Programación en Python (I)

Miguel Ortuño
Escuela de Ingeniería de Fuenlabrada
Universidad Rey Juan Carlos

Noviembre de 2023

Contenidos I

Contenidos II

- Ficheros
- Excepciones
- Fechas
- Cadenas binarias

2. Librerías de Python
   - Librería sys
   - Librería subprocess
   - Librerías os, shutil
   - Librerías pickle: Persistencia

El Lenguaje Python

- Lenguaje *de autor* creado por Guido van Rossum en 1989
- Muy relacionado originalmente con el S.O. *Amoeba*
- Disponible en Unix, Linux, macOS, Windows,
- Libre
- Lenguaje de Script Orientado a Objetos (no muy puro)
- Muy alto nivel
- Librería muy completa

- Verdadero lenguaje de propósito general
- Sencillo, compacto
- Sintaxis clara
- Interpretado => Lento
- Ofrece persistencia
- Recolector de basuras
- Muy maduro y muy popular
- Aplicable para software de uso general

Programa python

```
for x in xrange(1000000):
          print x
```

Su equivalente Java

```
public class ConsoleTest {
       public static void main(String[] args) {
              for (int i = 0; i < 1000000; i++) {
                                  System.out.println(i);
                                                  }
                           }
}
```

Programa python

```
for i in xrange(1000):
          x={}
          for j in xrange(1000):
              x[j]=i
              x[j]
```

Su equivalente Java

```
import java.util.Hashtable;
public class HashTest {
    public static void main(String[] args) {
       for (int i = 0; i < 1000; i++) {
           Hashtable x = new Hashtable();
           for (int j = 0; j < 1000; j++) {
               x.put(new Integer(i), new Integer(j));
               x.get(new Integer(i));
           }
       }
    } }
```

Librerías

Python dispone de librerías *Nativas* y *Normalizadas* para

- Cadenas, listas, tablas hash, pilas, colas
- Números Complejos
- Serialización, Copia profunda y Persistencia de Objetos
- Regexp
- Unicode, Internacionalización del Software
- Programación Concurrente
- Acceso a BD, Ficheros Comprimidos, Control de Cambios...

Librerías relacionadas con Internet:

- CGIs, URLs, HTTP, FTP,
- pop3, IMAP, telnet
- Cookies, Mime, XML, XDR
- Diversos formatos multimedia
- Criptografía

La referencia sobre todas las funciones de librería podemos encontrarlas en la documentación oficial, disponible en el web en muchos formatos. Basta con localizar en cualquier buscador la *python standard library*

Inconvenientes de Python

Además de su velocidad limitada y necesidad de intérprete (Como todo lenguaje interpretado)

- No siempre compatible hacia atrás
- Uniformidad.
  Ej: `función len()`, `método items()`
- Algunos aspectos de la OO
  Python is a hybrid language. It has functions for procedural programming and objects for OO programming. Python bridges the two worlds by allowing functions and methods to interconvert using the explicit "self" parameter of every method def. When a function is inserted into an object, the first argument automagically becomes a reference to the receiver.
- ...

Versiones de python

- Python 1 (año 1991)
- Python 2 (año 2001)
  Incompatible con Python 1
- Python 3 (año 2009)
  Incompatible con Python 2. Versión problemática, durante la década de 2010 fue muy habitual seguir usando Python 2

El soporte para Python 2 finalizó oficialmente el 1 de enero de 2020. En la actualidad:

- Sigue habiendo mucho código antiguo en Python 2
- Sigue siendo recomendable que en cualquier sistema *python a secas* sea Python 2
- Deberíamos programar siempre en Python 3, indicando explícitamente que el intérprete es *python3*

El intérprete de python se puede usar

- En modo interactivo

```
koji@mazinger:~$ python3
Python 3.9.0 (default, Oct 27 2020, 14:13:35)
[Clang 11.0.0 (clang-1100.0.33.17)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hola, mundo")
Hola, mundo
>>> 3/2
1.5
```

- Mediante scripts

```
#!/usr/bin/env python3
# holamundo.py
print("hola mundo")     # esto es un comentario
euros=415
pesetas=euros*166.386
print("{} euros son {} pesetas".format(euros, pesetas))
```

La línea `#!/usr/bin/env python3` indica al S.O. dónde está la utilidad *env*, que buscará en el *path* el ejecutable *python3* y le pasaré el fuente para que lo ejecute

- Debe ser exactamente la primera línea
- No puede haber espacios entre la admiración y la barra

```
#Este ejemplo es doblemente incorrecto
#! /usr/bin/env python3
# ¡MAL!
```

En linux se puede especificar directamente la dirección del intérprete

```
#!/usr/bin/python3
print("Hola mundo")
```

Pero `env` es más general, permite que tengamos varios intérpretes en el sistema (algo muy normal en macOS)

Operadores

En orden de precedencia decreciente:

```
+x, -x, ~x    Unary operators
x ** y    Power
x * y, x / y, x % y    Multiplication, division, modulo
x + y, x - y    Addition, subtraction
x << y, x >> y    Bit shifting
x & y    Bitwise and
x | y    Bitwise or
x < y, x <= y, x > y, x >= y, x == y, x != y,
x <> y, x is y, x is not y, x in s, x not in s
                        Comparison, identity,
                        sequence membership tests
not x    Logical negation
x and y    Logical and
lambda args: expr            Anonymous function
```

Identificadores (nombre de objetos, de funciones...):

- Letras inglesas de 'a' a 'z', en mayúsculas o minúsculas. Barra baja '_' y números
- Sensible a mayúsculas/minúsculas

Naturalmente, en las cadenas de texto y en los comentarios podemos escribir cualquier carácter en la codificación habitual en la actualidad (utf-8)

Python es

- Dinámicamente tipado, (*dynamically typed*). No es estáticamente tipado (*statically typed*)
  Una variable puede cambiar su tipo, dinámicamente
- Fuertemente tipado, (*strongly typed*). No es débilmente tipado (*weakly typed*)
  Este concepto no es absoluto, decimos que ciertos lenguajes tienen tipado más fuerte o más débil que otros
  Si algún objeto, variable, método, función... espera cierto tipo de objeto/de dato:
  - Un lenguaje fuertemente tipado ha de recibir o bien exactamente ese tipo o bien uno muy parecido, de forma que pueda hacerse una conversión automática sin pérdida de información
    Obliga al programador a conversiones explícitas. Esto resulta rígido, tal vez farragoso, pero facilita la seguridad
  - Un lenguaje débilemente tipado, admite casi cualquier cosa. Esto resulta cómodo, flexible, potencialmente peligroso

