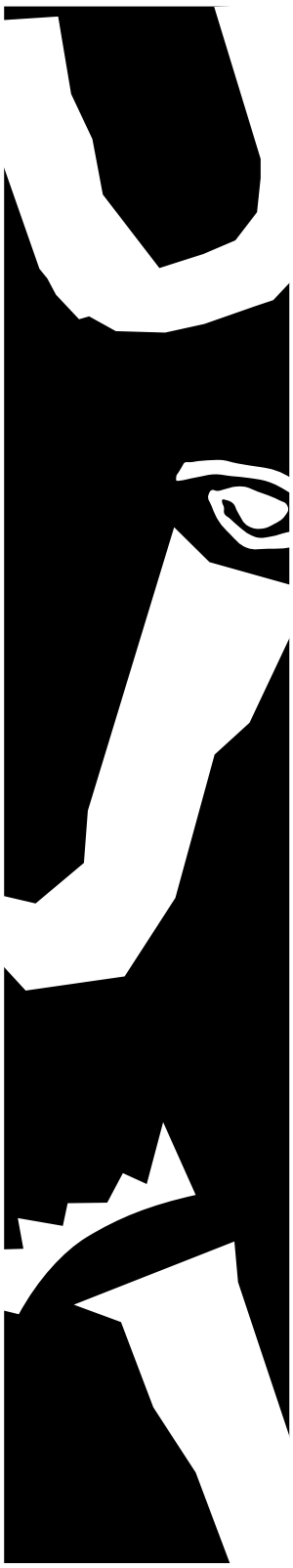


UNIVERSITAT JAUME·I

4º Ingeniería Informática

II26 Procesadores de lenguaje

Python: Conceptos básicos y ejercicios

Índice

1. Introducción

Python es el lenguaje de programación que utilizaremos para las prácticas de la asignatura. Algunas de las características que lo hacen interesante para nosotros son:

- Es fácil de utilizar.
- Es un lenguaje "completo"; no sirve sólo para programar *scripts*.
- Tiene gran variedad de estructuras de datos incorporadas al propio lenguaje.
- Tiene una gran cantidad de bibliotecas (*libraries*).
- Permite la programación modular, orientada a objetos y su uso como un lenguaje imperativo tradicional.
- Es interpretado. Esto facilita el desarrollo (aunque ralentice la ejecución).
- Se puede utilizar desde un entorno interactivo.
- Se puede extender fácilmente.
- Es muy expresivo: un programa Python ocupa mucho menos que su equivalente en C.

Este cuaderno pretende presentar los conceptos básicos de Python y capacitarte para comenzar a utilizarlo. Esto no supone que todas las características interesantes del lenguaje (ni siquiera todas las útiles para la realización de las prácticas) hayan sido exhaustivamente recogidas aquí.[1]

La exposición de los conceptos básicos del lenguaje ocupa la mayor parte de este cuaderno, intercalando en las explicaciones algunos ejercicios propuestos. Posteriormente, en la sección 11 se propone un nuevo ejercicio, mucho más completo que los vistos hasta entonces. En el apéndice A se presentan soluciones a los ejercicios básicos. Finalmente, en el apéndice B, puedes encontrar (junto con las correspondientes respuestas) algunas de las preguntas más frecuentes de los alumnos de cursos pasados.

2. Invocación del intérprete Python

Python suele estar instalado en `/usr/local/bin` o en `/usr/bin`,[2] así que para invocar al intérprete interactivo basta con escribir

```
$ python
```

El sistema arrancará con un mensaje parecido a:

```
Python 2.4.3 (#1, May 24 2008, 13:47:28)
[GCC 4.1.2 20070626 (Red Hat 4.1.2-14)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Esto nos permite empezar a trabajar (`>>>` es el *prompt* del intérprete). Para salir del intérprete interactivo tenemos que pulsar Ctrl+D.

Si hemos almacenado un programa Python en un fichero (p.ej. `fich.py`), podemos ejecutarlo con la orden

```
$ python fich.py
```

[1] En `http://www.python.org` (así como en `http://marmota.dlsi.uji.es/python/`, un *mirror* local) existe material suficiente como para resolver prácticamente cualquier duda que pueda surgir sobre Python.

[2] Puedes averiguar dónde está en tu máquina mediante la orden `which python`.

Si nuestro sistema lo permite, también podemos ejecutar el programa directamente si, además de dar permisos de ejecución al fichero, escribimos como primera línea del fichero los caracteres `#!` seguidos de la ruta completa del intérprete Python. Así, si el intérprete está en `/usr/bin`, escribiríamos:

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

Si el sistema está razonablemente bien instalado, también se puede utilizar:

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

Esto tiene la ventaja de que no depende de la situación exacta de Python (`env` suele estar en `/usr/bin`).

3. Tipos de datos

3.1. Tipos numéricos

Es posible usar Python como una calculadora avanzada. Para ello, una vez arrancado el intérprete, basta con escribir una expresión tras el *prompt*. El intérprete calculará el resultado, lo escribirá y presentará un nuevo *prompt*. Por ejemplo[3]:

```
>>> 2+2
4
>>> 2+4*8
34
>>> (100-20)/6
13
>>> (100-20)%6
2
>>> (100.0-20)/6
13.333333333333334
>>> 2**3
8
>>> 0xFF ^ 0xF0
15
>>> oct(87)
'0127'
```

Python sigue las reglas habituales de precedencia y asociatividad; además, el operador potencia (`**`) es asociativo a derechas y más prioritario que el menos unario (`-2**2` devuelve `-4`).

Para realizar operaciones lógicas tenemos los operadores `and`, `or` y `not`. Los valores que devuelven son:

- El operador `and` devuelve su primer operando si este es falso y el segundo en caso contrario.
- El operador `or` devuelve su primero operando si este es cierto y el segundo en caso contrario.
- El operador `not` devuelve `False` si su operando es cierto y `True` si es falso.

Puede que te preguntes qué es cierto y qué es falso. Para Python, falso es (luego veremos estos tipos con más detalle):

- El valor `False`.
- El valor 0.
- Una secuencia vacía (lista, tupla o cadena).
- Un diccionario vacío.
- El valor `None`.

Todo lo demás es cierto. Las reglas pueden parecer liosas, pero en el contexto de las estructuras de control funcionan como se espera de ellas y simplifican algunas tareas.

[3] El símbolo `^` representa el OR exclusivo y la función `oct` genera la representación octal de un número.

Algunos ejemplos de expresiones lógicas:

```
>>> 2 and 4
4
>>> 0 and 4
0
>>> 2 or 4
2
>>> 0 or 4
4
>>> not 0
True
>>> not 4
False
>>> '2' and 4
4
>>> '' and 4
''
```

También disponemos de operadores sobre bits, que no debes confundir con los anteriores. El "y" binario se representa con `&`, el "o" con `|`, el o-exclusivo con `^`:

```
>>> 2 & 4
0
>>> 2 | 4
6
>>> 2 ^ 4
6
>>> 3 | 2
3
>>> 3 ^ 2
1
```

En cuanto a los tipos de los valores numéricos, Python puede trabajar con *enteros*, *enteros largos* (tienen una precisión ilimitada), *flotantes* y *números complejos*:

Tipo	Ejemplo	Observaciones
Entero	`123`	En decimal
	`0x7b`	En hexadecimal (123 decimal).
	`0173`	En octal (123 decimal).
Entero largo	`12345678901L`	Este valor no se puede representar con un entero.
Flotante	`12.3`	
Complejo	`1+2j`	La `j` puede escribirse en minúscula o mayúscula.

A partir de la versión 2.2 de Python, las operaciones con enteros que provoquen desbordamiento son automáticamente realizadas con enteros largos:

```
>>> 2**30
1073741824
>>> 2**31
2147483648L
```

Como en C, la asignación a variables se realiza mediante el operador `=`.

```
>>> x=20
>>> dolar=0.98
>>> euro=x*dolar
>>> euro
19.600000000000001
```

También como en C, se pueden realizar asignaciones múltiples:

```
>>> x=y=z=0
```

asigna el valor cero a las tres variables. Pero, a diferencia de C, Python no considera las asignaciones como expresiones:

```
>>> 2+(a=3)
Traceback (most recent call last):
  File "<stdio>", line 1
    2+(a=3)
       ^
SyntaxError: invalid syntax
```

Además, pueden realizarse asignaciones concurrentes:

```
>>> x,y=2,3
>>> x
2
>>> y
3
```

Es posible realizar asignaciones aumentadas con un operador binario:

```
>>> a=2
>>> a+=3
>>> a
5
>>> a*=5
>>> a
25
```

Las posibilidades son: `+=`, `-=`, `*=`, `/=`, `%=`, `**=`, `<<=`, `>>=`, `&=`, `^=`, `|=`.

3.2. El valor `None`

Existe un valor especial en Python para representar aquellos casos en los que "no hay valor"; es el valor `None`. Este valor es el único de su tipo. Se interpreta como falso en las expresiones lógicas y es el devuelto por las funciones que no devuelven ningún valor explícitamente. Cuando el resultado de una expresión es `None`, el intérprete no lo muestra:

```
>>> None
>>>
```

3.3. Cadenas

Python también trabaja con cadenas. Para escribirlas, hay que proporcionar una secuencia de caracteres encerrada entre comillas simples o dobles. La elección del delimitador sólo afecta a cómo se escriben las comillas en la cadena: si el delimitador es una comilla doble, las comillas dobles se escribirán escapándolas con una barra invertida (\); análogamente, si el delimitador es una comilla sencilla, las comillas sencillas deberán ir escapadas:

```
>>> "Quiero comprar un CD de Sinead O'Connor"
"Quiero comprar un CD de Sinead O'Connor"
>>> 'Quiero comprar un CD de Sinead O\'Connor'
"Quiero comprar un CD de Sinead O'Connor"
```

Como te habrás imaginado, esto funciona porque la barra invertida es el carácter de escape en Python. Por eso, si queremos escribir una barra invertida, hay que escribirla dos veces (la orden `print` hace que Python escriba el resultado de la expresión correspondiente)[4]:

```
>>> print "a\\b"
a\b
```

En el cuadro 1 puedes ver las secuencias de escape que admite Python. Las secuencias de escape que no son reconocidas no se interpretan; la barra permanece:

