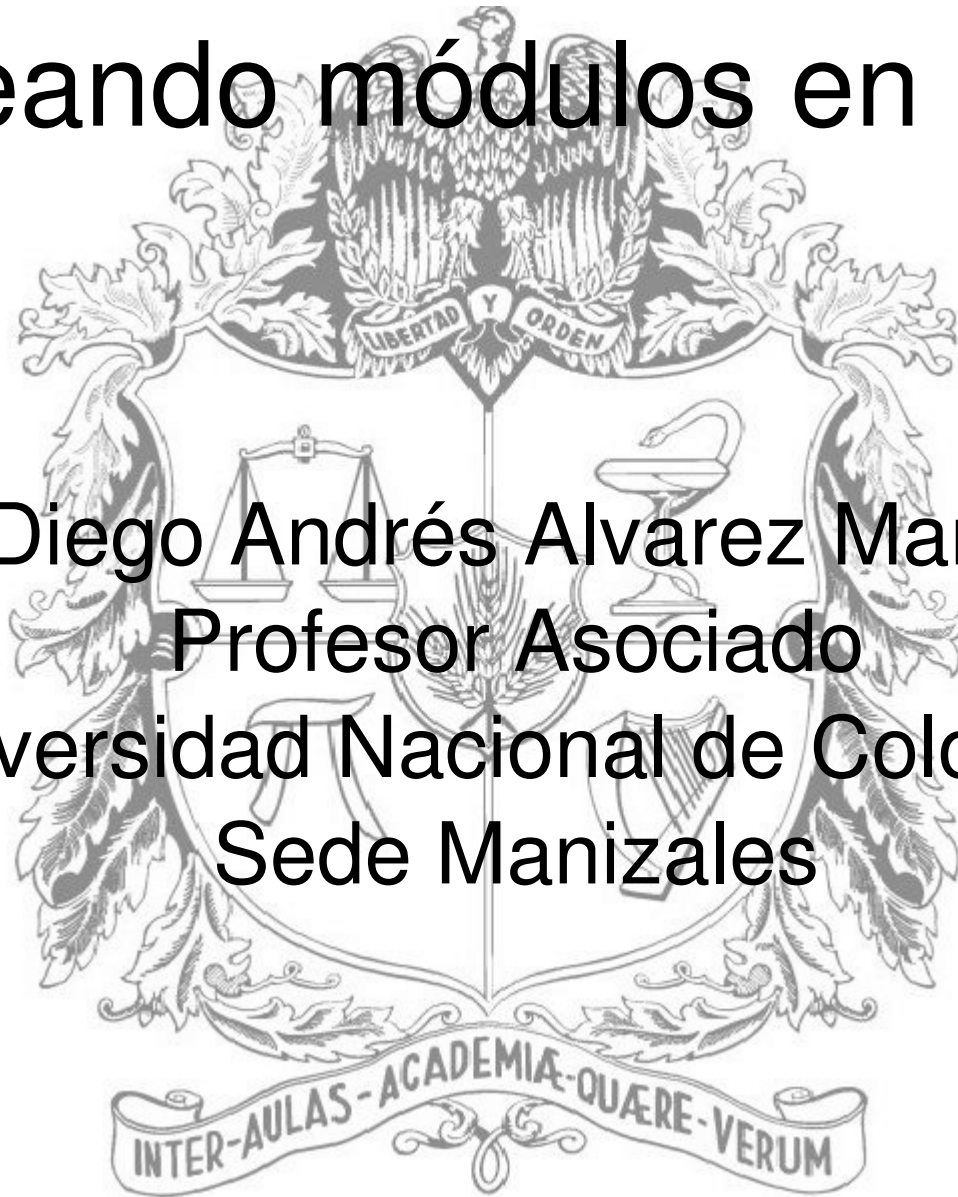


10 – Creando módulos en Python 3

Diego Andrés Álvarez Marín
Profesor Asociado
Universidad Nacional de Colombia
Sede Manizales



¿Qué es un módulo?

[http://es.wikipedia.org/wiki/Biblioteca_\(informática\)](http://es.wikipedia.org/wiki/Biblioteca_(informática))

Si bien Python incluye muchas funciones por defecto (built-in functions), algunas funciones existen en programas separados llamados módulos. Los módulos son programas de Python que contienen funciones adicionales y se llaman utilizando el comando **import**.

