Salida y Ficheros

Entrada/Salida y Ficheros Los programas serían de muy poca utilidad si no fueran capaces de interaccionar con el usuario. En los programas vistos hasta ahora utilizamos la palabra clave *print* para mostrar mensajes por pontalla y, también, la función **input()** para obtener información del usuario.

Sumario

Entrada estándar

- La forma más sencilla de obtener información por parte del usuario es mediante la función **input()**. Esta función toma como parámetros un *string*, que será el texto a mostrar al usuario pidiendo la entrada y, luego, devolverá una cadena con los caracteres introducidos por el usuario, hasta que este pulse la tecla **Enter**.

  Veamos un ejemplo sencillo:

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
nombre = input("Introduce tu nombre: ")
print("Encantado de conocerce {}'.format(nombre))
```

- Si necesitáramos un entero como entrada en lugar de una cadena, por ejemplo, podríamos utilizar la función int para convertir la cadena a

entero, aunque sería conveniente tener en cuenta que puede lanzarse una excepción si lo que introduce el usuario no es un número.

```
# !/usr/bin/python3
# -*- coding: utf-8 -*-

try:
    edad = int(input("Edad: "))
    dias = edad * 365
    print('Has vivido {} días'.format(dias))
except ValueError:
    print('Escribe un número entero')
```

Parámetros de línea de comando

Para recibir datos introducidos por el usuario también se pueden utilizar **parámetros** a la hora de llamar al programa (tal y como estamos acostumbrados al utilizar la línea de comandos).

Por ejemplo:

```
$ media.py 12 5 10 27
```

En este caso los parámetros serían los números: 12, 5, 10 y 27, a los que se puede acceder a traves de la lista **sys.argv**.
Antes de poder utilizar esta variable tendremos que importar el módulo **sys**, siendo cada uno de los parámetros llamado de la forma: sys.argv[1], sys.arg[2], etc.
El parámetro sys.argv[0] contiene siempre el nombre del programa (en este caso media.py).

```
# !/usr/bin/python3
# -*- coding: utf-8 -*-

import sys
## Variables
nelementos = len(sys.argv)
suma = 0
media = 0
```

```
## Hacemos la suma de todos los elementos
#El elemento 0 lo saltamos
#pues es el nombre del script
for i in range(1, nelementos):
    suma += float(sys.argv[i])
# y luego, con ella, la media
media = float(suma / (nelementos - 1))
## Mostramos por pantalla los resultados
print('El número de elementos introducidos es {}'.format(nelementos-1))
print('\nLa suma de todos los números es: {}'.format(suma))
print('\nLa media de los números es: {}'.format(media))
```

Existen módulos, como argparse, que facilitan el trabajo con los argumentos de la línea de comandos...

Parámetros de línea de comando utilizando el módulo argparse

Utilizamos el módulo **argparse** del siguiente modo, donde vemos que, incluso, podemos forzar a que uno de los argumentos sea de un tipo determinado de datos.

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

import argparse

parser = argparse.ArgumentParser(description='Paso de parámetros')
#El parámetro 1 es obligatorio que sea un entero
parser.add_argument("-p1", dest="param1", help="parametro1 - entero", type=int)
#No forzamos ningún tipo para el parámetro 2
parser.add_argument("-p2", dest="param2", help="parametro2")
params = parser.parse_args()
print(params.param1)
print(params.param2)
```

Ahora lo ejecutamos del siguiente modo

```
#Ver la ayuda
$ python3 prueba.py -h
#Ejecutar el comando
$ python3 prueba.py -p1 1 -p2 dos
```

Salida estándar

La forma más sencilla de mostrar algo en la salida estándar es mediante el uso de la función **print()**, como ya se fue utilizando en varias ocasiones.
· En su forma más básica, a la función *print()* se le pasa una cadena, que se mostrará en la salida estándar (normalmente la pantalla):

```
>>> print("Hola mundo")
Hola mundo
```

· Después de imprimir la cadena pasada como parámetro el puntero se sitúa en la siguiente línea de la pantalla.
· Si se quiere forzar que se salte a una nueva línea se utilizará el carácter especial **\n** (también se puede utilizar otros caracteres especiales como **\t**, este introducirá un tabulado en el texto).
· Si, por el contrario, se desea que la siguiente impresión se realice en la misma línea hay que configurar el parámetro de la función **print()** del siguiente modo: **end=""**.

```
for i in range(5):
    print(i, end='')
```

