Notas sobre el lenguaje Python

26 de agosto de 2003

Walter Moreira

Copyright © 2002 Walter Moreira.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled “GNU Free Documentation License”.

El código fuente en LaTeX para estas notas está disponible en `http://www.cmat.edu.uy/~walterm/python`.

Índice

0. Introducción **4**

1. El intérprete de Python **5**
Ejecutando el intérprete . . . 5
Primeras pruebas . . . 5
Programas . . . 6

2. Objetos **8**
Tipos de objetos básicos . . . 8
Números . . . 8
Poniéndole nombres a los objetos . . . 9
Cadenas de caracteres . . . 10
Listas . . . 12
Diccionarios . . . 13
El objeto 'None' . . . 14

3. Decisiones **15**
Condiciones . . . 15
Instrucción 'if' . . . 15
Un ejemplo . . . 17

4. Iteraciones **18**
Bucle 'for' . . . 18
Bucle 'while' . . . 19
Abortando los bucles . . . 19
Salteando iteraciones . . . 20

5. Funciones **21**
Funciones . . . 21
Usando funciones . . . 21
Definiendo funciones . . . 21
Módulos . . . 22
Variables locales . . . 23

6. Objetos II **25**
Más sobre las variables . . . 25
Objetos mutables e inmutables . . . 26

7. Funciones II y listas por comprehensión **29**
Más sobre las funciones . . . 29
Parámetros con nombres . . . 30
Listas por comprehensión . . . 30

8. Clases **32**

A. Referencias y material adicional **36**
Referencias sobre el lenguaje Python . . . 36
Material adicional . . . 36

B. Licencia de Documentación Libre GNU **37**

0. Introducción

Estas notas son una muy sintética y resumida documentación sobre el *núcleo* del lenguage Python; y tratan de describir los elementos básicos del lenguaje en pocas páginas. Es probable que necesites algún otro material de referencia, como los que se indican en el apéndice A. Y sin duda, después que hayas leído estas notas, es conveniente que recurras a textos más avanzados para obtener una visión global del Python y de la programación.

Sin embargo, con el material expuesto aquí, ya dispones de suficiente conocimiento como para consultar el *Python Library Reference*, en donde se describen todas las bibliotecas que incluye el lenguaje Python. Con ellas puedes realizar la mayor parte de las tareas que aparecen usualmente, en el ámbito de la programación; desde manejar archivos hasta conectarte a Internet, o desde funciones para el manejo de texto hasta funciones para el manejo de base de datos. Es aconsejable que pasees por el índice del *Python Library Reference* para tener una idea de la potencia del Python.

Estas notas utilizan la versión 2.3 del Python; procura instalar esta versión o una posterior, dado que algunas construcciones descriptas aquí no funcionaran con versiones anteriores. No damos instrucciones respecto a la instalación pues varían mucho de un sistema operativo a otro; consulta el sitio `www.python.org` para obtener información detallada sobre ese tema.

Por último, este material todavía está en proceso de construcción. Tus comentarios, observaciones o correcciones serán muy útiles para perfeccionarlo. Envíalos a `walterm@cmat.edu.uy`.

1. El intérprete de Python

Ejecutando el intérprete

El primer paso para correr Python en tu computadora es instalarlo. Busca la documentación para ese proceso en el sitio www.python.org o en el CD que se distribuye. Una vez instalado, sigue las instrucciones según el sistema operativo que dispongas:

En una computadora con Windows

1. Presiona el botón "Start" o "Inicio".
2. Busca el ítem que dice "Programs" o "Programas" y presiónalo.
3. Uno de los ítems del menú debe ser "Python", selecciona ese ítem.
4. Selecciona la línea que dice "IDLE (Python GUI)"; o en su defecto selecciona el item "Python".

Alternativamente, es posible que haya un ícono en el escritorio de Windows con una serpiente. Si es así, simplemente haz doble click sobre ese ícono.

En una computadora con Linux

1. Abre una ventana de comandos.
2. Ejecuta el comando 'idle'; o en caso de que dicho comando no exista, ejecuta el comando 'python'.

Si todo anduvo bien, en cualquiera de los dos casos se obtendrá una ventana con unas líneas de la forma:

```
Python 2.3+ (#2, Aug 10 2003, 11:33:47)
[GCC 3.3.1 (Debian)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> _
```

Si no obtienes algo como lo anterior, consulta con los profesores. Los comandos 'python' e 'idle' ejecutan ambos el mismo intérprete. La única diferencia consiste en que 'idle' incorpora un editor para escribir los programas, junto con algunas otras facilidades. Si en tu instalación no encuentras el comando 'idle', simplemente ejecuta 'python' y utiliza cualquier editor de texto que dispongas.

El texto en `caracteres de máquina` *representa información impresa por la computadora. El texto* `subrayado` *representa la información que debes tipear.*

Primeras pruebas

Una vez que el intérprete de Python está corriendo (el símbolo '>>>' se denomina "prompt", y significa que el Python está listo para recibir instrucciones) podemos empezar a tipear nuestros programas. Intenta lo siguiente (al final de cada línea debes presionar la tecla `Enter`):

```
>>> import math
>>> print math.pi
3.14159265359
```

Puedes divertirte usando Python como una calculadora común y corriente:

```
>>> 3+2
5
>>> 4*0.5                              El símbolo '*' es la multiplicación.
2.0
>>> 2**100                             El símbolo '**' representa la exponenciación,
1267650600228229401496703205376L       es decir, calculamos 2^100.
```

La 'L' que aparece al final del resultado anterior indica que el número es un entero *largo* (ver la sección 1). También pueden aparecer algunas sorpresas (que serán explicadas más adelante):

```
>>> 5/2                                Calculamos 5 dividido por 2...
2                                      ?!... (ver la sección 1)
>>> 0.1+0.2
0.30000000000000004                    ¡Otra sorpresa!... (ver también la sección 1)
```

Es importante que experimentes en la máquina todos los ejemplos que aparecen en estas hojas; más aún, que los modifiques o que pruebes los que se te ocurran. No te limites simplemente a leerlos.

Programas

Tipear comandos de a uno a la vez es conveniente cuando estamos probando cosas, ya que se obtienen los resultados inmediatamente. Pero para hacer cosas más complejas será mejor guardar la secuencia de instrucciones en un archivo, al cual se le prodrán hacer cambios y ejecutarlo varias veces. Este archivo es lo que llamamos *programa*.

Creando programas

Para crear un programa, debes crear un archivo con extensión '.py'. Puedes hacerlo con cualquier editor; o si estás usando el 'idle', selecciona el menú "File" y luego cliquea en "New window". Ahí puedes escribir el programa. Como ejemplo intenta:

```
import math
r = float(raw_input('Radio de la circunferencia: '))
print 'El area de la circunferencia de radio', r, 'es', math.pi*(r**2)
```

Explicaremos el significado de estas líneas más adelante.

Luego, salva el archivo, por ejemplo con nombre 'area.py' (en 'idle' selecciona "File" y luego "Save"). Si estás trabajando en Windows asegúrate de crear ese archivo en la carpeta 'C:\Python23'.

Ejecutando los programas

Para ejecutar el programa, escribe la siguiente linea en el prompt de Python:

```
>>> import area                        Observa que no se necesita la extensión '.py'
```

El intérprete de Python leerá el archivo 'area.py' y lo ejecutará. Si, por el contrario, obtienes un error de la forma:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ImportError: No module named area
```

asegúrate que salvaste el archivo 'area.py' en el mismo directorio en donde ejecutaste el comando 'python', en el caso que uses Linux; o que guardaste el archivo en la carpeta 'C:\Python23' si usas Windows.

Cargando programas anteriores

Para recuperar programas que hayan sido salvados en sesiones anteriores, simplemente abre el archivo con el editor que estés usando (si usas '`idle`', selecciona "File" en la ventana principal, donde está el prompt de Python, y luego selecciona "Open"). Luego puedes modificar el archivo, volver a salvarlo y ejecutarlo como se mostró más arriba.

2. Objetos

Los objetos son el componente básico del lenguaje Python. Cualquier cosa a ser manipulada es un objeto; por ejemplo: números, letras, nombres, dibujos, gráficos, matrices, funciones, archivos,..., cualquier cosa. Sin embargo, hay una cierta cantidad de esos objetos que pueden ser vistos como los bloques fundamentales, a partir de los cuales se pueden construir los demás. Aquí veremos algunos de ellos, en las secciones 6 y 7 se describen varios otros y en la sección 8 se explica como construir tipos de objetos arbitrarios usando otros ya construidos.

Tipos de objetos básicos

Los tipos básicos más sencillos son:

- Números.
- Cadenas de caracteres (texto).
- Listas.
- Diccionarios.

Números

En Python hay, esencialmente, tres tipos de números: enteros, reales y complejos; que corresponden a los conjuntos usuales que encontramos en Matemática. Podemos preguntar el tipo de un objeto escribiendo `type(`⟨*objeto*⟩`)`. Vamos algunos ejemplos de números:

`>>> type(5)`	*Preguntamos el tipo del objeto* 5.
`<type 'int'>`	*Resulta ser un entero (*`int` *denota 'integer').*
`>>> type(3.24)`	*Preguntamos el tipo del objeto* $3{,}24$.
`<type 'float'>`	*Es un número real (*`float` *denota 'floating point number').*
`>>> type(5.0)`	*El tipo de* $5{,}0\ldots$
`<type 'float'>`	*también es* `float`*, diferente del tipo de* 5.
`>>> type(2+3j)`	*Preguntamos el tipo del objeto* $2+3j$.
`<type 'complex'>`	*Es un número complejo.*

Los tres tipos de números se denotan, entonces, como `int`, `float` y `complex`.

Opcional

Los número reales son representados por lo que se denomina números en coma flotante. Naturalmente, no podemos representar a todos los números reales, por lo que es necesario aproximarlos. Es conveniente tener en cuenta que la computadora utiliza la representación binaria y no la decimal, por lo que números que tienen una representación decimal finita pueden tener una representación binaria infinita, por ejemplo, $0{,}2$*. Por lo tanto, la representación binaria debe ser truncada y eso es lo que justifica que* $0{,}2$ *no sea "exactamente"* $0{,}2$ *en la computadora:*

```
>>> 0.2
0.20000000000000001
```

Como ejercicio, calcula la expansión binaria de $0{,}2$*, es decir, encuentra una sucesión* (a_n) *de ceros y unos tal que*

$$0{,}2 = \frac{1}{5} = \sum_{k=0}^{\infty} \frac{a_k}{2^k}$$

Existe otro tipo de número, además de los mencionados antes, que se denomina `long`. Este tipo representa a los números enteros de longitud arbitraria, dado que los enteros de tipo `int` tienen cotas superior e inferior. Cuando alguna operación

sobre números enteros supera esas cotas, el número se convierte a un entero largo. Los números de tipo `long` se diferencian de los de tipo `int` con una 'L' a la derecha del número.

