ها و پکیج‌ها همه ساختارهایی در پایتون هستند که به مدولارسازی کد کمک می‌کنند.

ماژول‌های پایتون: مروری در واقع، سه روش مختلف برای تعریف یک ماژول در پایتون وجود دارد:

1. یک ماژول می‌تواند به زبان پایتون نوشته شود.
2. یک ماژول می‌تواند به زبان C نوشته شده و در زمان اجرا به صورت پویا بارگذاری شود، مانند ماژول re (عبارات با قاعده).
3. یک ماژول داخلی در خود ترجمه‌گر وجود دارد، مانند ماژول itertools.

محتویات یک ماژول در همه سه حالت به همان شکل با استفاده از دستور import قابل دسترسی هستند.

در اینجا، تمرکز بیشتر بر روی ماژول‌های نوشته شده به زبان پایتون خواهد بود. چیز جالب دربارهٔ ماژول‌های نوشته شده به زبان پایتون این است که بسیار ساده هستند. تنها کافی است یک فایل ایجاد کنید که حاوی کد پایتون قابل قبول باشد، سپس به فایل یک نام با پسوند py. بدهید. همین است! نیاز به دستورات خاص یا جادو ندارید.

برای مثال، فرض کنید یک فایل با نام mod.py ایجاد کرده‌اید که شامل مطالب زیر است:

In [ ]:

```
## wrtie this in separate file named mod.py
s = "If Comrade Napoleon says it, it must be right."
a = [100, 200, 300]

def foo(arg):
    print(f'arg = {arg}')

class Foo:
    pass
```

چندین شی در mod.py تعریف شده‌اند:

- s (یک رشته)
- a (یک لیست)
- foo() (یک تابع)
- Foo (یک کلاس)

با فرض اینکه mod.py در محل مناسبی قرار دارد، که به زودی بیشتر درباره آن یاد خواهید گرفت، این شی‌ها با وارد کردن ماژول به عنوان زیرمجموعه به صورت زیر قابل دسترسی هستند:

In [1]:

```
import mod
```

In [2]:

```
dir()
```

...

In [3]:

```
print(mod)
```

```
<module 'mod' from 'C:\\Users\\babak\\Python For Everyone\\07-Modules and
Packages\\mod.py'>
```

In [4]:

```
print(mod.a)
```

```
[100, 200, 300]
```

In [5]:

```
print(mod.s)
```

```
If Comrade Napoleon says it, it must be right.
```

In [6]:

```
mod.foo(['quux', 'corge', 'grault'])
```

```
arg = ['quux', 'corge', 'grault']
```

In [8]:

```
x = mod.Foo()
x
```

Out[8]:

```
<mod.Foo at 0x159d0648e50>
```

دستور import

محتویات ماژول با دستور import به فراخوانی کننده (caller) ارائه می‌شود. دستور import در چندین فرم مختلف وجود دارد که در زیر نشان داده شده‌اند.

import ساده‌ترین فرم، همان فرمی است که در بالا نشان داده شده است:

In [ ]:

```
import <module_name>
```

توجه کنید که این کار محتویات ماژول را به صورت مستقیم برای caller قابل دسترسی نمی‌کند. هر ماژول دارای جدول نمادهای خصوصی خود است که به عنوان جدول نمادهای سراسری برای تمام شی‌های تعریف شده در ماژول عمل می‌کند. بنابراین، یک ماژول فضای نام جداگانه‌ای ایجاد می‌کند، همانطور که در قبل گفته شد.

با دستور import ، فقط در جدول نماد caller قرار می‌گیرد. شی‌هایی که در ماژول تعریف شده‌اند، در جدول نماد خصوصی ماژول باقی می‌مانند.

شی‌های موجود در ماژول، فقط زمانی با استفاده از نام و با استفاده از نشان دهندهٔ نقطه، به صورت . قابل دسترسی هستند.

