**Notas de Python**

**Índice**.

1. [**Módulos**.](#Modulos)
   1. [**Más sobre módulos**.](#Mas_sobre_modulos)
      1. Ejecutando módulos como scripts.
      2. La ruta de búsqueda del módulo.
      3. “Compilado” archivos de Python.
   2. [**Módulos standard**.](#Modulos_Standard)
   3. [**La función** dir()](#La_funcion_dir)
   4. [**Paquetes**.](#Paquetes)
      1. Importando \* From a Package.
      2. Referencia Intra-package.
      3. Paquetes en múltiples directorios.

**Módulos**

Si sale del intérprete de Python y vuelve a ingresar, las definiciones que usted realizo (funciones y variables) se pierden. Por lo tanto, si usted quiere escribir un programa algo más largo, es mejor utilizar un editor de texto para preparar la entrada para el intérprete y ejecutarlo con ese archivo como entrada. Esto se conoce como crear un script. A medida que su programa crece, puede dividirlo en varios archivos para facilitar el mantenimiento. También puede usar una función práctica que haya escrito en varios programas sin copiar su definición en cada programa.

Para respaldar esto, Python tiene una forma de poner definiciones en un archivo y usarlas en un script o en una instancia interactiva del intérprete.

Un módulo es un archivo que contiene definiciones y sentencias de Python. El nombre del archivo es el nombre del módulo con el sufijo .py adjunto. Dentro de un módulo, el nombre del módulo (como una cadena) está disponible como el valor de la variable global \_\_name\_\_. Por ejemplo use su editor de texto favorito para crear un archivo llamado fibo.py en el directorio actual con los siguientes contenidos.

# 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

Ahora ingrese al intérprete de Python e importe esté modulo con el siguiente comando:

>>> import fibo

Esto no ingresa los nombres de las funciones definidas en fibo directamente en la tabla del símbolo actual; solo ingresa el nombre del módulo fibo allí. Usando el nombre del módulo puede acceder a las funciones.

>>> 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'

Si tiene la intención de utilizar una función a menudo, puede asignarla a un nombre local:

>>> fib = fibo.fib

>>> fib(500)

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377

**Más sobre módulos**

Un módulo puede tener sentencias ejecutables, así como definiciones de funciones. Estas sentencias están destinadas a inicializar el módulo. Se ejecutan solo la primera vez que se encuentra el nombre del módulo en una declaración de importación. [[1]](#Pie_de_pagina_1) (También se ejecutan si el archivo se ejecuta como un script).

Cada módulo tiene su propia tabla de símbolos privada, que se utiliza como la tabla de símbolos global para todas las funciones definidas en el módulo. Por lo tanto el autor de un módulo puede usar variables globales en el módulo sin preocuparse por los choques accidentales con las variables globales de un usuario. Por otro lado, si sabe lo que está haciendo, puede tocar las variables globales de un módulo con la misma notación utilizada para referirse a sus funciones, nombremodulo.nombreitem.

Los módulos pueden importar otros módulos. Es habitual, pero no obligatorio, colocar las declaraciones [import](https://docs.python.org/3/reference/simple_stmts.html#import) al comienzo de un módulo (o script, para el caso). Los nombres de los módulos importados se colocan en la tabla de símbolos globales del módulo de importación.

Existe una variante de la declaración [import](https://docs.python.org/3/reference/simple_stmts.html#import) que importa los nombres de un módulo directamente en la tabla de símbolos del módulo de importación. Por ejemplo:

>>> from fibo import fib, fib2

>>> fib(500)

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377

Esto no importa el nombre del módulo desde el que se toman las importaciones en la tabla de símbolos local (por lo que en el ejemplo, fibo no está definido).

Incluso hay un variante para importar todos los nombres que define un módulo:

>>> from fibo import \*

>>> fib(500)

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377

Esto importa todos los nombres, excepto los que comiencen con un guion bajo (\_). En la mayoría de los casos, los programadores de Python no utilizan esta función ya que introduce un conjunto desconocido de nombres en el intérprete, posiblemente ocultando algunas cosas que ya ha definido.

