Taller Python Documentation

Release 0.1

Juan Ignacio Rodriguez de Leon

23 de May de 2013

Índice general

Contents:

CAPÍTULO 1

Introducción al lenguaje Python

1.1 Breve historia de Python

El lenguaje **Python** fue creado por el Holandés **Guido Van Rossum** a finales de la decada de los 80. Van Rossum trabajaba entonces en el Centro para las Matemáticas y la Informática de los Países Bajos (CWI - *Centrum Wiskunde & Informatica*). En algún momento de 1989 empezó a crear un nuevo lenguaje que reemplazaría al lenguaje de programación ABC, que el mismo había ayudado a desarrollar.

Inicialmente la idea era quedarse con todas las buenas ideas de ABC, eliminar aquellas que demostraron no ser tan buenas, incorporar alguna de los conceptos que circulaban por los entornos académicos, como el manejo de excepciones para gestionar los errores y, sobre todo, añadir soporte para Amoeba, un sistema operativo distribuido desarrollado por el equipo de Andrew S. Tanenbaun para la *Vrije Universiteit* (en español: "Universidad Libre"), una universidad en Ámsterdam, Holanda [1].

A medida que se fue desarrollando, Python incorporaba (o "tomaba prestadas") características de otros lenguajes que parecían prometedoras o simplemente interesantes. Desde las primeras versiones ya se había incluido un sistema de módulos y una gestión de excepciones directamente inspiradas en el lenguaje Modula3 (Emparentando así con la larga tradición de lenguajes procedimentales encabezadas por Pascal), se le añadieron algunas primeras, o quizá primerizas, capacidades funcionales (Las utilidades *lambda*, *map*, *filter* y *reduce* se incorporaron desde la versión 1.0 [2]). Para la versión 1.4 ya se habían incluido el paso de parámetros por nombre (herencia de Modula3 e indirectamente de Common Lisp) y el soporte de serie para números complejos. Muchos de los problemas de gestión de memoria de lenguajes como C se eliminarían casi totalmente con la inclusión de un sistema de "recolección de basura", que liberaba al programador de las tediosas -y a menudo problemáticas- operaciones de reserva y liberación de memoria. Python 2.0 introdujo el concepto, "robado" de lenguajes funcionales como SETL y Haskell, llamado comprensión de listas (*list comprehension*) [3]. Python 2.2 añadio el concepto de generadores, inspirados por el lenguaje Icon.

1.2 Python 2.7 frente a Python 3.x

Las versiones de Python 3.0 y posteriores (colectivamente llamadas también Python 3000 o Py3K) tienen como principal objetivo corregir determinados errores o debilidades del lenguaje. El problema de una revisión en profundidad como esta es que no se puede mantener la compatibilidad hacia atrás. Esto está indicado, entre otras cosas, por el salto de numero en las versiones. El principio rector de esta actualización fue: "reducir la duplicación de capacidades

[1] La Vrije Universiteit es conocida también como «VU University».

[2] Segun Guido Van Rossum, por un agradecido hacker de Lisp que, al parecer, las echaba de menos y escribió un parche para incluirlas.

[3] La sintaxis de Python para esta estructura es prácticamente idéntica a la de Haskell, si no tomamos en cuenta la preferencia de Haskell por los signos de puntuación frente a la de Python por palabras claves alfebéticas.

eliminando las antiguas maneras de hacer las cosas", lo que deja bastante claro que, en algunas cosas al menos, el código antiguo tendría que ser modificado para que funcionara con el nuevo interprete. La última versión de la rama 2.x, la 2.7, está pensada como el paso natural entre las dos ramas; las versiones 2.7.x son capaces de ejecutar, con pequeños ajustes, tanto código 3.x como 2.x.

1.3 ¿Porqué Python?