En Python la declaración de objetos (variables) es implícita (no hay declaración explícita)

- Las variables "nacen" cuando se les asigna un valor
- Las variables "desaparecen" cuando se sale de su ámbito
- La declaración implícita de variables como en perl puede provocar resultados desastrosos

```
#!/usr/bin/perl
$sum_elementos= 3 + 4 + 17;
$media=suma_elementos / 3;    # deletreamos mal la variable
print $media;   # y provocamos resultado incorrecto
```

- Pero Python no permite referenciar variables a las que nunca se ha asignado un valor.

```
#!/usr/bin/env python3
sum_elementos= 3 + 4 + 17
media=suma_elementos / 3    # deletreamos mal la variable
print(media)   # y el intéprete nos avisa con un error
```

Funciones predefinidas

- `abs()` valor absoluto
- `float()` convierte a float
- `int()` convierte a int
- `str()` convierte a string
- `round()` redondea
- `input()` acepta un valor desde teclado

Sangrado y separadores de sentencias

- ¡En Python NO hay llaves ni `begin-end` para encerrar bloques de código! Un mayor nivel de sangrado indica que comienza un bloque, y un menor nivel indica que termina un bloque.
- Las sentencias se terminan al acabarse la línea (salvo casos especiales donde la sentencia queda "abierta": en mitad de expresiones entre paréntesis, corchetes o llaves).
- El carácter \ se utiliza para extender una sentencia más allá de una linea, en los casos en que no queda "abierta".
- El carácter : se utiliza como separador en sentencias compuestas. Ej.: para separar la definición de una función de su código.
- El carácter ; se utiliza como separador de sentencias escritas en la misma línea.

- La recomendación oficial es emplear 4 espacios para cada nivel de sangrado
  - *PEP-8 Style Guide for Python Code*
  - David Goodger, *Code Like a Pythonista: Idiomatic Python*
    Traducción al español:
    Programa como un Pythonista: Python Idiomático
- Emplear 8 espacios o emplear tabuladores es legal, con tal de que no lo mezclemos [1]

[1] En python 2 era necesario añadir la opción -tt para que el intérprete detectase este problema

Tipos de objeto

En python todo son objetos: cadenas, listas, diccionarios, funciones, módulos...

- En los lenguajes de scripting más antiguos como bash o tcl, el único tipo de datos es la cadena
- Los lenguajes imperativos más habituales (C, C++, pascal...) suelen tener (con variantes) los tipos: booleano, carácter, cadena, entero, real y matriz
- Python tiene booleanos, enteros, reales y cadenas. Y además, cadenas unicode, listas, tuplas, números complejos, diccionarios, conjuntos...
  - En terminología python se denominan *tipos de objeto*
  - Estos tipos de objeto de alto nivel facilitan mucho el trabajo del programador

En python es muy importante distinguir entre

- Objetos inmutables: Números, cadenas y tuplas
  - Se pasan a las funciones por valor
  - Si están declarados fuera de una función son globales y para modificarlos dentro de la función, es necesaria la sentencia *global*
- Objetos mutables: Todos los demás
  - Se pasan a las funciones por referencia
  - Si están declarados fuera de una función son globales, pero no hace falta la sentencia *global* para modificarlos dentro de la función, puesto que pueden ser modificados a través de sus métodos

Comprobación de tipos

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

if isinstance("Bla blá", str):
    print("ok, es una cadena")
else:
    print("no es una cadena")
```

Tipos de objeto habituales:

```
bool
int
float
list
str
dict
tuple
```

Cadenas

- No existe tipo `char`
- Podemos emplear indistintamente la comilla simple o la doble (con tal de que el cierre coincida con la apertura)

```
print("hola")
print('hola')
print('me dijo "hola"')
```

Lo que resulta más legible que escapar los caracteres especiales

```
print('me dijo \'hola\'')
```

- Se permiten caracteres especiales, p.e. nueva línea

```
print("hola\nque tal")
```

- Cadenas crudas: esto imprime la barra y la n, literalmente

```
print(r"""hola\nque tal""")
```

- El operador + concatena cadenas, y el * las repite un número entero de veces
- Para concatenar una cadena con un objeto de tipo diferente, podemos convertir el objeto en cadena mediante la función `str()`

```
>>> gamma=0.12
>>> print "gamma vale "+str(gamma)
gamma vale 0.12
```

- Se puede acceder a los caracteres de cadenas mediante índices y rodajas como en las listas
- Las cadenas son inmutables. Sería erróneo `a[1]=...`

Listas

- Tipo de datos predefinido en Python, va mucho más allá de los arrays
- Es un conjunto *indexado* de elementos, no necesariamente homogéneos
- Sintaxis:Identificador de lista, mas índice entre corchetes
- Cada elemento se separa del anterior por un carácter ,

```
#!/usr/bin/env python3
# listas01.py
a=['rojo','amarillo']
a.append('verde')
print(a)      # ['rojo', 'amarillo', 'verde']
print(a[2])   # verde
print(len(a)) # 3

b=['uno',2, 3.0]  # Lista heterogénea
```

- El primer elemento tiene índice 0.
- Un índice negativo accede a los elementos empezando por el final de la lista. El último elemento tiene índice -1.
- Pueden referirse *rodajas* (*slices*) de listas escribiendo dos índices entre el carácter :
- La rodaja va desde el *primero, incluido*, al *último, excluido*.
- Si no aparece el primero, se entiende que empieza en el primer elemento (0)
- Si no aparece el segundo, se entiende que termina en el último elemento (incluido).

