



Manual de Python

Tabla de contenidos

1. ¿Qué es Python?	5
2. Variables	5
2.1. ¿Qué es una variable?	5
2.2. Definir una variable	5
2.3. El operador asignación	6
2.4. Tipos de datos en Python	7
2.4.1. El tipo de datos booleanos y las condiciones lógicas	7
2.4.2. Cadenas	9
2.5. Definir varias variables a la vez	9
2.6. Mostrar el valor de las variables en IDLE	10
2.7. Utilizar variables ya definidas	10
2.8. Conversión de tipos	11
2.9. La función type	12
2.10. Intercambio de los valores de dos variables (swap)	13
3. Entrada y salida	13
3.1. Salida por pantalla	13
3.2. Entrada por teclado	14
4. Operaciones aritméticas elementales	15
4.1. Variables numéricas enteras y decimales	15
4.2. Las cuatro operaciones básicas	16
4.2.1. La división en Python	16
4.3. Potencias y raíces	17
4.4. Notaciones compactas	18
5. If... elif... else	18
5.1. If... else	18
5.2. Elif	20
6. Setencias iterativas (bucles)	21
6.1. La sentencia while	21
6.2. La sentencia for	23

6.2.1. La función range.....	23
6.2.2. for.....	24
6.2.3. Ejemplos de for sin la sentencia range.....	25
6.3. La sentencia break.....	25
7. Tipos de datos estructurados.....	26
7.1. Cadenas de caracteres (strings).....	26
7.1.1. Conceptos generales.....	26
7.1.2. Métodos de las cadenas.....	29
7.1.3. ¿Cómo se recorre un string?.....	32
7.1.4. ¿Cómo se recorre un string de forma inversa?.....	33
7.1.5. ¿Cómo buscar un carácter en un string?.....	33
7.2. Listas.....	33
7.2.1. Conceptos básicos.....	33
7.2.2. Añadir elementos a una lista:.....	34
7.2.3 Eliminación de elementos de una lista.....	35
7.2.4. Manipular elementos individuales de una lista.....	35
7.2.5. Manipular sublistas.....	36
7.2.6. La instrucción del.....	37
7.2.7. Copiar una lista.....	37
7.2.8. Recorrer una lista.....	38
7.2.9. Saber si un valor está en una lista.....	38
7.2.10. Forma de manipular els strings continguts en una llista.....	39
7.2.11. Resumen de métodos para listas.....	40
7.3. Tuplas.....	41
7.3.1. Conceptos básicos.....	41
7.4. Diccionarios.....	42
7.4.1. Conceptos básicos.....	42
7.4.2. Resumen de métodos para diccionarios.....	43
8. Subprogramas (subrutinas).....	43
8.1. Introducción.....	43

8.2. Delaración de funciones.....	47
8.3. Declaración de procedimientos.....	48
8.4. Parámetros con valores por defecto.....	48

1. ¿Qué es Python?

Python es un lenguaje de programación interpretado, interactivo y orientado a objetos. A menudo se le compara con Tcl, Perl, Scheme o Java.

Python combina una potencia notable con una sintaxis muy clara. Python tiene módulos, clases, excepciones, tipos de datos dinámicos de muy alto nivel, y tipificado dinámico. Existen interfaces para muchas bibliotecas y llamadas al sistema, así como para varios sistemas de ventanas (X11, Motif, Tk, Mac, MFC). Pueden escribirse fácilmente nuevos módulos en C o en C++. Python puede también utilizarse como lenguaje de extensión para cualquier aplicación que necesite un interface programable.

Python es multiplataforma: existen versiones para muchas variantes de UNIX, para Windows, DOS, OS/2, Mac, Amiga, etc. Si tu sistema favorito no es ninguno de los que se acaban de mencionar, Python también puede funcionar en él, siempre que exista un compilador de C en ese sistema. Pregunta en `comp.lang.python`, o prueba a compilarlo tú mismo.

Python está registrado, pero puedes utilizarlo y distribuirlo libremente, incluso para fines comerciales.

Website: <http://www.python.org>

Entornos de programación: PythonG, DrPython.

Curiosidad: El nombre de Python viene de los Monty Python.

2. Variables

2.1. ¿Qué es una variable?

En Informática, una **variable** es "**algo**" en lo que puedes almacenar información para su uso posterior. En Python, una variable puede almacenar un número, una letra, un conjunto de números o de letras o incluso conjuntos de conjuntos.

2.2. Definir una variable

Las variables en Python se crean cuando se definen, es decir, cuando se les asigna un valor. Para crear una variable, escribe una igualdad con la variable en la izquierda y el valor que quieras darle a la derecha. Los siguientes ejemplos son definiciones de variables:

```
>>> x = 2.5
>>> ABCISA = 4 + 5
>>> Nombre1 = "Pepito Conejo"
>>> _y2 = 'p'
>>> dias_de_la_semana = [ 'Lunes', 'Martes', 'Miércoles', 'Jueves', 'Viernes', 'Sábado', 'Domingo' ]
>>> Fecha_de_nacimiento = [ 'Lunes', 27, 'Octubre', 1997]
>>> año = 1492
SyntaxError: invalid syntax
```

El **nombre de una variable** debe empezar por una letra o por un carácter subrayado (`_`) y puede seguir con más letras, números o subrayados. Puedes utilizar mayúsculas, pero ten en cuenta que Python distingue entre mayúsculas y minúsculas. Es decir, que A y a son para Python variables distintas. Las letras permitidas son las del alfabeto inglés, por lo que están prohibidas la ñ, la ç o las vocales acentuadas. Las **palabras reservadas** del lenguaje (en los ejemplos de este manual aparecerán en naranja) también están prohibidas. En caso de que intentes dar un nombre incorrecto a una variable, Python mostrará un mensaje de error al ejecutar el programa.

Las palabras reservadas de Python son las siguientes:

and	del	for	is	raise
assert	elif	from	lambda	return
break	else	global	not	try
class	except	if	or	while
continue	exec	import	pass	yield
def	finally	in	print	

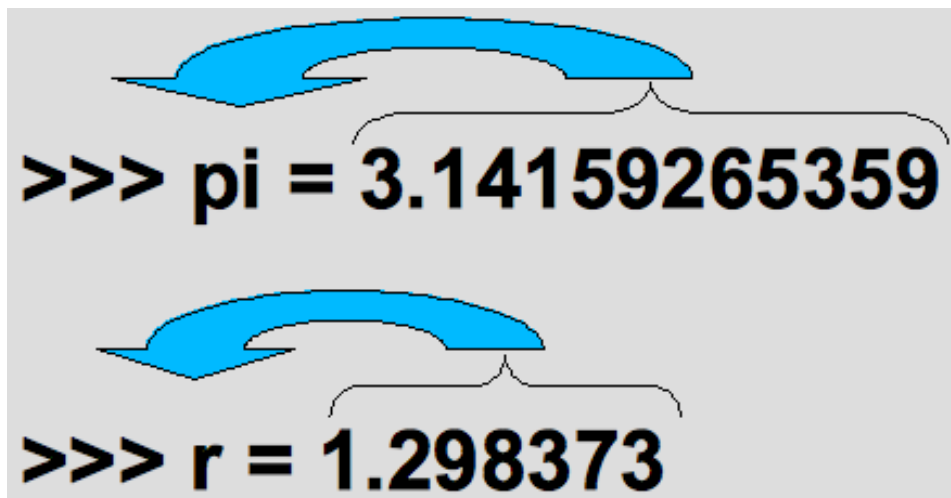
Cada variable se identifica por su nombre, así que en principio **no puede haber dos variables distintas con el mismo nombre**. (La verdad es que sí que puede haber dos variables distintas con el mismo nombre, pero sólo si cada una de las variables "existe" en su propio espacio, separada de la otra. Lo veremos más adelante.)

Aunque no es obligatorio, normalmente **conviene que el nombre de la variable esté relacionado con la información que se almacena en ella**, para que sea más fácil entender el programa. Mientras estás escribiendo un programa, esto no parece muy importante, pero si consultas un programa que has escrito hace tiempo (o que ha escrito otra persona), te resultará mucho más fácil entender el programa si los nombres están bien elegidos. También se acostumbra a utilizar determinados nombres de variables en algunas ocasiones, como irás viendo más adelante, pero esto tampoco es obligatorio.

Al definir una **cadena**, es decir una variable que contiene letras, debes escribir su valor entre comillas (") o apóstrofes ('), como prefieras.

2.3. El operador asignación

El **operador asignación** = nos permite guardar un valor dentro de una variable:



```
>>> pi = 3.14159265359
>>> r = 1.298373
```

2.4. Tipos de datos en Python

En el ejemplo anterior hay definiciones de algunos tipos de variables que hay en Python: **números decimales**, **números enteros**, **cadenas** (una o más letras) y **listas** (conjuntos de números, cadenas o listas).

Aunque se definan de forma similar, para Python no es lo mismo un número entero, un número decimal, una cadena o una lista ya que, por ejemplo, dos números se pueden multiplicar pero dos cadenas no (curiosamente, una cadena sí que se puede multiplicar por un número).

Por tanto, estas tres definiciones de variables **no son equivalentes**:

```
>>> Fecha = 1997
>>> Fecha = 1997.0
>>> Fecha = "1997"
```

En el primer caso la variable Fecha está almacenando un **número entero**, en el segundo **Fecha** está almacenando un **número decimal** y en el tercero Fecha está almacenando una **cadena** de cuatro letras.

Este ejemplo demuestra también que se puede volver a definir una variable cambiando su tipo.

Por tanto, los tipos de variables disponibles en Python (los iremos repasando a lo largo del curso):

Tipo	Clase	Notas	Ejemplo
str	String	Inmutable	'Wikipedia'
unicode	String	Versión Unicode de str	u'Wikipedia'
list	Secuencia	Mutable, puede contener diversos tipos	[4.0, 'string', True]
tuple	Secuencia	Inmutable	(4.0, 'string', True)
set	Conjunto	Mutable, sin orden, no contiene duplicados	set([4.0, 'string', True])
frozenset	Conjunto	Inmutable, sin orden, no contiene duplicados	frozenset([4.0, 'string', True])
dict	Mapping	Grupo de pares claves, valor	{'key1': 1.0, 'key2': False}
int	Número entero	Precisión fija	42
long	Número entero	Precisión arbitraria	42L ó 456966786151987643L
float	Número	Coma flotante	3.1415927
bool	Booleano	Valor booleano verdadero o falso	True

Algunos tipos son bastante avanzados y no los veremos en clase.