Es útil crear módulos para evitar tener que repetir el código común a ciertos programas; esto mejora la mantenibilidad del código.

```

1  # Módulo para el cálculo de las series de Fibonacci
2
3  def fib(n):
4      # escribe en pantalla la serie de Fibonacci hasta en número n
5      a, b = 0, 1
6      while b < n:
7          print(b, end=' ')
8          a, b = b, a+b
9      print()
10
11 def fib2(n): # return Fibonacci series up to n
12     # retorna una lista con la serie de Fibonacci hasta en número
13     result = []
14     a, b = 0, 1
15     while b < n:
16         result.append(b)
17         a, b = b, a+b
18     return result
19
20 print("Inicializando Fibonacci ...")

```

Archivo: **fibonacci.py** →

El nombre del módulo es el mismo nombre del archivo (sin la extensión .py).

Line: 22 of 22 Col: 1 LINE INS fibonacci.py UTF-8

```

>>> import fibonacci
Inicializando Fibonacci ...
>>> fibonacci.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibonacci.fib2(1000)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987]
>>> f = fibonacci.fib2
>>> f(1000)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987]
>>> fibonacci.__name__
'fibonacci'

```

→ **nombre del módulo**

Identificando el nombre del módulo dentro de un módulo

El nombre del módulo es el mismo nombre del archivo sin la extensión .py. Dentro de un módulo, el nombre del módulo está disponible como una cadena como el valor de la variable global `__name__`. En el programa principal, el nombre del módulo es `__main__`

```
>>> import fibonacci
Iniciando Fibonacci ...
>>> fibonacci.__name__
'fibonacci'
>>> __name__
'__main__'
>>> |
```

Un módulo puede contener definiciones de variables, funciones y sentencias ejecutables (es decir un "programa principal"). Esas sentencias ejecutables se utilizan para inicializar el módulo. Dichas sentencias se ejecutan sólo la primera vez que el módulo se importe con "**import**" (También se ejecutan si el archivo se ejecuta como un script.)

Cada módulo tiene su propia tabla de símbolos privada, que se utiliza como si fuera la tabla de símbolos global por todas las funciones definidas en el módulo. Así pues, el autor de un módulo puede utilizar variables globales en el módulo sin tener que preocuparse porque de pronto el usuario de dicho módulo llame a sus variables globales de su programa de la misma forma que se ha hecho en el módulo. De otro lado, si usted ha leído la documentación del módulo cuidadosamente, está facultado para invocar a las variables globales de un módulo con la misma notación que se utiliza para referirse a sus funciones: "nombre_modulo.identificador_en_modulo".

Los módulos pueden importar otros módulos. Se acostumbra (pero no se requiere) colocar todas las declaraciones de importación al comienzo de un módulo (o script). Los nombres de los módulos importados se colocan en la tabla de símbolos global del módulo de importación.

Existe una variante del comando `import` que importa los identificadores de un módulo directamente, a la tabla de símbolos del módulo donde está el `import`, sin importar toda la tabla de símbolos del módulo. Por ejemplo:

```
>>> from fibonacci import fib, fib2
Iniciizando Fibonacci ...
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
>>> fibonacci
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    fibonacci
NameError: name 'fibonacci' is not defined
>>> |
```

Observe que en este ejemplo, el módulo como tal (es decir, la tabla de símbolos del módulo) no fue importado; solo se importaron las funciones referidas a la tabla de símbolos global.

Utilizando un comando similar a:

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

se pueden importar todas los identificadores (variables, funciones, etc) definidas dentro del módulo, excepto aquellos cuyo nombre empieza con un guión bajo "_" (esto no sucede con `import fibonacci`). Es una mala práctica de programación ejecutar este comando, ya que puede introducir identificadores desconocidos en el código (lo cual puede dificultar su lectura); incluso este comando puede reescribir funciones o variables que fueron previamente definidas.

Este comando se utiliza generalmente en la consola, ya que en esta se quiere ensayar con rapidez o facilidad ciertos comandos de diferentes módulos.

