Python

Guía para ser un Pythonista

Juan José Lozano Gómez – j2logo.com

16
Ejecución de Scripts

Índice de contenidos

Introducción

Una de las habilidades más importantes que necesitas desarrollar como programador Python es poder ejecutar scripts. Esta será la única forma de saber si tu código funciona como planeaste. ¡Incluso es la única forma de hacer que un programa funcione!

Este tema recopila y muestra diferentes modos de ejecutar una aplicación Python, buenas prácticas y un detalle que no te puedes perder si quieres ser un auténtico Pythonista: hablaremos de entornos virtuales.

Scripts vs Módulos

En informática, la palabra script se usa para referirse a un archivo que contiene una secuencia de instrucciones o un archivo de procesamiento por lotes. Este suele ser un programa simple, almacenado en un archivo de texto sin formato.

Los scripts siempre son procesados por algún tipo de intérprete, que es responsable de ejecutar cada comando secuencialmente.

Sin embargo, en nuestro caso, a un archivo de texto que contiene código Python el cuál está destinado a ser ejecutado directamente por el usuario, generalmente también se le llama script. Por tanto, en Python, **un script suele ser un fichero que se utiliza como punto de entrada de un programa** o módulo principal.

Por otro lado, **a un archivo de texto que contiene código Python que está diseñado para ser importado** y utilizado desde otro archivo Python, **se le llama módulo**.

En definitiva, la principal diferencia entre un módulo y un script es que los módulos deben importarse, mientras que los scripts se ejecutan directamente.

En cualquier caso, lo importante es saber cómo ejecutar el código Python que escribes en tus módulos y scripts. Es lo que veremos en las siguientes secciones.

Ejecución de scripts por consola

Ya vimos en el *Tema 2* que para ejecutar código Python hace falta tener instalado un Intérprete de Python.

Como recordarás, el Intérprete puede ejecutar código de dos maneras:

- En una sesión interactiva.
- Guardando las instrucciones en un fichero y pasando ese fichero como argumento del comando `python3`.

En este tema vamos a profundizar en este segundo método, para que conozcas algunas de las posibilidades que existen a la hora de ejecutar scripts por consola.

__main__

En un tema anterior ya expliqué que cuando pasas un fichero como argumento del comando `python3`, el script se lee y se ejecuta bajo el nombre `__main__`. Esto también es así para el módulo principal de una sesión interactiva.

La particularidad de este hecho es que es posible definir en un script un bloque de código con las instrucciones para inicializar un programa.

Ya sabes que cuando se importa un módulo, todas las instrucciones que contenga (exceptuando las incluidas en las definiciones de funciones, clases, etc.) son ejecutadas.

Por tanto, la manera más habitual de definir un script en Python (el módulo que se utiliza como punto de entrada a un programa) es la siguiente:

```
# mi_programa.py

def main():
    """Función principal del programa"""
    # Código de la función

if __name__ == '__main__':
    main()
```

```
$> python3 mi_programa.py
```

Redirigir la salida

A veces es útil guardar la salida de un script para un análisis posterior. Así es como puedes hacer eso:

```
# hola.py

print('¡Hola Mundo!')
```

```
$> python3 hola.py > salida.txt
```

Esta operación redirige la salida del script `hola.py` al fichero `salida.txt`, en lugar de a la salida estándar del sistema (`stdout`). El proceso se conoce comúnmente como redirección de flujo y está disponible tanto en sistemas *Windows* como en *Unix*.

Si `salida.txt` no existe, entonces se crea automáticamente. Por otro lado, si el archivo ya existe, su contenido será reemplazado por la nueva salida.

Finalmente, si lo que quieres es añadir la salida de ejecuciones consecutivas al final del fichero, debes usar dos corchetes angulares `>>` en lugar de uno, así:

```
$> python3 hola.py >> salida.txt
```

Ejecución de módulos con la opción -m

Python ofrece una serie de opciones de línea de comandos que puedes usar según tus necesidades al invocar al intérprete. Por ejemplo, es posible ejecutar un módulo Python a partir de su nombre usando el comando `python3 -m <nombre-módulo>`.

