



UNIVERSIDAD DE GRANADA

ACTIVIDADES DE FORMACIÓN DOCENTE
EN CENTROS, TITULACIONES Y DEPARTAMENTOS

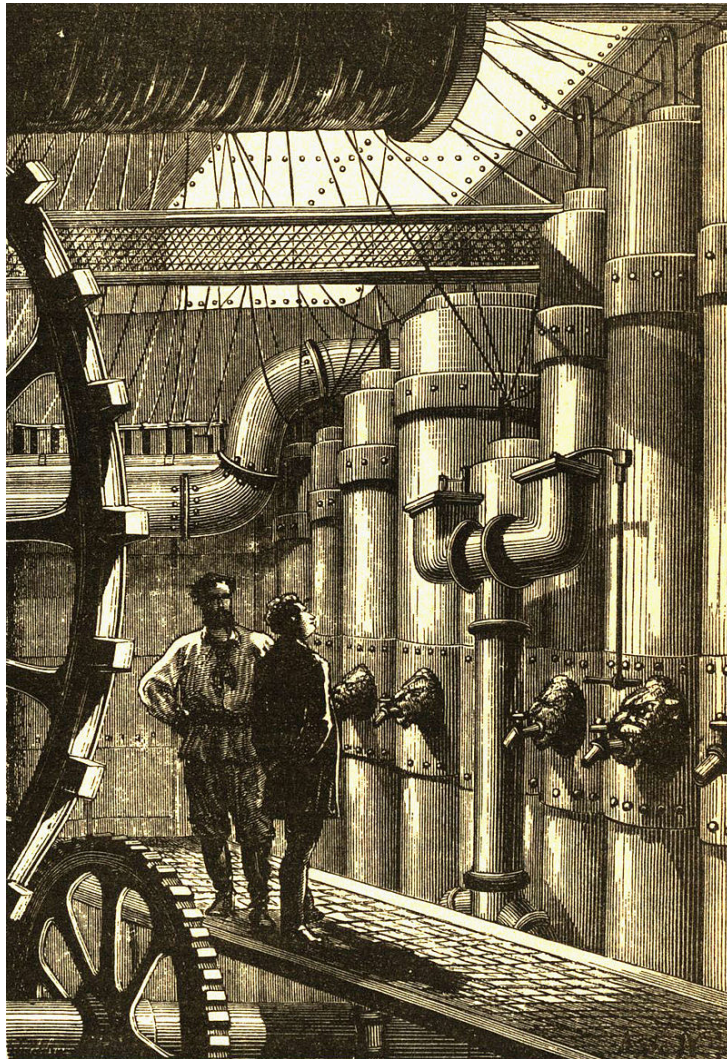
Python
para el
Cálculo Científico y Técnico
Cuaderno de Prácticas (lec. 3, 4, 5 y 8)

Autor:
Francisco M. GARCÍA OLMEDO
Dept. de Álgebra
Univ. de Granada

15 de septiembre de 2015



This entire document containing lectures on Python by [F.M. García Olmedo](#) is licensed under a [Creative Commons Reconocimiento-NoComercial-CompartirIgual 3.0 Unported License](#) Permissions beyond the scope of this license may be available at [F.M. García Olmedo](#).



Índice general

1. Resumen de Nociones con Ejemplos	11
1.1. Introducción	11
1.2. Instalación y Complementación Básica	12
1.3. Uso Básico de Python	14
1.4. Generalidades	16
1.5. Tipos de datos estándar	18
1.6. Operadores básicos de Python	20
1.7. Resumen y ejemplos sobre sentencias	22
1.8. Listas en Python	30
1.9. Definición de Funciones	34
1.10. Transparencia Referencial	36
1.11. Funciones Anónimas con Lambda	37
1.12. Recursividad en Python	38
1.13. Herramientas de Programación Funcional	39
1.14. Listas por Compresión	40
1.15. Listas por Compresión Anidadas	40
1.16. Tuplas	41
1.17. Conjuntos	42
1.18. Diccionarios	43
1.19. Medición del Rendimiento	49
1.20. Argumentos de Funciones con Valores por Defecto	53
1.21. Palabras Clave como Argumento	54
1.22. Lista de Argumentos Arbitrarios	56
1.23. Desempaquetando una Lista de Argumentos	57
1.24. Módulos	60
1.25. Archivos “Compilados” de Python	61
1.26. Leer y Escribir Ficheros en Python	62
1.27. Interacción con el Sistema: el módulo os	65
1.28. Definición de Nuevos Tipos de Datos	65
1.29. Formateo Elegante de la Salida	75
1.30. Manipulación del contenido de un fichero en Python	77
1.31. Legendo del Teclado	84
1.32. Manejo de Excepciones	86
2. Sesión Tercera	91
2.1. Ejercicios	91
3. Sesión Cuarta	93
3.1. Ejercicios	93
4. Sesión Quinta	95
4.1. Ejercicios	95

5. Sesión Octava	97
5.1. Ejercicios	97
A. Complemento base-menos-uno	99
B. Diferencia entre <code>_</code>, <code>__</code> y <code>__xx__</code> en Python	101
C. El Algoritmo de Euclides	105
D. Descripción del Cifrado Afin	107
E. Sobre el ábaco y ... los exponentes elevados.	109

Índice de figuras

1.1. Palabras reservadas de Python.	17
1.2. Precedencia de operadores en Python.	22
1.3. Uso del módulo os de Python.	66
2.1. Primos con cantidad de dígitos elevada.	91
C.1. Algoritmo EUCLIDES	105
D.1. Enumeración del alfabeto.	107

Introducción

Python

La gran herramienta de programación que ha protagonizado una escalada fulgurante a lista *top ten* de los lenguajes más usados tiene **grandes ventajas**: simple y rápido, elegante y flexible, de programación sana y productiva, ordenado y limpio, portable, con gran batería de librerías incluidas, etc. Es el gran ayudante de la pedagogía y enseñanza de la programación de computadoras; objeto muy popularizado que, como sabemos, ha llegado a ser casi “un electrodoméstico” en la casa. Se puede hacer un uso trivial de la máquina, pero la verdad es que en la actualidad los más modestos equipos pueden ser de gran utilidad en la investigación y la pedagogía.

Pero no podemos hablar de la computadora en los términos del párrafo anterior sin incluir en nuestro discurso los lenguajes de programación. Éstas son realmente las herramientas que sirven para establecer el nexo entre máquina y usuario.

Son muchísimos los lenguajes vigentes en la actualidad; la mayoría son de propósito específico. Pueden ser clasificados atendiendo a gran variedad de rasgos: alto o bajo nivel, programación imperativa versus declarativa, etc.

Por sus condiciones destaca entre ellos uno de mucha popularidad, a saber Python. Python es un lenguaje de programación interpretado cuya filosofía hace hincapié en una sintaxis muy limpia que favorece un código legible. Se trata de un lenguaje de programación multiparadigma, ya que soporta orientación a objetos, programación imperativa y, en menor medida, programación funcional. Es un lenguaje interpretado, usa tipado dinámico y es multiplataforma. Es administrado por la Python Software Foundation. Posee una licencia de código abierto, denominada Python Software Foundation License, que es compatible con la licencia pública general de GNU a partir de la versión 2.1.1, e incompatible en ciertas versiones anteriores.

Python fue creado a finales de los ochenta por *Guido van Rossum* en el *Centro para las Matemáticas y la Informática (CWI, Centrum Wiskunde & Informatica)*, en los Países Bajos, como un sucesor del lenguaje de programación ABC, capaz de manejar excepciones e interactuar con el sistema operativo Amoeba. El nombre del lenguaje proviene de la afición de su creador original, Guido van Rossum, por los humoristas británicos *Monty Python*.

Python es un lenguaje de programación multiparadigma, como hemos aclarado antes. Esto significa que más que forzar a los programadores a adoptar un estilo particular de programación, permite varios estilos: programación orientada a objetos, programación imperativa y programación funcional. Otros paradigmas están soportados mediante el uso de extensiones. Python usa tipado dinámico y conteo de referencias para la administración de memoria. Una característica importante de Python es la resolución dinámica de nombres; es decir, lo que enlaza un método y un nombre de variable durante la ejecución del programa (también llamado enlace dinámico de métodos). Otro objetivo del diseño del lenguaje es la facilidad de extensión. Se pueden escribir nuevos módulos fácilmente en C o C++. Python puede incluirse en aplicaciones que necesitan una interfaz programable. Aunque la programación en Python podría considerarse hostil, en algunas situaciones, a la programación funcional tradicional del Lisp, existen bastantes analogías entre Python y los lenguajes minimalistas de la familia Lisp como puede ser Scheme.

Los usuarios de Python se refieren a menudo a la Filosofía Python que es bastante análoga a la filosofía de Unix. El código que sigue los principios de Python de legibilidad y transparencia se dice que es “*pythonico*”.

Contrariamente, el código *opaco* u ofuscado es bautizado como “*no pythonico*” (“*unpythonic*” en inglés). Estos principios fueron famosamente descritos por el desarrollador de Python *Tim Peters* en *El Zen de Python*:

- Bello es mejor que feo.
- Explícito es mejor que implícito.
- Simple es mejor que complejo.
- Complejo es mejor que complicado.
- Plano es mejor que anidado.
- Disperso es mejor que denso.
- La legibilidad cuenta.
- Los casos especiales no son tan especiales como para quebrantar las reglas.
- Aunque lo práctico gana a la pureza.
- Los errores nunca deberían dejarse pasar silenciosamente.
- A menos que hayan sido silenciados explícitamente.
- Frente a la ambigüedad, rechaza la tentación de adivinar.
- Debería haber una —y preferiblemente sólo una— manera obvia de hacerlo.
- Aunque esa manera puede no ser obvia al principio a menos que usted sea holandés.¹
- Ahora es mejor que nunca.
- Aunque nunca es a menudo mejor que ya mismo.
- Si la implementación es difícil de explicar, es una mala idea.
- Si la implementación es fácil de explicar, puede que sea una buena idea.
- Los espacios de nombres (*namespaces*) son una gran idea ¡Hagamos más de esas cosas!

Y las buenas y bellas ideas son reconocidas por sus frutos.

Objetivo

No era necesario un manual más. Si en lo que sigue incluimos nociones de uso de Python y programación, es porque la información ha sido entresacada de la habitual, encontrada en los manuales y páginas más famosas, para facilitar la resolución de los ejercicios que posteriormente son propuestos. Hemos tomado como base esencial los manuales de [Guido van Rossum](#) y el [sitio oficial web de Python](#). Cuando se usan otras excelentes páginas especializadas de autores, se suele extender la cita.

Así pues, se trataría en este curso de seguir las explicaciones orales ofrecidas, complementarlas con los apartados del resumen de programación y abordar la resolución de los problemas propuestos. Al final del curso, en lo que se refiere a este apartado, el alumno que hubiera acometido la resolución de los ejercicios selectos tendría una base elemental de programación en Python.

¹ “Holandés” hace referencia a Guido van Rossum, el autor del lenguaje de programación Python, que es holandés. También hace referencia a la gran concentración de desarrolladores holandeses conocidos en relación a otras nacionalidades.

Capítulo 1

Resumen de Nociones con Ejemplos

¡Explíqueselo, Sr. Blakeney!

*Capt. Jack Aubrey
Master and Commander*

1.1. Introducción

Bajo `sagemath` subyace el lenguaje de programación `Python`, en pleno auge en la actualidad. Contrasta este hecho con la situación de `maxima` que se soporta sobre `lisp`. Así, aunque es evidente que `sagemath` necesita impulso en su desarrollo, no nos cabe duda de que pronto estará a la altura de `maxima` en los pocos aspectos en los que ha quedado rezagado.

Dedicamos este capítulo al estudio de los fundamentos de `Python` en los aspectos usuales. No obstante no entraremos más que en los detalles que pueden ayudar al buen uso de `sagemath` y dejaremos aparte, por ahora, los aspectos que nos llevarían a las profundidades de la programación en el interesante y poderoso lenguaje de programación `Python`.

`Python` fue creado a finales de los ochenta del siglo XX por *Guido van Rossum* en el CWI, acrónimo de *Centrum Wiskunde & Informatica* (Centro para las Matemáticas y la Informática), en los Países Bajos, como un sucesor del lenguaje de programación ABC, capaz de manejar excepciones e interactuar con el sistema operativo Amoeba. Recibe influencias de ABC, ALGOL 68, C, Haskell, Icon, Lisp, Modula-3, Perl, Smalltalk y Java.

`Python` es un lenguaje de programación interpretado cuya filosofía hace hincapié en una sintaxis muy limpia y que favorezca un código legible.

Se trata de un lenguaje de programación multiparadigma, ya que soporta orientación a objetos, programación imperativa y, con las debidas reservas, “programación funcional”.¹ Es un lenguaje interpretado, usa tipado dinámico, es fuertemente tipado y multiplataforma.

Es administrado por la *Python Software Foundation*. Posee una licencia de código abierto, denominada *Python Software Foundation License*, que es compatible con la licencia pública general de GNU a partir de la versión 2.1.1, e incompatible en ciertas versiones anteriores.

`Python` es un lenguaje de alto nivel, interpretado, interactivo, orientado a objetos y que permite el uso de scripts. Entre sus rasgos principales destacaremos que:

1. es interpretado
2. es interactivo

¹Ver una explicación más detallada al respecto en la [Sección 1.10](#)

3. es orientado a objetos
4. es el lenguaje de los principiantes
5. es fácil de aprender
6. es fácil de leer
7. es fácil de mantener
8. tiene una amplia biblioteca básica
9. puede usarse en modo interactivo
10. es portable
11. es extensible
12. interactúa con bases de datos
13. admite programación GUI ("Graphical User Interface", representa la información y acciones disponibles para el usuario a través de iconos gráficos e indicadores visuales)
14. mantiene la efectividad aún variando la escala del trabajo

1.2. Instalación y Complementación Básica

La información más amplia sobre instalación, no sólo para Ubuntu, la tenemos en [esta dirección](#). Debemos saber que, dada la extrema importancia de Python en programación, la mayoría de las instalaciones de Ubuntu y Mac OS X incluyen una versión estable. Para saber cuál es la versión instalada por defecto ejecutaremos en la terminal:

```
python -V
```

En la instalación básica de Ubuntu 12.10 la respuesta a esta orden es

```
Python 2.7.3
```

Si queremos o lo necesitamos, podemos hacer convivir esta versión con otras. Para la instalación de las nuevas podemos ejecutar, por ejemplo:

```
sudo apt-get install python3.3-minimal
```

con lo que instalaríamos lo básico de python 3.3.0. Creemos que no es conveniente cambiar la versión de Python por defecto en el sistema, ya que hay incompatibilidades entre ellas y cuando el sistema recurra a Python debe encontrar su versión por defecto, si queremos evitar problemas.

Como hemos dicho, la instalación por defecto es bastante básica. Puede interesar instalar herramientas para el setup de nuestro Python; con ellas podremos instalar, por ejemplo, pip que es a Python respecto a librerías algo así —para entendernos— como apt-get es a Ubuntu para paquetes y metapaquetes. Para instalar estas herramientas haremos:

```
sudo apt-get install python-setuptools
```

y si lo queremos para Python 3 podemos ejecutar

```
sudo apt-get install python3-setuptools
```

Hecho esto tenemos a nuestra disposición `easy_install`. Si nos fijamos tenemos una para Python 2.7.3 y otra para Python 3:

```
easy_install      easy_install-2.7  easy_install3
```

(`easy_install` y `easy_install-2.7` deben ser lo mismo).

Ahora podemos instalar `pip`:

```
sudo easy_install pip
```

o bien

```
sudo easy_install3 pip
```

o los dos y así quedarían instalados

```
pip      pip-2.7  pip-3.2
```

(`pip` y `pip-2.7` deben ser lo mismo).

Para mostrar un ejemplo de cómo instalar algo (en el ejemplo, el paquete “Markdown”)

```
sudo pip install Markdown
```

Para saber qué tenemos instalado con `pip`, y en cuál versión está, ejecutamos:

```
pip freeze
```

Para desinstalar puede usarse la opción `uninstall`. Por ejemplo, si quisieramos desinstalar `Markdown`, previamente instalado, ejecutaríamos:

```
pip uninstall Markdown
```

En fin, el mejor manual de `pip` que hemos encontrado es [éste](#). Para ver generalidades, y algo menos específico, puede consultarse también [éste otro](#).

Por `emacs` no hemos de preocuparnos pues la instalación básica viene preparada para abrir y entender ficheros `.py` (los ficheros de Python acaban en `.py`). Si no hemos instalado `emacs` podemos hacerlo con la orden

```
sudo apt-get install emacs24
```

o alternativamente

```
sudo apt-get install emacs
```

si somos conservadores y sólo queremos tener `emacs23`, la opción por defecto de Ubuntu.

1.3. Uso Básico de Python

Para ejemplificar lo que expliquemos en esta sección nos basaremos en los siguientes dos ficheros que hemos editado: `fibo.py` y `tools.py`.

El contenido de `fibo.py` es el siguiente:

```
# modulo de numeros Fibonacci

def fib(n):
    # escribe la serie Fibonacci hasta n
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b

def fib2(n): # devuelve la serie Fibonacci hasta n
    resultado = []
    a, b = 0, 1
    while b < n:
        resultado.append(b)
        a, b = b, a+b
    return resultado

if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

El contenido de `tools.py` es el siguiente:

```
# module tools

def sortSr(lista):
    resultado = []
    if len(lista) != 0:
        lista.sort()
        for i in range(len(lista)-1):
            if lista[i] != lista[i+1]:
                resultado.append(lista[i])
        resultado.append(lista[len(lista)-1])
    return resultado

def listasToLista(lista):
    resultado = []
    if len(lista) != 0:
        for i in range(len(lista)):
            if (type(lista[i]) == list or type(lista[i]) == tuple):
                for j in range(len(lista[i])):
                    resultado.append(lista[i][j])
            else:
                resultado.append(lista[i])
    return resultado

def desempquetadoDeListas(lista):
    semilla, resultado = lista, listasToLista(lista)
    while resultado != semilla:
```

```
    semilla, resultado = resultado, listasToLista(resultado)
    return resultado
```

En primer lugar, podemos hacer invocaciones de librerías instaladas como ésta:

```
python -c "import markdown; print markdown.markdown('**Excellent**')"
```

Si hemos codificado un fichero, digamos nuestro `tools.py`, abrimos una terminal (`Ctrl + alt + t`) y nos desplazamos dentro de ella al lugar en el que hemos guardado `tools.py` —digamos

```
mi_usuario@mi_usuario-nombre_equipo:~/Documents/taller_python
```

podemos tener dentro del terminal el siguiente diálogo:

```
miUsuario@miUsuario-miEquipo:~/Documents/tallerPython\$ python3
Python 3.2.3 (default, Oct 19 2012, 20:10:41)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import tools
>>> a = [0,0,-1,5,4,4,7]
>>> tools.sortSr(a)
[-1, 0, 4, 5, 7]
```

Como se ve, hemos cargado el módulo `tools.py` y podemos usar todo lo que este pone a nuestra disposición, en particular el método `sortSr`. Pero si de todo `tools.py` sólo quisiéramos el método `sortSr` y no quisiéramos cargar el resto, entonces podríamos hacer:

```
miUsuario@miUsuario-miEquipo:~/Documents/tallerPython$ python3
Python 3.2.3 (default, Oct 19 2012, 20:10:41)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from tools import sortSr
>>> a = [0,0,-1,5,4,4,7]
>>> sortSr(a)
[-1, 0, 4, 5, 7]
>>> quit()
miUsuario@miUsuario-miEquipo:~/Documents/tallerPython$
```

Como se puede entender, la orden `quit()` nos saca del intérprete de Python hasta la línea de órdenes del sistema operativo.

Si mostramos el contenido de `fibo.py`, por ejemplo con la orden:

```
cat fibo.py
```

veremos que al final tiene el siguiente código

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

Esto nos permite el siguiente uso:

```
miUsuario@miUsuario-miEquipo:~/Documents/tallerPython$ python3 fibo.py 50
1 1 2 3 5 8 13 21 34
miUsuario@miUsuario-miEquipo:~/Documents/tallerPython$
```

Como se comprueba, sin necesidad de interpretar `fibo.py` hemos podido usar una función señalada dentro de él como método `main` y lo hemos hecho con 50 como argumento previamente convertido en entero.

He aquí el código del hola mundo en Python

```
miUsuario@miUsuario-miEquipo:~/Documents/tallerPython$ python3
Python 3.2.3 (default, Oct 19 2012, 20:10:41)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> s = 'Hola mundo.'
>>> str(s)
'Hola mundo.'
>>> quit()
miUsuario@miUsuario-miEquipo:~/Documents/tallerPython$
```

1.4. Generalidades

Un *identificador* de Python es un nombre usado para identificar una variable, función, clase, módulo o cualquier otro objeto. Los identificadores comienzan por una letra de la A a la Z, mayúscula o minúscula, o un guión bajo (`_`) figurando estos caracteres solos o seguido de letras, guiones bajos o dígitos (0 a 9).

Python no permite caracteres de puntuación y otros como `@`, `$` y `%` dentro de los identificadores. Python distingue entre mayúsculas y minúsculas, así `Ejemplo` y `ejemplo` son identificadores distintos ante Python.

Los siguientes son algunos convenios de Python:

1. Los nombres de clases comienzan por letra mayúscula y todos los demás identificadores pueden comenzar por minúscula.
2. Cuando hacemos comenzar un identificador por un único guión bajo indicamos por convenio que el identificador está destinado a ser privado.
3. Si el identificador comienza por dos guiones bajos indicamos que el identificador es fuertemente privado.
4. Si el identificador también acaba en dos guiones bajos, se considera un nombre especial definido en el lenguaje.

En la Figura 1.1 aparece la lista de palabras reservadas en Python; dichas palabras no pueden ser usadas como constantes o variables o como cualquier otro identificador:

En Python no hay llaves para indicar bloques de código para clases, definiciones de funciones o control de flujo. Los bloques de código se denotan por *sangría* o *indentación* de línea rígidamente aplicadas. El número de espacios en la indentación es variable, pero todas las sentencias dentro del bloque deben estar indentadas en la misma cantidad. El siguiente código es un ejemplo de lo dicho:

```
a = True
if a:
    print 'Verdadero'
else:
    print 'Falso'
```


no obstante este otro dará error:

```
a = True
if a:
    print 'Respuesta'
    print 'Verdadero'
else:
    print 'Respuesta'
    print 'Falso'
```

Las declaraciones en Python terminan con una nueva línea. Sin embargo, Python permite el uso del carácter de continuación de línea (\) para indicar que la línea debe continuar. Por ejemplo:

```
totales = item_uno + \
        item_dos + \
        item_tres
```

Las declaraciones contenidas en [], {} o () no necesitan el uso del carácter de continuación de línea. Por ejemplo:

```
dias = [ 'lunes', 'martes', 'miercoles',
        'jueves', 'viernes']
```

Python acepta comillas simples ('), dobles (") y triples ("" o """) para denotar listas de caracteres (string literals), en el bien entendido de que el mismo juego de comillas que se elija para comenzar debe usarse para acabar la lista. Las comillas triples pueden ser usadas para cubrir la cadena a través de varias líneas. Por ejemplo:

```
palabra = 'palabra'
frase = "Ésta es una frase"
parrafo = """ Este es un parrafo. Es
             compuesto por varias lineas"""
```

Para comentar una línea de código Python se hace que comience por el símbolo #, entonces el intérprete de Python “ignora” la línea o el resto de la línea a partir de él. Si se quiere comentar un párrafo, éste puede incluirse entre dos juegos de comillas triples (""").

Una línea que contiene sólo espacios en blanco, posiblemente con un comentario, es conocida como una línea en blanco, y Python totalmente ignora. En una sesión de intérprete interactivo, debe introducir una línea física vacía para terminar una sentencia de varias líneas.

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

Figura 1.1: Palabras reservadas de Python.

El punto y coma (;) permite varias sentencias en una única línea, por ejemplo:

```
import sys; x = 'foo'; sys.stdout.write(x + '\n')
```

En Python una *suite* es un grupo de sentencias individuales que forman un solo bloque de código. Una sentencia *compuesta*, como `if`, `while`, `def` y `class`, es aquella que necesita un encabezamiento y una suite. El encabezamiento comienza la sentencia (con la palabra clave) y termina con un punto y coma (;), la suite le sigue en una o varias líneas. Por ejemplo:

```
if expresion :
    suite
elif expresion :
    suite
else :
    suite
```

Las *variables* son denominadores que sirven para reservar posiciones de memoria para almacenar valores. Basándose en el tipo de dato, el intérprete aloja memoria y decide qué puede ser almacenado en la memoria reservada. Por tanto, asignando diferentes tipos de dato a las variables, puede almacenar enteros, decimales o caracteres en dichas variables.

La asignación de valores a una variable se realiza con el símbolo de igual (=); a la izquierda se pone el nombre de la variable y a la derecha el término que será el valor que tome la variable. Por ejemplo:

```
contador = 100    # asignacion de un entero
miles     = 1000.0 # coma flotante
nombre    = 'Juan' # una cadena de caracteres

print contador
print miles
print nombre
```

1.5. Tipos de datos estándar

Python tiene cinco *tipos de dato* estándar:

- número
- cadena de caracteres (en lo sucesivo *string*)
- lista
- upla o tupla
- diccionario

En lo que sigue creamos dos variables de tipo *número*:

```
var1 = 1
var2 = 2
```

Python soporta cuatro variantes del tipo de dato número:

- `int`, enteros simples
- `long`, enteros largos que pueden ser representados en octal o hexadecimal

- float, valores reales en coma flotante
- complex, números complejos

Aquí tenemos algunos ejemplos de números:

int	long	float	complex
10	51924361L	0.0	3.14j
100	-0x19323L	15.20	45.j
-786	0122L	-21.9	9.322e-36j
080	0xDEFABCECBDAECBFBAEI	32.3+e18	.876j
-0490	535633629843L	-90.	-.6545+0j
-0x260	-052318172735L	-32.54e100	3e+26j
0x69	-4721885298529L	70.2-E12	4.53e-7j

Los string en Python se identifican por ser juegos de caracteres contiguos entre comillas. Aquí tenemos varios ejemplos:

```
str = 'hola ... mundo'

print str          # imprime el string completo
print str[0]       # imprime el primer caracter del string
print str[2:5]     # imprime desde el tercer caracter al 4
print str * 2      # imprime el contenido de str dos veces
print str + 'PRUEBA' # imprime dos string concatenados
```

El string es un tipo de dato estático.

El tipo *lista* es el más versátil de los tipos de datos compuestos. Las listas contienen items separados por comas y encerradas dentro de corchetes ([]).