```
>>> print "a\p"
a\p
```

[4]Puedes estar preguntándote por qué incluimos aquí la orden `print` y no lo hemos hecho en los ejemplos anteriores. La razón es que el intérprete escribe una representación de la cadena que es a su vez una representación válida en Python para dicha cadena (la puedes utilizar como entrada al intérprete). Sin embargo, `print` escribe la propia cadena, si la entendemos como una secuencia de bytes. Si no acabas de ver la diferencia, prueba a teclear los ejemplos con y sin `print`.

Cuadro 1: Secuencias de escape en las cadenas.

Secuencia	Significado
newline	No se incluye en la cadena (sirve para escribir literales de cadena que ocupen más de una línea).
\\	Barra invertida (*backslash*).
\'	Comilla simple.
\"	Comillas dobles.
\a	Campanilla (*bell*)
\b	Retroceso (*backspace*).
\f	Nueva página.
\n	Nueva línea.
\r	Retorno de carro.
\t	Tabulador horizontal.
\v	Tabulador vertical.
ooo	Carácter ASCII con código octal *ooo*.
\x*hh*	Carácter ASCII con código hexadecimal *hh*.

Si queremos evitar tener que escribir tantas barras, podemos utilizar las denominadas *raw strings*. Simplemente, precede la cadena por una `r` y no se interpretarán las secuencias de escape:

```
>>> print 'a\na'
a
a
```

```
>>> print r'a\na'
a\na
```

Ejercicio 1

Escribe una cadena que se imprima como `'\/"`.

Es posible operar con cadenas: `+` es la concatenación, `*` la repetición y `len` devuelve la longitud de una cadena:

```
>>> ('a'+"b")*3
'abababa'
```

```
>>> len("ab")
2
```

Un operador muy utilizado con las cadenas, especialmente para formatear la salida del programa, es `%`. Como operando izquierdo recibe una cadena con indicaciones de formato similares a las de `printf`. El operando derecho es una expresión o una tupla. El resultado es la cadena resultante de sustituir las marcas de formato en el operando izquierdo por el valor o valores del operando derecho:

```
>>> a= 10
>>> "El resultado es %d" % a
'El resultado es 10'
>>> "%d-%d" % (3,6)
'3-6'
```

```
>>> "%d en hexadecimal es %x" % (123,123)
'123 en hexadecimal es 7b'
>>> "(%f)" % 3.4
'(3.400000)'
```

En el cuadro 2 puedes ver algunos de los caracteres de formato que se pueden emplear.

Cuadro 2: Caracteres de formato para el operador `%` de cadenas.

Carácter	Significado
`d`, `i`	Entero en decimal.
`o`	Entero en octal.
`x`, `X`	Entero en hexadecimal.
`e`, `E`	Número en coma flotante con exponente.
`f`, `F`	Número en coma flotante sin exponente.
`g`, `G`	Número en coma flotante con o sin exponente, según la precisión y la talla del exponente.
`s`	Transforma el objeto en cadena usando `str`.

También es posible aplicar determinados métodos sobre las cadenas[5]. Por ejemplo, si queremos convertir la cadena a mayúsculas o minúsculas podemos hacer:

```
>>> c="una cadena"
>>> c.upper()
'UNA CADENA'
```

```
>>> c.lower()
'una cadena'
```

Algunos de los métodos aplicables a cadenas son:

`capitalize()` devuelve una copia de la cadena con la primera letra en mayúscula.

`center(`n`)` devuelve una copia de la cadena centrada y con longitud n.

`find(`*sub*`,[,`*desde*`[,`*hasta*`]])` devuelve la posición de la primera aparición de *sub* en la cadena; si se incluye *desde*, la búsqueda comienza en esa posición y termina en *hasta*, si se especifica.

`isalnum()` devuelve cierto si la cadena es no vacía y sólo contiene letras y dígitos.

`isalpha()` devuelve cierto si la cadena es no vacía y sólo contiene letras.

`isdigit()` devuelve cierto si la cadena es no vacía y sólo contiene dígitos.

`islower()` devuelve cierto si todas las letras de la cadena son minúsculas y hay al menos una minúscula.

`isspace()` devuelve cierto si la cadena es no vacía y todos sus caracteres son espacios.

`isupper()` devuelve cierto si todas las letras de la cadena son mayúsculas y hay al menos una mayúscula.

`lower()` devuelve una copia de la cadena con las letras convertidas a minúsculas.

`lstrip()` devuelve una copia de la cadena con los blancos iniciales omitidos.

`replace(`v`, `n`)` devuelve una copia de la cadena donde se han sustituido todas las apariciones de la cadena v por n.

`rstrip()` devuelve una copia de la cadena con los blancos finales omitidos.

`split([`s`])` devuelve una lista que contiene las palabras de la cadena. Si se incluye la cadena s, se utiliza como separador.

[5] Más adelante hablaremos con más detalle de los objetos: de momento, considera los métodos como funciones con una "sintaxis peculiar".

`strip()` devuelve una copia de la cadena con los blancos iniciales y finales omitidos.

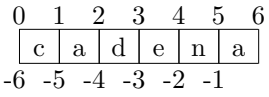
`upper()` devuelve una copia de la cadena convertida a mayúsculas.

3.4. Subsecuencias

Las cadenas son simplemente un tipo especial de secuencias (otro son las listas, que veremos luego). Todas las secuencias pueden ser "troceadas" utilizando la notación de *slices*:

```
>>> c="cadena"
>>> c[3]
'e'
>>> c[3:5]
'en'
>>> c[-1]
'a'
>>> c[2:-2]
'de'
```

Si entre los corchetes sólo hay un número positivo, se devuelve el elemento de la posición correspondiente (se cuenta desde cero). Cuando hay dos números, se devuelve la subsecuencia que comienza en el primero y llega hasta antes del último. Los números negativos se consideran posiciones desde el final de la secuencia. Una manera de ver cómo se elige el subintervalo es la siguiente. Imagina que las posiciones corresponden a marcas *entre* los caracteres:



Para saber qué se obtiene con `c[`i`:`j`]`, basta con mirar entre las marcas correspondientes a i y j. Si se omite i, se asume que es cero; si se omite j, se asume que es la longitud de la cadena:

```
>>> c="cadena"
>>> c[:3]
'cad'
>>> c[3:]
'ena'
```

Una propiedad que es útil saber es que `c[:i]+c[i:]` es igual a `c`.

Ejercicio 2

Asigna a una variable la cadena `esto es un ejemplo`, y haz que se escriba por pantalla la cadena `un ejemplo es esto` utilizando la notación de *slices*.

3.5. Listas

Las listas son secuencias de elementos de *cualquier tipo* separados por comas. Para escribirlas se utiliza una secuencia de expresiones separadas por comas entre corchetes:

```
>>> s=[1,1+1,6/2]
>>> s
[1, 2, 3]
```

Mediante la notación para subsecuencias, es posible acceder a elementos individuales de la lista o partes de ella:

```
>>> s=[1,2,3]
>>> s[1]
2
>>> s[1:-1]
[2]
>>> s[0]=s[2]
>>> s[2]=s[1]/2
>>> s
[3, 2, 1]
```

Fíjate en dos cosas importantes. No es lo mismo una lista de un elemento que el elemento solo[6]. Además, podemos asignar nuevos valores a partes de la lista, tanto a elementos individuales como a subsecuencias:

```
>>> a=[1,2,3,4,5]
>>> a[2:4]
[3, 4]
>>> a[2:4]=[3,4,5,6]
>>> a
[1, 2, 3, 4, 5, 6, 5]
>>> a[2:-1]=[]
>>> a
[1, 2, 5]
```

Observa que no es necesario que las cadenas origen y destino de la asignación tengan la misma longitud; esto se puede emplear para borrar partes de la lista asignándoles la lista vacía (que se representa mediante `[]`).

Como las cadenas, las listas tienen las operaciones `+`, `*` y `len()`:

```
>>> s=[1,2]
>>> s*3
[1, 2, 1, 2, 1, 2]
>>> s+s
[1, 2, 1, 2]
>>> len(s)
2
```

Hemos dicho que los elementos de las listas pueden tener cualquier tipo; en particular, pueden ser listas:

```
>>> s=[1,2]
>>> a=[s,s+[3]]
>>> a
[[1, 2], [1, 2, 3]]
>>> a[1][2]=[1,2,3]
>>> a
[[1, 2], [1, 2, [1, 2, 3]]]
>>> a[1][0]=[[],[2,3]]
>>> a
[[1, 2], [[[], [2, 3]], 2, [1, 2, 3]]]
```

Jugando un poco, podemos hacer que una lista se contenga a sí misma. Python trabaja cómodamente con este tipo de estructuras:

```
>>> a= [1,2]
>>> a[1]=a
>>> a
[1, [...]]
>>> a[1]
[1, [...]]
>>> a[0]
1
>>> a[1][0]
1
```

Ejercicio 3

Convierte la lista `a=[1,[2,[3,4]],5]` en `[1,[2,[3, 4],[6,7]], 5]`.

Las listas son objetos, así que se pueden invocar métodos sobre ellas:

```
>>> a=[1,4,3]
>>> a.sort()
>>> a
[1, 3, 4]
```

Algunos métodos útiles son:

`insert(`i`,`$x`)` inserta el elemento x en la posición i (después de la inserción, x ocupará la posición i).

`append(`$x`)` añade el elemento x al final de la lista (`a.append(x)`es equivalente a `a.insert(len(a),x)`).

`index(`$x`)` devuelve el índice del primer elemento de la lista igual a x.

[6] A diferencia de lo que sucede con las cadenas, ya que un carácter no es sino una cadena de longitud 1.

`remove(`x`)` elimina el primer elemento de la lista igual a x.

`sort()` ordena los elementos de la lista.

`reverse()` invierte el orden de los elementos de la lista.

`count(`x`)` cuenta el número de veces que x aparece en la lista.

Para eliminar elementos de la lista dando su posición en lugar de su valor, se puede utilizar `del`:

```
>>> a=[1,2,3,4,5]
>>> del a[2]
>>> a
[1, 2, 4, 5]
```

```
>>> del a[1:3]
>>> a
[1, 5]
```

También se puede utilizar `del` para eliminar variables completas:

```
>>> x=7
>>> x
7
>>> del x
```

```
>>> x
Traceback (most recent call last):
  File "<stdio>", line 1, in <module>
NameError: name 'x' is not defined
```

Ten cuidado con lo siguiente: al asignar un objeto compuesto a una variable, no se hace una copia, simplemente se almacena una referencia. Esto puede llevar a resultados no deseados. Por ejemplo:

```
>>> a=[1,2,3]
>>> b=a
>>> b[2]=1
```

```
>>> a
[1, 2, 1]
```

Esto también pasa al utilizar variables para crear objetos mayores:

```
>>> b=[1,2]
>>> c=[b,b]
>>> c
[[1, 2], [1, 2]]
```

```
>>> b[0]=2
>>> c
[[2, 2], [2, 2]]
```

Sin embargo:

```
>>> b=[1,2]
>>> c=[b,b]
>>> c
[[1, 2], [1, 2]]
```

```
>>> b=[3]
>>> c
[[1, 2], [1, 2]]
```

Ejercicio 4

¿Puedes explicar lo que pasa en el segundo caso?

3.6. Tuplas

Las tuplas son secuencias inmutables de objetos, el que no puedan modificarse hace que se puedan tratar de una manera más eficiente. Como en el caso de las listas, estos objetos pueden ser

de cualquier tipo, pero no pueden modificarse:

```
>>> a=(1,2,3)
>>> a[1:3]
(2, 3)
>>> a[1]=9
Traceback (most recent call last):
  File "<stdio>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Las tuplas se crean mediante una lista de elementos separados por comas y encerrados entre paréntesis. Esto crea el problema de cómo especificar tuplas de un solo elemento. Para ello basta con escribir el elemento seguido de una coma:

```
>>> a=3,
>>> a
(3,)
```

3.7. Diccionarios

Python permite utilizar diccionarios (también llamados *vectores asociativos*). Estos funcionan como vectores normales en los que los índices pueden ser cualquier tipo de objeto inmutable (números, cadenas y tuplas cuyos componentes sean inmutables). Esto es especialmente útil para aquellos casos en los que la información se indexa de forma natural mediante cadenas.