Las caracterísiticas de Python lo hacen muy interesante y adecuado para diferentes situaciones: En el ámbito educativo, es un excelente "primer lenguaje", debido a su sintaxis elegante y a su facilidad de lectura (ABC, el lenguaje del que deriva, era un lenguaje destinado al aprendizaje). En el entorno científico, es apreciado por su claridad, potencia y por el gran número de librerías adicionales que han eclosionado a su alrededor. A los administradores de sistemas les permite escribir programas pequeños, adecuados para resolver problemas muy específicos, rápidamente y sin necesidad de fases de compilado adicionales. Es muy fácil de extender y de embeber, lo que lleva a encontrarnos el intérprete de Python en los lugares más insospechados: dentro de juegos para ordenador, como Eve Online, de programas de animación como Blender, o de procesado de imágenes, como Gimp.

En este apartado veremos con más detalles algunas de las características mas sobresalientes de Python.

1.3.1 Desarrollo rápido

Python es un lenguaje ideal para el desarrollo rápido, debido a sus características: Lenguaje interpretado, facilidad de lectura, libertad del programador, así como a la ingente cantidad de librerías disponibles. La velocidad en la que se puede tener desarrollado un prototipo funcional es enorme, comparada con lenguajes más "pesados": Compilados, difíciles de leer, restrictivos, etc...

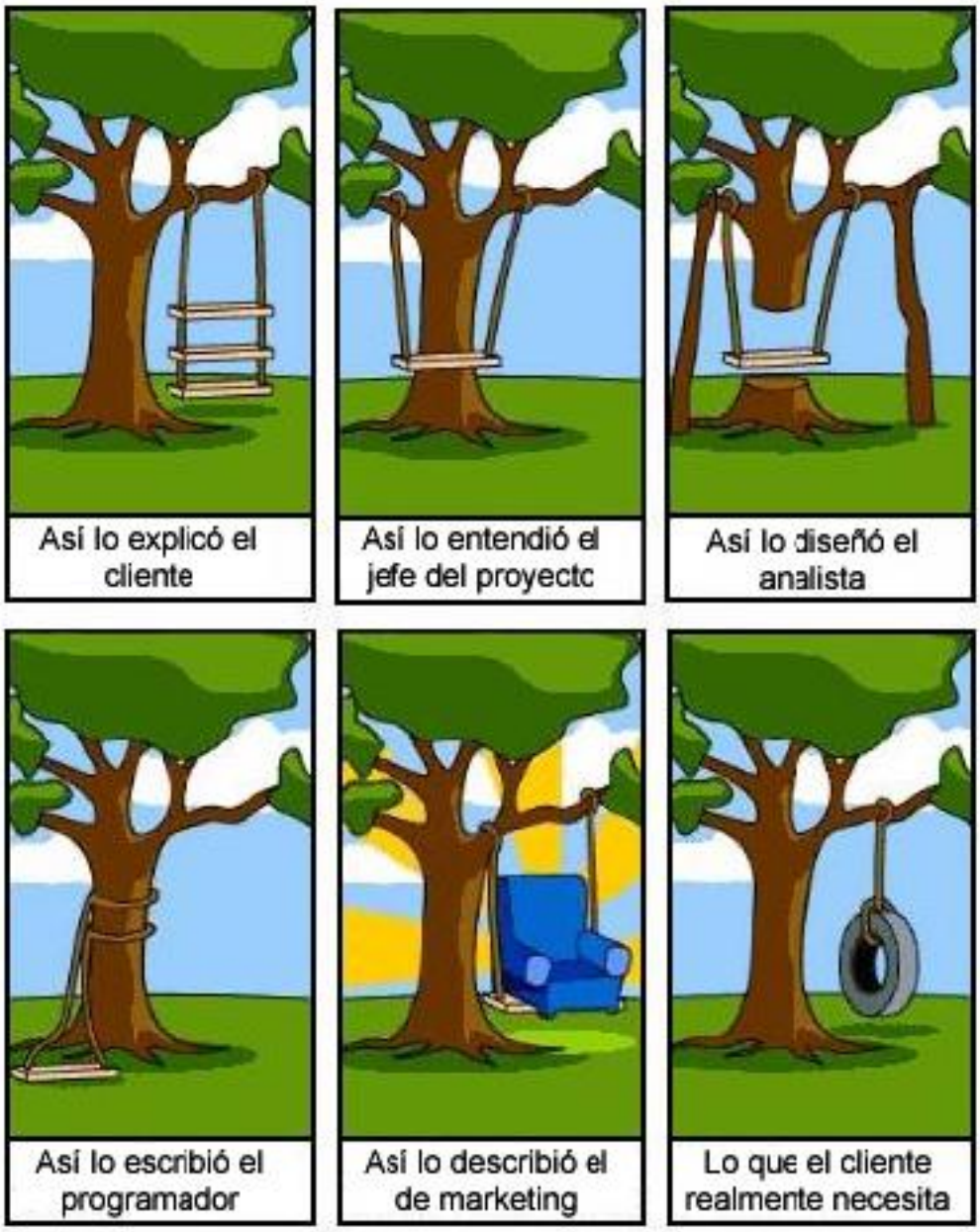
En muchos casos, es conveniente desarrollar un prototipo rápido, que nos permite a la vez aprender sobre el problema en cuestión y comprobar si nuestros diseños e ideas son adecuados.

