Python

Guía para ser un Pythonista

Juan José Lozano Gómez – j2logo.com

14

Espacios de nombres, módulos y paquetes

Índice de contenidos

Introducción

A lo largo de los temas previos hemos visto lo suficiente del lenguaje para que entiendas lo que aquí te voy a explicar.

Como sabes, un programa en Python no se escribe directamente en el intérprete, sino en ficheros con extensión ".py". Pero ¿cómo se organiza el código en estos ficheros? ¿Cómo puedo reutilizar código que ya he escrito? Precisamente estas y otras cuestiones es lo que repasaremos en este tema.

Básicamente, estudiaremos (de nuevo) qué son los nombres, dónde son válidos esos nombres y cómo una aplicación se estructura en módulos y paquetes.

Nombres y espacios de nombres

Nombres

Lo primero que debes tener claro antes de profundizar en los conceptos de este tema es **qué es un nombre**.

Como ya te he señalado en varias ocasiones a lo largo del curso, en Python todo es un objeto. El número `2` es un objeto, el texto `'Hola mundo'` es un objeto, las funciones son objetos, ... Pues bien, **un nombre o identificador es la forma que existe en Python de referenciar a un objeto concreto**. Lo que comúnmente se conoce como variable. En definitiva, una variable en Python no es más que el nombre con el que nos referimos a un objeto.

Dado que en el tema anterior vimos lo que era una función, ¿todavía no te crees que las funciones son objetos y pueden ser referenciadas por variables? Aquí tienes la prueba:

```
>>> def f():
...     print('Soy la función f')
...
>>> type(f)
<class 'function'>
```

```
>>> soy_f = f
>>> soy_f()
>>> Soy la función f
```

Como ves, la función `f` es un objeto cuya clase es `function` (en el tema siguiente veremos qué es una clase y un objeto). Además, como puedes apreciar, `f` se ha asignado a la variable `soy_f`.

Por tanto, queda claro que, en Python, una función es un objeto y, como tal, se puede pasar como argumento de otra función, utilizarse como valor de retorno o, simplemente, asignarse a una variable.

Espacios de nombres

Una vez aclarado qué es un nombre, paso a explicarte qué son los espacios de nombres en Python.

Un espacio de nombres es una colección aislada de nombres (o identificadores) que referencian a objetos.

Básicamente, un espacio de nombres es un sistema que asegura que todos los nombres definidos en un programa sean únicos y puedan usarse sin ningún conflicto.

Como veremos a continuación, en un mismo script o programa Python pueden coexistir varios espacios de nombres a la vez.

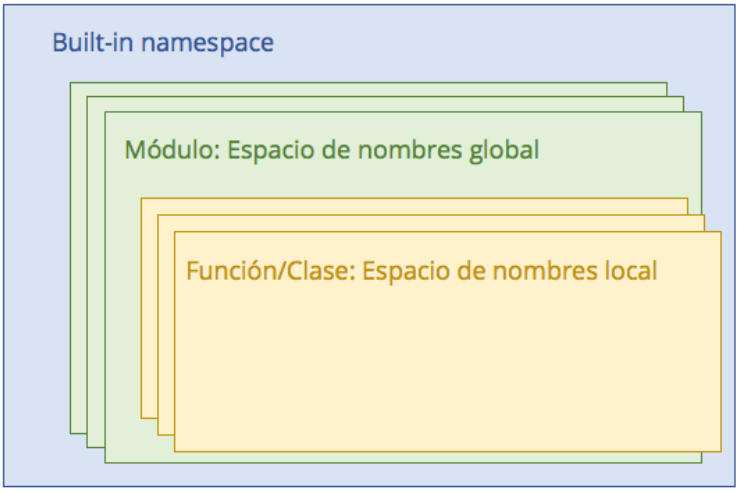
¿Qué espacios de nombres existen?

Cuando accedemos a un intérprete de Python o ejecutamos un programa, hay una serie de identificadores (que no forman parte del lenguaje en sí) que son añadidos a un espacio de nombres al que es posible acceder desde cualquier punto de un script. Es por esto que funciones como `print()`, `len()` o `id()` están siempre accesibles (también se incluyen excepciones y constantes como `None`). Este espacio de nombres es conocido como *espacio de nombres incorporado* (o *built-in namespace*). El espacio de nombres incorporado se crea cuando arranca el intérprete de Python y se mantiene hasta que este finaliza.