```
>>> import sys
>>> sys.maxint                          Este es el máximo número de tipo int.
2147483647
>>> sys.maxint+1
2147483648L                             Observa la 'L' a la derecha.
>>> type(sys.maxint+1)
<type 'long'>
```

Sobre los números se pueden realizar las operaciones aritméticas básicas (y muchas más). Por ejemplo:

```
>>> 2+3
5
>>> 3*1.5
4.5
>>> 2**4
16
>>> 3/2                     El símbolo '/' es la división. Si los números son ambos
1                               enteros, calcula la división entera.
>>> 3.0/2                   Si alguno de los números es un real,
1.5                             el símbolo '/' calcula la división usual.
>>> import math
>>> math.cos(math.pi)       Calculamos cos π.
-1.0
>>> (1+1j)*(2.5-3j)         El símbolo '*' también calcula el producto de números
(5.5-0.5j)                      complejos.
```

Observa que la división '/' se comporta como división entera o real según el tipo de los argumentos. Para obtener el resto de una división entera se puede usar el operador '%'. Por ejemplo, la expresión `10%3`, devuelve 1, que es el resto de dividir 10 por 3.

Poniéndole nombres a los objetos

Antes de continuar con la descripción de otros tipos de objetos, comentaremos lo que es un *nombre*. Supongamos que queremos hacer varios cálculos con el número 123456. Podríamos tipearlo cada vez que lo necesitemos:

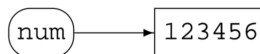
```
>>> 123456*3
370368
>>> 123456/6
20576
>>> 123456-1000
122456
```

Pero esto se vuelve muy aburrido y podríamos cometer errores cada vez que tipeamos el número. Se puede resolver esto dándole un nombre a ese número, por ejemplo `num`, y luego usando ese nombre:

```
>>> num = 123456
>>> num
123456
>>> num*3
370368
>>> num/6
20576
>>> num-1000
122456
```

Cada vez que hagamos referencia al nombre `num`, nos estaremos refiriendo al número 123456. Podemos visualizar esto

pensando en el nombre `num` como una etiqueta que le asociamos al objeto 123456:



Este diagrama y los comentarios anteriores son válidos para cualquier otro objeto, no sólo números. Es conveniente tener este ejemplo en mente cada vez que se hable de *nombres*. Usaremos la palabra *variables* como sinónimo de *nombres*.

Cadenas de caracteres

Las cadenas de caracteres representan cualquier clase de texto (en inglés se llaman *strings*, y usaremos ese nombre por brevedad). Un string puede contener letras, números y símbolos, los cuales se deben escribir entre apóstrofes: `'Esto es un string'`, o entre comillas: `"Esto es un string"`; ambas formas son completamente equivalentes. Los siguientes son todos objetos de tipo `str` (string):

```
"Esto es un string"
'Esto es otro "string" con comillas'      Observar que para poder poner el caracter " es necesario
                                              escribir la cadena de caracteres entre apóstrofes.
"Este 'string' tiene apostrofes"           Para poder poner el caracter ' hay que escribir
                                              la cadena entre comillas.
'Un string con eñe y 12 (números)'
''                                         Este es el string vacío
```

Es importante observar que los objetos `"5"` y `5` son *diferentes*, a pesar de la apariencia similar; el primero es un string y el segundo es un número:

```
>>> type("5")
<type 'str'>
>>> type(5)
<type 'int'>
>>> "4"+1                                  No podemos sumar strings y números, por ejemplo.
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

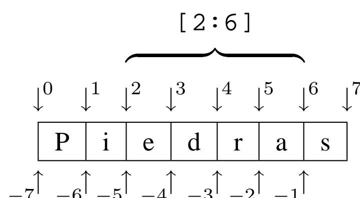
Las últimas líneas son un mensaje de error de Python. Sólo necesitas leer, por ahora, la última línea de tales mensajes, en la cual se indica el error cometido.

> ***Importante***
>
> *Observa el siguiente ejemplo:*
>
> ```
> >>> numero = 4
> >>> numero
> 4
> >>> 'numero'
> 'numero'
> ```
>
> *Observa la diferencia entre los dos últimos casos. La expresión* `'numero'` *es un string, el cual es impreso textualmente. La expresión* `numero` *(sin las comillas) es la variable que, en el ejemplo, refiere al objeto* 4.

Python provee de muchas operaciones básicas para ser aplicadas a las cadenas de caracteres:

```
>>> cadena = 'Pica'                      Asignamos nombres a algunos strings.
>>> otra_cadena = 'piedras'
>>> cadena+otra_cadena                   Los strings se pueden sumar.
'Picapiedras'
>>> 3*cadena                             También se pueden multiplicar por enteros.
'PicaPicaPica'
>>> len(cadena)                          La función len calcula cuantos caracteres contiene
4                                          el string
```

Un cierto número de operaciones se refiere a extraer partes de las cadenas de caracteres. Para eso es útil considerar el siguiente diagrama:




```
>>> cadena = 'Piedras'
>>> cadena[0]                            Tomamos la primer letra del string.
'P'
>>> cadena[2:6]                          Tomamos el fragmento entre el índice 2 y
'edra'                                     el índice 6.
>>> >>> cadena[-1]                       Con índices negativos podemos contar desde la derecha.
's'                                      Por ejemplo, tomamos la última letra del string.
```

Observa que los índices señalan al espacio entre los caracteres, *no* a los caracteres. Cuando extraemos fragmentos de las cadenas de caracteres, podemos omitir algunos de los extremos:

```
>>> cadena[:2]
'Pie'
>>> cadena[4:]
'ras'
>>> cadena[:]
'Piedras'
```

Por último, todos los objetos de Python, sin importar el tipo, tienen asociados un cierto número de funciones (a las cuales se les denomina *métodos* del objeto). Puedes averiguar cuales son esas funciones con la instrucción `dir(objeto)`, de la siguiente manera:

```
>>> dir('')                              Preguntamos por los métodos de los strings.
[ resultado omitido, pruébalo en la computadora ]
>>> dir(5)                               Preguntamos por los métodos de los enteros.
[ también omitido ]
```

Puedes ignorar los nombres de la forma `__nombre__`. Veamos algunos ejemplos de utilización de métodos de los strings:

```
>>> cadena = 'hola, mundo!'
>>> cadena.upper()
'HOLA, MUNDO!'
>>> cadena.replace('mundo', 'gente')
'hola, gente!'
```

Las líneas de la forma `objeto.funcion(...)` se deben leer como "ejecuta la función `funcion` sobre el objeto `objeto` con los parámetros `...`". Experimenta con varios de estos métodos para tener una idea de lo que hacen.

Importante

Puedes encontrar ayuda sobre cualquier función o método de Python con la instrucción `help`*, de la siguiente forma:*

```
>>> help(len)
[ omitido, pruébalo ]
>>> help(''.upper)
[ omitido ]
```

Más aún, puedes pedir ayuda en general con el comando:

```
>>> help()
```

Listas

Las listas son secuencias de objetos. Una lista puede ser pensada como una generalización de los strings, los cuales son secuencias de letras. Las listas pueden ser secuencias de cualquier tipo de objetos, incluso, no necesariamente todos del mismo tipo. Los siguientes son ejemplos de objetos de tipo `list` (listas):

```
[1, 2, 3, 4]                            Una lista de enteros.
[1, 3, 'hola', 3.5-4j, 'otro string']   Un ejemplo de lista con distintos objetos.
['hola']                                Una lista con un solo elemento.
[]                                      La lista vacía.
```

Las listas comparten muchas de las operaciones básicas con los strings. Se pueden extraer elementos y sublistas, de la misma forma que con las cadenas de caracteres:

```
>>> lis = [3, 5, 'hola', 1+1j]
>>> type(lis)
<type 'list'>
>>> len(lis)                            Longitud de una lista.
4
>>> lis[0]                              Primer elemento de una lista.
3
>>> lis[-1]                             Último elemento (recuerda el diagrama de la página 11).
(1+1j)
>>> lis[1:3]                            Extraemos sublistas.
[5, 'hola']
>>> otra_lista = [100, 101, 102]
>>> lis + otra_lista                    Concatenamos listas.
[3, 5, 'hola', (1+1j), 100, 101, 102]
>>> 2*otra_lista                        Repetición de listas.
[100, 101, 102, 100, 101, 102]
```

Las listas pueden ser modificadas despues de creadas, asignándoles nuevos objetos en distintos lugares:

```
>>> lis = [4, 5, 6]                     Creamos una lista.
>>> lis[1] = 'Un string'                Asignamos un nuevo objeto a la posición 1.
>>> lis
[4, 'Un string', 6]
```

Al igual que con las cadenas de caracteres, podemos conocer los métodos de las listas con el comando `dir([])`. Veamos algunos ejemplos de utilización (recuerda usar `help` para saber lo que hace cada función):

```
>>> lis = []                              Creamos una lista vacía.
>>> lis.append(4)                         Le agregamos el objeto 4.
>>> lis
[4]
>>> lis.extend([1, 2, 10, 8])             Extendemos la lista. Esto es, esencialmente,
>>> lis                                     equivalente a: lis = lis + [1,2,10,8].
[4, 1, 2, 10, 8]
>>> lis.sort()                            Ordenamos la lista.
>>> lis
[1, 2, 4, 8, 10]                          ¡Está ordenada!
```

Ya dijimos que una lista puede contener cualquier tipo de objeto. Dado que una lista es en sí misma un objeto, ella puede contener otras listas. Por ejemplo:

```
>>> lis = [8, 'un string', ['A', 'B']]    El último elemento de la lista lis es otra lista.
>>> lis[2]
['A', 'B']                                lis[2] refiere a la lista ['A', 'B'],
>>> lis[2][0]                               por lo que lis[2][0] refiere al primer
'A'                                         elemento de esa lista.
```

Diccionarios

Un *diccionario* es una secuencia de "asociaciones". Una *asociación* consiste de un objeto al cual se le asocia otro. Por ejemplo, en un diccionario de la vida real una asociación es una palabra junto con su definición. En Python los diccionarios pueden ser bastante más generales. Los siguientes son objetos de tipo `dict` (diccionario):

```
{'nombre': 'Juan Perez', 'edad': 21}
{1: 'uno', 2: 'dos', 33: 'treinta y tres'}
{'lista1': [1,2,3], 'lista2': [], 123.5: [45]}
{}                                        Diccionario vacío.
```

A las dos partes de una asociación se les denomina *clave* y *valor* (en inglés se denominan *key* y *value*). En la asociación `'edad': 21`, el string `'edad'` es la clave y el número 21 el valor. Hay ciertas restricciones sobre los tipos de objetos que pueden ser claves pero no las mencionaremos aquí (ver sección 6). Veamos algunos ejemplos de la utilización de diccionarios en Python:

```
>>> dic = {'nombre': 'Juan Perez', 'edad': 21}
>>> type(dic)
<type 'dict'>
>>> dic['nombre']                         Consultamos el objeto asociado con la clave 'nombre'.
'Juan Perez'
>>> dic['edad']
21
```

Los diccionarios pueden ser vistos como una generalización de las listas. Mientras que en estas últimas nos referimos a sus objetos mediante índices (`lista[0]`, `lista[1]`, etc.), en los diccionarios podemos usar otros tipos de claves (números de cualquier tipo, strings, etc.).