2.4.1. El tipo de datos booleanos y las condiciones lógicas

El **tipo booleano** sólo puede tener dos valores: **True** (cierto) y **False** (falso). Estos valores son especialmente importantes para las expresiones condicionales (=condiciones) y los bucles, como veremos más adelante. Antes de la versión 2.2.1, Python no tenía un tipo booleano (se utilizaban los valores enteros 0 y 1).

En realidad el **tipo bool** (el tipo de los booleanos) es una subclase del tipo int.

Estos son los distintos tipos de operadores con los que podemos trabajar con valores booleanos, los llamados **operadores lógicos o condicionales**:

Operador	Descripción	Ejemplo
and	¿se cumple a y b?	r = True and False # r es False
or	¿se cumple a o b?	r = True and False # r es True
not	No a	r = not True # r es False

Los valores booleanos son además el resultado de expresiones que utilizan operadores relacionales (comparaciones entre valores):

Operador	Descripción	Ejemplo
==	¿son iguales a y b?	r = 5 == 3 # r es False
!=	¿son distintos a y b?	r = 5 != 3 # r es True

<	¿es a menor que b?	r = 5 < 3
>	¿es a mayor que b?	r = 5 > 3 # r es True
<=	¿es a menor o igual que b?	r = 5 <= 5 # r es True
>=	¿es a mayor o igual que b?	r = 5 >= 3 # r es True

Las tablas de verdad de los operadores and, or y nor son las siguientes:

• AND			
Op. Esq.	Op. Dreta	Resultat	Coneixeu les lleis d'en De Morgan?? Ja ho heu estudiat!! Cercau a Vikipèdia!
True	True	True	
True	False	False	
False	True	False	
False	False	False	
• OR			
Op. Esq.	Op. Dreta	Resultat	
True	True	True	
True	False	True	
False	True	True	
False	False	False	
• NOT			
Operador	Resultat		
True	False		
False	True		

2.4.2. Cadenas

Las **cadenas** no son más que **texto encerrado entre comillas** simples ('cadena') o dobles ("cadena").

```
>>> 'fiambre huevos'
'fiambre huevos'
>>> 'L\'Hospitalet'
"L'Hospitalet"
>>> "L'Hospitalet"
"L'Hospitalet"
>>> '"Sí," dijo.'
'"Sí," dijo.'
>>> "\"Sí,\" dijo."
'"Sí," dijo.'
>>> '"En L\'Hospitalet," dijo.'
'"En L\'Hospitalet," dijo.'
```

Las cadenas pueden ocupar varias líneas de diferentes maneras. Se puede impedir que el final de línea física se interprete como final de línea lógica mediante usando una barra invertida, por ejemplo:

```
hola = "Esto es un texto bastante largo que contiene\n\
varias líneas de texto, como si fuera C.\n\
    Observa que el espacio en blanco al principio de la línea es\
significativo.\n"
print hola
```


Mostraría lo siguiente:

```
Esto es un texto bastante largo que contiene
varias líneas de texto, como si fuera C.
    Observa que el espacio en blanco al principio de la línea es significativo.
```

Se puede concatenar cadenas (pegarlas) con el operador + y repetirlas con *:

```
>>> palabra = 'Ayuda' + 'Z'
>>> palabra
'AyudaZ'
>>> '<' + palabra*5 + '>'
'<AyudaZAyudaZAyudaZAyudaZAyudaZ>'
```

Más adelante encontraréis un capítulo titulado “manejo de cadenas” donde se ampliará este apartado.

2.5. Definir varias variables a la vez

En una misma línea puedes definir simultáneamente varias variables, con el mismo valor o con valores distintos, como muestra el siguiente ejemplo:

```
>>> a = b = 99
>>> c, d, e = "Mediterráneo", 10, ["pan", "queso"]
```

En el primer caso las dos variables tendrán el mismo valor. En el segundo caso la primera variable tomará el primer valor y así sucesivamente. Si en este segundo caso no coincide el número de variables con el de valores, Python muestra un mensaje de error. Haz la prueba.

2.6. Mostrar el valor de las variables en IDLE

Para que IDLE muestre el valor de una variable, sólo tienes que escribir su nombre. También puedes conocer el valor de varias variables a la vez escribiéndolas entre comas (IDLE las mostrará entre paréntesis), como muestra el siguiente ejemplo:

```
>>> a, b, c = 2, 'pepe', "a"
>>> a
2
>>> c, b
('a', 'pepe')
```

2.7. Utilizar variables ya definidas

Una vez has definido una variable, puedes utilizarla para hacer cálculos o para definir nuevas variables, como muestran los siguientes ejemplos:

```
>>> a = 2
>>> a + 3
5

>>> horas = 1
>>> minutos = 2
>>> segundos = 3
>>> segundos + 60*minutos + 3600*horas
3723
```

```
>>> horas = 5
>>> minutos = 60 * horas
>>> segundos = 60 * minutos
>>> segundos
18000
```

En caso de que utilices una variable no definida anteriormente, Python mostrará un mensaje de error.

Es importante recordar que los nombres de las variables no tienen sentido real para Python, son simples etiquetas para referirse al contenido.

Prueba este ejemplo, en el que el ordenador crea dos variables y las suma, aunque en el mundo real es sabido que no se pueden sumar peras y manzanas:

```
>>> peras = 7
>>> manzanas = 22
>>> peras + manzanas
29
```

Prueba este otro ejemplo:

```
>>> distancia = 100
>>> tiempo = 4
>>> distancia + tiempo
104
>>> distancia * tiempo
400
>>> distancia / tiempo
25
```

En este ejemplo, una vez definidas las variables distancia y tiempo, Python te deja que hagas las operaciones que quieras, **tengan o no sentido físico**. De hecho, en los tres cálculos del ejemplo solamente la división tiene sentido físico (la velocidad).

Puedes modificar el valor que almacena una variable en cualquier momento, pero eso no modificará las otras variables calculadas a partir de ella, como puedes ver en el siguiente ejemplo:

```
>>> a = 10
>>> b = a + 5
>>> a, b
(10,15)
>>> a = 20
>>> a, b
(20,15)
```

Puedes asignar el valor a una variable una variable utilizando su propio valor. Por ejemplo:

```
>>> a = 10
>>> a = a + 5
>>> a
15
```

Fíjate en que **a = a + 5 no es una ecuación** (no tendría solución), **sino una asignación**. Es decir,

Python coge el valor almacenado en la variable `a`, le suma 5, y el resultado lo guarda en la variable `a`. Existe una notación más compacta, que es la siguiente:

```
>>> a = 10
>>> a += 5
>>> a
15
```

2.8. Conversión de tipos

Python proporciona un conjunto de funciones incorporadas que convierten valores de un tipo a otro. La **función** `int` toma un valor y lo convierte en un número entero, si es posible, o si no, se queja:

```
>>> int("32")
32
>>> int("Hola")
ValueError: invalid literal for int(): Hola
```

`int` también puede convertir valores decimales en números enteros, pero recuerda que eso trunca la parte fraccionaria. También debes tener en cuenta que Python está escrito en inglés, por lo que los valores de coma flotante están expresados mediante un punto, y no mediante coma, como sucede en español:

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

La **función** `float` convierte números enteros y cadenas en números decimales:

```
>>> float(32)
32.0
>>> float("3.14159")
3.14159
```

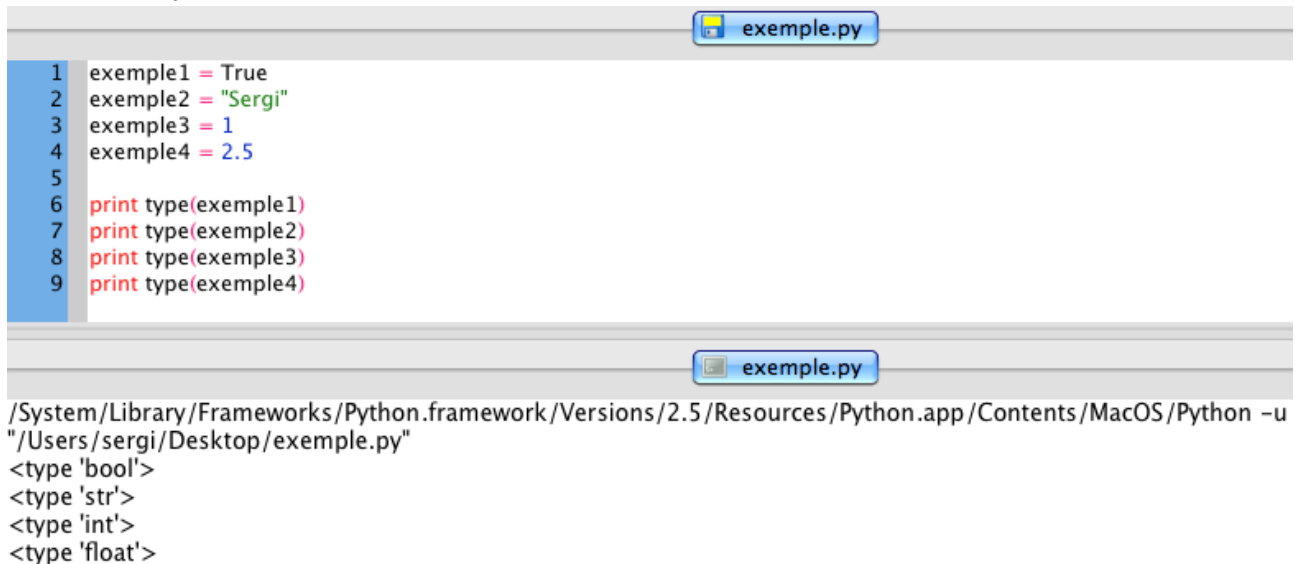
Finalmente, la **función** `str` convierte al tipo cadena:

```
>>> str(32)
'32'
>>> str(3.14149)
'3.14149'
```

Puede parecer raro que Python distinga el valor del número entero 1 del valor de coma flotante 1,0. Ambos representan el mismo número, pero pertenecen a tipos diferentes. El motivo es que están representados de forma diferente dentro del ordenador (¿recordáis la Unidad 1?).