```
#!/usr/bin/python

list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tinylist = ['john']

print list          # imprime la lista completa
print list[0]       # imprime el primer elemento de la lista
print list[1:3]     # imprime los items segundo y tercero
print list[2:]      # imprime los items a partir del tercero
print tinylist * 2   # imprime la lista dos veces
print list + tinylist # imprime listas concatenadas
```

El tipo de dato *upla* idéntico al de lista solamente que se sustituyen los corchetes por paréntesis (()). Sin demasiada pérdida de generalidad, las uplas puede ser consideradas como listas estáticas, es decir, de sólo lectura.

Los diccionarios de Python son tipos de tablas hash. Funcionan como colecciones (*array*) hash asociativas como las que se encuentran en Perl y consisten en pares de valores clave.

```
dict = {'nombre': 'juan', 'codigo': 6734, 'dept': 'ventas'}
```

```
print dict['nombre']    # imprime el valor para la clave 'nombre'
print dict              # imprime el diccionario completo
print dict.keys()       # imprime todas las claves
print dict.values()     # imprime todos los valores
```

1.6. Operadores básicos de Python

En el ambiente del siguiente diálogo en el intérprete de Python:

```
>>> a, b = 10, 20
```

podemos entender el siguiente cuadro:

Operador	Descripción	Ejemplo
+	Suma - Suma los valores a ambos lados del operador	a + b dará 30
-	Resta - Resta el valor a la derecha del operador del de la izquierda	a - b dará -10
*	Multiplicación - Multiplica los valores a ambos lados del operador	a * b dará 200
/	División - Divide el valor a la izquierda del operador entre el valor a la derecha	b / a dará 2
%	Módulo - Divide el valor a la izquierda del operador entre el valor a la derecha y devuelve el resto	b % a dará 0
**	Exponenciación - Lleva a cabo el cálculo de la exponenciación (potencia) de los operandos	a**b dará 10 elevado a 20
//	División truncada (floor division) - División de los operandos donde el resultado es el cociente en el cual los dígitos tras la coma decimal son suprimidos	9//2 es igual a 4 y 9.0//2.0 es igual a 4.0
==	Comprueba si el valor de los dos operandos es igual o no, en caso afirmativo la condición es true	(a == b) no es true
!=	Comprueba si el valor de los dos operandos es igual o no, si los valores no son iguales la condición es true	(a != b) es true
<>	Comprueba si el valor de los dos operandos es igual o no, si los valores no son iguales la condición es true	(a <> b) es true. Esto es similar al operador !=
>	Comprueba si el valor a la izquierda del operador es mayor que el valor a la derecha del operador, en caso afirmativo la condición es true	(a > b) no es true
<	Comprueba si el valor a la izquierda del operador es menor que el valor a la derecha del operador, en caso afirmativo la condición es true	(a < b) es true
>=	Comprueba si el valor a la izquierda del operador es mayor o igual que el valor a la derecha del operador, en caso afirmativo la condición es true	(a >= b) no es true
<=	Comprueba si el valor a la izquierda del operador es menor que el valor a la derecha del operador, en caso afirmativo la condición es true	(a <= b) es true
=	Operador de asignación simple, asigna los valores a su derecha a lo de su izquierda	c = a + b asignará a + b a c
+=	Operador de asignación suma AND (add AND), suma lo de la derecha a lo de la izquierda y asigna el resultado a lo de la izquierda	c += a es equivalente a c = c + a
-=	Operador de asignación resta AND (subtract AND), resta lo de la derecha a lo de la izquierda y asigna el resultado a lo de la izquierda	c -= a es equivalente a c = c - a
*=	Operador de asignación multiplica AND (multiply AND), multiplica lo de la derecha por lo de la izquierda y asigna el resultado a lo de la izquierda	c *= a es equivalente a c = c * a
/=	Operador de asignación divide AND (divide AND), divide lo de la izquierda por lo de la derecha y asigna el resultado a lo de la izquierda	c /= a es equivalente a c = c / a

<code>%=</code>	Operador de asignación módulo AND (Modulus AND), toma módulo usando los operandos y asigna el resultado al operando de la izquierda	<code>c%= a</code> es equivalente a <code>c = c% a</code>
<code>**=</code>	Operador de asignación exponenciación AND (Exponent AND), lleva a cabo el cálculo de la exponenciación (potencia) sobre los operandos y asigna el resultado al operando de la izquierda	<code>c **= a</code> es equivalente a <code>c = c ** a</code>
<code>//=</code>	Operador de asignación división truncada AND (floor division AND), lleva a cabo la división truncada sobre los operandos y asigna el resultado al operando de la izquierda	<code>c //= a</code> es equivalente a <code>c = c // a</code>
<code>&</code>	Operador binario AND, copia un bit en el resultado si existe en ambos operandos	<code>(a & b)</code> dará 0 que es 0000 0000
<code> </code>	Operador binario OR, copia un bit si existe en alguno de los operandos	<code>(a b)</code> dará 30 que es 0001 1110
<code>^</code>	Operador binario XOR, copia el bit si aparece en un operando pero no en ambos	<code>(a ^ b)</code> dará 30 que es 0001 1110
<code>~</code>	Operador complemento diez-menos-uno ²	<code>~b</code> dará 79 que es 99-21+1 representado como -21
<code><<</code>	Operador binario de desplazamiento a izquierda. El valor del operando de la izquierda es movido a la izquierda el número de bits especificados por el operando de la derecha	<code>a <<2</code> dará 40 que es 101000 transformado de 1010
<code>>></code>	Operador binario de desplazamiento a derecha. El valor de la izquierda del operando es movido a la derecha el número de bits especificados por el operando de la derecha	<code>a >>2</code> dará 2 que es 10 como transformado de 1010 y <code>b >>2</code> dará 5 que es 101 como transformado de 10100
<code>and</code>	Llamado operador lógico AND. La condición es True si, y sólo si, ambos operandos son True	<code>True and True</code> es True
<code>or</code>	Llamado operador lógico OR. La condición es True si, y sólo si, cualquiera de los operandos (o los dos) es True	<code>False or True</code> es True y también lo es <code>True or True</code>
<code>not</code>	Llamado operador lógico NOT. Conmuta el estado lógico de su operando; la condición es True si, y sólo si, el operando es False	<code>not True</code> es False
<code>in</code>	Se evalúa cierta si, y sólo si, el valor de la variable de la izquierda es encontrado en la secuencia que es el operando de la derecha	<code>x in y</code> resulta True sii <code>x</code> es miembro de la secuencia <code>y</code>
<code>not in</code>	Se evalúa cierta si, y sólo si, el valor de la variable de la izquierda no es encontrado en la secuencia que es el operando de la derecha	<code>x not in y</code> , resulta True sii <code>x</code> no es miembro de la secuencia <code>y</code>
<code>is</code>	Se evalúa a True sii las variables a ambos lados del operador señalan al mismo objeto	<code>x is y</code> resulta True sii <code>id(x)</code> es igual a <code>id(y)</code>
<code>is not</code>	Se evalúa a False sii las variables a ambos lados del operador señalan al mismo objeto	<code>x is not y</code> resulta True sii <code>id(x)</code> no es igual a <code>id(y)</code>

Los operadores de Python están sujetos a la precedencia que esquematiza la tabla de la [Figura 1.2](#).

De la tabla de operadores se derivan algunas peculiaridades, por ejemplo el comportamiento de `is` versus `==`. Para entenderlo veamos el siguiente diálogo:

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

²Cfr. [Apéndice A](#)

```

>>> b = a
>>> b is a
True
>>> b == a
True
>>> b = a[:]
>>> b is a
False
>>> b == a
True
>>> [1,2] is [1,2]
False
>>> 2 is 2
True
>>> 1000 is 1000
True
>>> 1000 is 999 + 1
False
>>> "a" is "a"
True
>>> "aa" is "a" * 2
True
>>> a = "a"
>>> "aa" is a*2
False

```

1.7. Resumen y ejemplos sobre sentencias

La sentencia if ...elif...else

Una sentencia else puede ser combinada con una sentencia if. La sentencia else contiene el bloque de código que se ejecuta si la condición de la sentencia if se evalúa como 0 o False. La sentencia else es opcional y si aparece siguiendo a un if, debe aparecer a lo sumo una. La sintaxis es la siguiente:

```

if expresion:
    sentencia(s)
else:

```

Operador	Descripción
**	Exponenciación, elevar a la potencia
~+-	complemento diez-menos-uno, más y menos unarios
*/% //	multiplicación, división, módulo y división truncada
+-	suma y resta
>><<	desplazamiento bit a bit a derecha e izquierda
&	AND bit a bit
^	OR bit a bit exclusivo y regular
<=<>>=	Operadores de comparación
<>==!=	Operadores de igualdad
= %= /= // -= += = &= >>=<<=***=	Operadores de asignación
is is not	Operadores de identidad
in not in	Operadores de pertenencia
not or and	Operadores lógicos

Figura 1.2: Precedencia de operadores en Python.

```
sentencia(s)
```

Pongamos el siguiente ejemplo:

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

var1 = 100
if var1:
    print "1 - la variable no tiene valor nulo"
    print var1
else:
    print "1 - la variable tiene valor nulo"
    print var1

var2 = 0
if var2:
    print "2 - la variable no tiene valor nulo"
    print var2
else:
    print "2 - la variable tiene valor nulo"
    print var2

print "¡Adiós!"
```

Cuando el anterior código es interpretado, el resultado es:

```
equipo:last-of-the-track miUsuario$ python if_ejemplo_00.py
1 - la variable no tiene valor nulo
100
2 - la variable tiene valor nulo
0
¡Adiós!
equipo:last-of-the-track miUsuario$
```

La sentencia elif

La sentencia `elif` permite probar múltiples expresiones por si se evalúan `True` y ejecutar el bloque correspondiente de código en cuanto que alguna se evalúe a `True`. Como en el caso del `else`, la sentencia `elif` es opcional. No obstante, a diferencia del `else`, puede haber un número arbitrario de `elif` siguiendo a un `if`. La sintaxis es la siguiente:

```
if expresion1:
    sentencia(s)
elif expresion2:
    sentencia(s)
elif expresion3:
    setencia(s)
else:
    sentencia(s)
```

El núcleo de Python no provee sentencias `switch` o `case` como otros lenguajes, pero podemos usar sentencias `if...elif...` para simularlas. Por ejemplo:

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

temperatura = int(input("¿Cuál es la temperatura en Fahrenheit? "))
```

```

if temperatura > 90 and temperatura <= 110:
    print("Hace calor fuera")
elif temperatura > 110 :
    print("¡Casi podemos freír un huevo sobre el asfalto!")
elif temperatura < 30:
    print("Hace frío fuera")
else:
    print("Se está bien afuera")

print ("Hecho")

```

La sentencia for

Con la sentencia for podemos iterar una variable sobre los items de cualquier secuencia, tal como una lista o un string. Su sintaxis es:

```

for variable_a_iterar in secuencia:
    sentencia(s)

```

Si la secuencia contiene una expresión lista, ésta se evalúa primero. Entonces, el primer item en la secuencia es asignado a la variable a iterar. Seguidamente, el bloque de sentencia(s) es ejecutado. Cada item en la lista es asignado a la variable a iterar y el bloque de sentencia(s) es ejecutado hasta que la secuencia entera es recorrida. Pongamos el siguiente ejemplo:

```

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

for letra in 'Python':
    print 'Actualmente letra es:', letra

frutas = ['banana', 'manzana', 'mango']
for fruta in frutas:
    print 'Actualmente fruta es:', fruta

print "¡Adiós!"

```

El resultado de interpretar el anterior retazo de código es:

```

equipo:last-of-the-track miUsuario$ python for_ejemplo_01.py
Actualmente letra es: P
Actualmente letra es: y
Actualmente letra es: t
Actualmente letra es: h
Actualmente letra es: o
Actualmente letra es: n
Actualmente fruta es: banana
Actualmente fruta es: manzana
Actualmente fruta es: mango
¡Adiós!
equipo:last-of-the-track miUsuario$

```

Una forma alternativa de iterar en una secuencia es por medio de los índices de la secuencia:

```

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

frutas = ['banana', 'manzana', 'mango']

```



```
for i in xrange(len(frutas)):
    print 'Actualmente fruta es:', frutas[i]

print "¡Adiós!"
```

El resultado de interpretar el anterior retazo de código es:

```
equipo:last-of-the-track miUsuario$ python for_ejemplo_01.py
Actualmente fruta es: banana
Actualmente fruta es: manzana
Actualmente fruta es: mango
¡Adiós!
equipo:last-of-the-track miUsuario$
```

len() provee de la longitud de la lista y xrange() construye una nueva secuencia, a saber, la de los índices de dicha lista en orden creciente.

for en combinación con else

Python soporta el uso de un else en combinación con un bucle:

- Si la sentencia else es usada con un bucle for, se ejecuta cuando el bucle ha concluido la iteración de la lista.
- Si la sentencia else es usada con un bucle while, se ejecuta cuando la condición llega a tener el valor False

Veamos el siguiente ejemplo:

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

for num in xrange(10,20): # para iterar de 10 a 20
    for i in xrange(2,num): # para iterar en los factores de los números
        if num % i == 0:    # para determinar el primer factor
            j=num/i        # para calcular el segundo factor
            print '%d es igual a %d * %d' % (num,i,j)
            break # para ir al siguiente número del primer for
    else:                # parte else de bucle
        print num, 'es un número primo'
```

El resultado de interpretar ese retazo de código es el siguiente:

```
equipo:last-of-the-track miUsuario$ python for_ejemplo_02.py
10 es igual a 2 * 5
11 es un número primo
12 es igual a 2 * 6
13 es un número primo
14 es igual a 2 * 7
15 es igual a 3 * 5
16 es igual a 2 * 8
17 es un número primo
18 es igual a 2 * 9
19 es un número primo
```

Anidamiento de sentencias for

La programación en Python permite el anidamiento de bucles. En el caso de la sentencia for la sintaxis es la siguiente:

```
for variable_a_iterar in secuencia:
    for variable_a_iterar in secuencia:
        sentencia(s)
    sentencias(s) # este bloque puede faltar
```

Como ejemplo pongamos:

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

for i in range(1,10):
    for j in range(0,i):
        print i,
    print "\n"
```

que interpretado tiene el siguiente efecto:

```
equipo:last-of-the-track miUsuario$ python for_ejemplo_03.py
1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
6 6 6 6 6 6
7 7 7 7 7 7 7
8 8 8 8 8 8 8 8
9 9 9 9 9 9 9 9 9

equipo:last-of-the-track miUsuario$
```

Si bien tendríamos análogo resultado con un código más elegante:

```
for i in range(10):
    print(str(i) * i)
```

que interpretado tiene el siguiente efecto:

```
equipo:last-of-the-track miUsuario$ python for_ejemplo_04.py
1
22
333
4444
55555
666666
7777777
88888888
999999999
equipo:last-of-the-track miUsuario$
```

La sentencia while

El bucle `while` en Python ejecuta repetidamente un paquete de sentencias en tanto que una condición dada tiene el valor `True`; cuando deviene de valor `False`, si es el caso, el control de ejecución pasa a la sentencia siguiente en el programa. La sintaxis del bucle `while` es la siguiente:

```
while condicion:
    sentencia(s)
```

Valga de ejemplo el siguiente:

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

contador = 0
while (contador < 9):
    print 'El contador es: ', contador
    contador = contador + 1

print "Adiós!"
```

que al ejecutarse da este resultado:

```
El contador es: 0
El contador es: 1
El contador es: 2
El contador es: 3
El contador es: 4
El contador es: 5
El contador es: 6
El contador es: 7
El contador es: 8
Adiós!
```

Puede ocurrir que el bucle sea infinito. Esto ocurre cuando la condición nunca alcanza el valor `False`. Valga el siguiente ejemplo:

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

var = 1
while var == 1:
    num = raw_input("Introduzca un número: ")
    print "Usted introdujo: ", num

print ";Adiós!"
```

while en combinación con else

Python soporta el uso de un `else` en combinación con un bucle:

- Si la sentencia `else` es usada con un bucle `for`, se ejecuta cuando el bucle ha concluido la iteración de la lista.
- Si la sentencia `else` es usada con un bucle `while`, se ejecuta cuando la condición llega a tener el valor `False`

Como ejemplo el siguiente:

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

contador = 0
while contador < 5:
    print contador, " es menor que 5"
    contador = contador + 1
else:
    print contador, " no es menor que 5"
```

que ofrecer el siguiente diálogo:

```
0 es menor que 5
1 es menor que 5
2 es menor que 5
3 es menor que 5
4 es menor que 5
5 no es menor que 5
```

Una única sentencia

Similar al caso de la sentencia if, si su cláusula while tiene una única sentencia, puede ser colocada en la misma línea de la cabecera while:

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

bandera = 1

while bandera: print ';la bandera proporcionada es realmente verdadera!'

print ";Adiós!"
```

Al interpretar este código cae en un bucle infinito. Necesitará pulsar ctrl + c para parar el proceso.

La sentencia break

La sentencia break en Python interrumpe el bucle en el que se encuentra y reanuda la ejecución en la siguiente sentencia, igual que el break tradicional del que disfruta C. Se puede usar en ambos bucles: while y for.

El uso más común para el break es cuando alguna condición externa es activada requiriendo una salida precipitada del bucle. Veamos el siguiente ejemplo:

```
for letra in 'Python':          # Primer ejemplo
    if letra == 'h':
        break
    print 'Letra actual :', letra

var = 10                         # Segundo ejemplo
while var > 0:
    print 'Valor actual de la variable :', var
    var = var -1
    if var == 5:
        break
print "Hasta la vista!"
```

La sentencia continue

La sentencia continue en Python devuelve el control al principio del bucle while. La sentencia continue rechaza todas las sentencias restantes en la iteración actual del bucle y retrotrae el control a la parte superior del bucle. Se puede usar en ambos bucles: while y for.

```
for letra in 'Python':          # Primer ejemplo
    if letra == 'h':
        continue
    print 'Letra actual :', letra

var = 10                        # Segundo ejemplo
while var > 0:
    print 'Valor actual de la variable :', var
    var = var -1
    if var == 5:
        continue
    print "Hasta la vista!"
```

La sentencia pass

En Python la sentencia pass se usa cuando una sentencia es requerida sintácticamente pero no se desea ejecutar ninguna orden o código. La sentencia pass es la operación null (nula); no sucede nada cuando es ejecutada. La sentencia pass es también útil en lugares donde irá el código, pero todavía no ha sido escrito. Por ejemplo:

```
for letra in 'Python':
    if letter == 'h':
        pass
    print 'This is pass block'
    print 'La letra actual es: ', letra
print "Hasta pronto!"
```

Anidamiento de bucles while

El bucle while, como hemos visto antes con el bucle for, se pueden anidar con la sintaxis obvia que es

```
while expresion:
    while expresion:
        sentencia(s)
    setencia(s)
```

Veamos el siguiente ejemplo:

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

i = 2
while i < 100:
    j = 2
    while j <= i/j:
        if not i%j: break
        j += 1
    if (j > i/j) : print i, " es primo"
    i = i + 1

print ";Adiós!"
```

```
equipo:last-of-the-track miUsuario$ python while_ejemplo_04.py
2 es primo
3 es primo
5 es primo
7 es primo
11 es primo
13 es primo
17 es primo
19 es primo
23 es primo
29 es primo
31 es primo
37 es primo
41 es primo
43 es primo
47 es primo
53 es primo
59 es primo
61 es primo
67 es primo
71 es primo
73 es primo
79 es primo
83 es primo
89 es primo
97 es primo
¡Adiós!
```

Se puede anidar cualquier tipo de bucle dentro de cualquier otro.

1.8. Listas en Python

Python tiene varios tipos de datos *compuestos*, usados para agrupar otros valores. El más versátil es la *lista*, la cual puede ser escrita como una secuencia de valores separados por como (*items* o *entradas*) entre corchetes. No es necesario que las entradas de la lista tengan todos el mismo tipo, importa el orden y puede haber repeticiones.

Generalidades

En el siguiente diálogo vemos como asignar una lista a una variable, presentar el contenido de ésta y extraer por sus posiciones los datos coleccionados en la lista:

```
>>> a = ['toyota', 'honda', 100, 123]; a
['toyota', 'honda', 100, 123]
>>> a[0]; a[1]; a[2]; a[3]
'toyota'
'honda'
100
123
>>> a[-1]; a[-2]; a[-3]; a[-4]
123
100
'honda'
'toyota'
>>> a[3] == a[-1]
```

True

Como podemos observar Python dota a las listas de una estructura con cierta “circularidad”, por ello `a[1]` coincide con `a[-3]` en nuestro ejemplo. Obsérvese también que la primera posición de la lista está indexada con 0.