Al igual que con las cadenas de caracteres y las listas, se pueden conocer los métodos de los diccionarios mediante `dir({})` y la función `help`. Por ejemplo:

```
>>> len(dic)                              La longitud de un diccionario es el número
2                                           de asociaciones.
>>> dic.keys()                            Lista todas las claves del diccionario.
['nombre', 'edad']
>>> dic.values()                          Lista todos los valores del diccionario.
['Juan Perez', 21]
```

El objeto ’None’

Python incluye un tipo de objeto para indicar la ausencia de objetos o para representar un objeto *nulo*. Existe un único objeto de este tipo, y su nombre es `None`. En la sección 5 veremos que las funciones que no especifican un valor de retorno, devuelven `None`.

3. Decisiones

A partir de esta sección describimos como utilizar los objetos que consideramos en la sección 1. En esta sección veremos como comparar objetos y realizar acciones condicionales de acuerdo a la respuesta obtenida.

Condiciones

Las *condiciones* son expresiones que pueden ser verdaderas o falsas. En Python, el valor *verdadero* se representa por el objeto `True` y el valor *falso* se representa por el objeto `False`, ambos de tipo `bool` (booleanos). Las siguientes son condiciones:

```
>>> 1 < 2                                 ¿Es 1 menor que 2?
True                                      Si.
>>> 3 == 4                                ¿Es 3 igual a 4?
False                                     No.
>>> 'hola' in ['hello', 'hi', 'hola']     ¿Es 'hola' un elemento de la lista?
True                                      Si.
```

Hay muchos tipos de expresiones que verifican si algo es cierto o no, algunas de ellas son:

`x < y`	verdadero si `x` es menor que `y`,
`x <= y`	verdadero si `x` es menor o igual que `y`,
`x == y`	verdadero si `x` es igual a `y`,
`x != y`	verdadero si `x` es distinto de `y`,
`x in lista`	verdadero si `x` es un elemento de `lista`,
`x not in lista`	verdadero si `x` no es un elemento de `lista`,
`not` *condición*	verdadero si *condición* es falsa,
cond1 `and` *cond2*	verdadero si *cond1* y *cond2* son verdaderas,
cond1 `or` *cond2*	verdadero si *cond1* o *cond2* (o ambas) son verdaderas.

Las condiciones se pueden combinar para formar condiciones más complejas utilizando `and`, `or` y `not`. También se pueden combinar realizando verificaciones múltiples, como por ejemplo la expresión `1 < x <= 2`, la cual es cierta si lo son `1<x` y `x<=2`.

Instrucción 'if'

A partir de ahora es conveniente que los ejemplos sean tipeados en un archivo y ejecutados como se indica en la sección 0. También puedes escribirlos directamente en el intérprete, aunque tendrás que escribir el código cada vez que quieras ejecutarlo. En este caso, observa que en algunas situaciones, el prompt de Python cambiará de '`>>>`' a '`...`'. Esto significa que se está continuando la línea anterior. Para finalizar y volver al prompt normal, presiona `Enter` *en una línea vacía.*

Las condiciones son usadas para ejecutar acciones de forma condicional mediante la instrucción `if`. La estructura básica de la instrucción `if` tiene la forma:

```
if ⟨condición⟩:
    ⟨código a ejecutar si la condición es verdadera⟩
else:
    ⟨código a ejecutar si la condición es falsa⟩
```

Para saber donde empiezan y donde terminan los bloques de código dentro de la instrucción `if`, se deben indentar respecto a la columna donde comienza el `if` o el `else`. Usualmente se utilizan cuatro espacios. Este comentario es válido para todas las demás construcciones que veremos más adelante.

La parte de la instrucción `else` es opcional, puedes omitirla cuando no es necesaria. Veamos algún ejemplo:

```
hora = 8
if hora <= 12:
    print 'Antes del mediodía'
else:
    print 'Después del mediodía'
```

La instrucción `print` *imprime en pantalla un objeto cualquiera (ver más adelante).*

El programa anterior imprimirá el texto `Antes del mediodía`, puesto que la condición `hora <= 12` es cierta. Si se quieren verificar varias condiciones hasta que una de ellas sea cierta se puede usar una construcción de la forma:

```
if numero < 0:
    print numero, 'es negativo'
elif numero > 0:
    print numero, 'es positivo'
else:
    print 'Es cero'
```

`elif` *es la abreviación de* `else if`.

Ya dijimos que Python considera al valor `True` como cierto y al `False` como falso. Sin embargo el concepto de verdad y falsedad es más general. Todo objeto representa "cierto" o "falso". Los objetos "llenos" o "no vacíos" representan verdadero. Los objetos "vacíos" representan falso. Por ejemplo, los objetos `5`, `'pepe'`, `[4, 5, 6]`, `{'edad': 21}`, `1+1j`, representan verdadero. Los objetos `0`, `[ ]`, `{ }`, `None`, representan falso. Se puede consultar el valor lógico de un objeto con la función `bool`:

```
>>> bool([4, 5, 6])
True
>>> bool({})
False
```

La instrucción `if` acepta cualquier objeto como condición, y toma su valor lógico para decidir si el objeto representa verdadero o falso. De esta manera es válido escribir:

```
if lista:
    print 'Lista NO vacía'
else:
    print 'Lista vacía'
```

La instrucción 'print'

La instrucción `print` que hemos usado en los ejemplos anteriores imprime cualquier tipo de objeto en la pantalla. Se pueden imprimir varios objetos en la misma línea separándolos con '`,`' (coma).

```
>>> numero = 4
>>> print 'El objeto asignado a "numero" es', numero
El objeto asignado a "numero" es 4
```

Cuando se requiere imprimir objetos con un formato determinado, son útiles las expresiones de formateo de texto. Vemos un ejemplo para ilustrarlo:

```
>>> print 'El primer número es %d y el segundo es %d' %(8, 9)
El primer número es 8 y el segundo es 9
```

Observa que los caracteres '`%d`' dentro del string fueron sustituidos por los números que colocamos a continuación del string, precedidos por el caracter '`%`'. Los caracteres `%d` indican que lo que se va a colocar en ese lugar es un número entero. Hay varios indicadores que se pueden utilizar, algunos de ellos son:

`%d`	lugar para un entero
`%.nf`	lugar para un real con *n* dígitos después del punto decimal
`%s`	lugar para un objeto cualquiera

Por ejemplo:

```
>>> import math
>>> print 'Pi = %.3f' %(math.pi)
Pi = 3.142
```

Un ejemplo

Como ejemplo más completo de lo que hemos visto hasta ahora prueba el siguiente código. Sálvalo en un archivo y ejecútalo como se mostró en la sección 1.

```
fulano = raw_input('Ingresa tu nombre: ')
print 'Bienvenido,', fulano

nacimiento = int(raw_input('Año de nacimiento: '))
actual = int(raw_input('Año actual: '))
cumple = raw_input('¿Ya cumpliste años? ')

cumple = cumple.lower()                    ¿Para qué sirve esta linea?
if cumple in ['si', 's', 'yes', 'y']:
    edad = actual-nacimiento
elif cumple in ['no', 'n']:
    edad = actual-nacimiento-1
else:
    print 'No entiendo lo que dices'
    edad = -1                              ¿Por qué asignamos -1 a edad?

if edad >= 0:
    print fulano, 'tu edad es', edad
```

La función `raw_input` imprime el string que le fue pasado como argumento, espera que el usuario teclee una respuesta y la devuelve como un string. La función `int` convierte un string a un entero (por ejemplo `int("12")` devuelve el entero 12).

4. Iteraciones

En esta sección describimos la forma de repetir instrucciones un cierto número de veces. Estas construcciones se denominan *bucles*. Python tiene dos tipos de bucles, los cuales se usan en situaciones bien diferenciadas. Uno de ellos (el bucle `for`) es usado, en general, cuando sabemos de antemano cuantas iteraciones es necesario realizar. El otro (el bucle `while`) es utilizado cuando no sabemos cuantas iteraciones son necesarias, sino que depende de alguna condición.

Bucle 'for'

La estructura del bucle `for` tiene la forma:

```
for ⟨var⟩ in ⟨secuencia⟩:
    ⟨código⟩
```

El bucle consiste de una variable ⟨*var*⟩, una secuencia ⟨*secuencia*⟩ a ser recorrida (que puede ser una lista, un string, un diccionario o cualquier objeto para el que tenga sentido "iterar" sobre sus elementos) y un fragmento de código ⟨*código*⟩ a ser ejecutado tantas veces como elementos tiene la lista. En cada una de las repeticiones, la variable ⟨*var*⟩ hace referencia a uno de los elementos de la lista ⟨*lista*⟩. Por ejemplo:

```
for j in [1,2,3,4]:
    print 'La variable "j" vale', j
```

El programa anterior, al ser ejecutado, imprime:

```
La variable "j" vale 1
La variable "j" vale 2
La variable "j" vale 3
La variable "j" vale 4
```

Una función útil para utilizar con el bucle `for` es la función `range`. La expresión `range(`n`)` genera una lista de números enteros desde 0 hasta $n-1$. Como ejemplo de utilización, el siguiente programa calcula la suma de los números enteros desde 0 hasta 999:

```
c = 0
for i in range(1000):
    c = c+i                                  Toma el valor de c, le suma el valor de i y asigna
                                                el resultado nuevamente a c.
print 'La suma desde 0 hasta 999 es', c      Debería dar (100 × 99)/2.
```

En los ejemplos anteriores, la lista recorrida era de números enteros. Es importante tener en cuenta que el bucle `for` puede recorrer una lista de objetos de cualquier tipo, y no necesariamente todos del mismo tipo, como muestra el siguiente ejemplo:

```
for x in ['un string', 3.5, 1+1j, [1,2,3], 15]:
    print 'La variable "x" tiene tipo', type(x)
```

Cuando el bucle `for` se usa con un string, en lugar de una lista, la variable del bucle recorre los caracteres del string uno a uno. Por ejemplo, los dos bucles siguientes son equivalentes:

```
for c in 'piedra':
    print c

for c in ['p', 'i', 'e', 'd', 'r', 'a']:
    print c
```

Por otra parte, si el bucle `for` se utiliza con un diccionario, la variable recorre las *claves* del diccionario. Por ejemplo:

```
dic = {'pepe': 20, 'juan': 25, 'ana': 18}
for x in dic:
    print x, 'tiene', dic[x], 'años'
```

produce el siguiente resultado (observa que el orden en que se recorre un diccionario no necesariamente es el mismo en que se escriben los elementos):

```
juan tiene 25 años
pepe tiene 20 años
ana tiene 18 años
```

Bucle 'while'

La estructura del bucle `while` tiene la forma:

```
while ⟨condición⟩:
    ⟨código⟩
```

Este bucle repite el fragmento de código ⟨*código*⟩ mientras la condición ⟨*condición*⟩ sea verdadera. Un ejemplo:

```
respuesta = 0
intentos = 0
while respuesta != 5:
    respuesta = int(raw_input('¿Cuanto es 2+3? '))
    intentos = intentos + 1

print 'Bien, te costó', intentos, 'intentos'
```

Este bucle se ejecuta mientras la variable `respuesta` refiera a un número distinto de 5. La primera vez que `respuesta` sea 5, la ejecución continua inmediatamente después del bucle (es decir, en la instrucción `print`, donde la indentación vuelve a ser la misma que el `while`). En cada una de las iteraciones, la variable `intentos` se incrementa en 1 para contar la cantidad de veces que se hizo la pregunta. Observa que este es un claro ejemplo donde no sabemos cuantas iteraciones serán necesarias.