Tenga en cuenta que, en general, la práctica de importar \* desde un módulo o paquete está mal visto, ya que a menudo causa un código poco legible. Sin embargo, está bien usarlo para guardar la escritura en sesiones interactivas.

Si el nombre del módulo es seguido por un [as](https://docs.python.org/3/reference/compound_stmts.html#as), entonces el nombre siguiente al [as](https://docs.python.org/3/reference/compound_stmts.html#as) se enlazará directamente al módulo importado.

>>> import fibo as fib

>>> fib.fib(500)

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377

Esto efectivamente está importando el módulo de la misma manera que import fibo lo hace, con la única diferencia de que está disponible como fib.

También se puede usar cuando se utiliza [from](https://docs.python.org/3/reference/simple_stmts.html#from) con efectos similares:

>>> from fibo import fib as fibonacci

>>> fibonacci(500)

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377

**Nota:** Por razones de eficiencia, cada módulo solo se importa una vez por sesión de intérprete. Por lo tanto, si cambia sus módulos, debe reiniciar el interprete – o si es solo un módulo que desea probar de forma interactiva, use [importlib.reload()](https://docs.python.org/3/library/importlib.html#importlib.reload), ejemplo, import importlib; importlib.reload(nombredelmodulo).

**Ejecutando módulos como scripts**

Cuando ejecuta un módulo de Python con

python fibo.py <arguments>

El código en el módulo se ejecutará, como si lo importara, pero con el \_\_name\_\_ establecido en “\_\_main\_\_”. Eso significa que al agregar este código al final de su módulo

if \_\_name\_\_ == "\_\_main\_\_":

import sys

fib(int(sys.argv[1]))

Puede hacer que el archivo sea utilizable como un script, así como un módulo importable, porque el código que analiza la línea de comandos se ejecuta si el módulo se ejecuta como el archivo “main”

$ python fibo.py 50

0 1 1 2 3 5 8 13 21 34

Si el módulo es importado, el código no se ejecuta:

>>> import fibo

>>>

Esto se usa a menudo para proporcionar una interfaz de usuario conveniente para un módulo o para propósitos de prueba (la ejecución del módulo como un script ejecuta un conjunto de pruebas).

**La ruta de búsqueda del módulo**

Cuando un módulo llamado spam es importado, el intérprete primero busca un módulo incorporado con ese nombre. Si no se encuentra, luego busca un archivo llamado spam.py en una lista de directorios que ofrece la variable [sys.path](https://docs.python.org/3/library/sys.html#sys.path). [sys.path](https://docs.python.org/3/library/sys.html#sys.path) se inicializa desde estas ubicaciones.

* El directorio que contiene el script de entrada (o el directorio actual cuando no se especifica ningún archivo).
* [PYTHONPATH](https://docs.python.org/3/using/cmdline.html#envvar-PYTHONPATH) (una lista de nombres de directorio, con la misma sintaxis que la variable de Shell PATH).
* El predeterminado dependiente de la instalación.

**Nota:** En los sistemas de archivos que admiten enlaces simbólicos, el directorio que contiene el script de entrada se calcular después de seguir el enlace simbólico. En otras palabras, el directorio que contiene el enlace simbólico **no** se agrega a la ruta de búsqueda del módulo.

Después de la inicialización, los programas de Python pueden modificar sys.path. El directorio que contiene el script que se está ejecutando se ubica al comienzo de la ruta de búsqueda, antes de la ruta standard de la biblioteca. Esto significa que los scripts en este directorio se cargarán en lugar de los módulos del mismo nombre en el directorio de la biblioteca. Esto es un error a menos que el reemplazo esté destinado. Consulte la sección Standard Modules para más información.

**Archivos de Python “Compilados”**

Para acelerar la carga de módulos, Python almacena en caché la versión compilada de cada módulo en el directorio \_\_pycache\_\_ bajo el nombre module.version.pcy, donde la versión codifica el formato del archivo compilado; generalmente contiene el número de versión de Python. Por ejemplo, en la versión 3.3 de CPython la versión compilada de spam.py se almacenaría en caché como \_\_pyche\_\_/spam.cpython-33.pyc. Esta conversión de nomenclatura permite que los módulos compilados de diferentes lanzamientos y diferentes versiones de Python coexistan.