El diccionario vacío se crea mediante `{}`. Para añadir elementos basta con hacer las correspondientes asignaciones. Después se puede recuperar cualquier elemento a partir de la clave:

```
>>> tel={}
>>> tel["luis"]=1234
>>> tel["maria"]=3456
>>> tel["pepe"]=2323
```

```
>>> print tel["maria"]
3456
>>> tel
{'luis': 1234, 'pepe': 2323, 'maria': 3456}
```

Es un error intentar recuperar la información a partir de una clave que no esté en el diccionario. Para evitarlo, se puede utilizar el método `has_key()`, que devuelve `True` si la clave está y `False` en caso contrario. Además, puedes utilizar el método `keys()` para obtener la lista de claves del diccionario.

```
>>> tel.has_key("pepe")
True
>>> tel.has_key("antonio")
False
>>> tel.keys()
['luis', 'pepe', 'maria']
```

Otra alternativa es utilizar `get`. Este método recibe dos parámetros: el primero es la clave que se va a buscar y el segundo el valor que se devolverá en caso de que no se encuentre. Si no se especifica el segundo parámetro, el valor devuelto al no encontrar la clave es `None`.

```
>>> tel.get('maria', 0)
3456
>>> tel.get('ines', 0)
0
```

```
>>> tel.get('maria')
3456
>>> print tel.get('ines')
None
```

4. Entrada/salida

Como ya has visto, al utilizar Python en modo interactivo, los resultados se muestran directamente por pantalla. Sin embargo, al utilizar Python para interpretar un fichero, es necesario indicarle explícitamente que muestre el resultado. Para ello, disponemos de la orden `print`. Para

utilizarla, se escribe `print` seguido de aquello que queremos mostrar por pantalla. En caso de que sea más de una expresión, estas se separan mediante comas (que provocarán la aparición de los correspondientes espacios):

```
>>> print 2+3, 4+5
5 9
```

Si no queremos que se añada un salto de línea, basta con terminar la orden con una coma.

Podemos emplear el operador `%` para formatear la salida de manera cómoda:

```
>>> n=4
>>> m=0.5
>>> print "Resumen: %d datos, media %g." % (n, m)
Resumen: 4 datos, media 0.5.
```

Para crear un objeto de tipo fichero, se emplea `open`. Normalmente se emplea con dos argumentos: el nombre del fichero y el modo con que se abre (por ejemplo, `"r"` para leer y `"w"` para escribir). Una vez creado un objeto de tipo fichero (llamémosle `f`), se pueden emplear sobre él los siguientes métodos:

`f.read()` lee todo el fichero y lo devuelve en forma de cadena. Si se especifica una talla, leerá como mucho ese número de bytes. Si se ha alcanzado el fin del fichero, devuelve la cadena vacía.

`f.readline()` lee una línea del fichero y la devuelve en forma de cadena, dejando un carácter fin de línea al final, a menos que la última línea del fichero no termine en nueva línea. Si se ha alcanzado el fin del fichero, devuelve la cadena vacía.

`f.readlines()` llama a `f.readline()` repetidamente y devuelve una lista que contiene todas las líneas del fichero.

`f.write(`s`)` escribe la cadena s en el fichero. Devuelve `None`.

`f.close()` cierra el fichero.

Existen otros métodos para los ficheros como `tell`, `seek`, `isatty` o `truncate`. Consulta el manual (Python Library Reference, Built-in Types, File Objects) si necesitas usarlos.

5. Control de flujo

Hasta ahora, el orden de ejecución de las sentencias era secuencial. En Python es posible especificar órdenes distintos mediante estructuras de control de flujo.

5.1. Sangrado

A diferencia de otros lenguajes, en Python no hay marcas explícitas para señalar el comienzo y fin de un bloque de sentencias. Para ello se utiliza el sangrado (*indentation*). ¡Cuidado al copiar programas! La idea básica es que todas las sentencias que forman un bloque subordinado a una construcción tienen un sangrado más profundo que la primera línea de la construcción a la que pertenecen. Antes de un bloque de sentencias que necesitan un nuevo nivel de sangrado, suele aparecer un carácter dos puntos.

Así, las sentencias que se ejecutan cuando la condición de un `if` es cierta están "a la derecha" con respecto a ese `if`. El bloque termina cuando el sangrado es el mismo que el de la construcción o menor, o, en el intérprete interactivo, cuando hay una línea en blanco (dentro de un fichero se pueden dejar líneas en blanco sin afectar al significado del programa).

Para hacer el sangrado se pueden emplear indistintamente espacios en blanco o caracteres de tabulación en cualquier número (siempre que este número sea el mismo dentro de líneas que estén "al mismo nivel" de anidamiento).

5.2. Ejecución condicional

La estructura de control más simple es la construcción `if`. Como es habitual en otros lenguajes de programación, la sentencia `if` tiene asociada una condición y una serie de instrucciones que se ejecutarán en caso de cumplirse aquélla.

Por ejemplo:

```
>>> i=5
>>> if i>4:
...   print "Mayor"
...
Mayor
```

Los tres puntos son el *prompt* secundario: recuerdan que estamos en un nuevo nivel de anidamiento.

Como otros lenguajes, Python ofrece la posibilidad de completar el `if` con un `else`, pero además tiene la abreviatura `elif` que equivale a un `else` seguido de un `if`:

```
>>> i=5
>>> if i>4:
...   print "Mayor"
... elif i==4:
...   print "Igual"
... else:
...   print "Menor"
...
Mayor
```

La condición puede ser cualquier expresión. Para decidir si el resultado es cierto, se siguen las mismas reglas que comentamos en la sección 3.1 para los operadores lógicos:

```
>>> if [1]:
...   print "cierto"
... else:
...   print "falso"
...
cierto
```

```
>>> if []:
...   print "cierto"
... else:
...   print "falso"
...
falso
```

5.3. Instrucción vacía

Python tiene la instrucción `pass` para aquellos casos en que la sintaxis requiera una instrucción, pero no queramos que se emplee. Por ejemplo:

```
>>> i=5
>>> if i<3:
...   print "Pequeño"
... elif i<6:
...   pass
... else:
...   print "Grande"
```

5.4. Bucles

Para realizar bucles, existe la construcción `while` con el significado habitual:

```
>>> a,b=0,1
>>> while b<=8:
...   print b,
...   a,b= b,a+b
...
1 1 2 3 5 8
```

La condición del `while` sigue las mismas reglas que la del `if`:

```
>>> l= range(8)
>>> while l:
...   print l[-1],
...   del l[-1]
...
7 6 5 4 3 2 1 0
```

La construcción `for` permite iterar sobre los valores de una secuencia:

```
>>> a=['primero','segundo','tercero']
>>> for i in a:
...   print i,
...
primero segundo tercero
```

```
>>> for i in "hola":
...   print "(%s)" % i,
...
(h) (o) (l) (a)
```

Con frecuencia es interesante que una variable tome los valores desde el cero hasta un número dado. Para esto, se puede utilizar la función `range` que devuelve una lista que comienza en cero y termina en el número anterior al que se le pasa como parámetro:

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

Si se le pasan dos parámetros, a y b, la lista que se construye va desde a hasta $b-1$. Si se le pasan tres parámetros, a, b y c, la lista va desde a hasta b (sin alcanzarla) en saltos de c (que puede ser negativo):

```
>>> range(3,9)
[3, 4, 5, 6, 7, 8]
>>> range(3,9,5)
[3, 8]
>>> range(3,9,4)
[3, 7]
>>> range(3,9,-4)
[]
>>> range(9,3,-4)
[9, 5]
```

Combinando la función `len` con `range` se pueden hacer bucles sobre los índices de una secuencia:

```
>>> a="-"*3+">"
>>> for i in range(len(a)):
...   print a[-i-1:]
...
>
->
-->
--->
```

En ocasiones, queremos saber tanto el elemento como el índice sobre el que estamos iterando. Podemos hacerlo mediante `enumerate`:

```
>>> a="hola"
>>> for pos,c in enumerate(a):
...   print pos,c
...
0 h
1 o
2 l
3 a
```

Como en C, es posible salir de un bucle mediante `break` y pasar a la iteración siguiente con `continue`:

```
>>> a=[1,2,4,0,3]
>>> for i in a:
...     if i==0:
...         break
...     print i,
...
1 2 4
```

```
>>> for i in a:
...     if i== 0:
...       continue
...     print i,
...
1 2 4 3
```

Ejercicio 5

Escribe un programa que utilice un bucle para crear una lista con los números del 1 al 10 y luego la muestre por pantalla.

Ejercicio 6

Escribe un programa que, a partir de una tupla cualquiera, obtenga una lista con todos los elementos de la tupla. Utiliza un bucle.

Ejercicio 7

Escribe un programa que muestre por pantalla los números múltiplos de 7 entre el 1 y el 1000. Utiliza `range(1001)` en un bucle `for` con los `if` necesarios. Después haz lo mismo empleando un `range` con tres parámetros.