Otro ejemplo:

```
while True:
    print 'Para siempre...'
```

En este caso la condición `True` es, naturalmente, siempre verdadera, por lo que el bucle se ejecuta eternamente, imprimiendo el texto `Para siempre...` en cada iteración (presiona `Ctrl-C` para detener el bucle).

Abortando los bucles

A veces es necesario abandonar un bucle antes de que termine normalmente, o sea antes de agotar la lista a recorrer en el caso del `for`, o antes de que la condición no se verifique en un bucle `while`. La instrucción `break` permiter romper un bucle en cualquier momento y continuar inmediatamente después del final de este:

```
for i in range(10):
    if i>5:                  Salimos del bucle si i es mayor que 5.
        break
    print i,                 Observa la coma al final de la instrucción.
```

Este bucle imprime `0 1 2 3 4 5`, pues cuando la variable `i` refiera al número 6, se ejecutará la instrucción `break`, que hará terminar el bucle.

> ***Más sobre la instrucción 'print'***
>
> *En general, la instrucción* `print obj` *imprime el objeto* `obj` *y luego produce un cambio de línea, es decir, la próxima instrucción* `print` *imprime en la siguiente línea. Por ejemplo:*
>
> ```
> print 1
> print 2
> ```
>
> *imprime*
>
> ```
> 1
> 2
> ```
>
> *Cuando la instrucción* `print` *tiene una coma '`,`' al final, no se produce el cambio de línea, por lo que el siguiente código:*
>
> ```
> print 1,
> print 2
> ```
>
> *imprime*
>
> ```
> 1 2
> ```

A los bucles, tanto `for` como `while`, se les puede agregar una sección al final, que detecta si el bucle terminó de forma normal o si fue abortado por un `break`. Esta sección se denomina `else`. Es importante no confundirla con el `else` de la instrucción `if`; verifica siempre con qué instrucción está alineado el `else`, para saber si corresponde a un bucle o a una decisión.

Por ejemplo, para saber si un número es primo podemos hacer:

```
for i in range(2,n):                    Probamos con todos los números desde 2 hasta n-1.
    if n % i == 0:                      Si i divide a n...
        print n, 'tiene divisor', i
        break                             ... no es necesario continuar.
else:
    print n, 'es primo'                 Si el bucle no fue abortado, n no tiene divisores propios.
```

Observa que el `else` está alineado con el `for` y **no** con el `if`.

Salteando iteraciones

Otras veces es necesario abortar solamente una de las iteraciones y continuar con las siguientes. La instrucción `continue` permite implementar esto.

Por ejemplo:

```
for i in range(10):
    if i == 5:
        continue                        Salteamos la iteración correspondiente a i==5.
    print i,
```

El bucle anterior imprime:

```
0 1 2 3 4 6 7 8 9                       observa que falta el 5.
```

5. Funciones

En esta sección vemos dos formas de encapsular el código: las funciones y los módulos.

Funciones

Las funciones son fragmentos de código, a los cuales se les da un nombre y pueden ser usados tantas veces como sea necesario. Además, esos fragmentos pueden contener variables a las cuales se les asignan valores distintos cada vez que se ejecuta la función (a esas variables se les denomina *parámetros*). Una función es la forma básica de hacer que un fragmento de código sea reutilizable en distintas situaciones.

Usando funciones

Python dispone de muchas funciones ya implementadas, listas para ser usadas. Ya hemos usado algunas de ellas. Por ejemplo, la función `type` que devuelve el tipo de un objeto. También hemos usado algunas funciones matemáticas. Veamos algún ejemplo con más detalle:

```
>>> import math            Incluimos el módulo math (ver módulos más adelante).
>>> math.cos(1)            Invocamos la función math.cos (coseno).
0.54030230586813977
```

En este ejemplo, utilizamos la función `math.cos`, que refiere a la función coseno que se encuentra en el módulo `math`. Esta función toma el entero 1 como argumento y devuelve el real $\cos(1)$. La función `math.cos` es un ejemplo de una función que toma uno o más argumentos y devuelve un valor; que es el significado usual que tiene la palabra *función* en matemática.

Sin embargo, en Python una función puede hacer otras acciones antes de devolver un valor. Un ejemplo de esto es la función `raw_input`, que ya hemos visto antes. Cuando invocamos esta función, por ejemplo: `raw_input('¿Si o no?')`, le pasamos como argumento un string `'¿Si o no?'`; la función realiza dos acciones que consisten en imprimir su argumento y requerir una entrada por teclado; y finalmente devuelve un string que es el texto que se ingresó en el teclado.

Veremos otras posibilidades de las funciones, como parámetros opcionales o parámetros con nombres, en la sección 7.

Definiendo funciones

Además de usar las funciones ya implementadas, podemos crear nuestras propias funciones. La forma general de definir una función es la siguiente:

```
def una_funcion(⟨parámetros⟩):
    ⟨código⟩
```

Tal código define una función con nombre `una_funcion`, que acepta los argumentos indicados por ⟨*parámetros*⟩. Cuando se invoca la función se ejecuta el código ⟨*código*⟩. Dentro de este código se puede incluir una instrucción de la forma `return` ⟨*valor*⟩. Si la ejecución alcanza ese punto, la función termina retornando el valor ⟨*valor*⟩. En caso de que no se incluya la instrucción `return` la función devuelve el objeto `None`.

Veamos un ejemplo:

```
def doble(x):
    return 2*x
```

La función `doble` toma un argumento, le asigna el nombre `x` y devuelve el resultado de hacer `2*x`. Con esta definición, la

expresión `doble(3.5)` devuelve el real 7,0 y la expresión `doble('pe')` devuelve el string `'pepe'`.

Para utilizar la función hay que tener en cuenta dos situaciones. Supongamos que salvamos la definición de la función `doble` en un archivo con nombre '`archi.py`'. Dicha función puede ser usada en el mismo archivo en que se define, en cuyo caso basta con escribir `doble(3.5)`. O puede ser usada fuera de '`archi.py`', ya sea en otro archivo o directamente en el intérprete. En este último caso es necesario hacer:

```
>>> import archi                    Importamos el módulo archi.
>>> archi.doble(3.5)                Invocamos la función doble del módulo archi.
7.0
```

Veamos otro ejemplo. En el archivo '`archi.py`' escribimos:

```
def es_divisible(x, y):             Decide si x es divisible por y.
    if x % y == 0:
        return True                 Verdadero, si x dividido por y da resto 0.
    else:
        return False                Falso, en otro caso.
```

Observa que puede haber más de una instrucción `return`. Según cual se ejecute será el valor que devuelve la función. Para usarla hacemos:

```
>>> import archi
>>> archi.es_divisible(7, 4)        ¿Es 7 divisible por 4?
False
>>> if archi.es_divisible(8, 4):    Utilizamos la función en un if.
...     print '4 divide a 8'
...
4 divide a 8
```

Módulos

Antes de continuar con la descripción de las funciones, comentaremos un poco más en detalle lo que es un módulo. Formalmente un *módulo* es un archivo que contiene código en Python. En general, los módulos contienen varias definiciones de funciones. Esta es una forma de agrupar las funciones que están relacionadas entre sí.

Por ejemplo, en el módulo `math` están agrupadas todas las funciones matemáticas, en el módulo `os` están agrupadas las funciones relacionadas con el sistema operativo, etc. Puedes averiguar cuales son los módulos disponibles en tu sistema con la instrucción `help('modules')`. Luego de obtener los nombres de los módulos, puedes hacer, por ejemplo, `help('sys')`, para obtener la descripción de todas las funciones que contiene dicho módulo.

La instrucción `import` ⟨*módulo*⟩ carga un archivo desde el disco y lo hace disponible en el intérprete de Python. El nombre ⟨*módulo*⟩ es el nombre del archivo *sin la extensión* `.py`. Por ejemplo, el archivo '`archi.py`' se considera como el módulo `archi`. Es importante observar que la instrucción `import` *solamente* carga el archivo la primera vez que es ejecutada. Para volver a cargar un módulo que fue modificado es necesario usar la función `reload(`⟨*módulo*⟩`)`. Para explicar esto, considera la siguiente situación:

```
>>> import archi
>>>                                 Ahora modificamos el archivo 'archi.py':
                                      sustituimos 2*x por el string 'modificada' en
                                      la definición de la función doble.
>>> import archi
>>> archi.doble(2)                  Importar nuevamente el módulo no surte efecto.
4
>>> reload(archi)                   Es necesario usar la función reload.
<module 'archi' from 'archi.pyc'>
>>> archi.doble(2)
'modificada'
```

Ya hemos visto que para usar un módulo es necesario utilizar la instrucción `import`, y luego usar las funciones de ese módulo con la expresión `modulo.funcion(...)`. Hay otra forma que puede ser más cómoda en algunas situaciones:

```
>>> from math import *            Observa la diferencia con import math.
>>> cos(1)                        No necesitamos el prefijo math.
0.54030230586813977
```

> ***Importante***
>
> *Es necesario tener cuidado con la forma anterior de importar módulos. Supongamos que tenemos dos módulos* `mod1` *y* `mod2`*, y ambos contienen una función con nombre* `fun`*. Eventualmente, pueden ser funciones diferentes. Si importamos estos módulos con las expresiones* `from mod1 import *` *y* `from mod2 import *`*, las definiciones de la función* `fun` *se sobreescriben. En este caso sería necesario usar la forma* `import mod1` *e* `import mod2`*. Las funciones quedan, entonces, diferenciadas por las expresiones* `mod1.fun` *y* `mod2.fun`*.*

Por último, una receta para los módulos que escribas. Puede suceder que generes un archivo, digamos '`archi.py`', y que cuando intentes importarlo se produzca el siguiente error:

```
>>> import archi
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ImportError: No module named archi
```

Esto significa que Python no pudo encontrar el archivo. Lo que sucede es que Python **no** busca en todo el disco, eso sería demasiado lento. En general, sólo busca en ciertos directorios donde están los módulos propios de Python y luego en el directorio de donde se llamó al intérprete (en Linux), o en el directorio donde fue instalado el Python (en Windows). Para instruir al intérprete a buscar en otros lugares puedes hacer, por ejemplo:

```
>>> import sys
>>> sys.path.append('C:\\Mis Documentos\\Mis Módulos')
```

La lista `sys.path` contiene todos los directorios donde Python busca los módulos. En el ejemplo anterior le hemos agregado un directorio personal en donde podemos salvar nuestros módulos. Observa que las dos barras '`\\`' denotan el caracter '`\`'.

Variables locales

Una característica importante de las funciones son las *variables locales*. El código que se escribe en la definición de una función es independiente del resto del código. Las variables usadas en el cuerpo de una función se denominan variables locales. Las asignaciones a variables locales sólo tienen efecto dentro de la función, y fuera de la misma desaparecen o retoman el valor que tenían antes. Veamos un ejemplo:

```
def al_cuadrado(x):
    y = x**2                      La variable y es una variable local.
    print 'Dentro de la función: x = %s, y = %s' %(x, y)

y = 0
x = [1, 2, 3]
print 'Antes de la función: x = %s, y = %s' %(x, y)
al_cuadrado(5)
print 'Después de la función: x = %s, y = %s' %(x, y)
```

Al ejecutarlo, este programa imprime:

```
Antes de la función: x = [1, 2, 3], y = 0
Dentro de la función: x = 5, y = 25
Después de la función: x = [1, 2, 3], y = 0
```

También observa que el parámetro x es una variable local, que no interfiere con otra variable definida afuera de la función.