La opción `-m` busca el nombre del módulo en `sys.path` y ejecuta su contenido como `__main__`:

```
$> python3 -m hola
¡Hola Mundo!
```

La principal ventaja de ejecutar un script de este modo es que, en lugar de indicar una ruta específica, el módulo es buscado en todo el `sys.path`. Además, como puedes observar, no hay que especificar la extensión `.py`.

Los nombres de paquetes (incluidos los paquetes de espacios de nombres) también están permitidos.

Cuando se proporciona un nombre de paquete en lugar de un módulo normal, el intérprete ejecutará `<pkg>.__main__` como el módulo principal.

En los videotutoriales del tema puedes ver un ejemplo de cómo ejecutar un paquete directamente como si fuera un módulo.

Usando el nombre del archivo de script

En versiones recientes de *Windows*, es posible ejecutar scripts de Python simplemente introduciendo el nombre del archivo en el terminal:

```
C:\workspace> hola.py
¡Hola Mundo!
```

Esto es así porque *Windows* usa el registro del sistema y la asociación de archivos para determinar qué programa usar para ejecutar un archivo en particular.

En sistemas *Unix*, es posible lograr algo similar. Para ello, hay que añadir una primera línea al script con el texto `#! /usr/bin/env python3`.

```
#! /usr/bin/env python3
# hola.py

print('¡Hola Mundo!')
```

Para Python, este es un comentario simple, pero para el sistema operativo, dicha línea indica qué programa debe usarse para ejecutar el archivo.

En la línea que comienza con `#!` se indica la ruta del intérprete. Hay dos formas de especificar la ruta del intérprete:

- Escribiendo la ruta absoluta: `#! /usr/bin/python3`
- Usando el comando `env` del sistema operativo, que localiza y ejecuta Python buscando en la variable de entorno `PATH`: `#! /usr/bin/env python3`.

Esta última opción es útil dado que no todos los sistemas *Unix* instalan el intérprete de Python en la misma ubicación.

Finalmente, para ejecutar un script como este, hay que asignarle permisos de ejecución y luego escribir el nombre del archivo en la línea de comandos.

Aquí te muestro un ejemplo de cómo hacer esto:

```
$> # Asignar permisos de ejecución
$> chmod + x hola.py
$> # Ejecuta el script
$> ./hola.py
¡Hola Mundo!
```

Este modo de ejecutar un script es útil cuando se trata de un trabajo por lotes o de tareas de automatización que no dependen de muchos módulos. En otro caso, utiliza alguno de los otros modos propuestos.

Pasando argumentos al script

A continuación, te mostraré varias formas de parametrizar un script y pasar argumentos al ejecutarlo.

sys.argv

Cuando se ejecuta un script, hay ciertos valores que se asignan a la variable `argv` del módulo `sys`. Dichos valores son asignados como una lista de strings.

Esta lista tiene, como mínimo, un elemento, siendo el primero de ellos, `argv[0]`, el nombre del script.

Cuando se lanza simplemente el intérprete, `argv[0]` es una cadena vacía.

```
# argumentos.py
import sys

def listar_argumentos():
    for a in sys.argv:
        print(a)

if __name__ == '__main__':
    listar_argumentos()
```

Al ejecutar el script anterior de este modo

```
$> python3 argumentos.py uno 3 hola
```

la salida correspondiente es la siguiente:

```
argumentos.py
uno
3
hola
```

argparse

Aunque es posible obtener valores por medio de la variable `argv`, como hemos visto en la sección anterior, la forma recomendada por Python para implementar aplicaciones de consola *elegantes* y procesar sus opciones es usando el módulo `argparse`.