```
#!/usr/bin/env python3
# listas02.py
a=[0,1,2,3,4]
print(a)      # [0, 1, 2, 3, 4]
print(a[1])   # 1
print(a[0:2]) # [0,1]
print(a[3:])  # [3,4]
print(a[-1])  # 4
print(a[:-1]) # [0, 1, 2, 3]
print(a[:-2]) # [0, 1, 2]
```

La misma sintaxis se aplica a las cadenas

```
a="niño"
print(a[-1])  # o
```

- `append()` añade un elemento al final de la lista
- `insert()` inserta un elemento en la posición indicada

```
#!/usr/bin/env python3
# listas03.py
a=['cero', 'uno']
a.append('dos')
print(a)       # ['cero', 'uno', 'dos']
a.insert(1, 'cero cinco')
print(a)       # ['cero', 'cero cinco', 'uno', 'dos']
```

- `index()` busca en la lista un elemento y devuelve el índice de la primera aparición del elemento en la lista. Si no aparece se eleva una excepción.
- El operador `in` devuelve *true* si un elemento aparece en la lista, y *false* en caso contrario.

```
>>> lista=['cero', 'uno', 'dos']
>>> print(lista)
['cero', 'uno', 'dos']
>>> lista.index('uno')
1
>>> 'doce' in lista
False
>>> print(lista.index('doce'))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 'doce' is not in list
```

- `remove()` elimina la primera aparición de un elemento en la lista. Si no aparece, eleva una excepción.
- `pop()` devuelve el último elemento de la lista, y lo elimina. (Pila)
- `pop(0)` devuelve el primer elemento de la lista, y lo elimina. (Cola)

```
#!/usr/bin/env python3
# listas04.py
a = ['rojo', 'verde', 'azul', 'negro']
a.remove('negro')
print(a)     # ['rojo', 'verde', 'azul']
print(a.pop())  # azul
print(a)     # ['rojo', 'verde']
print(a.pop(0))  # rojo
print(a)     # ['verde']
```

- El operador + concatena dos listas, devolviendo una nueva lista
- El operador * concatena repetitivamente una lista a sí misma

```
#!/usr/bin/env python3
# listas05.py
a = ['rojo', 'verde' ]
a = a + ['azul']
print(a)      # ['rojo', 'verde', 'azul']
a += ['negro']
print(a)      # ['rojo', 'verde', 'azul', 'negro']
a = ['x', 'y']
a = a * 3
print(a)      # ['x', 'y', 'x', 'y', 'x', 'y']
```

Funciones, métodos y operadores

El lenguaje python:

- Emplea el modelo de programación imperativa convencional
  Por tanto usa funciones, cuya sintaxis es
  `funcion(objeto)`
- Emplea el modelo de programación orientada a objetos
  Por tanto usa métodos, cuya sintaxis es
  `objeto.metodo()`
- Es de muy alto nivel, cuenta con operadores con funcionalidad avanzada
  La sintaxis de un operador es
  `elemento1 operador elemento2`

Esto puede provocar confusión, es fácil equivocarse e intentar usar una función como un método o viceversa

Este script emplea la función `len()`, el método `pop()` y el operador `in`

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

lista=["rojo","amarillo","verde"]
print(len(lista))          # 3
print("blanco" in lista)   # False
print(lista.pop())         # verde
print(lista)               # ['rojo', 'amarillo']
```

Inversión de una lista

- El método `reverse()` invierte las posiciones de los elementos en una lista.

No devuelve nada, simplemente altera la lista sobre la que se aplican.

```
>>> a=['sota', 'caballo', 'rey']
>>> a.reverse()
>>> print(a)
['rey', 'caballo', 'sota']
```

Errores típicos:

```
a = reverse(a)   # ¡Mal! Esta función no existe
a = a.reverse()  # Ahora valdría None
```

El primer error lo indica el intérprete. El segundo es más peligroso

Ordenar una lista

- La función `sorted()` devuelve una lista ordenada (no la modifica)
- El método `sort()` ordena una lista (Modifica la lista, devuelve *None*)

Ambas admiten personalizar la ordenación de elementos complejos, pasando como argumento una función que devuelva la parte del elemento a usar como clave. Esta función argumento debe llamarse *key*

```python
#!/usr/bin/env python3
# ordenar01.py
mi_lista=[ "gamma", "alfa", "beta"]

print( sorted(mi_lista) )  # alfa, beta, gamma
print( mi_lista  )         # gamma, alfa, beta. No ha cambiado.

print( mi_lista.sort() )   # Devuelve 'None'
print( mi_lista )          # alfa, beta, gamma. La ha ordenado
```

```python
#!/usr/bin/env python3
# ordenar02.py
mi_lista=[  ['IV',4] , ['XX',20], ['III',3]  ]

def clave_lista(lista):
    return lista[1]

mi_lista.sort(key = clave_lista)
print( mi_lista )
```

Split, join

Es muy frecuente trocear una cadena para formar en un lista (split) y concatenar los elementos de una lista para formar una cadena (join)

```
#!/usr/bin/env python3
# split_join.py
mi_cadena="esto es una prueba"
print(mi_cadena.split()) # ['esto', 'es', 'una', 'prueba']
print("esto-tambien".split("-"))    # ['esto', 'tambien']

mi_lista=["as","dos","tres"]
#print(mi_lista.join()) # ¡ERROR! Parecería lógico que join()
                        # fuera un método del tipo lista. Pero no
                        # lo es.
print("".join(mi_lista)) # Es un método del tipo string, hay
                         # que invocarlo desde una cadena cualquiera,
                         # que será el separador.
                         # Devuelve  "asdostres"

print(",".join(mi_lista)) # Devuelve "as,dos,tres"
```

Otros métodos de los objetos string

```
#!/usr/bin/env python3
# cadenas.py
print("hola mundo".upper())  # HOLA MUNDO
print("HOLA MUNDO".lower())  # hola mundo

# Estos métodos devuelven una cadena,
# sin modificar la cadena original
a="prueba"
print(a.upper())             # PRUEBA
print(a)                     # prueba

# find() indica la posición de una subcadena
print("buscando una subcadena".find("una")) # 9
print("buscando una subcadena".find("nohay")) # -1

# strip() devuelve una copia de la cadena quitando
# espacios a derecha e izda, retornos de carro, etc
print("    hola  \n".strip())  # 'hola'

print("te digo que no".replace("digo","diego"))
      # imprime "te diego que no"
```