Python compara la fecha de modificación de la fuente con la versión compilada para ver si está desactualizada y necesita ser recompilada. Esto es un proceso completamente automático. Además, los módulos compilados son independientes de la plataforma, por lo que la misma biblioteca se puede compartir entre sistemas con diferentes arquitecturas.

Pyhton no comprueba el caché en dos circunstancias. Primero, siempre recompila y no almacena el resultado para el módulo que se carga directamente desde la línea de comando. En segundo lugar, no comprueba el caché si no hay un módulo fuente.

Algunos consejos para expertos:

* Usted puede usar los modificadores [-o](https://docs.python.org/3/using/cmdline.html#cmdoption-o) ó [-oo](https://docs.python.org/3/using/cmdline.html#cmdoption-oo) en el comando Python para reducir el tamaño de un módulo compilado. El modificador -o elimina declaraciones assert, el modificador -oo elimina declaraciones assert y cadenas de texto \_\_doc\_\_. Dado que algunos programas pueden depender de tenerlos disponibles, solo debe usar esta opción si sabe lo que está haciendo. Los módulos “optimizados” tienen una etiqueta opt- y suelen ser más pequeños. Las futuras versiones pueden cambiar los efectos de la optimización.
* Un programa no se ejecuta más rápido cuando se lee de un archivo .pyc que cuando se lee de un archivo .py; lo único que es más rápido acerca de los archivos .pyc es la velocidad con la que se cargan.
* El módulo [compileall](https://docs.python.org/3/library/compileall.html#module-compileall) puede crear archivos .pyc para todos los módulos en un directorio.
* Hay más detalles sobre este proceso, incluido un diagrama de flujo de las decisiones, en [**PEP 3147**](https://www.python.org/dev/peps/pep-3147/).

**Módulos Standard**

Python viene con una biblioteca de módulos standard, que se describe en un documento separado, la Python Library References (en adelante “Library References”). Algunos modulo están integrados en el intérprete; estos proporcionan acceso a operaciones que no forman parte del núcleo del lenguaje pero que, sin embargo, están integradas, ya sea por eficiencia o para proporcionar acceso a primitivas del sistema operativo como las llamadas al sistema. El conjunto de dichos módulos es una opción de configuración que también depende de la plataforma subyacente. Por ejemplo, el módulo [winreg](https://docs.python.org/3/library/winreg.html#module-winreg) solo se proporciona en sistemas Windows. Un módulo en particular merece algo de atención: [sys](https://docs.python.org/3/library/sys.html#module-sys), que está integrado en cada intérprete de Python. Las variables sys.ps1 y sis.ps2 definen las cadenas utilizadas como indicaciones primarias y secundarias:

>>> import sys

>>> sys.ps1

'>>> '

>>> sys.ps2

'... '

>>> sys.ps1 = 'C> '

C> print('Yuck!')

Yuck!

C>

Estas dos variables solo se definen si el intérprete esta en modo interactivo.

La variable sys.path es una lista de cadenas que determina la ruta de búsqueda de los módulos para el intérprete. Se inicializa en una ruta predeterminada tomada de la variable de entorno [PYTHONPATH](https://docs.python.org/3/using/cmdline.html#envvar-PYTHONPATH), o de un valor predeterminado incorporado si [PYTHONPATH](https://docs.python.org/3/using/cmdline.html#envvar-PYTHONPATH) no está configurado. Puedes modificarlo usando las operaciones de lista estándar:

>>> import sys

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

**La función** [dir()](https://docs.python.org/3/library/functions.html#dir)

La función incorporada [dir()](https://docs.python.org/3/library/functions.html#dir) se usa para averiguar que nombres define un módulo. Devuelve una lista ordenada de cadenas:

>>> 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', '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']

La función [dir()](https://docs.python.org/3/library/functions.html#dir) sin argumentos, lista los nombres que ha definido actualmente:

>>> a = [1, 2, 3, 4, 5]

>>> import fibo

>>> fib = fibo.fib

>>> dir()

['\_\_builtins\_\_', '\_\_name\_\_', 'a', 'fib', 'fibo', 'sys']

Tenga en cuenta que lista todos los tipos de nombres: variables, módulos, funciones, etc.