Además, cada módulo en Python (en la siguiente sección veremos qué es un módulo) crea su propio *espacio de nombres global*, que contiene todos los nombres definidos en el módulo. Como te decía, los espacios de nombres están aislados. Esa es la razón por la que en diferentes módulos se pueden usar los mismos nombres y estos no interfieren entre sí. El espacio de

nombres global de un módulo se crea cuando el módulo se incluye en el proyecto y dura hasta que finaliza el intérprete.

A su vez, en un módulo existen funciones y clases. La definición de una función añade el nombre de esta en el *espacio de nombres global* del módulo en que se define. Sin embargo, cuando se invoca a una función se crea un *espacio de nombres local* asociado a dicha función que contiene todos los nombres definidos dentro de la misma, incluidos sus parámetros. Este espacio de nombres se mantiene hasta que la función retorna o lanza una excepción no controlada por la propia función. Sucede algo similar para las clases.



Ámbitos de un nombre

Los espacios de nombres nos ayudan a identificar de manera única todos los nombres dentro de un programa. Sin embargo, esto no implica que podamos usar un nombre de variable en cualquier lugar. Un nombre también tiene un alcance o ámbito.

Básicamente, **un ámbito es la parte de un programa desde dónde se puede acceder a un espacio de nombres directamente, es decir, sin utilizar ningún prefijo** para referenciar a un nombre.

A continuación, te muestro una lista de algunos ámbitos que pueden existir durante la ejecución de un programa:

- ***Ámbito local***. Es el ámbito más interno y contiene una lista de los nombres locales disponibles en la función actual.
- ***Ámbito de la función exterior***, conocido como `nonlocal`. La búsqueda de un nombre comienza desde el ámbito externo más cercano y se propaga hacia ámbitos superiores.
- ***Ámbito global***. Contiene todos los nombres globales del módulo actual.

- ***El ámbito más externo***. Contiene una lista de todos los nombres incorporados.

Así, por ejemplo, cuando se referencia a un nombre dentro de una función, la búsqueda de dicho nombre comienza por el ámbito local y va escalando por los ámbitos de nivel superior hasta encontrarlo. Si el nombre no se encuentra en ninguno de los espacios de nombres, el programa lanza una excepción `NameError`.

local, nonlocal y global

Para entender mejor cómo funciona la resolución de nombres según los ámbitos, a continuación, te voy a mostrar varios ejemplos para intentar aclarar este asunto.

Observa el código siguiente:

```
def funcion_a():
    y = 2
    def funcion_b():
        z = 3
        print(z)
    funcion_b()
    print(y)
x = 1
funcion_a()
print(x)
```

En el programa de arriba tenemos una variable `x` que está definida en el *espacio de nombres global*, una variable `y` definida en el *espacio de nombres local* de la función `funcion_a` y una variable `z` que está definida en el *espacio de nombres local* de la función `funcion_b`.

Imagina por un momento que estamos dentro de la función `funcion_b`. La variable `z` es *local* para nosotros (está en el ámbito local), `y` es *no local* y `x` es *global*. Esto quiere decir que, dentro de `funcion_b`, podemos acceder y modificar la variable `z`, pero solo podemos consultar el valor de `x` e `y`. ¿Por qué solo se puede consultar el valor de `x` e `y`?

Si dentro de la función `funcion_b` asignamos un valor a `y`, realmente estamos creando una nueva variable `y` en nuestro *espacio de nombres local*. Esta variable es diferente a la variable *no local* que está definida en la función `funcion_a`. Lo mismo ocurriría con la variable global `x`.

Para poder modificar la variable `x` dentro de `funcion_b`, debemos definir la variable como `global`. Y para modificar la variable `y`, hay que definirla como `nonlocal`.

