

# Curso Python en 8 clases

## Clase 5: CLI, templates y excepciones

Autor: Sebastián Bassi



Versión: 1.0

Licencia: Creative Commons BY-NC-SA 2.5. ([ver texto completo](#))



### CLI: Línea de comandos

#### Consideraciones básicas

Shebang: Primera línea que se usa en \*nix para que el SO sepa que interprete ejecutar.

Genérico:

```
#!/usr/bin/env python
```

Esta línea invoca al intérprete disponible "por defecto". Para especificar alguna versión o instalación de Python en particular:

```
#!/usr/bin/python
#!/usr/bin/python2.6
```

Además el archivo tiene que estar marcado como ejecutable (`chmod a+x archivo.py`). **Tener en cuenta estas instrucciones al correr un script en un server.**

Otra instrucción para incluir al principio de un script es el "encoding", útil cuando se incluyen símbolos no ASCII en el código fuente:

```
# -*- coding: iso-8859-15 -*-
```

o

```
# -*- coding: UTF-8 -*-
```

#### ¡Atención!

El encoding del documento es para el archivo del programa. Sirve para que el interprete sepa el encoding del código fuente. No tiene relación con el o los encoding que leerá o escribirá el programa.

#### Argumentos via CLI (modo simple)

La manera mas básica es usando `sys.argv`:

```
import sys
print sys.argv
```

**sys.argv** contiene una lista de strings, cada str es un argumento pasado en la línea de comandos, salvo el primer elemento que es el nombre del programa.

Código de ejemplo:

```
#!/usr/bin/python

import sys
def main():
    print 'Hola', sys.argv[1]

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

Uso:

```
sbassi@sbassi-msi:~/cursoPy$ programa.py Seba
Hola Seba
```

## ***Entrada via tuberías (pipelines)***

### ***Definición***

"En informática, una tubería (pipe o '|') consiste en una cadena de procesos conectados de forma tal que la salida de cada elemento de la cadena es la entrada del próximo." (Fuente: [http://es.wikipedia.org/w/index.php?title=Tuber%C3%ADa\\_\(inform%C3%A1tica\)&oldid=29870368](http://es.wikipedia.org/w/index.php?title=Tuber%C3%ADa_(inform%C3%A1tica)&oldid=29870368))

Para leer datos entrados con pipe, usar **sys.stdin** que es el equivalente a una referencia a un archivo (*filehandle*).

El siguiente código lee un archivo que entra via STDIN:

```
#!/usr/bin/python

import sys

def main():
    print 'El contenido del archivo es:'
    for linea in sys.stdin:
        print linea,

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

Una manera de ejecutarlo:

```
sbassi@sbassi-msi:~/cursoPy$ programa.py < 1.txt
```

## Argumentos via CLI (avanzado)

Para tener un mejor control de los parametros que pueden entrarse via línea de comandos, existe el módulo *optparse*. Este módulo tiene soporte de argumentos, opciones y ayuda. Ayuda a que no tenemos que preocuparnos por la posición de los argumentos y se adapta al estilo habitual de los programas de línea de comando usados en \*nix.

### Terminología (en el contexto de *optparse*)

**Argumentos:** Strings ingresados en la línea de comandos, recuperables via `sys.argv[1:]`.

**Opciones:** Un argumento que es optativo. Normalmente tienen un guión (-) seguido de una letra, o 2 guiones (--) seguida de una palabra.

**Argumento de opción:** Un argumento que sigue a una opción. Ejemplo:

```
-f archivo.txt
--file archivo.txt
```

**Argumento posicional:** Los argumentos que quedan luego de que se procesan las opciones. Son argumentos que los programas requieren de manera obligatoria.

### Uso de *optparse*

Como primera medida hay que importar e instanciar *OptionParser*:

```
from optparse import OptionParser
parser = OptionParser()
```

Luego hay que agregar opciones con el método *add\_option*, por ejemplo:

```
parser.add_option("-i", "--input", dest="input_file",
                  action="store", type="string",
                  help="Input file. ")
```

Los primeros argumentos son las cadenas que serán reconocidas en la línea de comandos ("-i" e "--input" en este caso). El argumento con clave *dest* indica el nombre con el que será reconocido en el programa. El argumento con clave *help* es la línea de ayuda que el usuario verá en la línea de comando asociada con esa opción. *action* es opcional (por defecto tiene el valor de "store") e indica como guardar el valor ingresado. En este caso (cuando *action*="store") se guarda directamente el valor ingresado. Si *action*="store\_const", se guarda una constante que tiene que ser especificada con el argumento "const". Un caso particular de "store\_const" es "store\_true" que guarda "true" en la variable asociada en *dest*. Otro valor posible para *action* es "append", que agrega un elemento a una lista de destino. El argumento *type* indica el tipo de datos que almacena, por defecto es "string".