El espacio de nombres (Namespace)

Un espacio de nombres es un conjunto de nombres en el cual todos los nombres son únicos; este es un contenedor abstracto en el que un grupo de uno o más identificadores únicos pueden existir. Un identificador definido en un espacio de nombres está asociado con ese espacio de nombres. El mismo identificador puede independientemente ser definido en múltiples espacios de nombres, eso es, el sentido asociado con un identificador definido en un espacio de nombres es independiente del mismo identificador declarado en otro espacio de nombres.

El espacio de nombres (Namespace)

Por ejemplo, Pedro trabaja para la compañía X y su número de empleado es 123. María trabaja para la compañía Y y su número de empleada también es 123. La razón por la cual Pedro y María pueden ser identificados con el mismo número de empleado es porque trabajan para compañías diferentes. Diferentes compañías simbolizan en este caso diferentes espacios de nombres.

En programas grandes o en documentos no es infrecuente tener cientos o miles de identificadores. Los espacios de nombres (o técnicas similares como la emulación de espacios de nombres) disponen de un mecanismo para ocultar los identificadores locales. Proporcionan los medios para agrupar lógicamente los identificadores relacionados en sus correspondientes espacios de nombres, haciendo así el sistema más modular.

Renombrando identificadores y espacios de nombres

Renaming a Namespace

While importing a module, the name of the `namespace` can be changed:

```
>>> import math as mathematics
>>> print(mathematics.cos(mathematics.pi))
-1.0
```

After this import there exists a namespace `mathematics` but no namespace `math`. It's possible to import just a few methods from a module:

```
>>> from math import pi, pow as power, sin as sinus
>>> power(2,3)
8.0
>>> sinus(pi)
1.2246467991473532e-16
```

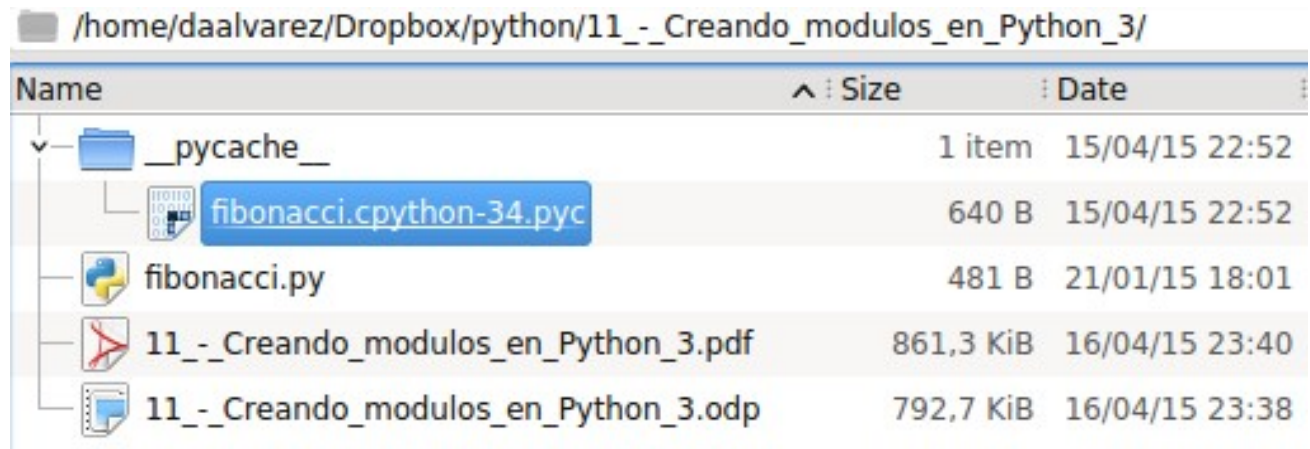
Los archivos .pyc

(archivos de Python "compilados")

Cuando se importa por primera vez un módulo, y con el objeto de acelerar la carga de los módulos en ejecuciones posteriores, Python crea una versión compilada de cada módulo en el directorio `__pycache__` bajo el nombre "`module.version.pyc`", donde la versión codifica el formato del archivo compilado; Por lo general, contiene el número de la versión de Python. Por ejemplo, cuando se hace importa el módulo fibonacci por primera vez:

```
import fibonacci
```

se crea el archivo:



Name	Size	Date
__pycache__	1 item	15/04/15 22:52
fibonacci.cpython-34.pyc	640 B	15/04/15 22:52
fibonacci.py	481 B	21/01/15 18:01
11_-_Creando_modulos_en_Python_3.pdf	861,3 KiB	16/04/15 23:40
11_-_Creando_modulos_en_Python_3.odp	792,7 KiB	16/04/15 23:38

```
./__pycache__/fibonacci.cpython-34.pyc.
```

Los archivos .pyc

(archivos de Python "compilados")

Python comprueba la fecha de modificación del archivo .py y la compara contra aquella de la versión compilada para ver si el archivo .py es más reciente y necesita ser recompilado. Este es un proceso completamente automático.