بعد از دستور import زیر، mod در جدول نماد محلی قرار می‌گیرد. بنابراین، mod در زمینهٔ محلی caller معنا دارد:

In [12]:

```
import mod

mod
```

Out[12]:

```
<module 'mod' from 'C:\\Users\\babak\\Python For Everyone\\07-Modules and
Packages\\mod.py'>
```

In [13]:

```
s
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[13], line 1
----> 1 s

NameError: name 's' is not defined
```

In [14]:

```
foo('quux')
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[14], line 1
----> 1 foo('quux')

NameError: name 'foo' is not defined
```

اما شما می توانید با استفاده از نام ماژول و علامت . به فضای خصوصی ماژول دسترسی پیدا نمایید:

In [15]:

```
mod.s
```

Out[15]:

```
'If Comrade Napoleon says it, it must be right.'
```

In [16]:

```
mod.foo('Quuux')
```

```
arg = Quuux
```

همچنین در صورت نیاز می توانین چندین ماژول رو بصورت همزمان وارد نمایید:

In [ ]:

```
import <module_name>[, <module_name> ...]
```

در صورتیکه به بخشی از یک ماژول نیاز دارید می توانید به نحو دیگری عملیات import را انجام دهید:

In [ ]:

```
from <module_name> import <name(s)>
```

In [17]:

```
from mod import s, foo
```

In [18]:

```
s
```

Out[18]:

```
'If Comrade Napoleon says it, it must be right.'
```

In [19]:

```
foo('Quux')
```

```
arg = Quux
```

حتی در صورت نیاز می توانید تمام فضای خصوصی ماژول را بصورت کامل و عمومی در کد خود وارد کنید: (اما در برنامه های بزرگ به هیچ عنوان توصیه نمیشود!)

In [ ]:

```
from <module_name> import *
```

In [1]:

```
from mod import *
```

In [2]:

```
a
```

Out[2]:

```
[100, 200, 300]
```

In [3]:

```
s
```

Out[3]:

```
'If Comrade Napoleon says it, it must be right.'
```

In [4]:

```
x = Foo()

x
```

Out[4]:

```
<mod.Foo at 0x1b16963d660>
```

در صورت نیاز می توانید نام جایگزین نیز برای ماژول یا بخشی از ماژول وارد شده تعریف نمایید:

In [ ]:

```
from <module_name> import <name> as <alt_name>[, <name> as <alt_name> …]
```

In [1]:

```
from mod import s as string, a as alist
```

In [2]:

```
s
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[2], line 1
----> 1 s

NameError: name 's' is not defined
```

In [3]:

```
string
```

Out[3]:

```
'If Comrade Napoleon says it, it must be right.'
```

In [4]:

```
a
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[4], line 1
----> 1 a

NameError: name 'a' is not defined
```

In [5]:

```
alist
```

Out[5]:

```
[100, 200, 300]
```

یا

In [ ]:

```
import <module_name> as <alt_name>
```

In [1]:

```
import mod as my_module
```

In [2]:

```
my_module.a
```

Out[2]:

```
[100, 200, 300]
```

اجرای یک ماژول بعنوان یک اسکریپت مستقل

اجرای یک ماژول به عنوان یک اسکریپت هر فایل py. که شامل یک ماژول است، در واقع همچنین یک اسکریپت پایتون است و هیچ دلیلی برای این نیست که مانند یک اسکریپت اجرا نشود.

دوباره mod.py را همانطور که در بالا تعریف شده بود، در زیر نشان می‌دهیم:

mod.py

In [ ]:

```
s = "If Comrade Napoleon says it, it must be right."
a = [100, 200, 300]

def foo(arg):
    print(f'arg = {arg}')

class Foo:
    pass
```

حالا در cmd خود می توانید فایل مربوطه را اجرا نمایید:

In [ ]:

```
C:\Users\john\Documents>python mod.py
```

هیچ خطایی وجود ندارد، بنابراین به نظر می‌رسد کار کرده است. البته، خیلی جالب نیست. همانطور که نوشته شده است، فقط شی‌ها را تعریف می‌کند. هیچ کاری با آن‌ها انجام نمی‌دهد و هیچ خروجی‌ای تولید نمی‌کند.

