Módulos en Python

Un programa Python está compuesto de varios archivos de código fuente. Cada archivo de código fuente corresponde a un módulo, que agrupa código y datos de programa para reutilizar. Los módulos son por lo general son indepedientes unos de otros para que otros programas puedan reutilizar los módulos específicos que necesiten. Un modulo de forma explicita establece dependencia en otros módulos utilizando sentencias import o from.

En algunos lenguajes de programación, las variables globales pueden suministrar un 'conducto' para realizar el acoplamiento entre módulos. En Python, las variables globales no son globales para todos los módulos, sino que son atributos de un objeto de módulo explícito. De este modo, los módulos Python se comunican de forma explícita y sostenible.

Hay varias maneras de manejar código Python:

- funciones
- clases
- módulos
- paquetes

Los módulos de Python se utilizan para organizar el código Python. Por ejemplo, el código relacionado con la base de datos se coloca dentro de un módulo, código de seguridad de base de datos en un mdulo de seguridad, etc. Los scripts de Python mas pequeños pueden tener un módulo. Pero los programas más grandes se dividen en varios módulos. Los módulos se agrupan para formar paquetes.

1. Objetos de módulos

Un módulo es un objeto Python con atributos nominales arbitrarios a los que podemos vincular y hacer referencia. El nombre de un módulo es un archivo con la extensión .py. Cuando tenemos un archivo llamado ejemplo.py ese es el nombre de módulo. La variable `__name__` almacena el nombre de módulo que está siendo referenciado. El actual módulo, el modulo que esta siendo ejecutado (también llamado módulo principal) tiene un nombre especial: `__main__`. Con este nombre puede hacer referencia en el código Python.

En Python,los módulos son objetos (valores) y son gestionados como otros objetos. De este modo podemos transmitir un módulo como argumento en una llamada a función. Del mismo modo, una función puede devolver un módulo como el resultado de una llamada. Un módulo, como cualquier otro objeto, puede ser vinculado a una variable, un elemento en el contenedor o a un atributo de un objeto. Por ejemplo, el diccionario `sys.modules` contiene objetos de módulo como sus valores. Los módulos son objetos de primera clase.

Tenemos dos archivos `module1.py` y `module2.py`. El segundo módulo es el principal módulo, el cúal es ejecutado. Importa el primer módulo. Los módulos son importados como se dijo usando la palabra import.

```
"""
Un modulo vacio

"""
```

Este el archivo module1.py y en el código que sigue importamos dos módulos. Una contruido, parte de Python y un módulo module1.py.

```
#/usr/bin/python

import module1
```

```
import sys

print __name__
print module1.__name__
print sys.__name__
```

Si ejecutamos imprimimos, los nombres de los módulos en la consola.

```
 %run module2.py
__main__
module1
sys
```

El nombre del módulo, que está siendo ejecutado es siempre `'__main__'`. Otros módulos son llamados después del nombre de archivo. Los módulos se pueden importar a otros módulos usando la palabra clave import.

Cuando un módulo se importa el intérprete primero busca por un módulo incorporado con ese nombre. Si no lo encuentra, entonces busca en una lista de directorios dados por la variable `sys.path`. El `sys.path` es una lista de cadenas que especifica la ruta de búsqueda de módulos. Consiste del directorio actual de trabajo, nombres de directorios especificados en la variable de entorno `PYTHONPATH` más algunos directorios dependientes de instalación adicionales. Si no se encuentra el módulo, un ImportError se eleva.

El siguiente script imprime todos los directorios desde la variable `syspath`. El módulo `textwrap` es usado para un facilitar el formateo de los párrafos. Recuperamos una lista de directorios de la variable `sys.path` y los ordenamos.

```
#!/usr/bin/python

import sys
import textwrap

sp = sorted(sys.path)
dname = ', '.join(sp)

print textwrap.fill(dname)
```