· Si queremos concatenar texto podemos utilizar una coma **(,)** o un signo más **(+)**. La diferencia está en que con la coma se añade automáticamente un espacio entre los textos concatenados, con el signo más esto no ocurre:

```
>>> print("Hola", "mundo")
Hola mundo
######
>>> print("Hola" + "mundo")
Holamundo
```

· Al utilizar el operador **+** tendríamos que convertir antes cada argumento en una cadena (si no lo es ya).

```
>>> print("Cuesta", 3, "euros")
Cuesta 3 euros
#####
>>> print("Cuesta " + 3 + " euros")
TypeError: Can't convert 'int' object to str implicitly
#####
>>> print("Cuesta " + str(3) + " euros")
Cuesta 3 euros
```

· Utilizando la función format() podemos darle forma al texto mostrado por pantalla de un modo más versátil.

```
#Trabajo con variables
saludo = "Hola"
lugar = "Mundo"
print('\n{} {}'.format(saludo, lugar))
##Indicando la posición del elemento
print('\n{0} {1}'.format(saludo, lugar))
##Indicando, además, el número de caracteres que ocupa cada string
####(10 primero, 8 segundo).
print('{0:10s} {1:8s}'.format(saludo, lugar))
```

En este ejemplo estamos indicando {Elemento:Número_caracteresTipo_elemento} el tipo de elemento a mostrar puede ser:

Especificador	Formato
s	Cadena
d	Entero
o	Octal
x	Hexadecimal
f	Real

_ En el caso de los reales es posible indicar la precisión a utilizar precediendo la **f** de un punto seguido del número de decimales que queremos mostrar:

```
from math import pi
print('Pi con cuatro decimales: {0:.4f}'.format(pi))
```

· Para representar datos en formato tabla podemos hacerlo de dos modos:

- Utilizando el método de los objetos string *str.rjust()*. Y, ojo, los espacios que existen entre las columnas son los que añadimos entre las comas.

```
for x in range(1,11):
    print(repr(x).rjust(2), repr(x**2).rjust(3), \
          repr(x**3).rjust(4))
```

- Definiendo las columnas de la tabla y el ancho de cada una de ellas:

```
#Crear una tabla de enteros (d)
for x in range(1,11):
    print('{0:2d} {1:4d} {2:4d}'.format(x, x * x, x * x * x))
    #Indicamos el tipo de elemento y el número de caracteres
```

· Si queremos ajustar números añadiendo ceros a la izquierda de, como mínimo, un tamaño determinado, utilizamos el método del elemento string *str.zfill()*. Fijarse que si se pide una salida de menos caracteres que el número original, la salida es el número original:

```
print('12'.zfill(5))
print('-3.14'.zfill(7))
print('3.14159265359'.zfill(7))
################################
#00012
```

```
#-003.14
#3.14159265359
```

· Para mostrar datos de una tupla, lista o diccionario también es interesante el siguiente método:

```
tabla = {'Pepe': 5.5, 'Andrea': 7.25, 'Juan': 6}

for nombre, nota in tabla.items():
    print('{0:7} ==> {1:2.2f}'.format(nombre, nota))
```

Salida:

```
Juan    ==> 6.00
Andrea  ==> 7.25
Pepe    ==> 5.50
```

Archivos

Los ficheros en Python son **objetos de tipo *file*** creados mediante la función ***open*** (abrir).
Esta función toma como parámetros una cadena con la ruta del fichero a abrir, una cadena opcional indicando el modo de acceso (si no se especifica se accede en modo lectura) y, por último, un entero opcional para especificar un tamaño de *buffer* distinto del utilizado por defecto.
El modo de acceso puede ser cualquier combinación lógica de los siguientes modos:

- **r** : *read*, lectura. Abre el archivo en modo lectura. El archivo tiene que existir previamente, en caso contrario se lanzará una excepción de tipo *IOError*.
- **w** : *write*, escritura. Abre el archivo en modo escritura. Si el archivo no existe se crea. Si existe, sobreescribe el contenido.
- **x** : abre el archivo en modo escritura. Si el archivo no existe, se crea; si no, se genera una excepción del tipo *FileExistsError*.
- **a** : *append*, añadir. Abre el archivo en modo escritura. Se diferencia del modo **w** en que en este caso no se sobreescribe el contenido del archivo, sino que se comienza a escribir al final del archivo.
- **b** : *binary*, binario.
- **t** : *text mode* (predeterminado), el archivo se abre en modo texto, por lo que la salida será una cadena de texto, así como la entrada.
- **+** : permite lectura y escritura simultáneas (r+, w+, x+ y a+).
- **U** : *universal newline*, saltos de línea universales. Permite trabajar con archivos que tengan un formato para los saltos de línea que no coincide con el de la plataforma actual.

Ejemplos:

```
#Abrir archivo en modo lectura
fr = open(?archivo.txt?, ?r?)
#Abrir archivo en modo escritura
fw = open(?archivo.txt?, ?w?)
```

Tras crear el objeto que representa nuestro archivo mediante la función *open* podremos realizar operaciones de lectura/escritura.
Una vez terminemos de trabajar con el archivo debemos cerrarlo utilizando el método ***close***.

- **Lectura de archivos**

Para la lectura de archivos se utilizan los métodos ***read***, ***readline*** y ***readlines***.

- El método *read* devuelve una cadena con el contenido del archivo o bien el contenido de los primeros **n bytes**, si se especifica el tamaño máximo a leer.

```
#Contenido completo
completo = fr.read()
#Los 512 primeros bytes
parte = fr.read(512)
```

- El método *readline* sirve para leer las líneas del fichero una por una. Es decir, cada vez que se llama a este método, se devuelve el contenido del archivo desde el puntero hasta que se encuentra un carácter de nueva línea, incluyendo este carácter.

```
...
while True:
    linea = fr.readline()
    if not linea:
        break
    print(linea)
```

```
...
```

- El método *readlines* funciona leyendo todas las líneas del archivo y devolviendo una lista con las líneas leídas.

Cuando se termina de utilizar un archivo para leerlo, o escribirlo, siempre hay que cerrarlo:

```
f = open('/proc/cpuinfo')
for line in f:
    if (line.strip().startswith('model name')):
        model_name = line.split(':')[1].strip()
        print("Microprocesador : ", model_name)
f.close()
```

Un modo más cómodo de abrir y cerrar un archivo es utilizando la instrucción *with*, ésta cierra siempre el archivo al terminar aunque se produzca una excepción:

```
with open('/proc/cpuinfo') as f:
    for line in f:
        if (line.strip().startswith('model name')):
            model_name = line.split(':')[1].strip()
            print("Microprocesador : ", model_name)
```

- split y strip en python.org

- **Escritura de archivos**

  Para la escritura de archivos se utilizan los métodos ***write*** y ***writelines***.

  - El método *write* funciona escribiendo en el archivo una cadena de texto que toma como parámetro.

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

import urllib.request

##Guardar el html de una web en un archivo
respuesta = urllib.request.urlopen('http://manuais.iessanclemente.net/')
html = respuesta.read()
f1 = open(".\manuais.html", "w")
f1.write(repr(html))
f1.close()
```

  - El método *writelines* toma como parámetro una lista de cadenas de texto indicando las líneas que queremos escribir en el fichero.

- **Mover el puntero de lectura/escritura**

  Hay situaciones en las que nos puede interesar mover el puntero de lectura/escritura a una posición determinada del archivo. Por ejemplo, si queremos empezar a escribir en una posición determinada y no al final o al principio del archivo.

  - El método *seek* toma como parámetro un número positivo o negativo a utilizar como desplazamiento. También se puede utilizar un segundo parámetro para indicar desde dónde se quiere hacer el desplazamiento:

  · **0** indica que el desplazamiento se refiere al principio del fichero (comportamiento por defecto).
  · **1** indica la posición actual.
  · **2** indica el final del fichero.

  - El método *tell()* determina la posición en la que se encuentra actualmente el puntero. Este método devuelve un entero indicando la distancia en bytes desde el principio del fichero.

-- Volver