بیایید ماژول پایتون بالا را به گونه‌ای تغییر دهیم که هنگام اجرا به عنوان یک اسکریپت، خروجی تولید کند:

In [ ]:

```
s = "If Comrade Napoleon says it, it must be right."
a = [100, 200, 300]

def foo(arg):
    print(f'arg = {arg}')

class Foo:
    pass

print(s)
print(a)
foo('quux')
x = Foo()
print(x)
```

حالا در cmd خود می توانید فایل مربوطه را اجرا نمایید:

In [ ]:

```
C:\Users\john\Documents>python mod.py
```

با این وجود، تغییر اخیر باعث خواهد شد تا در زمان import هم دستورات داخل ماژول اجرا شوند.

In [1]:

```
import mod
```

```
If Comrade Napoleon says it, it must be right.
[100, 200, 300]
arg = quux
<mod.Foo object at 0x00000197302ACA30>
```

این احتمالاً آنچه می‌خواهید نیست. معمول نیست که یک ماژول هنگام وارد شدن، خروجی تولید کند.

آیا خوب نیست که بتوانید بین زمان بارگیری فایل به عنوان یک ماژول و زمان اجرای آن به عنوان یک اسکریپت مستقل، تفاوت قائل شوید؟

بپرسید و دریافت کنید.

زمانی که یک فایل py. به عنوان یک ماژول وارد می‌شود، پایتون متغیر خاص داندر __name__ را به نام ماژول تنظیم می‌کند. با این حال، اگر یک فایل به عنوان یک اسکریپت مستقل اجرا شود، __name__ (به صورت خلاقانه) به رشتهٔ __main__ تنظیم می‌شود. با استفاده از این واقعیت، می‌توانید در زمان اجرا تشخیص دهید کدام حالت صادق است و رفتار را به‌منظور تغییرات مناسب تغییر دهید:

In [ ]:

```
# mod.py

s = "If Comrade Napoleon says it, it must be right."
a = [100, 200, 300]

def foo(arg):
    print(f'arg = {arg}')

class Foo:
    pass

if (__name__ == '__main__'):
    print('Executing as standalone script')
    print(s)
    print(a)
    foo('quux')
    x = Foo()
    print(x)
```

حالا در cmd خود می توانید فایل مربوطه را اجرا نمایید:

In [ ]:

```
C:\Users\john\Documents>python mod.py
```

اما با این تغییر، دیگر دستورات داخل if در هنگام import اجرا نخواهند شد

In [1]:

```
import mod
```

In [2]:

```
mod.foo('grault')
```

```
arg = grault
```

برای بارگذاری مجدد یک پکیج کافی هست ان را دوباره import نمایید.

پکیج های پایتون

فرض کنید یک برنامه بسیار بزرگ توسعه داده‌اید که شامل بسیاری از ماژول‌ها است. با افزایش تعداد ماژول‌ها، اگر آن‌ها در یک مکان قرار داده شوند، سخت است که همه را پیگیری کرد. این به خصوص در صورتی است که نام‌ها یا عملکردهای مشابه داشته باشند. شما ممکن است به دنبال یک روش برای گروه‌بندی و سازماندهی آن‌ها باشید.

پکیج‌ها به شما اجازه می‌دهند تا با استفاده از نمایش نقطه‌ای، فضای نام ماژول را به صورت سلسله مراتبی سازماندهی کنید. به همان شکلی که ماژول‌ها به جلوگیری از تداخل نام متغیر جهانی کمک می‌کنند، بسته‌ها به جلوگیری از تداخل نام ماژول کمک می‌کنند.

