

# ★ Python 3 para impacientes ★

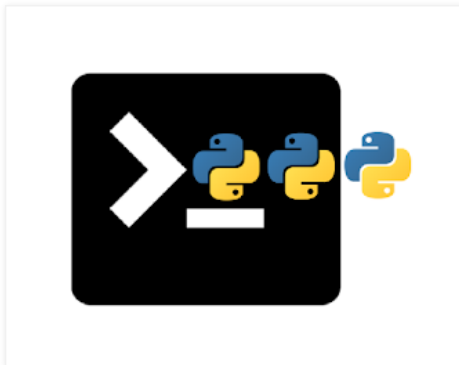


"Simple es mejor que complejo" (Tim Peters)

Python	IPython	EasyGUI	Tkinter	JupyterLab	Numpy
--------	---------	---------	---------	------------	-------

lunes, 25 de julio de 2016

## Cmd: Construyendo un intérprete de comandos



El módulo **cmd** se utiliza para construir de forma sencilla un intérprete de comandos orientado a líneas. En general, un intérprete de comandos se usa para desarrollar herramientas administrativas aunque también es útil para dotar a los programas de una interfaz básica sin emplear mucho tiempo.

El módulo **cmd** incorpora la clase **Cmd** de la que se pueden derivar otras clases que hereden sus métodos. Después, estos métodos pueden ser reescritos a conveniencia.

### Métodos de la clase Cmd

#### Entrando en bucle: Cmd.cmdloop()

El método **cmdloop()** muestra un aviso cíclico o en bucle que indica el lugar donde se pueden introducir los comandos cuando un intérprete de comandos construido con la clase **Cmd** está en ejecución:

#### Cmd.cmdloop(intro=None)

El siguiente ejemplo muestra el modo de crear un intérprete de comandos elemental basado en la clase **Cmd**. Después de importar el módulo **cmd** se declara la clase **Comandos** que hereda atributos y métodos de la clase **cmd.Cmd**. La clase **Comandos** consta de varios métodos cuyos nombres comienzan con el prefijo **"do\_"** y finalizan con los nombres de los comandos que aceptará el intérprete: **comando1**, **comando2** y **salir**.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
#

import cmd

class Comandos(cmd.Cmd):
    """Interprete de comandos"""
    prompt = "Introduza un comando: "

    def do_comando1(self, args):
        """Ayuda del comando1"""
        print("comando1 se ha ejecutado")

    def do_comando2(self, args):
        """Ayuda del comando2"""
        print("comando2 se ha ejecutado")

    def do_salir(self, args):
        """Salir del interprete"""
        print("Hasta pronto")
```

#### Buscar

#### Python para impacientes

[Python](#)  
[IPython](#)  
[EasyGUI](#)  
[Tkinter](#)  
[JupyterLab](#)  
[Numpy](#)

#### Anexos

[Guía urgente de MySQL](#)  
[Guía rápida de SQLite3](#)

#### Entradas + populares

##### [Dar color a las salidas en la consola](#)

En Python para dar color a las salidas en la consola (o en la terminal de texto) existen varias posibilidades. Hay un método basado ...

##### [Instalación de Python, paso a paso](#)

Instalación de Python 3.6 A finales de 2016 se produjo el lanzamiento de Python 3.6. El propósito de esta entrada es mostrar, pas...

##### [Añadir, consultar, modificar y suprimir elementos en Numpy](#)

Acceder a los elementos de un array. [], [..], ... Acceder a un elemento de un array. Para acceder a un elemento se utiliz...

##### [Variables de control en Tkinter](#)

Variables de control Las variables de control son objetos especiales que se asocian a los widgets para almacenar sus valore...

##### [Cálculo con arrays Numpy](#)

Numpy ofrece todo lo necesario para obtener un buen rendimiento cuando se trata de hacer cálculos con arrays. Por como está concebido...

##### [Tkinter: interfaces gráficas en Python](#)

Introducción Con Python hay muchas posibilidades para programar una interfaz gráfica de usuario ( GUI ) pero Tkinter es fácil d...

##### [Operaciones con fechas y horas. Calendarios](#)

Los módulos datetime y calendar amplían las posibilidades del módulo time que provee funciones para manipular expresiones de ti...

##### [Convertir, copiar, ordenar, unir y dividir arrays Numpy](#)

Esta entrada trata sobre algunos métodos que se utilizan en Numpy para convertir listas en arrays y viceversa; para copiar arrays d...

##### [Tkinter: Tipos de ventanas](#)

Ventanas de aplicación y de diálogos En la entrada anterior tratamos los distintos gestores de geometría que se utilizan para di...

##### [Threading: programación con hilos \(I\)](#)

En programación, la técnica que permite que una aplicación ejecute

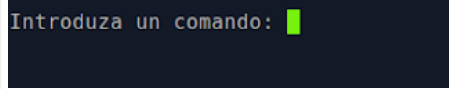
```