Nombres de objeto

Con frecuencia hablamos de *variables*, porque es el término tradicional en programación. Pero Python no tiene *variables*, sino *nombres*. Son referencias a objetos

```
#!/usr/bin/env python3
# nombres.py
x=['uno']
y=x       # y apunta al mismo objeto
print(x)  # ['uno']
print(y)  # ['uno']

x=['dos']  # x apunta a un nuevo objeto

print(x)  # ['dos']  # El objeto nuevo
print(y)  # ['uno']  # El objeto antiguo

x=['uno']
y=x       # y apunta al mismo objeto
x.append('dos')   # modificamos el objeto
print(x)  # ['uno','dos']  # el objeto modificado
print(y)  # ['uno','dos']  # el mismo objeto, modificado
```

Diccionarios

- Es un conjunto *desordenado* de elementos
- Cada elemento del diccionario es un par clave-valor.
- Se pueden obtener valores a partir de la clave, pero no al revés.
- Longitud variable
- Hace las veces de los *registros* en otros lenguajes
- Atención: Se declaran con `{}`, se refieren con `[]`

- Asignar valor a una clave existente reemplaza el antiguo
- Una clave de tipo cadena es sensible a mayúsculas/minúsculas
- Pueden añadirse entradas nuevas al diccionario
- Los diccionarios se mantienen desordenados
- Los valores de un diccionario pueden ser de cualquier tipo
- Las claves pueden ser enteros, cadenas y algún otro tipo
- Pueden borrarse un elemento del diccionario con `del`
- Pueden borrarse todos los elementos del diccionario con `clear()`

Otras operaciones con diccionarios:

- `len(d)` devuelve el número de elementos de `d`
- `d.has_key(k)` devuelve 1 si existe la clave `k` en `d`, 0 en caso contrario
- `k in d` equivale a: `d.has_key(k)`
- `d.items()` devuelve la lista de elementos de `d` (pares clave:valor)
- `d.keys()` devuelve la lista de claves de `d`

```python
#!/usr/bin/env python3
# diccionarios.py
pais={'de': 'Alemania', 'fr': 'Francia', 'es': 'España'}
print(pais["fr"])

extension={}
extension['py']='python'
extension['txt']='texto plano'
extension['mp3']='MPEG layer 3'

for x in pais.keys():
   print(x, pais[x])

del pais['fr']    # Borramos francia
print(len(pais))  # Quedan 2 paises
print('es' in pais)    # True
pais['es']="Spain"  # modificamos un elemento
pais.clear()  # Borramos todas las claves
```

```python
#!/usr/bin/env python3
# diccionarios02.py

diccionario={"juan": ["empanada"] ,
             "maria": ["refrescos","vino"]}

diccionario["luis" ] = ["patatas fritas","platos plastico"]
diccionario["luis"].append("vasos plastico")

claves = diccionario.keys() # Devuelve un tipo dict_keys
claves = list(claves)     # Lo convertimos en lista para ordenarlo
claves.sort()
for clave in claves:
     print(clave, diccionario[clave])
```

Resultado de la ejecución:

```
juan ['empanada']
luis ['patatas fritas', 'platos plastico', 'vasos plastico']
maria ['refrescos', 'vino']
```

Acceso a las claves mediante el operador in

Una forma alternativa de obtener las claves de un diccionario:

```
for clave in d:
    print(clave)
```

- Esto es más eficiente que emplear el método `keys()`
- Es aplicable a listas y tuplas
- Aunque en ocasiones seguiremos necesitando el método `keys()`

```
claves=list(diccionario.keys())
claves.sort()
```

Tuplas

- Tipo predefinido de Python para una lista inmutable
- Se define de la misma manera que las listas, pero con los elementos entre paréntesis
- Las tuplas no tienen métodos: no se pueden añadir elementos, ni cambiarlos, ni buscar con `index()`
- Sí puede comprobarse la existencia con el operador `in`.

```
>>> t = ("a", "b", "blablabla", "z", "example")
>>> t[0]
'a'
>>> 'a' in t
True
>>> t[0] =  "b"
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
```

Utilidad de las tuplas:

- Son más rápidas que las listas
- Pueden ser una clave de un diccionario (no así las listas)
- Se usan en el formateo de cadenas

`tuple(li)` devuelve una tupla con los elementos de la lista `li`
`list(t)` devuelve una lista con los elementos de la tupla `t`

Asignaciones múltiples

- Pueden hacerse también tuplas de nombres de objetos

```
>>> v = ('a', 'b', 'e')
>>> (x, y, z) = v
>>> x
'a'
```

If

```
#!/usr/bin/env python3
# if01.py
x = True
if x :
     print('verdadero')
else:
     print('falso')
```

Nótese como el carácter : introduce cada bloque de sentencias.
Si en una rama queremos poner la sentencia nula: `pass`
No confundir con el valor nulo: `None`

```
#!/usr/bin/env python3
# if02.py

x = int(input("Escribe un entero: "))
if x < 0:
     x = 0
     print('Valor negativo, modificado a cero')
elif x == 0:
     print('Cero')
elif x == 1:
     print('Uno')
else:
     print('Más de uno')
```

La sentencia case aparece en la versión 3.10 (Octubre 2021)

For

En los lenguajes *convencionales*, la cláusula *for* sirve para que un entero recorra una serie de valores.
En python es diferente: recorre un objeto iterable, como una lista o una tupla. Por cada elemento del iterable, ejecuta el bloque de código

```
lista = ["sota","caballo","rey"]
for x in lista:
    print(x) # Imprime el elemento y nueva línea
```

Resultado:

```
sota
caballo
rey
```

Si necesitamos un bucle *convencional* podemos emplear la función *range()*

```
#!/usr/bin/env python3
# rangos01.py
rango = range(3)
for x in rango:
    print(x, end='')
```

Resultado:

```
012
```

Observa que la función *print*

- Por omisión añade un carácter *nueva línea* tras cada argumento
- Para evitarlo, podemos añadir el parámetro *end* indicando qué añadir. Por ejemplo una cadena vacía, como en este caso

A range() le podemos pasar

- Un elemento: el final del rango
- Dos elementos: principio y final
- Tres elementos: principio, final e incremento