[dir()](https://docs.python.org/3/library/functions.html#dir) no lista los nombres de funciones y variables incorporadas. Si desea una lista de estos, se definen en el módulo estándar [builtins](https://docs.python.org/3/library/builtins.html#module-builtins):

>>> import builtins

>>> dir(builtins)

['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 forma de estructurar el espacio de nombres de los módulos de Python utilizando “nombres de módulos punteados”. Por ejemplo, el nombre del módulo A.B designa un submódulo llamado B en un paquete llamado A. Al igual que el uso de módulos, evita que los autores de diferentes módulos tengan que preocuparse por los nombres de las variables globales de los demás, el uso de los nombres de los módulos de punto evita que los autores de paquetes de múltiples módulos como NumPy o Pillow tengan que preocuparse por los nombres de los módulos de los demás.

Suponga que desea diseñar una colección de módulos (un “paquete”) para el manejo uniforme de archivos de sonido y datos de sonido. Hay muchos formatos de archivo diferente (generalmente reconocidos por su extensión, por ejemplo .wav, .aiff, .au), por lo que es posible que deba crear y mantener una colección creciente de módulos para la conversión entre los distintos formatos de archivo. También es posible que desee realizar muchas operaciones diferentes en los datos de sonido (como mezclar, agregar eco, aplicar una función de ecualizador, crear un efecto estéreo artificial), por lo que además escribirá un flujo interminable de módulos para realizar estas operaciones. Aquí hay una posible estructura para su paquete (expresada en términos de un sistema de archivos jerárquico):

sound/ Top-level package

\_\_init\_\_.py Initialize the sound package

formats/ Subpackage for file format conversions

\_\_init\_\_.py

wavread.py

wavwrite.py

aiffread.py

aiffwrite.py

auread.py

auwrite.py

...

effects/ Subpackage for sound effects

\_\_init\_\_.py

echo.py

surround.py

reverse.py

...

filters/ Subpackage for filters

\_\_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 son necesarios para hacer que Python trate los directorios como paquetes que contienen, esto se hace para evitar que los directorios con un nombre común, como string, oculten involuntariamente los módulos válidos que aparecen más adelante en la ruta de búsqueda del módulo. En el caso más simple, \_\_init\_\_.py solo puede ser un archivo vacío, pero también puede ejecutar el código de inicialización del paquete o establecer la variable \_\_all\_\_, que se describe más adelante.

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

import sound.effects.echo

Esto cargara el submódulo sound.effects.echo. Debe ser referenciado con su nombre completo.

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

Una forma alternativa de importar el submódulo es:

from sound.effects import echo

Esto también carga el submódulo echo, y lo hace disponible sin su prefijo de paquete, por lo que puede usarse de la siguiente manera:

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

Otra variación es importar la función o variables deseadas directamente:

from sound.effects.echo import echofilter

De nuevo, esto cargara el submódulo echo, pero hace que su función echofilter() esté disponible directamente:

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

Tenga en cuenta que al usar from package import ítem, el elemento puede ser un submódulo (o subpaquete) del paquete, o algún otro nombre definido en el paquete, como una función, clase o variable. La declaración import primero prueba si el ítem está definido en el paquete; si no, asume que es un módulo e intenta cargarlo. Si no puede encontrarlo, se genera una excepción [ImportError](https://docs.python.org/3/library/exceptions.html#ImportError).

Por el contrario, cuando se usa una sintaxis como import item.subitem.subsubitem, cada elemento, excepto el último, debe ser un paquete; el último elemento puede ser un módulo o un paquete, pero no puede ser una clase o función o variable definida en el elemento anterior.

**Importando** \* From a Package

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 qué submódulos están presentes en el paquete y los importe todos. Esto puede llevar mucho tiempo y la importación de submódulos puede tener efectos secundarios no deseados que solo deberían ocurrir cuando el submódulo se importa explícitamente.

