

Programación en Python

Escuela Tec. Superior de Ingeniería de Telecomunicación

gsync-profes (arroba) gsync.urjc.es

Noviembre de 2017



©2017 GSyC

Algunos derechos reservados.
Este trabajo se distribuye bajo la licencia

Creative Commons Attribution Share-Alike 4.0

- 1 El Lenguaje Python
- 2 Programación en Python
 - Tipos de objeto
 - Cadenas
 - Listas
 - Diccionarios
 - Tuplas
 - Cadenas Unicode
 - Sentencias de control
 - Funciones
 - Ficheros
 - Cadenas de documentación
 - Excepciones
- 3 Librerías
 - 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:

- CGI, 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

- Hasta la versión 2.5.4 (diciembre de 2008), se denomina *python library reference*
- Desde la versión 2.6, se denomina *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 automatically becomes a reference to the receiver.

- ...

El intérprete de python se puede usar

- En modo interactivo

```
koji@mazinger:~$ python
Python 2.5.2 (r252:60911, Oct  5 2008, 19:24:49)
[GCC 4.3.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print "hola mundo"
hola mundo
>>> 3/2
1
>>> 3/2.0
1.5
```

- Mediante scripts

```
#!/usr/bin/python -tt
# -*- coding: utf-8 -*-
print "hola mundo"      #esto es un comentario
euros=415
pesetas=euros*166.386
print str(euros) + " euros son " + str(pesetas) + " pesetas"
```

La línea `#!/usr/bin/python` indica al S.O. dónde está el intérprete que sabe procesar el fuente

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

```
#Este ejemplo es doblemente incorrecto
#! /usr/bin/python -tt
# -*- coding: utf-8 -*-
# ¡MAL!
```

En distintos Unix el intérprete puede estar en distintos sitios. Para aumentar la compatibilidad, a veces se usa

```
#!/usr/bin/env python
print "Hola mundo"
```

Aunque (en Linux) esto no permite pasar parámetros como `-tt`

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

Se puede usar utf-8 y latin-1 (iso-8859-1) en las cadenas y comentarios

- Si el editor no marca adecuadamente la codificación del fichero, aparecerá un error

```
SyntaxError: Non-ASCII character '\xc3' in file ./holamundo.py on
line 4, but no encoding declared;
see http://www.python.org/peps/pep-0263.html for details
```

y será necesario añadir en la segunda línea del fuente

```
# -*- coding: utf-8 -*-
```

o bien

```
# -*- coding: iso-8859-1 -*-
```

o bien

```
# -*- coding: Win-1252 -*-
```

- En cualquier Unix/Linux a partir de mediados de la década del año 2000, la codificación habitual es utf-8
- En Windows lo habitual es Win-1252, aunque suele indicarse iso-8859-1 (latin-1), que es muy similar (y más general)
- Si vamos a trabajar en Linux y en Windows, cualquier editor de calidad podrá trabajar en ambos formatos, no es necesario recodificar cada vez que cambiemos la plataforma ¹

¹Consulta las transparencias sobre editores de texto para ver cómo configurar vim en Windows para que siempre use 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 débilmente tipado, admite casi cualquier cosa.
Esto resulta cómodo, flexible, potencialmente peligroso

En Python la declaración de 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/python
sum_elementos= 3 + 4 + 17
media=suma_elementos / 3    # deletreamos mal la variable
print media;    # y el intérprete nos avisa con un error
```

Funciones predefinidas

- `abs()` valor absoluto
- `float()` convierte a float
- `int()` convierte a int
- `str()` convierte a string
- `round()` redondea
- `raw_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 línea, 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
- Mezclar espacios con tabulares es muy peligroso.
Para que el intérprete lo advierta

```
#!/usr/bin/python -t
```

Para que el intérprete lo prohíba

```
#!/usr/bin/python -tt
```

En python3 no es necesario, no permite mezclar espacios y tabuladores

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/python -tt
# -*- coding: utf-8 -*-
import types
if type("a") == types.StringType:
    print "ok, es una cadena"
else:
    print "no es una cadena"
```

Tipos de objeto habituales:

- BooleanType
- IntType
- LongType
- FloatType
- StringType
- ListType
- TupleType
- DictType

Cadenas

- No existe tipo char
- Comilla simple o doble

```
print "hola"
print 'hola'
print 'me dijo "hola"'
más legible que print 'me dijo \'hola\''
```
- Puede haber caracteres especiales

```
print "hola\nque tal"
```
- Cadenas crudas