Si uno borra el archivo .pyc no pasará nada grave: sencillamente, Python lo volverá a crear cuando se ejecute de nuevo el `"import"`.

Modificando los módulos

Por razones de eficiencia, cada módulo se importa una sola vez por sesión del interpretador. Por lo tanto, si usted cambia uno de sus módulos, usted debe:

- o reiniciar el interpretador de Python,
- o si es solo un módulo que usted quiere ensayar interactivamente, use `import importlib`

```
Python 3.4.0 (default, Apr 11 2014, 13:05:11)
[GCC 4.8.2] on linux
Type "copyright", "credits" or "license()" for more information.
>>> import fibonacci
Inicializando Fibonacci ...
>>> import fibonacci
>>> import importlib
>>> importlib.reload(fibonacci)
Inicializando Fibonacci ...
<module 'fibonacci' from '/home/daalvarez/fibonacci.py'>
>>> |
```

Observe que aquí no se importó de nuevo el módulo.

When you run a Python module with

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

the code in the module will be executed, just as if you imported it, but with the `__name__` set to `"__main__"`. That means that by adding this code at the end of your module:

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

Ejecutando módulos como scripts

you can make the file usable as a script as well as an importable module, because the code that parses the command line only runs if the module is executed as the “main” file:

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

If the module is imported, the code is not run:

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

```
>>>
```

This is often used either to provide a convenient user interface to a module, or for testing purposes (running the module as a script executes a test suite).

minmax.py

```
1 def min(a, b):
2     if a < b:
3         return a
4     else:
5         return b
6
7 def max(a, b):
8     if a > b:
9         return a
10    else:
11        return b
12
13 if __name__ == '__main__':
14     print('El máximo de 3 y 10 es', max(3,10))
15     print('El máximo de 3 y -10 es', max(3,-10))
16     print('El mínimo de 3 y 10 es', min(3,10))
17     print('El mínimo de 3 y -10 es', min(3,-10))
```

Si lo que hacemos es *importar* el módulo *minmax* desde otro fichero, así:

programa.py

```
1 from minmax import min, max
```

la variable `__name__` vale `'minmax'`, que es como se llama el módulo. De este modo podemos saber si el código del fichero se está ejecutando o importando. Pues bien, el truco está en ejecutar la batería de pruebas solo cuando el fichero se está ejecutando.

Ruta de búsqueda de los módulos

“`import modulo`” hace lo siguiente:

- Busca un módulo estándar de Python con ese nombre en el directorio actual.
- Si no se encuentra busca el archivo `modulo.py` en la lista de directorios especificado en `sys.path`:
 - Contiene el subdirectorio actual
 - Contiene otros subdirectorios: se busca en el mismo orden que aparecen estos subdirectorios en `sys.path`

```
>>> import sys
>>> sys.path
['', '/home/daa', '/usr/bin', '/usr/lib/python3.4', '/usr/lib/python3.4/plat-i386-linux-gnu', '/usr/lib/python3.4/lib-dynload', '/usr/local/lib/python3.4/dist-packages', '/usr/lib/python3/dist-packages']
>>> |
```

Ruta de búsqueda de los módulos

When a module named `spam` is imported, the interpreter first searches for a built-in module with that name. If not found, it then searches for a file named `spam.py` in a list of directories given by the variable `sys.path`. `sys.path` is initialized from these locations:

- The directory containing the input script (or the current directory when no file is specified).
- `PYTHONPATH` (a list of directory names, with the same syntax as the shell variable `PATH`).
- The installation-dependent default.

Note: On file systems which support symlinks, the directory containing the input script is calculated after the symlink is followed. In other words the directory containing the symlink is **not** added to the module search path.

After initialization, Python programs can modify `sys.path`. The directory containing the script being run is placed at the beginning of the search path, ahead of the standard library path. This means that scripts in that directory will be loaded instead of modules of the same name in the library directory. This is an error unless the replacement is intended. See section [Standard Modules](#) for more information.