6. Funciones

Python permite definir funciones con una sintaxis muy sencilla: tras la palabra reservada `def` se escribe el nombre de la función, el nombre de los parámetros entre paréntesis y dos puntos. Las líneas que se encuentran a continuación formarán el cuerpo de la función. Por ejemplo, si en el fichero `fibonacci.py` tenemos:

```
#!/usr/bin/env python
def fib(n): # Números de Fibonacci
   a,b=0,1
   r=[] # r tendrá el resultado
   while b<n:
      r.append(b)
      a,b=b,a+b
   return r
print fib(10)
```

Al ejecutarlo obtenemos `[1, 1, 2, 3, 5, 8]`. En este ejemplo podemos ver varias cosas interesantes. En primer lugar, que los comentarios comienzan por el símbolo `#` y terminan al finalizar la línea. Por otro lado, vemos que se pueden utilizar variables dentro de la función. Estas variables son locales, a menos que se declaren explícitamente como globales (con `global`). Por ejemplo, el siguiente programa escribe `4 3`:

```
a,b =2,3
def algo():
   global a
   a,b=4,4
algo()
print a,b
```

Por otro lado, los parámetros se pasan por *referencia a objeto*. En la práctica, esto quiere decir que los parámetros de tipo elemental (numérico *o cadena*), se pasan por valor:

```
>>> def cambia(n):
...   n= 3
...
>>> a=9
>>> cambia(a)
>>> print a
9
>>> a="hola"
>>> cambia(a)
>>> print a
hola
```

Sin embargo, si el parámetro tiene un tipo compuesto, se pasa por algo similar a referencia:

```
>>> def cambia(n):
...     n[1]=2
...
>>> a=[3,3,3]
>>> cambia(a)
>>> print a
[3, 2, 3]
```

Esto es equivalente a lo que pasa con la asignación de variables (recuerda lo que se comentó en la sección 3.5).

Ejercicio 8

Reescribe el programa del ejercicio 7 utilizando una función a la que se le pase un número y devuelva `True` si es múltiplo de 7, o `False` si no lo es.

6.1. Un ejemplo

Vamos a utilizar los conceptos anteriores para crear un árbol binario de búsqueda. Como ya sabes, un árbol binario de búsqueda es un árbol en el que todos los elementos situados en el subárbol izquierdo de un nodo son menores (en un cierto orden) que el situado en ese nodo y todos los del subárbol derecho son mayores (no permitiremos repeticiones).

Como representación utilizaremos tuplas de tres elementos: el primer elemento será el subárbol izquierdo, el segundo la información del nodo y el tercero el subárbol derecho. Para el árbol vacío utilizaremos `None`.

Crear un árbol es sencillo:

```
def crea_arbol():
    return None
```

Para insertar un elemento, hacemos una búsqueda binaria:

```
def inserta(A,c):
   if A==None: # Árbol vacío
     return (None,c,None)
   elif c<A[1]: # Almacenarlo a la izquierda
     return (inserta(A[0],c),A[1],A[2])
   elif c>A[1]: # Almacenarlo a la derecha
      return (A[0],A[1],inserta(A[2],c))
   else:
      return A
```

Comprobar si un elemento pertenece al árbol es trivial:

```
def pertenece(A,c):
   if A==None:
      return True
   elif c==A[1]:
      return False
   elif c<A[1]:
      return pertenece(A[0],c)
   else:
      return pertenece(A[2],c)
```

Vamos a probarlo:

```
A= crea_arbol()
A= inserta(A,"diminuto")
A= inserta(A,"pequeño")
A= inserta(A,"mediano")
A= inserta(A,"grande")
```

```
A= inserta(A,"diminuto")
print A
print pertenece(A,"ínfimo")
print pertenece(A,"grande")
```

escribe

```
(None, 'diminuto', (((None, 'grande', None), 'mediano', None), 'pequeño', None))
False
True
```

6.2. Parámetros con valores por defecto

Es posible escribir funciones con parámetros que tienen determinados valores cuando no se especifican en la llamada (valores por defecto). Para ello se escribe, tras el nombre del parámetro, el valor que tiene cuando no aparece en la llamada. Por ejemplo:

```
def convierte(n,base=10):
   if n==0:
       return [0]
   elif n<0:
       n=-n
       neg=True
   else:
       neg=False
   r= []
   while n>0:
       r.append(n % base)
       n=n/base
   if neg:
       r.append('-')
   r.reverse()
   return r
```

La llamada `convierte(123)` devuelve `[1,2,3]` mientras que la llamada `convierte(123,8)` devuelve `[1,7,3]`.

6.3. Algunas funciones predefinidas

En esta sección veremos algunas de las funciones predefinidas en Python.

`abs(`x`)` valor absoluto.

`chr(`i`)` devuelve una cadena de un solo carácter: aquél cuyo código ASCII es i.

`cmp(`x`,`y`)` compara x e y, devolviendo 0 si son iguales, un valor negativo si $x < y$ y un valor positivo si $x > y$.

`eval(`s`)` evalúa s interpretándola como una expresión Python.

`filter(`f`,`l`)` devuelve una lista formada por los elementos de la lista l que hacen que f devuelva cierto.

`float(`x`)` convierte un número entero a coma flotante.

`hex(`x`)` devuelve una cadena con la representación en hexadecimal del entero x.

`int(`x`)` convierte un número o una cadena en entero.

`len(`s`)` devuelve la longitud de la secuencia s.

`long(`x`)` convierte un número o una cadena en entero largo.

`max(`s`)` devuelve el máximo de la secuencia s.

`min(`s`)` devuelve el mínimo de la secuencia s.

`oct(`x`)` devuelve una cadena con la representación en octal del entero x.

`open(`f`[,`m`[,`b`]])` devuelve un objeto de tipo fichero ligado al fichero de nombre f (una cadena). El modo de apertura se indica mediante m, que puede ser: `"r"` para lectura (valor por defecto), `"w"` para escritura, `"r+"` para lectura y escritura y `"a"` para escritura al final del fichero. Si se incluye, b determina el tipo de *buffer* que se utilizará: 0, sin *buffer*; 1, con *buffer* de línea; y cualquier otro valor indica un *buffer* de (aproximadamente) ese tamaño.

`ord(`c`)` devuelve el código ASCII del carácter c.

`pow(`x`,`y`)` equivale a x`**`y.

`range([`a`,]`b`[,`c`])` devuelve la lista `[`a`, `$a+c$`, `$a+2c$`, ... ]`. El valor por omisión de a es 0, el de c es 1. Si c es positivo, el último elemento de la lista es el mayor $a + ci$ que es menor que b; si c es negativo, el último elemento de la lista es el menor $a + ci$ que es mayor que b. Con dos argumentos, se supone que el omitido es c.

`repr(`v`)` devuelve una cadena r (la representación de v) tal que `eval(`r`)` sea igual a v.

`str(`v`)` devuelve una cadena r que representa a v de manera "legible" pero que puede no ser aceptable para `eval`.

`tuple(`s`)` devuelve una tupla construida con los elementos de la secuencia s.

`type(`v`)` devuelve el tipo de v.

`xrange([`a`,]`b`[,`c`])` tiene el mismo efecto que `range` al usarlo en un bucle, pero no crea una lista en memoria.

7. Bibliotecas

Una de las ventajas de Python es la gran cantidad de objetos y funciones que hay disponibles en forma de bibliotecas (o módulos). Para acceder a las funciones de una biblioteca, tenemos varias opciones:

1. Importar el módulo en cuestión y después acceder a la función mediante el nombre de la biblioteca. Así, para utilizar la función `sin`, que está en `math`, haríamos:

```
import math
print math.sin(2.0)
```

2. Importar la función del módulo y utilizarla directamente:

```
from math import sin
print sin(2.0)
```

3. Importar todas las funciones del módulo y utilizarlas directamente:

```
from math import *
print sin(2.0)
```

Es aconsejable hacer esto únicamente si sabemos que el módulo está preparado para ello, en otro caso, puede haber problemas.

Si queremos importar varios módulos en una sola sentencia, podemos separarlos por comas:

```
import sys, math, re
```

Algunas de las bibliotecas que pueden ser útiles para la asignatura son:

`sys`	da acceso a variables del intérprete y a funciones que interactúan con él.
`types`	define nombres para los tipos de objetos del intérprete.
`string`	operaciones sobre cadenas.
`re`	expresiones regulares.
`math`	funciones matemáticas.
`tempfile`	creación de ficheros temporales.

7.1. La biblioteca `sys`

`exit(`n`)` aborta la ejecución y devuelve el valor n a la *shell*.

`argv` lista de los argumentos que se le pasan al programa.

Ejemplo: el siguiente programa escribe los argumentos con los que se le llama:

```
import sys

for i in sys.argv:
   print i
```

`stdin, stdout, stderr` objetos de tipo fichero que corresponden a la entrada estándar, salida estándar y salida de error. Sobre ellos pueden invocarse los métodos explicados en la sección 4.

7.2. La biblioteca `types`

Define nombres para los tipos que proporciona el intérprete. Por ejemplo: `NoneType`, para el tipo de `None`; `IntType`, para el tipo entero; `LongType`, para el tipo entero largo; `FloatType`, el tipo real; etc.

La siguiente función recibe como parámetros una lista y un segundo parámetro. Si este parámetro es un entero, borra de la lista el elemento de la posición correspondiente; si no es entero, elimina el primer elemento igual a él:

```
from types import *
def delete(list, item):
   if type(item) is IntType:
       del list[item]
   else:
       list.remove(item)
```

7.3. La biblioteca `string`

Esta biblioteca define diversas variables y funciones útiles para manipular cadenas. Muchas de sus funciones han sido sustituidas por métodos sobre las cadenas.

Entre las variables tenemos:

`digits`	`'0123456789'`.
`hexdigits`	`'0123456789abcdefABCDEF'`.
`letters`	Letras minúsculas y mayúsculas.
`lowercase`	Letras minúsculas.
`octdigits`	`'01234567'`.
`uppercase`	Letras mayúsculas.
`whitespace`	Blancos (espacio, tabuladores horizontal y vertical, retorno de carro, nueva línea y nueva página)

En cuanto a las funciones:

`atof(`s`)` convierte la cadena s en un flotante.

`atoi(`s`,[`b`])` convierte la cadena s en un entero (en base b si se especifica).

`atol(`s`,[`b`])` convierte la cadena s en un entero largo (en base b si se especifica).

`lower(`s`)` pasa s a minúsculas.

`join(`l`[,`p`])` devuelve la cadena que se obtiene al unir los elementos de la lista l separados por blancos o por p si se especifica.

`split(`s`[,`p`])` devuelve la lista que se obtiene al separar la cadena s por los blancos o por p si se especifica.

`strip(`s`)` devuelve la cadena que se obtiene al eliminar de s sus blancos iniciales y finales.