Por omisión, el principio es 0 y el incremento es +1
Esta función devuelve un objeto de tipo *range*. Si es necesario, podemos convertirlo en lista con la función *list*

```
#!/usr/bin/env python3
# rangos02.py
print(range(2,5))              # range(2,5)
print(list(range(2,5)))        # 2, 3, 4
print(list(range(2,-2,-1)))    # 2, 1, 0, -1
```

No deberíamos usar range para los bucles a menos que sea imprescindible. No es idiomático en python, añade complejidad innecesaria.
No hagas bucles *al estilo Pascal*

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

lista=["sota","caballo","rey"]
# ¡¡NO HAGAS ESTO!!
for i in range(len(lista)):
    print(lista[i])

# Lo idiomático en python es
for x in lista:
    print(x)
```

While

```
>>> a=0
>>> while a<10:
...     print(a, end='')
...     a+=1
...
0 1 2 3 4 5 6 7 8 9
```

`break` sale de un bucle. (Aunque según los principios de la *programación estructurada*, `break` no debería usarse nunca. Empléalo solo si estás muy seguro de lo que haces)

```
#!/usr/bin/env python3
# while01.py
a=10
while a > 0:
    print(a, end='')
    a-=1
```

equivale a

```
#!/usr/bin/env python3
# while02.py
a=10
while True:
    print(a, end='')
    if a==1:
        break
    a-=1
```

format

Las cadenas cuentan con el método `format()`

- Dentro de una cadena, podemos indicar, entre llaves, qué campos se mostrarán y con qué formato.
  Format tiene un microlenguaje para esto
- Los argumentos de `format()` serán los campos

Ejemplo: Indicar qué campo mostrar, a partir del ordinal

```
#!/usr/bin/env python3

name="Juan"
surname="García"
print("Se llama {0} y se apellida {1}".format(name,surname))
print("Se llama {} y se apellida {}".format(name,surname))

persona=["Juan","García"]
print("Se llama {0[0]} y se apellida {0[1]}".format(persona))

persona={"name":"Juan", "surname":"García"}
print("Se llama {0[name]} y se apellida {0[surname]}".format(persona))
```

Resultado:

```
Se llama Juan y se apellida García
Se llama Juan y se apellida García
Se llama Juan y se apellida García
Se llama Juan y se apellida García
```

Después de indicar qué campo mostrar, separado por el carácter dos puntos, podemos especificar cuántos caracteres debe ocupar la salida, y si estará alineada a la derecha (signo de mayor), a la izquierda (signo de menor o ningún signo) o al centro (acento circunflejo)

Ejemplo: mostrar una palabra, ocupando siempre 12 caracteres

```
#!/usr/bin/env python3

print("{0:>12}{1:>12}".format("sota","caballo"))
print("{0:<12}{1:<12}".format("sota","caballo"))
print("{0:12}{1:12}".format("sota","caballo"))
print("{0:^12}{1:^12}".format("sota","caballo"))
```

Resultado:

```
        sota      caballo
sota            caballo
sota            caballo
    sota          caballo
```

- Si solo hay un campo, podemos omitir el 0 a la izquierda del carácter dos puntos
- Con el carácter `d` podemos indicar que el campo contiene un número entero. En este caso, la alineación por omisión es a la derecha
- Con el carácter `f` indicamos que el campo es un número real Podemos especificar cuántos decimales representar. Por ejemplo 4: `.4f`

```
print("{:<6d} metros".format(592))
print("{:>6d} metros".format(592))
print("{0:6d} metros".format(592))
print("Pi vale {:.4f}".format(3.14159265358979))
```

Resultado:

```
592    metros
   592 metros
   592 metros
Pi vale 3.1416
```

Naturalmente, la cadena no tiene por qué ser una constante, puede ser una variable

```
#!/usr/bin/env python3

x=12.3
y=0.345
z=34000
template="{:8},{:8},{:8}"
msg=template.format(x,y,z)
print(msg) #    12.3,   0.345,   34000
```

Funciones

```python
#!/usr/bin/env python3
# grados.py
def a_centigrado(x):
    """Convierte grados farenheit en grados centígrados."""
    return (x-32)*(5/9.0)


def a_farenheit(x):
    """Convierte grados centígrados en grados farenheit."""
    return (x*1.8)+32
```

Los nombres de objeto declarados fuera de una función son globales, y los declarados dentro, locales

```
#!/usr/bin/env python3
# ambito01.py
a=3
def f():
    b=4
    print(a) # 3
    print(b) # 4
    return
f()
print(a)     # 3
print(b)     # ¡Error! B es un objeto local
```

- Algunas metodologías establecen que los objetos globales deben usarse lo mínimo posible. Otras los prohiben por completo
- Los objetos globales pueden leerse dentro (y fuera) de la función.
- Los objetos locales, declarados dentro de una función, son invisibles fuera de ella

Supongamos que intentamos modificar el objeto global de esta forma

```
#!/usr/bin/env python3
# ambito02.py
a=3
def f():
    a=0
    print(a) # 0
    return
f()
print(a)      # 3 . No se ha modificado
```

No podemos modificar el objeto global sin más, lo que sucede es que python crea un nuevo objeto local, con el mismo nombre que el global. El objeto local hace que el objeto global sea invisible, el local *tapa* al global

Las modificaciones similares a esta siempre generarán un error

```
#!/usr/bin/env python3
# ambito03.py
c=3
def f():
    c=c-1   # ERROR: la variable global ya no es visible y la
            # local aún no está definida
    return

f()
```

En cuanto el intérprete procesa el nombre del objeto a la izquierda de un signo igual, crea un objeto local que aún no está definido, pero que hace invisible al objeto global

Para poder modificar un objeto global, es necesario declararlo con la sentencia `global`

```
#!/usr/bin/env python3
# ambito04.py
c=3
def f():
    global c
    c=0        # Esto modifica el objeto global
    print(c)   # 0
    return
f()
print(c)        #0
```

La sentencia global evita que al declarar un objeto en una función, se cree un nuevo objeto con el mismo nombre pero de ámbito local. Por tanto permite modificar el objeto global

En muchos lenguajes, para hacer que una variable sea global, la declararíamos `global` en la la *zona global* del código, haríamos un código similar a este, pero que en python es incorrecto