> **Plan to throw one away. You will anyway**
>
> —Fred Brooks, The Mythical Man-Month

Una vez construido un prototipo, resulta más sencillo reescribir las partes críticas en un lenguaje más eficiente - normalmente C o C++, o incluso ensamblador, dependiendo de lo crítica que sea la velocidad de ejecución para el proyecto. Estas partes críticas resultan muy fáciles de identificar gracias a las herramientas incorporadas de análisis de rendimiento, y al principio de pareto o regla del 80-20.

En el peor de los casos, habría que reescribir todo el código usando un lenguaje más eficiente, como C, pero aun y así, es mucho más fácil programar teniendo como referencia un programa funcional [4] que unas especificaciones técnicas, no importa lo elaboradas que sean.

[4] Siempre y cuando dispongamos del código fuente, claro.



1.3.2 Sencillo (pero potente)

Python es un lenguaje de uso general, con el que somos capaces de escribir desde un sencillo *script* de 20 líneas a un complejo desarrollo de miles de líneas de código (Django, un potente *framework* de desarrollo web escrito en Python, consta de una 66.000 líneas de código). Sin embargo, esta potencia no proviene de una sintaxis oscura ni complicada, sino todo lo contrario.

Aunque la sintaxis de Python aun mantiene algún que otro rincón relativamente oscuro, estos rincones son pocos y normalmente bien señalizados. Por lo general se pueden evitar o aislar en unos pocos módulos o clases. Por supuesto, si uno se esfuerza es perfectamente posible escribir código python confuso y difícil de leer, pero la mayor parte del código es, por lo general, casi tan fácil de leer como el psuedo-codigo.

Eric S. Raymond, en el New Hacker's Dictionary, da la siguiente definición de *compacto*:

> **compacto** adj. Dicho de un diseño, describe la deseable propiedad de poder ser aprehendido de una vez en la cabeza de uno. por lo general, esto significa que los objetos creados a partir de dicho diseño tienden a ser más fáciles de usar y provocan menos errores que una herramienta equivalente que no sea compacta.
>
> El ser compacto no implica trivialidad o falta de potencia: Por ejemplo, el lenguaje C es más compacto que Fortran, y a la vez, más potente. Los diseños devienen no compactos a medida que se les agregan capacidades y lastres que no encajan de forma natural con el esquema general del diseño (De ahí que muchos fans del C clásico mentengan que el ANSI C ya no es compacto).