Podemos seleccionar un intervalo en la lista; en el siguiente diálogo lo hacemos de dos formas:

```
>>> a[1:3]
['honda', 100]
>>> a[1:-1]
['honda', 100]
>>>
```

y podemos manipular las posiciones:

```
>>> a[2] = a[2] - 1; a
['toyota', 'honda', 99, 123]
>>> a[0:2] = ['subaru', 'suzuki']; a
['subaru', 'suzuki', 99, 123]
>>> a[0:2] = ['datsun']; a
['datsun', 99, 123]
>>> a[:0] = ['lexus']; a
['lexus', 'datsun', 99, 123]
>>> a[:0] = a; a
['lexus', 'datsun', 99, 123, 'lexus', 'datsun', 99, 123]
>>> a[1:1] = ['nissan']; a
['lexus', 'nissan', 'datsun', 99, 123, 'lexus', 'datsun', 99, 123]
>>> a[:] = []; a
[]
```

Ahora destacamos cómo poner una nueva entrada a la cabeza de la lista, en su cola y en una posición intermedia (digamos la 1):

```
>>> a[:0] = ['datsun']; a
['datsun']
>>> a.append('toyota'); a
['datsun', 'toyota']
>>> a[1:1] = ['lexus']; a
['datsun', 'lexus', 'toyota']
```

Yuxtaponemos dos listas con el operador `+`:

```
>>> a + ['mitsubishi', 'isuzu']
['datsun', 'lexus', 'toyota', 'mitsubishi', 'isuzu']
```

La función `len` aplicada a una lista proporciona la longitud de la misma:

```
>>> q = ['mazda', 'daihatsu']; q
['mazda', 'daihatsu']
>>> b = ['lexus', q, 'subaru']; b
['lexus', ['mazda', 'daihatsu'], 'subaru']
>>> len(b)
3
```

La Función range

La función `range` genera listas de números naturales:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(5,15)
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

aunque no tienen por qué ser consecutivos ni aparecer en orden creciente:

```
>>> range(5,15,2)
[5, 7, 9, 11, 13]
>>> range(100,15,-10)
[100, 90, 80, 70, 60, 50, 40, 30, 20]
```

Si incluimos en nuestro código una expresión como `range(0,10000000)` para que un contador la recorra, creará —antes de proseguir y de forma “ansiosa”— una lista de números enteros de longitud 10000000, lo cual conlleva la necesidad de almacenarla. En casos como éste, merece la pena proceder de forma perezosa usando la alternativa `xrange(0,10000000)`. Esta orden crea el índice bajo demanda y no reserva el espacio para almacenar la lista, porque realmente no la crea en un acto, sino que lo va haciendo poco a poco. El problema es el rango de variabilidad que puede soportar:

```
xrange(2**32-1, 2**32+1) # OverflowError: cannot convert to int
range(2**32-1, 2**32+1)   # OK --> [4294967295L, 4294967296L]
```

No obstante, en cuanto al tiempo sí saldremos bien parados:

```
$ python -m timeit 'for i in range(1000000):' ' pass'
10 loops, best of 3: 90.5 msec per loop
$ python -m timeit 'for i in xrange(1000000):' ' pass'
10 loops, best of 3: 51.1 msec per loop
```

No obstante en Python 3, `range` está implementado mediante el `xrange()` de Python 2. Si se requiriese generar realmente la lista de, digamos, el 1 al 100 habríamos de hacer:

```
list(range(1,100))
```

Más sobre Modificación de Listas

El método `extend()` permite también yuxtaponer listas; en efecto, `lst.extend(mst)` genera la lista `lst` a la que se le ha yuxtapuesto la lista `mst`:

```
>>> a = ['toyota','suzuki','honda','subaru']; a
['toyota', 'suzuki', 'honda', 'subaru']
>>> a.extend(['lexus','mazda']); a
['toyota', 'suzuki', 'honda', 'subaru', 'lexus', 'mazda']
```

Podemos insertar entradas en posiciones con el método `insert()`; en efecto, `lst.insert(n,x)` genera la lista `mst` que coincide con `lst` salvo que en la posición `n` ahora está `x`:

```
>>> a.insert(2,'nissan'); a
['toyota', 'suzuki', 'nissan', 'honda', 'subaru', 'lexus', 'mazda']
```

El método `remove()` puede ser utilizado para eliminar entradas; en efecto, `lst.remove(n)` elimina de la lista `lst` la primera entrada de `n` y produce un `ValueError` si `n` no tiene ocurrencias en `lst`:


```
>>> a.insert(2,'nissan'); a
['toyota', 'suzuki', 'nissan', 'nissan', 'honda', 'subaru', 'lexus', 'mazda']
>>> a.remove('nissan'); a
['toyota', 'suzuki', 'nissan', 'honda', 'subaru', 'lexus', 'mazda']
>>> a.insert(2,'nissan'); a
['toyota', 'suzuki', 'nissan', 'nissan', 'honda', 'subaru', 'lexus', 'mazda']
>>> a.index('nissan')
2
```

El método `count()` cuenta el número de ocurrencias de una entrada en una lista y la función `del` borra entradas por su posición:

```
>>> a.count('nissan')
2
>>> a; del(a[1]); a
['toyota', 'suzuki', 'nissan', 'nissan', 'honda', 'subaru', 'lexus', 'mazda']
['toyota', 'nissan', 'nissan', 'honda', 'subaru', 'lexus', 'mazda']
```

Ordenación

El método `sort()` ordena la lista a la que se aplica; en efecto, `lst.sort()` sustituye el contenido de `lst` por su ordenación:

```
>>> a = ['toyota', 'suzuki', 'honda', 'subaru', 'isuzu', 'daihatsu']; a
['toyota', 'suzuki', 'honda', 'subaru', 'isuzu', 'daihatsu']
>>> z = a; z
['toyota', 'suzuki', 'honda', 'subaru', 'isuzu', 'daihatsu']
>>> z.sort(); z; a
['daihatsu', 'honda', 'isuzu', 'subaru', 'suzuki', 'toyota']
['daihatsu', 'honda', 'isuzu', 'subaru', 'suzuki', 'toyota']
```

pero obsérvese que si `lst` es copia de otra variable, digamos `mst`, también esta otra variable queda reasignada a la ordenación. Sin embargo, la función `sorted` genera de la lista `lst` la lista `mst` que es la ordenación de `lst`; pero queda intacta `lst` o cualquier copia suya.

```
>>> a = ['toyota', 'suzuki', 'honda', 'subaru', 'isuzu', 'daihatsu']; a
['toyota', 'suzuki', 'honda', 'subaru', 'isuzu', 'daihatsu']
>>> z = a; z
['toyota', 'suzuki', 'honda', 'subaru', 'isuzu', 'daihatsu']
>>> x = sorted(z); x; a
['daihatsu', 'honda', 'isuzu', 'subaru', 'suzuki', 'toyota']
['toyota', 'suzuki', 'honda', 'subaru', 'isuzu', 'daihatsu']
```

Se puede revertir el orden de una lista por medio del método `reverse()`:

```
>>> x.reverse(); x
['toyota', 'suzuki', 'subaru', 'isuzu', 'honda', 'daihatsu']
```

aunque también se podía haber hecho lo siguiente:

```
>>> a = ['toyota', 'suzuki', 'honda', 'subaru', 'isuzu', 'daihatsu']
>>> z = a; z
['toyota', 'suzuki', 'honda', 'subaru', 'isuzu', 'daihatsu']
>>> z.sort(reverse=True); z; a
```

```
['toyota', 'suzuki', 'subaru', 'isuzu', 'honda', 'daihatsu']  
['toyota', 'suzuki', 'subaru', 'isuzu', 'honda', 'daihatsu']
```

También se puede extraer de una lista no vacía su último elemento mediante el método `pop()`:

```
>>> x.pop(); x  
'daihatsu'  
['toyota', 'suzuki', 'subaru', 'isuzu', 'honda']
```

Orden entre Listas

Las listas están ordenadas por el orden lexicográfico. En efecto:

```
>>> a, b = [2,1], [1,2,3]  
>>> a <= b  
False  
>>> b <= a  
True
```

y, por supuesto,

```
>>> e = []  
>>> l1 = [1,2,3]  
>>> e <= l1  
True  
>>> l1 <= e  
False
```

1.9. Definición de Funciones

Se puede definir una *función* para proveer de la suficiente funcionalidad a nuestros programas; las reglas son simples:

1. Los bloques de función comienzan con la palabra reservada `def` seguida por el nombre de la función y éste por paréntesis y dos puntos `()`:
2. Dentro del paréntesis deberían ser colocados los parámetros o argumentos. Dentro del paréntesis también se podrían definir parámetros.
3. La primera sentencia de una función puede ser el texto comentado que contiene la documentación de la función (`docstring`). Este bloque debe estar indentado.
4. El bloque de código debe estar en uno de los renglones que siguen al primero (el cual acaba en `:`) y debe guardar la pauta de indentación.
5. La salida de una función, de existir, es una expresión precedida de la palabra reservada `return`, igualmente indentada. Un `return` al que no le sigue expresión alguna equivale a `return None`. De haberlo, no se exige que haya un único `return`, pero es conveniente esforzarse en que sea así.

Lo anterior se resume con lo siguiente:

```
def functionname( parameters ):  
    "docstring_de_la_función"  
    bloque_de_código  
    return [expresión]
```

y como ejemplo tenemos:

```
def imprimeme(str):
    "Esto imprime el string que
    se pasa a la función"
    print str
    return
```

o este otro:

```
def letraDelDNI(n):
    """
    esta función calcula la letra del DNI español,
    cuando se le proporciona el número de dicho
    documento
    """
    permutacion = "TRWAGMYFPDXBNJZSQVHLCKE"
    return permutacion[n % 23]
```

y en el siguiente definimos una función que escribe los términos de la *sucesión de Fibonacci* que no superan un número natural dado.

```
>>> def fib(n):
...     """
...     Escribe la sucesión de Fibonacci hasta
...     n en una lista.
...     """
...     a, b, lst = 0, 1, [0]
...     while b < n:
...         lst.append(b)
...         a, b = b, a + b
...     return lst

>>> fib(2000)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597]
```

Hay herramientas que usan las docstrings para producir automáticamente documentación en línea o imprimible, o para permitirle al usuario que navegue el código en forma interactiva; es una buena práctica incluir docstrings en el código que uno escribe, por lo que se debe intentar hacer un hábito de esto.

La ejecución de una función introduce una nueva tabla de símbolos usada para las variables locales de la función. Más precisamente, todas las asignaciones de variables en la función almacenan el valor en la tabla de símbolos local; asimismo la referencia a variables primero mira la tabla de símbolos local, luego en la tabla de símbolos local de las funciones externas, luego la tabla de símbolos global, y finalmente la tabla de nombres predefinidos. Así, no se les puede asignar directamente un valor a las variables globales dentro de una función (a menos que se las nombre en la sentencia global), aunque sí pueden ser referenciadas.

Los parámetros reales (argumentos) de una función se introducen en la tabla de símbolos local de la función llamada cuando ésta es llamada; así, los argumentos son pasados por valor (donde el valor es siempre una referencia a un objeto, no el valor del objeto). Cuando una función llama a otra función, una nueva tabla de símbolos local es creada para esa llamada.

La definición de una función introduce el nombre de la función en la tabla de símbolos actual. El valor del nombre de la función tiene un tipo que es reconocido por el interprete como una función definida por el usuario. Este valor puede ser asignado a otro nombre que luego puede ser usado como una función. Esto sirve como un mecanismo general para renombrar:

```
>>> fib
<function fib at 0x113329b90>
>>> f = fib
>>> f(100)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Este ejemplo demuestra algunas características más de Python:

- La sentencia `return`, como se ha dicho, devuelve un valor en una función. `return` sin una expresión como argumento retorna `None`. Si se alcanza el final de un procedimiento, también se retorna `None`.
- La sentencia `lst.append(b)` llama a un método del objeto lista `lst`. Un método es una función que “pertenece” a un objeto y se nombra `obj.methodname`, donde `obj` es algún objeto (puede ser una expresión), y `methodname` es el nombre del método que está definido por el tipo del objeto. Distintos tipos definen distintos métodos. Métodos de diferentes tipos pueden tener el mismo nombre sin causar ambigüedad. (Es posible definir tipos de objetos propios, y métodos, usando clases, como se discutirá más adelante en el tutorial). El método `append()` mostrado en el ejemplo está definido para objetos lista; añade un nuevo elemento al final de la lista. En este ejemplo es equivalente a `lst = lst + [b]`, pero más eficiente.

Consideremos el siguiente código:

```
>>> def fbn(n):
...     a, b = 0, 1
...     while b < n:
...         print b,
...         a, b = b, a + b
```

se tiene el siguiente diálogo:

```
>>> fbn(0)
>>> print fbn(0)
None
```

Se puede objetar que `fbn` no es una función, sino un procedimiento. En Python, como en C, los procedimientos son sólo funciones que no retornan un valor. De hecho, técnicamente hablando, los procedimientos sí retornan un valor, aunque uno aburrido. Este valor es llamado `None` (es un nombre predefinido). El intérprete por lo general no escribe el valor `None` si va a ser el único valor escrito. Si realmente se quiere, se puede verlo usando `print`, como se revela más arriba.

1.10. Transparencia Referencial

Se ha difundido la opinión de que Python es un lenguaje funcional. Sin embargo, Python no cuenta entre sus propiedades con la que se conoce como **transparencia referencial**, que se puede entender como sigue: el resultado de evaluar una expresión compuesta depende únicamente del resultado de evaluar las subexpresiones que la componen y de nada más; no depende de la historia del programa en ejecución ni del orden de evaluación de las subexpresiones que la componen. Para ver un ejemplo, consideremos el fichero `transparencia.py` que tiene el siguiente contenido:

```
counter = 0

def incremental(n):
    global counter
    i = 0
    while i <= n:
```

```
        counter += 1
        i += 1
    print counter
```

```
incremental(5)
incremental(5)
```

que nos ofrece el siguiente diálogo:

```
equipo:last-of-the-track miUsuario$ python transparencia.py
6
12
equipo:last-of-the-track miUsuario$
```

Donde vemos que al interpretar la “función” `incremental`, la primera vez se evalúa como 6 y seguidamente se vuelve a evaluar pero ahora como 12. Es decir, Python permite un modo de programar que no respeta la transparencia referencial y, aunque se pueda programar conservándola, lo cierto es que se puede programar violándola.

Esto evidencia que es posible programar en Python usando “funciones” que más bien deberían ser entendidas como “subrutinas”. No obstante, procediendo con esmero también es posible programar con funciones de carácter más próximo al matemático, a las que podríamos llamar “funciones puras”. Tengamos en cuenta que la **transparencia referencial** es una característica esencial de la programación con *funciones puras*, que es la única posible en los lenguajes funcionales puros.

En definitiva, en Python es posible hacer una programación muy parecida a la funcional, aunque también es fácil encontrar programas que violan este estilo. Python no es un lenguaje funcional, pero en él se puede simular una suerte de programación funcional.

1.11. Funciones Anónimas con Lambda

Por demanda popular, algunas características comúnmente encontradas en lenguajes de programación funcionales como Lisp fueron añadidas a Python. Con la palabra reservada `lambda` se pueden crear pequeñas funciones anónimas. Esta es una función que retorna la suma de sus dos argumentos:

```
lambda a, b: a+b
```

Las formas con `lambda` pueden ser usadas en cualquier lugar que se requieran funciones. Semánticamente, son sólo *azúcar sintáctico*³ para la definición de funciones. Cómo en la definición de funciones anidadas, las formas con `lambda` pueden referenciar variables del ámbito en el que son contenidas:

```
>>> def hacer_incrementador(n):
...     return lambda x: x + n
...
>>> f = hacer_incrementador(42)
>>> f(0)
42
>>> f(1)
43
```

³Azúcar sintáctico es un término acuñado por *Peter J. Landin* para referirse a los añadidos a la sintaxis de un lenguaje de programación que no afectan a su funcionalidad, pero que facilitan expresar algunas construcciones de una forma más clara o concisa, o en un estilo alternativo. Por ejemplo, en los lenguajes de programación imperativos los bucles `for` se pueden sustituir por bucles `while`, y éstos a su vez por `GOTOs`.

1.12. Recursividad en Python

“La programación mediante definiciones recursivas es ineficiente” es un enunciado falso por inconcreto. Es tan cierto que hay lenguajes ineficientes en extremo para la recursividad como que hay lenguajes muy eficientes con ella; más aún, hay lenguajes en los que no se puede programar si no es en términos recursivos (Haskell). Es posible en Python definir funciones usando la recursividad y, por desgracia, dicho lenguaje no es de los eficientes con ella.

Aquí y en lo sucesivo, cuando deseemos usar la función signo `sign`, debemos importarla previamente de la librería `numpy`. El código a incluir sería el siguiente:

```
from numpy import sign
```

La idea es bien conocida, es la tentación del que no conoce cómo hacer pero sí sabe reducir el problema global al problema en subcasos suyos. Consideremos la siguiente definición:

```
def f(n):
    if n <= 1: return 1
    elif n % 2 == 0: return 2 * f(n/2)
    else: return 3 * f((n-1)/2)
```

He aquí una famosa forma de multiplicar, la conocida como *Método del Campesino Ruso*, en formato no recursivo:

```
def prodCR(m,n):
    """
    multiplicación por el método del
    Campesino Ruso.
    """
    sm, sn = signo(m), signo(n)
    sp = sm * sn
    m, n = int(m) * sm, int(n) * sn
    s = 0
    while n:
        if n % 2:
            s += m
        n = n // 2
        m *= 2
    return s * sp
```

y he aquí una implementación recursiva de la misma idea:

```
def prodRCR(m,n):
    """
    multiplicación por el método del
    Campesino Ruso, explicado recursivamente
    """
    sm, sn = signo(m), signo(n)
    s = sm * sn
    m, n = int(m) * sm, int(n) * sn
    def r(m,n):
        if n == 1:
            return m
        elif s == 0:
            return 0
        elif n % 2:
            return r(m * 2, (n - 1) // 2) + m
        else:
            return r(m * 2, n // 2)
    return s * r(m,n)
```

1.13. Herramientas de Programación Funcional

Hay tres funciones integradas que son muy útiles cuando se usan con listas: `filter()`, `map()`, y `reduce()`. `filter(funcion, secuencia)` devuelve una secuencia con aquellos ítems de la secuencia para los cuales `funcion(item)` es verdadero. Si `secuencia` es un `string` o `tuple`, el resultado será del mismo tipo; de otra manera, siempre será `list`. Por ejemplo, para calcular unos números primos:

```
>>> def f(x): return x % 2 != 0 and x % 3 != 0
>>> filter(f, range(2, 25))
[5, 7, 11, 13, 17, 19, 23]
```

`map(funcion, secuencia)` llama a `funcion(item)` por cada uno de los ítems de la secuencia y devuelve una lista de los valores retornados. Por ejemplo, para calcular unos cubos:

```
>>> def cubo(x): return x^3
>>> map(cubo, range(1, 11))
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

Se puede pasar más de una secuencia; la función debe entonces tener tantos argumentos como secuencias haya y es llamada con el ítem correspondiente de cada secuencia (o `None` si alguna secuencia es más corta que otra). Por ejemplo:

```
>>> sec = range(8)
>>> def add(x, y): return x+y
>>> map(add, sec, sec)
[0, 2, 4, 6, 8, 10, 12, 14]
```

`reduce(funcion, secuencia)` devuelve un único valor que se construye llamando a la función binaria `funcion` con los primeros dos ítems de la secuencia, entonces con el resultado y el siguiente ítem, y así sucesivamente. Por ejemplo, para calcular la suma de los números de 1 a 10:

```
>>> def sumar(x,y): return x+y
>>> reduce(sumar, range(1, 11))
55
```

Si sólo hay un ítem en la secuencia, se devuelve su valor; si la secuencia está vacía, se lanza una excepción.

Puede pasarse un tercer argumento para indicar el valor inicial. En este caso el valor inicial se devuelve para una secuencia vacía, y la función se aplica primero al valor inicial y al primer ítem de la secuencia, entonces al resultado y al siguiente ítem, y así sucesivamente. Por ejemplo:

```
>>> def sum(sec):
...     def sumar(x,y): return x+y
...     return reduce(sumar, sec, 0)
...
>>> sum(range(1, 11))
55
>>> sum([])
0
```

No use la definición de `sum()` dada en este ejemplo ya que la sumatoria es una necesidad tan común que se provee una función integrada `sum(secuencia)` que funciona exactamente así.

1.14. Listas por Compresión

Las listas por comprensión proveen una forma concisa de crear listas sin tener que recurrir al uso de `map()`, `filter()` y/o `lambda`. La definición resultante de la lista a menudo tiende a ser más clara que las listas formadas usando esas construcciones. Cada lista por comprensión consta de una expresión seguida por una cláusula `for`, luego cero o más cláusulas `for` o `if`. El resultado será una lista que resulta de evaluar la expresión en el contexto de las cláusulas `for` e `if` que sigan. Si la expresión evalúa a una tupla, debe encerrarse entre paréntesis. Considérese el siguiente diálogo:

```
>>> frutafresca = ['banana', 'mora de Logan', 'maracuya']
>>> [arma.strip() for arma in frutafresca]
['banana', 'mora de Logan', 'maracuya']
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [3*x for x in vec if x < 2]
[]
>>> [[x, x**2] for x in vec]
[[2, 4], [4, 16], [6, 36]]
>>> [x, x**2 for x in vec] # error - se requieren paréntesis para tuplas
File "<stdin>", line 1, in ?
[x, x**2 for x in vec]
      ^
SyntaxError: invalid syntax
>>> [(x, x**2) for x in vec]
[(2, 4), (4, 16), (6, 36)]
>>> vec1 = [2, 4, 6]
>>> vec2 = [4, 3, -9]
>>> [x*y for x in vec1 for y in vec2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
>>> [x+y for x in vec1 for y in vec2]
[6, 5, -7, 8, 7, -5, 10, 9, -3]
>>> [vec1[i]*vec2[i] for i in range(len(vec1))]
[8, 12, -54]
```

Las listas por comprensión son mucho más flexibles que `map()` y pueden aplicarse a expresiones complejas y funciones anidadas:

```
>>> [str(round(355/113.0, i)) for i in range(1,6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

1.15. Listas por Compresión Anidadas

Las listas por comprensión pueden anidarse. Son una herramienta poderosa pero, como toda herramienta poderosa, deben usarse con cuidado, o ni siquiera usarse.

Considérese el siguiente ejemplo de una matriz de 3x3 como una lista que contiene tres listas, una por fila:

```
>>> mat = [
...     [1, 2, 3],
...     [4, 5, 6],
...     [7, 8, 9]
... ]
```


Ahora, si se quisiese intercambiar filas y columnas, se podría usar una lista por comprensión:

```
>>> print [[fila[i] for fila in mat] for i in [0, 1, 2]]
[[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

Se debe tener cuidado especial para la lista por comprensión anidada. Para evitar aprensión cuando se anidan lista por comprensión, leer de derecha a izquierda.

Una versión más detallada de este retazo de código muestra el flujo de manera explícita:

```
>>> mat = [[1,2,3],[4,5,6],[7,8,9]]
>>> for i in range(3):
...     for fila in mat:
...         print fila[i],
...     print
...
1 4 7
2 5 8
3 6 9
>>>
```

Pero en el mundo real, se debería preferir funciones predefinidas a declaraciones con flujo complejo. La función `zip()` haría un buen trabajo para este caso de uso:

```
>>> zip(*mat)
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
```

donde el valor de `*` es el desempaquetar la lista según explica lo siguiente:

```
>>> zip([1,2,3],[4,5,6],[7,8,9])
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
```

1.16. Tuplas

Las listas y cadenas tienen propiedades en común, como el indexado y las operaciones de seccionado. Estos son dos ejemplos de datos de tipo secuencia. Como Python es un lenguaje en evolución, pueden agregarse otros datos de tipo secuencia. Existe otro dato de tipo secuencia estándar: la tupla.

Una tupla consiste en un número de valores separados por comas, por ejemplo:

```
>>> t = 12345, 54321, 'hola!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hola!')
>>> # Las tuplas pueden anidarse:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hola!'), (1, 2, 3, 4, 5))
```

Como se puede ver, en la salida las tuplas siempre se encierran entre paréntesis, para que las tuplas anidadas puedan interpretarse correctamente; pueden introducirse con o sin paréntesis, aunque a menudo los paréntesis son necesarios de todas formas (si la tupla es parte de una expresión más grande).

Las tuplas tienen muchos usos. Por ejemplo: pares ordenados (x, y), registros de empleados de una base de datos, etc. Las tuplas, al igual que las cadenas, son inmutables: no es posible asignar a los ítems individuales de una tupla (aunque puedes simular bastante ese efecto mediante seccionado y concatenación). También es posible crear tuplas que contengan objetos mutables como listas.

Un problema particular es la construcción de tuplas que contengan 0 ó 1 ítem: la sintaxis presenta algunas peculiaridades para estos casos. Las tuplas vacías se construyen mediante un par de paréntesis vacío; una tupla con un ítem se construye poniendo una coma a continuación del valor, no basta con encerrar un único valor entre paréntesis. Por ejemplo:

```
>>> vacia = ()
>>> singleton = 'hola',
>>> len(vacia)
0
>>> len(singleton)
1
>>> singleton
('hola')
```

La declaración `t = 12345, 54321, 'hola!'` es un ejemplo de empaquetado de tuplas: los valores 12345, 54321 y 'hola!' se empaquetan juntos en una tupla. La operación inversa también es posible:

```
>>> t = 12345, 54321, 'hola!'
>>> t
(12345, 54321, 'hola!')
>>> x,y,z = t
>>> x
12345
>>> y
54321
>>> z
'hola!'
```

Esto se llama, apropiadamente, desempaqueado de secuencias. El desempaqueado de secuencias requiere que la lista de variables a la izquierda tenga el mismo número de elementos que el tamaño de la secuencia. Nótese que la asignación múltiple es en realidad sólo una combinación de empaquetado de tuplas y desempaqueado de secuencias.

Hay una pequeña asimetría aquí: empaquetando múltiples valores siempre crea una tupla, y el desempaqueado funciona con cualquier secuencia.

1.17. Conjuntos

Python también incluye un tipo de dato para conjuntos. Un conjunto es una colección no ordenada y sin elementos repetidos. Los usos básicos de éstos incluyen verificación de pertenencia y eliminación de entradas duplicadas. Los conjuntos también soportan operaciones matemáticas como la unión, intersección, diferencia, y diferencia simétrica.

Valga esta pequeña demostración:

```
>>> canasta = ['manzana', 'naranja', 'manzana', 'pera', 'naranja', 'banana']
>>> fruta = set(canasta)           # crea un conjunto sin repetidos
>>> fruta
set(['pera', 'manzana', 'banana', 'naranja'])
```

```

>>> 'naranja' in fruta          # verificación de pertenencia rápida
True
>>> 'tornillo' in fruta
False
>>> # veamos las operaciones para las letras únicas de dos palabras
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                          # letras únicas en a
set(['a', 'r', 'b', 'c', 'd'])
>>> a - b                      # letras en a pero no en b
set(['r', 'b', 'd'])
>>> a | b                      # letras en a o en b
set(['a', 'c', 'b', 'd', 'm', 'l', 'r', 'z'])
>>> a & b                      # letras en a y en b
set(['a', 'c'])
>>> a ^ b                      # letras en a o b pero no en ambos
set(['b', 'd', 'm', 'l', 'r', 'z'])
>>> a^b == (a | b) - (a & b)
True
>>> a^b == (a | b) - (a & b)
True

```

1.18. Diccionarios

Ya nos hemos familiarizado con las listas en capítulos anteriores. En el presente de nuestro curso de Python vamos a presentar los diccionarios, sus operadores y sus métodos. Son casi inconcebibles los programas o scripts de Python sin el uso de listas y/o diccionarios. Al igual que las listas, los diccionarios se pueden cambiar fácilmente, pueden ser acortados y aumentados *ad libitum* en tiempo de ejecución; se encogen y crecen sin la necesidad de hacer copias. Los diccionarios pueden estar contenidos en las listas, y viceversa. Pero ¿cuál es la diferencia entre las listas y los diccionarios? Las listas son conjuntos ordenados de objetos, mientras que los diccionarios son conjuntos no ordenados. La principal diferencia es que a los *items* en los diccionarios se accede a través de claves y no a través de su posición. Un diccionario es una matriz asociativa (también conocido como *hashes*). Cualquier clave del diccionario está asociada con (o asigna) un valor. Los valores de un diccionario pueden ser cualquier tipo de datos Python. Así que los diccionarios son parejas clave-valor en desorden.