`upper(`s`)` pasa s a mayúsculas.

Ejemplo: la siguiente función recibe como parámetros una cadena y un entero y devuelve la palabra indicada por el entero pasada a mayúsculas:

```
def mayus(c,i):
   l=string.split(c)
   return string.upper(l[i])
```

7.4. La biblioteca `re`

Esta biblioteca define las funciones adecuadas para trabajar con expresiones regulares. Un aviso: existen otras dos bibliotecas (`regex` y `regsub`) con la misma utilidad, sin embargo están siendo abandonadas (son "obsolescentes").

En el cuadro 3(a) puedes ver algunos de los elementos que se pueden emplear para escribir expresiones regulares.

El carácter barra invertida tiene un significado especial dentro de las expresiones regulares. Si el carácter siguiente es uno de los de la tabla anterior, este pierde su significado y pasa a representarse a sí mismo (así `\+` representa el carácter `+` en lugar de una o más barras). En el cuadro 3(b) puedes ver el significado de algunas secuencias de escape.

A la hora de escribir expresiones regulares hay que tener en cuenta lo que se comentó en la sección 3.3 acerca de los caracteres de escape. Así, para conseguir `\d` hay que escribir `"\\d"` y para conseguir una barra, hay que escribir ¡cuatro!. Por eso merece la pena utilizar *raw strings*.

Entre las funciones que ofrece este módulo, tenemos:

`match(`p`,`s`)` devuelve una instancia de `MatchObject` si cero o más caracteres del comienzo de s concuerdan con la expresión regular p. En caso contrario devuelve `None`.

`search(`p`,`s`)` devuelve una instancia de `MatchObject` si cero o más caracteres de s (no necesariamente del principio) concuerdan con la expresión regular p. En caso contrario devuelve `None`.

`compile(`p`)` devuelve una versión compilada de p. Esto es útil si se va a utilizar la misma expresión muchas veces. Sobre el objeto resultante es posible llamar a los métodos `match` y `search` con el mismo significado que antes. Por ejemplo:

```
>>> import re
>>> id=re.compile("[a-zA-Z][a-zA-Z0-9]*")
>>> if id.match("var23"):
...     print "sí"
... else:
...     print "no"
...
sí
>>>
```

Los objetos de la clase `MatchObject` (los que se devuelven si las búsquedas tienen éxito) guardan los "grupos referenciables" que se han encontrado. Para recuperarlos se utiliza el siguiente método:

`group(`n`)` devuelve el valor del n-ésimo grupo referenciable.

```
>>> import re
>>> x=re.compile("(\w+)@([a-zA-Z.]+)")
>>> m=x.match("guido@python.org")
>>> print m.group(1)
guido
>>> print m.group(2)
python.org
```

Además de comprobar si un determinado texto aparece en la cadena, se pueden hacer sustituciones de texto y dividir la cadena mediante expresiones regulares utilizando las siguientes funciones:

`sub(`p`, `r`, `s`)` sustituye en la cadena s las apariciones de la expresión regular p (que puede estar compilada) por la cadena r.

`split(`p`,`s`)` tiene la misma función que `string.split` pero permite utilizar una expresión regular (que puede estar compilada) como separador.

Cuadro 3: Algunos componentes de las expresiones regulares.

(a) Operadores:

	Explicación	Ejemplo	
`.`	Cualquier carácter excepto el de nueva línea.	`a.b`	Cadena de tres caracteres que empieza por `a` y termina por `b`.
`^`	Comienzo de cadena.	`^a..`	Cadena que empieza por `a`, tiene tres caracteres y está al comienzo.
`$`	Fin de cadena.	`a..$`	Cadena que empieza por `a`, tiene tres caracteres y está al final.
`|`	Disyunción.	`a|b`	La letra `a` o la `b`.
`*`	Clausura de la expresión de su izquierda.	`a*`	Cero o más `a`es.
`+`	Clausura positiva de la expresión de su izquierda.	`a+`	Una o más `a`es.
`?`	Opcionalidad de la expresión de la izquierda.	`a?`	Una o cero `a`es.
`{`m`,`n`}`	Entre m y n repeticiones de la expresión a su izquierda.	`a{3,5}`	Entre tres y cinco `a`es.
`[]`	Conjunto de caracteres. Los caracteres pueden enumerarse uno a uno o utilizar un guion (`-`) entre dos caracteres para expresar la idea de rango. Si quiere incluirse en el conjunto el carácter `]`, este ha de ser el primero de la enumeración, y lo mismo sucede con `-` (o se pueden utilizar las notaciones `\]` y `\-` en cualquier posición). Si el primer carácter es `^`, se considera el complementario del conjunto de caracteres enumerados.	`[ac2-5]`	Una `a`, una `c`, un `2`, un `3`, un `4` o un `5`.
`()`	Cambia el orden de aplicación de las expresiones. Además, crea un "grupo referenciable".	`(a|b)+`	Una secuencia de una o más `a`es y `b`es.

(b) Secuencias de escape:

Secuencia	Significado
`\\`	Corresponde al propio carácter barra invertida.
`\`n	Representa al "grupo referenciable" con el número n. Por ejemplo: `(.+)-\1` incluye a la cadena `11-11` pero no a `12-21`.
`\b`	Cadena vacía al final de una palabra. Las palabras son secuencias de caracteres alfanuméricos que tienen "pegadas" dos cadenas vacías.
`\B`	Una cadena vacía pero que no está ni al principio ni al final de una palabra.
`\s`	Espacios en blanco, equivale a `[ \t\n\r\f\v]`
`\S`	El complementario del anterior.
`\d`	Dígitos, equivale a `[0-9]`.
`\D`	No dígitos, equivale a `[^0-9]`.
`\w`	Caracteres que pueden estar en una palabra, equivale a `[a-zA-Z0-9_]`.
`\W`	Complementario del anterior.

7.5. La biblioteca `math`

Define una serie de funciones matemáticas idénticas a las de la librería estándar de C:

`acos(`x`)`, `asin(`x`)`, `atan(`x`)`, `atan2(`x`, `y`)`, `ceil(`x`)`, `cos(`x`)`, `cosh(`x`)`, `exp(`x`)`, `fabs(`x`)`, `floor(`x`)`, `fmod(`x`, `y`)`, `frexp(`x`)`, `hypot(`x`, `y`)`, `ldexp(`x`, `y`)`, `log(`x`)`, `log10(`x`)`, `modf(`x`)`, `pow(`x`, `y`)`, `sin(`x`)`, `sinh(`x`)`, `sqrt(`x`)`, `tan(`x`)`, `tanh(`x`)`.

También define las constantes `pi` y `e`.

7.6. La biblioteca `tempfile`

Define una función para facilitar la creación de ficheros temporales:

`mktemp()` devuelve un nombre de fichero temporal único, garantizando que ningún otro fichero se llama así.

También define la variable `tempdir` que tiene una cadena con la ruta del directorio en el que se crearán los ficheros. En un sistema UNIX, si `tempdir` vale `None`, se obtiene el valor del directorio de la variable de entorno `TEMPDIR`.

8. Escritura de módulos

También es posible para el usuario escribir sus propias bibliotecas o módulos. Por ejemplo, podemos crear un fichero `circulos.py` que contenga lo siguiente:

```
import math
pi2=math.pi*2

def perimetro(radio):
   return pi2*radio

def area(radio):
   return pi*radio*radio
```

Ahora podemos utilizar tanto la nueva variable como las dos funciones en nuestros programas. Para eso debemos incluir la sentencia `import circulos`. Después accederemos a la variable y las funciones mediante `circulos.pi2`, `circulos.perimetro` y `circulos.area`.

Al importar un módulo, Python lo "precompila", dejando el resultado en un fichero con el mismo nombre que el módulo y extensión `.pyc`. Si este módulo ya existiera y tuviera una fecha de modificación posterior al `.py`, la precompilación no se habría producido y el módulo se habría cargado directamente.

En muchas ocasiones es útil hacer que un módulo actúe como programa. Por ejemplo, durante el desarrollo de un módulo se puede incluir código para probarlo. Para facilitar esto, se puede utilizar la construcción `if __name__=="__main__":` e incluirse después el código de prueba. Puedes encontrar un ejemplo de esto en la sección 10.6.

9. Excepciones

Cuando hay un error de ejecución, Python emite lo que se llama una *excepción*. Por ejemplo:

```
>>> i=5
>>> while i>-2:
...     print 100/i,
...     i=i-1
...
20 25 33 50 100
Traceback (most recent call last):
  File "<stdio>", line 2, in <module>
ZeroDivisionError: integer division or modulo by zero
```

El mensaje indica que se ha producido una excepción y su tipo. Además indica en qué fichero se ha producido y da una traza de las llamadas. Esto es muy útil para depurar programas.

Si queremos controlar las excepciones, podemos hacerlo mediante la construcción `try-except`. El significado de la construcción es el siguiente. Se ejecutan las sentencias asociadas a la parte `try` hasta que finalicen normalmente o se produzca alguna excepción. En el primer caso, la ejecución continúa en la sentencia siguiente al `try-except`. En caso de que se produzca una excepción, se interrumpe la ejecución. Si la excepción coincide con la que se indica en el `except`, la ejecución continúa en las sentencias asociadas a él. Por ejemplo:

```
>>> l=[2,0,10,12,0.0,8]
>>> for i in l:
...     try:
...         print i,1.0/i
...     except ZeroDivisionError:
...         print i,'no tiene inversa'
...
```

```
2 0.5
0 0 no tiene inversa
10 0.1
12 0.0833333333333
0.0 0.0 no tiene inversa
8 0.125
```

Si no se asocia ninguna excepción al `except`, este las capturará todas, lo que no es aconsejable ya que puede enmascarar errores que no se tratan adecuadamente. Las excepciones también pueden tener un valor asociado en el que se den detalles adicionales. Para capturarlo, basta con poner una variable detrás del tipo de excepción:

```
>>> try:
...     a=1/0
... except ZeroDivisionError,a:
...     print a
...
integer division or modulo by zero
```

```
>>> try:
...     a=1.0/0
... except ZeroDivisionError,a:
...     print a
...
float division
```

Los programas también pueden generar las excepciones explícitamente mediante la orden `raise`. Se le pasan uno o dos argumentos. El primero es el tipo de excepción, el segundo indica los detalles:

```
>>> raise ZeroDivisionError,"esta es voluntaria"
Traceback (most recent call last):
  File "<stdio>", line 1, in <module>
ZeroDivisionError: esta es voluntaria
```