Mas ejemplos de *add\_option*:

```
parser.add_option("-q", "--quiet",
                  action="store_const", const=0, dest="verbose")
parser.add_option("-v", "--verbose",
                  action="store_const", const=1, dest="verbose")
parser.add_option("--noisy",
                  action="store_const", const=2, dest="verbose")
parser.add_option("-t", "--tracks", action="append", type="int")
```

Una vez que se definen las opciones, hay que hacer el procesamiento:

```
(options, args) = parser.parse_args()
```

Ejemplo de código integrador:

```
#!/usr/bin/env python

import sys
from optparse import OptionParser
import string

usage = "usage: %prog [password_file] [dictionary_file]\n"
usage += "or %prog [options] [password_file]\n\n"
usage += "Type %prog -h for more information"
parser = OptionParser(usage, version="%prog 1.0")
parser.add_option("-i", "--input", dest="input_file",
                  help="Input file. ")
parser.add_option("-d", "--dictionary", dest="dict_file",
                  help="Use a dictionary file. ")
parser.add_option("-a", "--aspell", dest="aspell",
                  help="Use aspell dictionary. ")
(opts, args) = parser.parse_args()
if opts.input_file:
    fh = open(opts.input_file, 'U')
elif args:
    fh = open(args[0], 'U')
elif sys.stdin:
    fh = sys.stdin
else:
    parser.error("Enter an input file")

if len(args)==2:
    fd = open(args[1], 'U')
elif opts.dict_file:
    fd = open(opts.dict_file, 'U')
elif opts.aspell:
    try:
        import aspell
    except:
        parser.error("Please install aspell-python")
else:
    parser.error('Enter a dictionary file or aspell dictionary')
```

Mas información: <http://docs.python.org/library/optparse.html>

## Salida con plantillas (templates)

Hay varios sistemas de templates disponibles para Python (PML, Spyce, Django template system, Mako, Jinja 2 y Cheetah). En este curso usamos **Cheetah**.

### Cheetah

**Cheetah** no es parte de la biblioteca estándar, por lo que hay que descargarlo e instalarlo por separado. Usando easy\_install:

```
# easy_install cheetah
```

O usando el gestor de paquetes:

```
# apt-get install python-cheetah
```

### Uso de Cheetah

Hay que importar e instanciar el objeto *Template*. Los parametros son: Referencia al archivo de plantilla y un argumento con clave *searchList* que tiene una lista con un elemento que es un diccionario con las variables que serán reemplazadas en los lugares señalados en la plantilla.

Ejemplo de programa que usa Cheetah:

Con el mismo código solo hay que cambiar una línea para que la salida sea un CSV, un HTML o un SQL. Un ejemplo de separación de lógica con presentación.

```
import random
from Cheetah.Template import Template

# Hacer datos al azar (and call it 'data')
data = []
for x in range(50):
    point = random.randint(5,200)
    data.append((x+1,point))

# Guardar como archivo CSV
dataout = Template(open('data_csv').read(), searchList=[{'data': data}])
open('data.csv','w').write(str(dataout))
# Guardar como tabla HTML
dataout = Template(open('data_html').read(), searchList=[{'data': data}])
open('data.html','w').write(str(dataout))
# Guardar como query SQLite
dataout = Template(open('data_sql').read(), searchList=[{'data': data}])
open('data.sql','w').write(str(dataout))
```

Ejemplo de archivos de plantillas:

data\_csv file:

```
#for $d in $data:
$d[0],$d[1]
#end for
```

data\_html file:

```
<html><head></head>
<body>
<table border=1>
<tr><th>Order</th><th>Value</th></tr>
#for $d in $data:
<tr><td>$d[0]</td><td>$d[1]</td></tr>
#end for
</table>
</body></html>
```

data\_sql file:

```
BEGIN TRANSACTION;
#for $d in $data:
INSERT INTO "mytable" VALUES($d[0],$d[1]);
#end for
COMMIT;
```

A tener en cuenta:

- En Cheetah los templates son archivos de texto con “marcas”.
- La marca insertar código es “#”.
- Las variables van precedidas con “\$”.
- Como no se puede indentar, hay que marcar el fin de los ciclos de manera explícita (#end if, #end for).
- Se puede insertar un template en otro.
- Los templates pueden ser “compilados”.

### **Compilando plantillas**

Comando:

```
$ cheetah-compile nombre_plant
```

Esto genera un archivo tipo nombre\_plant.py. Este archivo debe copiarse a un directorio donde pueda ser accedido como módulo (ver clase 4). Este módulo debe ser importado en el programa:

```
from index_template import index_template as it
```

Luego se usa de esta forma:

```
dataout = it(searchList=[DICT])
```

(donde DICT es un diccionario con todas las variables a sustituir).