En este sentido, Python es muy compacto: consiste en unas pocas ideas, reutilizadas en múltiples sitios. Tomemos por ejemplo los espacios de nombres: Si importamos un nuevo módulo con *import math*, estamos creando un nuevo espacio de nombres llamado *math*. Las clases son espacios de nombres que comparten muchas de las propiedades de los módulos, y añaden unas pocas más. Las instancias de objetos son, de nuevo, espacios de nombres. Los espacios de nombres están implementados como diccionarios de python, así que tienen los mismos métodos que cualquier diccionario: *keys()* retorna una lista de claves, etc...

Python es simple porque es compacto, no porque sea limitado. No se debe subestimar la importancia de la simplicidad: no solo nos permite aprender el lenguaje más rápido, también nos permite luego escribir más rápido, corregir más rápido y cometer menos errores.

1.3.3 Fácil de leer

Normalmente el software se escribe una vez, pero se lee muchas, a medida que se revisa en busca de errores o se modifica. De ahí parte la filosofía de que, más importante que ser fácil de escribir, un lenguaje debe ser fácil de leer.

> **Programs must be written for people to read, and only incidentally for machines to execute.**
>
> —Abelson & Sussman, Structure and Interpretation of Computer Programs

A medida que fue evolucionando, muchas de las decisiones claves que ahora definen el lenguaje se tomaron principalmente para poder hacer el código más legible.

La mayor parte de la gente con experiencia en programación, e incluso muchas personas que nunca han programado, pueden leer, y en la mayoría de los casos interpretar correctamente, lo que hace una parte de código escrita en Python.

Como ejemplo, veamos el siguiente fragmento de código:

```
lista = [7, 23, 44, -2, 52]
suma = 0.0
for i in lista:
    suma = suma + i
m = suma/len(lista)

print("Promedio = ", m)
```

¿Puede adivinar la salida del programa?

> Promedio = 24.8

Otra forma de escribir el programa podría ser:

```
lista = [7, 23, 44, -2, 52]
print("Promedio = ", sum(lista)/len(lista))
```

1.3.4 Fácil de aprender

En python se ha logrado esta facilidad de aprender en parte gracias a su diseño compacto, y también son importantes todos los esfuerzos hechos para que fuera fácil de leer, como vimos antes. Pero quiza lo mas determinante es que

se puede aprender por niveles. Se puede empezar utilizando el paradigma de la programación imperativa, pasar a la programación orientada a objetos y seguir explorando las posibilidades de la programación funcional.

Esto se consigue porque, aunque incorpora la posiblidad de trabajar con clases y objetos, esto no es obligatorio. De igual manera, aunque incorpora muchos aspectos de programación funcional, no es necesario usarlos si no se desea. A medida que leemos y aprendemos más posibilidades, el lenguaje va creciendo con nosotros, por así decirlo.

1.3.5 No se entromete (entre tú y el problema)

Cuanto llegas a tener cuerta familiaridad con el lenguaje, te resulta muy fácil concentrarte en resolver un problema o en implementar un algoritmo, sin preocuparte por los detalles, a menudo enojosos, de la implementación.

Esto se debe en parte a la facilidad de leer el código, a su sintaxis compacta y a que las estructuras de datos y las de control se combinan entre si para ofrecer mucha más potencia. Mientras que en otros lenguajes necesitas crear estructuras de datos adicionales para dar soporte al problema, en Python a menudo descubres -a veces, con cierto placer culpable- que usando las predefinidas (tuplas, listas, conjuntos y, sobre todo, diccionarios) es más que suficiente.

Evidentemente, también ayuda la potencia de la librería estándar, aunque esta no sea, estrictamente hablando, una característica del lenguaje.