2.9. La función `type`

La función `type` nos retorna el tipo de datos de una variable:



```

1 exemple1 = True
2 exemple2 = "Sergi"
3 exemple3 = 1
4 exemple4 = 2.5
5
6 print type(exemple1)
7 print type(exemple2)
8 print type(exemple3)
9 print type(exemple4)

```

```

/System/Library/Frameworks/Python.framework/Versions/2.5/Resources/Python.app/Contents/MacOS/Python -u
"/Users/sergi/Desktop/exemple.py"
<type 'bool'>
<type 'str'>
<type 'int'>
<type 'float'>

```

2.10. Intercambio de los valores de dos variables (swap)

Para intercambiar los valores de dos variables se necesita una variable temporal:

```

1 a = 23
2 b = 2
3 #anem a intercambiar els valors entre a i b
4 aux = a
5 a = b
6 b = aux

```

3. Entrada y salida

En Informática, la "entrada" de un programa son los **datos que llegan al programa desde el exterior** (normalmente a través del teclado) y la "salida" de un programa son los **datos que el programa proporciona al exterior** (normalmente en la pantalla del ordenador).

3.1. Salida por pantalla

Aunque en IDLE puedas ver el valor de una variable escribiendo simplemente el nombre de la variable, dentro de un programa tienes que utilizar la orden **print**.

Puedes intercalar texto y variables en una misma orden, separándolas con comas, como muestra el siguiente ejemplo:

```

a, saludo = 5, 'Hola'
print 'a contiene el valor' , a, 'y saludo contiene el valor', saludo

```

Si escribes una **coma al final de una orden print**, el siguiente texto se escribirá en la misma línea, como muestra el siguiente ejemplo:

```

corto, medio, largo = 28, 30, 31
print 'Hay siete meses que tienen' , largo , 'días.'
print 'Hay cuatro meses que tienen' , medio , 'días ',
print 'y uno que tiene' , corto , 'o', corto+1, 'días '

```

Como las comillas o los apóstrofes indican el principio o final de una cadena, para incluir **comillas** o apóstrofes dentro de una cadena hay que indicarlo de una forma especial, concretamente escribiendo `\"` o `\'`, como muestra el siguiente ejemplo.

```
print 'Un tipo le dice a otro: \'¿Cómo estás?\'"  
print 'Y el otro le contesta: \'¡Pues anda que tú!\'"
```

Puedes forzar que las variables se escriban como enteros o decimales, utilizando las funciones `int()` o `float()`.

```
>>> print int(2.9)  
2  
>>> print float(2)  
2.0  
>>> print float(2/3)  
0.0
```

3.2. Entrada por teclado

La función `raw_input()` permite que un programa almacene en una variable lo que escribas en el teclado. Al llegar a la función, **el programa se detiene esperando que escribas algo y pulses la tecla Intro**. Prueba el siguiente ejemplo:

```
print '¿Cómo te llamas?'  
nombre = raw_input()  
print 'Me alegro de conocerte,' , nombre
```

Las dos primeras líneas se pueden comprimir en una, escribiendo la cadena como argumento de la función `raw_input()`:

```
nombre = raw_input('¿Cómo te llamas?')  
print 'Me alegro de conocerte,' , nombre
```

Por defecto, la función `raw_input()` convierte la entrada en una cadena. Si quieres que Python interprete la entrada como un número entero, debes utilizar la función `int()` de la siguiente manera:

```
print 'Dime una cantidad en pesetas: ',  
cantidad = int(raw_input())  
print cantidad, 'pesetas son' , cantidad/166.386, 'euros'
```

Fíjate en que al haber una coma al final de la primera línea la entrada se escribe a continuación y no en la línea siguiente.

Para que Python interprete la entrada como un número decimal, debes utilizar la función `float()` de la siguiente manera:

```
print 'Dime una cantidad en euros (hasta con dos decimales): ',  
cantidad = float(raw_input())  
print cantidad, 'euros son' , cantidad*166.386, 'pesetas'
```

Hay otra forma de incluir el valor de variables en una sentencia `print`:

```
#Uso de variables

nombre = "Leo"
edad = 17
pais = "Venezuela"

print "Me llamo %s tengo %d y vivo en %s" % (nombre, edad, pais)
```

Este programa nos mostraría el siguiente resultado por pantalla:

```
Me llamo Leo tengo 17 y vivo en Venezuela
```

Vamos a fijarnos en la siguiente línea de código:

```
print "Me llamo %s tengo %d y vivo en %s" % (nombre, edad, pais)
```

Fijémonos en **%d** y **%s**, tenemos que imaginar esos signos como una “comodín”, donde los ponemos ahí se va a mostrar la primera variable en el orden que está el símbolo. La relación que existe entre estos símbolos y los tipos de datos es la siguiente:

- %s: strings (cadenas)
- %d: enteros
- %f: floats

4. Operaciones aritméticas elementales

En este apartado aprenderás las operaciones aritméticas elementales que incluye Python por defecto. En un apartado posterior haremos un repaso de las operaciones matemáticas que puede incorporar Python al importar el módulo `math`.

4.1. Variables numéricas enteras y decimales

Existen dos tipos de **variables numéricas**: **enteras** (sin decimales) y **decimales** (con decimales). Según cómo escribas el número, Python lo asignará a una variable entera o decimal.

Prueba este ejemplo:

```
>>> a = 5
>>> b = 4.5
>>> c = 12.0
```

En este caso la variable `a` es de tipo entero, `b` es de tipo real y `c` es de tipo real. Fíjate en el “truco” utilizado en el caso de la variable `c`. Si hubieras escrito `>>> c = 12`, `c` sería una variable de tipo entero, pero al escribir `>>> c = 12.0` Python entiende que quieres utilizar una variable de tipo real aunque almacenes un número entero.

Un detalle importante es que si las variables se definen enteras, Python muestra el resultado de los cálculos como números enteros. Si quieres que Python utilice decimales, tienes que definir las variables como reales. Compara la diferencia en el siguiente ejemplo:

```
>>> distancia = 100
>>> tiempo = 3
>>> distancia / tiempo
33
>>> distancia = 100.0
>>> tiempo = 3.0
>>> distancia / tiempo
33.333333333333336
```

En el primer caso el resultado no tiene decimales, mientras que en el segundo caso el resultado sí que sale con decimales (en realidad hubiera sido suficiente con que una de las dos variables tuviera decimales para que el resultado se mostrara con decimales). Observa también que la última cifra del resultado de la división con decimales es incorrecta. Este error se debe a la forma en que Python almacena internamente los números decimales y hay formas de resolverlo. Este problema lo tienen casi todos los lenguajes de programación.

4.2. Las cuatro operaciones básicas

Como has visto en lecciones anteriores, las cuatro operaciones aritméticas básicas son la **suma (+)**, la **resta (-)**, la **multiplicación (*)** y la **división (/)**. Cuando en una fórmula aparecen varias operaciones, Python las efectúa aplicando las reglas usuales de prioridad de las operaciones.

4.2.1. La división en Python

El problema de la división

En las versiones 2.X de Python, el resultado de la división de dos números enteros es un número entero. Para obtener ese número entero, Python trunca (redondea hacia abajo) el resultado de la división. Si no estás atento al escribir el programa, esta forma de funcionar puede provocar a veces errores de cálculo. Por ejemplo:

```
>>> variable = 100
>>> variable / 60 * 60
60
>>> variable * 60 / 60
100
```

Como Python efectúa los cálculos de izquierda a derecha, en el primer caso primero divide ($100 / 60 = 1$ al tratarse de números enteros) y después multiplica ($1 \times 60 = 60$). En el segundo caso primero multiplica ($100 \times 60 = 6000$) y luego divide ($6000 / 60 = 100$).

Para resolver este problema, los creadores de Python han decidido modificar la forma de dividir de Python. El cambio se realizará a partir de la versión 3.0. Así, el resultado de una división será siempre decimal, aunque se dividan números enteros. Mientras tanto, habrá que estar atento.

Cociente de una división

Puedes calcular directamente el cociente de una división entre dos números con el **operador //**. El resultado será de tipo entero o decimal dependiendo del tipo de los números empleados. Por ejemplo:


```
>>> 10//3
3
>>> 10//4
2
>>> 20.0//7
2.0
>>> 20//6.0
3.0
```

Resto de una división

Puedes calcular directamente el resto de la división entre dos números con el **operador %**. Por ejemplo:

```
>>> 10%3
1
>>> 10%4
2
>>> 10%5
0
```

También puedes hacer la división modular entre números decimales, pero normalmente los resultados no son exactos.

```
>>> 10.2%3
1.1999999999999993 #El resultado correcto es 1.2
>>> 10%4.2
1.5999999999999996 #El resultado correcto es 1.6
>>> 10.1%5.1
5.0 #Este resultado sí que es correcto
```

4.3. Potencias y raíces

Para calcular potencias puedes utilizar el símbolo ******, teniendo en cuenta que $x**y = x^y$. Puedes utilizar números negativos o decimales, con lo que esta operación te permite calcular inversas y raíces además de potencias. Aquí tienes unos ejemplos:

```
>>> 2**3
8
>>> 10**-4
0.0001 #Recuerda que  $a^{-b} = 1/a^b$ 
>>> 9**0.5
3.0 #Recuerda que  $a^{1/b}$  es la raíz b-ésima de a
>>> (-1)**0.5 #Esto va a dar error porque es la raíz cuadrada de -1
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in ?
    (-1)**0.5 ValueError: negative number cannot be raised to a fractional power
>>> -9**0.5 #Esto no da error porque hace primero la raíz y luego le pone el -
-3
```

Otra manera de calcular x^y es mediante la función `pow(x,y)`. Si en vez de dos argumentos escribes tres, `pow(x,y,z)`, la función calcula primero x elevado a y y después calcula el resto de la división por z .


```
>>> pow(2,3)
8
>>> pow(4,0.5)
2.0
>>> pow (2,3,5)
3
```

4.4. Notaciones compactas

Para modificar el valor de una variable a partir de su propio valor, en Python se puede utilizar las siguientes notaciones compactas:

Notación	a += b	a -= b	a *= b	a /= b	a **= b	a //= b	a %= b
es equivalente a	a = a + b	a = a - b	a = a * b	a = a / b	a = a ** b	a = a // b	a = a % b

Ampliación de conocimientos con las funciones del **módulo math**: abs(x); complex(real[, imag]); max(s[, args...]); min(s[, args...]); round(x[, n])... Enlace:

<http://www.python.org/doc/2.5.2/lib/module-math.html>

5. If... elif... else...

NOTA: En un principio no iba a explicar “elif” porque me interesa que aprendáis a anidar sentencias if (anidar es “meter” una sentencia if dentro de otra sentencia if).