Veamos el siguiente script llamado `suma.py`, en el cuál se definen dos parámetros. Un parámetro `n`, posicional y obligatorio, y un parámetro opcional `--suma`. `n` se utiliza para generar una lista con los números del 1 al `n`. Además, cuando se especifica el parámetro `--suma`, el programa suma los números de la lista:

```
# suma.py

import argparse

parser = argparse.ArgumentParser(
    description='Genera una lista de enteros de 1 a n.')
parser.add_argument('n', type=int,
                    help='valor máximo de la lista')
parser.add_argument('-s', '--suma',
                    help='suma los elementos de la lista',
                    action='store_true')

args = parser.parse_args()

n = args.n

l = list(range(1, n+1))
print(l)

if args.suma:
    print(f'La suma es {sum(l)}')
```

Al ejecutar el programa sin pasar ningún argumento, vemos que se muestra un error indicando este hecho. Además, se muestra una ayuda de cómo se debería usar el programa:

```
$> python3 suma.py
usage: suma.py [-h] [-s] n
suma.py: error: the following arguments
are required: n
```

Si pasamos la opción `-h` o `--help`, se muestra una descripción detallada de uso del script con la información proporcionada a `argparse`:

```
$> python3 suma.py -h
usage: suma.py [-h] [-s] n

Genera una lista de enteros de 1 a n.

positional arguments:
  n           valor máximo de la lista

optional arguments:
  -h, --help  show this help message and
exit
  -s, --suma  suma los elementos de la
lista
```

A continuación, se muestra un ejemplo correcto de uso de la aplicación (`-s` o `--suma` es opcional):

```
$> python3 suma.py -s 5
[1, 2, 3, 4, 5]
La suma es 15
```

En los videotutoriales del tema puedes ver una explicación más detallada y ejemplos de uso de `argparse`.

Variables de entorno

El último método que veremos para pasar a un script o programa información del exterior es hacer uso de variables de entorno.

Las variables de entorno son variables que se definen a nivel de sistema operativo y pueden contener, entre otros, valores de configuración. Dichos valores pueden hacer referencia a la propia aplicación, pero también al sistema global en el que esta se ejecuta.

Para leer variables de entorno, hay que usar la función `getenv()` perteneciente al módulo `os`.

El módulo `os` es un módulo muy importante de Python que proporciona funcionalidad dependiente del sistema operativo.

`os.getenv(var, default)` devuelve un string con el valor de la variable de entorno `var` si existe, `default` en caso contrario. Si no se especifica `default` devuelve `None`.

El siguiente ejemplo muestra cómo leer la variable de entorno `PATH`:

```
>>> import os
>>> os.getenv('PATH')
'/Users/Juanjo/workspac…'
```

Entornos virtuales

Hemos llegado a una sección, yo diría, casi imprescindible si quieres ser un buen Pythonista.

Muchos principiantes, esto que te voy a explicar, es algo que ignoran o desconocen, pero **es una de las lecciones más importantes que debes aprender en este curso: qué es un entorno virtual Python**.

Por qué es necesario un entorno virtual

Cuando estás desarrollando varios proyectos Python en un mismo ordenador, es posible que tengas más de un intérprete de Python instalado, por ejemplo, *Python 2.7* y *Python 3.8*. Además, puede que en dos proyectos distintos hagas uso de una misma librería, pero uses una versión diferente en cada uno de ellos.

Una librería es un conjunto de módulos y/o funcionalidades que no forman parte del núcleo de Python. Generalmente, suelen ser desarrolladas por un tercero y pueden incluirse en un proyecto para reutilizar la totalidad o parte de las utilidades que ofrece. Las veremos más en detalle en la siguiente sección.

¿Cómo haces para que dichas librerías no entren en conflicto? Imagina que tienes dos proyectos llamados `A` y `B` y que ambos hacen uso de una misma librería `lib_x`. El proyecto `A` enlaza con la versión `1.0.0` de la librería `lib_x` y el proyecto `B`, por una serie de requisitos, utiliza nuevas funcionalidades de la librería `lib_x` que están recogidas en la versión `2.0.0`.

