2. Shell scripting

2.3 Creación de *scripts* en Python

Elena García-Morato, Felipe Ortega, Enrique Soriano, Gorka Guardiola

GSyC, ETSIT. URJC.

Laboratorio de Sistemas (LSIS)

20 febrero, 2023

Universidad
Rey Juan Carlos

Tema 2 - Shell scripting

2.3 Creación de *scripts* en Python

Contenidos

2.3.1 Estructura de un *script* en Python

Scripts en Python

- Se utiliza la técnica *hash bang* para indicar el intérprete de Python que queremos que ejecute el programa.
- Diversas opciones:
  - El intérprete por defecto instalado en el sistema (`/usr/bin/python3`).
  - El intérprete de un entorno virtual de Anaconda o Miniconda. Por ejemplo: `/home/jfelipe/miniconda3/envs/mi-entorno/bin/python3`.
  - El intérprete de un entorno virtual creado con `virtualenv` o `pipenv`.
    - Documentación de virtualenv.

Scripts en Python

- Ejemplo: `hello.py`.
- Se puede ejecutar con `./hello.py`, si tiene permisos de ejecución.
- No hace falta llamar a `python3` en la línea de comandos (ya se indica en la primera línea del archivo).

```
#!/usr/bin/python3
# Esto es un script en lenguaje Python
print('Hello World!')
```

Scripts vs módulos en Python

- También se puede crear un archivo Python para utilizarlo de dos formas posibles (sin *hash bang* en la primera línea).
  - Importarlo o ejecutarlo como un módulo.
    - `python3 -m miarchivo.py`
  - Ejecutarlo como un *script*. El bloque de código que se ejecuta debe ir dentro de una cláusula **`if`** especial.
    - `if __name__ == "__main__":`

Scripts vs módulos en Python

```python
# echo.py

def echo(text: str, repetitions: int = 3) -> str:
    """Imitate a real-world echo."""
    echoed_text = ""
    for i in range(repetitions, 0, -1):
        echoed_text += f"{text[-i:]}\n"
    return f"{echoed_text.lower()}."

if __name__ == "__main__":
    text = input("Yell something at a mountain: ")
    print(echo(text))
```

Fuente: RealPython.

Scripts vs módulos en Python

- Podemos ejecutarlo como un *script*.
- O también, podemos importarlo en un intérprete de Python como un módulo para utilizar la función `echo` que hemos definido.
  - En este segundo caso, el bloque dentro de la sentencia `if __name__ == "__main__":` no se ejecuta.
  - Lo que hace esta cláusula `if` es comprobar si el intérprete está ejecutando el programa en el *top-level code environment* (como un script) o no (como un módulo).

Scripts vs módulos en Python

1. Ejecuta el siguiente *script* en la línea de comandos. Observa el resultado.
2. Ahora, importa el *script* en un intérprete de Python y observa el nuevo resultado.

```
# namemain.py

print(__name__, type(__name__))
```

Fuente: RealPython.

Scripts vs módulos en Python

- En todo caso, cuando estamos creando un *script* puro en Python es mejor evitar utilizar el idiom `if __name__ == "__main__":` .

Finalización: `sys.exit()`

- Es recomendable finalizar nuestro *script* de Python devolviendo un código de status, como en los programas de shell.
- Para ello podemos usar la función `sys.exit(code)`.
- Por defecto, devuelve 0 (ejecución terminó con éxito). Podemos pasarle un código diferente para devolver el código de estado adecuado.

```
import sys
...
sys.exit()
```

2.3.2 Bibliotecas estándar en Python

Lectura de argumentos de entrada

- Podemos leer argumentos de la línea de comandos con la variable `sys.argv`.

```python
# echo.py

import sys

def echo(text: str, repetitions: int = 3) -> str:
    """Imitate a real-world echo."""
    echoed_text = ""
    for i in range(repetitions, 0, -1):
        echoed_text += f"{text[-i:]}\n"
    return f"{echoed_text.lower()}."

if __name__ == "__main__":
    text = " ".join(sys.argv[1:])
    print(echo(text))
```

Interfaces de línea de comandos

- El módulo `argaparse` permite parsear opciones, argumentos y subcomandos que pasamos al invocar a nuestro *script* para ejecutarlo. Ejemplo `prog.py`.

```python
# ejemplo_parser.py

import argparse

parser = argparse.ArgumentParser(description='Process some integers.')
parser.add_argument('integers', metavar='N', type=int, nargs='+',
                    help='an integer for the accumulator')
parser.add_argument('--sum', dest='accumulate', action='store_const',
                    const=sum, default=max,
                    help='sum the integers (default: find the max)')

args = parser.parse_args()
print(args.accumulate(args.integers))
```

Interfaces de línea de comandos

- Ejemplos de ejecución de `ejemplo_parser.py`.

```
$ python ejemplo_parser.py -h
usage: ejemplo_parser.py [-h] [--sum] N [N ...]

Process some integers.

positional arguments:
    N           an integer for the accumulator

options:
    -h, --help  show this help message and exit
    --sum       sum the integers (default: find the max)
```

Interfaces de línea de comandos

- Ejemplos de ejecución de `ejemplo_parser.py`.

```
$ python ejemplo_parser.py 3 4 5
5

$ python ejemplo_parser.py --sum 3 4 5
12
```