Los diccionario no son compatibles con las operaciones de los tipos de datos secuenciados como cadenas, tuplas y listas. Los diccionarios pertenecen al tipo incorporado `dict`. ¡Son el único representante de este tipo!

Ejemplos

Nuestro primer ejemplo es el diccionario más simple:

```

>>> vacio = {}
>>> vacio
{}

```

Inmediatamente podemos hacer que no lo sea:

```

vacio['toyota'] = 'Japon'

```

y esa es una forma de añadir parejas al diccionario. Un ejemplo no trivial podría ser este:

```
>>> coches = {'Toyota' : 'Japon', 'Honda' : 'Japon', 'Volkswagen' : 'Alemania'}
>>> coches
{'Honda': 'Japon', 'Toyota': 'Japon', 'Volkswagen': 'Alemania'}
>>> coches['Toyota'] = 'Made in Japon'
>>> coches
{'Honda': 'Japon', 'Toyota': 'Made in Japon', 'Volkswagen': 'Alemania'}
>>>
```

Salvo que el diccionario sea vacío, está constituido por parejas clave/valor con la siguiente sintaxis: clave : valor; por ejemplo, en el diccionario coches hay tres parejas, una de las cuales es 'Honda' : 'Japon'. Nuestro siguiente ejemplo es un script con un glosario simple Inglés-Alemán y otro Alemán-Francés:

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

```
en_de = {"red" : "rot", "green" : "grün", "blue" : "blau", "yellow":"gelb"}
print en_de
print en_de["red"]
de_fr = {"rot" : "rouge", "grün" : "vert", "blau" : "bleu", "gelb":"jaune"}
print "The French word for red is: " + de_fr[en_de["red"]]
```

Podemos utilizar tipos arbitrarios como valores en un diccionario, pero hay una restricción para las claves. Sólo los tipos de datos inmutables pueden ser utilizados como claves, es decir, no se puede usar como claves listas o diccionarios. Si utiliza un tipo de dato mutable como clave, se obtiene un mensaje de error:

```
>>> dic = { [1,2,3]: "abc" }
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: list objects are unhashable
```

Se pueden usar tuplas como claves, según se ve en el siguiente ejemplo:

```
>>> dic = { (1,2,3): "abc", 3.1415: "abc" }
>>> dic
{(1, 2, 3): 'abc'}
```

Nuestro diccionario sobre lenguas naturales algo mejorado:

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

```
en_de = {"red" : "rot", "green" : "grün", "blue" : "blau", "yellow":"gelb"}
de_fr = {"rot" : "rouge", "grün" : "vert", "blau" : "bleu", "gelb":"jaune"}
```

```
dictionaries = {"en_de" : en_de, "de_fr" : de_fr }
print dictionaries["de_fr"]["blau"]
```

que ofrece el siguiente diálogo:

```
equipo:last-of-the-track miUsuario$ python diccionario_natural.py
bleu
equipo:last-of-the-track miUsuario$
```

Operadores para diccionarios

En la siguiente tabla representamos los operadores fundamentales para diccionarios:

Operador	Descripción
len(d)	devuelve el número de entradas almacenadas, es decir, el número de pares (clave, valor).
del d[k]	elimina la clave k junto con su valor
k in d	True, if la clave k figura en el diccionario d
k not in d	True, if la clave k no figura en el diccionario d

Que funciona como sigue:

```
>>> coches
{'Honda': 'Japon', 'Toyota': 'Made in Japon', 'Volkswagen': 'Aleman'}
>>> len(coches)
3
>>> del coches['Honda']
>>> coches
{'Toyota': 'Made in Japon', 'Volkswagen': 'Aleman'}
>>> 'Honda' in coches
False
>>> 'Honda' not in coches
True
>>>
```

Métodos Importantes

Copiar diccionarios

Un diccionario puede ser copiado con el método `copy()`:

```
>>> coches
{'Volvo': 'Suecia', 'Toyota': 'Made in Japon', 'Volkswagen': 'Aleman'}
>>> marcasC = coches.copy()
>>> marcasC
{'Volkswagen': 'Aleman', 'Toyota': 'Made in Japon', 'Volvo': 'Suecia'}
>>> marcasC['Toyota'] = 'Made in Japon'
>>> marcasC['Lada'] = 'Rusia'
>>> marcasC
{'Lada': 'Rusia', 'Volkswagen': 'Aleman', 'Toyota': 'Made in Japon', 'Volvo': 'Suecia'}
>>> coches
{'Volvo': 'Suecia', 'Toyota': 'Made in Japon', 'Volkswagen': 'Aleman'}
>>>
```

Se puede pensar en copiar un diccionario usando `=`, pero esto no hace sino establecer un sinónimo del nombre del propio diccionario:

```
>>> c = coches
>>> coches
{'Volvo': 'Suecia', 'Toyota': 'Made in Japon', 'Volkswagen': 'Aleman'}
>>> c
{'Volvo': 'Suecia', 'Toyota': 'Made in Japon', 'Volkswagen': 'Aleman'}
>>> coches['Fiat'] = 'Italia'
>>> coches
{'Fiat': 'Italia', 'Volvo': 'Suecia', 'Toyota': 'Made in Japon', 'Volkswagen': 'Aleman'}
>>> c
{'Fiat': 'Italia', 'Volvo': 'Suecia', 'Toyota': 'Made in Japon', 'Volkswagen': 'Aleman'}
>>> del c['Fiat']
>>> c
{'Volvo': 'Suecia', 'Toyota': 'Made in Japon', 'Volkswagen': 'Aleman'}
>>> coches
{'Volvo': 'Suecia', 'Toyota': 'Made in Japon', 'Volkswagen': 'Aleman'}
>>> c.clear()
>>> c
{}
>>> coches
{}
>>>
```

Si en el ejemplo eliminamos el designador `c`, permanece `coches` hasta que no lo borremos a él:

```
>>> del c
>>> c
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'c' is not defined
>>> coches
{'Volvo': 'Suecia', 'Toyota': 'Made in Japon', 'Volkswagen': 'Aleman'}
```

Borrar diccionarios

Como hemos visto, el contenido de un diccionario puede ser borrado con el método `clear()`. El diccionario no se elimina, solamente es vaciado:

```
>>> coches = {'Volkswagen': 'Aleman', 'Toyota': 'Made in Japon', 'Volvo': 'Suecia'}
>>> coches
{'Volvo': 'Suecia', 'Toyota': 'Made in Japon', 'Volkswagen': 'Aleman'}
>>> coches.clear()
>>> coches
{}
```

Mezclar diccionarios

Análogamente al caso de la concatenación de listas, podemos mezclar los diccionarios con el método `update()`. La novedad es que el método `update()` mezcla las etiquetas y los valores de un diccionario con otro, actualizandolos valores con la misma clave y añadiendo las parejas clave/valor que no estuviese en el primero y sí en el segundo:

```
>>> coches
{'Volkswagen': 'Aleman', 'Toyota': 'Made in Japon', 'Volvo': 'Suecia'}
>>> coches01
{'Fiat': 'Italia', 'Toyota': 'Japon'}
>>> coches.update(coches01)
>>> coches
{'Fiat': 'Italia', 'Volkswagen': 'Aleman', 'Toyota': 'Japon', 'Volvo': 'Suecia'}
>>> coches01
{'Fiat': 'Italia', 'Toyota': 'Japon'}
>>>
```

Por supuesto, esta operación no es conmutativa.

Iteraciones en un diccionario

No es imprescindible ningún método para iterar sobre un diccionario:

```
for key in d:
    print key
```

y también

```
for key in d:
    print d[key]
```

Pero es posible usar el método `iterkeys()` para iterar sobre claves:

```
for key in d.iterkeys():
    print key
```

y el método `itervalues()` para iterar directamente sobre valores:

```
for val in d.itervalues():
    print val
```

Crear un diccionario de claves todas con el mismo valor

El método `fromkeys(lst, val)` permite crear un diccionario con las claves de la lista `lst` y todas significando el mismo valor `val`:

```
>>> lFerreteria = ['tornillos', 'puntillas', 'cancamos']
>>> ferreteria = dict.fromkeys(lFerreteria, 0)
>>> ferreteria
{'cancamos': 0, 'tornillos': 0, 'puntillas': 0}
>>>
```

Accediendo a una clave inexistente

Si tratamos de acceder a una clave que no existe, obtendremos un error:

```
>>> coches = {'Toyota' : 5, 'Isuzu': 10, 'Subaru' : 35}
>>> coches['Mercedes']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Mercedes'
```

Esto se puede prevenir utilizando el operador `in`:

```
>>> if 'Mercedes' in coches: print coches['Mercedes']
...
>>> if 'Isuzu' in coches: print coches['Isuzu']
...
10
```

Otra forma de acceder al valor de una clave es mediante el método `.get()`, que tiene dos argumentos: el primero es la clave que estamos probando y es obligatorio, el segundo es el valor por defecto que será devuelto si la clave o etiqueta no es encontrada:

```
>>> rst = [('Mercedes', 5), ('Fiat', 3), ('Subaru', 12)]
>>> coches = dict(rst)
>>> coches
{'Mercedes': 5, 'Fiat': 3, 'Subaru': 12}
>>> coches['Subaru']
12
>>> coches.get('Subaru')
12
>>> coches.get('Subaru', 'no encontrado')
12
>>> coches.get('Toyota')
>>> print coches.get('Toyota')
None
>>> coches.get('Toyota', 'no encontrado')
'no encontrado'
```

`None` es un valor especial como `null` o `nil`. Significa "ningún valor"; `get()` lo devuelve si no encuentra valor alguno en un diccionario.

Conexión entre listas y diccionarios

Al trabajar con Python, es inevitable el momento de tener que convertir listas en diccionarios y viceversa. Resultaría muy duro escribir una función que hiciese esto. Pero Python no sería Python si no ofreciese estas funcionalidades.

Si tenemos un diccionario

```
{"list": "Liste", "dictionary": "Wörterbuch", "function": "Funktion"}
```

podríamos convertirlo en una lista de tres pares:

```
[("list", "Liste"), ("dictionary", "Wörterbuch"), ("function", "Funktion")]
```

Es posible crear listas a partir de diccionarios usando los métodos `items()`, `keys()` y `values()`. Como el nombre indica, el método `keys()` crea una lista que consta solamente de las claves del diccionario, mientras que `values()` produce una lista consistente en los valores. `items()` puede ser usado para crear una lista consistente en los pares ordenados (clave, valor):

```
>>> coches = {'Lada' : 'Rusia', 'Fiat' : 'Italia', 'Audi' : 'Alemania'}
>>> coches.items()
[('Fiat', 'Italia'), ('Lada', 'Rusia'), ('Audi', 'Alemania')]
>>> coches.keys()
['Fiat', 'Lada', 'Audi']
>>> coches.values()
['Italia', 'Rusia', 'Alemania']
>>>
```

Si aplicamos el método `items()` a un diccionario, no hay pérdida de información, es decir, es posible recrear el diccionario original de la lista creada por `items()`. Aún siendo esta lista de pares, tiene la misma entropía, es decir, el contenido de información es el mismo; sin embargo la eficiencia de ambas aproximaciones es muy diferente. El tipo de dato diccionario provee métodos de acceso, borrado y cambio de elementos del diccionario altamente eficientes; mientras que en el caso de las listas, estas funciones deben ser implementadas por el programador.

Veremos cómo convertir listas en diccionarios, si las listas satisfacen ciertas condiciones. Supongamos que tenemos dos listas, una conteniendo platos y la otra los países correspondientes:

```
>>> coches = ['Toyota', 'Fiat', 'Audi', 'Lada']
>>> paises = ['Japon', 'Italia', 'Alemania', 'Rusia', 'España']
```

Ahora crearemos un diccionario que asigna un plato a un país, por supuesto de acuerdo con ciertos prejuicios. Para este propósito necesitamos la función `zip()`. El nombre `zip` fue bien elegido, porque las dos listas con combinadas como en una cremallera:

```
>>> coches_paises = zip(coches, paises)
>>> coches_paises
[('Toyota', 'Japon'), ('Fiat', 'Italia'), ('Audi', 'Alemania'), ('Lada', 'Rusia')]
```

La variable `coches_paises` contiene ahora el "diccionario" en forma de lista de pares. Esta forma puede ser fácilmente transformada en un diccionario real con la función `dict()`

```
>>> coches_paises_dict = dict(coches_paises)
>>> coches_paises_dict
{'Fiat': 'Italia', 'Lada': 'Rusia', 'Toyota': 'Japon', 'Audi': 'Alemania'}
```

Permanece aún una cuestión en relación con la función `zip()`. ¿Qué sucede si uno de los dos argumentos lista contiene más elementos que el otro? Es fácil: no serán usados los argumentos superfluos:

```
>>> coches = ['Toyota', 'Fiat', 'Audi', 'Lada']
>>> paises = ['Japon', 'Italia', 'Alemania', 'Rusia', 'Francia']
>>> coches_paises = zip(coches, paises)
>>> coches_paises
[('Toyota', 'Japon'), ('Fiat', 'Italia'), ('Audi', 'Alemania'), ('Lada', 'Rusia')]
>>> coches = ['Toyota', 'Fiat', 'Audi', 'Lada', 'Peugeot']
>>> paises = ['Japon', 'Italia', 'Alemania', 'Rusia']
>>> coches_paises = zip(coches, paises)
>>> coches_paises
[('Toyota', 'Japon'), ('Fiat', 'Italia'), ('Audi', 'Alemania'), ('Lada', 'Rusia')]
```


Diccionarios y Memoización

La *memoización* es la optimización clásica y puede ser fácilmente implementada con un diccionario. En la memoización una función calcula un resultado y se almacena en la memoria caché. En nuestra implementación, el argumento será la clave y el resultado el valor. Si la función sólo se llama una vez con cada argumento, la memoización no tiene ningún beneficio; si hay muchos argumentos, por lo general trabaja mal.

1.19. Medición del Rendimiento

Redactado a partir del contenido de [Apuntes sobre cálculo de rendimiento](#).

Este capítulo está dedicado al control del tiempo que emplea la ejecución de una función y la memoria que la misma utiliza. Para ejemplificar la exposición usaremos la función generadora de los términos de la sucesión de Fibonacci, codificada de dos formas distintas: iterativamente `fibonacciIte(n)` y recursivamente `fibonacciRec(n)`, donde `n` es el subíndice del término que deseamos obtener. La codificación iterativa viene dada por:

```
def fibonacciIte(n):
    a, b = 0, 1
    for i in xrange(n):
        a, b = b, a + b
    return a
```

y la recursiva por:

```
def fibonacciRec(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacciRec(n-1) + fibonacciRec(n-2)
```

La versión recursiva podrá ganar eficiencia si es implementada mediante la técnica de *memoización*:

```
mem = {0:0,1:1}
def fibonacciMem(n):
    global mem
    if n == 0:
        return 0
    elif n == 1:
        return 1
    if not n in mem:
        mem[n] = fibonacciMem(n-1) + fibonacciMem(n-2)
    return mem[n]
```

Para medir el tiempo ocupado por la ejecución de las anteriores funciones, daremos al fichero `fibonacci.py` el siguiente contenido:

```
from time import time

def fibonacciIte(n):
    a, b = 0, 1
    for i in xrange(n):
        a, b = b, a + b
    return a
```

```

def fibonacciRec(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacciRec(n-1) + fibonacciRec(n-2)

mem = {0:0,1:1}
def fibonacciMem(n):
    global mem
    if n == 0:
        return 0
    elif n == 1:
        return 1
    if not n in mem:
        mem[n] = fibonacciMem(n-1) + fibonacciMem(n-2)
    return mem[n]

t0 = time()
i = fibonacciIte(40)
t1 = time()

print 'ite toma %f' %(t1-t0)

t2 = time()
m = fibonacciMem(40)
t3 = time()

print 'mem toma %f' %(t3-t2)

t4 = time()
r = fibonacciRec(40)
t5 = time()

print 'rec toma %f' %(t5-t4)

print i, m , r

```

Al ser interpretado `fibonacci.py` tenemos el siguiente diálogo:

```

equipo:last-of-the-track miUsuario$ python fibonacci.py
ite toma 0.000008
mem toma 0.000040
rec toma 77.244675
102334155 102334155 102334155
equipo:last-of-the-track miUsuario$

```

Para controlar el tiempo de ejecución bastaría con importar `time` del módulo `time`, hacer un registro antes y otro después de la ejecución de la función y calcular la diferencia.

Podemos emplear un *decorador*:

```

@function
def f():
    pass

```

que es una abreviatura de:

```
def f():  
    pass  
f=function(f)
```

Es, como se suele decir, "zucar sintáctico".

En una versión de información menos extensa, y usando el decorador, considerar el fichero `rend.py` con el siguiente contenido:

```
import time  
  
def timeit(func):  
    def timed(*args, **kw):  
        ti = time.time()  
        result = func(*args, **kw)  
        tf = time.time()  
        print('Time:{:.10f} sec'.format(tf-ti))  
        return result  
    return timed  
  
@timeit  
def fibIte(n):  
    def fibonacciIte(n):  
        a, b = 0, 1  
        for i in xrange(n):  
            a, b = b, a + b  
        return a  
    return fibonacciIte(n)  
  
@timeit  
def fibRec(n):  
    def fibonacciRec(n):  
        if n == 0:  
            return 0  
        elif n == 1:  
            return 1  
        else:  
            return fibonacciRec(n-1) + fibonacciRec(n-2)  
    return fibonacciRec(n)  
  
@timeit  
def fibMem(n):  
    mem = {0:0,1:1}  
    def fibonacciMem(n):  
        if n == 0:  
            return 0  
        elif n == 1:  
            return 1  
        if not n in mem:  
            mem[n] = fibonacciMem(n-1) + fibonacciMem(n-2)  
        return mem[n]  
    return fibonacciMem(n)  
  
fibIte(40)  
fibMem(40)  
fibRec(40)
```

Interpretando el fichero `rend.py` tenemos el siguiente diálogo:

```
equipo:last-of-the-track miUsuario$ python rend.py
Time:0.0000090599 sec
Time:0.0000488758 sec
Time:86.9133579731 sec
equipo:last-of-the-track miUsuario$
```

En otra versión que proporciona información más extensa consideramos el fichero `decotime.py` que contiene el decorador y aporta una más completa información:

- proceso PID
- nombre de la función testada
- tiempo empleado en la evaluación medido en segundos
- RAM empleada en la evaluación medida en MB

El contenido de `decotime.py` es el siguiente:

```
import resource
import time
import os

def timeit(func):
    def timed(*args, **kw):
        ts = time.time()
        pid = os.getppid()
        result = func(*args, **kw)
        te = time.time()
        maxmem = (resource.getrusage(resource.RUSAGE_SELF).ru_maxrss)/1000
        info = (pid, func.__name__, te-ts, maxmem)
        print('PID:{}\nName:{}\nTime:{:.10f} sec\nRAM:{} MB'.format(*info))
        return result
    return timed
```

y es usado en el fichero `rendPlus.py` de la siguiente forma:

```
import decotime

@decotime.timeit
def fibIte(n):
    def fibonaccIte(n):
        a, b = 0, 1
        for i in xrange(n):
            a, b = b, a + b
        return a
    return fibonaccIte(n)

@decotime.timeit
def fibRec(n):
    def fibonaccIRec(n):
        if n == 0:
            return 0
        elif n == 1:
            return 1
        else:
            return fibonaccIRec(n-1) + fibonaccIRec(n-2)
```

```

    return fibonacciRec(n)

@decotime.timeit
def fibMem(n):
    mem = {0:0,1:1}
    def fibonacciMem(n):
        if n == 0:
            return 0
        elif n == 1:
            return 1
        if not n in mem:
            mem[n] = fibonacciMem(n-1) + fibonacciMem(n-2)
        return mem[n]
    return fibonacciMem(n)

print fibIte(40)
print fibMem(40)
print fibRec(40)

```

Interpretando el fichero rendPlus.py tenemos el siguiente diálogo:

```

equipo:last-of-the-track miUsuario$ python rendPlus.py
PID:934
Name:fibIte
Time:0.0000138283 sec
RAM:4493 MB
102334155
PID:934
Name:fibMem
Time:0.0002071857 sec
RAM:4517 MB
102334155
PID:934
Name:fibRec
Time:84.0719640255 sec
RAM:4517 MB
102334155
equipo:last-of-the-track miUsuario$

```

1.20. Argumentos de Funciones con Valores por Defecto

La forma más útil es especificar un valor por defecto para uno o más argumentos. Esto crea una función que puede ser llamada con menos argumentos que los que permite. Por ejemplo:

```

def pedir_confirmacion(prompt, reintentos=4, queja='Si o no, por favor!'):
    while True:
        ok = raw_input(prompt)
        if ok in ('s', 'S', 'si', 'Si', 'SI'): return True
        if ok in ('n', 'no', 'No', 'NO'): return False
        reintentos = reintentos - 1
        if reintentos < 0: raise IOError, 'usuario duro'
        print queja

```

Esta función puede ser llamada tanto así: `pedir_confirmacion('¿Realmente quiere salir?')` como así: `pedir_confirmacion('¿Sobreescribir archivo?', 2)`. Este ejemplo también introduce la palabra reservada `in`. Prueba si una secuencia contiene o no un determinado valor.

Los valores por defecto son evaluados en el momento de la definición de la función, en el ámbito de definición, entonces:

```
>>> i = 5
>>> def f(arg=i):
...     print arg
...
>>> f()
5
>>> i = 6
>>> f()
5
>>> f(6)
6
```

Advertencia importante: El valor por defecto es evaluado sólo una vez. Existe una diferencia cuando el valor por defecto es un objeto mutable como una lista, diccionario, o instancia de la mayoría de las clases. Por ejemplo, la siguiente función acumula los argumentos que se le pasan en subsiguientes llamadas:

```
>>> def g(a,lst=[]):
...     lst.append(a)
...     return lst
...
>>> g(1)
[1]
>>> g(2)
[1, 2]
>>> g(3)
[1, 2, 3]
```

Si no se quiere que el valor por defecto sea compartido entre subsiguientes llamadas, se puede escribir la función así:

```
>>> def f(a,lst=None):
...     if lst is None:
...         lst = []
...     lst.append(a)
...     return lst
...
>>> f(0)
[0]
>>> f(1)
[1]
>>> f(2)
[2]
```

1.21. Palabras Clave como Argumento

Las funciones también puede ser llamadas usando palabras claves como argumentos de la forma `keyword = value`. Por ejemplo, la siguiente función:

```
def loro(voltage, estado='muerto', accion='explotar', tipo='Azul Nordico'):
    print "-- Este loro no va a", accion,
    print "si le aplicas", voltage, "voltios."
```

```
print "-- Gran plumaje tiene el", tipo
print "-- Esta", estado, "!"
```

puede ser llamada de cualquiera de las siguientes formas:

```
loro(1000)
loro(accion = 'EXPLOTARRRRR', tension = 1000000)
loro('mil', estado= 'boca arriba')
loro('un millon', 'rostizado', 'saltar')
```

Así tenemos lo siguiente:

```
>>> def loro(voltage, estado='muerto', accion='explotar', tipo='Azul Nordico'):
...     print "-- Este loro no va a", accion,
...     print "si le aplicas", voltage, "voltios."
...     print "-- Gran plumaje tiene el", tipo
...     print "-- Esta", estado, "!"
...
>>> loro(1000)
-- Este loro no va a explotar si le aplicas 1000 voltios.
-- Gran plumaje tiene el Azul Nordico
-- Esta muerto !
>>> loro(accion = 'EXPLOTARRRRR', voltage = 1000000)
-- Este loro no va a EXPLOTARRRRR si le aplicas 1000000 voltios.
-- Gran plumaje tiene el Azul Nordico
-- Esta muerto !
>>> loro('mil', estado= 'boca arriba')
-- Este loro no va a explotar si le aplicas mil voltios.
-- Gran plumaje tiene el Azul Nordico
-- Esta boca arriba !
>>> loro('un millon', 'rostizado', 'saltar')
-- Este loro no va a saltar si le aplicas un millon voltios.
-- Gran plumaje tiene el Azul Nordico
-- Esta rostizado !
```

pero estas otras serían inválidas:

```
loro()                # falta argumento obligatorio
loro(tension=5.0, 'muerto') # argumento no-de palabra clave seguido de
                           # uno que si
loro(110, tension=220)  # valor duplicado para argumento
loro(actor='Juan Garau') # palabra clave desconocida
```

En general, una lista de argumentos debe tener todos sus argumentos posicionales seguidos por los argumentos de palabra clave, dónde las palabras claves deben ser elegidas entre los nombres de los parámetros formales. No es importante si un parámetro formal tiene un valor por defecto o no. Ningún argumento puede recibir un valor más de una vez (los nombres de parámetros formales correspondientes a argumentos posicionales no pueden ser usados como palabras clave en la misma llamada). Aquí hay un ejemplo que falla debido a esta restricción:

```
>>> def function(a):
...     pass
...
>>> function(0, a=0)
```

```
Traceback (most recent call last):
File "<stdin>", line 1, in ?
TypeError: function() got multiple values for keyword argument 'a'
```

Cuando un parámetro formal de la forma `**name` está presente al final, recibe un diccionario (ver `typesmapping`) conteniendo todos los argumentos de palabras clave excepto aquellos correspondientes a un parámetro formal. Esto puede ser combinado con un parámetro formal de la forma `*name` (descrito en [Sección 1.22](#)) que recibe una tupla conteniendo los argumentos posicionales además de la lista de parámetros formales. `*name` debe ocurrir antes de `**name`; por ejemplo, si definimos una función así:

```
def ventadequeso(tipo, *argumentos, **palabrasclaves):
    print "-- ¿Tiene", tipo, '?'
    print "-- Lo siento, nos quedamos sin", kind
    for arg in argumentos: print arg
    print '-'*40
    claves = palabrasclaves.keys()
    claves.sort()
    for c in claves: print c, ': ', palabrasclaves[c]
```

puede ser llamada así:

```
ventadequeso('Limburger', "Es muy liquido, sr.",
             "Realmente es muy muy liquido, sr.",
             cliente='Juan Garau',
             vendedor='Miguel Paez',
             puesto='Venta de Queso Manchego')
```

e imprimirá:

```
-- ¿Tiene Limburger ?
-- Lo siento, nos quedamos sin Limburger
Es muy liquido, sr.
Realmente es muy muy liquido, sr.
-----
cliente : Juan Garau
vendedor : Miguel Paez
puesto : Venta de Queso Manchego
```

Se debe notar que el método `sort()` de la lista de nombres de argumentos de palabra clave es llamado antes de imprimir el contenido del diccionario `palabrasclaves`; si esto no se hace, el orden en que los argumentos son impresos no está definido.