5.1. If... else...

A menudo te interesará que un programa ejecute unas órdenes cuando se cumplan unas condiciones y otras cuando no. En esos casos se utiliza la orden if ... else En inglés "if" significa "si" (condición) y "else" significa "si no". La orden en Python se escribe así:

```
if condición:
    aquí van las órdenes que se ejecutan si la condición es cierta
    y que pueden ocupar varias líneas
else:
    y aquí van las órdenes que se ejecutan si la condición es
    falsa y que también pueden ocupar varias líneas
```

La primera línea contiene la **condición que quieres evaluar** (revisa el apartado “2.4.1. El tipo de datos booleanos y las condiciones lógicas”). Fíjate en que **la línea termina por dos puntos**.

A continuación viene el **bloque de órdenes** que se ejecutan cuando la condición se cumple (es decir, cuando la condición es verdadera). Fíjate en que todo **el bloque está indentado**. Este detalle es bastante importante, puesto que Python utiliza el indentado para reconocer las líneas que forman un bloque de instrucciones. De hecho, cada vez que acabes una línea con dos puntos, Python indentará las líneas siguientes. Cuando quieras **terminar un bloque, basta con volver al principio de la línea**.

Después viene una **línea con la orden else**, que indica a Python que el bloque que viene a continuación se tiene que **ejecutar cuando la condición no se cumpla** (es decir, cuando sea falsa). Fíjate en que **también hay dos puntos al final de la línea**. En último lugar está el bloque de instrucciones indentado que corresponde al else.

Vamos a ver un ejemplo:

```
edad = int(raw_input('¿Cuántos años tienes?'))
if edad<18:
    print 'Eres menor de edad'
else:
    print 'Eres mayor de edad'
print '¡Hasta la próxima!'
```

Este programa te pregunta cuántos años tienes y almacena la respuesta en la variable "edad". Después comprueba si la edad es inferior a 18 años. Si esta comparación es cierta, el programa escribe que eres menor de edad y si es falsa escribe que eres mayor de edad. Finalmente el programa **siempre se despid**e, ya que la última instrucción está fuera de cualquier bloque y por tanto se ejecuta siempre. **¡Fíjate en la indentación del último print!**

El bloque **else** puede omitirse, como muestra el siguiente ejemplo:

```
numero = int(raw_input('Escribe un número positivo: '))
if numero<0:
    print '¡Te he dicho que escribas un número positivo!'
    print 'Has escrito el número', numero
```

El bloque if no puede omitirse porque contiene la comparación, pero si por algún motivo no quisieras que se ejecutara ninguna orden, el bloque de órdenes del if debe contener al menos la **orden pass** (esta orden le dice a Python que no tiene que hacer nada).

```
edad = int(raw_input('¿Cuántos años tienes?'))
if edad<120:
    pass
else:
    print '¡Mentiroso!'
    print 'Dices que tienes', edad, 'años'
```

Evidentemente este programa podría simplificarse cambiando la desigualdad. Es sólo un ejemplo para mostrarte cómo se utiliza la orden pass. No se debe programar como en el ejemplo anterior.

5.2. Elif...

Cuando no quieras simplemente elegir entre dos opciones, sino entre **varias**, puedes utilizar la orden elif.

```
if condición_1:
    bloque 1
elif condición_2:
    bloque 2
else:
    bloque 3
```

Puedes poner tantos elif como quieras. Si se cumple la condición 1 se ejecuta el bloque 1, si no se cumple la condición 1 pero sí que se cumple la condición 2 se ejecuta el bloque 2 (y así sucesivamente si hubiera varios elif). El bloque else (que es opcional) se ejecuta si no se cumple ninguna de las condiciones anteriores. Vamos a ver un ejemplo:

```
edad = int(raw_input('¿Cuántos años tienes?'))
if edad<0:
    print '¿Todavía no has nacido?'
elif edad<16:
    print 'Legalmente todavía no puedes trabajar'
elif edad<65:
    print 'A esta edad, puedes trabajar'
elif edad<120:
    print 'Supongo que estarás ya jubilado/a'
else:
    print '¿Seguro que tienes', edad, 'años?'
```

El orden en que aparecen las distintas comparaciones es importante, pues cuando se cumple una de las comparaciones, Python ya no considera las siguientes condiciones, aunque se cumplan. En general, se deben escribir primero los casos particulares y después los casos generales.

6. Setencias iterativas (bucles)

6.1. La sentencia while

En inglés, while significa “mientras. La sentencia while se usa así:

```
while condición:
    acción
    acción
    ...
    acción
```

y permite expresar en Python acciones cuyo significado es:

Mientras se cumpla esta condición, repite estas acciones.

La condición puede ser cualquiera de las que se pueden construir siguiendo las reglas dela apartado 2.4. Las sentencias que denotan repetición se denominan **bucles**.

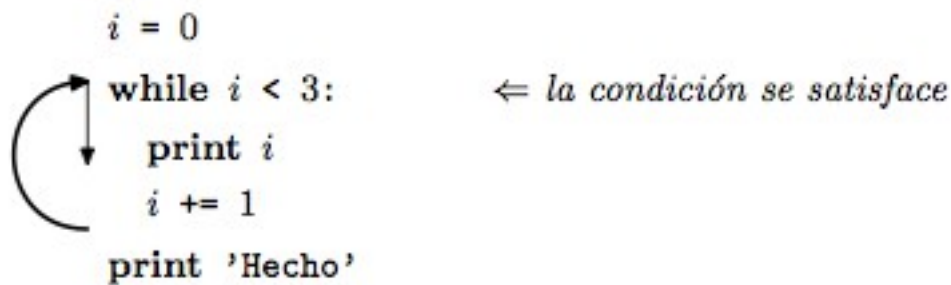
Observemos el siguiente ejemplo:

```
1 i = 0
2 while i < 3:
3     print i
4     i += 1
5 print 'Hecho'
```

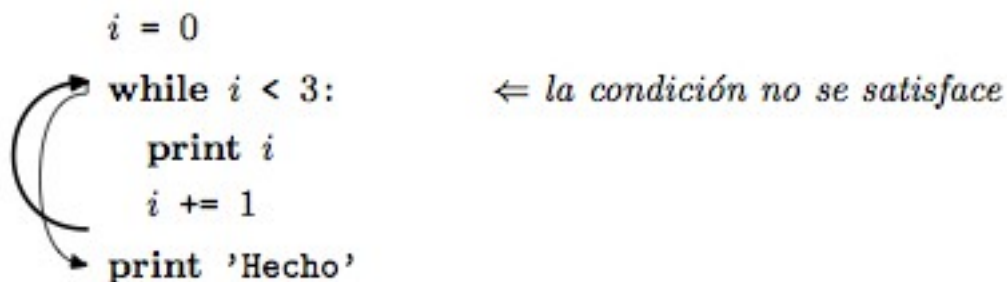
El resultado que muestra por pantalla:

```
0
1
2
Hecho
```

Primero veamos qué pasa cuando la condición se cumple:



Ahora veamos qué pasa cuando la condición no se cumple:



Hacer una **traza del programa** (o **seguimiento de un programa**) sería describir “en papel” lo que hace. Miremos lo que hace en el ejemplo anterior:

• **i = 0**

- imprime por pantalla la cifra 0
- incrementa la variable i en una unidad (que es el valor que guarda i)

• **i = 1**

- imprime por pantalla la cifra 1
- incrementa la variable i en una unidad (que es el valor que guarda i)

• **i = 2**

- imprime por pantalla la cifra 2
- incrementa la variable i en una unidad (que es el valor que guarda i)

• **i = 3**

- se sale del bucle porque no se cumple la condición
- imprime por pantalla el texto “Hecho” (sentencia que está fuera del bucle)

Hacer las trazas de los siguientes ejemplos:

1.

```
edad = 0
while edad < 18:
    edad = edad + 1
    print "Felicidades, tienes " + str(edad)
```

2.

```
1 i = int(raw_input('Valor inicial:'))
2 while i < 10:
3     print i
4     i += 1
```

3.

```
1 i = int(raw_input('Valor inicial:'))
2 limite = int(raw_input('Límite:'))
3 while i < limite:
4     print i
5     i += 1
```

4.

```
1 i = int(raw_input('Valor inicial:'))
2 limite = int(raw_input('Límite:'))
3 incremento = int(raw_input('Incremento:'))
4 while i < limite:
5     print i
6     i += incremento
```

6.2. La sentencia for

6.2.1. La función range

En principio, range se usa con dos argumentos: un valor inicial y un valor final (con matices).

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

Observa que la lista devuelta contiene todos los enteros comprendidos entre los argumentos de la función, incluyendo al primero pero **no al último**.

Fíjate lo que hacen los siguientes ejemplos de range:

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

- ```
>>> range(5) ↓  
[0, 1, 2, 3, 4]
```

- ```
>>> range(2, 10, 2) ↓
[2, 4, 6, 8]
>>> range(2, 10, 3) ↓
[2, 5, 8]
```

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

- ```
>>> range(2, 5, 1) ↓
[2, 3, 4]
>>> range(2, 5) ↓
[2, 3, 4]
```

#### 6.2.2. for

Hay otro tipo de bucle en Python: el bucle for-in, que se puede leer como:

para todo elemento de una serie, hacer...