6. Objetos II

En esta sección describimos algunas propiedades adicionales de los objetos y las variables de Python. Esta sección es una continuación de la sección 2.

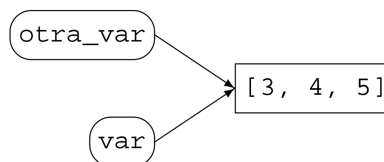
Más sobre las variables

Como indicamos en la sección 1, las variables o nombres son etiquetas que se le asignan a los objetos para poder referenciarlos mas tarde. Un nombre puede referenciar a distintos objetos en distintas circunstancias; es decir, el nombre **no** tiene información sobre el tipo del objeto al cual apunta:

```
>>> var = 1                    Asignamos el nombre var al entero 1.
>>> type(var)
<type 'int'>
>>> var = 'un string'          Ahora el nombre var apunta a un string.
>>> type(var)
<type 'str'>
```

También puede haber varios nombres apuntando al mismo objeto:

```
>>> var = [3, 4, 5]
>>> otra_var = var             Asignamos el nombre otra_var al mismo objeto
                                  al que apunta var.
```

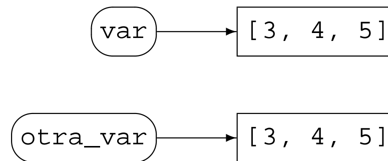


Para comprobar que `var` y `otra_var` apuntan al mismo objeto puedes tratar de modificar el objeto al que referencia `var` y verificar que la misma modificación se refleja en el objeto al cual referencia `otra_var`; o puedes utilizar el operador `is`, como se muestra en el siguiente ejemplo:

```
>>> var[0] = 'un string'       Modificamos el objeto al que referencia var.
>>> var
['un string', 4, 5]
>>> otra_var                   Y el que referencia otra_var también se modifica.
['un string', 4, 4]
>>> var is otra_var            La expresión obj1 is obj2 devuelve True si
True                              ambos objetos son el mismo.
>>> var == otra_var            Naturalmente, deben ser iguales también.
True
```

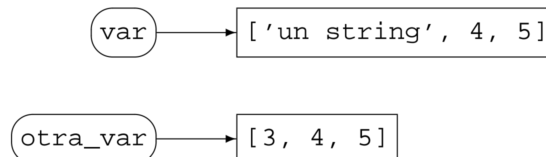
Pero es necesario tener mucho cuidado con situaciones como la que siguen. Compara cuidadosamente el diagrama anterior con el que sigue:

```
>>> var = [3, 4, 5]            Creamos una lista y le asignamos el nombre var.
>>> otra_var = [3, 4, 5]       Creamos otra lista y le asignamos el nombre otra_var.
```



En este último caso hemos creado dos listas, que son iguales elemento a elemento, pero sin embargo son dos objetos diferentes. Podemos comprobar esto modificando una de las listas, igual que antes:

```
>>> var is otra_var                     ¿Son el mismo objeto?
False                                   No.
>>> var == otra_var                     ¿Son iguales?
True                                    Si.
>>> var[0] = 'un string'
>>> var
['un string', 4, 5]
>>> otra_var
[3, 4, 5]                               El objeto al que apunta otra_var no fue modificado.
```



No te preocupes si no entiendes completamente estas diferencias la primera vez. Lo importante es que pienses en las variables como etiquetas que se asignan a los objetos; *nunca* como casilleros en donde se colocan objetos.

Objetos mutables e inmutables

Algunos objetos en Python son *mutables*, es decir, pueden ser modificados. Ejemplos de estos objetos son las listas o los diccionarios. Ya hemos visto que a una lista se le puede cambiar uno de los elementos; por ejemplo:

```
>>> lista = [3, 4, 5]
>>> lista[-1] = 1000
>>> lista
[3, 4, 1000]
>>> lista.append('hola')
>>> lista
[3, 4, 1000, 'hola']
```

Las operaciones anteriores **no** crean listas nuevas, sino que modifican la lista original. Lo mismo le sucede a los diccionarios con respecto a la mayoría de sus funciones asociadas.

Sin embargo, otros objetos son *inmutables*, lo que significa que no pueden ser modificados. Por ejemplo, los números o los strings son objetos inmutables. Cuando realizamos operaciones para modificar esos objetos, en general, creamos **nuevos** objetos. Por ejemplo:

```
>>> texto = 'Hola, mundo!'
>>> nuevo_texto = texto.upper()
>>> nuevo_texto is texto                ¿Apunta nuevo_texto al mismo objeto que texto?
False                                   No.

>>> texto[0] = 'h'                      Intentamos modificar la primer letra de texto.
Traceback (most recent call last):      No podemos.
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
```

La principal razón para diferenciar estas dos categorías de objetos es la siguiente: solamente objetos inmutables pueden ser usados como claves en los diccionarios. Puedes intentar construir diccionarios cuyas claves sean listas:

```
>>> d = {}
>>> lista = [3, 4, 5]
>>> d[lista] = 100                         Tratamos de asociar a la clave [3,4,5] el objeto 100.
Traceback (most recent call last):         Error.
  File "<stdin>", line 1, in ?
TypeError: list objects are unhashable
```

Existe un tipo muy similar a las listas, que se denomina `tuple` (tuplas). La principal diferencia con las listas es que éstas son mutables y las tuplas son inmutables (por lo que pueden ser usadas como claves en los diccionarios). Las tuplas son secuencias de objetos que se escriben entre paréntesis:

```
>>> var = (3, 4, 5)                        Creamos una tupla.
>>> type(var)
<type 'tuple'>
>>> var[0] = 100                           Tratamos de modificar el primer elemento de la tupla.
Traceback (most recent call last):         Observa que es el mismo error que obtuvimos
  File "<stdin>", line 1, in ?               en el ejemplo con strings.
TypeError: object doesn't support item assignment
```

Como las tuplas son inmutables, pueden ser usadas como claves en los diccionarios:

```
>>> d = {}
>>> d[(3, 4, 5)] = 'hola'
>>> d[()] = 'hello'                        () es la tupla vacía.
>>> d
{(): 'hello', (3, 4, 5): 'hola'}
```

Algunos otros ejemplos con tuplas:

```
>>> var1 = (4)                             Observa la diferencia entre var1 y var2.
>>> var2 = (4,)
>>> type(var1)
<type 'int'>                               var1 referencia al entero 4.
>>> type(var2)
<type 'tuple'>                             var2 referencia a una tupla con un solo elemento.
>>> len(var2)
1
>>> var2[0]
4
```

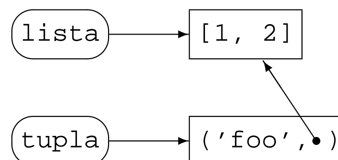
La moraleja es usar tuplas cuando no sea necesario modificar el objeto después de creado y usar listas en los demás casos. Siempre es posible convertir listas en tuplas y viceversa, con las funciones `tuple` y `list`. La expresión `tuple(`⟨*lista*⟩`)` devuelve una tupla con exactamente los mismos objetos que contiene ⟨*lista*⟩ y la expresión `list(`⟨*tuple*⟩`)` realiza la operación inversa.

A pesar de que las tuplas no pueden ser modificadas, **si** pueden ser modificados sus elementos, en caso de que estos sean mutables. Observa con cuidado el próximo ejemplo:

```
>>> lista = [1, 2]
>>> tupla = ('foo', lista)                Creamos una tupla cuyo segundo elemento es una lista.
>>> tupla
('foo', [1, 2])
>>> tupla[1] = 100                        Cualquier intento de modificar el objeto tupla. . .
Traceback (most recent call last):          produce un error.
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment

>>> lista.append('modificada!')           Modificamos la lista lista.
>>> tupla
('foo', [1, 2, 'modificada!'])            ¡La tupla se modificó!
```

Lo que en realidad fue modificado no fue la tupla, sino uno de sus elementos, puesto que la lista `lista` es mutable. El objeto tupla sigue siendo exactamente el mismo, con los mismo elementos que antes. El diagrama que sigue ilustra la situación, en la cual se observa facilmente que la modificación de los elementos de `lista` *no* modifica el objeto al que apunta `tupla`, dado que no se modifican las flechas.



Cuando este comportamiento no sea lo que desees, puedes crear la tupla `tupla` con una *copia* de la lista `lista`. Cuando creamos una copia de una lista, estamos creando otro objeto distinto que contiene los mismos elementos que la lista original. Esto puede ser realizado de la siguiente manera:

```
>>> orig = [1, 2]
>>> nueva = list(orig)                    Copiamos la lista orig.
>>> nueva                                 Por lo que nueva es exactamente igual a orig.
[1, 2]
>>> nueva.append(100)                     Modificamos nueva.
>>> nueva
[1, 2, 100]
>>> orig                                  Pero orig no es modificada
[1, 2]
```

7. Funciones II y listas por comprehensión

En esta sección comentamos algunos usos avanzados de las funciones, así como la construcción de listas por comprehensión.

Más sobre las funciones

En la sección 4 describimos la forma de definir funciones y vimos algunos ejemplos de su uso. En Python una función es un objeto como cualquier otro. Podemos preguntar el tipo de una función (que será tipo `function` o `builtin_function_or_method`) o podemos pasar una función como argumento a otra función. Por ejemplo:

```
>>> import math
>>> type(math.cos)
<type 'builtin_function_or_method'>

>>> math.cos
<built-in function cos>
>>> math.cos(3)
-0.98999249660044542
```

Observa la diferencia entre la expresión `math.cos` *y la expresión* `math.cos(3)`.

El nombre `math.cos` refiere al objeto de tipo función, que es la función matemática coseno. La expresión `math.cos(3)` es la invocación a la función coseno con el argumento 3.

Como ejemplo, supongamos que queremos escribir en Python un programa que derive (aproximadamente) funciones reales. Una posible solución es:

```
import math

def derivada(f, x):
    epsilon = 0.0001
    return (f(x+epsilon)-f(x))/epsilon

print derivada(math.cos, math.pi/2)
```

Si ejecutas este fragmento de código obtendrás el resultado: $-0{,}99999999833322317$ (o un número similar), bastante cercano a -1 que es el valor de la derivada de $\cos x$ en $x = \pi/2$. Observa que la función `derivada` toma dos parámetros: `f` y `x`. El primero debe ser la función a derivar y el segundo el punto en donde calcular la derivada. Dentro de la función `derivada`, la expresión `f(x)` calcula `math.cos(x)`, ya que la variable `f` referencia a la función que le pasamos como argumento (en este caso la función coseno).