Para crear una excepción, basta con definir una variable que contenga una cadena. Es costumbre que la cadena sea igual al nombre de la variable:

```
>>> miexc='miexc'
>>> try:
...     raise miexc, "funciona"
... except miexc, val:
...     print val
...
Traceback (most recent call last):
  File "<stdio>", line 2, in <module>
TypeError: exceptions must be classes or instances, not str
>>> raise miexc, "es la mía"
Traceback (most recent call last):
  File "<stdio>", line 1, in <module>
TypeError: exceptions must be classes or instances, not str
```

Otra posibilidad, que es la recomendada, es crear una clase (veremos las clases en el siguiente punto) derivada de `Exception`:

```
>>> class MiExcepcion (Exception):
...     # El método __init__ sirve
...     # como constructor:
...     def __init__(self, mensaje):
...        self.mensaje= mensaje
...
```

```
>>> try:
...   raise MiExcepcion("Esto lo he lanzado yo")
... except MiExcepcion, excepcion:
...   print excepcion.mensaje
...
Esto lo he lanzado yo
```

10. Clases y objetos

Como se comentó en la introducción, Python es un lenguaje orientado a objetos. Además, la complicación sintáctica para esto es mínima. Haciendo una simplificación tal vez excesiva, podemos entender la clase como una especie de "registro" cuyas instancias además de los campos habituales (que contienen datos como enteros o listas) pueden tener campos que son funciones. Por ejemplo, podríamos tener un objeto del tipo vehículo. Este objeto tendría un campo (`pos`) que sería su posición. Además tendría dos métodos: `mueve` que cambia la posición añadiendo los kilómetros que se pasan como parámetros y `posicion` que devuelve la posicion actual. Un ejemplo de uso sería:

```
>>> v= vehiculo()
>>> print v.posicion()
0
>>> v.mueve(10)
```

```
>>> v.mueve(14)
>>> print v.posicion()
24
```

Lo primero que hacemos es crear una instancia de la clase vehículo. Después aplicamos los métodos sobre el vehículo para cambiar su posición. Podemos imaginarnos que al aplicar un método sobre un objeto, el método en cuestión "sabe" sobre qué objeto se está aplicando. Esto se hace pasando al método un parámetro adicional: el propio objeto.

Para definir la clase anterior haríamos lo siguiente:

```
class vehiculo:
   def __init__(self):
      self.pos=0

   def mueve(self,k):
```

```
        self.pos=self.pos+k

    def posicion(self):
        return self.pos
```

Podemos destacar lo siguiente:

- Para definir una clase, basta con poner su nombre tras la palabra reservada `class` y lo que sigue forma parte de la definición de la clase (hasta la primera línea que tenga el mismo nivel de indentación que la línea donde está `class`).
- Generalmente, la definición de la clase consiste en una serie de funciones que serán los métodos de la clase.
- Todos los métodos tienen `self` como primer parámetro. Este es el parámetro con el que se pasa el objeto. El nombre `self` es convencional, puede ser cualquier otro.
- Existe un método especial `__init__`. A este método se le llama cuando se crea un objeto de la clase.
- Los atributos (campos de datos) de la clase se crean como las variables de Python, mediante una asignación sin declaración previa.

El método `__init__` puede tener parámetros adicionales. En este caso, se pasan con el nombre de la clase. Por ejemplo, si quisiéramos dar una posición inicial para el vehículo haríamos:

```
def __init__(self,pos):
     self.pos=pos
```

Y para crear un objeto en la posición 10, la llamada sería `v=vehiculo(10)`. Si combinamos esto con los parámetros por defecto, podemos hacer que si no se especifica posición, el vehículo comience en 0 y, en caso contrario, donde se especifique:

```
def __init__(self,pos=0):
     self.pos=pos
```

10.1. Métodos especiales

Hemos visto que `__init__` es un método que se invoca implícitamente al crear un objeto. Existen otros métodos que también se invocan de forma implícita en determinadas circunstancias. Todos ellos tienen nombres rodeados por `__`. Vamos a estudiar algunos de ellos.

Creación y destrucción de los objetos

`__init__(self[,`*args*`])` es el método que se llama al crear instancias de la clase. Los argumentos opcionales se pasan como parámetros al nombre de la clase.

`__del__(self)` se llama al destruir un objeto.

Representación de los objetos como cadenas

`__str__(self)` se llama al utilizar sobre el objeto la función predefinida `str`, al aplicarle la orden `print` o con la marca de formato `%s`. Debe devolver como resultado una cadena.

`__repr__(self)` se llama, por ejemplo, al utilizar sobre el objeto la función predefinida `repr` y, por tanto, se espera que devuelva una cadena que, de ser evaluada con `eval`, diera lugar a un objeto igual al de partida.

Emulación de secuencias y diccionarios

Con los siguientes métodos se pueden "simular" los objetos nativos del Python.

`__len__(self)` debe devolver la longitud del objeto. Se llama al utilizar la función `len` sobre el objeto.

`__getitem__(self,`k`)` se invoca sobre el objeto `x` al hacer `x[`k`]`. Permite utilizar el objeto como diccionario.

`__setitem__(self,`k`,`v`)` se invoca sobre el objeto `x` al hacer `x[`k`]=`v. Permite utilizar el objeto como diccionario.

`__delitem__(self,`k`)` se invoca sobre el objeto `x` al hacer `del x[`k`]`.

`__getslice__(self,`i`,`j`)` se invoca al hacer `x[`i`:`j`]`.

`__setslice__(self,`i`,`j`,`s`)` se invoca al hacer `x[`i`:`j`]=`s.

`__delslice__(self,`i`,`j`)` se invoca al hacer `del x[`i`:`j`]`.

Emulación de tipos numéricos

Con los siguientes métodos se pueden emplear los operadores que normalmente se utilizan con números.

`__add__(self,`r`)` se invoca con la expresión `x+`r. Existen funciones análogas para otros operadores: `__sub__`, `__mul__`, `__div__`, `__mod__`, `__divmod__`, `__pow__`, `__lshift__`, `__rshift__`, `__and__`, `__or__`, `__xor__`.

`__radd__(self,`l`)` se invoca con la expresión l`+x`. De manera análoga, se pueden definir las operaciones de la lista anterior pero empezando por `__r`.

`__neg__(self)` se invoca con `-x`. Para los operadores unarios `+`, `abs` y `~` tenemos `__pos__`, `__abs__` e `__invert__`, respectivamente.

10.2. Atributos especiales

Además de los métodos especiales, los objetos tienen algunos atributos especiales que nos permiten cierta "introspección". Algunos de estos atributos son:

`__dict__` es un diccionario que contiene los atributos "normales" de la clase.

`__class__` contiene la clase a la que pertenece el objeto. De la clase te puede interesar el atributo `__name__` que contiene el nombre de la clase.

Así:

```
>>> class Clase:
...   def __init__(self, v):
...     self.v= v
...
>>> a= Clase(3)
>>> a.__dict__
{'v': 3}
>>> a.__class__.__name__
'Clase'
```

10.3. Clases derivadas

Crear clases derivadas en Python es especialmente sencillo. Basta con poner, en la declaración de la clase derivada, el nombre de la clase base entre paréntesis. La clase derivada puede redefinir cualquier método de la clase base. En caso de que alguno de los métodos no se redefina, la invocación del método en una instancia de la clase derivada provoca una invocación del correspondiente método de la clase base:

```
>>> class Base:
...   def __init__(self, v):
...     self.v= v
...     print "Inicializo la base con un %d" % v
...   def f1(self, n):
...     return "f1 de la base recibe un %d" % n
...   def f2(self, n):
...     return "f2 de la base recibe un %d" % n
...
>>> class Derivada(Base):
...   def __init__(self, v):
...     Base.__init__(self,v)
...     print "Se ha inicializado la derivada con %d" % self.v
...   def f2(self, n):
...     return "f2 de la derivada recibe un %d" % n
...
>>> a= Derivada(4)
Inicializo la base con un 4
Se ha inicializado la derivada con 4
>>> a.f1(2)
'f1 de la base recibe un 2'
>>> a.f2(2)
'f2 de la derivada recibe un 2'
```

Fíjate en lo que hemos hecho para llamar al constructor de la clase base, si se define un constructor para la clase derivada, no se llama al de la clase base. En este caso particular, habría bastado con no definir ningún constructor para la derivada y todo habría funcionado.

10.4. Ejemplo: una clase para árboles de búsqueda

Vamos a crear una clase que nos permita almacenar árboles de búsqueda. Las clases nos permiten reflejar de manera natural la estructura recursiva del árbol. Un objeto de tipo árbol tendrá tres atributos:

- `elto`: el elemento en la raíz del árbol.
- `izdo`, `dcho`: los subárboles izquierdo y derecho.

Al constructor le pasaremos el elemento que se almacenará en la raíz:

```
class Arbol:
  def __init__(self, elto):
    self.elto= elto
    self.izdo= None
    self.dcho= None
```

Para insertar un elemento, primero comprobamos si coincide con el de la raíz. Si no está, lo insertamos en el subárbol izquierdo o derecho, según corresponda. Tenemos que tener cuidado para evitar problemas cuando el subárbol donde vamos a hacer la inserción no existe.

```
def inserta(self, nuevo):
  if nuevo== self.elto:
    return self
  elif nuevo< self.elto:
    if self.izdo== None:
      self.izdo= Arbol(nuevo)
    else:
      self.izdo= self.izdo.inserta(nuevo)
  else:
    if self.dcho== None:
      self.dcho= Arbol(nuevo)
    else:
      self.dcho= self.dcho.inserta(nuevo)
  return self
```

Hemos hecho que devuelva el árbol para permitir inserciones "en cadena" como

```
Arbol(3).inserta(2).inserta(4).
```

Finalmente, comprobar si un elemento está en el árbol es trivial:

```
def esta(self, buscado):
  if buscado== self.elto:
    return True
  elif buscado< self.elto:
    if self.izdo== None:
      return False
    else:
      return self.izdo.esta(buscado)
  else:
    if self.dcho== None:
      return False
    else:
      return self.dcho.esta(buscado)
```

Ejercicio 9

Define el método `__str__` de modo que devuelva una representación apropiada del árbol.

Ejercicio 10

También podemos implementar los árboles guardándonos un atributo que indique si está o no vacío. El constructor sería algo así como:

```
class Arbol:
  def __init__(self):
    self.vacio= True
```

La inserción crearía el atributo `elto` y los atributos `izdo` y `dcho`.

Implementa esta versión de la clase `Arbol`.

10.5. Ejemplo: una clase para representar expresiones sencillas

Vamos a definir una clase `AST` para representar expresiones numéricas con sumas y multiplicaciones. Necesitarás una estructura similar a esta para construir el árbol de sintaxis abstracta, por lo que es aconsejable que entiendas perfectamente este programa.

