Lectura 4: El tutorial de Python (Páginas 36-43)

En este material de lectura veremos qué son los módulos y paquetes. Los mismos sirven para estructurar el código. Además veremos la función dir.

Cantidad de páginas: 8

Módulos

Si salís del intérprete de Python y entrás de nuevo, las definiciones que hiciste (funciones y variables) se pierden. Por lo tanto, si querés escribir un programa más o menos largo, es mejor que uses un editor de texto para preparar la entrada para el interprete y ejecutarlo con ese archivo como entrada. Esto es conocido como crear un *guión*, o *script*. Si tu programa se vuelve más largo, quizás quieras separarlo en distintos archivos para un mantenimiento más fácil. Quizás también quieras usar una función útil que escribiste desde distintos programas sin copiar su definición a cada programa.

Para soportar esto, Python tiene una manera de poner definiciones en un archivo y usarlos en un script o en una instancia interactiva del intérprete. Tal archivo es llamado *módulo*; las definiciones de un módulo pueden ser *importadas* a otros módulos o al módulo *principal* (la colección de variables a las que tenés acceso en un script ejecutado en el nivel superior y en el modo calculadora).

Un módulo es una archivo conteniendo definiciones y declaraciones de Python. El nombre del archivo es el nombre del módulo con el sufijo `.py` agregado. Dentro de un módulo, el nombre del mismo (como una cadena) está disponible en el valor de la variable global `__name__`. Por ejemplo, usá tu editor de textos favorito para crear un archivo llamado `fibo.py` en el directorio actual, con el siguiente contenido:

```
# módulo de números Fibonacci

def fib(n):    # escribe la serie Fibonacci hasta n
    a, b = 0, 1
    while b < n:
        print(b, end=' ')
        a, b = b, a+b
     print()

def fib2(n):   # devuelve la serie Fibonacci hasta n
    resultado = []
    a, b = 0, 1
    while b < n:
        resultado.append(b)
        a, b = b, a+b
    return resultado
```

Ahora entrá al intérprete de Python e importá este módulo con la siguiente orden:

```
>>> import fibo
```

Esto no mete los nombres de las funciones definidas en `fibo` directamente en el espacio de nombres actual; sólo mete ahí el nombre del módulo, `fibo` . Usando el nombre del módulo podés acceder a las funciones:

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

Si pensás usar la función frecuentemente, podés asignarla a un nombre local:

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

Más sobre los módulos

Un módulo puede contener tanto declaraciones ejecutables como definiciones de funciones. Estas declaraciones están pensadas para inicializar el módulo. Se ejecutan solamente la *primera* vez que el módulo se encuentra en una sentencia import. [7] (Son también ejecutados si el archivo es ejecutado como un script).

Cada módulo tiene su propio espacio de nombres, el que es usado como espacio de nombres global por todas las funciones definidas en el módulo. Por lo tanto, el autor de un módulo puede usar variables globales en el módulo sin preocuparse acerca de conflictos con una variable global del usuario. Por otro lado, si sabés lo que estás haciendo podés tocar las variables globales de un módulo con la misma notación usada para referirte a sus funciones,`nombremodulo.nombreitem`.

Los módulos pueden importar otros módulos. Es costumbre pero no obligatorio el ubicar todas las declaraciones `import` al principio del módulo (o script, para el caso). Los nombres de los módulos importados se ubican en el espacio de nombres global del módulo que hace la importación.

Hay una variante de la declaración `import` que importa los nombres de un módulo directamente al espacio de nombres del módulo que hace la importación. Por ejemplo:

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

Esto no introduce en el espacio de nombres local el nombre del módulo desde el cual se está importando (entonces, en el ejemplo, `fibo` no se define).

Hay incluso una variante para importar todos los nombres que un módulo define:

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

Esto importa todos los nombres excepto aquellos que comienzan con un subrayado_(). La mayoría de las veces los programadores de Python no usan esto ya que introduce un conjunto de nombres en el intérprete, posiblemente escondiendo cosas que ya estaban definidas.

Notá que en general la práctica de importar `*` de un módulo o paquete está muy mal vista, ya que frecuentemente genera un código poco legible. Sin embargo, está bien usarlo para ahorrar tecleo en sesiones interactivas.

> ***Nota***
>
> Por razones de eficiencia, cada módulo se importa una vez por sesión del intérprete. Por lo tanto, si modificás los módulos, tenés que reiniciar el intérprete -- o, si es sólo un módulo que querés probar interactivamente, usá `imp.reload()`, por ejemplo `import importlib; importlib.reload(modulename)`.