1.22. Lista de Argumentos Arbitrarios

La opción menos frecuentemente usada es especificar que una función puede ser llamada con un número arbitrario de argumentos. Estos argumentos serán organizados en una tupla. Antes del número variable de argumentos, cero o más argumentos normales pueden estar presentes:

```
def fprintf(file, template, *args):
    file.write(template.format(args))
```


1.23. Desempaquetando una Lista de Argumentos

La sintaxis especial `*args` y `**kwargs` en la definición de funciones es usada esencialmente para pasar a una función un número variable de argumentos. El asterisco simple `*args` es usado para pasar una lista de longitud variable y el asterisco doble `**kwargs` es usado para pasar un diccionario con un número arbitrario de parejas clave-valor.

Para el asterisco simple valga de ejemplo el siguiente:

```
def test_var_args(farg, *args):
    print "formal arg:", farg
    for arg in args:
        print "another arg:", arg
```

```
test_var_args(1, "two", 3)
```

que produce el siguiente resultado

```
formal arg: 1
another arg: two
another arg: 3
```

Para el asterisco doble valga lo siguiente:

```
def test_var_kwargs(farg, **kwargs):
    print "formal arg:", farg
    for key in kwargs:
        print "another keyword arg: %s: %s" % (key, kwargs[key])
```

```
test_var_kwargs(farg=1, myarg2="two", myarg3=3)
```

que produce el siguiente resultado

```
formal arg: 1
another keyword arg: myarg2: two
another keyword arg: myarg3: 3
```

Esta sintaxis especial puede ser usada no sólo en la definición de funciones sino también al llamar a una función:

```
def test_var_args_call(arg1, arg2, arg3):
    print "arg1:", arg1
    print "arg2:", arg2
    print "arg3:", arg3
```

```
args = ("two", 3)
test_var_args_call(1, *args)
```

que produce el siguiente resultado:

```
arg1: 1
arg2: two
arg3: 3
```

y también:

```
def test_var_args_call(arg1, arg2, arg3):
    print "arg1:", arg1
    print "arg2:", arg2
    print "arg3:", arg3
```

```
kwargs = {"arg3": 3, "arg2": "two"}
test_var_args_call(1, **kwargs)
```

que produce lo siguiente:

```
arg1: 1
arg2: two
arg3: 3
```

Por ejemplo, la función predefinida `range()` espera los argumentos inicio y fin. Si no están disponibles en forma separada, se puede escribir la llamada a la función con el operador para desempaquetar argumentos de una lista o una tupla *:

```
>>> range(3,6)           # llamada normal con argumentos separados
[3, 4, 5]
>>> args = [3,6]
>>> range(args)          # llamada errónea, sin argumento final
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    TypeError: range() integer end argument expected, got list.
>>> range(*args)          # llamada con argumentos desempquetados de una lista
[3, 4, 5]
```

Del mismo modo, los diccionarios pueden entregar argumentos de palabra clave con el operador **:

```
>>> def loro(voltage, estado='rostitizado', accion='explotar'):
...     print "-- Este loro no va a", accion,
...     print "si le aplicas", voltage, "voltios.",
...     print "Esta", estado, "!"
...
>>> d = {"tension": "cuatro millones", "estado": "demacrado",
...       "accion": "VOLAR"}
>>> loro(**d)
-- Este loro no va a VOLAR si le aplicas cuatro millones
   voltios. Esta demacrado !
```

Errores

Algunos errores son los siguientes. El retazo de código:

```
def fun(a, b, c):
    print a, b, c
```

```
l = [1,2,3]
fun(*l)
s = range(4)
fun(*s)
r = range(3)
fun(*r)
```

produce el siguiente efecto:

```
equipo:last-of-the-track miUsuario$ python desempaquetar.py
1 2 3
Traceback (most recent call last):
  File "desempaquetar.py", line 7, in <module>
    fun(*s)
TypeError: fun() takes exactly 3 arguments (4 given)
equipo:last-of-the-track miUsuario$
```

Hay que entender en la segunda parte de la prueba que al ir a desempaquetar para aplicar la función `fun`, ha sido convertido en argumento de la función uno más de los prescritos en la definición de `fun`; de ahí el error que ha sido elevado. De forma análoga, pero ahora por defecto y no por exceso, se explica el siguiente comportamiento.

```
def fun(a, b, c):
    print a, b, c
```

```
l = [1,2,3]
fun(*l)
r = range(2)
fun(*r)
```

que redundo en

```
equipo:last-of-the-track miUsuario$ python desempaquetar.py
```

```
1 2 3
```

```
Traceback (most recent call last):
```

```
File "desempaquetar.py", line 7, in <module>
```

```
    fun(*r)
```

```
TypeError: fun() takes exactly 3 arguments (2 given)
```

```
equipo:last-of-the-track miUsuario$
```

Sin embargo el código:

```
def fun(a, b, c):
    print a, b, c
```

```
r = range(2)
fun(2,*r)
```

no genera error:

```
equipo:last-of-the-track miUsuario$ python desempaquetar.py
```

```
2 0 1
```

```
equipo:last-of-the-track miUsuario$
```

Aunque hay que saber que en la llamada a la función los argumentos que completan no pueden suceder a una expresión `*args` o `*kwargs`, deben precederla siempre. Así, el código:

```
def fun(a, b, c):
    print a, b, c
```

```
r = range(2)
fun(*r,2)
```

hará que sea elevado un error al ser interpretado:

```
equipo:last-of-the-track miUsuario$ python desempaquetar.py
```

```
File "desempaquetar.py", line 5
```

```
    fun(*r,2)
```

```
SyntaxError: only named arguments may follow *expression
```

```
equipo:last-of-the-track miUsuario$
```

Como otra ilustración de lo mismo, también cosecharemos un fracaso con el código:

```
def fun(a, b, c):
    print a, b, c
```

```
r = range(2)
s = range(1)
fun(*r,*s)
```

1.24. Módulos

Un *módulo* en Python sirve para dar una organización lógica al código; agrupando código afín en un módulo se hace más fácil de entender y de usar. Un módulo es simplemente un archivo que consta de código Python: puede definir funciones, clases y variables, también puede incluir código ejecutable.

El código de un módulo llamado, digamos `unNombre`, está contenido en un fichero `unNombre.py`. Aquí hay un ejemplo de un módulo simple:

```
def printFunc(par):
    """
    imprime 'Hola:' seguido del nombre
    que se le d\`e en el string (o lo que sea)
    par
    """
    print "Hola: ", par
    return
```

que sería el contenido de un fichero llamado `hola.py`. Para invocar este módulo sencillo desde otro fichero, digamos `importHola.py` usaremos dentro de él la orden `import` en el siguiente modo⁴:

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

# Importar el módulo hola
import hola

# Ahora puede invocar así funciones definidas en el módulo
hola.printFunc('Pétula')
```

que producirá el siguiente diálogo:

```
miUsuario@equipo:~/Documents/ejemplos/python$ python importHola.py
Hola: Pétula
```

Un módulo es cargado una sola vez, independientemente del número de veces que sea importado al interpretar un fichero. Esto evita que la ejecución del módulo suceda una y otra vez si se producen varias importaciones.

En la sección [Sección 1.3](#), pese a ser introductoria, incluimos la información que supone el siguiente paso para el uso avanzado de módulos.

Python viene con una biblioteca de módulos estándar, descrita en un documento separado, la Referencia de la Biblioteca de Python (de aquí en más, "Referencia de la Biblioteca"). Algunos módulos se integran en el intérprete; estos proveen acceso a operaciones que no son parte del núcleo del lenguaje pero que sin embargo están integrados, tanto por eficiencia como para proveer acceso a primitivas del sistema operativo, como llamadas al sistema. El conjunto de tales módulos es una opción de configuración el cual también depende de la plataforma subyacente. Por ejemplo, el módulo `winreg` sólo se provee en sistemas Windows. Un módulo en particular merece algo de atención: `sys`, el que está integrado en todos los intérpretes de Python. Las variables `sys.ps1` y `sys.ps2` definen las cadenas usadas como cursores primarios y secundarios:

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
```

⁴La línea `# -*- coding: utf-8 -*-` permite trabajar codificando en UTF-8

```
,... '  
>>> sys.ps1 = '>> '  
>> print 'le voilà!  
le voilà  
>>
```

Estas dos variables están solamente definidas si el intérprete está en modo interactivo. La variable `sys.path` es una lista de cadenas que determinan el camino de búsqueda del intérprete para los módulos. Se inicializa por omisión a un camino tomado de la variable de entorno `PYTHONPATH`, o a un valor predefinido en el intérprete si `PYTHONPATH` no está configurada. Lo puede modificar usando las operaciones estándar de listas:

```
>>> import sys  
>>> sys.path.append('/ufs/guido/lib/python')
```

1.25. Archivos “Compilados” de Python

Como una importante aceleración del tiempo de arranque para programas cortos que usan en muchos de los módulos estándar, si un archivo llamado `spam.pyc` existe en el directorio donde se encuentra `spam.py`, se asume que contiene una versión ya “compilada a byte” del módulo `spam` (lo que se denomina *bytecode*). La fecha y hora de modificación del archivo `spam.py` usado para crear `spam.pyc` se graba en este último, y el `.pyc` se ignora si estos no coinciden.

Normalmente, no se necesita hacer nada para crear el archivo `spam.pyc` aparte de importarlo desde una sesión de Python. Siempre que se compile satisfactoriamente el `spam.py`, se hace un intento de escribir la versión compilada al `spam.pyc`. No es un error si este intento falla, si por cualquier razón el archivo no se escribe completamente el archivo `spam.pyc` resultante se reconocerá como inválido luego. El contenido del archivo `spam.pyc` es independiente de la plataforma, por lo que un directorio de módulos puede ser compartido por máquinas de diferentes arquitecturas.

Algunos consejos dados por gente experimentada:

- Cuando se invoca el intérprete de Python con la opción `-O`, se genera código optimizado que se almacena en archivos `.pyo`. El optimizador actualmente no ayuda mucho; sólo remueve las declaraciones `assert`. Cuando se usa `-O`, se optimiza todo el *bytecode*; se ignoran los archivos `.pyc` y los archivos `.py` se compilan a *bytecode* optimizado.
- Pasando dos opciones `-O` al intérprete de Python (`-OO`) causará que el compilador realice optimizaciones que en algunos raros casos podría resultar en programas que funcionen incorrectamente. Actualmente, solamente se remueven del *bytecode* a las cadenas `__doc__`, resultando en archivos `.pyo` más compactos. Ya que algunos programas necesitan tener disponibles estas cadenas, sólo deberías usar esta opción si sabés lo que estás haciendo.
- Un programa no corre más rápido cuando se lee de un archivo `.pyc` o `.pyo` que cuando se lee del `.py`; lo único que es más rápido en los archivos `.pyc` o `.pyo` es la velocidad con que se cargan.
- Cuando se ejecuta un script desde la línea de órdenes, nunca se escribe el *bytecode* del script a los archivos `.pyc` o `.pyo`. Por lo tanto, el tiempo de comienzo de un script puede reducirse moviendo la mayor parte de su código a un módulo y usando un pequeño script de arranque que importe el módulo. También es posible nombrar a los archivos `.pyc` o `.pyo` directamente desde la línea de órdenes.
- Es posible tener archivos llamados `spam.pyc` (o `spam.pyo` cuando se usa la opción `-O`) sin un archivo `spam.py` para el mismo módulo. Esto puede usarse para distribuir el código de una biblioteca de Python en una forma que es moderadamente difícil de hacerle ingeniería inversa.
- El módulo `compileall` puede crear archivos `.pyc` (o archivos `.pyo` cuando se usa la opción `-O`) para todos los módulos en un directorio.

1.26. Leer y Escribir Ficheros en Python

Para poder leer y escribir ficheros en Python es necesario conocer previamente la forma de abrirlos y cerrarlos. En Python los ficheros se abren con la función `open()`. Como primer parámetro se pasa el nombre del fichero y como segundo parámetro una cadena con caracteres similares a los de `fopen()` de C. Sólo algunos ejemplos:

- Abrir fichero para lectura (debe existir previamente): `f = open("fichero.txt","r")`
- Crea el archivo y lo abre para escribir desde cero : `f = open ("fichero.txt","w")`
- Abrir fichero de lectura en binario : `f = open("fichero.txt","rb")`
- Abre el archivo para escribir, creándolo si no existe, y sólo podemos agregar datos al final: `f = open ("fichero.txt","a")`

Para cerrarlo, basta llamar a `f.close()`. Valga como ejemplo el siguiente código:

```
[GCC 4.7.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> f = open('fichero.txt','w')
>>> f
<open file 'fichero.txt', mode 'w' at 0x1fdb5d0>
>>> f.close()
>>> f
<closed file 'fichero.txt', mode 'w' at 0x1fdb5d0>
>>>
```

Si se quiere, incluso se puede definir una función que abra ficheros:

```
>>> def creaciontxt():
...     archi=open('datos.txt','w')
...     archi.close()
...
>>> creaciontxt()
```

o si se prefiere, se puede usar este otro código que permite elegir el nombre del fichero que se va a crear:

```
[GCC 4.7.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> def creaciontxt(str):
...     archivo = open(str,'w')
...     archivo.close()
...
>>> creaciontxt('fichero.txt')
```

La función `open` retorna la referencia del objeto `file`. Luego llamamos al método `close` de la clase `file`. Si luego queremos ver si se ha creado el archivo de textos podemos hacerlo desde algún explorador de archivos, en la carpeta donde se encuentra nuestro programa en Python veremos un archivo llamado `datos.txt` o el nombre elegido, que tiene un tamaño de 0 bytes.

La siguiente es una lista con diferentes modos de abrir un fichero:

r	Abre un fichero para sólo lectura. El puntero del archivo se coloca en el principio del mismo. Éste es el modo predeterminado
---	---

rb	Abre un archivo de sólo lectura en formato binario. El puntero del archivo se coloca en el principio del mismo. Éste es el modo predeterminado
r+	Abre un archivo para lectura y escritura. El puntero del archivo estará en el principio del mismo
rb+	Abre un archivo para la lectura y escritura en formato binario. El puntero del archivo estará en el principio del mismo
w	Abre un archivo para escribir solamente. Sobrescribe el archivo si existe. Si el archivo no existe, se crea un nuevo archivo para escritura
wb	Abre un archivo para sólo escribir y en formato binario. Sobrescribe el archivo si el archivo existe. Si el archivo no existe, se crea un nuevo archivo para escritura
w+	Abre un fichero para escritura y lectura. Sobrescribe el archivo existente si existe el archivo. Si el archivo no existe, se crea un nuevo archivo para la lectura y la escritura
wb+	Abre un archivo, tanto para escritura como lectura en formato binario. Sobrescribe el archivo existente si existe ya. Si el archivo no existe, se crea un nuevo archivo para la lectura y la escritura
a	Abre un archivo para anexar. El puntero del archivo se encuentra al final del mismo si existe. Es decir, el archivo está en el modo de adición. Si el archivo no existe, se crea un nuevo archivo para escribir
ab	Abre un archivo para adjuntar en formato binario. El puntero del archivo se encuentra al final del mismo si existe. Es decir, el archivo está en el modo de adición. Si el archivo no existe, se crea un nuevo archivo para escritura
a+	Abre un fichero para anexar y leer. El puntero del archivo se encuentra al final del mismo si existe. El archivo se abre en el modo de adición. Si el archivo no existe, se crea un nuevo archivo para lectura y escritura
ab+	Abre un fichero para anexar y leer en formato binario. El puntero del archivo se encuentra al final del mismo si existe. El archivo se abre en el modo de adición. Si el archivo no existe, se crea un nuevo archivo para lectura y escritura

Abierto el fichero, queremos incluir texto en él. En el siguiente ejemplo mostramos cómo hacerlo:

```
Python 2.7.3 (default, Apr 10 2013, 05:13:16)
[GCC 4.7.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> def creartxt():
...     archi=open('datos.txt','w')
...     archi.close()
...
>>> def grabartxt():
...     archi=open('datos.txt','a')
...     archi.write('Linea 1\n')
...     archi.write('Linea 2\n')
...     archi.write('Linea 3\n')
...     archi.close()
...
>>> creartxt()
>>> grabartxt()
```

La función `creartxt` es similar al ejemplo anterior, y la función `grabartxt` tiene por objetivo abrir el archivo en modo de agregado. Cada vez que grabamos un string en el archivo de texto insertamos un salto de línea `\n`. Finalmente liberamos el archivo llamando al método `close`. Para ver el contenido de nuestro archivo de texto debemos utilizar un editor de texto; también podemos usar la orden `cat` del sistema:

```
miUsuario@miEquipo:~/Documents/miCamino$ cat datos.txt
Linea 1
Linea 2
Linea 3
```

Es posible la lectura "línea a línea" de un fichero. La clase `file` tiene un método llamado `readline()` que retorna toda una línea del archivo de texto y deja posicionado el puntero de archivo en la siguiente línea. Cuando llega al final del archivo `readline` retorna un string vacío. Nuestra sesión de Python sigue evolucionando:

```
Python 2.7.3 (default, Apr 10 2013, 05:13:16)
[GCC 4.7.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> def creartxt():
...     archi=open('datos.txt','w')
...     archi.close()
...
>>> def grabartxt():
...     archi=open('datos.txt','a')
...     archi.write('Linea 1\n')
...     archi.write('Linea 2\n')
...     archi.write('Linea 3\n')
...     archi.close()
...
>>> creartxt()
>>> grabartxt()
>>> def leertxt():
...     archi=open('datos.txt','r')
...     linea=archi.readline()
...     while linea!="":
...         print linea
...         linea=archi.readline()
...     archi.close()
...
>>> leertxt()
Linea 1

Linea 2

Linea 3

>>>
```

Si se desea se pueden guardar en una lista las líneas leídas. Esto nos lo permitiría el siguiente código añadido al anterior:

```
>>> def leertxtenlista():
...     archi=open('datos.txt','r')
...     lineas=archi.readlines()
```



```
...     print lineas
...     archi.close()
...
>>> leertxttenlista()
['Linea 1\n', 'Linea 2\n', 'Linea 3\n']
>>>
```

Hemos utilizado el método `readlines()` en lugar de `readline()`. El método `readlines()` retorna una lista con cada línea del archivo de texto.

El método `tell()` devuelve la posición del puntero de lectura/escritura dentro del fichero. La sintaxis del método es:

```
objetoFichero.tell()
```

Válganos el siguiente ejemplo:

```
>>> fichero = open('datos.txt', 'r')
>>> print "Nombre del fichero: ", fichero.name
Nombre del fichero:  datos.txt
>>> linea = fichero.readline()
>>> print "Linea leida: %s" % (linea)
Linea leida: Linea 1
>>> pos = fichero.tell()
>>> print "Posicion actual: %d" % (pos)
Posicion actual: 8
>>> fichero.close()
```

1.27. Interacción con el Sistema: el módulo *os*

Python provee un módulo estándar de suma utilidad llamado *os*. Mediante él podemos usar el sistema operativo subyacente (Linux, Mac OS X o ... Windows) desde la consola de Python.

El módulo es importado en la forma habitual. La [Figura 1.3](#) contiene un resumen esquemático de la utilización del módulo *os* de Python.

1.28. Definición de Nuevos Tipos de Datos

Resumen de Terminología en PDO

Python es un lenguaje orientado a objetos desde que existe. Debido a ello la creación y el uso de clases y objetos es francamente fácil.

Demos ahora algo de terminología frecuente en este tipo de lenguajes:

1. *Clase*: un prototipo definido por el usuario para un objeto que define un conjunto de atributos caracterizadores de cualquier objeto de la clase. Los atributos son datos de los miembros (variables de clase y variables de instancia) y métodos, a los atributos se accede vía la notación punto.
2. *Variable de clase*: una variable que comparte cualquier instancia de la clase. Se definen en el código de la clase pero fuera del código de cualquier método de la clase. No se usan con tanta frecuencia como las *variables de instancia*.
3. *Variable de instancia*: una variable que se define dentro de un método. No es accesible fuera del código de ese método.

<code>os.system()</code>	Ejecutar una orden de la shell
<code>os.environ()</code>	Proporciona el ambiente de los usuarios
<code>os.getcwd()</code>	Proporciona el directorio de trabajo actual
<code>os.getgid()</code>	Devuelve el id real de grupo del proceso actual
<code>os.getuid()</code>	Devuelve la id de usuario del proceso actual
<code>os.getpid()</code>	Devuelve la id real del proceso actual
<code>os.umask(mask)</code>	Define la umask numérica actual y devuelve la umask previa
<code>os.uname()</code>	Devuelve información identificativa del sistema operativo en uso
<code>os.chroot(path)</code>	Cambia el directorio raíz del proceso actual al camino dado. Para usar este método serán necesarios privilegios de superusuario.
<code>os.listdir(path)</code>	Devuelve una lista de string, cada uno de los cuales es el nombre de uno de los ficheros en el camino 'path'. Los ficheros <code>.</code> y <code>..</code> son exceptuados en la enumeración.
<code>os.mkdir(path)</code>	Crea un directorio llamado path que debe ser un string
<code>os.makedirs(path)</code>	Es una función creadora de directorios recursivamente. Es como <code>os.mkdir()</code> pero creando todos los directorios intermedios necesarios para contener la hoja del string path
<code>os.remove(path)</code>	Elimina el fichero path. Si el string path es un directorio se emite un <code>OSError</code>
<code>os.removedirs(path)</code>	Elimina directorios recursivamente. Si el directorio hoja es eliminado con éxito, <code>removedirs</code> intenta eliminar sucesivamente cada directorio padre relacionado en el string path
<code>os.rename(src, dst)</code>	Renombra el fichero o directorio src a dst. Si dst es un fichero o directorio ya presente, será emitido un <code>OSError</code>
<code>os.rmdir(path)</code>	Elimina el directorio del string path. Funciona sólo cuando el directorio está vacío, en otro caso es emitido un <code>OSError</code>

Figura 1.3: Uso del módulo `os` de Python.

4. *Instancia*: un objeto individual de cierta clase de objetos es lo que se llama una *instancia de dicha clase*.
5. *Instanciación*: creación de una instancia de una clase.
6. *Método*: cualquier función definida en el código que define una clase.
7. *Objeto*: la estructura de dato que es definida por su clase. Comprende los atributos: datos de miembros y métodos.

Creando Clases y e Instancias de Objetos

El código que define cualquier clase responde al siguiente esquema:

```
class ClassName(object):
    'lista de caracteres de la documentación opcional de la clase'
    código_de_la_clase
```

Sabiendo que:

- La clase tiene una documentación en forma de string o lista de caracteres a la que se puede acceder vía `ClassName.__doc__`
- El `código_de_la_clase` consta del código definitorio de los miembros de la clase: datos de miembros y métodos.