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

- Una cadena se puede expandir en más de una línea

```
print "hola\  
      que 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 ,

```
a=['rojo','amarillo']
a.append('verde')
print a
print a[2]
print len(a)
```

```
b=['uno',2, 3.0]
```

- 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/python -tt
# -*- coding: utf-8 -*-
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]
```

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

```
>>> li
['a', 'b', 'blablabla', 'z', 'example']
>>> li.append("new")
>>> li
['a', 'b', 'blablabla', 'z', 'example', 'new']
>>> li.insert(2, "new")
>>> li
['a', 'b', 'new', 'blablabla', 'z', 'example', 'new']
```

- `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
>>> lista=['cero','uno','dos']
>>> print lista
['cero', 'uno', 'dos']
>>> "doce" in lista
False
>>> print lista.index('uno')
1
>>> 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)

```
>>> li
['a', 'b', 'new', 'blablabla', 'z', 'example', 'new', 'two', 'elements']
>>> li.remove("new")
>>> li
['a', 'b', 'blablabla', 'z', 'example', 'new', 'two', 'elements']
>>> li.remove("c")
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ValueError: list.remove(x): x not in list
>>> li.pop()
'elements'
>>> li
['a', 'b', 'blablabla', 'z', 'example', 'new', 'two']
```


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

```
>>> li = ['a', 'b', 'blablabla']
>>> li = li + ['example', 'new']
>>> li
['a', 'b', 'blablabla', 'example', 'new']
>>> li += ['two']
>>> li
['a', 'b', 'blablabla', 'example', 'new', 'two']
>>> li = [1, 2] * 3
>>> li
[1, 2, 1, 2, 1, 2]
```

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`

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

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

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']
```

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, pasando como argumento una función que compare dos elementos y devuelva

- Un valor negativo si están ordenados
- Cero si son iguales
- Un valor positivo si están desordenados

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

```
#!/usr/bin/python -tt
# -*- coding: utf-8 -*-
mi_lista=[ ['IV',4] , ['XX',20], ['III',3] ]

def mi_ordena(a,b):
    if a[1] < b[1]:
        return -1
    elif a[1] > b[1]:
        return 1
    else:
        return 0

mi_lista.sort(mi_ordena)
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/python -tt
# -*- coding: utf-8 -*-
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 "asdos tres"

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


Otros métodos de los objetos string

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

En las primeras versiones de python no había métodos para los objetos de tipo *string*, se usaban funciones de un módulo *string*. A partir de python 2.x esta forma se va considerando obsoleta, en python 3.x desaparece

```
#!/usr/bin/python -tt
# -*- coding: utf-8 -*-
import string # ¡Forma obsoleta!
a="más vale pájaro en mano"
print string.split(a)
print string.upper(a)

c=['rojo', 'amarillo', 'verde']
print string.join(c)
```

- Métodos actuales para tratar cadenas: *Built-in Types, String Methods*
- Funciones antiguas: *String module*

Nombres de objeto

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

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

```
#!/usr/bin/python -tt
# -*- coding: utf-8 -*-
pais={'de': 'Alemania', 'fr': 'Francia', 'es': 'España'}
print pais
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'] # Borrarnos francia
print len(pais) # Quedan 2 paises
print 'es' in pais # True
pais['es']="Spain" # modificamos un elemento
pais.clear() # Borrarnos todas las claves
```

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

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

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

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

Tuplas

Tipo predefinido de Python para una lista inmutable.

Se define de la misma manera, 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 y rangos

- Pueden hacerse también tuplas de variables:

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

- La función `range()` permite generar listas al vuelo:

```
>>> range(7)
[0, 1, 2, 3, 4, 5, 6]
>>> (MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
... FRIDAY, SATURDAY, SUNDAY) = range(7)
>>> MONDAY
0
>>> SUNDAY
6
```

Cadenas Unicode

Hasta los años 90, en prácticamente cualquier ámbito de la informática, un carácter equivalía a un byte. Pero codificando en UTF-8 esto ya no es cierto

```
>>> pais={'es':'españa'}
>>> print pais
{'es': 'espa\xc3\xb1a'}
>>> print pais['es']
españa
```

- `\xc3\xb1` significa *C3 en hexadecimal, B1 en hexadecimal* (Letra ñe en UTF-8)
- Cuando imprimimos el diccionario, se muestra la representación interna de la ñe
- Cuando imprimimos la cadena, python muestra correctamente el grafema correspondiente