Ejecutando módulos como scripts

Cuando ejecutás un módulo de Python con

```
python fibo.py <argumentos>
```

...el código en el módulo será ejecutado, tal como si lo hubieses importado, pero con `__name__` con el valor de `"__main__"`. Eso significa que agregando este código al final de tu módulo:

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

...podés hacer que el archivo sea utilizable tanto como script, como módulo importable, porque el código que analiza la linea de órdenes sólo se ejecuta si el módulo es ejecutado como archivo principal:

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

Si el módulo se importa, ese código no se ejecuta:

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

Esto es frecuentemente usado para proveer al módulo una interfaz de usuario conveniente, o para propósitos de prueba (ejecutar el módulo como un script ejecuta el juego de pruebas).

El camino de búsqueda de los módulos

Cuando se importa un módulo llamado **`spam`**, el intérprete busca primero por un módulo con ese nombre que esté integrado en el intérprete. Si no lo encuentra, entonces busca un archivo llamado`spam.py` en una lista de directorios especificada por la variable **`sys.path`**. **`sys.path`** se inicializa con las siguientes ubicaciones:

- el directorio conteniendo el script (o el directorio actual cuando no se especifica un archivo).
- **`PYTHONPATH`** (una lista de nombres de directorios, con la misma sintaxis que la variable de entorno **`PATH`**.
- el directorio default de la instalación.

> ***Nota***
>
> En sistemas de archivos que soportan enlaces simbólicos, el directorio conteniendo el script de entrada es calculado después de que el enlace simbólico sea seguido. En otras palabras, el directorio conteniendo el enlace simbólico **no** es agregado al módulo de busca del path.

Luego de la inicialización, los programas Python pueden modificar **`sys.path`**. El directorio que contiene el script que se está ejecutando se ubica al principio de la búsqueda, adelante de la biblioteca estándar. Esto significa que se cargarán scripts en ese directorio en lugar de módulos de la biblioteca estándar con el mismo nombre. Esto es un error a menos que se esté reemplazando intencionalmente. Mirá la sección *Módulos estándar* para más información.

Archivos "compilados" de Python

Para acelerar la carga de módulos, Python cachea las versiones compiladas de cada módulo en el directorio `__pycache__` bajo el nombre `module.version.pyc` dónde la versión codifica el formato del archivo compilado; generalmente contiene el número de version de Python. Por ejemplo, en CPython release 3.3 la version compilada de spam.py sería cacheada como `__pycache__/spam.cpython-33.pyc`. Este convensión de nombre permite compilar módulos desde diferentes releases y versiones de Python para coexistir.

Python chequea la fecha de modificación de la fuente contra la versión compilada para evr si esta es obsoleta y necesita ser recompilada. Esto es un proceso completamente automático. También, los módulos compilados son independientes de la plataforma, así que la misma librería puede ser compartida a través de sistemas con diferentes arquitecturas.

Python no chequea el caché en dos circuntancias. Primero, siempre recompila y no graba el resultado del módulo que es cargado directamente desde la línea de comando. Segundo, no chequea el caché si no hay módulo fuente.

Algunos consejos para expertos:

- Podés usar la *-O* o *-OO* en el comando de Python para reducir el tamaño de los módulos compilados. La `-O` quita assert statements, la `-OO` quita ambos, assert statements y cadenas de caracteres __doc__. Debido a que algunos programas se basan en que estos estén disponibles, deberías usar esta opción únicamente si sabés lo que estás haciendo. Los módulos "optimizados" tienen una etiqueta `opt-` y son normalmente más pequeños. Releases futuras quizás cambien los efectos de la optimización.
- Un programa no corre más rápido cuando se lee de un archivo *.pyc* que cuando se lee del *.py*; lo único que es más rápido en los archivos *.pyc* es la velocidad con que se cargan.
- Hay más detalles de este proceso, incluyendo un diagrama de flujo de la toma de decisiones, en la PEP 3147.
- El módulo **`compilleall`** puede crear archivos .pyc para todos los módulos en un directorio.

Módulos estándar

Python viene con una biblioteca de módulos estándar, descrita en un documento separado, la Referencia de la Biblioteca de Python (de aquí en más, "Referencia de la Biblioteca"). Algunos módulos se integran en el intérprete; estos proveen acceso a operaciones que no son parte del núcleo del lenguaje pero que sin embargo están integrados, tanto por eficiencia como para proveer acceso a primitivas del sistema operativo, como llamadas al sistema. El conjunto de tales módulos es una opción de configuración el cual también depende de la plataforma subyacente. Por ejemplo, el módulo **`winreg`** sólo se provee en sistemas Windows. Un módulo en particular merece algo de atención: **`sys`**, el que está integrado en todos los intérpretes de Python. Las variables `sys.ps1` y `sys.ps2` definen las cadenas usadas como cursores primarios y secundarios:

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

Estas dos variables están solamente definidas si el intérprete está en modo interactivo.