Un bucle for-in presenta el siguiente aspecto:

```
for variable in serie de valores:
 acción
 acción
 ...
 acción
```

**Las cosas que se pueden hacer con la sentencia for se pueden hacer también con la sentencia while i viceversa.** Vamos el mismo ejemplo que vimos para el while pero ahora con for:

```
for i in range(0, 3):
```

## Manual de Python

```
print i

print 'Hecho'
```

Hagamos la traza del programa anterior:

Hacer una traza del programa (o seguimiento de un programa) sería describir “en papel” lo que hace. Miremos lo que hace en el ejemplo anterior:

### • **i = 0**

- se le asigna el valor 0 a la variable `i`
- imprime por pantalla la cifra 0 (que es el valor que guarda `i`)

### • **i = 1**

- se le asigna el valor 1 a la variable `i`
- imprime por pantalla la cifra 1 (que es el valor que guarda `i`)

### • **i = 2**

- se le asigna el valor 2 a la variable `i`
- imprime por pantalla la cifra 2 (que es el valor que guarda `i`)
- se sale del bucle y se imprime por pantalla el texto “Hecho” (sentencia que está fuera del bucle). **Muy importante: en este momento `i` contiene el valor 2.**

## 6.2.3. Ejemplos de `for` sin la sentencia `range`

Se pueden especificar series de valores sin utilizar la función `range`. Fijaos en los siguientes ejemplos:

```
1 for nombre in ['Pepe', 'Ana', 'Juan']:
2 print 'Hola, %s.' % nombre
```

```
1 numero = int(raw_input('Dame un número: '))
2
3 for potencia in [2, 3, 4, 5]:
4 print '%d elevado a %d es %d' % (numero, potencia, numero ** potencia)
```

## 6.3. La sentencia `break`

La sentencia `break` sirve para salir del bucle de golpe y aunque queden iteraciones por realizar y se cumpla la condición. Veamos un ejemplo:

```
while True:
 entrada = raw_input("> ")
 if entrada == "adios":
 break
 else:
 print entrada
```

Fijaos que la condición es “mientras verdadero”, es decir, se cumple siempre. Se saldrá del bucle anterior cuando se ejecute la sentencia `break`.

Este ejemplo anterior sería equivalente al siguiente bucle:

```
salir = False
while not salir:
 entrada = raw_input()
 if entrada == "adios":
 salir = True
 else:
 print entrada
```

**Está prohibido utilizar break durante la primera evaluación.**

## 7. Tipos de datos estructurados

Hasta el momento hemos tratado con datos de tres tipos distintos: **enteros**, **flotantes** y **cadenas**. Los dos primeros son tipos de datos escalares. Las cadenas, por contra, son tipos de datos secuenciales. Un **dato de tipo escalar es un elemento único**, atómico. Por contra, un **dato de tipo secuencial se compone de una sucesión de elementos** y una cadena es una sucesión de caracteres. **Los datos de tipo secuencial son datos estructurados**. En Python es posible manipular los datos secuenciales de diferentes modos, facilitando así la escritura de programas que manejan conjuntos o series de valores.

### 7.1. Cadenas de caracteres (strings)

#### 7.1.1. Conceptos generales

Además de los números, Python también sabe manipular cadenas, que se pueden expresar de diversas maneras. Se pueden encerrar entre **comillas simples o dobles**, serían equivalentes (ojo que una usa `\` y la otra no):

- `'L\Hospitalet'`
- `"L'Hospitalet"`

Las cadenas pueden ocupar varias líneas de diferentes maneras. Se puede impedir que el final de línea física se interprete como final de línea lógica usando una barra invertida al final de cada línea parcial:

```
hola = "Esto es un texto bastante largo que contiene\n\
varias líneas de texto, como si fuera C.\n\
Observa que el espacio en blanco al principio de la línea es\
significativo."
print hola
```



## Manual de Python

Observa que los saltos de línea se han de incrustar en la cadena con `\n`, pues el salto de línea físico se desecha. El ejemplo mostraría lo siguiente:

```
Esto es un texto bastante largo que contiene
varias líneas de texto, como si fuera C.
Observa que el espacio en blanco al principio de la línea es
significativo.
```

Se puede **concatenar cadenas** (pegarlas) con el operador `+` y repetirlas con `*`:

```
>>> palabra = 'Ayuda' + 'Z'
>>> palabra
'AyudaZ'
>>> '<' + palabra*5 + '>'
'<AyudaZAyudaZAyudaZAyudaZAyudaZ>'
```

Para todo lo que se va a explicar a partir de ahora se asume que se ha definido la siguiente variable: `palabra = 'AyudaZ'`

Se puede **indexar una cadena**. Como en C, el primer carácter de una cadena tiene el índice 0. **No hay un tipo carácter diferente**; un carácter es una cadena de longitud uno. Las subcadenas se especifican mediante la **notación de corte: dos índices separados por dos puntos**.

```
>>> palabra[4]
'a'
>>> palabra[0:2]
'Ay'
>>> palabra[2:4]
'ud'
```

Los índices de corte tienen valores por omisión muy prácticos; si se omite el primer índice se supone cero y si se omite el segundo se supone el tamaño de la cadena sometida al corte.

```
>>> palabra[:2]# Los primeros dos caracteres
'Ay'
>>> palabra[2:]# Todos menos los primeros dos caracteres
'udaZ'
```

He aquí un comportamiento útil en las operaciones de corte: `s[:i] + s[i:]` equivale a `s`.

```
>>> palabra[:2] + palabra[2:]
'AyudaZ'
>>> palabra[:3] + palabra[3:]
'AyudaZ'
```

A diferencia de las cadenas en C, las cadenas de Python no se pueden cambiar. Si se intenta asignar a una posición indexada dentro de la cadena se produce un error:

```
>>> palabra[0] = 'x'
Traceback (most recent call last):
 File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
>>> palabra[1] = 'Choof'
Traceback (most recent call last):
 File "<stdin>", line 1, in ?
TypeError: object doesn't support slice assignment
```

Sin embargo crear una nueva cadena con el contenido combinado es fácil y eficaz:

```
>>> 'x' + palabra[1:]
'xyudaZ'
>>> 'Choof' + palabra[4]
'ChoofZ'
```

Los índices degenerados se tratan con elegancia: un índice demasiado grande se reemplaza por

el tamaño de la cadena, un índice superior menor que el inferior devuelve una cadena vacía.

```
>>> palabra[1:100]
'yudaZ'
>>> palabra[10:]
''
>>> palabra[2:1]
''
```

Los índices pueden ser negativos, para hacer que la cuenta comience por el final. Por ejemplo:

```
>>> palabra[-1] # El último carácter
'Z'
>>> palabra[-2] # El penúltimo carácter
'a'
>>> palabra[-2:]# Los dos últimos caracteres
'aZ'
>>> palabra[:-2]# Todos menos los dos últimos
'Ayud'
```

Pero date cuenta de que -0 es 0, así que ¡no cuenta desde el final!

```
>>> palabra[-0] # (porque -0 es igual a 0)
'A'
```

Los índices de corte negativos fuera de rango se truncan, pero no ocurre así con los índices simples (los que no son de corte):

```
>>> palabra[-100:]
'AyudaZ'
>>> palabra[-10]# error
Traceback (most recent call last):
 File "<stdin>", line 1, in ?
IndexError: string index out of range
```

El mejor modo de recordar cómo funcionan los índices es pensar que apuntan al espacio entre los caracteres, estando el borde izquierdo del primer carácter numerado 0. El borde derecho del último carácter de una cadena de n caracteres tiene el índice n, por ejemplo:

|    |   |    |   |    |   |    |   |    |   |    |   |   |   |   |   |   |   |   |
|----|---|----|---|----|---|----|---|----|---|----|---|---|---|---|---|---|---|---|
| +  | - | -  | + | -  | - | +  | - | -  | + | -  | - | + | - | - | + | - | - | + |
|    | A |    | y |    | u |    | d |    | a |    | z |   |   |   |   |   |   |   |
| +  | - | -  | + | -  | - | +  | - | -  | + | -  | - | + | - | - | + | - | - | + |
| 0  |   | 1  |   | 2  |   | 3  |   | 4  |   | 5  |   | 6 |   |   |   |   |   |   |
| -6 |   | -5 |   | -4 |   | -3 |   | -2 |   | -1 |   |   |   |   |   |   |   |   |

La primera fila de números da la posición de los índices 0..5 de la cadena; la segunda da los índices negativos correspondientes. El corte desde i hasta j consta de todos los caracteres entre los bordes etiquetados i y j, respectivamente.

Para los índices no negativos, la longitud del corte es la diferencia entre los índices, si los dos están entre límites. Por ejemplo, la longitud de palabra[1:3] es 2.

La función interna **len()** devuelve la longitud de una cadena:

```
>>> s = 'supercalifragilisticoexpialidoso'
>>> len(s)
32
```

Otra función interesante es **str**: se le pasa un entero o flotante y devuelve una cadena con una representación del valor como secuencia de caracteres.

Además tenemos la función **ord**: acepta una cadena compuesta por un único carácter y devuelve su código ASCII (un entero). Finalmente la función **chr** acepta un número (código ASCII) y nos devuelve el carácter que representa este número. Veámoslo en un ejemplo:

```
1 print ord("A")
2 print chr(65)
```

65  
A

### 7.1.2. Métodos de las cadenas

Un método, en orientación a objetos, se refiere a una porción de código asociada una clase. No hemos visto nada de esto en esta evaluación, pero básicamente un método se usa de la siguiente forma: `nombre_variable.nombre_metodo([argumentos])`. Un ejemplo de código:

```
cadena = "prova de text"

print cadena.capitalize()

print cadena.upper()

print cadena.find("a")
```

Imprimiria per pantalla el codi anterior:

```
Prova de text

PROVA DE TEXT
```

4

Éstos son los métodos de cadena que soportan tanto las cadenas de caracteres de 8 bit como los objetos Unicode:

#### **capitalize ()**

Devuelve una copia de la cadena con el primer carácter en mayúscula.

#### **center (width)**

Devuelve la cadena centrada en una cadena de longitud width. Se rellena con espacios.

#### **count (sub[, start[, end]])**

Devuelve cuántas veces aparece sub en la cadena S[start:end]. Los argumentos opcionales start y end se interpretan según la notación de corte.