        return(True)

if __name__ == '__main__':
    interprete = Comandos()
    interprete.cmdloop()

```

Si ejecutamos el ejemplo el intérprete éste mostrará en la salida estándar el literal "Introduzca un comando:" y quedará a la espera de la entrada de algún comando.



Después de escribir un comando (al que pueden seguir argumentos) se presionará la tecla **[Return]** para su ejecución. Después, el intérprete analizará la cadena y si el comando existe llamará al método correspondiente. En caso contrario, aparecerá el mensaje "Unknown syntax". Finalmente, volverá a mostrar el mensaje de petición, permaneciendo a la espera de la entrada de un nuevo comando.

La tecla **[Return]** repite el comando anterior. Y el comando **salir** hará que finalice el programa.

El argumento opcional **intro** de **cmdloop()** es una cadena que sirve para mostrar un mensaje (de bienvenida, con instrucciones, etc.) cuando se inicia el bucle.

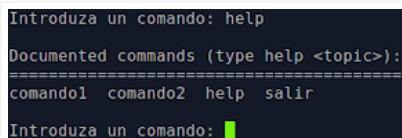
### Readline

El módulo **cmd** utiliza el módulo **readline** para permitir la edición de los comandos en el punto de petición, habilitando, entre otras, las siguientes teclas o combinaciones de teclas:

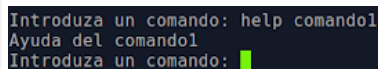
- **Control + P** retrocede al comando anterior,
- **Control + N** avanza al comando siguiente,
- **Control + F** mueve cursor a derecha (sin borrado),
- **Control + B** mueve cursor a izquierda (sin borrado), etc.

### Ayuda del intérprete

Para mostrar la ayuda del intérprete introducir en el punto de petición el comando **help** o el carácter **?**. La ayuda general muestra todos los comandos que acepta el intérprete.



También, es posible mostrar la ayuda específica de un comando introduciendo **help comando** o **? comando**. La ayuda a mostrar de un comando será la cadena de documentación que contenga su método **do\_comando** correspondiente.



Todas las subclases de **Cmd** heredan el método predefinido **do\_help**. Este método se puede reescribir para personalizar la información de ayuda que se desea ofrecer a los usuarios y/o para definir otra funcionalidad.

```

import cmd

class Comandos(cmd.Cmd):
    """Interprete de comandos"""
    prompt = "Introduzca un comando: "

    def do_comando1(self, args):
        """Ayuda del comando1"""
        print("comando1 se ha ejecutado")

    def do_comando2(self, args):
        """Ayuda del comando2"""

```

simultáneamente varias operaciones en el mismo espacio de proceso se...

#### Archivo

julio 2016 (2) ▾

#### python.org



#### pypi.org



#### Sitios

- [ActivePython](#)
- [Anaconda](#)
- [Bpython](#)
- [Django](#)
- [Flask](#)
- [Ipython](#)
- [IronPython](#)
- [Matplotlib](#)
- [MicroPython](#)
- [Numpy](#)
- [Pandas](#)
- [Pillow](#)
- [PortablePython](#)
- [PyBrain](#)
- [PyCharm](#)
- [PyDev](#)
- [PyGame](#)
- [Pypi](#)
- [PyPy](#)
- [Pyramid](#)
- [Python.org](#)
- [PyTorch](#)
- [SciPy.org](#)
- [Spyder](#)
- [Tensorflow](#)
- [TurboGears](#)