```
#!/usr/bin/env python3
# ambito05
global c   #ERROR, esto no sirve de nada
c=3
def f():
    c=0       # Esto es un objeto local
    print(c) # 0
    return
f()
print(c)       #3  el global no ha cambiado
```

- Observa que en python se usa la sentencia `global` en la función local que vaya a modificar el objeto

- Dicho de otro modo: la sentencia `global` no significa *haz que este objeto sea global*, sino *haz que este objeto global pueda ser modificado aquí*
- Seguramente resultaría más intutivo si la sentencia `global` tuviera un nombre distinto. Tal vez `global-write` o `GlobalModify`

Los objetos mutables (listas, diccionarios...) declarados dentro de una función también son locales, en este aspecto se comportan igual que los objetos inmutables

```
#!/usr/bin/env python3
# ambito06.py
l= ["uno","dos"]
def f():
    l=["cuatro"]  # nuevo objeto mutable, local

print(l) #  ["uno","dos"]
f()
print(l) #  ["uno","dos"]
```

Hay una diferencia entre los objetos mutables y los inmutables. Como hemos visto

- Los objetos inmutables globales se pueden leer localmente
- Para poder modificar un objeto inmutable global, es necesario usar la sentencia `global`
  Por tanto, un objeto global sin la sentencia `global` queda *protegido contra escritura*

Los objetos mutables globales no se pueden *proteger contra escritura* de esta manera

Un objeto mutable sí puede ser modificado en una función local, a pesar de no estar declarado `global`

```
#!/usr/bin/env python3
# ambito07.py
l= ["uno","dos"]
def f():
    l.pop()

print(l) #  ["uno","dos"]
f()
print(l) #  ["uno"]  . El objeto mutable fue modificado por la función
```

El objeto mutable puede ser modificado a través de sus métodos. (No debo pensar que la ausencia de la sentencia `global` hace que el objeto esté en modo *solo lectura*)

```python
#!/usr/bin/env python3
# ambito08.py
l= ["uno","dos"]
def f():
    l=["uno"]
print(l) #  ["uno","dos"]
f()
print(l) #  ["uno","dos"]  No cambia .
```

En el caso de que la modificación se haga redefiniendo el objeto (no mediante métodos), como ya sabemos, implica la declaración implícita de un objeto nuevo, local, que oculta al objeto global. Por tanto, el objeto global no es modificado

Si al ejemplo anterior le añadimos `global` de esta manera, como cabría esperar, permite modificar el objeto global

```
#!/usr/bin/env python3
# ambito09.py
l= ["uno","dos"]
def f():
    global l
    l=["uno"]
print(l) #  ["uno","dos"]
f()
print(l) #  ["uno"]  Ha cambiado.
```

Resumen:

- Los objetos declarados fuera de una función son globales
- Los objetos declarados dentro de una función son locales
- Los objetos globales siempre se pueden leer dentro de una función
- Para modificar un objeto global dentro de una función
  - Si es inmutable, hay que usar `global` dentro de la función
  - Si es mutable
    - Para modificarlo mediante una asignación, hay que usar `global`
    - Para modificarlo mediante sus métodos, no es necesario usar `global`

En las llamadas a funciones

- Los objetos inmutables se pasan por valor. La función recibe una copia del valor, por lo que una posible modificación de la copia no altera el original
- Los objetos mutables se pasan por referencia. La función recibe una referencia al objeto original, una modificación del objeto en la función modifica el objeto original

```
#!/usr/bin/env python3
# valor_referencia.py
def f(x,y):
    x=x-1
    y.pop()

def g():
    v=3
    l=["uno","dos"]
    f(v,l)
    print(v)  # 3  . La función creó copia local, no tocó el parámetro
    print(l)  # ['uno'] La función recibió el parámetro por referencia

g()
```

Cadenas de documentación

- No son obligatorias pero sí muy recomendables (varias herramientas hacen uso de ellas).
- La cadena de documentación de un objeto es su atributo `__doc__`
- En una sola línea para objetos sencillos, en varias para el resto de los casos.
- Entre triples comillas-dobles (incluso si ocupan una línea).
- Si hay varias líneas:
  - La primera línea debe ser una resumen breve del propósito del objeto. Debe empezar con mayúscula y acabar con un punto
  - Una línea en blanco debe separar la primera línea del resto
  - Las siguientes líneas deberían empezar justo debajo de la primera comilla doble de la primera línea

De una sola línea:

```
def kos_root():
    """Return the pathname of the KOS root directory."""
    global _kos_root
    ...
```

De varias:

```
def complex(real=0.0, imag=0.0):
    """Form a complex number.

    Keyword arguments:
    real -- the real part (default 0.0)
    imag -- the imaginary part (default 0.0)

    """
    if imag == 0.0 and real == 0.0: return complex_zero
```

Documentando el código (tipo Javadoc)

- Permite documentar el código -generalmente las funciones- dentro del propio código
- Genera la documentación del código en formatos legibles y navegables (HTML, PDF...)
- Se basa en un lenguaje de marcado simple
- PERO... hay que mantener la documentación al día cuando se cambia el código

Ejemplo

```
def interseccion(m, b):
    """
    Devuelve la interseccion de la curva M{y=m*x+b} con el eje X.
    Se trata del punto en el que la curva cruza el eje X (M{y=0}).

    @type  m: número
    @param m: La pendiente de la curva
    @type  b: número
    @param b: La intersección con el eje Y

    @rtype:   número
    @return:  la interseccíoin con el eje X de la curva M{y=m*x+b}.
    """
    return -b/m
```

Ficheros

- `open(nombre_fichero,modo)` devuelve un objeto fichero. modo:
  - `w`: Escritura. Destruye contenido anterior
  - `r`: Lectura. Modo por defecto
  - `r+`: Lectura y escritura
  - `a`: Append
- `write(cadena)` escribe la cadena en el fichero. Solo escribe cadenas, para otros tipos, es necesario pasar a texto o usar librerías como *json* o *pickle*
- `read()` devuelve una cadena con todo el contenido del fichero
- `readlines()` devuelve una lista donde cada elemento es una línea del fichero
- `close()` cierra el fichero