De la clase `AST` derivamos las dos clases `Operacion` y `Numero` que formarán respectivamente los nodos y las hojas del árbol.

```
#!/usr/bin/env python
class AST:             #------------------------------------------------------------
  def __init__(self): pass
  def mostrar(self): pass            # muestra la expresión utilizando paréntesis
  def numero_operaciones(self): pass # muestra el número de operaciones de la expresión
  def interpreta(self): pass         # evalua la expresión

class Numero(AST):     #------------------------------------------------------------
  # p.e. para crear una hoja con el dígito 7 haríamos Numero(7)
  def __init__(self, valor):
    self.valor = valor
  def mostrar(self):
    return str(self.valor)
  def numero_operaciones(self):
    return 0
  def interpreta(self):
    return self.valor

class Operacion(AST):  #------------------------------------------------------------
  # p.e. para crear un nodo con la expresión 5*3 haríamos
  #   Operacion('*',Numero(5),Numero(3))
  def __init__(self, op, izda, dcha):
    self.op   = op
    self.izda = izda
    self.dcha = dcha
  def mostrar(self):
    return '(' + self.izda.mostrar() + self.op + self.dcha.mostrar() + ')'
  def numero_operaciones(self):
    return 1 + self.izda.numero_operaciones() + self.dcha.numero_operaciones()
  def interpreta(self):
    if self.op=="+":
      return self.izda.interpreta() + self.dcha.interpreta()
    else: # si no, debe ser '*'
      return self.izda.interpreta() * self.dcha.interpreta()

# Programa principal ---------------------------------------------------------

# Introducimos el árbol de la expresión 4*5 + 3*2
num1=Numero(4)
num2=Numero(5)
num3=Numero(3)
num4=Numero(2)
arbol1=Operacion('*',num1,num2) # 4*5
arbol2=Operacion('*',num3,num4) # 3*2
arbol_final=Operacion('+',arbol1,arbol2) # arbol1+arbol2

# Accedemos al árbol de tres formas diferentes mediante funciones miembro
```

```
print 'El árbol contiene la expresión:', arbol_final.mostrar()
print 'El árbol contiene en total %d operaciones' % arbol_final.numero_operaciones()
print 'La expresión se evalua como:', arbol_final.interpreta()
```

Al ejecutar el programa anterior, se construye un árbol con la expresión $4*5+3*2$ y se obtiene la siguiente salida:

```
El árbol contiene la expresión: ((4*5)+(3*2))
El árbol contiene en total 3 operaciones
La expresión se evalua como: 26
```

Ejercicio 11

Amplía la clase `Operacion` para que incluya restas y divisiones.

Difícil (pero no mucho): modifica el método mostrar para que no escriba paréntesis si no es necesario según las reglas de prioridad habituales.

10.6. Ejemplo: una clase para representar conjuntos de enteros no negativos

Vamos a definir una clase para representar conjuntos de enteros no negativos. Para ello trabajaremos internamente con un entero largo en el que cada bit indicará la presencia o ausencia del elemento.

```
#!/usr/bin/env python
from types import *
from math import *

class IntSet:
    def __init__(self): # inicialización
        self.c = 0L

    def __str__(self):  # impresión del resultado
        c = self.c
        s = "{ "
        i = 0
        while c != 0:
            j = 2**long(i)
            if c & j:
                s = s + str(i) + ' '
                c = c - j
            i = i + 1
        s = s + "}"
        return s

    # inserción de un entero o
    # de una lista de enteros
    def insert(self, n):
        if type(n) != ListType:
            self.c = self.c | 2**long(n)
        else:
            for i in n:
                self.c = self.c | 2**long(i)

    def __len__(self):
        c = self.c
        i = 0
        n = 0
        while c != 0:
            j = 2**long(i)
            if c & j:
                c = c - j
                n = n + 1
            i = i + 1
        return n

    def __getitem__(self, i):
        return (self.c & 2**long(i)) != 0

    def __setitem__(self, i, v):
        s = 2**long(i)
        self.c = self.c | s
        if v == 0:
            self.c = self.c ^ s

    def __or__(self, right):   # Unión
        R = IntSet()
        R.c = self.c | right.c
        return R

    def __and__(self, right):   # Intersección
        R = IntSet()
        R.c = self.c & right.c
        return R

    def __sub__(self, right):   # Diferencia
        R = IntSet()
        R.c = self.c ^ (self.c & right.c)
        return R

if __name__ == "__main__":
    A = IntSet()
    A.insert(10)
    A.insert([2, 4, 32])
    print "Conjunto A:", A
    print "Long:     ", len(A)
    print "¿Está el 2?",A[2], "¿Y el 3?", A[3]
    A[3] = 1; A[2] = 0
    print "¿Está el 2?",A[2], "¿Y el 3?", A[3]
    print "Conjunto A:", A

    B = IntSet()
    B.insert([3,7,9,32])

    print "B es", B
    print "A unión B es       ", A|B
```

```
print "A intersección B es", A&B                    print ((A|B) - (A&B)) - A
print "A \\ B es          ", A-B
```

11. Un ejercicio adicional

Este ejercicio consistirá, básicamente, en escribir un programa que permita obtener información sobre los usuarios de un sistema Unix a partir del contenido del fichero `/etc/paswd`. En realidad, lo que se pretende es escribir versiones cada vez más refinadas del programa siguiente:

```
#!/usr/bin/env python
import sys,string

nomfp="/etc/passwd"

fp=open(nomfp,"r")

usuarios={}
while True:
   linea=fp.readline()
   if not linea:
     break
   campos=string.split(linea,":")
   usuarios[campos[0]]=campos[4]
fp.close()

while True:
   sys.stdout.write("¿Usuario?: ")
   linea=sys.stdin.readline()
   if not linea:
     break
   login=string.strip(linea)
   if usuarios.has_key(login):
     print "  "+usuarios[login]
   else:
     print "  Usuario desconocido"
print
print "¡Hasta otra!"
```

Una vez hayas entendido perfectamente cómo funciona el programa anterior, ve introduciéndole las siguientes modificaciones:

1. Haz que tu programa muestre más información sobre el usuario, no sólo el quinto campo de la correspondiente línea del fichero de contraseñas (esto es, nombre del usuario y comentarios). Ten en cuenta, por ejemplo, que el tercer campo corresponde al *uid* y el cuarto, al *gid*. Define una clase `DatosUsuario` cuyos diferentes atributos correspondan a diferentes datos del usuario, de forma que el diccionario `usuarios` relacione el *login* de cada usuario con un objeto de la clase. Dota a la clase de un método de inicialización que permita asignar valores a los diferentes atributos del objeto en el momento de su creación. Considera la posibilidad de dotar también a la clase de un método `__str__()`.

2. Haz que la información que se muestre de cada usuario incluya también el número de usuarios en su grupo y una relación de los mismos. Puedes hacer que no se imprima el *login* de todos y cada uno de los miembros del grupo, sino sólo el de los diez primeros en orden alfabético. Parece razonable almacenar la información de los diferentes grupos en un diccionario `grupos` que se vaya construyendo al mismo tiempo que `usuarios`.

3. Haz que, al pedirle la información de un usuario, el programa muestre el nombre del grupo, en vez de su *gid*, si este dato está presente en `/etc/group`. En este fichero, en el que los diferentes campos también están separados por el carácter dos puntos, el primero corresponde al nombre del grupo y el tercero, a su *gid*. Haz que se procese el fichero en una sola pasada para actualizar con él la información recogida en el diccionario `grupos`.

4. Haz que exista la posibilidad de especificar en la línea de órdenes, al llamar al programa, unos ficheros de contraseñas y de grupos diferentes de los habituales. Suponiendo que el programa se llame `infouser`, la orden deberá tener el siguiente perfil:

```
usage: infouser [fich_usuarios [fich_grupos]]
```

5. Introduce diferentes controles de error en el programa. Define una función `error()` que reciba como argumento una cadena que describe el error, la muestre por la salida de error estándar y finalice la ejecución del programa devolviendo al intérprete de órdenes (*shell*) un *status* 1. Los diferentes errores que puedes controlar son: un número inadecuado de argumentos en la línea de órdenes, problemas en la apertura de los ficheros y que sus líneas no sigan el formato esperado. Los dos últimos tipos de errores puedes intentar capturarlos con construcciones `try-except`. Trata los errores de apertura de ficheros de forma que, si estos afectan a ambos ficheros, se informe al usuario de ello antes de abortar la ejecución.

6. Haz que el *uid* también sea una forma válida de solicitar al programa la información de un usuario. Utiliza `re.match()` para discriminar si lo leído puede ser un *uid* (esto es, si es una secuencia de dígitos) o no, en cuyo caso será un posible *login*. Considera la utilidad de haber construido un tercer diccionario para relacionar cada *uid* con su correspondiente *login*.

7. Considera la posibilidad de mejorar el formato de la salida de tu programa utilizando las herramientas descritas al comienzo del capítulo 7 del Python Tutorial.

A. Soluciones a algunos ejercicios

A.1. Sobre tipos de datos

Ejercicio 1:

Veamos dos posibilidades distintas, según qué delimitador se utilice para la cadena:

```
>>> print "'\\/\""
'\/"
>>> print '\'\\/"'
'\/"
```

Ejercicio 2:

Una posible solución sería esta:

```
>>> a='esto es un ejemplo'
>>> print a[8:]+a[4:8]+a[:4]
un ejemplo es esto
```

Ejercicio 3:

La modificación de `a` podría llevarse a cabo como sigue:

```
>>> a=[1,[2,[3,4]],5]
>>> a[1]=a[1]+[[6,7]]
>>> a
[1, [2, [3, 4], [6, 7]], 5]
```

Ejercicio 4:

La asignación de la lista `[3]` a la variable `b` no modifica el objeto al que anteriormente hacía referencia esa variable (la lista `[1,2]`), que sigue tal cual, ligado ahora tan sólo a las referencias `c[0]` y `c[1]`. Esto es a diferencia de lo que sucedía en el primer caso, ya que la asignación `b[0]=2` sí modificaba un objeto: el ligado a las referencias `b`, `c[0]` y `c[1]`, que pasaba de ser la lista `[1,2]` a ser la lista `[2,2]`.