Redefinamos el programa anterior del siguiente modo:

```
def funcion_a():
    x = 2
    def funcion_b():
        x = 3
        print(x)
    funcion_b()
    print(x)
x = 1
funcion_a()
print(x)
```

Ahora, el resultado sería el siguiente:

```
3
2
1
```

Sin embargo, si definimos la variable `x` como global, la cosa cambia:

```
def funcion_a():
    global x
    x = 2
    def funcion_b():
        global x
        x = 3
        print(x)
    funcion_b()
    print(x)
x = 1
funcion_a()
print(x)
```

En esta ocasión el resultado sería este otro:

```
3
3
3
```

En resumen: Por defecto, las asignaciones a nombres se producen en el ámbito más interno. La sentencia `global` indica que una variable pertenece al ámbito global y cualquier modificación se verá reflejada en dicho ámbito. La sentencia `nonlocal` indica que una variable pertenece al ámbito de la función superior y

cualquier modificación se verá reflejada en dicho ámbito.

La sentencia del

Una vez que sabes qué es un nombre, un espacio de nombres y un ámbito, vamos a profundizar en la sentencia del lenguaje `del`.

Ya hemos visto la sentencia `del` en temas anteriores. Por ejemplo, hemos utilizado esta sentencia para eliminar elementos de una lista. ¿Recuerdas?

```
>>> l = [1, 2, 3]
>>> del l[1]
>>> l
[1, 3]
```

No obstante, la sentencia `del` se puede utilizar en otro contexto. `del x`, donde `x` es una variable (nombre), elimina el enlace de `x` del espacio de nombres al que hace referencia el ámbito actual (puede ser el espacio de nombres *local* o *global*). Esto implica que ya no se pueda hacer uso de `x`, a no ser que se vuelva a asignar otro valor con posterioridad.

Módulos

En Python, un módulo no es más que un fichero que contiene instrucciones y definiciones (de variables, de funciones, de clases, ...). El fichero debe tener extensión `.py` y su nombre se corresponde con el nombre del módulo.

NOTA: Dentro de un módulo, puedes acceder a su nombre a través de la variable global `__name__`.

Los módulos tienen un doble propósito:

- Dividir un programa con muchas líneas de código en partes más pequeñas.
- Extraer un conjunto de definiciones que utilizas frecuentemente en tus programas para ser reutilizadas. Esto evita, por ejemplo, tener que estar copiando funciones de un programa a otro.

Es una buena práctica que un módulo solo contenga instrucciones y definiciones relacionadas entre sí.

Nuestro primer módulo

¡Vamos a crear un módulo!

Sitúate en un directorio para un nuevo proyecto y crea en él un fichero llamado `mis_funciones.py` con el siguiente contenido:

```
def saludo(nombre):
    print(f'Hola {nombre}')
```

Ahora, desde un terminal, sitúate en el directorio anterior y ejecuta el comando `python3` para lanzar el intérprete de Python.

Una vez dentro del intérprete, ejecuta la siguiente sentencia:

```
>>> import mis_funciones
```

Con la instrucción anterior estamos importando el módulo `mis_funciones` en el intérprete. Esto permite acceder a sus definiciones en el ámbito actual. Prueba a llamar ahora a la función `saludo()` de este modo:

```
>>> mis_funciones.saludo('Pythonista')
Hola Pythonista
```

Python trae consigo decenas de módulos estándar adicionales al propio lenguaje. Normalmente, estos módulos se encuentran en el directorio lib dentro de la carpeta de instalación de Python. Algunos de los más importantes los veremos en detalle en la parte final del curso. En los videotutoriales de este tema puedes ver algún ejemplo de uso de estos módulos.

Importar un módulo

Como has visto al final del apartado anterior, para usar las definiciones de un módulo en el intérprete o en otro módulo, primero hay que importarlo. Para ello, se usa la sentencia `import`. Una vez que un módulo ha sido importado, se puede acceder a sus definiciones a través del operador punto `.`.

Aunque puedes importar los módulos y sus definiciones dónde quieras, es una buena práctica que aparezcan al principio del módulo que los importa.

Ya sabes que un módulo puede contener instrucciones y definiciones. Normalmente, las instrucciones son utilizadas para inicializar el módulo y solo se ejecutan la primera vez que aparece el nombre del módulo en una sentencia `import`.

Utilizar la sentencia `import modulo_x`, introduce el nombre `modulo_x` en el espacio de nombres global del módulo que lo importa. No importa ninguna definición contenida en el módulo. Como te indiqué anteriormente, el acceso a sus definiciones se realiza por medio del operador `.`.