#### **encode ([encoding[,errors]])**

Devuelve una versión codificada de la cadena. La codificación predeterminada es la codificación predeterminada de cadenas. El parámetro opcional errors fija el esquema de gestión de errores. Su valor predeterminado es 'strict', que indica que los errores de codificación harán saltar ValueError. Otros valores posibles son 'ignore' (ignorar) y 'replace' (reemplazar).

#### **endswith (suffix[, start[, end]])**

Devuelve verdadero si la cadena finaliza con el sufijo suffix especificado, en caso contrario falso. Si se da valor al parámetro opcional start, la comprobación empieza en esa posición. Si se da valor al parámetro opcional end, la comprobación finaliza en esa posición.

### **expandtabs ([tabsize])**

Devuelve una copia de la cadena con todos los tabuladores expandidos a espacios. Si no se indica el paso de tabulación tabsize se asume 8.

### **find (sub[, start[, end]])**

Devuelve el menor índice de la cadena para el que sub se encuentre, de tal modo que sub quede contenido en el rango [start, end). Los argumentos opcionales start y end se interpretan según la notación de corte. Devuelve -1 si no se halla sub.

### **index (sub[, start[, end]])**

Como find(), pero lanza ValueError si no se encuentra la subcadena.

### **isalnum ()**

Devuelve verdadero si todos los caracteres de la cadena son alfanuméricos y hay al menos un carácter. En caso contrario, devuelve falso.

### **isalpha ()**

Devuelve verdadero si todos los caracteres de la cadena son alfabéticos y hay al menos un carácter. En caso contrario, devuelve falso.

### **isdigit ()**

Devuelve verdadero si todos los caracteres de la cadena son dígitos y hay al menos un carácter. En caso contrario, devuelve falso.

### **islower ()**

Devuelve verdadero si todos los caracteres alfabéticos de la cadena están en minúscula y hay al menos un carácter susceptible de estar en minúsculas. En caso contrario, devuelve falso.

### **isspace ()**

Devuelve verdadero si todos los caracteres de la cadena son espacio en blanco (lo que incluye tabuladores, espacios y retornos de carro) y hay al menos un carácter. En caso contrario, devuelve falso.

### **istitle ()**

Devuelve verdadero la cadena tiene forma de título (anglosajón) y hay al menos un carácter. En caso contrario, devuelve falso. Se considera que una cadena tiene formato de título si todas sus palabras están en minúsculas a excepción de la primera letra de cada una, que debe ser mayúscula.

### **isupper ()**

Devuelve verdadero si todos los caracteres alfabéticos de la cadena están en mayúscula y hay al menos un carácter susceptible de estar en mayúsculas. En caso contrario, devuelve falso.

### **join (seq)**

Devuelve una cadena formada por la concatenación de todos los elementos de la secuencia seq. Los elementos se separan por la cadena que proporciona el método. Se lanza TypeError si alguno de los elementos no es una cadena.

### **ljust (width)**

Devuelve la cadena justificada a la izquierda en una cadena de longitud width. Se rellena la cadena con espacios. Se devuelve la cadena original si width es menor que len(s).

### **lower ()**

Devuelve una copia de la cadena convertida en minúsculas.

### **lstrip ()**

Devuelve una copia de la cadena con el espacio inicial eliminado.

### **replace (old, new[, maxsplit])**

Devuelve una copia de la cadena en la que se han sustituido todas las apariciones de old por new. Si se proporciona el argumento opcional maxsplit, sólo se sustituyen las primeras maxsplit apariciones.

### **rfind (sub [,start [,end]])**

Devuelve el índice máximo de la cadena para el que se encuentra la subcadena sub, tal que sub está contenido en cadena[start,end]. Los argumentos opcionales start y end se interpretan según la notación de corte. Devuelve -1 si no se encuentra sub.

### **rindex (sub[, start[, end]])**

Como rfind() pero lanza ValueError si no se encuentra sub.

### **rjust (width)**

Devuelve la cadena justificada a la derecha en una cadena de longitud width. Se rellena la cadena con espacios. Se devuelve la cadena original si width es menor que len(s).

### **rstrip ()**

Devuelve una copia de la cadena con el espacio al final suprimido.

### **split ([sep [,maxsplit]])**

Devuelve una lista de las palabras de la cadena, usando sep como delimitador de palabras. Si se indica maxsplit, se devolverán como mucho maxsplit valores (el último elemento contendrá el resto de la cadena). Si no se especifica sep o es None, cualquier espacio en blanco sirve de separador.

### **splitlines ([keepends])**

Devuelve una lista de las líneas de la cadena, dividiendo por límites de línea. No se incluyen los caracteres limitadores en la lista resultante salvo que se proporcione un valor verdadero en keepends.

### **startswith (prefix[, start[, end]])**

Devuelve verdadero si la cadena comienza por prefix, en caso contrario, devuelve falso. Si se proporciona el parámetro opcional start, se comprueba la cadena que empieza en esa posición. Si se proporciona el parámetro opcional end, se comprueba la cadena hasta esa posición.

### **strip ()**

Devuelve una copia de la cadena con el espacio inicial y final suprimido.

## **swapcase ()**

Devuelve una copia de la cadena con las mayúsculas pasadas a minúsculas y viceversa.

## **title ()**

Devuelve una versión con formato título, es decir, con todas las palabras en minúsculas excepto la primera letra, que va en mayúsculas.

## **translate (table[, deletechars])**

Devuelve una copia de la cadena donde se han eliminado todos los caracteres de deletechars y se han traducido los caracteres restantes según la tabla de correspondencia especificada por la cadena table, que debe ser una cadena de longitud 256.

## **upper ()**

Devuelve una copia de la cadena en mayúsculas.

### **7.1.3. ¿Cómo se recorre un string?**

Básicamente el recorrido de un string se debe realizar con la sentencia **for** recorriendo desde la posición 0 hasta la posición de la longitud del string menos 1.

Ejemplo donde se recorre un string y se imprime cada uno de los caracteres que lo forman en líneas separadas.

```
frase = str(raw_input("Introduceixi una frase: "))

for i in range(0, len(frase)):

 print frase[i]
```

### **7.1.4. ¿Cómo se recorre un string de forma inversa?**

Fijaos en la sentencia **range**:

```
frase = str(raw_input("Introduceixi una frase: "))

for i in range(len(frase)-1, -1, -1):

 print frase[i]
```

¿La entendéis?

### **7.1.5. ¿Cómo buscar un carácter en un string?**

Básicamente se haría con un bucle recorriendo desde la posición 0 hasta la posición de la longitud del string menos 1 y comprobando si cada elemento es igual al que buscamos. En caso de encontrar el elemento, estaría justificado salir del bucle usando la sentencia **break**. O bien podríamos usar una variable booleana para indicar que continúe: **"while not encontrado:"**...

Ejemplo: contar el número del carácter "a" que aparecen en una frase.

Leer una frase y contar el número de vocales (de cada una) que aparecen.

```
frase = str(raw_input("Introduceixi una frase: "))

numa = 0

for i in range(0, len(frase)):
```

## Manual de Python

```
if frase[i] == "a":

 numa = numa + 1

print "a=%d" % (numa)
```

### 7.2. Listas

#### 7.2.1. Conceptos básicos

Las listas son conjuntos ordenados de elementos (números o cadenas o listas). Para identificar a las listas se pueden definir variables igualándolas a listas. Estos son ejemplos de listas:

```
>>> laborables = ['Lunes', 'Martes', 'Miércoles', 'Jueves', 'Viernes']
>>> fecha = ['Lunes', 27, 'Octubre', 1997]
>>> peliculas = [['Senderos de Gloria', 1957], ['Hannah y sus hermanas', 1986]]
```

Las listas se pueden concatenar con el símbolo de la suma:

```
>>> vocales = ['E', 'I', 'O']
>>> print vocales
['E', 'I', 'O']
>>> vocales = vocales + ['U']
>>> print vocales
['E', 'I', 'O', 'U']
>>> vocales = ['A'] + vocales
>>> print vocales
['A', 'E', 'I', 'O', 'U']
>>> vocales = vocales + 'Y'
Traceback (most recent call last):
 File "<pyshell#6>", line 1, in ?
 vocales = vocales + 'Y'
TypeError: can only concatenate list (not "str") to list
```

El último ejemplo da error porque 'Y' no es una lista, sino una cadena: **¡no es lo mismo una lista que una cadena!** La forma correcta es la siguiente:

```
>>> vocales = vocales + ['Y']
>>> print vocales
['A', 'E', 'I', 'O', 'U', 'Y']
```

#### 7.2.2. Añadir elementos a una lista:

```
>>> li
['a', 'b', 'mpilgrim', 'z', 'ejemplo']
>>> li.append("nuevo") ❶
>>> li
['a', 'b', 'mpilgrim', 'z', 'ejemplo', 'nuevo']
>>> li.insert(2, "nuevo") ❷
>>> li
['a', 'b', 'nuevo', 'mpilgrim', 'z', 'ejemplo', 'nuevo']
>>> li.extend(["dos", "elementos"]) ❸
>>> li
['a', 'b', 'nuevo', 'mpilgrim', 'z', 'ejemplo', 'nuevo', 'dos', 'elementos']
```

1.**append** añade un único elemento al final de la lista.

2.**insert** inserta un único elemento en una lista. El argumento numérico es el índice del primer elemento que cambia de posición. Observe que los elementos de la lista no tienen por qué ser únicos; ahora hay dos elementos con el valor 'nuevo', li[2] y li[6].

3.**extend** concatena listas. Verá que no se llama a extend con varios argumentos; se le llama con uno, una lista. En este caso, esa lista tiene dos elementos.

### 7.2.3 Eliminación de elementos de una lista

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

1.**remove** elimina la primera aparición de un valor en una lista.

2.**remove** elimina sólo la primera aparición de un valor. En este caso, new aparece dos veces en la lista, pero li.remove("new") sólo elimina la primera.

3.Si el valor no se encuentra en la lista, Python lanza una excepción. Esto imita el comportamiento del método index.