Ejemplo 1.28.1. Aquí damos un ejemplo sencillo de lo dicho:

```
class Empleado(object):
    'Clase base para todos los empleado'
    empContador = 0

    def __init__(self, nombre, salario):
        self.nombre = nombre
        self.salario = salario
        Empleado.empContador += 1

    def presentaContador(self):
        print "Total Employee %d" % Empleado.empContador

    def presentaEmpleado(self):
        print "Nombre : ", self.nombre, ", Salario: ", self.salario
```

Seguidamente podemos crear instancias de la clase `Empleado` y “acceder” a sus propiedades:

```
"Esto crearía el primer empleado de la clase"
emp1 = Empleado("Rafaela", 2000)
"Esto crearía el segundo empleado de la clase"
emp2 = Empleado("Mariano", 5000)

emp1.presentaEmpleado()
emp2.presentaEmpleado()

print "Total de Empleados %d" % Empleado.empContador
```

En definitiva podemos juntar todo en el fichero `empleado.py`:

```
class Empleado(object):
    'Clase base para todos los empleado'
    empContador = 0
```

```

def __init__(self, nombre, salario):
    self.nombre = nombre
    self.salario = salario
    Empleado.empContador += 1

def presentaContador(self):
    print "Total Employee %d" % Empleado.empContador

def presentaEmpleado(self):
    print "Nombre: ", self.nombre, ", Salario: ", self.salario

"Esto crearía el primer empleado de la clase"
emp1 = Empleado("Rafaela", 2000)
"Esto crearía el segundo empleado de la clase"
emp2 = Empleado("Mariano", 5000)

emp1.presentaEmpleado()
emp2.presentaEmpleado()

print "Total de Empleados %d" % Empleado.empContador

```

y ejecutarlo en la consola con la orden y efecto:

```

equipo:last-of-the-track miUsuario$ python empleado
Nombre : Rafaela , Salario: 2000
Nombre : Mariano , Salario: 5000
Total de Empleados 2

```

Se pueden añadir, modificar o suprimir atributos de objetos en cualquier momento. Para ilustrar esta afirmación se puede añadir el siguiente código al fichero empleado.py:

```

emp1.edad = 36 # Añade un atributo de 'edad'.
print emp1.edad
emp1.edad = 38 # Modifica el atributo de 'edad'.
print emp1.edad
del emp1.edad # Borra el atributo de 'edad'.
print emp1.edad

```

En lugar de usar las sentencias normales a los atributos se pueden usar los siguientes métodos:

- `getattr(obj, nombre[, default])`: para acceder al atributo del objeto.
- `hasattr(obj,nombre)`: comprueba si el atributo name existe o no, devolviendo True si, y sólo si, existe.
- `setattr(obj,nombre,valor)`: da al atributo nombre el valor valor y si no existe, lo crea previamente.
- `delattr(obj, nombre)`: borra el atributo nombre

Atributos de Clase Incorporados

Cada clase de Python tiene incorporados los siguientes atributos:

- `__dict__`: Diccionario que contiene el espacio de nombres de la clase.
- `__doc__`: Cadena de documentación de clase o nada, si no ha sido aportada.
- `__name__`: nombre de la clase

- `__module__`: Nombre del módulo en el que se define la clase. Este atributo es `__main__` en modo interactivo.
- `__bases__`: Una tupla posiblemente vacía que contiene las clases de base en el orden de su aparición en la lista de la clase base.

Ejemplo 1.28.2. Para la clase `Empleado`, demos el siguiente contenido al fichero `empleado_metodos_builtin.py`:

```
class Empleado(object):
    'Clase base para todos los empleado'
    empContador = 0

    def __init__(self, nombre, salario):
        self.nombre = nombre
        self.salario = salario
        Empleado.empContador += 1

    def presentaContador(self):
        print "Total Employee %d" % Empleado.empContador

    def presentaEmpleado(self):
        print "Nombre: ", self.nombre, ", Salario: ", self.salario

print "Empleado.__doc__:", Empleado.__doc__
print "Empleado.__name__:", Empleado.__name__
print "Empleado.__module__:", Empleado.__module__
print "Empleado.__bases__:", Empleado.__bases__
print "Empleado.__dict__:", Empleado.__dict__
```

entonces se tiene el siguiente diálogo:

```
equipo:last-of-the-track miUsuario$ python empleado_metodos_builtin.py
Empleado.__doc__: Clase base para todos los empleado
Empleado.__name__: Empleado
Empleado.__module__: __main__
Empleado.__bases__: ()
Empleado.__dict__: {'__module__': '__main__', 'presentaContador':
<function presentaContador at 0x107b87c80>, 'presentaEmpleado':
<function presentaEmpleado at 0x107b8aaa0>, '__doc__':
'Clase base para todos los empleado', '__init__':
<function __init__ at 0x107b87578>, 'empContador': 0}
equipo:last-of-the-track miUsuario$
```

Herencia

Es la propiedad de crear nuevos datos a partir de los ya existentes (progenitores). Es la transferencia de las características de una clase a otras clases que son derivadas de ella, heredando sus atributos y métodos. A veces podemos sobrescribirlos para adaptarlos a la clase heredada (clase hija):

- La herencia es el mecanismo de reutilización de código por excelencia en Programación Orientada a Objetos.
- Sirve para ampliar, particularizar o mejorar determinadas clases en otras nuevas. Las clases padre/-madre siguen vigentes, por lo que no es necesario retocar el código que ya funcionaba.

Dada una clase `Madre` podemos crear otra clase `Hija` de la siguiente forma:

```
class Hija(Madre):
    codigo_hija
```

sabiendo que:

- El código de la hija puede sobrescribir los métodos de la madre, para los que ello esté permitido en la definición de ésta, e introducir nuevos atributos, si son necesarios.
- Según se puede observar, toda clase es hija de la clase object.

En primer lugar veamos la **herencia junto a la sobrecarga de métodos** en un ejemplo. Creemos el archivo HerenciaSobrecarga.py con el siguiente contenido:

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

class Madre (object):
    def __init__(self,parametro1):
        self.par1 = parametro1
    def metodo1 (self):
        print 'metodo1 de la clase Madre'
    def metodo2 (self):
        print 'metodo2 de la clase Madre'

class Hija (Madre):
    """ Clase Hija. Deriva de Madre """
    def metodo1 (self):
        'metodo1: imprime un mensaje simple'
        print 'metodo1 de la clase Hija'
```

que nos ofrece el siguiente diálogo:

```
equipo:last-of-the-track miUsuario$ python
Python 2.7.6 (default, Sep  9 2014, 15:04:36)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.39)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from HerenciaSobrecarga import *
>>> dir(Madre)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__',
 '__getattr__', '__hash__', '__init__', '__module__', '__new__',
 '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
 '__sizeof__', '__str__', '__subclasshook__', '__weakref__',
 'metodo1', 'metodo2']
>>> objeto_madre = Madre('Madre1')
>>> dir(objeto_madre)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__',
 '__getattr__', '__hash__', '__init__', '__module__', '__new__',
 '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
 '__sizeof__', '__str__', '__subclasshook__', '__weakref__',
 'metodo1', 'metodo2', 'par1']
>>> objeto_hija = Hija ()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __init__() takes exactly 2 arguments (1 given)
>>> objeto_hija = Hija (24)
>>> objeto_hija.par1
24
>>> objeto_hija.metodo1()
metodo1 de la clase Hija
>>> objeto_hija.metodo2()
metodo2 de la clase Madre
>>>
```

En segundo lugar veremos la **herencia junto a nuevos atributos**. Cuando la clase hija tiene nuevos atributos, para inicializarla tenemos dos posibilidades:

Caso 1 Iniciamos todos, lo que resulta sencillo en el caso de tener pocos atributos.

Caso 2 Utilizamos la inicialización de la clase madre para los atributos heredados y nueva inicialización para los nuevos. Implica un diseño más elaborado, pero una programación orientada a objetos más reutilizable y organizada.

Para el **Caso 1** creemos el fichero HerenciaNuevosAtributos1.py con el siguiente contenido:

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

class Madre (object):
    def __init__ (self, par):
        self.uno = par

class Hija (Madre):
    def __init__ (self, par1, par2):
        self.uno = par1
        self.dos = par2

if __name__ == '__main__':
    m = Madre ('madre')
    h = Hija ('hija1', 'hija2')
    print 'atributo de la madre =', m.uno
    print 'atributo uno de la hija =', h.uno
    print 'atributo dos de la hija =', h.dos
```

que ofrece el siguiente diálogo:

```
equipo:last-of-the-track miUsuario$ python HerenciaNuevosAtributos1.py
atributo de la madre = madre
atributo uno de la hija = hija1
atributo dos de la hija = hija2
equipo:last-of-the-track miUsuario$
```

Para el **Caso 2** creemos el fichero HerenciaNuevosAtributos2.py con el siguiente contenido:

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

class Madre (object):
    def __init__ (self, par):
        print 'constructor de la clase Madre.'
        self.uno = par

class Hija (Madre):
    def __init__ (self, par1, par2):
        print 'constructor de la clase Hija.'
        super (Hija,self).__init__(par1)
        self.dos = par2

if __name__ == '__main__':
    m = Madre ('madre')
    h = Hija ('hija1', 'hija2')
    print 'atributo de la madre =', m.uno
    print 'atributo uno de la hija =', h.uno
    print 'atributo dos de la hija =', h.dos
```

Como se ve, llamamos al constructor de la clase madre y le pasamos los parámetros necesarios. El resto se inicializa en el constructor de la hija. El constructor de la clase madre se llama de la siguiente forma:

```
super(clase_hija, self).__init__([parametros])
```

Se nos ofrece el siguiente diálogo:

```
equipo:last-of-the-track miUsuario$ python HerenciaNuevosAtributos2.py
constructor de la clase Madre.
constructor de la clase Hija.
constructor de la clase Madre.
atributo de la madre = madre
atributo uno de la hija = hija1
atributo dos de la hija = hija2
equipo:last-of-the-track miUsuario$
```

La herencia múltiple es posible y sería a través del siguiente código:

```
class Clase_Hija (ClaseMadre1,ClaseMadre2,...):
    codigo_Clase_Hija
```

Sobrecarga

La siguiente tabla enumera algunas funciones genéricas cuya definición puede ser modificada en nuestras clases:

Method	Description	Sample Call
<code>__init__ (self,[args])</code>	Constructor (with any optional arguments)	<code>obj = className(args)</code>
<code>__del__(self)</code>	Destructor, deletes an object	<code>del(obj)</code>
<code>__repr__(self)</code>	Evaluatable string representation	<code>repr(obj)</code>
<code>__str__(self)</code>	Printable string representation	<code>str(obj)</code>
<code>__cmp__ (self, x)</code>	Object comparison	<code>cmp(obj, x)</code>

Ejemplo 1.28.3. Definamos la clase de los vectores de dos componentes y sobrecarguemos oportunamente el método `__add__`. Para ello construyamos el fichero `Vector.py` con el siguiente contenido:

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

class Vector (object):
    def __init__(self, a = 0, b = 0):
        self.a = a
        self.b = b

    def __str__(self):
        return 'Vector (%d, %d)' % (self.a, self.b)

    def __add__(self,otro):
        return Vector(self.a + otro.a, self.b + otro.b)
```

que nos ofrece el siguiente diálogo:

```
equipo:last-of-the-track miUsuario$
Python 2.7.6 (default, Sep  9 2014, 15:04:36)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.39)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from vectores import *
>>> v1 = Vector(2,10)
>>> v2 = Vector(5,-2)
```



```
>>> v = v1 + v2
>>> v
<vectores.Vector object at 0x102a342d0>
>>> print v
Vector (7, 8)
>>> v.a
7
>>> v.b
8
>>>
```

Ocultando datos

Un atributo de un objeto puede ser o no visible fuera del código de definición de la clase. Para que no lo sea debemos prefijarlo con un doble guión bajo.

Considérese el siguiente ejemplo:

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

class JustCounter:
    __secretCount = 0

    def count(self):
        self.__secretCount += 1
        print self.__secretCount

counter = JustCounter()
counter.count()
counter.count()
print counter.__secretCount
```

que produce el siguiente diálogo:

```
1
2
Traceback (most recent call last):
  File "test.py", line 12, in <module>
    print counter.__secretCount
AttributeError: JustCounter instance has no attribute '__secretCount'
```

Python protege esos miembros cambiándole internamente el nombre incluyendo el nombre de la clase. Se puede acceder a dichos atributos mediante `objeto._className__attrName`. Es decir, el código:

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

class JustCounter:
    __secretCount = 0

    def count(self):
        self.__secretCount += 1
        print self.__secretCount

counter = JustCounter()
counter.count()
counter.count()
print print counter._JustCounter__secretCount
```

produciría el diálogo:

```
1
2
2
```

Destrucción de Objetos (Recolectores de Basura)

Python elimina objetos que no son ya necesarios (tipos incorporados o instancias de clase) automáticamente para liberar espacio de memoria. El proceso por el cual Python reclama periódicamente bloques de memoria que ya no están en uso se denomina *recolección de basura*.

El recolector de basura de Python se ejecuta durante la ejecución del programa y se activa cuando el contador de referencia de un objeto llega a cero. El contador de referencia de un objeto cambia a medida que el número de alias que apuntan a sus cambios.

El contador de referencia de un objeto aumenta cuando se le asigna al objeto un nuevo nombre o es colocado en un contenedor (lista, upla, diccionario). El contador de referencia de un objeto decrece cuando es borrado con `del`, su referencia es reasignada o su referencia sale del rango. Cuando la el contador de referencia de un objeto llega a cero, Python recolecta al objeto automáticamente.

```
a = 40      # Se crea el objeto <40>
b = a      # Aumenta el count de ref. de <40>
c = [b]    # Aumenta el count de ref. de <40>

del a      # Decrece el contador de ref. de <40>
b = 100    # Decrece el contador de ref. de <40>
c[0] = -1  # Decrece el contador de ref. de <40>
```

Normalmente no percibiremos que el recolector de basura destruye una instancia huérfana y reclama su espacio. Sin embargo, una clase puede implementar el método especial `__del__()`, llama destructor, que es invocada cuando la instancia está a punto de ser destruida. Este método podría ser utilizado para limpiar los recursos, no de memoria, utilizados por una instancia.

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

class Point:
    def __init__( self, x=0, y=0):
        self.x = x
        self.y = y
    def __del__(self):
        class_name = self.__class__.__name__
        print class_name, "destroyed"

pt1 = Point()
pt2 = pt1
pt3 = pt1
print id(pt1), id(pt2), id(pt3) # prints the ids of the ojects
del pt1
del pt2
del pt3
```

Al ejecutar este código produce el siguiente diálogo:

```
3083401324 3083401324 3083401324
Point destroyed
```

1.29. Formateo Elegante de la Salida

Para imprimir un formateo elegante a nuestras salidas usaremos los métodos `str()` o `repr()` sin entrar ahora en los matices diferenciadores para las distintas versiones de Python, los cuales se han ido atenuando —todo hay que decirlo. Veamos el siguiente diálogo:

```
equipo:last-of-the-track miUsuario$ python
Python 2.7.6 (default, Jun 22 2015, 17:58:13)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'El valor de x es ' + repr(x) + ' y es ' + repr(y) + ' ...'
>>> print s
El valor de x es 32.5 y es 40000 ...
>>> hola = 'hola ... mundo\n'
>>> holas = repr(hola)
>>> print holas
'hola ... mundo\n'
>>> repr((x,y,('honda','toyota')))
'(32.5, 40000, ('honda', 'toyota'))'
>>>
```

Ejemplo 1.29.1. Ahora vamos a dar una forma de escribir una tabla de cuadrados y cubos:

```
equipo:last-of-the-track miUsuario$ python
Python 2.7.6 (default, Jun 22 2015, 17:58:13)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> for x in xrange(1,11):
...     print repr(x).rjust(2), repr(x*x).rjust(3),
...     print repr(x*x*x).rjust(4)
...
1   1   1
2   4   8
3   9  27
4  16  64
5  25 125
6  36 216
7  49 343
8  64 512
9  81 729
10 100 1000
>>>
```

Obsérvese que fue agregado un espacio entre las columnas. Ello es por la forma en que `print` trabaja: siempre agrega espacios entre sus argumentos. De forma equivalente tenemos:

```
equipo:last-of-the-track miUsuario$ python
Python 2.7.6 (default, Jun 22 2015, 17:58:13)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> for x in xrange(1,11):
...     print '{0:2d} {1:3d} {2:4d}'.format(x,x*x,x*x*x)
...
1   1   1
2   4   8
3   9  27
```

```

4 16 64
5 25 125
6 36 216
7 49 343
8 64 512
9 81 729
10 100 1000
>>>

```

Este ejemplo muestra el método `rjust()` de los objetos cadena, el cual ordena una cadena a la derecha en un campo del ancho dado llenándolo con espacios a la izquierda. Hay métodos similares `ljust()` y `center()`. Estos métodos no escriben nada, sólo devuelven una nueva cadena. Si la cadena de entrada es demasiado larga, no la truncan, sino que la devuelven intacta; esto romperá la alineación de las columnas, pero es normalmente mejor que la alternativa, que estaría mintiendo sobre el valor. Si realmente deseamos que se recorte, siempre podemos agregarle una operación de rebanado, como en `x.ljust(n)[:n]`.

El método `zfill()` rellena una cadena numérica a la izquierda con ceros interaccionando bien con signos positivos y negativos:

```

>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
>>>

```

El método `format()`

El uso básico del método `str.format()` es según lo siguiente:

```

>>> print 'Somos los {0} quienes decimos "{1}"'.format('caballeros', 'No')
Somos los caballeros quienes decimos "No"

```

Las llaves y caracteres dentro de las mismas (llamados campos de formato) son reemplazadas con los objetos pasados en el método `format()`. El número en las llaves se refiere a la posición del objeto pasado en el método:

```

>>> print '{0} y {1}'.format('carne', 'huevos')
carne y huevos
>>> print '{1} y {0}'.format('carne', 'huevos')
huevos y carne

```

Si se usan argumentos nombrados en el método `format()`, sus valores serán referidos usando el nombre del argumento:

```

>>> print 'Esta {comida} es {adjetivo}'.format(comida='carne',adjetivo='espantosa')
Esta carne es espantosa.

```

Se pueden combinar arbitrariamente argumentos posicionales y nombrados:

```

>>> print 'La historia de {0}, {1}, y {otro}'.format('Bill', 'Manfred', otro='Georg')
La historia de Bill, Manfred, y Georg.

```

Tras el nombre del campo pueden ir un ":" y especificadores de formato opcionales. Esto aumenta el control sobre cómo el valor es formateado. El siguiente ejemplo trunca π a tres lugares tras el punto decimal redondeando:

```

>>> print 'El valor de PI es aproximadamente {0:.3f}'.format(math.pi)
El valor de PI es aproximadamente 3.142.

```

Pasando un entero detrás del ":" causará que el campo sea de un mínimo número de caracteres de ancho. Esto es útil para hacer tablas bonitas:

```
>>> tabla = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for nombre, telefono in tabla.items():
...     print '{0:10} ==> {1:10d}'.format(nombre, telefono)
...
Dcab      ==>      7678
Jack      ==>      4098
Sjoerd    ==>      4127
```

Si se tiene una cadena de formato realmente larga que no se quiere separar, podría ser bueno que se pudiera hacer referencia a las variables a ser formateadas por el nombre en vez de la posición. Esto puede hacerse simplemente pasando el diccionario y usando corchetes [] para acceder a las claves:

```
>>> tabla = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print 'Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; Dcab: {0[Dcab]:d}'.format(tabla)
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

Esto se podría hacer también pasando la tabla como argumentos nombrados con la notación `***`:

```
>>> tabla = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print 'Jack: {Jack:d}; Sjoerd: {Sjoerd:d}; Dcab: {Dcab:d}'.format(**tabla)
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

Esto es particularmente útil en combinación con la nueva función integrada `vars()`, que devuelve un diccionario conteniendo todas las variables locales.

Para ver una descripción completa del formato de cadenas con `str.format()` se puede consultar `formatstrings`.

Viejo formato de cadenas

El operador `%` también puede usarse para formato de cadenas. Interpreta el argumento de la izquierda con el estilo de formato de `sprintf` para ser aplicado al argumento de la derecha, y devuelve la cadena resultante de esta operación de formato. Por ejemplo:

```
>>> import math
>>> print 'El valor de PI es aproximadamente %5.3f.' % math.pi
El valor de PI es aproximadamente 3.142.
```

Ya que `str.format()` es bastante nuevo, un montón de código Python todavía usa el operador `%`. Sin embargo, ya que este viejo estilo de formato será eventualmente eliminado del lenguaje, en general debería usarse `str.format()`.

1.30. Manipulación del contenido de un fichero en Python

Para asignar a una variable un valor de tipo `file`, basta recurrir a la función integrada `open()`, la cual está destinada a la apertura del archivo, requiriendo dos parámetros:

- El primero de ellos es la ruta hacia el archivo que se desea abrir.
- El segundo, el modo en el cual abrirlo. La información detallada sobre este segundo parámetro se encuentra en la tabla de la [Sección 1.26](#).

Veamos el siguiente ejemplo:

```
equipo:last-of-the-track miUsuario$ python
Python 2.7.6 (default, Sep  9 2014, 15:04:36)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.39)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> f = open("passacaglia_della_vita.txt")
>>> f
<open file 'passacaglia_della_vita.txt', mode 'r' at 0x104a2a6f0>
>>> f.close()
>>> f
<closed file 'passacaglia_della_vita.txt', mode 'r' at 0x104a2a6f0>
>>>
```

Lectura del fichero

Para leer el fichero podemos usar las órdenes `f.read()` y `f.readline()`:

- Lectura de todo el fichero de una vez: `dato = f.read()`
- Lectura de 100 bytes: `dato = f.read(100)`
- Lectura de una línea completa: `dato = f.readline()`

```
equipo:last-of-the-track miUsuario$ python
Python 2.7.6 (default, Sep  9 2014, 15:04:36)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.39)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
```

```
>>> f = open("passacaglia_della_vita.txt")

>>> f.read()
'0h come t\xe2\x80\x99inganni ... e puoi dire: bisogna morire.\n'
>>> f.close()

>>> f = open("passacaglia_della_vita.txt")
>>> dato = f.read(30)
>>> dato
'0h come t\xe2\x80\x99inganni se pensi c'
>>> f.close()

>>> f = open("passacaglia_della_vita.txt")
>>> dato = f.readline()
>>> dato
'0h come t\xe2\x80\x99inganni se pensi che gli anni\n'
>>> f.close()
```

Escritura del fichero

Para escribir en un fichero inexistente hasta el momento, digamos `ata.txt`, o destruyendo previamente su contenido si existiese:

```
>>> f = open("ata.txt", "w")
>>> f.write("ata\n la\n jaca\n a\n la\n estaca\n")
>>> f.close()
>>> f = open("ata.txt")
>>> f.read()
'ata\n la\n jaca\n a\n la\n estaca\n'
>>> f.close()
>>>
```

Para escribir sobre un fichero existente añadiendo al final

```
equipo:last-of-the-track miUsuario$ python
Python 2.7.6 (default, Sep  9 2014, 15:04:36)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.39)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
Python 2.7.6 (default, Sep  9 2014, 15:04:36)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.39)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> f = open("ata.txt", "a")
>>> f.write("tres tristes tigres\n")
>>> f.close()
>>> f = open("ata.txt")
>>> f.read()
'ata\n la\n jaca\n a\n la\n estaca\ntres tristes tigres\n'
>>> f.close()
>>>
```

Moviéndose por el fichero

Con `f.tell()` podemos saber en qué posición estamos del fichero y con `f.seek()` podemos desplazarnos por él, para leer o escribir en una determinada posición:

- `f.seek(n)`: Ir al byte `n` del fichero
- `f.seek(n,0)` : Equivalente al anterior
- `f.seek(n,1)`: Desplazarnos `n` bytes a partir de la posición actual del fichero
- `f.seek(n,2)` : Situarnos `n` bytes antes del final de fichero.

El segundo parámetro en estos ejemplos es:

- Ninguno ó 0: la posición es relativa al principio del fichero
- 1: la posición es relativa a la posición actual
- 2: la posición es relativa al final del fichero y hacia atrás.

```
>>> f = open ("pom.xml")
>>> f.seek(0,2)
>>> f.tell()
632L
>>> f.seek(625)
>>> f.read()
'object>\n'
>>> f.close()
```

En el ejemplo abrimos un fichero `pom.xml`, con `f.seek(0,2)` nos situamos al final del fichero, `f.tell()` nos dice la posición en la que estamos, que coincide con el número de bytes del fichero puesto que estamos al final del mismo. Con `f.seek(625)` nos situamos en la posición 625 del fichero. Leemos hasta final del fichero con `f.read()` y cerramos el fichero.

Ejemplos

Ejemplo 1.30.1. Escribamos un código que presente un fichero por líneas:

```
>>> with open('ata.txt') as f:
...     for line in f:
...         print line
...
ata

la

jaca

a

la

estaca

tres tristes tigres

>>>
```

Esto lo podemos hacer un script, digamos `cat.py`, dándole el siguiente contenido:

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

import sys

"""
    Ejemplo de uso:
    $python cat.py file
"""

if __name__ == "__main__":
    with open(sys.argv[1]) as f:
        for line in f:
            print line
```

Ejemplo 1.30.2. El objetivo ahora es abrir el contenido de una url y escribir todas las líneas que contenga una determinada palabra pasada como argumento. Para ello hacemos el script `SearchWordUrl.py` con el siguiente contenido:

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

import sys
import urllib2

if __name__ == "__main__":
    for line in urllib2.urlopen(sys.argv[2]):
        if sys.argv[1] in line:
            print line
```

que nos proporciona el siguiente diálogo:

```
equipo:last-of-the-track miUsuario$ python searchWordUrl.py UTC
http://tycho.usno.navy.mil/cgi-bin/timer.pl
<BR>Jul. 14, 17:42:11 UTC Universal Time
```



```
equipo:last-of-the-track miUsuario$
```

Ejemplo 1.30.3. Se trata de hacer un script en Python, digamos `splitFile.py`, que dado un fichero como argumento presente una lista por cada fila cuyas entradas sean las palabras de la fila. Demos a `splitFile.py` el siguiente contenido:

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

import sys

"""
    Ejemplo de uso:
        $python splitFile.py file
"""

if __name__ == "__main__":
    with open(sys.argv[1], 'r') as f:
        data = f.readlines()
        for line in data:
            words = line.split()
            print words
```

Ejemplo 1.30.4. Se trata de hacer un útil script en Python, digamos `convToUTF8.py`, que hará la labor de tomar el fichero que le pasemos como argumento y lo convertirá en otro con el mismo contenido pero codificado ya en UTF-8. La solución que traemos ahora es:

```
# -*- encoding: utf-8 -*-
#!/usr/bin/env python

# converting a unknown formatting file in utf-8

import codecs
import commands
import sys

file_location = sys.argv[1]
file_encoding = commands.getoutput('file -b --mime-encoding %s' % file_location)

file_stream = codecs.open(file_location, 'r', file_encoding)
file_output = codecs.open(file_location+"b", 'w', 'utf-8')

if __name__ == "__main__":
    for l in file_stream:
        file_output.write(l)

file_stream.close()
file_output.close()
```

Para verificar el funcionamiento de este script crearemos un fichero en texto llano que lleve acentos, digamos `golondrina.txt`, con las órdenes de consola de Linux le daremos el formato `iso-8859-1` —si no lo tiene ya— y procederemos a la conversión comprobando finalmente que `golondrina.txtb` tiene ya la codificación `utf-8`:

```
$ iconv -f UTF-8 -t ISO_8859-1 -o golondrinas_iso.txt golondrinas.txt
$ file -b --mime-encoding golondrinas_iso.txt
```

```
iso-8859-1
$ python convToUTF8.py golondrinas_iso.txt
$ file -b --mime-encoding golondrinas_iso.txtb
utf-8
```

Serialización

Tenemos una clara y completa explicación del significado de la serialización en [Mundo Geek](#)

Algunas veces tenemos la necesidad de guardar un objeto a disco para poder recuperarlo más tarde, o puede que nos sea necesario mandar un objeto a través de la red, a otro programa en Python ejecutándose en otra máquina.