- Cuando imprimimos la cadena completa, python la muestra correctamente
- Cuando imprimimos cada elemento, no

```
#!/usr/bin/python -tt
# -*- coding: utf-8 -*-
cadena="¿Procesa bien el español?"
print cadena
indice = 0
while indice < len(cadena):
    letra = cadena[indice]
    print letra,
    indice=indice+1
```

Resultado:

```
¿Procesa bien el español?
? ? P r o c e s a   b i e n   e l   e s p a ? ? o l ?
```

Cadenas Unicode

- En python 2.0 aparecen las cadenas unicode
- Una constante cadena unicode constante se crea anteponiendo u

```
cadena_unicode=u"Con cadenas unicode se trabaja mejor en español"
```

- Conversión desde objeto cadena ordinaria hasta cadena unicode
Empleamos la función *unicode*

```
cadena_unicode=unicode(cadena,"utf-8")
```

- Conversión desde cadena unicode hasta cadena ordinaria
Empleamos el método *encode*

```
cadena=cadena_unicode.encode("utf-8")
```

En el ejemplo anterior, basta con usar una cadena unicode para generar una salida correcta

```
cadena=u"¿Procesa bien el español?"
```

```
¿Procesa bien el español?
```

```
¿ P r o c e s a   b i e n   e l   e s p a ñ o l ?
```


- Es recomendable que en todos nuestros scripts
 - Aceptemos cadenas ordinarias
 - Conviertamos las cadenas ordinarias en unicode, y las procesemos siempre en unicode
 - En la salida, volvamos las cadenas unicode a ordinarias

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

def main():
    # cadena ordinaria
    cadena="probando"
    print type(cadena)    # <type 'str'>

    # convertimos la cadena ordinaria en cadena unicode
    cadena_unicode=unicode(cadena,"utf-8")
    print type(cadena_unicode) # <type 'unicode'>

    # convertimos de vuelta la cadena unicode en cadena ordinaria
    cadena=cadena_unicode.encode("utf-8")
    print type(cadena)    # <type 'str'>

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

If

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

x = 3
if x :
    print 'verdadero'
else:
    print 'falso'
```

Nótese como el carácter `:` introduce cada bloque de sentencias.

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

x = int(raw_input("Please enter an integer: "))
if x < 0:
    x = 0
    print 'Negative changed to zero'
elif x == 0:
    print 'Zero'
elif x == 1:
    print 'Single'
else:
    print 'More'
```

No existe switch/case

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 fin de línea
```

Resultado:

```
sota
caballo
rey
```

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

```
lista = range(3)
print lista
for x in lista:
    print x, # La coma evita la impresión del
            # fin de línea
```

Resultado:

```
[0, 1, 2]
0 1 2
```

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

```
>>> range(3)
[0, 1, 2]
>>> range(2,5)
[2, 3, 4]
>>> range(10,0,-1)
[10, 9, 8, 7, 6, 5, 4, 3, 2, 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*

```
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,
...     a=a+1
...
0 1 2 3 4 5 6 7 8 9
```

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

```
#!/usr/bin/python -tt
# -*- coding: utf-8 -*-
a=10
while a > 0:
    print a,
    a=a-1
```

equivale a

```
#!/usr/bin/python -tt
# -*- coding: utf-8 -*-
a=10
while 1:
    print a,
    if a==1:
        break
    a=a-1
```

Sentencia nula: `pass`

Valor nulo: `None`

Funciones

```
#!/usr/bin/python -tt
# -*- coding: utf-8 -*-
def a_centigrado(x):
    """Convierte grados fahrenheit en grados centígrados."""
    return (x-32)*(5/9.0)

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

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

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

```
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 *zona global* del código, haríamos un código similar a este, pero que en python es incorrecto

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

```
#!/usr/bin/python -tt
# -*- coding: utf-8 -*-
l= ["uno","dos"]
def f():
    l=["uno"]
print l # ["uno","dos"]
f()
print l # ["uno","dos"] .
```

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/python -tt
# -*- coding: utf-8 -*-
l= ["uno","dos"]
def f():
    global l
    l=["uno"]
print l # ["uno","dos"]
f()
print l # ["uno"] .
```

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/python -tt
# -*- coding: utf-8 -*-
def f(x,y):
    x=x-1
    y.pop()
```

```
v=3
l= ["uno","dos"]
f(v,l)
print v # 3 . La función creó copia local, no tocó el global
print l # ['uno'] . La función recibió referencia al global
```


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 librería *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

```
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()
fichero.seek(0) # vuelvo al principio del fichero

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

for linea in fichero.readlines():
    print linea,

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,
fichero.close()
```