```

        print("comando2 se ha ejecutado")

    def do_salir(self, args):
        """Salir del interprete"""
        print("Hasta pronto")
        return(True)

    def do_help(self, args):
        print("Ayuda")

if __name__ == '__main__':
    interprete = Comandos()
    interprete.cmdloop(intro="Bienvenido")

```

Una clase basada en la clase **Cmd** puede tener para otros cometidos otros métodos que no sean comandos, es decir, métodos que no sigan la nomenclatura **"do\_comando"**. Para implementar la ayuda de estos métodos es necesario otro método que se llame igual pero que tenga el prefijo **"help\_"**. Ejemplo: Si el método se llama **"calcular"** su método de ayuda se llamará **"help\_calcular"**.

Para finalizar, hacer mención al comando **!** que llama al método **do\_shell()** si está definido.

### Ejecutar comandos, uno a uno: **Cmd.onecmd()**

El método **onecmd()** ejecuta un comando como si se hubiera introducido en el punto de petición.

En el siguiente ejemplo se ha definido un método para el **comando3** que ejecuta secuencialmente los comandos **comando1** y **comando2**.

```

import cmd

class Comandos(cmd.Cmd):
    """Interprete de comandos"""
    prompt = "Introduza un comando: "

    def do_comando1(self, args):
        """Ayuda del comando1"""
        print("comando1 se ha ejecutado")

    def do_comando2(self, args):
        """Ayuda del comando2"""
        print("comando2 se ha ejecutado")

    def do_comando3(self, args):
        """Ayuda del comando3: ejecuta comando1 y comando2"""
        self.onecmd("comando1")
        self.onecmd("comando2")

    def do_salir(self, args):
        """Salir del interprete"""
        print("Hasta pronto")
        return(True)

if __name__ == '__main__':
    interprete = Comandos()
    interprete.cmdloop(intro="Bienvenido")

```

Otra alternativa es utilizar el método **onecmd()** prescindiendo del bucle de **cmdloop()** para ejecutar secuencialmente una lista de comandos. En este caso, cuando termine la ejecución de los comandos finalizará el programa. Puede ser útil para probar la ejecución de una secuencia de métodos predeterminada sin necesidad de ir tecleando, cada vez, los comandos necesarios.

```

import cmd

class Comandos(cmd.Cmd):
    """Interprete de comandos"""
    prompt = "Introduza un comando: "

    def do_comando1(self, args):
        """Ayuda del comando1"""
        print("comando1 se ha ejecutado")

    def do_comando2(self, args):
        """Ayuda del comando2"""
        print("comando2 se ha ejecutado")

    def do_salir(self, args):

```

```

        """Salir del interprete"""
        print("Hasta pronto")
        return(True)

if __name__ == '__main__':
    interprete = Comandos()
    interprete.onecmd("comando1")
    interprete.onecmd("comando2")

```

### Línea sin comando: Cmd.emptyline()

Si en el punto de petición no se escribe ningún comando y se presiona la tecla **[Return]** el intérprete repetirá el último comando introducido. Es posible implementar otra acción diferente reescribiendo el método **emptyline()**.

En el siguiente ejemplo cuando se presiona la tecla **[Return]** no se realiza ninguna acción.

```

import cmd

class Comandos(cmd.Cmd):
    """Interprete de comandos"""
    prompt = "Introduza un comando: "

    def do_comando1(self, args):
        """Ayuda del comando1"""
        print("comando1 se ha ejecutado")

    def do_comando2(self, args):
        """Ayuda del comando2"""
        print("comando2 se ha ejecutado")

    def do_salir(self, args):
        """Salir del interprete"""
        print("Hasta pronto")
        return(True)

    def emptyline(self):
        """No realiza ninguna accion"""
        pass

if __name__ == '__main__':
    interprete = Comandos()
    interprete.cmdloop(intro="Bienvenido")

```

### Comando erróneo: Cmd.default()

El método **default()** es llamado cuando un comando introducido no es reconocido por el intérprete. Por defecto, cuando un comando no existe se muestra el mensaje de error **"Unknown syntax"** antes de que el intérprete solicite un nuevo comando. Este comportamiento se puede modificar reescribiendo el método **default()**.

