**Nombres y espacios de nombres en Python**

**Nombres**

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

Como ya te he señalado en varias ocasiones a lo largo de los diferentes tutoriales, **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. Equivaldría al concepto de variable**. En definitiva, una variable no es más que el nombre con el que nos referimos a un objeto que existe en memoria.

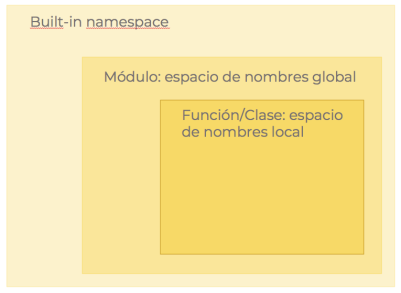
**Espacios de nombres**

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

Cuando accedemos a un intérprete de Python o ejecutamos un programa, **todos los identificadores que define el lenguaje son añadidos a un espacio de nombres al que es posible acceder desde cualquier punto de un script**. Es por esto que las funciones como print() o len() están siempre accesibles. Este espacio de nombres es conocido como *espacio de nombres incorporado* (o *built-in namespace*)

Además, **cada módulo en Python crea su propio espacio de nombres *global***. 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.

A su vez, en un módulo existen funciones y clases. **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** (sucede algo similar para las clases).



**Ámbito de una variable en Python**

**Un *ámbito* define los límites de un programa en los que un espacio de nombres puede ser accedido** sin utilizar un prefijo.

Como te he mostrado en el apartado anterior, en principio existen, como mínimo, tres ámbitos. Uno por cada espacio de nombres:

* Ámbito de la función actual, que tiene los nombres locales a la función.
* Ámbito a nivel de módulo, que tiene los nombres globales, los que se definen en el propio módulo.
* Ámbito incorporado, el más externo, que tiene los nombres que define Python.

Cuando desde dentro de una función se hace referencia a un nombre, este se busca en primer lugar en el espacio de nombres *local*, luego en el espacio de nombres *global* y finalmente en el espacio de nombres *incorporado*.

Si hay una función dentro de otra función, se anida un nuevo ámbito dentro del ámbito local.

**local, nonlocal y global en Python**

Veamos mejor los conceptos anteriores con unos ejemplos:

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 podemos acceder y modificar la variable z pero solo podemos consultar el valor de x e y puesto que se encuentran en un ámbito diferente al nuestro.

Si dentro de la función funcion\_b asignamos un valor a una variable y, realmente estamos creando una nueva variable en nuestro espacio de nombres local. Esta variable es diferente a la variable *no local* y 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.

Redefiniendo el programa del comienzo de este apartado de este modo

def funcion\_a():

x = 2

def funcion\_b():

x = 3

print(x)

funcion\_b()

print(x)

x = 1

funcion\_a()

print(x)

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)

Ahora el resultado sería este otro:

3

3

3

**Qué es un módulo en Python**

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

**NOTA:** Dentro de un módulo, puedes acceder al nombre del mismo 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.

¿Quieres crear tu primer 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 una consola, sitúate en el directorio anterior y ejecuta el comando python3 para lanzar el intérprete de Python.

Una vez dentro del intérprete, ejecuta el siguiente comando:

>>> import mis\_funciones

Con el comando anterior estamos importando el módulo mis\_funciones en el intérprete. Prueba a llamar ahora a la función saludo() de este modo:

>>> mis\_funciones.saludo('j2logo')

Hola j2logo

## Cómo importar módulos en Python

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 palabra reservada 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 y cuando quieras, es una buena práctica que aparezcan al principio del módulo.

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

Previamente te he enseñado una forma de importar un módulo en otro módulo para usar sus definiciones, sin embargo, existen otros modos de usarlas e importarlas.

### from … import …

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

from mis\_funciones import saludo, otra\_funcion

saludo('j2logo')

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 guión bajo \_.

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

### 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 parámetro al intérprete de Python:

>>> python 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, \_\_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('j2logo')

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

## Dónde y cómo busca Python los módulos

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](https://docs.python.org/3/py-modindex.html).

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.

Si a dir() no se le pasa ningún argumento, entonces devuelve todas las definiciones del módulo actual.

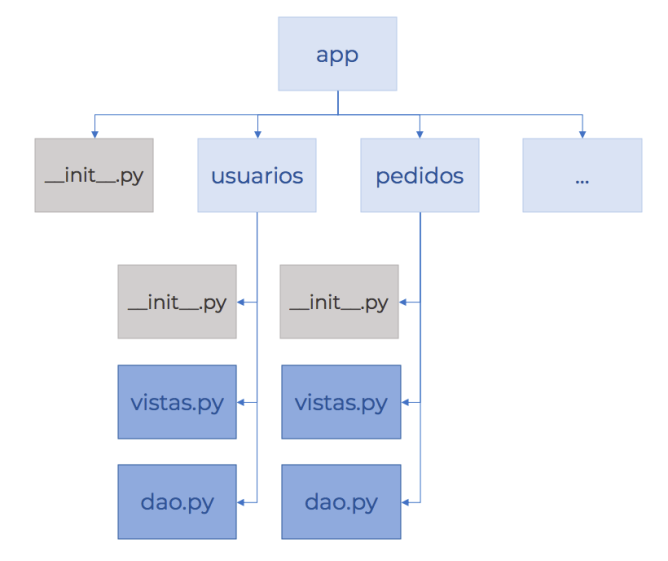
**Paquetes en Python**

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**. Además, en Python, para que un directorio sea considerado un paquete, este debe incluir un módulo llamado \_\_init\_\_.py. En la mayoría de 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:

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)

Y para terminar este tutorial, vamos a ver que 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 # Un punto referencia al paquete actual

from ..usuarios import vistas # Dos puntos referencian al paquete padre

Bueno, hemos llegado al final de este tutorial. Ha sido intenso pero necesario para conocer mejor cómo funciona Python internamente y poder sacar más provecho al lenguaje. ¡Te espero en el siguiente, en el que veremos conceptos de programación orientada a objetos!