Agregando subdirectorios a la ruta de búsqueda

- Se puede hacer simplemente utilizando las funciones de manipulación de listas:

```
>>> import sys
>>> sys.path
['', '/home/daa', '/usr/bin', '/usr/lib/python3.4', '/usr/lib/python
3.4/plat-i386-linux-gnu', '/usr/lib/python3.4/lib-dynload', '/usr/lo
cal/lib/python3.4/dist-packages', '/usr/lib/python3/dist-packages']
>>> sys.path.append('/home/daa/python')
>>> sys.path
['', '/home/daa', '/usr/bin', '/usr/lib/python3.4', '/usr/lib/python
3.4/plat-i386-linux-gnu', '/usr/lib/python3.4/lib-dynload', '/usr/lo
cal/lib/python3.4/dist-packages', '/usr/lib/python3/dist-packages',
'/home/daa/python']
>>> |
```

La ruta de búsqueda de archivos (the search path)

```
>>> import sys
>>> sys.path
['', '/home/daa', '/usr/bin', '/usr/lib/python3.4', '/usr/lib/python
3.4/plat-i386-linux-gnu', '/usr/lib/python3.4/lib-dynload', '/usr/lo
cal/lib/python3.4/dist-packages', '/usr/lib/python3/dist-packages']
>>> sys
<module 'sys' (built-in)>
>>> sys.path.insert(0, '/home/daa/Cursos_UNAL/Programacion_de_PCs/py
thon')
>>> sys.path
['/home/daa/Cursos_UNAL/Programacion_de_PCs/python', '', '/home/daa'
, '/usr/bin', '/usr/lib/python3.4', '/usr/lib/python3.4/plat-i386-li
nux-gnu', '/usr/lib/python3.4/lib-dynload', '/usr/local/lib/python3.
4/dist-packages', '/usr/lib/python3/dist-packages']
>>> |
```

Inserta al principio de la lista

Nota: los módulos “built-in” no tienen un archivo .py asociado, ya que fueron escritos en lenguaje C.

El efecto de agregar un archivo a la ruta desaparece al cerrar Python.²⁰

Ubicación de un módulo

Algunos módulos nos indican en su variable `__file__` donde están ubicados:

```
>>> import random
>>> random.__file__
'/usr/lib/python3.4/random.py'
>>>
>>> import os
>>> os.__file__
'/usr/lib/python3.4/os.py'
>>>
>>> import math
>>> math.__file__      # No todos los módulos tienen esta variable
Traceback (most recent call last):
  File "<pyshell#32>", line 1, in <module>
    math.__file__      # No todos los módulos tienen esta variable
AttributeError: 'module' object has no attribute '__file__'
```

El contenido de un módulo

La función `dir()` permite ver los identificadores que define un módulo dado; también permite ver las variables en memoria.

```
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__',
 '__spec__', 'a']
>>> import fibonacci
Iniciando Fibonacci ...
>>> dir(fibonacci)
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
 '__name__', '__package__', '__spec__', 'fib', 'fib2']
>>> a = [1, 2, 3]
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__',
 '__spec__', 'a', 'fibonacci']
>>>
```

Que quieren decir estos identificadores?

- `__builtins__`
- `__cached__`
- `__doc__`
- `__file__`
- `__loader__`
- `__name__`
- `__package__`
- `__spec__`

Ver: <https://docs.python.org/3/reference/import.html>

Con este comando se pueden listar todos los identificadores definidos por defecto en Python 3:

```
>>> 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__', '__loader__', '__name__', '__package__', '__spec__', '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']
```


Tipos de módulos

There are different kind of modules:

- Those written in Python
They have the suffix: .py
- Dynamically linked C modules
Suffixes are: .dll, .pyd, .so, .sl, ...
- C-Modules linked with the Interpreter:
It's possible to get a complete list of these modules:

```
>>> import sys
>>> sys.builtin_module_names
('_ast', '_bisect', '_codecs', '_collections', '_datetime', '_elementtree',
'_functools', '_heapq', '_imp', '_io', '_locale', '_md5', '_operator', '_pickle',
'_posixsubprocess', '_random', '_sha1', '_sha256', '_sha512', '_socket', '_sre', '_stat', '_string', '_struct', '_symtable', '_thread', '_tracemalloc', '_warnings', '_weakref', 'array', 'atexit', 'binascii', 'builtins', 'errno', 'faulthandler', 'fcntl', 'gc', 'grp', 'itertools', 'marshal', 'math', 'posix', 'pwd', 'pyexpat', 'select', 'signal', 'spwd', 'sys', 'syslog', 'time', 'unicodedata', 'xxsubtype', 'zipimport', 'zlib')
```