4.**pop** es un animal curioso. Hace dos cosas: elimina el último elemento de la lista, y devuelve el valor que ha eliminado. Advierta que esto es diferente de li[-1], que devuelve el valor sin modificar la lista, y de li.remove(valor), que modifica la lista sin devolver ningún valor.

### 7.2.4. Manipular elementos individuales de una lista



Cada elemento se identifica por su posición en la lista, teniendo en cuenta que se empieza a contar por 0. No se puede hacer referencia a elementos fuera de la lista, aunque también se pueden utilizar números negativos.

```
>>> fecha = [27, 'Octubre', 1997]
>>> print fecha[0], fecha[1], fecha[2]
27 Octubre 1997
>>> print fecha[3]
Traceback (most recent call last):
 File "<pyshell#3>", line 1, in ?
 print fecha[3] IndexError:
 list index out of range
>>> print fecha[-1], fecha[-2], fecha[-3]
1997 Octubre 27
```

Puedes modificar cualquier elemento de la misma manera que modificas cualquier variable.

```
>>> fecha = [27, 'Octubre', 1997]
>>> fecha[2] = 1998
>>> print fecha[0], fecha[1], fecha[2]
27 Octubre 1998
```

### 7.2.5. Manipular sublistas

De una lista puedes extraer sublistas, utilizando la notación `nombre_de_lista [ inicio : límite ]`, donde inicio y límite hacen el mismo papel que en la instrucción `range(inicio, límite)`.

```
>>> dias = ['Lunes', 'Martes', 'Miércoles', 'Jueves', 'Viernes', 'Sábado', 'Domingo']
>>> print dias[1:4] # Se extrae una lista con los valores 1, 2 y 3
['Martes', 'Miércoles', 'Jueves']
>>> print dias[4:5] # Se extrae una lista con el valor 4
['Viernes']
>>> print dias[4:4] # Se extrae una lista vacía
[]
>>> print dias[:4] # Se extrae una lista hasta el valor 4 (no incluido)
['Lunes', 'Martes', 'Miércoles', 'Jueves']
>>> print dias[4:] # Se extrae una lista desde el valor 4 (incluido)
['Jueves', 'Viernes', 'Sábado', 'Domingo']
>>> print dias[:] # Se extrae una lista con todos los valores
['Lunes', 'Martes', 'Miércoles', 'Jueves', 'Viernes', 'Sábado', 'Domingo']
```

Puedes modificar la lista modificando sublistas. De esta manera puedes modificar un elemento o varios a la vez e insertar o eliminar elementos.

```

>>> letras = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']
>>> letras[1:4] = ['X'] # Se sustituye la sublista ['B','C','D'] por ['X']
>>> print letras
['A', 'X', 'E', 'F', 'G', 'H']
>>> letras[1:4] = ['Y', 'Z'] # Se sustituye la sublista ['X','E','F'] por ['Y','Z']
['A', 'Y', 'Z', 'G', 'H']
>>> letras[0:1] = ['Q'] # Se sustituye la sublista ['A'] por ['Q']
>>> print letras
['Q', 'Y', 'Z', 'G', 'H']
>>> letras[3:3] = ['U', 'V'] # Inserta la lista ['U','V'] en la posición 3
>>> print letras
['Q', 'Y', 'Z', 'U', 'V', 'G', 'H']
>>> letras[0:3] = [] # Elimina la sublista ['Q','Y', 'Z']
>>> print letras
['U', 'V', 'G', 'H']

```

### 7.2.6. La instrucción del

La palabra reservada `del` permite eliminar un elemento o varios elementos a la vez de una lista, e incluso la misma lista.

```

>>> letras = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']
>>> del letras[4] # Elimina la sublista ['E']
>>> print letras
['A', 'B', 'C', 'D', 'F', 'G', 'H']
>>> del letras[1:4] # Elimina la sublista ['B', 'C', 'D']
>>> print letras
['A', 'F', 'G', 'H']
>>> del letras # Elimina completamente la lista
>>> print letras
Traceback (most recent call last):
 File "<pyshell#7>", line 1, in ?
 print letras
NameError: name 'letras' is not defined

```

### 7.2.7. Copiar una lista

Compara estas dos secuencias de instrucciones:

```

>>> lista1 = ['A', 'B', 'C']
>>> lista2 = lista1 #Esta línea es distinta en cada ejemplo
>>> print lista1, lista2
['A', 'B', 'C'] ['A', 'B', 'C']
>>> del lista1[1] # Elimina el elemento ['B']
>>> print lista1, lista2
['A', 'C'] ['A', 'C']

```

```

>>> lista1 = ['A', 'B', 'C']
>>> lista2 = lista1[:] #Esta línea es distinta en cada ejemplo
>>> print lista1, lista2
['A', 'B', 'C'] ['A', 'B', 'C']
>>> del lista1[1] # Elimina el elemento ['B']
>>> print lista1, lista2
['A', 'C'] ['A', 'B', 'C']

```

En el primer caso las variables `lista1` y `lista2` hacen referencia a la misma lista almacenada en la memoria del ordenador. Por eso al eliminar un elemento de `lista1`, también desaparece de `lista2`.

Sin embargo en el segundo caso `lista1` y `lista2` hacen referencia a listas distintas (aunque tengan los mismos valores, están almacenadas en lugares distintos de la memoria del ordenador). Por eso, al eliminar un elemento de `lista1`, no se elimina en `lista2`.

### 7.2.8. Recorrer una lista

Puedes recorrer una lista de principio a fin de dos formas distintas. Una forma es recorrer directamente los elementos de la lista:

```
>>> letras = ['A', 'B', 'C', 'D', 'E']
>>> for i in letras:
 print i,
A B C D E
```

La otra forma es recorrer indirectamente los elementos:

```
>>> letras = ['A', 'B', 'C', 'D', 'E']
>>> for i in range(len(letras)):
 print letras[i],
A B C D E
```

La primera forma es más sencilla, pero no permite la modificación de los elementos de la lista. La segunda forma es más complicada, pero permite más flexibilidad: se pueden modificar los elementos, la lista se puede recorrer al revés, etc.

A veces es necesario recorrer la lista en orden inverso, por ejemplo cuando se quieren eliminar elementos de la lista. Para recorrer la lista al revés, basta con utilizar la orden range adecuada.

```
>>> letras = ['A', 'B', 'C', 'D', 'E']
>>> for i in range(len(letras)-1,-1,-1):
 print letras[i],
E D C B A
```

### 7.2.9. Saber si un valor está en una lista

Para saber si un valor está en una lista puedes utilizar el operador in. Por ejemplo, el programa siguiente comprueba si el usuario es una persona autorizada:

```
personas_autorizadas = ['Alberto', 'Carmen']
nombre = raw_input('Dime tu nombre: ')
if nombre in personas_autorizadas:
 print 'Estás autorizado'
else:
 print 'No estás autorizado'
```

También se puede hacer con el método index:

```
>>> li
['a', 'b', 'new', 'mpilgrim', 'z', 'example', 'new', 'two', 'elements']
>>> li.index("example") ❶
5
>>> li.index("new") ❷
2
>>> li.index("c") ❸
Traceback (innermost last):
 File "<interactive input>", line 1, in ?
ValueError: list.index(x): x not in list
>>> "c" in li ❹
0
```

1.**index** encuentra la primera aparición de un valor en la lista y devuelve su índice.

2.**index** encuentra la primera aparición de un valor en la lista. En este caso, new aparece dos veces en la lista, en li[2] y en li[6], pero index devuelve sólo el primer índice, 2.

3.Si el valor no se encuentra en la lista, Python lanza una excepción. Esto es notablemente distinto de la mayoría de los lenguajes, que devolverán un índice no válido. Aunque esto parezca molesto, es bueno, ya que significa que su programa terminará en el lugar donde se origina el problema, y no más adelante cuando se intente utilizar el índice no válido.

4.Para comprobar si un valor está en la lista, utilice in, que devuelve **True** si se encuentra el valor y **False** si no.

#### 7.2.10. Forma de manipular els strings continguts en una llista

Imaginem que tenim la següent llista:

```
paraules = ["formiga", "moix", "elefant", "cavall", "cocodril"]
```

La forma d'accedir a una lletra d'una paraula seria la següent:

```
print paraules[i][j]
```

on j seria l'índex que senyala la paraula i j l'índex que senyala la lletra. Veiem un exemple més concret:

```
print paraules[1]
```

mostraria per pantalla:

o

Ho veis?

#### 7.2.11. Resumen de métodos para listas

**L.append(object)**

Añade un objeto al final de la lista.

**L.count(value)**

Devuelve el número de veces que se encontró value en la lista.

**L.extend(iterable)**

Añade los elementos del iterable a la lista.

**L.index(value[, start[, stop]])**

Devuelve la posición en la que se encontró la primera ocurrencia de value. Si se especifican, start y stop definen una sublista en la que buscar.

**L.insert(index, object)**

Inserta el objeto object en la posición index.

**L.pop([index])**

Devuelve el valor en la posición index y lo elimina de la lista. Si no se especifica la posición, se utiliza el último elemento de la lista.

**L.remove(value)**

Eliminar la primera ocurrencia de value en la lista.

**L.reverse()**

Invierte la lista. Esta función trabaja sobre la propia lista desde la que se invoca el método, no sobre una copia.

**L.sort(cmp=None, key=None, reverse=False)**

Ordena la lista. Si se especifica cmp, este debe ser una función que tome como parámetro dos valores x e y de la lista y devuelva -1 si x es menor que y, 0 si son iguales y 1 si x es mayor que y.

El parámetro reverse es un booleano que indica si se debe ordenar la lista de forma inversa, lo que sería equivalente a llamar primero a L.sort() y después a L.reverse().

Por último, si se especifica, el parámetro key debe ser una función que tome un elemento de la lista y devuelva una clave a utilizar a la hora de comparar, en lugar del elemento en si.

### 7.3. Tuplas

#### 7.3.1. Conceptos básicos

Una tupla es una lista inmutable. Una tupla no puede modificarse de ningún modo después de su creación.

```
>>> t = ("a", "b", "mpilgrim", "z", "example") ❶
>>> t
('a', 'b', 'mpilgrim', 'z', 'example')
>>> t[0] ❷
'a'
>>> t[-1] ❸
'example'
>>> t[1:3] ❹
('b', 'mpilgrim')
```

1.Una tupla se define del mismo modo que una lista, salvo que el conjunto se encierra entre paréntesis en lugar de entre corchetes.