Otro ejemplo. El método de listas `sort` ordena las listas. Por defecto las ordena de forma creciente:

```
>>> lista = [3, 1, 0, 8, 10.5]
>>> lista.sort()
>>> lista
[0, 1, 3, 8, 10.5]
```

Pero al método `sort` se le puede pasar un parámetro, el cual debe ser una función que tome dos parámetros, digamos `a` y `b`, y devuelva -1 si `a<b`, 0 si `a==b` y 1 si `a<b`. De esta manera podemos definir un orden distinto del usual. Por ejemplo, definamos el siguiente orden sobre los pares de números enteros cuya segunda coordenada es positiva:

$$(x, y) < (u, v) \text{ si y sólo si } xv < uy$$

(que es el orden usual sobre los racionales). El código en Python para representar este orden podría ser:

```
def comparar_pares(p, q):
    x, y = p                                  Esta expresión equivale a x=p[0]; y=p[1].
    u, v = q
    if x*v < u*y:
        return -1
    elif x*v == u*y:
        return 0
    else:
        return 1


lista = [(1,2), (2,3), (5,4), (2,6), (-3, 2)]
lista.sort(comparar_pares)
print lista
```

El código anterior produce la salida `[(-3, 2), (2, 6), (1, 2), (2, 3), (5, 4)]`, que es el orden correcto si pensamos los pares como números racionales. Observa que al método `lista.sort`, le pasamos como argumento la función `comparar_pares`.

Parámetros con nombres

Python ofrece la posibilidad de definir funciones en donde algunos de sus parámetros tienen valores asignados por defecto. De esa manera, cuando se invoca la función se puede elegir entre pasarle argumentos o aceptar los valores que tiene por defecto. Por ejemplo:

```
def opcional(n=10):                           El parámetro n es opcional, con valor por defecto 10
    return n
```

Si el código está salvado en el archivo '`archi.py`', podemos hacer:

```
>>> import archi
>>> archi.opcional()                          Utilizamos el valor por defecto.
10
>>> archi.opcional('hola')                    Le pasamos otro valor.
'hola'
```

Para declarar que un parámetro es opcional y que tiene un cierto valor por defecto, alcanza con agregar el código '=⟨*valor por defecto*⟩' a continuación del nombre del parámetro. La única precaución que es necesario tener es que los parámetros opcionales sean los últimos en la lista de parámetros de la función.

Listas por comprehensión

Las *listas por comprehensión* son una forma cómoda y sintética de crear listas utilizando una escritura similar a la definición de conjuntos por comprehensión.

Antes de definirlas formalmente, veamos algunos ejemplos. Supongamos que queremos construir la lista de los números del 1 al 10 elevados al cuadrado. Una posible forma es:

```
def cuadrados():
    lista = []
    for i in range(1,11):                     Recuerda que range(n,m) genera la
        lista.append(i**2)                       lista [n,n+1,...,m-1].

    return lista
```

Cuando ejecutamos este programa (supongamos que lo grabamos en el archivo '`archi.py`') obtenemos:

```
>>> import archi
>>> archi.cuadrados()
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Si pensamos la lista como un conjunto de números enteros, lo que hemos construído es el conjunto $\{i^2 \mid i \in [1 \dots 10]\}$. La sintaxis para simular esta construcción es:

```
>>> lista = [i**2 for i in range(1,11)]
>>> lista
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Esta construcción se denomina *lista por comprehensión*. Supongamos, ahora, que queremos generar la lista de todos los números del 1 al 10 al cuadrado *y tal que el cuadrado es múltiplo de* 8. Podríamos hacer:

```
>>> lista = [i**2 for i in range(1,11) if i**2 % 8 == 0]
[16, 64]
```

La construcción general de una lista por comprehensión tiene las siguientes formas:

```
lista1 = [⟨expresión⟩ for ⟨var⟩ in ⟨lista⟩]
lista2 = [⟨expresión⟩ for ⟨var⟩ in ⟨lista⟩ if ⟨condición⟩]
```

En el primer caso lo que se construye es la lista formada por la evaluación de la expresión ⟨*expresión*⟩ con la variable ⟨*var*⟩ variando sobre la lista ⟨*lista*⟩. La primera expresión es completamente equivalente al siguiente código:

```
lista1 = []
for ⟨var⟩ in ⟨lista⟩:
    lista1.append(⟨expresión⟩)
```

En el segundo caso se construye la lista formada por la evaluación de la expresión ⟨*expresión*⟩ con la variable ⟨*var*⟩ variando sobre la lista ⟨*lista*⟩, pero *solamente* cuando la condición ⟨*condición*⟩ es cierta. La segunda expresión es completamente equivalente al código:

```
lista2 = []
for ⟨var⟩ in ⟨lista⟩:
    if ⟨condición⟩:
        lista2.append(⟨expresión⟩
```

8. Clases

En esta sección explicamos como definir nuevos tipos de objetos. Esto es parte de lo que se denomina *programación orientada a objetos*, aunque sólo tocaremos algunos puntos de esta posibilidad del lenguaje Python. La programación orientada a objetos es un tema muy rico y extenso, por lo que sugerimos que consultes también otros materiales. Puedes encontrar más información en el apéndice (ver *Python Tutorial* y *How to think like a Computer Scientist*). También puedes consultar algún libro sobre programación orientada a objetos, por ejemplo: *Thinking in Python*.

Comentaremos la construcción de nuevos tipos mediante un ejemplo. Ya vimos que Python posee tres tipos básicos de números: enteros, reales y complejos. Vamos a definir un tipo de dato para representar a los números racionales. La instrucción básica para definir un nuevo tipo es `class`:

```
class Racional(object):                        Definimos el tipo Racional.

    def __init__(self, p, q):                  Método para inicializar el objeto.
        self.numerador = p
        self.denominador = q

    def mostrarse(self):                       Método para imprimir el objeto.
        print 'El número racional: %d/%d' %(self.numerador,
                                            self.denominador)
```

En este fragmento de código hemos definido un nuevo tipo con nombre `Racional`. Un objeto de este tipo es inicializado como indica el método '`__init__`', e incluye un método adicional con nombre '`mostrarse`'. Estos métodos son los análogos a los que encuentras cuando haces, por ejemplo, `dir([])`. Supongamos que este código se salva en el archivo '`racionales.py`'. Para usarlo se puede escribir:

```
>>> from racionales import *
>>> x = Racional(3, 6)                         Creamos un objeto de tipo Racional.
>>> x
<racionales.Racional object at 0x81560fc>
>>> type(x)
<class 'racionales.Racional'>
>>> dir(x)                                     Preguntamos los métodos de x.
[ ... ,'denominador', 'mostrarse', 'numerador']
>>> x.mostrarse()
El número racional: 3/6
```

Observa que la función `type` retorna como tipo de `x` el nombre de nuestro nuevo tipo. Comentaremos un poco más sobre la definición del método `__init__`. Todo tipo debe incluir un método con este nombre. Cuando ejecutamos la expresión `x = Racional(3, 6)`, se crea un nuevo objeto, al cual se le asigna el nombre `x`, y se evalúa la expresión `Racional.__init__(x, 3, 6)`, que se encarga de inicializar el nuevo objeto:

```
def __init__(self, p, q):
    self.numerador = p
    self.denominador = q
```

En este ejemplo, la inicialización consiste en asignar a los atributos `numerador` y `denominador` del objeto que está siendo inicializado, los valores `p` y `q`. Un *atributo* de un objeto es simplemente una variable asociada al objeto, de la misma manera que un método es una función asociada al objeto. Puedes verificar esto haciendo:

```
>>> x = Racional(3, 6)
>>> x.numerador
3
>>> x.denominador
6
```

Las demás funciones que se pueden definir son métodos adicionales, como lo son, por ejemplo, los métodos `append` o `sort` de listas. Todos los métodos deben llevar un primer parámetro adicional, al igual que `__init__`. Al primer parámetro se le suele denominar `self`, puesto que refiere al propio objeto que se está construyendo. En este caso el método `mostrarse` está escrito para imprimir el objeto en una forma legible:

```
>>> print x
<racionales.Racional object at 0x81560fc>
>>> x.mostrarse()
El número racional: 3/6
```

La impresión usual imprime algo difícil del leer.

Hasta el momento, el nuevo tipo es algo incómodo de manejar. Algunos de los inconvenientes son:

1. La forma de imprimir un racional es complicada. Con los otros objetos que hemos visto alcanza con utilizar la instrucción `print`, sería deseable hacer lo mismo con los racionales.

2. Los objetos `Racional(1, 3)` y `Racional(2, 6)` son distintos:

   ```
   >>> Racional(1,3) == Racional(2,6)
   False
   ```

   aunque sería deseable que fueran considerados iguales, ya que representan el mismo número racional, y que fuera impreso en forma normalizada, es decir en la expresión más sencilla.

3. La única forma de usar un objeto de tipo `Racional` es extrayendo la información que contiene (o sea el numerador y el denominador). Por ejemplo, si queremos multiplicar dos racionales deberíamos hacer:

   ```
   >>> x = Racional(1/2)
   >>> y = Racional(2/3)
   >>> z = Racional(x.numerador * y.numerador,
   ... x.denominador * y.denominador)
   ...
   >>> z.mostrarse()
   El número racional: 2/6
   ```

   Queremos calcular `z = x * y`.

   que es una forma bastante engorrosa de manejar los racionales.

Vamos a ver la forma de solucionar cada uno de estos inconvenientes. La estrategia consiste en definir métodos adicionales para los objetos de tipo `Racional` con ciertos nombres especiales. Esos métodos serán invocados cuando se requieran operaciones como comparar objetos, multiplicarlos o imprimirlos. De esa manera podremos controlar completamente como se comportan nuestros objetos.

Por ejemplo, cuando se ejecuta la instrucción `print obj`, se invoca un método con nombre `__repr__` sobre el objeto `obj`. Ese método debe devolver un string y eso es lo que se imprime. Es decir que la instrucción `print obj` es equivalente a `print obj.__repr__()`[1]. Cambia la definición de `Racional` por el siguiente código:

```
def mcd(a, b):
    if a % b == 0:
        return b
    else:
        return mcd(b, a%b)

class Racional(object):

    def __init__(self, p, q):
        d = mcd(p, q)
        self.numerador = p/d
        self.denominador = q/d

    def __repr__(self):
        return '%d/%d' %(self.numerador, self.denominador)
```

Esta función calcula el máximo común divisor mediante el algoritmo de Euclides.

Normalizamos el numerador y el denominador.

[1] Esto no es estrictamente cierto pero es bastante cercano a la realidad. Consulta el *Python Reference Manual* para conocer los detalles exactos.