Interfaces de línea de comandos

- Si se invoca al programa con argumentos no válidos (por ejemplo, cadenas de caracteres) da error.

```
$ python ejemplo_parser.py a b c
usage: ejemplo_parser.py [-h] [--sum] N [N ...]
ejemplo_parser.py: error: argument N: invalid int value: 'a'
```

Interfaces de línea de comandos

- Primero se crea un objeto de tipo `ArgumentParser`, para leer e interpretar los argumentos de entrada que se pasan al programa.
- Tiene muchos posibles parámetros. Algunos son:
  - `usage` String que describe el uso del programa (se genera automáticamente, pero se puede configurar aquí).
  - `description` Texto que se muestra antes de la ayuda de los argumentos (por defecto no imprime nada).
  - `epilog` Texto que se muestra después de la lista de ayuda de los argumentos (por defecto no imprime nada).
  - `add_help` Añade automáticamente la opción `-h/--help` al parseador (por defecto `True`).

Interfaces de línea de comandos

- El método `add_argument()` permite añadir nuevos argumentos a la lista reconocida por el parser.
  - `name or flags` Primer argumento obligatorio, puede ser un nombre o una lista de opciones como `-f, -file`.
  - `action` Tipo de acción que se realiza al encontrar este argumento. Algunos posibles valores:
    - `'store'` Valor por defecto, simplemente guarda el valor del argumento.
    - `'store\_const'` Almacena el valor especificado en el argumento clave `const` (cuidado, por defecto `const=`**`None`**).
    - `'store\_true'` o `'store\_false'` Son un caso especial del anterior, para almacenar valores booleanos **`True`** o **`False`**.
    - `'append'` Crea una lista y añade cada argumento a esa lista. Es útil cuando una opción se puede especificar muchas veces.

Interfaces de línea de comandos

- Más argumentos de `add_argument()`.
  - `const` Guarda un valor constante que se añade a uno de los atributos parseados. Puedes ser el nombre de un método o función (como en el ejemplo anterior).
  - `default` Indica el valor que se asigna a este argumento si no está presente en la línea de comandos al invocar el programa.
  - `type` Indica si el tipo del argumento leído debe ser diferente de un string, por ejemplo, un `int` o un `float`.
  - `dest` Cambia el nombre de la variable de destino que guarda el valor final del argumento.
  - `help` Descripción breve de este argumento (para la ayuda).
  - `metavar` Identificador que se usará en el texto de ayuda para identificar al valor que se pasa con este argumento.

Interfaces de línea de comandos

- El método `parse_args()` activa el parseo de los argumentos y opciones de entrada introducidos al ejecutar del programa.
- Se pueden pasar opciones directamente como un argumento en la llamada del método. Por ejemplo:

```
parser.parse_args(['--foo=F00'])
```

- Verifica de forma automática posibles errores: opciones ambiguas, tipo de dato inválido, opción no válida, número erróneo de argumentos posicionales, etc. Al detectar un error, aborta la ejecución e imprime un mensaje describiendo la utilización correcta.

Interfaces de línea de comandos: referencias

- Documentación de `argparse`.
- Tutorial sobre `argparse` en RealPython.
- Comparativa con otras alternativas para parsear argumentos y opciones de la línea de comandos.

2.3.3 Automatización de tareas

Módulos para automatización de tareas

- El módulo `re` incorpora soporte para usar expresiones regulares en Python. Pero no lo vamos a ver (solo a título informativo).
- El módulo `os` ofrece una interfaz para acceder a los servicios del sistema operativo.
  - Métodos para gestionar y obtener información sobre procesos y usuarios.
  - Información sobre el sistema. Por ejemplo `os.cpu_count()` devuelve el número de CPUs presentes en el sistema.
  - `os.path` ofrece funciones para trabajar con rutas de ficheros/directorios. Otra opción es el módulo `pathlib`.

Módulos para automatización de tareas

- El módulo `time` devuelve la fecha y hora actuales.
- También puede servir para programar la ejecución de tareas, que comiencen en una fecha y hora determinada.

2.3.4 Ejemplos

Ejemplos propuestos

- Crea un *script* en Python que reciba 2 argumentos obligatorios de tipo `float` por línea de comandos y 4 posibles argumentos opcionales que indican cómo se comporta:
  - `-s, -suma`
    Suma los valores de los dos argumentos obligatorios y almacena el resultado en la variable `result`.
  - `-r, -resta`
    Resta el primer argumento obligatorio al segundo y almacena el resultado en la variable `result`.
  - `-p, -producto`
    Multiplica entre sí los valores de los dos argumentos obligatorios y almacena el resultado en la variable `result`.
  - `-e, -elevado`
    Eleva el primer argumento obligatorio a la potencia que indica el segundo argumento y almacena el resultado en la variable `result`.

Para saber más

- El libro de Sweigart [1] es perfecto como ampliación y referencia para el contenido de esta sección. La primera presenta de forma concisa los principales elementos de programación en Python. La segunda parte describe mediante proyectos prácticos y código un buen número de tareas útiles que se pueden automatizar con *scripts* en Python. Accede al libro en O'Reilly Learning.

- El curso en vídeo *Complete Python Scripting for Automation* contiene muchos ejemplos y explicaciones adicionales sobre desarrollo de scripts usando Python para automatización de tareas.

Referencias I

[1] A. Sweigart. *Automate the Boring Stuff with Python: Practical Programming for Total Beginners.* 2ª ed. No Starch Press, 2019.