La variable `sys.path` es una lista de cadenas que determinan el camino de búsqueda del intérprete para los módulos. Se inicializa por omisión a un camino tomado de la variable de entorno **`PYTHONPATH`**, o a un valor predefinido en el intérprete si **`PYTHONPATH`** no está configurada. Lo podés modificar usando las operaciones estándar de listas:

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

La función dir

La función integrada **`dir()`** se usa para encontrar qué nombres define un módulo. Devuelve una lista ordenada de cadenas:

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)  # doctest: +NORMALIZE_WHITESPACE
['__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', 'gettotalrefcount',
 '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', 'settrace', 'stderr', 'stdin', 'stdout',
 'thread_info', 'version', 'version_info', 'warnoptions']
```

Sin argumentos, `dir()` lista los nombres que tenés actualmente definidos:

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

Notá que lista todos los tipos de nombres: variables, módulos, funciones, etc.

`dir()` no lista los nombres de las funciones y variables integradas. Si querés una lista de esos, están definidos en el módulo estándar `builtins`:

```
>>> import builtins
>>> dir(builtins)  # doctest: +NORMALIZE_WHITESPACE
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning',
 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError',
 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning',
 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False',
 'FileExistsError', 'FileNotFoundError', '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']
```

Paquetes

Los paquetes son una manera de estructurar los espacios de nombres de Python usando "nombres de módulos con puntos". Por ejemplo, el nombre de módulo `A.B` designa un submódulo llamado `B` en un paquete llamado `A`. Tal como el uso de módulos evita que los autores de diferentes módulos tengan que preocuparse de los respectivos nombres de variables globales, el uso de nombres de módulos con puntos evita que los autores de paquetes de muchos módulos, como NumPy o la Biblioteca de Imágenes de Python (Python Imaging Library, o PIL), tengan que preocuparse de los respectivos nombres de módulos.

Suponete que querés designar una colección de módulos (un "paquete") para el manejo uniforme de archivos y datos de sonidos. Hay diferentes formatos de archivos de sonido (normalmente reconocidos por su extensión, por ejemplo: .wav, .aiff, .au), por lo que tenés que crear y mantener una colección siempre creciente de módulos para la conversión entre los distintos formatos de archivos. Hay muchas operaciones diferentes que quizás quieras ejecutar en los datos de sonido (como mezclarlos, añadir eco, aplicar una función ecualizadora, crear un efecto estéreo artificial), por lo que ademas estarás escribiendo una lista sin fin de módulos para realizar estas operaciones. Aquí hay una posible estructura para tu paquete (expresados en términos de un sistema jerárquico de archivos):

```
sound/                          Paquete superior
      __init__.py               Inicializa el paquete de sonido
      formats/                  Subpaquete para conversiones de formato
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Subpaquete para efectos de sonido
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Subpaquete para filtros
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

Al importar el paquete, Python busca a través de los directorios en `sys.path`, buscando el subdirectorio del paquete.

Los archivos `__init__.py` se necesitan para hacer que Python trate los directorios como que contienen paquetes; esto se hace para prevenir directorios con un nombre común, como `string`, de esconder sin intención a módulos válidos que se suceden luego en el camino de búsqueda de módulos. En el caso más simple, `__init__.py` puede ser solamente un archivo vacío, pero también puede ejecutar código de inicialización para el paquete o configurar la variable `__all__`, descrita luego.

Los usuarios del paquete pueden importar módulos individuales del mismo, por ejemplo:

```
import sound.effects.echo
```

Esto carga el submódulo `sound.effects.echo`. Debe hacerse referencia al mismo con el nombre completo.

```
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

Otra alternativa para importar el submódulos es:

```
from sound.effects import echo
```

Esto también carga el submódulo **`echo`**, lo deja disponible sin su prefijo de paquete, por lo que puede usarse así:

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

Otra variación más es importar la función o variable deseadas directamente:

```
from sound.effects.echo import echofilter
```

De nuevo, esto carga el submódulo **`echo`**, pero deja directamente disponible a la función **`echofilter()`** :