Ahora, puedes intentar imprimir un objeto de tipo `Racional` y el resultado será más placentero:

```
>>> x = Racional(3, 6)
>>> print x
1/2
```

De la misma manera, cuando se comparan dos objetos `o1 == o2`, la expresión que se evalua es `o1.__eq__(o2)`. De manera que si queremos controlar la comparación de racionales, debemos escribir ese método en nuestra definición de `Racional`. Agrega el siguiente código dentro de la instrucción `class`:

```
def __eq__(self, otro):
    if type(otro) is int:                 ¿Es otro un entero?
        otro = Racional(otro, 1)          Si lo es, lo convertimos a un racional de denominador 1.

    if type(otro) is Racional:
        return self.numerador == otro.denominador \
               and self.denominador == otro.numerador
    else:
        return False                      En cualquier otro caso, retornamos falso.
```

Con esta modificación, la comparación se comporta como se debe:

```
>>> x = Racional(6,3)
>>> y = Racional(8,4)
>>> z = 2
>>> w = Racional(1,2)
>>> x == y
True
>>> x == z
True
>>> x == w
False
```

Otros métodos similares pueden usarse para controlar el orden de los racionales:

- `__lt__`: `(x < y)` equivale a `x.__lt__(y)`
- `__le__`: `(x <= y)` equivale a `x.__le__(y)`
- `__gt__`: `(x > y)` equivale a `x.__gt_(y)`
- `__ge__`: `(x >= y)` equivale a `x.__ge__(y)`
- `__ne__`: `(x != y)` equivale a `x.__ne__(n)`

Por último, para solucionar el tercer problema, existen varios métodos que permiten controlar la forma en que los objetos se suman, restan, multiplican, dividen y exponencian. Por simplicidad, sólo definimos la suma (codifica el resto de los métodos como ejercicio). Los métodos especiales tienen los nombres:

- `__add__`: `(x + y)` equivale a `x.__add__(y)`
- `__sub__`: `(x - y)` equivale a `x.__sub__(y)`
- `__mul__`: `(x * y)` equivale a `x.__mul__(y)`
- `__div__`: `(x / y)` equivale a `x.__div__(y)`
- `__pow__`: `(x ** n)` equivale a `x.__pow__(n)`

Incluye el siguiente código a continuación del código para `__eq__`:

```
def __add__(self, otro):
    if type(otro) is int:                    Si otro es un entero, lo pasamos a un racional.
        otro = Racional(otro, 1)

    den_suma = self.denominador * otro.denominador
    num_suma = self.numerador * otro.denominador + \
               otro.numerador * self.denominador
    return Racional(num_suma, den_suma)
```

Lo probamos para ver si funciona correctamente:

```
>>> x = Racional(1, 3)
>>> y = Racional(2, 3)
>>> z = Racional(1, 2)
>>> x + y
1/1
>>> x + z
5/6
>>> x + 2
7/3
```

Como regla general, todos las construcciones del lenguaje Python pueden ser modificadas definiendo métodos adecuados sobre los objetos. Consulta la documentación del lenguaje para obtener una lista completa de los métodos especiales existentes.

A. Referencias y material adicional

Referencias sobre el lenguaje Python

`www.python.org`:
Es el sitio principal del lenguaje Python. Ahí puedes bajar la última versión de lenguaje, la documentación, etc.

`www.cmat.edu.uy/python`:
Página para el curso de *Introducción a la Computación*. Ahí puede encontrar prácticos, ejemplos y módulos adicionales usados en el curso.

Python Tutorial:
Es una introducción al Python escrita por el creador de lenguaje, Guido van Rossum. Es una introducción bastante rápida, muy adecuada si ya tienes experiencia en programación. Está incluida en la instalación del lenguaje o puedes obtenerla en `http://www.python.org/doc/current/tut/tut.html`.

Python Library Reference:
Es la documentación definitiva sobre los módulos del lenguaje Python. Está incluida en la instalación del lenguaje o puedes obtenerla en `http://www.python.org/doc/current/lib/lib.html`.

Python Reference Manual:
Es la especificación formal del lenguaje Python. Disponible con la instalación del lenguaje o en `http://www.python.org/doc/current/ref/ref.html`.

Material adicional

How to think like a Computer Scientist:
Es un excelente libro de texto para complementar estas notas. Los temas están tratados en diferente orden, pero de forma más completa. Disponible libre en la web: `http://www.ibiblio.org/obp/thinkCSpy`.

Structure and interpretation of computer programs:
Es uno de los mejores libros sobre programación general escritos hasta el momento. Utiliza el lenguaje *Scheme*, pero puede ser usado aunque no se conozca el lenguaje. Puede obtenerse en `http://www-mitpress.mit.edu/sicp`.

Thinking in Python:
Es un libro en construcción sobre programación orientada a objetos usando Python. Puede bajarse de `http://www.mindview.net/Books/TIPython`.

This is an unofficial translation of the GNU Free Documentation License into /spanish/. It was not published by the Free Software Foundation, and does not legally state the distribution terms for software that uses the GNU FDL–only the original English text of the GNU FDL (http://www.gnu.org/copyleft/fdl.html) does that. However, we hope that this translation will help /spanish/ speakers understand the GNU FDL better.

Esta es una traducción no oficial de la GNU Free Document License (GFDL), versión 1.1 (de Marzo de 2000), que cubre manuales y documentación. Esta traducción no tiene ningún valor legal, ni ha sido comprobada de acuerdo a la legislación de ningún país en particular. Se incluye sólo con propósitos educativos. Para efectos legales por favor remítase al original en inglés (http://www.gnu.org/copyleft/fdl.html). Esta traducción fue realizada en Mayo de 2000 por Igor Támara (ikks@bigfoot.com) y Pablo Reyes (reyes_pablo@hotmail.com). Fue revisada en Enero de 2001 por Vladimir Támara (vtamara@gnu.org) y actualizada por Luis Miguel Arteaga Mejíap (http://www.geocities.com/larteaga/index.html).

B. Licencia de Documentación Libre GNU

Version 1.1, Marzo 2000

Copyright © 2000 Free Software Foundation, Inc. 59 Temple Place, Suite 330, Boston, MA 02111-1307, USA

Se permite la copia y distribución de copias literales de este documento de licencia, pero no se permiten cambios.

Preámbulo

El propósito de esta licencia es hacer que un manual, libro de texto, u otro documento escrito sea /libre/ en el sentido de libertad: para asegurar a todo el mundo la libertad efectiva de copiarlo y redistribuirlo, con o sin modificaciones, bien de manera comercial o no comercial. En segundo término, esta licencia preserva para el autor o para quien publica una manera de obtener reconocimiento por su trabajo, al tiempo que no es considerado responsable de las modificaciones realizadas por terceros.

Esta licencia es una especie de "copyleft" que significa que los trabajos derivados del documento deben a su vez ser libres en el mismo sentido. Esta licencia complementa la Licencia Pública General GNU, que es una licencia de copyleft diseñada para el software libre.

Hemos diseñado esta Licencia para usarla en manuales de software libre, ya que el software libre necesita documentación libre: un programa libre debe venir con los manuales que ofrezcan la mismas libertades que da el software. Pero esta licencia no se limita a manuales de software; puede ser usada para cualquier trabajo textual, sin tener en cuenta su temática o si se publica como libro impreso. Recomendamos esta licencia principalmente para trabajos cuyo propósito sea instructivo o de referencia.

Aplicabilidad y Definiciones

Esta Licencia se aplica a cualquier manual u otro trabajo que contenga una nota del propietario de los derechos de reproducción que indique que puede ser distribuido bajo los términos de esta Licencia. El "Documento", en adelante, se refiere a cualquiera de dichos manuales o trabajos. Cualquier miembro del público es un licenciatario, y será denominado como "Usted".

Una "Versión Modificada" del Documento designa cualquier trabajo que contenga el Documento o una porción del mismo, ya sea una copia literal o con modificaciones y/o traducciones a otro idioma.

Una "Sección Secundaria" es un apéndice titulado o una sección preliminar al prólogo del Documento que tiene que ver exclusivamente con la relación de quien publica o los autores del Documento con el tema general del Documento (o asuntos relacionados) y cuyo contenido no entra directamente en tal tema general. (Por ejemplo, si el Documento es en parte un texto de matemáticas, una Sección Secundaria puede no explicar matemáticas.) La relación puede ser un asunto de conexión histórica, o de posición legal, comercial, filosófica, ética o política con el tema o con materias relacionadas.

Las "Secciones Invariantes" son ciertas Secciones Secundarias cuyos títulos son denominados como Secciones Invariantes, en la nota que indica que el documento es liberado bajo esta Licencia.

Los "Textos de Cubierta" son ciertos pasajes cortos de texto que se listan, como Textos de Portada o Textos de Contra Portada, en la nota que indica que el documento es liberado bajo esta Licencia.

Una copia "Transparente" del Documento significa una copia para lectura en máquina, representada en un formato cuya especificación está disponible al público general, cuyos contenidos pueden ser vistos y editados directamente con editores de texto genéricos o (para imágenes compuestas por pixeles) con programas genéricos para gráficas o (para dibujos) algún editor de dibujos ampliamente disponible, y que sea adecuado para exportar a formateadores de texto o para traducción automática a una variedad de formatos adecuados para ingresar a formateadores de texto. Una copia hecha en un formato que de otra forma sería Transparente pero cuyo formato ha sido diseñado para impedir o dificultar subsecuentes modificaciones por parte de los lectores no es Transparente. Una copia que no es "Transparente" es llamada "Opaca".

Los ejemplos de formatos adecuados para copias Transparentes incluyen ASCII plano sin formato, formato de Texinfo, formato de LaTeX, SGML o XML que usen un DTD disponible ampliamente, y HTML simple que siga los estándares y esté diseñado para modificaciones humanas. Los formatos Opacos incluyen PostScript, PDF, formatos propietarios que pueden ser leídos y editados unicamente con procesadores de palabras propietarios, SGML o XML para los cuáles los DTD y/o herramientas de procesamiento no están disponibles generalmente, y el HTML generado en una máquina, producido por algún procesador de palabras solo con propósitos de presentación.

La "Portada" significa, para un libro impreso, la portada misma más las páginas siguientes necesarias para mantener, legiblemente, el material que esta Licencia requiere que aparezca en la portada. Para trabajos en formatos que no tienen Portada como tal, "Portada" significa el texto cerca a la aparición más prominente del título del trabajo, precediendo el comienzo del cuerpo del texto.

Copia Literal

Usted puede copiar y distribuir el Documento en cualquier medio, sea en forma comercial o no comercial, siempre y cuando esta Licencia, las notas de derecho de autor, y la nota de licencia que indica que esta Licencia se aplica al Documento se reproduzcan en todas las copias, y que usted no adicione ninguna otra condición sobre las expuestas en en esta Licencia. No puede usar medidas técnicas para obstruir o controlar la lectura o copia posterior de las copias que usted haga o distribuya. Sin embargo, usted puede aceptar compensación a cambio de las copias. Si distribuye un número suficientemente grande de copias también deberá seguir las condiciones de la sección 3.

Usted también puede prestar copias, bajo las mismas condiciones establecidas anteriormente, y puede exhibir copias públicamente.

Copiado En Cantidades

Si publica copias impresas del Documento que sobrepasen las 100, y la nota de Licencia del Documento exige Textos de Cubierta, debe incluir las copias con cubiertas que lleven en forma clara y legible, todos esos textos de Cubierta: Textos de Portada en la portada, y Textos de Contra Portada en la contra portada. Ambas cubiertas deben identificarlo a usted clara y legiblemente como quien publica tales copias. La portada debe presentar el título completo con todas las palabras del título igualmente prominentes y visibles. Usted puede adicionar otro material en las cubiertas. Las copias con cambios limitados a las cubiertas, siempre que preserven el título del Documento y satisfagan estas condiciones, puede considerarse como copia literal.

Si los textos requeridos para la cubierta son muy voluminosos para que ajusten legiblemente, debe colocar los primeros listados (tantos como sea razonable colocar) en la cubierta real, y continuar con el resto en páginas adyacentes.