Sin embargo, hay otras formas de importar un módulo y/o sus definiciones. Te las muestro todas a continuación:

from ... import ...

Permite importar uno o varios nombres de un módulo del siguiente modo:

```
>>> from mis_funciones import saludo, otra_funcion
>>> saludo('j2logo')
```

De este modo, los nombres importados se añaden al espacio de nombres global del módulo que los importa. No se añade el nombre del módulo origen. Esto nos permite acceder directamente a los nombres definidos en el módulo sin tener que utilizar el operador `.`.

from ... import *

```
>>> from mis_funciones import *
>>> saludo('j2logo')
```

Es similar al caso anterior, solo que importa todas las definiciones del módulo a excepción de los nombres que comienzan por guion bajo `_`.

IMPORTANTE: No es una buena práctica importar así las definiciones de un módulo porque dificulta la lectura. Además, los nombres importados pueden ocultar identificadores y nombres usados en el módulo en el que se importan.

Import ... as / from ... import ... as

Por último, podemos redefinir el nombre con el que una definición será usada dentro de un módulo utilizando la palabra reservada `as`:

```
>>> from mis_funciones import saludo as hola
>>> hola('j2logo')
Hola j2logo
```

Ejecutar módulos como scripts

Un módulo puede ser ejecutado como un *script* o como punto de entrada de un programa cuando se pasa directamente como argumento al intérprete de Python:

```
$> python3 mis_funciones.py
```

Cuando esto ocurre, el código del módulo se ejecuta como si se hubiera importado, con la particularidad de que el nombre del módulo, contenido en el atributo `__name__`, es `__main__`.

Esto hace realmente interesante añadir al final del módulo las siguientes líneas de código, que solo se ejecutarán en caso de que dicho módulo se haya ejecutado como el principal:

```
if __name__ == '__main__':
    hola(Pythonista)
```

No se ejecutarán en caso de que el módulo se importe en otro módulo.

Veremos más sobre ejecución de scripts y aplicaciones en el Tema 16.

Las instrucciones y definiciones de una sesión interactiva del intérprete también forman parte del módulo `__main__`, las cuáles están asociadas a su espacio de nombres global.

Dónde y cómo busca Python los módulos

Como te he indicado previamente, Python trae consigo un gran catálogo de módulos estándar con multitud de funciones y clases que puedes usar en tus aplicaciones. Este catálogo es conocido como la *Biblioteca de Referencia*.

Ahora bien, cuando importamos un módulo, ¿cómo sabe Python a qué módulo nos referimos? ¿Dónde busca Python los ficheros correspondientes a los módulos?

En primer lugar, Python busca el módulo en el catálogo de módulos estándar y si no lo encuentra, entonces busca el fichero en el listado de directorios definidos en la variable `sys.path`. Esta variable es inicializada con las siguientes rutas y localizaciones:

- El directorio en el que se encuentra el script principal.
- PYTHONPATH, una variable de entorno similar a PATH.

- Directorios de instalación por defecto de Python.

La función dir()

La función `dir()` devuelve una lista con todas las definiciones (variables, funciones, clases, ...) contenidas en un módulo.

Por ejemplo, si ejecutamos `dir()` sobre el módulo `mis_funciones` que creamos previamente, obtendríamos el siguiente resultado:

```
>>> dir(mis_funciones)
['__builtins__', '__cached__',
'__doc__', '__file__', '__loader__',
'__name__', '__package__', '__spec__',
'saludo']
```

Fíjate en que al final aparece el nombre `saludo` referente a la función que hemos definido. Los atributos del tipo `__nombre__` son atributos Python por defecto asociados con el módulo.

Si a dir() no se le pasa ningún argumento, entonces devuelve todos los nombres definidos en el ámbito actual, es decir, el ámbito donde se ejecute la función.