2.Los elementos de una tupla tienen un orden definido, como los de una lista. Las tuplas tienen primer índice 0, como las listas, de modo que el primer elemento de una tupla no vacía es siempre t[0],

3.Los índices negativos cuentan desde el final de la tupla, como en las listas.

4.Las porciones funcionan como en las listas. Advierta que al extraer una porción de una lista, se obtiene una lista nueva; al extraerla de una tupla, se obtiene una tupla nueva.

Las tuplas no tienen métodos:

```
>>> t
('a', 'b', 'mpilgrim', 'z', 'example')
>>> t.append("new") ❶
Traceback (innermost last):
 File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'append'
>>> t.remove("z") ❷
Traceback (innermost last):
 File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'remove'
>>> t.index("example") ❸
Traceback (innermost last):
 File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'index'
>>> "z" in t ❹
1
```

1.Una tupla se define del mismo modo que una lista, salvo que el conjunto se encierra entre paréntesis en lugar de entre corchetes.

2.Los elementos de una tupla tienen un orden definido, como los de una lista. Las tuplas tienen primer índice 0, como las listas, de modo que el primer elemento de una tupla no vacía es siempre `t[0]`,

3.Los índices negativos cuentan desde el final de la tupla, como en las listas.

4.Las porciones funcionan como en las listas. Advierta que al extraer una porción de una lista, se obtiene una lista nueva; al extraerla de una tupla, se obtiene una tupla nueva.

## 7.4. Diccionarios

### 7.4.1. Conceptos básicos

Los diccionarios, también llamados matrices asociativas, deben su nombre a que son colecciones que relacionan una clave y un valor. Por ejemplo, veamos un diccionario de películas y directores:

```
d = {"Love Actually ": "Richard Curtis",
 "Kill Bill": "Tarantino",
 "Amélie": "Jean-Pierre Jeunet"}
```

El primer valor se trata de la clave y el segundo del valor asociado a la clave. Como clave podemos utilizar cualquier valor inmutable: podríamos usar números, cadenas, booleanos, tuplas, ... pero no listas o diccionarios, dado que son mutables.

La diferencia principal entre los diccionarios y las listas o las tuplas es que a los valores almacenados en un diccionario se les accede no por su índice, porque de hecho no tienen orden, sino por su clave, utilizando de nuevo el operador `[]`.

```
d["Love Actually "] # devuelve "Richard Curtis"
```

Al igual que en listas y tuplas también se puede utilizar este operador para reasignar valores.

`d["Kill Bill"] = "Quentin Tarantino"`

## 7.4.2. Resumen de métodos para diccionarios

**D.has\_key(k)**

Comprueba si el diccionario tiene la clave k. Es equivalente a la sintaxis `k in D`.

**D.items()**

Devuelve una lista de tuplas con pares clave-valor.

**D.keys()**

Devuelve una lista de las claves del diccionario.

**D.pop(k[, d])**

Borra la clave k del diccionario y devuelve su valor. Si no se encuentra dicha clave se devuelve d si se especificó el parámetro o bien se lanza una excepción.

**D.values()**

Devuelve una lista de los valores del diccionario.

## 8. Subprogramas (subrutinas)

### 8.1. Introducción

En algunas ocasiones se debe llamar un bloque de código<sup>1</sup> más de una vez, una forma de hacerlo es escribir las instrucciones tantas veces como se necesite (el famoso copy/paste), convirtiéndose de esta manera los programas en un exceso de código lo que conlleva dificultad para descubrir posibles errores. La otra forma es meter las instrucciones en subprogramas que se invocan cada vez que se necesiten. Las funciones que hemos utilizados hasta ahora en Python serían buenos ejemplos de lo que es un subprograma. Por ejemplo la función `pow` nos elevaba un número a otro: `pow(2, 3)` devolvería  $8 (2^3)$ .

En general, en programación, existen dos “tipos” de subprogramas: **procedimientos y funciones**. La diferencia entre un procedimiento y una función es que el primero sólo indica la ejecución de una secuencia de instrucciones, en función de unos parámetros, mientras que la segunda representa un valor que se genera como resultado de su ejecución. Es decir:

- **Una función:** es una subrutina que al término de su ejecución devuelve un valor.
- **Un procedimiento:** es una subrutina que no retorna valor alguno al termino de su ejecución.

En Python sólo disponemos de funciones (aunque podemos hacer funciones que no devuelvan datos con lo que ya tenemos los procedimientos). O dicho de otra forma: se declaran de la misma forma.

En resumen: Las subrutinas son esencialmente partes separadas de código que ejecutan tareas pequeñas de un programa grande.

Veámos un ejemplo con código. Imaginemos que quiero pintar con caracteres la soga de la **práctica del ahorcado**. Con lo que hemos visto hasta ahora en clase podríamos implementarlo de la siguiente forma:

```
num_intents = 6 #num. intentos que lleva el usuario
```

<sup>1</sup> ¿Tenéis claro lo que es un bloque de código?

## Manual de Python

```
print "_____"

print "| |"

if num_intents == 0:

 print "|"

 print "|"

 print "|"

else:

 if num_intents == 1:

 print "| 0"

 print "|"

 print "|"

 else:

 if num_intents == 2:

 print "| 0"

 print "| |"

 print "|"

 else:

 if num_intents == 3:

 print "| 0"

 print "| /\|"

 print "|"

 else:

 if num_intents == 4:

 print "| 0"

 print "| /\| \"

 print "|"

 else:

 if num_intents == 5:

 print "| 0"

 print "| /\| \"

 print "| / \"

 else:

 if num_intents == 6:

 print "| 0"
```



## Manual de Python

```
print "| /\ \"
print "| / \"

print "|"
print "|__"
```

El resultado por pantalla de ejecutar el código anterior sería:

```

|
| O
| /\
| / \"
|__
```

Imaginemos que necesitamos imprimir la soga varias veces. Para hacer esto podríamos hacer un copy/paste del código que pinta la soga y el ahorcado o bien incluir éste en un bucle. Una forma más elegante sería definir una función:

#funcion que pinta la soga y el ahorcado

```
def pinta_ahorcado(num_intents = 0):

 print "_____"

 print "| |"

 if num_intents == 0:

 print "|"

 print "|"

 print "|"

 else:

 if num_intents == 1:

 print "| O"

 print "|"

 print "|"

 else:

 if num_intents == 2:

 print "| O"

 print "| |"

 print "|"

 else:

 if num_intents == 3:

 print "| O"

 print "| /|"

 print "|"

 else:
```

# Manual de Python

```

 if num_intents == 4:

 print "| 0"

 print "| /\ \"

 print "|"

 else:

 if num_intents == 5:

 print "| 0"

 print "| /\ \"

 print "| / "

 else:

 if num_intents == 6:

 print "| 0"

 print "| /\ \"

 print "| / \"

print "|"

print "|_____"

```

```
#ahora podemos llamar a la funcion pasandole como parametro el numero de intentos
```

```
pinta_ahorcado(0)
```

```
pinta_ahorcado(1)
```

```
pinta_ahorcado(2)
```

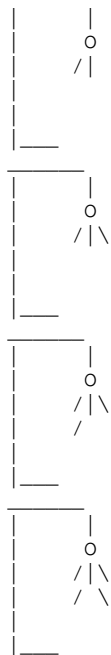
```
pinta_ahorcado(3)
```

```
pinta_ahorcado(4)
```

```
pinta_ahorcado(5)
```

```
pinta_ahorcado(6)
```

El resultado por pantalla de ejecutar el código anterior sería:



### 8.2. Delaración de funciones

Las funciones son declaradas en primer lugar por la palabra reservada `def`. Luego el nombre de la función seguido de los parámetros, separados por comas, que recibirá dicha función. Si ésta no recibe parámetros irán los paréntesis vacíos. Por ultimo van los dos puntos.

El bloque contenido en la función debe terminar con un `return`. Ya que, para que sea función, esta tiene que devolver un valor. Es decir, `return` indica el valor que retornará la función.

Como en la definición de variables en Python, en las funciones no hay que definir el tipo de datos del valor de retorno. Aunque éste irá implícito en el valor que se retorna (como pasaba con las variables).

Un ejemplo de una función sin parámetros seria este:

```
def fx(a,b):
 c = a+b
 return c
```

La función anterior retornaría la suma de los dos valores pasados por parámetro. Por ejemplo: `print fx(2, 5)` imprimiría por pantalla el valor 7.

Un ejemplo de función sin parámetros podría ser:

```
def fx():
 print "Hola mundo!"
 return True
```

### 8.3. Declaración de procedimientos

Es exactamente igual que el de las funciones, pero sin el `return` al final de su bloque de código. De hecho los procedimientos al no agregarle `return` al final del bloque de código, los emulamos en python, con funciones que siempre devolverán el valor especial `None`.

Un ejemplo de procedimiento sin parámetros podría ser:

## Manual de Python

```
def pr():
 print "Hola mundo!"
```

Hagamos inciso en el valor de retorno `None`:

- Ejecutar `pr()` muestra por pantalla:

```
Hola mundo
```

- Ejecutar `print pr()` muestra por pantalla:

```
Hola mundo

None
```

Un ejemplo de procedimiento con parámetros sería:

```
def pr(a,b):
 print a+b
```

### **8.4. Parámetros con valores por defecto**

En Python podremos darle a los parámetros un valor por defecto si es que el usuario de la función o procedimiento no pasa ese parámetro.

Esto se haría de la siguiente forma:

```
def fx(a=0,b=0):
 return a+b
```

La salida de la función `fx` al invocarla sería por tanto:

```
>>> fx(1)

1

>>> fx(1,2)

3

>>> fx(b=2)

2
```

Como vemos en el último ejemplo Python también soporta el paso de parámetros por nombre.

**NOTA IMPORTANTE:** en el **paso de parámetros** a las funciones se dice que un parámetro se puede pasar "**por valor**" o "**por referencia**". Este punto lo explicaremos cuando vemos Visual Basic .Net.