### **Lenguaje de plantillas**

Comentarios:

```
## Una línea
## Multiples líneas *#
```

Incluir archivos:

Parseados: #include nombre\_archivo Sin parsear: #include raw nombre\_archivo

Importar módulos:

```
#import
#from xxxx import xxxxx
```

Variables:

Con “\$” o con \${}.

Ejemplo:

```
<title>$page_title</title>
```

Variables auxiliares:

```
#set $var = ''
```

Ejemplo:

```
#set $url = 'http://www.dominio.com'
```

## Manejo de errores

Estrategias de manejo de errores:

1. Ignorar la posibilidad de error y asumir que todo funcionará como planeado.
2. Codificar de manera que se eviten todos los errores (LBYL)
3. Asumir la posibilidad de error y actuar a consecuencia (EAFP).

Ej. de 1-

```
fh = open('myfile.csv')
line = fh.readline()
fh.close()
value = line.split('\t')[0]
fw = open('other', "w")
fw.write(str(int(value)*.2))
fw.close()
```

Ej de 2-

```
import os
while True:
    iname = raw_input("Enter input filename: ")
    oname = raw_input("Enter output filename: ")
    if os.path.exists(iname):
        fh = open(iname)
        line = fh.readline()
        fh.close()
        if "\t" in line:
            value = line.split('\t')[0]
            if os.access("/home/sb/"+oname, os.W_OK)==0:
                fw = open("/home/sb/"+oname, "w")
                if value.isdigit():
                    fw.write(str(int(value)*.2))
                    fw.close()
                    break
                else:
                    print("It can't be converted to int")
            else:
                print("Output file is not writable")
        else:
            print("There is no TAB. Check the input file")
    else:
```

```
print("The file doesn't exist")
```

El problema de este método es que el tratamiento de errores dificulta el seguimiento del hilo del programa.

Ejemplo general de estrategia EAFP:

```
try:
    code block 1
    # ...some error prone code...
except:
    code block 2
    # ...do something with the error...
[else:
    code block 3
    # ...to do when there is no error...
finally:
    code block 4
    #...some clean up code...]
```

Ejemplo:

```
try:
    print 0/0
except:
    print("Houston, we have a problem...")
```

## Como distinguir excepciones

Atrapando "genéricamente":

```
d = {"A": "Adenine", "C": "Cistine", "T": "Timine", "G": "Guanine"}
try:
    print d[raw_input("Ingrese letra: ")]
except:
    print "No existe ese nucleotido"
```

Distinguiendo entre EOF y clave errónea.

```
d = {"A": "Adenine", "C": "Cistine", "T": "Timine", "G": "Guanine"}
try:
    print(d[raw_input("Enter letter: ")])
except EOFError:
    print("Good bye!")
except KeyError:
    print("No such nucleotide")
```

Programa del "ejemplo 2" con manejo de excepciones:

```
import os, errno
while True:
    iname = raw_input("Enter input filename: ")
    oname = raw_input("Enter output filename: ")
    try:
```



```

fh = open(iname)
line = fh.readline()
value = str(int(line[:line.index("\t")])*2)
fw = open("/home/sb/"+oname,"w")
fw.write(value)
except IOError, (errno,errmsg):
    if errno==errno.EACCES:
        print("Permission denied")
    elif errno==errno.ENOENT:
        print("No such file")
except ValueError, strerror:
    if "substring not found" in strerror.message:
        print("There is no tab")
    elif "invalid literal for int" in strerror.message:
        print("The value can't be converted to int")
else:
    fw.close()
    fh.close()
    break

```

## Provocando excepciones

No hace falta esperar que ocurran ya que pueden ser invocadas por el programador. Esta función avg tendrá problemas con una lista vacía:

```

>>> def avg(num):
...     return sum(num)/len(num)
...
>>> avg([])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in avg
ZeroDivisionError: integer division or modulo by zero

```

Este error no es fácilmente comprensible si no se conoce el código de la función. Levantando apropiadamente un error podemos hacer que quede claro el problema:

```

>>> def avg(num):
...     if not num:
...         raise ValueError("Ingrese al menos un elemento")
...     return sum(num)/len(num)
...
>>> avg([])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in avg
ValueError: Ingrese al menos un elemento

```

Se podría imprimir un resultado sin levantar una excepción, pero esto iría contra el principio "pytónico" de "los errores no deben ocurrir sin que se noten, deben ser explícitos".

Mas información: <http://docs.python.org/tutorial/errors.html>