Documentación de un módulo

Un módulo estará incompleto sin una buena documentación.

```
1  """
2  Módulo de manejo de vectores y matrices
3  -----
4
5  Este módulo ... bla bla bla
6  """
7
8  import math
9
10 VAR = 10 # Una variable global del módulo
11
12 def crear_vector(n):
13     """
14     Crea un vector de n elementos, todos inicializados en cero.
15     """
16     pass
17
18 def crear_matriz(nfil, ncol):
19     """
20     Crea una matriz de nfil filas y ncol columnas: Todos los elementos de la
21     matriz se inicializan en cero.
22     """
23     pass
```

La documentación del módulo se hace con un docstring como primer comando de un módulo.

```
>>> import vecmat
>>> help(vecmat)
Help on module vecmat:
```

Como crear una buena documentación:
<https://www.python.org/dev/peps/pep-0257/>

NAME

vecmat

DESCRIPTION

Módulo de manejo de vectores y matrices

Este módulo ... bla bla bla

Observe que las funciones se listan alfabéticamente

FUNCTIONS

crear_matriz(nfil, ncol)

Crea una matriz de nfil filas y ncol columnas: Todos los elementos de la matriz se inicializan en cero.

crear_vector(n)

Crea un vector de n elementos, todos inicializados en cero.

DATA

VAR = 10 →

Las variables globales del módulo quedan sin docstring, por lo que se les debe poner un nombre con sentido o se deben describir en el docstring del módulo.

FILE

/home/daalvarez/Dropbox/python/11_-_Creando_modulos_en_Python_3/vecmat.py

```
>>> help(vecmat.crear_vector)
Help on function crear_vector in module vecmat:
```

crear_vector(n)

Crea un vector de n elementos, todos inicializados en cero.

```
>>>
```

Módulos predefinidos en Python 3

Ver: <https://docs.python.org/3/py-modindex.html>

Python Module Index

[_](#) | [a](#) | [b](#) | [c](#) | [d](#) | [e](#) | [f](#) | [g](#) | [h](#) | [i](#) | [j](#) | [k](#) | [l](#) | [m](#) | [n](#) | [o](#) | [p](#) | [q](#) | [r](#) | [s](#) | [t](#) | [u](#) | [v](#) | [w](#) | [x](#) | [z](#)

[__future__](#)

Future statement definitions

[__main__](#)

The environment where the top-level script is run.

[_dummy_thread](#)

Drop-in replacement for the `_thread` module.

[_thread](#)

Low-level threading API.

a

[abc](#)

Abstract base classes according to PEP 3119.

[aifc](#)

Read and write audio files in AIFF or AIFC format.

[argparse](#)

Command-line option and argument parsing library.

[array](#)

Space efficient arrays of uniformly typed numeric values.

[ast](#)

Abstract Syntax Tree classes and manipulation.

[asynchat](#)

Support for asynchronous command/response protocols.

[asyncio](#)

Asynchronous I/O, event loop, coroutines and tasks.

[asyncore](#)

A base class for developing asynchronous socket handling services.

[atexit](#)

Register and execute cleanup functions.

[audioop](#)

Manipulate raw audio data.

b

[base64](#)

RFC 3548: Base16, Base32, Base64 Data Encodings; Base85 and Ascii85

[bdb](#)

Debugger framework.

[binascii](#)

Tools for converting between binary and various ASCII-encoded binary representations.

Referencias

- Wikipedia
- <http://www.inventwithpython.com/>
- <http://www.diveintopython3.net/>
- Documentación de Python:
 - <https://docs.python.org/3/tutorial/index.html>
 - <https://docs.python.org/3/>
- Marzal Varó, Andrés; Gracia Luengo, Isabel; García Sevilla, Pedro (2014). Introducción a la programación con Python 3. Disponible en: <http://dx.doi.org/10.6035/Sapientia93>