Al proceso de transformar el estado de un objeto en un formato que se pueda almacenar, recuperar y transportar se le conoce con el nombre de *serialización* o *marshalling*.

En Python tenemos varios módulos que nos facilitan esta tarea, como `marshal`, `pickle`, `cPickle` y `shelve`.

El módulo `marshal` es el más básico y el más primitivo de los tres, y es que, de hecho, su propósito principal y su razón de ser no es el de serializar objetos, sino trabajar con bytecode Python (archivos `.pyc`).

`marshal` sólo permite serializar objetos simples (la mayoría de los tipos incluidos por defecto en Python), y no proporciona ningún tipo de mecanismo de seguridad ni comprobaciones frente a datos corruptos o mal formateados. Es más, el formato utilizado para guardar el bytecode (y por tanto el formato utilizado para guardar los objetos con `marshal`) puede cambiar entre versiones, por lo que no es adecuado para almacenar datos de larga duración.

`pickle`, por su parte, permite serializar casi cualquier objeto (objetos de tipos definidos por el usuario, colecciones que contienen colecciones, etc.) y cuenta con algunos mecanismos de seguridad básicos. Sin embargo, al ser más complejo que `marshal`, y, sobre todo, al estar escrito en Python en lugar de en C, como `marshal`, también es mucho más lento.

La solución, si la velocidad de la serialización es importante para nuestra aplicación, es utilizar `cPickle`, que no es más que es una implementación en C de `pickle`. `cPickle` es hasta 1000 veces más rápido que `pickle`, y prácticamente igual de rápido que `marshal`.

Si intentamos importar `cPickle` y se produce un error por algún motivo, se lanzará una excepción de tipo `ImportError`. Para utilizar `cPickle`, si está disponible, y `pickle` en caso contrario, podríamos usar un código similar al siguiente:

```
try:
    import cPickle as pickle
except ImportError:
    import pickle
```

as en un `import` sirve para importar el elemento seleccionado utilizando otro nombre indicado, en lugar de su nombre.

La forma más sencilla de serializar un objeto usando `pickle` es mediante una llamada a la función `dump` pasando como argumento el nombre del objeto a serializar y un objeto archivo en el que guardarlo (o cualquier otro tipo de objeto similar a un archivo, siempre que ofrezca métodos `read`, `readline` y `write`).

```
try:
    import cPickle as pickle
except ImportError:
    import pickle
```

```
fichero = file("datos.dat", "w")
animales = ["piton", "mono", "camello"]
```

```
pickle.dump(animales, fichero)
```

```
fichero.close()
```

La función `dump` también tiene un parámetro opcional `protocol` que indica el protocolo a utilizar al guardar. Por defecto su valor es 0, que utiliza formato texto y es el menos eficiente. El protocolo 1 es más eficiente que el 0, pero menos que el 2. Tanto el protocolo 1 como el 2 utilizan un formato binario para guardar los datos:

```
try:
```

```
    import cPickle as pickle
```

```
except ImportError:
```

```
    import pickle
```

```
fichero = file("datos.dat", "w")
```

```
animales = ["piton", "mono", "camello"]
```

```
pickle.dump(animales, fichero, 2)
```

```
fichero.close()
```

Para volver a cargar un objeto serializado se utiliza la función `load`, a la que se le pasa el archivo en el que se guardó:

```
try:
```

```
    import cPickle as pickle
```

```
except ImportError:
```

```
    import pickle
```

```
fichero = file("datos.dat", "w")
```

```
animales = ["piton", "mono", "camello"]
```

```
pickle.dump(animales, fichero)
```

```
fichero.close()
```

```
fichero = file("datos.dat")
```

```
animales2 = pickle.load(fichero)
```

```
print animales2
```

Supongamos ahora que queremos almacenar un par de listas en un fichero. Esto sería tan sencillo como llamar una vez a `dump` por cada lista, y llamar después una vez a `load` por cada lista:

```
fichero = file("datos.dat", "w")
```

```
animales = ["piton", "mono", "camello"]
```

```
lenguajes = ["python", "mono", "perl"]
```

```
pickle.dump(animales, fichero)
```

```
pickle.dump(lenguajes, fichero)
```

```
fichero = file("datos.dat")
```

```
animales2 = pickle.load(fichero)
```

```
lenguajes2 = pickle.load(fichero)
```

```
print animales2
```

```
print lenguajes2
```

Pero, ¿y si hubiéramos guardado 30 objetos y quisiéramos acceder al último de ellos? ¿o si no recordáramos en qué posición lo habíamos guardado? El módulo `shelve` extiende `pickle/cPickle` para proporcionar una forma de realizar la serialización más clara y sencilla, en la que podemos acceder a la versión serializada de un objeto mediante una cadena asociada, a través de una estructura parecida a un diccionario.

La única función que necesitamos conocer del módulo `shelve` es `open`, que cuenta con un parámetro `filename` mediante el que indicar la ruta a un archivo en el que guardar los objetos (en realidad se puede crear más de un archivo, con nombres basados en `filename`, pero esto es transparente al usuario).

La función `open` también cuenta con un parámetro opcional `protocol`, con el que especificar el protocolo que queremos que utilice `pickle` por debajo.

Como resultado de la llamada a `open` obtenemos un objeto `Shelf`, con el que podemos trabajar como si de un diccionario normal se tratase (a excepción de que las claves sólo pueden ser cadenas) para almacenar y recuperar nuestros objetos.

Como un diccionario cualquiera la clase `Shelf` cuenta con métodos `get`, `has_key`, `items`, `keys`, `values`, ...

Una vez hemos terminado de trabajar con el objeto `Shelf`, lo cerramos usando el método `close`.

```
import shelve

animales = ["piton", "mono", "camello"]
lenguajes = ["python", "mono", "perl"]

shelf = shelve.open("datos.dat")
shelf["primera"] = animales
shelf["segunda"] = lenguajes

print shelf["segunda"]

shelf.close()
```

1.31. Leyendo del Teclado

Nuestros programas serían de muy poca utilidad si no fueran capaces de interaccionar con el usuario. En ese círculo de ideas hemos visto el uso de la palabra clave `print` para mostrar mensajes en pantalla. Pero ¿cómo pedir información? Aprenderemos a utilizar el método `raw_input` para solicitar información.

La forma más sencilla de obtener información por parte del usuario es mediante el método `raw_input`. Este método toma como parámetro una cadena a usar como `prompt` (es decir, como texto a mostrar al usuario pidiendo la entrada) y devuelve una cadena con los caracteres introducidos por el usuario hasta que pulsó la tecla `intro`. Veamos un pequeño ejemplo:

```
nombre = raw_input("Como se llama usted? ")
print("Encantado, " + nombre)
```

Si necesitáramos un entero como entrada en lugar de una cadena, por ejemplo, podríamos utilizar el constructor `int` para convertir la cadena a entero, aunque sería conveniente tener en cuenta que puede lanzarse una excepción si lo que introduce el usuario no es un número:

```
try:
    edad = raw_input("¿Cuántos años tiene? ")
    dias = int(edad) * 365
    print ("Ha vivido " + str(dias) + " días, al menos")
except ValueError:
    print ("Eso no es un numero")
```

Podemos crear el fichero `juegoNaranja.py` con el siguiente contenido:

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

def jugar(intento=1):
    respuesta = raw_input("¿De qué color es una naranja? ")
    if respuesta != "naranja":
        if intento < 3:
            print ("\n ¡Falló! Inténtelo de nuevo en " + str(3-intento) + " intentos")
            intento += 1
            jugar(intento)
        else:
            print ("\n ¡Perdió!")
    else:
        print ("\n ¡Ganó!")

jugar()
```

Tendremos el siguiente diálogo:

```
equipo:last-of-the-track miUsuario$ python juego.py
¿De qué color es una naranja? blanca
```

```
    ¡Falló! Inténtelo de nuevo en 2 intentos
¿De qué color es una naranja? verde
```

```
    ¡Falló! Inténtelo de nuevo en 1 intentos
¿De qué color es una naranja? naranja
```

```
    ¡Ganó!
equipo:last-of-the-track miUsuario$
```

o este otro

```
equipo:last-of-the-track miUsuario$ python juego.py
¿De qué color es una naranja? blanca
```

```
    ¡Falló! Inténtelo de nuevo
¿De qué color es una naranja? roja
```

```
    ¡Falló! Inténtelo de nuevo
¿De qué color es una naranja? verde
```

```
    ¡Perdió!
equipo:last-of-the-track miUsuario$
```

El método `raw_input` sirve también para diseñar rudimentarios menús. El fichero `circle.py` contiene un ejemplo:

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

from math import pi

radio = float(raw_input('Deme el radio de un círculo: '))

option = ''
```

```

while option != 'd':
    print ('Escoja una opción: ')
    print ('a) Calcular el diámetro:')
    print ('b) Calcular el perímetro.')
    print ('c) Calcular el área.')
    print ('d) Finalizar.')

    option = raw_input('Teclee a o b o c o d y pulse "intro": ')

    if option == 'a':
        print 'El diámetro es: ', 2 * radio
    elif option == 'b':
        print 'El perímetro es: ', 2 * pi * radio
    elif option == 'c':
        print 'El área es: ', pi * radio ** 2
    elif option != 'd':
        print 'Sólo hay 3 opciones (a o b o c) y usted tecleó:', option

print ('Ha usado el la calculadora para el círculo')

```

que ofrece el siguiente diálogo:

```

equipo:last-of-the-track miUsuario$ python circle.py
Deme el radio de un círculo: 2
Escoja una opción:
a) Calcular el diámetro:
b) Calcular el perímetro.
c) Calcular el área.
Teclee a o b o c y pulse "intro": x
Sólo hay 3 opciones (a o b o c) y usted tecleó: x
equipo:last-of-the-track miUsuario$ python circle.py
Deme el radio de un círculo: 2.2
Escoja una opción:
a) Calcular el diámetro:
b) Calcular el perímetro.
c) Calcular el área.
Teclee a o b o c y pulse "intro": b
El perímetro es: 13.8230076758
equipo:last-of-the-track miUsuario$

```

1.32. Manejo de Excepciones

Las excepciones⁵ son errores detectados por Python durante la ejecución del programa. Cuando el intérprete se encuentra con una situación excepcional, como el intentar dividir un número entre 0 o el intentar acceder a un archivo que no existe, este genera o lanza una excepción, informando al usuario de que existe algún problema.

Si la excepción no se captura el flujo de ejecución se interrumpe y se muestra la información asociada a la excepción en la consola de forma que el programador pueda solucionar el problema.

Veamos un pequeño programa que lanzaría una excepción al intentar dividir 1 entre 0:

⁵Contenido tomado de [Mundo geek](#).

```
def division(a, b):  
    return a / b
```

```
def calcular():  
    division(1, 0)
```

```
calcular()
```

Si lo ejecutamos obtendremos el siguiente mensaje de error:

```
$ python ejemplo.py  
Traceback (most recent call last):  
File "ejemplo.py", line 7, in  
calcular()  
File "ejemplo.py", line 5, in calcular  
division(1, 0)  
File "ejemplo.py", line 2, in division  
a / b  
ZeroDivisionError: integer division or modulo by zero
```

Lo primero que se muestra es el trazado de pila o traceback, que consiste en una lista con las llamadas que provocaron la excepción. Como vemos en el trazado de pila, el error estuvo causado por la llamada a `calcular()` de la línea 7, que a su vez llama a `division(1, 0)` en la línea 5 y en última instancia por la ejecución de la sentencia `a/b` de la línea 2 de `division`.

A continuación vemos el tipo de la excepción, `ZeroDivionError`, junto a una descripción del error: `integer division or modulo by zero` (módulo o división entera entre cero).

En Python se utiliza una construcción `try-except` para capturar y tratar las excepciones. El bloque `try` (intentar) define el fragmento de código en el que creemos que podría producirse una excepción. El bloque `except` (excepción) permite indicar el tratamiento que se llevará a cabo de producirse dicha excepción. Muchas veces nuestro tratamiento de la excepción consistirá simplemente en imprimir un mensaje más amigable para el usuario, otras veces nos interesará registrar los errores y de vez en cuando podremos establecer una estrategia de resolución del problema.

En el siguiente ejemplo intentamos crear un objeto `f` de tipo fichero. De no existir el archivo pasado como parámetro, se lanza una excepción de tipo `IOError`, que capturamos gracias a nuestro `try-except`:

```
try:  
    f = file("archivo.txt")  
except:  
    print "El archivo no existe"
```

Python permite utilizar varios `except` para un solo bloque `try`, de forma que podamos dar un tratamiento distinto a la excepción dependiendo del tipo de excepción de la que se trate. Esto es una buena práctica, y es tan sencillo como indicar el nombre del tipo a continuación del `except`:

```
try:  
    num = int("3a")  
    print no_existe  
except NameError:  
    print "La variable no existe"  
except ValueError:  
    print "El valor no es un numero"
```

Cuando se lanza una excepción en el bloque `try`, se busca en cada una de las cláusulas `except` un manejador adecuado para el tipo de error que se produjo. En caso de que no se encuentre, se propaga la excepción.

Además podemos hacer que un mismo `except` sirva para tratar más de una excepción usando una tupla para listar los tipos de error que queremos que trate el bloque:

```
try:
    num = int("3a")
    print no_existe
except (NameError, ValueError):
    print "Ocurrio un error"
```

La construcción try-except puede contar además con una cláusula else, que define un fragmento de código a ejecutar sólo si no se ha producido ninguna excepción en el try:

```
try:
    num = 33
except:
    print "Hubo un error!"
else:
    print "Todo esta bien"
```

También existe una cláusula finally que se ejecuta siempre, se produzca o no una excepción. Esta cláusula se suele utilizar, entre otras cosas, para tareas de limpieza:

```
try:
    z = x / y
except ZeroDivisionError:
    print "Division por cero"
finally:
    print "Limpiando..."
```

También es interesante comentar que como programadores podemos crear y lanzar nuestras propias excepciones. Basta crear una clase que herede de Exception o cualquiera de sus hijas y lanzarla con raise:

```
class MiError(Exception):
    def __init__(self, valor):
        self.valor = valor

    def __str__(self):
        return "Error " + str(self.valor)

try:
    if resultado > 20:
        raise MiError(33)
except MiError, e:
    print e
```

Por último, a continuación se listan las excepciones disponibles por defecto, así como la clase de la que deriva cada una de ellas entre paréntesis:

- BaseException: Clase de la que heredan todas las excepciones.
- Exception(BaseException): Super clase de todas las excepciones que no sean de salida.
- GeneratorExit(Exception): Se pide que se salga de un generador.
- StandarError(Exception): Clase base para todas las excepciones que no tengan que ver con salir del intérprete.
- ArithmeticError(StandardError): Clase base para los errores aritméticos.
- FloatingPointError(ArithmeticError): Error en una operación de coma flotante.
- OverflowError(ArithmeticError): Resultado demasiado grande para poder representarse.

- `ZeroDivisionError(ArithmeticError)`: Lanzada cuando el segundo argumento de una operación de división o módulo era 0
- `AssertionError(StandardError)`: Falló la condición de un estamento `assert`.
- `AttributeError(StandardError)`: No se encontró el atributo.
- `EOFError(StandardError)`: Se intentó leer más allá del final de fichero.
- `EnvironmentError(StandardError)`: Clase padre de los errores relacionados con la entrada/salida.
- `IOError(EnvironmentError)`: Error en una operación de entrada/salida.
- `OSError(EnvironmentError)`: Error en una llamada a sistema.
- `WindowsError(OSError)`: Error en una llamada a sistema en Windows.
- `ImportError(StandardError)`: No se encuentra el módulo o el elemento del módulo que se quería importar.
- `LookupError(StandardError)`: Clase padre de los errores de acceso.
- `IndexError(LookupError)`: El índice de la secuencia está fuera del rango posible.
- `KeyError(LookupError)`: La clave no existe.
- `MemoryError(StandardError)`: No queda memoria suficiente.
- `NameError(StandardError)`: No se encontró ningún elemento con ese nombre.
- `UnboundLocalError(NameError)`: El nombre no está asociado a ninguna variable.
- `ReferenceError(StandardError)`: El objeto no tiene ninguna referencia fuerte apuntando hacia él.
- `RuntimeError(StandardError)`: Error en tiempo de ejecución no especificado.
- `NotImplementedError(RuntimeError)`: Ese método o función no está implementado.
- `SyntaxError(StandardError)`: Clase padre para los errores sintácticos.
- `IndentationError(SyntaxError)`: Error en la indentación del archivo.
- `TabError(IndentationError)`: Error debido a la mezcla de espacios y tabuladores.
- `SystemError(StandardError)`: Error interno del intérprete.
- `TypeError(StandardError)`: Tipo de argumento no apropiado.
- `ValueError(StandardError)`: Valor del argumento no apropiado.
- `UnicodeError(ValueError)`: Clase padre para los errores relacionados con unicode.
- `UnicodeDecodeError(UnicodeError)`: Error de decodificación unicode.
- `UnicodeEncodeError(UnicodeError)`: Error de codificación unicode.
- `UnicodeTranslateError(UnicodeError)`: Error de traducción unicode.
- `StopIteration(Exception)`: Se utiliza para indicar el final del iterador.
- `Warning(Exception)`: Clase padre para los avisos.
- `DeprecationWarning(Warning)`: Clase padre para avisos sobre características obsoletas.
- `FutureWarning(Warning)`: Aviso. La semántica de la construcción cambiará en un futuro.
- `ImportWarning(Warning)`: Aviso sobre posibles errores a la hora de importar.

- `PendingDeprecationWarning(Warning)`: Aviso sobre características que se marcarán como obsoletas en un futuro próximo.
- `RuntimeWarning(Warning)`: Aviso sobre comportamientos dudosos en tiempo de ejecución.
- `SyntaxWarning(Warning)`: Aviso sobre sintaxis dudosa.
- `UnicodeWarning(Warning)`: Aviso sobre problemas relacionados con Unicode, sobre todo con problemas de conversión.
- `UserWarning(Warning)`: Clase padre para avisos creados por el programador.
- `KeyboardInterrupt(BaseException)`: El programa fué interrumpido por el usuario.
- `SystemExit(BaseException)`: Petición del intérprete para terminar la ejecución.

Sesión Tercera

Leonardo Da Vinci

1. Ingeniar una forma de asignar a una variable el número primo de la **Figura 2.1(a)** y que nos libere de escribir 99 dígitos. Análoga petición para el primo de la **Figura 2.1(b)**.

(a) primo de 99 dígitos

(b) primo de 506 dígitos

2. Llamemos $1st$ a la lista de los dígitos del primo de 99 dígitos anterior y n al dígito 7. Proporcionar la lista de los índices de $1st$ con sus entradas igual a n .
3. Tomar una lista cualquiera, digamos $1st$, ordenarla suprimiendo repeticiones.

4. Considérese las listas `lst=['a',2,3,'xw','x',0,7,-1]` y `mst = ['ab',4,2,'x',9,2,'x',0,7]`. Hacer una operación en la consola de Python que nos proporcione ordenados los índices de las entradas en la lista `lst` que aparecen como entradas en la lista `mst`.
5. En una operación en la consola de Python, eliminar de una lista, digamos `lst=['a',2,3,'xw','x',0,7,-1]`, las entradas que ocurriesen al menos una vez en otra lista, digamos `mst = ['ab',4,2,'x',9,2,'x',0,7]`.
6. Conseguir dar la letra del DNI para el número 24121557.
7. Dar una impresión legible de un diccionario.
8. Usar lo conocido sobre diccionarios para suprimir las repeticiones de una lista. Generar a partir de la lista resultado, llamémosla `mst`, otra que esté ordenada, pero conservando `mst`. ¿Cuál es el significado de lo hecho?
9. Hacer un contador de frecuencias de caracteres en un string (texto) basado en diccionarios. Probarlo con el texto "ata la jaca a la estaca".

Capítulo 3

Sesión Cuarta

¡Que corra el código!

Yábir Benchakhtir

3.1. Ejercicios

1. Diseñar una función que tenga como único argumento un número (pensaremos que éste es de DNI) y que devuelva la letra que le correspondiese como número de DNI. Más aún, constrúyase una función que transforme el número del DNI en el NIF.
2. Diseñar una función, digamos `sortSr`, que tenga por argumento único una lista y devuelva otra que es la ordenada de su lista argumento pero suprimiendo las repeticiones.
3. Diseñar una función de dos argumentos el primero de los cuales sea una lista, digamos `lst`. Debe responder con la lista de los índices de `lst` correspondientes a las entradas de ésta iguales al segundo argumento.
4. Diseñar una función de dos argumentos y pensemos que son listas, digamos `lst` como primero y `mst` como segundo. La función debe presentar ordenados los índices de las entradas de `lst` que aparecen como entradas de `mst`.
5. Diseñar una función de dos argumentos y pensemos que son listas, digamos `lst` como primero y `mst` como segundo. La función debe borrar de `lst` cada ocurrencia de los elementos que aparecen en `mst`.
6. Diseñar una función booleana con dos argumentos, cada uno de ellos un string, que dé de resultado `True` sí, y sólo si, cualquiera de los string es una permutación del otro.
7. Diseñar una función que simule el funcionamiento de la orden `"".join(lst)`, donde `lst` es una instancia del tipo `list` con todas sus entradas del tipo `str`. Alterar la función para que convierta una lista de enteros en string.
8. Es conocido que dada una función $f: X \rightarrow Y$, la relación en X definida por aR_fb si, y sólo si, $f(a) = f(b)$ es una relación de equivalencia. Diseñar una función que dados conjuntos X e Y y una función $f: X \rightarrow Y$, construya el conjunto cociente X/R_f .
9. Escribir una función de un argumento entero positivo n que proporcione la lista de los números primos hasta n .
10. Escribir dos función que calculen la exponenciación rápida (mediante el *Método del Campesino Ruso*, [Sección 1.12](#)): una recursiva y otra que no lo sea.
11. Escribir una función que sume a un hito temporal (día, hora, minutos y segundos) un número arbitrario de nuevos hitos.

12. Escribir una función que aplique el *algoritmo de Euclides*. Extenderla luego para que calcule unos *coeficientes de Bezout*. Para fijar ideas, llamaremos `euclides(m,n)` a la función que implemente el Algoritmo de Euclides y `xeuclides(m,n)` a la función que implemente el Algoritmo de Euclides extendido.
13. Hacer una implementación para Python del *Criptosistema Afín* (cfr. [Apéndice D](#)).
14. Implementar la función `sign` que aplicada a cualquier número de 1, 0 ó -1 cuando éste sea respectivamente: positivo, cero o negativo.

Capítulo 4

Sesión Quinta

¡Don Michele, dammi un ordine!