```

import cmd

class Comandos(cmd.Cmd):
    """Interprete de comandos"""
    prompt = "Introduza un comando: "

    def do_comando1(self, args):
        """Ayuda del comando1"""
        print("comando1 se ha ejecutado")

    def do_comando2(self, args):
        """Ayuda del comando2"""
        print("comando2 se ha ejecutado")

    def do_salir(self, args):
        """Salir del interprete"""
        print("Hasta pronto")
        return(True)

    def default(self, args):
        print("Error. Comando no reconocido:", args)

if __name__ == '__main__':
    interprete = Comandos()
    interprete.cmdloop(intro="Bienvenido")

```

### Antes y después de ejecutar un comando: `Cmd.precmd()` y `Cmd.postcmd()`

El método `precmd()` es llamado justo antes del método del comando introducido. Este método se puede emplear para realizar comprobaciones o acciones previas antes de ejecutar el correspondiente comando.

En el ejemplo que sigue el método `precmd()` se utiliza para convertir la cadena del comando introducido a minúsculas antes de su ejecución. Así, si una persona escribe un comando en mayúsculas el intérprete no producirá ningún error.

El método `postcmd()` es llamado después de la ejecución del método de un comando. Utiliza la bandera `stop` para controlar si la ejecución se dará por terminada después de la llamada a `postcmd()`, siendo también el valor de retorno del método `onecmd()`. En el ejemplo se utiliza para recordar que el comando salir finaliza la ejecución del intérprete.

```
import cmd

class Comandos(cmd.Cmd):
    """Interprete de comandos"""
    prompt = "Introduza un comando: "

    def do_comando1(self, args):
        """Ayuda del comando1"""
        print("comando1 se ha ejecutado")

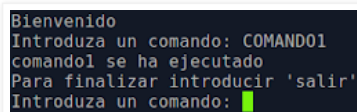
    def do_comando2(self, args):
        """Ayuda del comando2"""
        print("comando2 se ha ejecutado")

    def do_salir(self, args):
        """Salir del interprete"""
        print("Hasta pronto. ", end="")
        return(True)

    def precmd(self, args):
        args = args.lower()
        return(args)

    def postcmd(self, stop, args):
        if args == "salir":
            stop = True
        else:
            print("Para finalizar introducir 'salir'")
            stop = False
        return(stop)

if __name__ == '__main__':
    interprete = Comandos()
    interprete.cmdloop(intro="Bienvenido")
```



```
Bienvenido
Introduza un comando: COMANDO1
comando1 se ha ejecutado
Para finalizar introducir 'salir'
Introduza un comando: █
```

### Al iniciar y terminar el bucle: `Cmd.preloop()` y `Cmd.postloop()`

El método `preloop()` se ejecuta una vez al inicio del bucle, es decir, cuando se llama al método `cmdloop()`; y el método `postloop()` se ejecuta una vez cuando el bucle está a punto de finalizar.

```
import cmd

class Comandos(cmd.Cmd):
    """Interprete de comandos"""
    prompt = "Introduza un comando: "

    def do_comando1(self, args):
        """Ayuda del comando1"""
        print("comando1 se ha ejecutado")

    def do_comando2(self, args):
        """Ayuda del comando2"""
```

```

        print("comando2 se ha ejecutado")

    def do_salir(self, args):
        """Salir del interprete"""
        print("Hasta pronto. ", end="")
        return(True)

    def preloop(self):
        '''La sesion de trabajo ha comenzado'''
        print("La sesion de trabajo ha comenzado")

    def postloop(self):
        '''La sesion de trabajo ha finalizado'''
        print("La sesion de trabajo ha finalizado")

if __name__ == '__main__':
    interprete = Comandos()
    interprete.cmdloop(intro="Bienvenido")

```

### Atributos de la clase Cmd

La clase **Cmd** incorpora algunos atributos que permiten configurar un intérprete de comandos:

- **Cmd.identchars**. Define cadena con los caracteres que se pueden utilizar en la escritura de comandos.
- **Cmd.intro**. El texto asignado es utilizado para mostrar un mensaje inicial.
- **Cmd.lastcmd**. El comando asignado se ejecutará cuando se presione la tecla **[Return]** si previamente no se introdujo uno válido.
- **Cmd.prompt**. El texto asignado se mostrará en el lugar donde se introducen los comandos.
- **Cmd.doc\_header**. Cabecera a mostrar si la ayuda tiene una sección de comandos documentados.
- **Cmd.misc\_header**. Cabecera a mostrar si la ayuda tiene una sección miscelánea. La sección miscelánea la constituyen otros métodos distintos a los **do\_\***() que tienen su correspondientes métodos de ayuda **help\_\***()
- **Cmd.undoc\_header**. Cabecera a mostrar si la ayuda tiene una sección de comandos no documentados.
- **Cmd.ruler**. Carácter para subrayar los encabezados. Por defecto, es el signo **"=**".