```
#!/usr/bin/env python3
# ficheros01.py
lista=['sota','caballo','rey']
fichero=open('prueba.txt','w')
for x in lista:
    fichero.write(x+"\n")
fichero.close()

fichero=open('prueba.txt','r')
mi_cadena=fichero.read()           # Una sola cadena de 3 líneas
fichero.seek(0)                    # vuelvo al princio del fichero

lista_de_cadenas=fichero.readlines()  # ahora cada elemento incluye \n
fichero.seek(0)

for linea in fichero.readlines():
    print(linea, end='')

fichero.close()
```

Los métodos *read()* y *readlines()* crean una copia completa del fichero en memoria.
Para ficheros muy grandes es más eficiente trabajar línea a línea

```
fichero=open('prueba.txt','r')
for linea in fichero:
    print(linea, end='')
fichero.close()
```

No se deben mezclar estas dos maneras de acceder a un fichero

Excepciones

- Un programa sintácticamente correcto puede dar errores de ejecución

```
#!/usr/bin/env python3
# excepcion01.py
while 1:
    x=int(input("Introduce un nº" ))
    print(x)
```

Si el usuario no escribe un número sino por ejemplo *a*

```
  File "./excepcion01.py", line 4, in <module>
    x=int(input("Introduce un nº"))
ValueError: invalid literal for int() with base 10: 'a'
```

- Definimos una acción para determinada excepción

```
#!/usr/bin/env python3
# excepcion02.py
while 1:
    try:
        x=int(input("Introduce un nº:"))
        print(x)
    except ValueError:
        print("Número incorrecto")
```

- Se puede indicar una acción que se ejecute sea cual sea la excepción pero es *muy* desaconsejable (enmascara otros errores)
- El programador puede levantar excepciones

```
#!/usr/bin/env python3
# excepcion03.py
try:
    x=int(input("Introduce un nº:"))
    print(x)
except :      # para cualquier excepción
    raise Exception("Número incorrecto")
```

Fechas

- En Unix, y por tanto en internet, es habitual expresar la fecha y la hora como el número de segundos transcurridos desde el *epoch* (1 de enero de 1970 a las 00:00)
- En python, la librería *time* ofrece el método *time* que acepta una fecha en segundos desde el *epoch*, y la devuelve en un *struct_time*
  - Si invocamos a este método sin argumentos, nos devuelve la fecha actual
- Para conocer la hora Unix de modificación de un fichero `os.path.getmtime("/nombre/de/fichero")`

```python
#!/usr/bin/env python3
# fechas.py
import time,os

t_unix = time.time()       # Segundos desde el epoch
t = time.gmtime(t_unix)    #  Fecha en un struct_time
print(t)
print(t.tm_year)
print(t.tm_mon)
print(t.tm_mday)

print(os.path.getmtime("/etc/hosts"))
```

Resultado:

```
time.struct_time(tm_year=2021, tm_mon=11, tm_mday=24, tm_hour=16, tm_min=48,
tm_sec=3, tm_wday=2, tm_yday=328, tm_isdst=0)
2021
11
24
1637418845.3172204
```

Cadenas binarias

- Hasta la década de 1990, en prácticamente cualquier ámbito de la informática, un carácter equivalía a un byte. La codificación de caracteres no ingleses era compleja, con distintas normas incompatibles
- En las décadas de 1990, 2000 se emplea la norma ISO 8859. Para los lenguajes de Europa Occidental, ISO 8859-1, también llamada *latin 1*
- En la década de 2000 se empieza a usar la norma Unicode, concretamente con la codificación UTF-8
  - Es la codificación prácticamente universal en la actualidad, aunque no es raro encontrarse sistemas antiguos con normas anteriores
  - Permite representar cualquier lenguaje humano, incluyendo lenguajes extintos y lenguajes artificiales
  - UTF-8 es un superconjunto de ASCII, para cada caracter emplea entre 1 y 4 bytes

En python 3, todas las cadenas son unicode UTF-8 por omisión

- Si es necesario, también se pueden usar cadenas *al estilo antiguo*, llamadas tipo *bytes*[2]
- Podemos declarar una cadena de tipo *bytes* anteponiendo *b*

```
b"Esto es una cadena de tipo byte"
```

- Para detectar el tipo de una cadena usamos la función *type*

```
print(type('holamundo'))    #   <class 'str'>
print(type(b'holamundo'))   #   <class 'bytes'>
```

[2] Al contrario que en python 2, donde por omisión era de tipo byte aunque también existía el tipo unicode

Conversión entre cadenas de 8 bits y cadenas unicode en python 3

- De string (unicode) a byte (8 bits)

```
>>> "españa".encode('utf-8')
b'espa\xc3\xb1a'
```

- De bytes (8 bits) a string (unicode)

```
>>> b=b'123'
>>> print(b)
b'123'
>>> s=b.decode()
>>> print(s)
123
```

libreria sys

- Argumentos de linea de órdenes

`sys.argv` devuelve una lista con los argumentos pasados al script python desde la shell

```
#!/usr/bin/env python3
# argumentos.py
import sys
print(sys.argv[:])
```

```
koji@mazinger:~$ ./argumentos.py un_argumento otro_argumento
['./argumentos.py', 'un_argumento', 'otro_argumento']
```

(El argumento cero es el nombre del programa)

Escribir en stderr

```
#!/usr/bin/env python3
# error_estandar.py
import sys
sys.stderr.write('Error: \n')
```

Leer desde stdin, escribir en stdout

```
#!/usr/bin/env python3
# cat.py
import sys
for linea in sys.stdin.readlines():
    sys.stdout.write(linea)
```

subprocess

- `subprocess.check_output()` permite ejecutar una orden de la shell en un subproceso externo
- Aunque puede ser muy útil, el script deja de ser portable entre sistemas operativos diferentes
- Su primer argumento es una lista con los argumentos de la orden a ejecutar
- Devuelve la salida estándar del subproceso pero como *bytes* que habrá que convertir a cadena (unicode) con *decode*
  - En caso contrario obtendremos un error

    ```
    TypeError: a bytes-like object is required, not 'str'
    ```

- Si se produce algún error, eleva la excepción `subprocess.CalledProcessError`

Los comandos de la shell normalmente adaptan su salida al número de filas y columnas que tenga nuestro terminal

- Estas dimensiones se obtienen desde las variables de entorno `COLUMNS` y `LINES`
- Por si acaso estamos usando un terminal muy pequeño, es conveniente indicarle a la shell que ejecutamos con *subprocess* que considere un terminal de tamaño *normal*. Para ello, damos valor a estas variables