1.3.6 Interprete y modo interactivo

El *intérprete* interactivo de Python permite a los entudiantes probar las caracteristicas y posibilidades del lenguaje a la vez que programan. Es habitual mantener una ventana con el interprete funcionanado, y en otra el editor del código. Si en un momento dado no podemos recordar cuales eran los métodos disponibles para las listas, podemos recurrir al intérprete:

```
>>>
>>> l = [1, 2, 3, 5, 1, 5, 5]
>>> dir(l)
['append', 'count', 'extend', 'index', 'insert', 'pop', 'remove',
'reverse', 'sort']
>>> help(l.count)
Help on built-in function count(...)

L.count(value) -> integer -- return number of occurrences of value
>>> l.count(5)
3
```

Gracias al intérprete, la documentación siempre está accesible cuando programamamos.

1.3.7 "Hola, mundo" en Java

El programa Hola, Mundo es una convención usada a menudo para dar el primer paso cuando estamos aprendiendo un nuevo lenguaje de programación [5]. La idea es hacer el programa **más sencillo posible** que sea capaz de hacer algo; normalmente consiste en escribir el texto "hola mundo" en algún sitio. Aquí podemos encontrar versiones del hola, mundo para múltiples lenguajes.

Veamos el ejemplo típico del "hola, mundo" en Java:

```
class HolaMundo {
    public static void main (String args[]) {
        System.out.print("Hola, Mundo\n");
```

[5] Siempre había existido la constumbre de realizar pequeños programas de prueba o para el aprendizaje de nuevos lenguajes, pero la costumbre de formalizarlo y darle nombre corresponde a Brian Kernighan y Dennis Ritchie en su mítico libro "El lenguaje de programación C".

```
        }
    }
```

1.3.8 "Hola, mundo" en python

El código en Python es:

```
print("Hola, Mundo")
```

1.3.9 Interpretado (Pero también compilado)

Desde una terminal, si escribimos *python*, veremos que el programa mostrará información acerca de la versión y sistema python que tenemos instalado, tras lo cual imprime tres caracteres >>>

Python es un lenguaje compilado, en realidad, basado en una máquina virtual sobre la que se ejecuta el *bytecode*.

1.3.10 Diferentes paradigmas de programación

Lenguajes procedurales

Un lenguaje procedimental, como su nombre indica, hace incapíe en procesos (funciones o procedimientos) que cambian el estado del programa. Un programa imperativo es una secuencia de instrucciones que indican como se realiza una tarea.

Lenguajes como ensamblador o código máquina son por naturaleza imperativos, ya que reflejan la arquitectura de Máquinas de Turing. El estado del programa está almacenado en la memoria, y es modificado por la consecutivas instrucciones de bajo nivel.

Los lenguajes imperativos de más alto nivel utilizan el paradigma de la programación estructurada, cuyos objetivos son mejorar la claridad, calidad y tiempos de desarrollo mediante el uso de variables y sentencias más complejas, como subrutinas, bloques de código y el uso de bucles de tipo `for/while` y sentencias `if/else`, en vez de sentencias `goto`, considerada una mala práctica porque suelen derivar en código fuente confuso (*spaghetti code*), difícil de entender y de mantener.

Programación orientada a objetos

La programación orientada a objetos (A veces abreviada POO o OOP según sus siglas en inglés) es un paradigma de programación que se basa en objetos, y en las interacciones entre ellos, para diseñar programas informáticos. Permite el uso de abstracciones de más alto nivel, como herencia, encapsulación, polimorfismo, sobrecarga de operadores, abstracción, etc...

La mayoría de los conceptos de la programación orientada a objetos provienen del lenguaje Simula 67, un lenguaje diseñado en los años 60 principalmente para hacer simulaciones, como su nombre sugiere. Las ideas de Simula 67 influyeron sobre muchos lenguajes posteriores, incluyendo Smalltalk, derivados de LISP (CLOS), Object Pascal, y C++. El uso de este paradigma se popularizó a partir de la década de los 90. En la actualidad, existen múltiples lenguajes de programación que soportan la orientación a objetos.