Volviendo al segundo caso, obsérvese que las referencias `c[0]` y `c[1]` lo son *al mismo objeto*, de modo que la secuencia de órdenes al intérprete de Python podría haber seguido así:

```
>>> b=[1,2]
>>> c=[b,b]
>>> b=[3]
>>> c
[[1, 2], [1, 2]]
>>> c[0].append(7)
>>> c
[[1, 2, 7], [1, 2, 7]]
```

A.2. Sobre control de flujo

Ejercicio 5:

Utilizando un bucle `while`, el programa podría ser el siguiente:

```
#!/usr/bin/env python
lista=[]
i=1
while i<=10:
   lista.append(i)
   i=i+1
print lista
```

Con un bucle `for`, la solución podría ser esta:

```
#!/usr/bin/env python
lista=[]
for i in range(10):
   lista.append(i+1)
print lista
```

Obsérvese que el siguiente programa, que no utiliza bucle alguno, es equivalente:

```
#!/usr/bin/env python
lista=range(1,11)
print lista
```

Ejercicio 6:

Lo que se pide es, en realidad, un fragmento de programa. Supongamos que el nombre de la tupla dada es precisamente `tupla`. La construcción de la lista (que llamaremos `lista`) podría realizarse como sigue:

```
lista=[]
for elemento in tupla:
   lista.append(elemento)
```

Lo anterior es correcto porque `for` permite iterar sobre los elementos de cualquier secuencia (no sólo sobre listas, sino también sobre tuplas e incluso cadenas). De no haberlo tenido en cuenta, la siguiente hubiera sido una alternativa razonable:

```
lista=[]
for i in range(len(tupla)):
   lista.append(tupla[i])
```

Por último, podemos utilizar `list` para hacer lo mismo:

```
lista= list(tupla)
```

Ejercicio 7:
El ejercicio pide un programa como el siguiente:

```
#!/usr/bin/env python
for i in range(1,1001):
   if i%7==0:
     print i
```

Obviamente, el `if` podría evitarse recorriendo la lista de múltiplos de 7 que genera `range(7,1001,7)`, o bien recorriendo `range(1,1000/7+1)` e imprimiendo `7*i`.

A.3. Sobre funciones

Ejercicio 8:
El ejercicio pide un programa como el siguiente:

```
#!/usr/bin/env python
def multiplo7(n):
   if n%7==0:
     return True
   else:
     return False
for i in range(1,1001):
   if multiplo7(i)==1:
     print i
```

Además, sabiendo que en Python las operaciones de comparación devuelven un resultado `False` para representar el valor lógico falso y un resultado `True` para el valor verdadero, la función podría reescribirse simplemente como sigue:

```
def multiplo7(n):
   return n%7==0
```

También sucede que las condiciones de las construcciones `if` y `while` no es necesario que sean comparaciones, sino que pueden ser objetos cualesquiera: el lenguaje les asocia un valor lógico concreto siguiendo las reglas que vimos en la sección 3.1, lo que nos permitiría sustituir la comparación `multiplo7(i)==True` por, simplemente, `multiplo7(i)`.

B. Preguntas frecuentes

1: Escribo un programa Python en un fichero, escribo el nombre del fichero en el intérprete de órdenes (shell) y el programa no se ejecuta. ¿Por qué?

Hay tres errores que se cometen con cierta frecuencia y que conviene evitar. Comprueba lo siguiente:

- El fichero tiene una primera línea que indica correctamente a la *shell* que es Python quien ha de ejecutar el programa. Una línea como

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

debería servir en cualquier instalación razonable de Unix.

- El fichero tiene los permisos de ejecución adecuados (si no es así, utiliza `chmod`).
- El directorio donde se encuentra el fichero está en tu `PATH`. Puedes utilizar `echo $PATH` para comprobarlo. Si, por ejemplo, necesitas modificar el `PATH` para que incluya al directorio actual, haz lo siguiente:

  ```
  export PATH=.:$PATH
  ```

  Si te preocupa el problema de seguridad que representa poner `.` en el `PATH`, puedes ejecutar el programa añadiendo `./` delante del nombre.

2: ¿Por qué Python da un error cuando intento añadir un elemento a una lista y ordenar la lista resultante con `lista.append(elemento).sort()`?

Simplemente, porque `append()` no devuelve como resultado la lista con el elemento añadido, sino que sólo modifica la propia lista añadiéndole un nuevo elemento. Lo mismo sucede con `sort()`, que no devuelve como resultado la lista ordenada, sino que ordena la propia lista. Es importante que distingas lo que una operación devuelve como resultado y los efectos secundarios (*side effects*) de la operación. Así, tras `a=[3,5]`,

- `b=a.append(4)` haría que el valor de `a` fuera `[3,5,4]` y el de `b` fuera `None`, mientras que
- `b=a+[4]` no modificaría el valor de `a` (seguiría siendo `[3,5]`), pero el de `b` sería `[3,5,4]`.

Esto último es así porque `lista+[elemento]` sí devuelve como resultado una nueva lista, pero no tiene efectos secundarios sobre la lista original.

Volviendo a la pregunta original, el error se produce porque `lista.append(elemento)`, al no devolver resultado alguno, lo que hace es devolver un objeto `None`, sobre el cual es incorrecto intentar aplicar un método `sort()`.

3: En Python, ¿existe alguna diferencia entre las cadenas delimitadas por comillas simples y las delimitadas por comillas dobles?

En principio, tanto las comillas simples como las dobles son dos métodos alternativos para delimitar *literales de cadena* (esto es, aquellos lexemas que nos permiten representar en un lenguaje de programación, Python en el caso que nos ocupa, el valor de una cadena). Así, por ejemplo, `'hola'` y `"hola"` son notaciones alternativas y equivalentes para la cadena formada por los cuatro caracteres siguientes: hache, o, ele y a.

Cualquier cadena puede representarse en Python utilizando como delimitadores comillas simples, y lo mismo puede decirse de las dobles. No obstante, existe una pequeña diferencia entre unas y otras. Sabemos que a veces, para representar la presencia de un carácter en una cadena, es necesario utilizar en el literal de cadena más de un carácter (lo que se conoce como una *secuencia de escape*). Así, el salto de línea se representa mediante una barra invertida y el carácter ene (`\n`), la comilla simple se representa como `\'` y para la doble se utiliza `\"`. Ahora bien, cuando se utilizan comillas simples como delimitadores, las dobles pueden representarse a sí mismas en los literales de cadena, y viceversa, de modo que los cuatro literales siguientes representan todos ellos a la cadena formada por una comilla simple y otra doble:

- `'\'\"'`
- `'\'"'`
- `"\'\""`
- `"'\""`

4: Escribo un programa Python en el que se comparan dos cadenas, una leída mediante `readline()` y la otra representada mediante un literal, y el programa siempre las considera distintas (aunque aparentemente sean iguales). ¿Por qué?

Las cadenas que devuelve `readline()` son líneas enteras, *incluyendo el carácter de salto de línea* (salvo que este no esté presente en el fichero leído por haberse alcanzado antes el fin del fichero). Así, si por ejemplo el programa está leyendo una respuesta de la entrada estándar con `resp=sys.stdin.readline()`, el usuario teclea ene, o y *enter* y el programa evalúa la comparación `resp=='no'`, el resultado será de falsedad (`0`), porque la cadena `resp` tendrá, además de la ene y la o de `'no'`, un tercer carácter: el de salto de línea. En una primera aproximación, se podría solucionar este problema siguiendo una cualquiera de las dos estrategias siguientes:

- Escribir la comparación como `resp=='no\n'`, incluyendo el carácter de salto de línea también en la cadena representada mediante literal.
- Eliminar el salto de línea de la cadena leída, por ejemplo con `resp=resp[:-1]`.

Las soluciones anteriores, aparte de no ser muy elegantes, pueden dar lugar a resultados indeseados si el usuario, tras la ene y la o, introduce inmediatamente el fin del fichero. Una solución más general, que no adolece de estos problemas, sería utilizar `rstrip()` como se ejemplifica a continuación:

```
resp=resp.rstrip()
```

De este modo, se eliminarían de la cadena leída tanto un eventual carácter final de salto de línea como todos los posibles blancos iniciales y finales que el usuario hubiera podido introducir inadvertidamente.

5: Muy bien, pero ¿cómo puede una línea devuelta por `readline` no tener fin de línea?

Si el fichero se termina antes de que aparezca un fin de línea, la última línea leída no tendrá fin de línea. Si la pregunta es ¿cómo puede un fichero no tener un fin de línea en cada línea?, hay varias posibilidades. Puedes crear fácilmente con `emacs` un fichero así (`vi` no sirve para esto; siempre añade finales de línea). Otra posibilidad es hacer algo parecido a lo siguiente:

```
>>> f= open("aaa","w")
>>> f.write("hola\nhola")
>>> f.close()
>>> f= open("aaa","r")
>>> l= f.readline()
>>> l
'hola\n'
>>> l= f.readline()
>>> l
'hola'
>>>
```

6: En Python, ¿cómo puedo saber si una cadena pertenece al lenguaje definido por una expresión regular?

Para trabajar con expresiones regulares, Python ofrece el módulo `re`, con dos funciones, `match()` y `search()`, útiles para nuestros propósitos. No obstante, hay que tener en cuenta lo siguiente:

- `re.match(er,c)` comprueba si *algún prefijo* de la cadena `c` casa con la expresión regular `er` (análogamente sucede con `erc.match(c)` si `erc` es una expresión regular compilada con `erc=re.compile(er)`).
- `re.search(er,c)` (o bien `erc.search(c)` si `erc` es una versión compilada de `er`) comprueba si *alguna subcadena* de `c` casa con `er`.

Para saber si precisamente la cadena `c` (no alguno de sus prefijos o de sus subcadenas) casa con `er`, lo que puede hacerse es incorporar en la expresión regular los puntos de anclaje inicial (`^`) y final (`$`), que sólo casan con el inicio y el fin de la cadena, respectivamente (y luego utilizar indistintamente `match()` o `search()`). Por ejemplo, lo siguiente debería funcionar:

```
if re.match('^'+er+'$',c): print 'OK'
```

7: En Python, ¿qué diferencia hay entre las asignaciones `a[2] = [6,7]` y `a[2:3] = [6,7]`?

En el primer caso se sustituye el elemento en la posición dos por la lista asignada. La lista continúa teniendo cuatro elementos pero ahora uno de ellos es una lista:

```
>>> a=[0,1,2,3]
>>> a[2] = [6,7]
>>> a
[0, 1, [6, 7], 3]
```

En el segundo caso se sustituye el elemento en la posición dos por los elementos de la lista asignada. Para este ejemplo concreto, la lista pasa a tener un elemento más:

```
>>> a=[0,1,2,3]
>>> a[2:3] = [6,7]
>>> a
[0, 1, 6, 7, 3]
```