Si publica o distribuye copias Opacas del Documento cuya cantidad excedaí las 100, debe incluir una copia Transparente que pueda ser leída por una máquina con cada copia Opaca, o indicar en o con cada copia Opaca una dirección en una red de computadores publicamente accesible que contenga una copia completa y Transparente del Documento, libre de material adicional, a la cual el público general de la red tenga acceso para bajar anónimamente sin cargo, usando protocolos de redes públicos y estándares. Si usted hace uso de la última opción, deberá tomar medidas razonablemente prudentes, cuando comience la distribución de las copias Opacas en cantidad, para asegurar que esta copia Transparente permanecerá accesible en el sitio indicado por lo menos un año después de su última distribución al público de copias Opacas de esa edición (directamente o a

través de sus agentes o distribuidores).

Se solicita, aunque no es requisito, que contacte a los autores del Documento antes de redistribuir cualquier gran número de copias, para permitirle la oportunidad de que le provean una versión actualizada del Documento.

Modificaciones

Usted puede copiar y distribuir una Versión Modificada del Documento bajo las condiciones de las secciones 2 y 3 anteriores, siempre que usted libere la Versión Modificada bajo esta misma Licencia, con la Versión Modificada asumiendo el rol del Documento, por lo tanto licenciando la distribución y modificación de la Versión Modificada a quienquiera que posea una copia de este. En adición, debe hacer lo siguiente en la Versión Modificada:

1. Uso en la Portada (y en las cubiertas, si hay alguna) de un título distinto al del Documento, y de versiones anteriores (que deberían, si hay alguna, estar listados en la sección de Historia del Documento). Puede usar el mismo título que versiones anteriores del original siempre que quién publicó la primera versión lo permita.
2. Listar en la Portada, como autores, una o más personas o entidades responsables por la autoría o las modificaciones en la Versión Modificada, junto con por lo menos cinco de los autores principales del Documento (Todos sus autores principales, si hay menos de cinco).
3. Establecer en la Portada del nombre de quién publica la Versión Modificada, como quien publica.
4. Preservar todas las notas de derechos de reproducción del Documento.
5. Adyacente a las otras notas de derecho de reproducción, adicionar una nota de derecho de reproducción de acuerdo a sus modificaciones.
6. Incluir, immediatamente después de la nota de derecho de reproducción, una nota de licencia dando el permiso público para usar la Versión Modificada bajo los términos de esta Licencia, de la forma mostrada más adelante en el Addendum.
7. Preservar en esa nota de licencia el listado completo de Secciones Invariantes y de los Textos de las Cubiertas que sean requeridos como se especifique en la nota de Licencia del Documento.
8. Incluir una copia sin modificación de esta Licencia.
9. Preservar la sección con título “Historia”, y su título, y adicionar a esta una sección estableciendo al menos el título, el año,los nuevos autores, y quién publicó la Versión Modificada como reza en la Portada. Si no hay una sección titulada “Historia” en el Documento, crear una estableciendo el título, el año, los autores y quien publicó el Documento como reza en la Portada, añadiendo además un artículo describiendo la Versión Modificada como se estableció en la oración anterior.
10. Preservar la localización en red, si hay , dada en el Documento para acceso público a una copia Transparente del Documento, así como las otras direcciones de red dadas en el Documento para versiones anteriores en las cuáles estuviese basado. Estas pueden ubicarse en la sección “Historia”. Se puede omitir la ubicación en red para un trabajo publicado por lo menos 4 años antes que el Documento mismo, o si quien publicó originalmente la versión a la que se refiere da permiso.
11. En cualquier sección titulada “Agradecimientos” o “Dedicatorias”, preservar el título de la sección, y preservar en la sección toda la sustancia y el tono de los agradecimientos y/o dedicatorias de cada contribuyente que estén incluídas.
12. Preservar todas las Secciones Invariantes del Documento, sin alterar su texto ni sus títulos. Números de sección o el equivalente no son considerados parte de los títulos de la sección.
13. Borrar cualquier sección titulada “Aprobaciones”. Una tal sección no pueden estar incluida en las Versiones Modificadas.
14. No retitular ninguna sección existente como “Aprobaciones” o conflictuar con título de alguna Sección Invariante.

Si la Versión Modificada incluye secciones o apendices nuevos o preliminares al prólogo que califican como Secciones Secundarias y contienen material no copiado del Documento, puede opcionalmente designar algunas o todas esas secciones como invariantes. Para hacerlo, adicione sus títulos a la lista de Secciones Invariantes en la nota de licencia de la Versión Modificada. Tales títulos deben ser distintos de cualquier otro título de sección.

Puede adicionar una sección titulada “Aprobaciones”, siempre que contenga unicamente aprobaciones de su Versión Modificada por varias fuentes–por ejemplo, observaciones de peritos o que el texto ha sido aprobado por una organización como un estándar.

Puede adicionar un pasaje de hasta cinco palabras como un Texto de Portada, y un pasaje de hasta 25 palabras como un texto de Contra Portada, al final de la lista de Textos de Cubierta en la Versión Modificada. Solamente un pasaje de Texto de Portada y un Texto de Contra Portada puede ser adicionado por (o a manera de arreglos hechos por) cualquier entidad. Si el Documento ya incluye un texto de cubierta para la misma cubierta, previamente adicionado por usted o por arreglo hecho por la misma entidad, a nombre de la cual usted actua, no puede adicionar otra; pero puede reemplazar el anterior, con permiso explícito de quien previamente publicó y agregó tal texto.

El(los) autor(es) y quien(es) publica(n) el Documento no dan con esta Licencia permiso para usar sus nombres para publicidad o para asegurar o implicar aprobación de cualquier Versión Modificada.

Combinando Documentos

Puede combinar el Documento con otros documentos liberados bajo esta Licencia, bajo los términos definidos en la sección 4 anterior para versiones modificadas, siempre que incluya en la combinación todas las Secciones Invariantes de todos los documentos originales, sin modificar, y las liste como Secciones Invariantes de su trabajo combinado en la respectiva nota de licencia.

El trabajo combinado necesita contener solamente una copia de esta Licencia, y múltiples Secciones Invariantes idénticas pueden ser reemplazadas por una sola copia. Si hay múltiples Secciones Invariantes con el mismo nombre pero con contenidos diferentes, haga el título de cada una de estas secciones único adicionándole al final de este, entre paréntesis, el nombre del autor o de quien publicó originalmente esa sección, si es conocido, o si no, un número único. Haga el mismo ajuste a los títulos de sección en la lista de Secciones Invariantes en la nota de licencia del trabajo combinado.

En la combinación, debe combinar cualquier sección titulada “Historia” de los varios documentos originales, formando una sección titulada “Historia”; de la misma forma combine cualquier seción titulada “Agradecimientos”, y cualquier sección titulada “Dedicatorias”. Debe borrar todas las secciones tituladas “Aprobaciones.”

Colecciones De Documentos

Usted puede hacer una colección que consista del Documento y otros documentos liberados bajo esta Licencia, y reemplazar las copias individuales de esta Licencia en los varios documentos con una sola copia que esté incluida en la colección, siempre que siga las reglas de esta Licencia para una copia literal de cada uno de los documentos en cualquiera de todos los aspectos.

Usted puede extraer un solo documento de una de tales colecciones, y distribuirlo individualmente bajo esta Licencia, siempre que inserte una copia de esta Licencia en el documento extraido, y siga esta Licencia en todos los otros aspectos concernientes a la copia literal de tal documento.

Agregación Con Trabajos Independientes

Una recopilación del Documento o de sus derivados con otros documentos o trabajos separados e independientes, en cualquier tipo de distribución o medio de almacenamiento, no cuenta como un todo como una Versión Modificada del Documento, siempre que no se clamen derechos de reproducción por la compilación. Tal recopilación es llamada un "agregado", y esta Licencia no aplica a los otros trabajos auto-contenidos y por lo tanto compilados con el Documento, o a cuenta de haber sido compilados, si no son ellos mismos trabajos derivados del Documento.

Si el requerimiento de la sección 3 del Texto de la Cubierta es aplicable a estas copias del Documento, entonces si el Documento es menor que un cuarto del agregado entero, Los Textos de la Cubierta del Documento pueden ser colocados en cubiertas que enmarquen solamente el Documento entre el agregado. De otra forma deben aparecer en cubiertas enmarcando todo el agregado.

Traducción

La traducción es considerada como una clase de modificación, así que puede distribuir traducciones del Documento bajo los términos de la sección 4. Reemplazar las Secciones Invariantes con traducciones requiere permiso especial de los propietarios de los derechos de reproducción, pero usted puede incluir traducciones de algunas o todas las Secciones Invariantes además de las versiones originales de las Secciones Invariantes. Puede incluir una traducción de esta Licencia siempre que incluya también la versión original en ingles de esta Licencia. En caso de un desacuerdo entre la traducción y la versión original en Inglés de esta Licencia, la versión original en Inglés prevalecerá.

Terminación

Usted no puede copiar, modificar, sublicenciar, o distribuir el Documento excepto como lo permite expresamente esta Licencia. Cualquier otro intento de copia, modificación, sublicenciamiento o distribución del Documento es nulo, y terminarán automáticamente sus derechos bajo esta Licencia. Sin embargo, los terceros que hayan recibido copias, o derechos, de su parte bajo esta Licencia no tendrán por terminadas sus licencias siempre que tales terceros permenezcan en total conformidad.

Revisiones Futuras De Esta Licencia

La Free Software Foundation puede publicar nuevas y revisadas versiones de la GNU Free Documentation License de tiempo en tiempo. Tales versiones nuevas serán similares en espíritu a la presente versión, pero pueden diferir en detalles para solucionar problemas o intereses. Vea http://www.gnu.org/copyleft/.

Cada versión de la Licencia tiene un número de versión que la distingue. Si el Documento especifica que una versión numerada particularmente de esta licencia o "cualquier versión posterior" se aplica a este, tiene la opción de seguir los términos y condiciones de esa versión especificada o de cualquiera versión posterior que hubiera sido publicada (no como un borrador) por la Free Software Foundation. Si el Documento no especifica un número de versión de esta Licencia, puede escoger cualquier versión que haya sido publicada(no como un borrador) por la Free Software Foundation.

Addendum: Como usar esta Licencia para sus documentos

Para usar esta licencia en un documento que usted haya escrito, incluya una copia de la Licencia en el documento y ponga el siguiente derecho de reproducción y notas de licencia justo después del título de la página:

> Copyright © año su nombre.
>
> Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being list their titles, with the Front-Cover Texts being list, and with the Back-Cover Texts being list. A copy of the license is included in the section entitled "GNU Free Documentation License".

Si no tiene Secciones Invariantes, escriba "with no Invariant Sections" en vez de decir cuales son invariantes. Si no tiene Textos de Portada, escriba "no Front-Cover Texts" en vez de "Front-Cover Texts being list"; y de la misma forma para los Textos de Contra Portada.

Si su documento contiene ejemplos de código de programa no triviales, le recomendamos liberar estos ejemplos en paralelo bajo una licencia de software libre de su elección, tal como la GNU General Public License, para permitir su uso en software libre.