Si ejecutamos el script, yo tengo en mi sistema

```
 %run module3.py
, /home/cesarlaraavila/Python-ACECOM-GISCIA, /usr/bin,
/usr/lib/pymodules/python2.7, /usr/lib/python2.7, /usr/lib/python2.7
/dist-packages, /usr/lib/python2.7/dist-packages/IPython/extensions,
/usr/lib/python2.7/dist-packages/PILcompat, /usr/lib/python2.7/dist-
packages/gtk-2.0, /usr/lib/python2.7/lib-dynload, /usr/lib/python2.7
/lib-old, /usr/lib/python2.7/lib-tk, /usr/lib/python2.7/plat-x86_64
-linux-gnu, /usr/local/lib/python2.7/dist-packages,
/usr/local/lib/python2.7/dist-packages/Pweave-0.21.2-py2.7.egg,
/usr/local/lib/python2.7/dist-packages/setuptools-2.0.1-py2.7.egg
```

2. La palabra import

La palabra `import` puede ser usado de varias maneras:

```
from module import *
```

Esta construccin va a importar todas las definiciones de Python en el espacio de nombres del módulo. Hay una excepción aquí, los objetos que comienzan con guin bajo _ no se importan. Se espera que sean utilizados exclusivamente de manera interna por el módulo se importa. No se recomienda esta forma de importar módulos.

```
#!/usr/bin/python

from math import *

print cos(4)
print pi
```

Aquí se ha importado todas las definiciones del módulo `math` incorporado. Podemos llamar a las funciones matemáticas directamente, sin hacer referencia al módulo. El uso de esta forma de importación puede resultar en un desorden en el espacio de nombres. Podemos tener varios objetos del mismo nombre y sus definiciones se pueden anular.

```
from math import  *

pi = 3.14
print cos(4)
print pi
```

El ejemplo muestra 3.14 en la consola. Lo que no puede ser, no debe ser así. Los lios del espacio de nombres puede llegar a ser crítica en proyectos mas grandes.

El siguiente ejemplo mostrará definiciones, que no están siendo importadas utilizando esta forma de importación

```
#!/usr/bin/python

"""
names es un modulo test

"""

_version=0.1

names = ["Python", "R", "JavaScript", "C", "Java"]

def mostrar_names():
    for i in names:
        print i

def _mostrar_version():
    print _version
```

y

```
#!/usr/bin/python

from  module4 import*

print locals()
mostrar_names()
```

Vamos a ejecutarlo y ver que pasa

```
{'mostrar_names': <function mostrar_names at 0x1fbe7d0>,
'__nonzero__': <function <lambda> at 0x1fbe758>,
'__builtins__': {'bytearray': <type 'bytearray'>,
'IndexError': <type 'exceptions.IndexError'>,
'all': <built-in function all>,
'help': Type help() for interactive help,
or help(object) for help about object.,
'vars': <built-in function vars>,
'SyntaxError': <type 'exceptions.SyntaxError'>,
'__IPYTHON__active': 'Deprecated, check for __IPYTHON__',
'unicode': <type 'unicode'>, 'UnicodeDecodeError': <type 'exceptions.UnicodeDecodeError'>,
'memoryview': <type 'memoryview'>,
'isinstance': <built-in function isinstance>,.....

.....

Python
R
JavaScript
C
Java
```

La variable _version y la función `_mostrar_version()` no se importan en el módulo. No se ven en el espacio de nombres. La funcin `locals()` nos da todas las definiciones disponibles en el módulo5.

```
from module import func1, func2...
```

Esta forma de importación dada arriba, permite construir importaciones sólo de objetos específicos de un módulo. De esta manera nos importa sólo definiciones que necesitamos. También podríamos importar definiciones que comienzan con un guión bajo. Pero esto es una mala práctica. Como mostramos a continuación

```
#!/usr/bin/python

from module4 import _version, _mostrar_version

print _version
_mostrar_version()
```

Así que un uso más o menos adecuado sería

```
#!/usr/bin/python

from math import sin, pi

print sin(3)
print pi
```

```
import module
```

La última construcción es la más utilizada. Evita el desorden del espacio de nombres y habilita acceder a todas las definiciones de un módulo.

```
#!/usr/bin/python

import math

pi = 3.14

print math.cos(3)
print math.pi
print math.sin(3)
print pi
```