```
os.environ["COLUMNS"]="80"
os.environ["LINES"]="150"
```

```
#!/usr/bin/env python3
# subproceso01.py
import subprocess,sys,os
os.environ["COLUMNS"]="80"
os.environ["LINES"]="150"

mandato="ps -ef"
mandato_troceado=mandato.split()
try:
    salida=subprocess.check_output(mandato_troceado)
except subprocess.CalledProcessError:
    sys.stderr.write("La orden ha producido un error\n")
    raise SystemExit
salida=salida.decode("utf-8")   # De bytes a string
lineas=salida.split("\n")  # troceamos la salida línea a línea
lineas.pop(0)       # quitamos la primera línea, la cabecera del ps
for linea in lineas:
    if len(linea) != 0:
        campos_linea=linea.split()
        usuario = campos_linea[0]
        proceso =  campos_linea[7]
        print("Usuario:{} Proceso:{}".format(usuario,proceso))
```

Si necesitamos más control sobre los posibles errores

- Para redirigir la salida de error del subproceso a la salida estándar, pasamos el parámetro `stderr=subprocess.STDOUT`
- El atributo `returncode` de `CalledProcessError` contiene el código del error

```
#!/usr/bin/env python3
# subproceso02.py
import subprocess,sys,os

os.environ["COLUMNS"]="80"
os.environ["LINES"]="150"

mandato="ls inexistente"
mandato_troceado=mandato.split()
try:
    salida=subprocess.check_output(mandato_troceado,
                   stderr=subprocess.STDOUT)
except subprocess.CalledProcessError as e:
    plantilla = "{}Código:{}\n"
    msj_subproceso = e.output
    msj_subproceso = msj_subproceso.decode() # De bytes a string
    codigo_subproceso = e.returncode
    msj = plantilla.format(msj_subproceso, codigo_subproceso)
    sys.stderr.write(msj)
```

os.path

- Las funciones os.path.join() y os.path.split() unen y separan nombres de fichero con directorios
  - Son compatibles con cualquier S.O.
  - No importa si el path acaba en barra o no
- `os.path.exists()` devuelve un boolean indicando si un fichero existe

```
#!/usr/bin/env python3
# path01.py
import os
ejemplo=os.path.join("/etc/apt","sources.list")
print(ejemplo)   # /etc/apt/sources.list
print(os.path.split(ejemplo))  # ('/etc/apt', 'sources.list')

print(os.path.exists(ejemplo))  # True
print(os.path.exists("/usr/local/noexiste"))  # False
```

Enlazar, borrar

```
#!/usr/bin/env python
# enlazar_borrar.py
import os
if not os.path.exists("/tmp/test"):
    os.mkdir("/tmp/test")
os.chdir("/tmp/test")                          # cd /tmp/aa
os.symlink("/etc/hosts","enlace_hosts") # crea enlace simbólico
os.remove("enlace_hosts")                      # borra el fichero
os.rmdir("/tmp/test")                          # borra directorio (vacio)
```

copiar, copiar y borrar recursivamente

```
#!/usr/bin/env python3
# recursivo.py
import shutil,os
shutil.copytree("/home/koji/.gnome","/tmp/probando")
    # copia recursivamente. El destino no debe existir

shutil.copy("/etc/hosts","/tmp/probando")
    # copia 1 fichero (como el cp de bash)

shutil.move("/tmp/probando/hosts","/tmp/probando/mi_hosts")

shutil.rmtree("/tmp/probando")    # borra directorio lleno
```

os.walk

- Recorre recursivamente un directorio
- Por cada directorio devuelve una 3-tupla
  - Directorio
  - Subdirectorios
  - Ficheros

```
#!/usr/bin/env python3
# walk.py
import os
directorio_inicial=os.getcwd() # current working directory
os.chdir("/tmp/musica")        # cd

for x in os.walk("."):
    print(x)

os.chdir(directorio_inicial)
```

```
/tmp/musica
|-- listado.txt
|-- jazz
`-- pop
    |-- sabina
    |   |-- pirata_cojo.mp3
    |   `-- princesa.mp3
    `-- serrat
        |-- curro_el_palmo.mp3
        `-- penelope.mp3
```

```
('.', ['jazz', 'pop'], ['listado.txt'])
('./jazz', [], [])
('./pop', ['serrat', 'sabina'], [])
('./pop/serrat', [], ['curro_el_palmo.mp3', 'penelope.mp3'])
('./pop/sabina', [], ['princesa.mp3', 'pirata_cojo.mp3'])
```

Persistencia

Persistencia en Python:
Una librería de persistencia permite *serializar objetos*, esto es, convertirlos en una secuencia binaria o de texto que se pueda transmitir fuera de python, almacenar en disco, base de datos, etc

- Librería *Pickle*
  - Formato propio de Python, binario, muy eficiente
  - Soporta cualquire clase, predefinida en Python o por el usuario
- Librería *json*
  - Formato estándar de internet, modo texto, legible
  - Solo soporta cadenas, números, booleanos, diccionarios y listas

Persistencia con pickle

```
#!/usr/bin/env python3
# conserva.py
import pickle

cp={28:'madrid',8:'barcelona',33:'asturias'}
fich=open('prueba.pick','wb')  # Apertura modo binario
pickle.dump(cp,fich)
fich.close()

fich=open('prueba.pick','rb')  # Lectura modo binario
codigos_postales=pickle.load(fich)
fich.close()

for x in codigos_postales.keys():
    print(x,codigos_postales[x])
```

Persistencia con json

```python
#!/usr/bin/env python3
# ejemplo_json.py
import json

# Objeto python ordinario
diccionario = { 'MAD':'Madrid Barajas', 'MCV':'Madrid Cuatro Vientos'}
print(type(diccionario))    # <class 'dict'>

# Lo convertimos en cadena json
cadena_json = json.dumps(diccionario)
print(type(cadena_json))    # <class 'str'>
print(cadena_json)
# {"MAD": "Madrid Barajas", "MCV": "Madrid Cuatro Vientos"}

# Volvemos a convertir la cadena en objeto python
diccionario = json.loads(cadena_json)
print(type(diccionario))  # <class 'dict'>
```