El Padrino III

4.1. Ejercicios

1. Crear una clase *Calculadora* y los métodos suficientes para poder: sumar, restar, multiplicar y dividir.
2. Implementar la clase de los números racionales sujeto a las siguientes consideraciones:
 - a) Por defecto, el constructor debe representar a la fracción dada por su representante canónico.
 - b) Por defecto, ante un denominador igual a 1, el constructor debe ofrecer el numerador (esto permite representar los números enteros como fracciones)
 - c) Por defecto, el constructor debe dejar en el denominador un número natural no nulo.
 - d) Implementar la suma, resta, multiplicación y división por sobrecarga.
 - e) Implementar: un comparador de fracciones, un comprobador de no nulidad y una función signo.
3. Implementar una clase, digamos *Reloj*, que:
 - al crear un objeto cree una hora dada en horas, minutos y segundos inicializada por defecto a 0:0:0
 - prevea la forma de poder poner un objeto creado a una hora determinada que deseemos.
 - implemente un método *avance* para poder avanzar un segundo a una hora dada.
 - implemente los métodos necesarios para la presentación de los objetos.
4. Implementar una clase, digamos *Calendario*, que:
 - al crear un objeto cree una fecha dada en día, mes y año e inicializada por defecto a 1:1:1900
 - prevea la forma de poder poner un objeto creado a una fecha determinada que deseemos.
 - implemente un método para poder determinar si un año dado a partir de 1900 es bisiesto o no.
 - implemente un método *avance* para poder avanzar un día a una fecha dada.
 - implemente los métodos necesarios para la presentación de los objetos.
5. Implementar una clase, digamos *RelojCalendario*, que:
 - a) herede a las clases *Reloj* y *Calendario*
 - b) el constructor de tipo de la clase genere las instancias de tipo basándose en los generadores de tipo de las dos clases nombradas, con la salvedad de que si un tiempo (en horas, minutos y segundo) al ser ajustado en horas, minutos (<60) y segundos (<60) aportara 24 o más horas ($24 \leq \text{horas}$), produzca entonces el avance oportuno en la fecha.

Capítulo 5

Sesión Octava

No has de escribir: "Señor muerto esta tarde
llegamos" sino: "¡Señor, muerto está; tarde
llegamos!"

Hasnae García

5.1. Ejercicios

1. Hacer un script en Python, digamos `copyFile.py`, que sirva para copiar un fichero (no sólo de texto) en otro. Deberá tener dos argumentos: uno el fichero origen y otro el fichero destino.
2. Hacer un script en Python —digamos `countWords.py`— que, dado un fichero como argumento, presente el número de palabras de las que consta dicho fichero.
3. Hacer una función en un fichero, digamos `acertarNumero.py`, que al ser ejecutada nos permita acertar un número generado aleatoriamente, informando tras acertar del número de intentos empleados para ello.
4. Definir un script modificando el menú de la unidad didáctica, escrito en un fichero llamado por ejemplo `circleNew.py` que sea capaz de ofrecernos el siguiente diálogo:

```
equipo:last-of-the-track miUsuario$ python circle_new.py
Deme el radio de un círculo: x
no ha dado un valor convertible a número. Inténtelo de nuevo
Deme el radio de un círculo: 23.45
Escoja una opción:
a) Calcular el diámetro:
b) Calcular el perímetro.
c) Calcular el área.
d) Cambiar radio
e) Terminar.
Teclee a o b o c o d y pulse "intro": x
usted ha elegido la opción: x
Sólo hay 4 opciones (a o b o c o d) y usted tecleó: x
Escoja una opción:
a) Calcular el diámetro:
b) Calcular el perímetro.
c) Calcular el área.
d) Cambiar radio
e) Terminar.
Teclee a o b o c o d y pulse "intro": b
```

usted ha elegido la opción: b
El perímetro es: 147.340695453
Escoja una opción:
a) Calcular el diámetro:
b) Calcular el perímetro.
c) Calcular el área.
d) Cambiar radio
e) Terminar.
Teclee a o b o c o d y pulse "intro": d
usted ha elegido la opción: d
Deme el radio de un círculo: 1.3
Escoja una opción:
a) Calcular el diámetro:
b) Calcular el perímetro.
c) Calcular el área.
d) Cambiar radio
e) Terminar.
Teclee a o b o c o d y pulse "intro": a
usted ha elegido la opción: a
El diámetro es: 2.6
Escoja una opción:
a) Calcular el diámetro:
b) Calcular el perímetro.
c) Calcular el área.
d) Cambiar radio
e) Terminar.
Teclee a o b o c o d y pulse "intro": e
usted ha elegido la opción: e
Ha usado el la calculadora para el círculo
equipo:last-of-the-track miUsuario\$

Apéndice A

Complemento base-menos-uno

Es ampliamente conocido por los pedagogos que “se puede restar sumando”. Por supuesto que no se puede eliminar del esquema aritmético la resta, pero sí que se puede concebir de forma que no sea necesario el concepto de “acarreo”. Si el acarreo es difícil de entender y explicar para quien no se ha iniciado, ni inicia, en la aritmética vía el *ábaco*, restar a través del concepto de *complemento base-menos-uno* es un artificio cuando menos sofisticado, que termina siendo abandonado en el momento de aprender a dividir; de forma que por no dominar el uso del acarreo, puede que acabe el pupilo sin saber restar ni dividir, tal y como se está constatando en algunos alumnos al comienzo de la etapa universitaria: elegancia *versus* funcionalidad (o practicidad). ¿Los algoritmos idóneos para la máquina son siempre los más adecuados para la mente humana?

Definición A.0.1. Sea x un número natural expresado en base 10 y k un número natural no nulo y $n = \sum_{i=0}^k 9 \cdot 10^i$. El *complemento diez-menos-uno* sobre k , representado por $\sim_k x$, es por definición:

$$n - (x + 1) + 1$$

Si es sobreentendido el valor de n por el contexto del problema, a veces se escribe simplemente $\sim x$ y se representa por $-(x + 1)$.

Teorema A.0.1. Sean a y b números naturales no nulos tales que $b \leq a$. Entonces:

$$a - b = a + \sim_k (b - 1) - 10^k$$

donde $k = \lfloor \log a \rfloor + 1$.

Demostración. En efecto, se tiene que:

$$\begin{aligned} a - b &= a - b + 10^k - 10^k \\ &= a + \left(\left(\sum_{i=0}^k 9 \cdot 10^i \right) + 1 - b \right) - 10^k \\ &= a + \left(\left(\sum_{i=0}^k 9 \cdot 10^i \right) - b + 1 \right) - 10^k \\ &= a + \sim_k (b - 1) - 10^k \end{aligned}$$

□

Observación A.0.1. Por el **Teorema A.0.1** sabemos que la resta en base diez se puede sustituir por una suma y tres restas. La ventaja de ello es que las tres restas son sencillas de implementar pues la primera consiste en calcular el anterior de un número, la segunda en restar cifra a cifra sin acarreo y la tercera es olvidar el dígito más significativo —que es 1 siempre— de un número obtenido por suma. En el fondo, la gran ventaja de esta forma de restar es no tener que implementar el acarreo.

Ejemplo A.0.1. Calculemos la diferencia entre $a = 4328$ y $b = 231$. En este caso:

$$\begin{aligned} k &= \lfloor \log 4328 \rfloor + 1 \\ &= 3 + 1 \\ &= 4 \end{aligned}$$

así pues:

$$\begin{aligned} 4328 - 231 &= 4328 + \sim_4(230) - 10^4 \\ &= 4328 + (9999 - 231 + 1) - 10^4 && \text{la resta } 9999 - 231 \text{ es cifra a cifra, sin acarreo} \\ &= 4328 + (9768 + 1) - 10000 \\ &= 4328 + 9769 - 10000 \\ &= 14097 - 10000 && \text{olvidamos la cifra más significativa, que es 1} \\ &= 4097 \end{aligned}$$

Apéndice B

Diferencia entre `_`, `__` y `__xx__` en Python

Igor Sobreira

Al enseñar Python mucha gente no entiende realmente por qué tanto guión bajo en el principio de los métodos, a veces incluso en el final como `__esto__`. Yo ya he tenido que explicarlo muchas veces y es el momento de documentarlo.

Un guión bajo al principio

Python no tiene métodos privados reales, por lo que un guión bajo en el principio de un método o atributo significa que **no debe acceder a este método**, porque **no es parte de la API**.¹ Es muy común cuando se utilizan propiedades:

```
class BaseForm(StrAndUnicode):
    ...

    def _get_errors(self):
        "Returns an ErrorDict for the data provided for the form"
        if self._errors is None:
            self.full_clean()
        return self._errors

    errors = property(_get_errors)
```

Este retazo de código ha sido tomado de código fuente de django (`django/forms/forms.py`). Esto significa que `errors` es una propiedad y es parte de la API, pero el método de llamadas a esta propiedad, `_get_errors`, es “privado”, así que no debería acceder a él.

Dos guiones bajos al principio

Esto causa mucha confusión. Convendría no utilizarlo para marcar un método como privado, el objetivo aquí es **evitar que su método sea modificado (reescrito) por una subclase**. Veamos un ejemplo:

```
class A (objeto):
    def __method(self):
        print "I'm a method in A"

    def method(self):
        self.__method()
```

¹Interfaz de programación de aplicaciones.

```
a = A ()
a.method()
```

El diálogo aquí es:

```
$python example.py
I'm a method in A
```

Muy bien, como esperábamos. Ahora tengamos una subclase de A y personalizemos `__method`:

```
class B(A):
    def __method(self):
        print "I'm a method in B"
```

```
b = B()
b.method()
```

y ahora la salida es:

```
$ python example.py
I'm a method in A
```

como se puede ver, `A.method()` no llamó a `B.__method()`, como podríamos esperar. En realidad éste es el comportamiento correcto para `__`. Así que cuando se crea un método comenzando por `__` usted está diciendo que no quiere que nadie lo modifique (reescriba), será accesible sólo desde dentro de la propia clase.

¿Cómo lo hace Python? Simple, renombra el método. Eche un vistazo:

```
a = A()
a._A__method() # never use this!! please!
```

```
$ python example.py
I'm a method in A
```

Si intenta acceder a `a.__method`, no va a funcionar bien, sólo es accesible dentro de la propia clase como ya he dicho.²

Dos guiones bajos en el comienzo y dos al final

Cuando vea un método como `__esto__` la regla es simple: no lo invoque. ¿Por qué? Porque significa que es un método para que lo llame Python, no usted. Eche un vistazo:

```
>>> name = "igor"
>>> name.__len__()
4
>>> len(name)
4

>>> number = 10
>>> number.__add__(20)
30
>>> number + 20
30
```

Siempre hay un operador o función nativa que llama a estos *métodos mágicos*. La idea aquí es darle la capacidad de reescribir operadores en sus propias clases. A veces es sólo un enganche para llamadas de Python en situaciones específicas. `__init__()`, por ejemplo, se llama cuando el objeto es creado para que pueda inicializarlo. `__new__()` es llamado para construir la instancia, y así sucesivamente...

²El comentario de más arriba "never use this!! please!" no es del todo apropiado. A veces hay que acceder a esos atributos, sólo que hay que saber bien lo que se está haciendo. (N.T.)

He aquí un ejemplo:

```
class CrazyNumber(object):

    def __init__(self, n):
        self.n = n

    def __add__(self, other):
        return self.n - other

    def __sub__(self, other):
        return self.n + other

    def __str__(self):
        return str(self.n)
```

```
num = CrazyNumber(10)
print num          # 10
print num + 5      # 5
print num - 20     # 30
```

Otro ejemplo:

```
class Room(object):

    def __init__(self):
        self.people = []

    def add(self, person):
        self.people.append(person)

    def __len__(self):
        return len(self.people)

room = Room()
room.add("Igor")
print len(room)    # 1
```

La **documentación** cubre todos estos métodos especiales.

Conclusión

Utilice `_one_underline` para marcar sus métodos que no forman parte de la API. Utilice `__two_underlines__` al crear objetos que simulen a objetos nativos de Python o que ³quiere que sea personalizado su comportamiento en situaciones específicas. Y no use `__just_to_underlines`, a menos que realmente sepa lo que está haciendo.

³¿Se ha equivocado el autor escribiendo es "no" y ha de ser omitido en la lectura?. (N.T.)

Apéndice C

El Algoritmo de Euclides

En 1950, la palabra “algoritmo” era frecuentemente asociada con el *Algoritmo de Euclides*. Desde el siglo IV antes de *Jesús*, en los *Elementos* de *Euclides* se encuentra no sólo la geometría elemental, sino también las ideas básicas de la teoría de números. Las proposiciones 1 y 2 del libro VII de *Euclides* incluyen un ejemplo de algoritmo para determinar el máximo común divisor de dos enteros positivos y usando una eficiente técnica de resolución de un problema específico en un número finito de pasos. En este capítulo, el máximo común divisor de dos números enteros m y n se representa por (m, n) .

Lema C.0.1. Para todo $a, b \in \mathbb{Z}$ se cumple:

1. $(a, 0) = a$
2. Si $b \neq 0$, $(a, b) = (b, a \bmod b)$.

Teorema C.0.2. Sean $a \in \mathbb{Z}$ y $b \in \mathbb{Z}^*$. El algoritmo *EUCLIDES*, que aparece detallado en la figura C.1, calcula un máximo común divisor de a y b , así como aporta $u, v \in \mathbb{Z}$ tales que $d = ua + vb$.

input: $a \in \mathbb{N}$ y $b \in \mathbb{N}^*$.

output: El m.c.d. d de a y b , junto a $u, v \in \mathbb{Z}$ tales que $d = ua + vb$.

procedure EUCLIDES($a, b; d, u, v$)

begin

01.) $\langle a_0, a_1 \rangle \leftarrow \langle a, b \rangle$

02.) $\langle u_0, u_1 \rangle \leftarrow \langle 1, 0 \rangle$

03.) $\langle v_0, v_1 \rangle \leftarrow \langle 0, 1 \rangle$

04.) **while** $a_1 \neq 0$ **do**

05.) $q \leftarrow a_0 \text{ div } a_1$

06.) $\langle a_0, a_1 \rangle \leftarrow \langle a_1, a_0 - (q \cdot a_1) \rangle$

07.) $\langle u_0, u_1 \rangle \leftarrow \langle u_1, u_0 - (q \cdot u_1) \rangle$

08.) $\langle v_0, v_1 \rangle \leftarrow \langle v_1, v_0 - (q \cdot v_1) \rangle$

od

09.) $d \leftarrow a_0$

10.) $u \leftarrow u_0$

11.) $v \leftarrow v_0$

12.) **return** d, u y v .

end

Figura C.1: Algoritmo EUCLIDES para calcular el máximo común divisor de dos naturales no simultáneamente nulos, así como los coeficientes de Bezout.

Observación C.0.1. Al detallar el algoritmo de la figura C.1, muchos autores incluyen una rutina que hace que el divisor de la primera división computada sea el entero de menor valor absoluto. Una lectura atenta de nuestro algoritmo revelará que esta rutina está elegantemente implícita.

Observación C.0.2. Algunos autores han apuntado, y no les falta razón, que las líneas 3, 8 y 11 no son necesarias. En efecto, conociendo d y u , en la ecuación $d = ua + vb$ sólo hay una incógnita, que es v . Si damos por bueno el *teorema de Bezout*, y lo estamos dando, b debe dividir a $d - ua$, entonces $(d - ua) \div b$ debe ser v . Luego todo se podría arreglar con una división al final.

A continuación damos un ejemplo de cálculo del máximo común divisor y los coeficientes de Bezout siguiendo el algoritmo de la figura C.1.

Ejemplo C.0.1. Supongamos que deseamos calcular el máximo común divisor de $a = 228$ y $b = 612$. Entonces disponemos los cálculos en una tabla como sigue:

iteración	q	a_0	a_1	u_0	u_1	v_0	v_1
0	—	228	612	1	0	0	1
1	0	612	228	0	1	1	0
2	2	228	156	1	-2	0	1
3	1	156	72	-2	3	1	-1
4	2	72	12	3	-8	-1	3
5	6	12	0	-8	51	3	-19

El algoritmo aporta $d = (228, 612) = 12$, $u = -8$ y $v = 3$. Comprobar que, en efecto, $12 = -8 \cdot 228 + 3 \cdot 612$. En este caso, $[228, 612] = \frac{228 \cdot 612}{12} = 11628$

Apéndice D

Descripción del Cifrado Afín

Los criptosistemas afines son un ejemplo de cifrado monoalfabético por sustitución. El algoritmo parte de una biyección entre el alfabeto —para fijar ideas supondremos el alfabeto latino de 41 símbolos— y el segmento inicial de números naturales con 41 números. Ejemplo de ello está dado en la [Figura D.1](#).

Para cifrar, el trabajo consiste en asociar al número que representa a cada letra del texto llano otro número, que es su cifra, y presentarlo como la letra que corresponde a este nuevo número; de esta forma se obtiene el texto cifrado. La regla de transformación de números es la dada por la fórmula $C(x) = (ax + b) \pmod{N}$, donde: N es el número de letras del alfabeto (41 en este caso), a es un número natural tal que $0 \leq a \leq N-1$ cumpliendo $(a, N) = 1$ y b es un número natural tal que $0 \leq b \leq N-1$.¹

La función de descifrado sería $D(x) = (a'x + b') \pmod{N}$, donde $a' = a^{-1} \pmod{N}$ y $b' = -a^{-1}b \pmod{N}$. Si el número a elegido cumpliera $1 < (a, N)$, es fácil ver que más de un texto llano se podría cifrar en el mismo texto cifrado, de forma que no se podría recuperar de forma única el texto llano del cifrado.

0	g	7	n	14	u	21	1	28	8	35	
a	1	h	8	o	15	v	22	2	29	9	36
b	2	i	9	p	16	w	23	3	30	,	37
c	3	j	10	q	17	x	24	4	31	.	38
d	4	k	11	r	18	y	25	5	32	;	39
e	5	l	12	s	19	z	26	6	33	:	40
f	6	m	13	t	20	0	27	7	34		

Figura D.1: Enumeración del alfabeto.

Una vez calculadas estas llaves, para descifrar basta usar la función de cifrado con estas nuevas llaves.

Se expone a continuación un **ejemplo de Cifrado**. Si se escoge la llave de cifrado $\langle 13, 5 \rangle$ para cifrar con ella el recado:

el poder desgasta ... a quien no lo tiene

resulta el mensaje cifrado

2.eh9p27ep2fnrfsregggereu5:2wew9e.9es:2w2

Tienen especial interés estos dos casos particulares del sistema de cifrado afín:

1. $a = 1$; el cifrado afín da lugar en este caso a una mera traslación del alfabeto, el conocido como sistema de “Cifrado de Cesar” que hemos aludido y que era el utilizado por Provenzano.

¹La notación (m, n) se usa para representar el máximo común divisor de los números enteros m y n .

2. $b = 0$; el cifrado afín da lugar en este caso a una transformación lineal, nombre que indica que esta transformación lleva sumas en sumas: si C_i es el cifrado de P_i para $i = 1, 2$, entonces $C_1 + C_2$ es el cifrado de $P_1 + P_2$ (por supuesto, sumamos módulo N).

Apéndice E

Sobre el ábaco y ... los exponentes elevados.

Método del Campesino Ruso es el nombre que se le da a una forma de operar que, por ejemplo, permite multiplicar dos números mediante simples duplicaciones y una suma final o calcular potencias elevadas de números módulo un cierto valor positivo. Esencialmente, como pondremos de manifiesto, el secreto del método consiste en ver uno de los números —en el caso del producto— o el exponente —en el caso de las potencias— expresado en base dos. El Método del Campesino Ruso tiene una entrañable aplicación, a saber, la de poder multiplicar con ese curioso instrumento que conocemos con el nombre de *ábaco*.

La idea es que si n en base 2 tiene la expresión $(r_{k_n} \cdots r_1 r_0)_2$ entonces $n = r_0 2^0 + r_1 2^1 + \cdots + r_{k_n} 2^{k_n}$, donde $r_i \in \{0, 1\}$, para todo $0 \leq i \leq k_n$. Por tanto, para todo $m \in \mathbb{N}^*$,

$$mn = mr_0 2^0 + mr_1 2^1 + \cdots + mr_{k_n} 2^{k_n} \quad (\text{E.1})$$

y

$$\begin{aligned} m^n &= m^{r_0 2^0 + r_1 2^1 + \cdots + r_{k_n} 2^{k_n}} \\ &= m^{r_0 2^0} m^{r_1 2^1} \cdots m^{r_{k_n} 2^{k_n}} \end{aligned} \quad (\text{E.2})$$

Cuando encontremos un valor r_i que sea 0 no habrá necesidad de sumar $mr_i 2^i$ —en el caso de (E.1)— porque será 0 ni multiplicar $m^{r_i 2^i}$ —en el caso de (E.2)— porque será 1. Sin embargo, en esos casos será necesario efectuar las operaciones $m 2^i$ y m^{2^i} porque necesitaremos de tal resultado para la siguiente operación.

Todo lo dicho viene bellamente recogido en los teoremas E.0.1 y E.0.2.

Teorema E.0.1. Sean $x, y \in \mathbb{N}^*$ y sea la función $\xi(x, y)$ definida como sigue:

$$\begin{aligned} \xi(x, 1) &= x \\ \xi(x, y) &= \begin{cases} \xi(2x, \frac{y}{2}) & , \text{ si } y \text{ es par} \\ \xi(2x, \frac{y-1}{2}) + x & , \text{ si } y \text{ es impar} \end{cases} \end{aligned}$$

Entonces para todo $m, n \in \mathbb{N}^*$, $\xi(m, n) = mn$.

Observación E.0.1. En vista de que $\xi(m, n) = \xi(n, m)$ tomaremos como n el más pequeño entre los dos valores que deseamos multiplicar y ello nos conducirá a efectuar menos operaciones.

Teorema E.0.2. Sean $x, y \in \mathbb{N}^*$ y sea la función $\vartheta(x, y)$ definida como sigue:

$$\begin{aligned} \vartheta(x, 0) &= 1 \\ \vartheta(x, y) &= \begin{cases} \vartheta(x^2, \frac{y}{2}) & , \text{ si } y \text{ es par} \\ \vartheta(x^2, \frac{y-1}{2}) x & , \text{ si } y \text{ es impar} \end{cases} \end{aligned}$$

Entonces para todo $m, n \in \mathbb{N}^*$, $\vartheta(m, n) = m^n$.

Ejemplo E.0.1. La evaluación de $\xi(m, n)$ se organiza con una tabla como la que sigue para el ejemplo. Calculemos $478 \cdot 35$, para ello construimos la tabla

potencia	x	y	sumando
$2^0 \cdot 478$	478	35	478
$2^1 \cdot 478$	956	17	956
$2^2 \cdot 478$	1912	8	
$2^3 \cdot 478$	3824	4	
$2^4 \cdot 478$	7648	2	
$2^5 \cdot 478$	15296	1	15296
suma			16730

de lo cual resulta que $478 \cdot 35 = 16730$. Observar que la tabla responde al cómputo de $\xi(478, 35)$ y observar también que:

$$35 = 2 \cdot 17 + 1$$

$$17 = 2 \cdot 8 + 1$$

$$8 = 2 \cdot 4 + 0$$

$$4 = 2 \cdot 2 + 0$$

$$2 = 2 \cdot 1 + 0$$

$$1 = 2 \cdot 0 + 1$$

de donde, $35 = 1 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 0 \cdot 2^3 + 0 \cdot 2^4 + 1 \cdot 2^5$, o sea, 35 en base 2 es el número 100011. Por tanto,

$$\begin{aligned} 478 \cdot 35 &= 478 \cdot 1 \cdot 2^0 + 478 \cdot 1 \cdot 2^1 + 478 \cdot 0 \cdot 2^2 + 478 \cdot 0 \cdot 2^3 + 478 \cdot 0 \cdot 2^4 + 478 \cdot 1 \cdot 2^5 \\ &= 478 \cdot 2^0 + 478 \cdot 2^1 + 478 \cdot 2^5 \end{aligned}$$

Ejemplo E.0.2. En lo que sigue calculamos $12^{100} \pmod{34}$.

potencia	x	y	factor
$12^{2^0} \pmod{34}$	12	100	
$12^{2^1} \pmod{34}$	8	50	
$12^{2^2} \pmod{34}$	-4	25	-4
$12^{2^3} \pmod{34}$	16	12	
$12^{2^4} \pmod{34}$	-16	6	
$12^{2^5} \pmod{34}$	-16	3	-16
$12^{2^6} \pmod{34}$	-16	1	-16
producto			-1024

Por tanto $12^{100} \equiv -1024 \pmod{34}$ y como $-1024 \equiv 30 \pmod{34}$ resulta que $12^{100} \pmod{34} = 30$. Observar que la anterior tabla corresponde a la computación de $\vartheta(12, 100)$ reduciendo módulo 34. Observar también que

$$100 = 2 \cdot 50 + 0$$

$$50 = 2 \cdot 25 + 0$$

$$25 = 2 \cdot 12 + 1$$

$$12 = 2 \cdot 6 + 0$$

$$6 = 2 \cdot 3 + 0$$

$$3 = 2 \cdot 1 + 1$$

$$1 = 2 \cdot 0 + 1$$

de donde, $100 = 0 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^3 + 0 \cdot 2^4 + 1 \cdot 2^5 + 1 \cdot 2^6$, o sea, 100 en base 2 es el número 1100100. Por tanto,

$$\begin{aligned} 12^{100} \bmod 34 &= 12^{0 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^3 + 0 \cdot 2^4 + 1 \cdot 2^5 + 1 \cdot 2^6} \bmod 34 \\ &= 12^{1 \cdot 2^2 + 1 \cdot 2^5 + 1 \cdot 2^6} \bmod 34 \\ &= 12^4 \cdot 12^{32} \cdot 12^{64} \bmod 34 \\ &= (12^4 \bmod 34)(12^{32} \bmod 34)(12^{64} \bmod 34) \\ &= (-4)(-16)(-16) \bmod 34 \\ &= -1024 \bmod 34 \\ &= 30 \bmod 34 \end{aligned}$$

Referencias Web

- [Mundo geek](#)
- [Python Course](#)