```
echofilter(input, output, delay=0.7, atten=4)
```

Notá que al usar `from package import item` el ítem puede ser tanto un submódulo (o subpaquete) del paquete, o algún otro nombre definido en el paquete, como una función, clase, o variable. La declaración `import` primero verifica si el ítem está definido en el paquete; si no, asume que es un módulo y trata de cargarlo. Si no lo puede encontrar, se genera una excepción **`ImportError`**.

Por otro lado, cuando se usa la sintaxis como `import item.subitem.subsubitem`, cada ítem excepto el último debe ser un paquete; el mismo puede ser un módulo o un paquete pero no puede ser una clase, función o variable definida en el ítem previo.

Importando * desde un paquete

Ahora, ¿qué sucede cuando el usuario escribe `from sound.effects import *` ? Idealmente, uno esperaría que esto de alguna manera vaya al sistema de archivos, encuentre cuales submódulos están presentes en el paquete, y los importe a todos. Esto puede tardar mucho y el importar sub-módulos puede tener efectos secundarios no deseados que sólo deberían ocurrir cuando se importe explícitamente el sub-módulo.

La única solución es que el autor del paquete provea un índice explícito del paquete. La declaración **`import`** usa la siguiente convención: si el código del `__init__.py` de un paquete define una lista llamada `__all__`, se toma como la lista de los nombres de módulos que deberían ser importados cuando se hace `from package import *` . Es tarea del autor del paquete mantener actualizada esta lista cuando se libera una nueva versión del paquete. Los autores de paquetes podrían decidir no soportarlo, si no ven un uso para importar * en sus paquetes. Por ejemplo, el archivo `sound/effects/__init__.py` podría contener el siguiente código:

```
__all__ = ["echo", "surround", "reverse"]
```

Esto significaría que `from sound.effects import *` importaría esos tres submódulos del paquete **`sound`**.

Si no se define `__all__`, la declaración `from sound.effects import *` *no* importa todos los submódulos del paquete **`sound.effects`** al espacio de nombres actual; sólo se asegura que se haya importado el paquete **`sound.effects`** (posiblemente ejecutando algún código de inicialización que haya en `__init__.py`) y luego importa aquellos nombres que estén definidos en el paquete. Esto incluye cualquier nombre definido (y submódulos explícitamente cargados) por `__init__.py`. También incluye cualquier submódulo del paquete que pudiera haber sido explícitamente cargado por declaraciones **`import`** previas. Considerá este código:

```
import sound.effects.echo
import sound.effects.surround
from sound.effects import *
```

En este ejemplo, los módulos *echo* y *surround* se importan en el espacio de nombre actual porque están definidos en el paquete **`sound.effects`** cuando se ejecuta la declaración `from...import`. (Esto también funciona cuando se define `__all__`).

A pesar de que ciertos módulos están diseñados para exportar solo nombres que siguen ciertos patrones cuando usás `import *`, también se considera una mala práctica en código de producción.

Recordá que no está mal usar `from paquete import submodulo_especifico`! De hecho, esta notación se recomienda a menos que el módulo que estás importando necesite usar submódulos con el mismo nombre desde otros paquetes.

Referencias internas en paquetes

Cuando se estructuran los paquetes en subpaquetes (como en el ejemplo `sound`), podés usar `import` absolutos para referirte a submódulos de paquetes hermanos. Por ejemplo, si el módulo `sound.filters.vocoder` necesita usar el módulo `echo` en el paquete `sound.effects` , puede hacer `from sound.effects import echo` .

También podés escribir `import` relativos con la forma `from module import name`. Estos imports usan puntos adelante para indicar los paquetes actual o padres involucrados en el import relativo. En el ejemplo `surround`, podrías hacer:

```
from . import echo
from .. import formats
from ..filters import equalizer
```

Notá que los imports relativos se basan en el nombre del módulo actual. Ya que el nombre del módulo principal es siempre `"__main__"`, los módulos pensados para usarse como módulo principal de una aplicación Python siempre deberían usar `import` absolutos.

Paquetes en múltiples directorios

Los paquetes soportan un atributo especial más, `__path__`. Este se inicializa, antes de que el código en ese archivo se ejecute, a una lista que contiene el nombre del directorio donde está el paquete. Esta variable puede modificarse, afectando búsquedas futuras de módulos y subpaquetes contenidos en el paquete.

Aunque esta característica no se necesita frecuentemente, puede usarse para extender el conjunto de módulos que se encuentran en el paquete.

---

7 De hecho las definiciones de función son también 'declaraciones' que se 'ejecutan'; la ejecución de una definición de función a nivel de módulo mete el nombre de la función en el espacio de nombres global.