En este caso, hacemos referencia a las definiciones a través del nombre del módulo. Como podemos ver, estamos en condiciones de utilizar ambas variables `pi`. Nuestra definición y la del módulo `math`. Podemos cambiar el nombre a través de los cuales podemos hacer referencia al módulo. Para ello, utilizamos la palabra clave `as`.

```
#!/usr/bin/python


import math as m

print m.pi
print m.cos(5)
```

Un ImportError es lanzado, cuando el módulo no es importado.

```
#!/usr/bin/python
try:
    import python1
except ImportError, e:
    print 'Fallo la importacion:', e
```

3. Executando módulos

Los módulos se pueden importar a otros módulos o pueden ser también ejecutados. EL creador del módulo a menudo crean un conjunto de pruebas para probar su módulo. Sólo si el módulo se ejecuta como un script, el atributo `__name__` es igual a `__main__`.

Vamos a demostrar esto con un módulo de Fibonacci (module9). El módulo puede ser importado de manera usual. El módulo puede ser ejecutado

```
%run module9.py
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Si hacemos importar el módulo de fibonacci (module9), la prueba no se ejecuta automáticamente.

```
>>> import module9 as fib
>>> fib.fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

El módulo de fibonacci (module9) es importado y la función `fib()` es executada.

4. La funcion dir()

La función `dir()` da una lista ordenada de cadenas que contiene los nombres definidos por un módulo. En el siguiente ejemplo importamos dos módulos. Definimos una variable, una lista y una función.

```
#!/usr/bin/python

"""
Este es un modulo para usar dir() built

"""

import math, sys

version = 1.0

names = ['Python', 'C', 'JavaScript', 'Java']

def mostrar_names():
    for i in names:
        print i

print dir(sys.modules['__main__'])
```

La función `dir()` devuelve todos los nombres disponibles en el espacio de nombres actual del módulo. `'__main__'` es el nombre del módulo actual. El `sys.modules` es un diccionario que mapea nombres de los módulos a los módulos que ya han sido cargados.

```
%run module10.py
['__builtins__', '__doc__', '__file__', '__name__', '__nonzero__', '__package__',
 'math', 'mostrar_names', 'names', 'sys', 'version']
```

5. La función globals()

La función `globals()` devuelve un diccionario que representa el espacio de nombres global actual. Se trata de un diccionario de nombres globales y sus valores. Es el diccionario del módulo actual. Utilizamos la funcin `globals()` para imprimir todos los nombres globales del módulo actual.

```
#!/usr/bin/python
import sys
import textwrap


def myfun():
    pass

gl = globals()
gnames = ' , '.join(gl)

print textwrap.fill(gnames)
```

Así tenemos, todos los nombres globales en el actual módulo

```
%run module11.py
__nonzero__ , __builtins__ , __file__ , textwrap , __package__ , sys ,
myfun , __name__ , gl
```

6. El atributo __module__

El atributo de clase `__module__` tiene el nombre del módulo en el cúal la clase es definida.

```
"""
modulo animalitos
"""

class Cat:
  pass


class Perro:
  pass
```

Aquí tenemos dos clases en el siguiente código y usamos `__module__`. Desde el módulo `module12A` importamos la clase `Cat` y en el actual módulo B.

```
class B:
    pass
b = B()
print b.__module__
```

Y creamos la instancia de B. Imprimimos el nombre de este módulo.

```
#!/usr/bin/python

from module12A import Cat

class B:
    pass

b = B()
print b.__module__

c =Cat()
print c.__module__
```

Nosotros también creamos un objeto desde la clase Cat(). Imprimos este módulo donde fue definido.

```
c =Cat()
print c.__module__
```

El actual nombre de módulo `__main__`. Y el nombre del módulo `Cat` es `module12A`.

```
%edit module12.py
Editing... done. Executing edited code...
__main__
module12A
```

Python también admite extensiones, que son componentes escritos de otros lenguajes, como C, C++, Java, para utilizar con Python. Estas extensiones son reconocidas como módulos por el código Python que las usa (código cliente). A estos no les importa si un módulo es totalmente Python o una extensión. Siempre podemos comenzar codificando un módulo en lenguaje Python. A continuación, si necesitamos un mejor rendimiento, podremos recodificar algunos módulos en lenguajes de nivel inferior sin cambiar el código cliente que utilizan los módulos.