La única solución es que el autor del paquete proporcione un índice explícito del paquete. La declaración [import](https://docs.python.org/3/reference/simple_stmts.html#import) usa la siguiente convención: si el código \_\_init\_\_.py de un paquete define una lista llamada \_\_all\_\_, se toma como la lista de nombres de módulos que deben importarse cuando from package import \* es encontrado. Es responsabilidad del autor del paquete mantener esta lista actualizada cuando se lance una nueva versión del paquete. Los autores de paquetes también pueden decidir no admitirlo, si no ven un uso para importar \* from de su paquete. Por ejemplo, el archivo sound/effects/\_\_init\_\_.py podría contener el siguiente código:

\_\_all\_\_ = ["echo", "surround", "reverse"]

Esto significa que from sound.effects import \* importaría los tres submódulos nombrados del paquete sound.

Si \_\_all\_\_ no está definido, la declaración from sound.effects import \* no importa todos los submódulos del paquete sound.effects al espacio de nombres actual; solo garantiza que el paquete sound.effects haya sido importado (posiblemente ejecutando cualquier código de inicialización en \_\_init\_\_.py) y luego importa los nombres que estén definidos en el paquete. Esto incluye cualquier nombre definido (y submódulos cargados explícitamente) por \_\_init\_\_.py. También incluye cualquier submódulo del paquete que fue cargado explícitamente por las declaraciones [import](https://docs.python.org/3/reference/simple_stmts.html#import) anteriores. Considere 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 nombres actual porque se definen en el paquete sound.effects cuando la declaración from … import es ejecutada. (Esto también funciona cuando \_\_all\_\_ es definido.)

Aunque ciertos módulos están diseñados para exportar solo nombres que siguen ciertos patrones cuando se usa import \*, todavía se considera una mala práctica en el código de producción.

Recuerde, no hay nada malo en usar from Package import modulo\_especifico! De hecho, esta es la notación recomendada a menos que el módulo de importación necesite usar submódulos con el mismo nombre de diferentes paquetes.

**Referencia Intra-package**

Cuando los paquetes están estructurados en subpaquetes (como con el paquete sound en el ejemplo), puede usar importaciones absolutas para referirse a submódulos de paquetes hermanos. Por ejemplo, si el módulo sound.filters.vocoder necesita usar el módulo echo en el paquete sound.effects, puede usar from sound.effects import echo.

También puede escribir importaciones relativas, con la forma from modulo import nombre de la declaración import. Estas importaciones utilizan puntos iniciales para indicar los paquetes actuales y principales involucrados en la importación relativa. Desde el módulo surround por ejemplo, puede usar:

from . import echo

from .. import formats

from ..filters import equalizer

Tenga en cuenta que las importaciones relativas se basan en el nombre del módulo actual. Dado que el nombre del módulo principal siempre es “\_\_main\_\_”, los módulos destinados a ser utilizados como el módulo principal de una aplicación de Python siempre deben ser importaciones absolutas.

**Paquetes en múltiples directorios**

Los paquetes soportan un atributo especial más, [\_\_path\_\_](https://docs.python.org/3/reference/import.html#__path__). Este es inicializado para ser una lista que contiene el nombre del directorio que contiene el paquete \_\_init\_\_.py antes de que se ejecute el código en ese archivo. Esta variable puede ser modificada; hacerlo afecta a futuras búsquedas de módulos y subpaquetes contenidos en el paquete.

Si bien esta función no suele ser necesaria, se puede usar para ampliar el conjunto de módulos que se encuentran en un paquete.

**Notas de pie de página.**

[1] De hecho, las definiciones de funciones también son ‘sentencias’ que son ‘ejecutadas’; la ejecución de una definición de función de nivel de módulo ingresa el nombre de la función en la tabla de símbolos globales del módulo.

***Fuente:***[*https://docs.python.org/3/tutorial/modules.html#modules*](https://docs.python.org/3/tutorial/modules.html#modules)

*Septiembre de 2018*