Las definiciones del espacio de nombres incorporado se encuentran en el módulo `builtins`. Puedes consultarlas todas si ejecutas las siguientes instrucciones:

```
>>> import builtins
>>> dir(builtins)
```

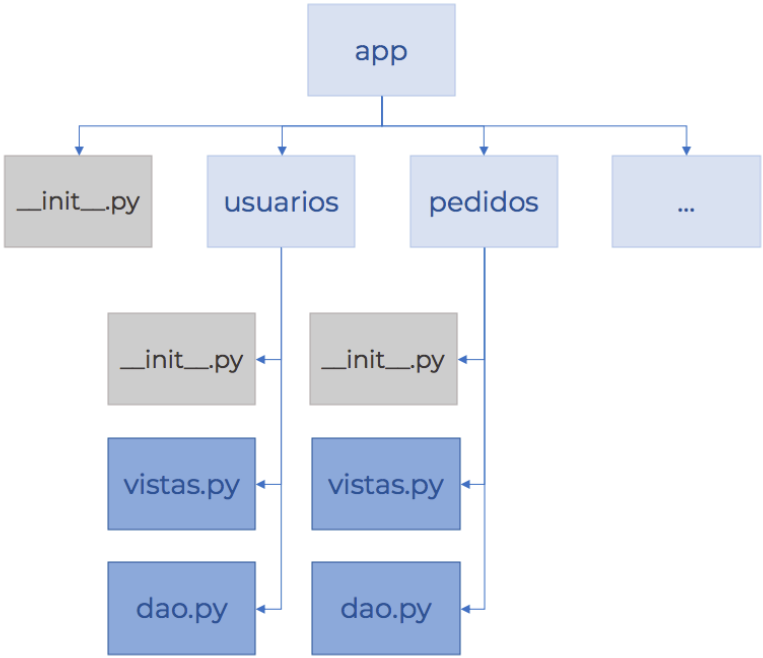
Paquetes

Del mismo modo en que agrupamos las funciones y demás definiciones en módulos, los paquetes en Python permiten organizar y estructurar de forma jerárquica los diferentes módulos que componen un programa. Además, los paquetes hacen posible que existan varios módulos con el mismo nombre y que no se produzcan errores.

Un paquete es simplemente un directorio que contiene otros paquetes y módulos. En Python, para que un directorio sea considerado un paquete, este debe incluir un módulo llamado `__init__.py`. En la mayoría de las ocasiones, el fichero `__init__.py` estará vacío, sin embargo, se puede utilizar para inicializar código relacionado con el paquete.

Al igual que sucede con los módulos, cuando se importa un paquete, Python busca a través de los directorios definidos en `sys.path` el directorio perteneciente a dicho paquete.

Para que lo veas todo de forma gráfica, te muestro los conceptos con una imagen. Imagina que estás haciendo una aplicación para gestionar pedidos. Una forma de organizar los diferentes módulos podría ser la siguiente:



Importar definiciones de un paquete

Para importar módulos y definiciones de módulos que están contenidos en paquetes, se usa el operador `.`. Las referencias se hacen indicando el nombre completo del módulo, es decir, especificando los paquetes hasta llegar al módulo en cuestión separándolos con puntos.

Teniendo en cuenta el diagrama de la sección anterior, si en el módulo `app.pedidos.vistas` se quiere

importar el módulo `app.usuarios.dao`, simplemente hay que añadir la siguiente sentencia:

```
# Módulo app.pedidos.vistas
import app.usuarios.dao
```

El único problema de hacerlo así, es que si, por ejemplo, dicho módulo define una función llamada `guardar()`, hay que especificar toda la jerarquía para invocar a esta función:

```
# Módulo app.pedidos.vistas
app.usuarios.dao.guardar(usuario)
```

Una forma mejor es importar el módulo. Esto se consigue de la siguiente manera:

```
# Módulo app.pedidos.vistas
from app.usuarios import dao
dao.guardar(usuario)
```

Incluso, se puede importar una definición de un módulo del siguiente modo:

```
# Módulo app.pedidos.vistas
from app.usuarios.dao import guardar
guardar(usuario)
```

Además, dentro de un módulo, los paquetes se pueden referenciar de forma relativa.

Imagina ahora que desde el módulo `app.pedidos.vistas` quieres importar los módulos `app.pedidos.dao` y `app.usuarios.vistas`. Se podría hacer como hemos visto hasta ahora:

```
# Módulo app.pedidos.vistas
from app.pedidos import dao
from app.usuarios import vistas
```

O también se podría hacer así:

```
# Módulo app.pedidos.vistas
from . import dao
from ..usuarios import vistas
```

Un punto `.` referencia al paquete actual y dos puntos `..` referencian al paquete padre.