Inicialmente no es posible tener instaladas las dos versiones de la librería en el sistema, o se usa la versión `1.0.0` o la `2.0.0`. Es aquí donde entran en juego los entornos virtuales.

Qué es un entorno virtual

Un entorno virtual es un ambiente de configuración y ejecución de Python totalmente aislado e independiente. En un sistema, es posible disponer de múltiples entornos virtuales, de manera que la configuración y dependencias de uno no entren en conflicto con los de otro.

Básicamente, un entorno virtual está compuesto por un intérprete de Python, una configuración específica y un conjunto de librerías independientes.

Cómo crear un entorno virtual

Para crear un entorno virtual, tan solo hay que ejecutar el siguiente comando dentro del directorio principal de un proyecto:

```
$> python3 -m venv env
```

Dicho comando creará un entorno virtual en el directorio `env` con la siguiente estructura:

```
/env
    |__/bin
    |__/include
    |__/lib
```

El directorio `bin` (se llama `Scripts` en *Windows*) contiene los ejecutables: como los ficheros de activación del entorno, el intérprete de Python o *pip*. Por su parte, los directorios `include` y `lib` contienen ficheros de cabecera y librerías, necesarios para la correcta ejecución de la aplicación en este entorno. Las librerías de terceros se instalan en `env/lib/pythonX.X/site-packages/`.

Usar el entorno

El comando anterior crea el entorno, sin embargo, si instalamos cualquier paquete o librería no lo haremos en dicho entorno, sino en el directorio de instalación global de Python. Para que eso ocurra, primero debemos activar el entorno del siguiente modo:

```
# Unix
$> source env/bin/activate

# Windows (cmd)
$> env\Scripts\activate.bat

# Windows (PowerShell)
$> env\Scripts\activate.ps1
```

Sabremos que estamos dentro del entorno virtual porque el prompt del terminal comienza con *(env)*:

Cualquier librería que instalemos a partir de ahora será dentro del entorno virtual.

Además, dentro del entorno, no es necesario invocar al intérprete de Python usando el comando `python3`, sino, simplemente, `python`.

Para salir del entorno virtual, simplemente hay que ejecutar el siguiente comando:

```
$> deactivate
```

IMPORTANTE: A partir de ahora, aunque yo no lo indique, crea un entorno virtual para cada uno de tus proyectos. Es una buena práctica hacerlo, te ahorrarás problemas en el futuro y mantendrás limpia la instalación principal de Python.

Leyendo variables de entorno en un entorno virtual

Cuando se usa un entorno virtual, las variables de entorno se suelen definir en el fichero *activate*, preferiblemente al final del este (será el único archivo del entorno que tengas/debas modificar a mano). A continuación, te muestro un ejemplo de cómo definir una variable de entorno en función del fichero *activate*:

```
# Unix env/bin/activate
export VARIABLE="valor"

# Windows (cmd) env\Scripts\activate.bat
set “VARIABLE=valor"

# Windows (PowerShell) env\Scripts\activate.ps1
$Env:VARIABLE = “valor”
```

Instalación de dependencias con pip

En la sección anterior te comenté que es posible usar dependencias o librerías de terceros, que no forman parte del core de Python, en una aplicación.

Para ello, es necesario instalarlas previamente en el equipo (hazlo siempre dentro de un entorno virtual, a no ser que de verdad quieras/debas instalar una librería en el directorio de instalación de Python).

Uno de los modos de instalar dependencias en Python es a través de la herramienta *pip*.

Pip es una utilidad de línea de comandos que permite instalar y administrar librerías y paquetes Python. Muchos de estos paquetes y librerías se encuentran alojados en el *Python Package Index* o *PyPI*, el repositorio oficial de Python para paquetes de terceros.

Normalmente, *pip* se incluye en la instalación por defecto de Python.

En los videotutoriales del tema puedes ver ejemplos de uso de *pip* junto con entornos virtuales.

Para instalar una librería, simplemente debes ejecutar el siguiente comando:

```
$> pip install <nombre-librería>
```