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

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 intersección con el eje X de la curva  $M\{y=m*x+b\}$   
    """  
    return -b/m
```

Excepciones

- Un programa sintácticamente correcto puede dar errores de ejecución

```
#!/usr/bin/python -tt
# -*- coding: utf-8 -*-
while 1:
    x=int(raw_input("Introduce un n°"))
    print x
```


- Definimos una acción para determinada excepción

```
#!/usr/bin/python -tt
# -*- coding: utf-8 -*-
while 1:
    try:
        x=int(raw_input("Introduce un nº:"))
        print x
    except ValueError:
        // (código que procese la excepción)
        raise Exception("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/python -tt
# -*- coding: utf-8 -*-
try:
    x=int(raw_input("Introduce un n°:"))
    print x
except :      # para cualquier excepción
    raise Exception("Número incorrecto")
```

librería sys

- Argumentos de línea de órdenes

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

```
koji@mazinger:~$ cat ejemplo.py
#!/usr/bin/python -tt
# -*- coding: utf-8 -*-
import sys
print sys.argv[:]
```

```
koji@doublas:~$ ./ejemplo.py un_argumento otro_argumento
['./ejemplo.py', 'un_argumento', 'otro_argumento']
```

(El argumento cero es el nombre del programa)

Escribir en stderr

```
#!/usr/bin/python -tt
# -*- coding: utf-8 -*-
import sys
sys.stderr.write('Error: \n')
```

Leer desde stdin, escribir en stdout

```
#!/usr/bin/python -tt
# -*- coding: utf-8 -*-
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 subprocesso 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 subprocesso
- En caso de error, eleva la excepción `subprocess.CalledProcessError`

```
#!/usr/bin/python -tt
# -*- coding: utf-8 -*-
import subprocess,sys
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
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:
    campos_linea=linea.split()
    print "Usuario:"+ campos_linea[0],
    print "Proceso:"+ campos_linea[7]
```

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

```
#!/usr/bin/python -tt
# -*- coding: utf-8 -*-
import subprocess,sys
mandato="ls inexistente"
mandato_troceado=mandato.split()
try:
    salida=subprocess.check_output(mandato_troceado,
                                   stderr=subprocess.STDOUT)
except subprocess.CalledProcessError as e:
    sys.stderr.write("La orden ha producido el error " +
                    str(e.returncode) + "\n")
```

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/python -tt
# -*- coding: utf-8 -*-
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)
print os.path.exists("/usr/local/noexiste")
```


Enlazar, borrar

```
#!/usr/bin/python -tt
# -*- coding: utf-8 -*-
import os
if not os.path.exists("/tmp/aa"):
    os.mkdir("/tmp/aa")
os.chdir("/tmp/aa")           # cd /tmp/aa
os.link("/etc/hosts","hosts") # crea enlace duro
os.symlink("/etc/hosts","enlace_hosts") # crea enlace blando
os.remove("enlace_duro_hosts")      # borra el fichero
os.remove("enlace_hosts")           # borra el fichero
os.rmdir("/tmp/aa")                 # borra directorio (vacío)
```

copiar, copiar y borrar recursivamente

```
#!/usr/bin/python -tt
# -*- coding: utf-8 -*-
import shutil,os
shutil.copypath("/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 arbol lleno
```

os.walk

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

```
#!/usr/bin/python -tt
# -*- coding: utf-8 -*-
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'])
```

Variables de entorno

```
#!/usr/bin/python -tt
# -*- coding: utf-8 -*-
import os, sys
mi_variable=os.getenv("MI_VARIABLE")
if mi_variable==None:
    msg="ERROR: variable de entorno MI_VARIABLE no definida"
    sys.stderr.write(msg+'\n')
    raise SystemExit
```

Atención: Cuando la shell crea un proceso (p.e. el intérprete de python), puede no pasarle todas las variables de entorno. Por tanto, las variables visibles desde la shell serán distintas a las visibles desde python

Persistencia

Persistencia en Python: La librería *Pickle*

Serializa Objetos

Permite:

- Transmitir objetos, almacenarlos en Disco ó SGBD
- Compartir objetos
- Clases definidas por el usuario y sus instancias

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

cp={28:'madrid',8:'barcelona',33:'asturias'}
fich=open('prueba.pick','w')
pickle.dump(cp,fich)
fich.close()

fich=open('prueba.pick','r')
codigos_postales=pickle.load(fich)
fich.close()

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