```

import cmd

class Comandos(cmd.Cmd):
    """Interprete de comandos"""
    prompt = "Introduza un comando: "
    identchars = "comandoslrhep123"
    lastcmd = "comando2"
    intro = "Hola, buenos días"
    doc_header = "Ayuda de comandos documentados"
    undoc_header = "Ayuda de comandos no documentados"
    misc_header = "Ayuda de otros metodos"
    ruler = "-"

    def do_comando1(self, args):
        """Ayuda del comando1"""
        print("comando1 se ha ejecutado")

    def do_comando2(self, args):
        """Ayuda del comando2"""
        print("comando2 se ha ejecutado")

    def do_comando3(self, args):
        print("comando3 se ha ejecutado")
        self.metodo1()

    def do_salir(self, args):
        """Salir del interprete"""
        print("Hasta pronto")
        return(True)

    def metodo1(self):
        """metodo1"""
        print("metodo1 se ha ejecutado")

    def help_metodo1(self):
        print("ayuda del metodo1")

    def default(self, args):

```

```

        print("Error: comando inexistente")

if __name__ == '__main__':
    interprete = Comandos()
    interprete.cmdloop()

```

### Analizar comandos que incluyen argumentos

Para finalizar, un ejemplo que acepta comandos con argumentos para mostrar de forma práctica el modo de analizar este tipo de comandos.

En este caso la clase **Comandos** incorpora dos métodos para sumar (**suma**) y multiplicar (**mult**) una lista de números enteros (como mínimo 2). Los argumentos se deben escribir a continuación del comando separados por espacios en blanco. Cuando la lista de argumentos se analice, se extraerán de ella los números enteros y se omitirá otro tipo de información (cadenas, números con decimales y otros caracteres).

Ejemplos de comandos válidos:

- suma 23 12 34 450
- suma 23 abc 12 qq 12.5
- mult 1 12 2

```

#!/usr/bin/env python
# -*- coding: utf-8 -*-
#

import cmd
from functools import reduce

class Comandos(cmd.Cmd):
    """Suma y multiplica números enteros"""
    prompt = "Introduzca un comando: "
    doc_header = "Comandos disponibles"
    ruler = "-"

    def do_suma(self, args):
        """Suma los argumentos que sean enteros. Ej.: suma 12 13"""
        enteros = obtener_enteros(args)
        if len(enteros) > 1:
            print('Total', sum(enteros))
        else:
            print('La operación requiere al menos dos números enteros')

    def do_mult(self, args):
        """Multiplica argumentos que sean enteros. Ej.: mult 12 13"""
        enteros = obtener_enteros(args)
        if len(enteros) > 1:
            print('Total', reduce(lambda x,y: x*y, enteros))
        else:
            print('La operación requiere al menos dos números enteros')

    def do_salir(self, args):
        """Salir"""
        print("¡Hasta pronto!")
        return(True)

    def obtener_enteros(args):
        """Divide cadena de argumentos y devuelve números enteros"""
        argumentos = args.split()
        enteros = list(filter(lambda x: x.isnumeric(), argumentos))
        enteros = list(map(int, enteros))
        return(enteros)

if __name__ == '__main__':
    Comandos().cmdloop()

```

[Ir al índice del tutorial de Python](#)

Publicado por Pherkad en [14:22](#)



Etiquetas: [Python3](#)

[Entrada más reciente](#)

[Inicio](#)

[Entrada antigua](#)

2014-2020 | Alejandro Suárez Lamadrid y Antonio Suárez Jiménez, Andalucía - España  
. Tema Sencillo. Con la tecnología de [Blogger](#).