ایجاد یک پکیج بسیار ساده است، زیرا از ساختار فایل سلسله مراتبی پایین‌سیستم عامل استفاده می‌کند. به عنوان یک نمونه، ترتیب زیر را در نظر بگیرید:

در اینجا، یک پوشه به نام pkg وجود دارد که دو ماژول mod1.py و mod2.py را شامل می‌شود. محتوای ماژول‌ها به صورت زیر است:

In [ ]:

```
# mod1.py

def foo():
    print('[mod1] foo()')

class Foo:
    pass
```

In [ ]:

```
# mod2.py

def bar():
    print('[mod2] bar()')

class Bar:
    pass
```

با توجه به این ساختار، اگر پوشه pkg در مکانی قرار داده شود که بتوان آن را پیدا کرد (در یکی از پوشه‌های موجود در sys.path)، می‌توانید با نمایش نقطه‌ای به دو ماژول با نام pkg.mod1 و pkg.mod2 ارجاع دهید و آن‌ها را با دستوری که قبلاً با آن آشنا شده‌اید، وارد کنید:

In [ ]:

```
import <module_name>[, <module_name> ...]
```

In [3]:

```
import pkg.mod1, pkg.mod2
```

In [4]:

```
pkg.mod1.foo()
```

```
[mod1] foo()
```

In [5]:

```
x = pkg.mod2.Bar()
```

In [6]:

```
x
```

Out[6]:

```
<pkg.mod2.Bar at 0x16209f2ff40>
```

In [7]:

```
from pkg.mod1 import foo
```

In [8]:

```
foo()
```

```
[mod1] foo()
```

In [9]:

```
from pkg.mod2 import Bar as Qux
```

In [10]:

```
x = Qux()
```

In [11]:

```
x
```

Out[11]:

```
<pkg.mod2.Bar at 0x16209f2ebf0>
```

Package Initialization

اگر یک فایل با نام `__init__.py` در یک پوشه پکیج وجود داشته باشد، هنگام وارد کردن بسته یا یک ماژول در پکیج فراخوانی می‌شود. این می‌تواند برای اجرای کد اولیه پکیج، مانند اولویت دادن به داده‌های سطح پکیج، استفاده شود.

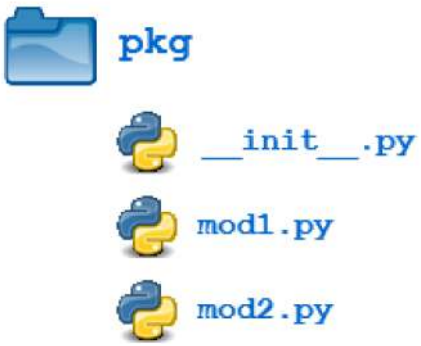
به عنوان مثال، فایل `__init__.py` زیر را در نظر بگیرید:

In [ ]:

```
## __init__.py

print(f'Invoking __init__.py for {__name__}')
A = ['quux', 'corge', 'grault']
```

بیایید این فایل را به پوشه pkg از مثال بالا اضافه کنیم:



حالا هنگام وارد کردن پکیج، لیست سراسری A مقداردهی اولیه می‌شود:

In [1]:

```
import pkg
```

```
Invoking __init__.py for pkg
```

In [2]:

```
pkg.A
```

Out[2]:

```
['quux', 'corge', 'grault']
```

یک ماژول در پکیج می‌تواند با وارد کردن آن به ترتیب به متغیر سراسری دسترسی پیدا کند:

`__init__.py` همچنین می‌تواند برای ایجاد وارد کردن خودکار ماژول‌ها از یک پکیج استفاده شود. به عنوان مثال، قبلاً دیدید که دستور import pkg فقط نام pkg را در جدول نمادین محلی caller قرار می‌دهد و هیچ ماژولی را وارد نمی‌کند. اما اگر `__init__.py` در پوشه pkg شامل موارد زیر باشد:

In [2]:

```
## __init__.py

print(f'Invoking __init__.py for {__name__}')
import pkg.mod1, pkg.mod2
```

در این صورت، هنگام اجرای import pkg، ماژول‌های mod1 و mod2 به طور خودکار وارد می‌شوند:

In [1]:

```
import pkg
```

```
Invoking __init__.py for pkg
```

In [2]:

```
pkg.mod1.foo()
```

```
[mod1] foo()
```

In [3]:

```
pkg.mod2.bar()
```

```
[mod2] bar()
```

Importing * From a Package

برای اهداف بحث زیر، پکیج تعریف شده قبلی به منظور شامل بودن برخی ماژول‌های اضافی گسترش می‌یابد:

در حال حاضر چهار ماژول در پوشه pkg تعریف شده‌اند. محتوای آن‌ها به شکل زیر است:

In [ ]:

```
## mod1.py
def foo():
    print('[mod1] foo()')

class Foo:
    pass
```

In [ ]:

```
## mod2.py

def bar():
    print('[mod2] bar()')

class Bar:
    pass
```

In [ ]:

```
## mod3.py

def car():
    print('[mod2] bar()')

class Car:
    pass
```

In [ ]:

```
## mod4.py

def door():
    print('[mod2] bar()')

class Door:
    pass
```

شما قبلاً دیده‌اید که هنگام استفاده از import * برای یک ماژول، تمام شی‌های ماژول وارد جدول نمادین محلی می‌شوند، به جز آن‌هایی که با یک زیرخط شروع می‌شوند، همانطور که همیشه:

In [1]:

```
dir()
```

...

In [2]:

```
from pkg.mod3 import *
```

In [ ]:

```
dir()
```

In [4]:

```
from pkg import *
pkg.mod1.foo()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[4], line 2
      1 from pkg import *
----> 2 pkg.mod1.foo()

NameError: name 'pkg' is not defined
```

چگونه در import * ماژول های قابل ورود را محدود نمایید؟

برای اینکار می توانیم از یک لیست با اسم خاص در داخل فایل __init__ استفاده نماییم. نام این لیست __all__ می باشد و مشخص می نماید که چه ماژول های اجازه دارند توسط در دستور import * به caller مستقیما وارد شوند.

In [ ]:

```
# pkg/__init__.py

__all__ = [
        'mod1',
        'mod2',
        'mod3',
        'mod4'
        ]
```

In [ ]:

```
dir()
```

In [2]:

```
from pkg import *
```

In [ ]:

```
dir()
```

علاوه بر این، شما می توانید داخل هر یک از ماژول های موجود نیز با استفاده از لیست __all__ توابع یا کلاس ها یا متغییرهای قابل import را مشخص نمایید

In [ ]:

```
## mod1.py

__all__ = ['foo']

def foo():
    print('[mod1] foo()')

class Foo:
    pass
```

In [ ]:

```
dir()
```

In [2]:

```
from pkg.mod1 import *
```

In [ ]:

```
dir()
```

Subpackages

بسته‌ها می‌توانند شامل زیرپکیج های تو در تو با عمق دلخواه باشند. به عنوان مثال، بیایید یک تغییر دیگر به پوشه بسته مثال اعمال کنیم:

چهار ماژول (mod1.py، mod2.py، mod3.py و mod4.py) به شکل قبلی تعریف شده‌اند. اما حالا به جای این که در پوشه pkg قرار داشته باشند، آن‌ها به دو پوشه زیرپکیج sub_pkg1 و sub_pkg2 تقسیم شده‌اند.

وارد کردن همچنان همانطور کار می‌کند که قبلاً نشان داده شده است. نحو شبیه است، اما نحو نقطه اضافی برای جداسازی نام بسته از نام زیرپکیج استفاده می‌شود:

In [1]:

```
import pkg1.sub_pkg1.mod1
```

In [2]:

```
pkg1.sub_pkg1.mod1.foo()
```

```
[mod1] foo()
```

In [3]:

```
from pkg1.sub_pkg1 import mod2
```

In [4]:

```
mod2.bar()
```

```
[mod2] bar()
```

In [6]:

```
from pkg1.sub_pkg2.mod3 import car
```

In [7]:

```
car()
```

```
[mod2] bar()
```

In [1]:

```
from pkg1.sub_pkg2.mod4 import door as grault
```

In [2]:

```
grault()
```

```
[mod2] door()
```