Los objetos se caracterizan por mantener un determinado estado interno, por tener asociados ciertos comportamientos (métodos) y por tener una identidad única en el sistema. Así, un objeto contiene toda la información necesaria para definirlo y para identificarlo frente a otros objetos. Los mensajes permiten la comunicación y cooperación entre objetos. Varios objetos pueden compartir los mismos métodos, pudiendo agrupándose entonces en una misma clase, lo que permite la reutilización de código.

La programación estructurada tradicional y la POO se diferencian en en que, en la primera, datos y procedimientos son entidades separados y sin relación. La programación estructurada anima al programador a pensar primero en términos de procedimientos o funciones, y en segundo lugar en las estructuras de datos que esos procedimientos manejan. Los programadores que emplean POO, en cambio, definen objetos que se relacionan y coordinan entre si mediante mensajes. Cada objeto se considera una unidad indivisible, una amalgama de estado (datos) y comportamiento (métodos y mensajes).

Programación funcional

La programación funcional es un paradigma de programación declarativa basado en la utilización de funciones que no manejan datos mutables o de estado. Enfatiza la aplicación de funciones, en contraste con el estilo de programación imperativa, que enfatiza los cambios de estado. La programación funcional tiene sus raíces en el cálculo lambda, un sistema formal desarrollado en los 1930s para investigar la definición de función, la aplicación de las funciones y la recursión. Muchos lenguajes de programación funcionales pueden ser vistos como elaboraciones del cálculo lambda.

En la práctica, la diferencia entre una función matemática y la noción de "función" en programación imperativa radica en que las funciones imperativas pueden tener efectos secundarios, al cambiar el valor de cálculos realizados previamente. Se dice por tanto que carecen de **transparencia referencial**, es decir, que una llamada a una función podría devolver diferentes valores, dependiendo del estado del programa. Con código funcional, por el contrario, el valor generado por una función depende única y exclusivamente de los argumentos de la función. Al eliminar los efectos secundarios se puede entender y predecir el comportamiento de un programa mucho más fácilmente, y esta es una de las principales motivaciones para utilizar la programación funcional.

1.4 Para qué no es bueno Python

Python, por sus características, no es adecuado para los siguientes campos:

- Desarrollo a bajo nivel ("Cerca de la máquina"), como drivers, kernels o sobre hardware limitado. Al ser un lenguaje de alto nivel, no hay control directo sobre el hardware, depende de los servicios del S.O.
- Aplicaciones que requieran sobre todo alta capacidad de cómputo, o aquellas en que sea crítico obtener el máximo rendimiento posible; aquí la respuesta sigue siendo C o ensamblador.
- Las aplicaciones multi-hilo o multi-thread sobre sistemas con múltiples procesadores pueden tener problemas de rendimiento si no se tienen en cuenta las restricciones que impone el GIL (Global interpreter Lock).

1.5 Filosofía del lenguaje Python

Abrir el interprete de python y escribir *import this*:

The Zen of Python, by Tim Peters

- 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
- 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!

1.6 REPL o uso en línea de comandos

1.6.1 La función `help`

La función `help()`, si la llamamos sin ningúin parámetro, nos muestra una interfaz de ayuda interactiva, desde la que podemos interrogar al sistema por cualquier módulo, palabra clave o tema de interes sobre programación en Python.

Para salir del sistema de ayuda y volver al interprete, teclear la orden `quit`.

Si le pasamos un objeto, nos mostrará la información del objeto que pueda conseguir, incluyendo la documentación embebida, si se ha incluido, y la informaciñon que pueda obtener a partir de la función `dir()` (explicada a continuación).

1.6.2 la función `dir`

La función `dir` puede ser llamada sin parámetros o con un parámetro. Si se la llama sin parámetros, devuelve los nombres de variables accesibles desde el ámbito (*scope*) desde el que se llama:

```
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__']
>>> a = 3
['__builtins__', '__doc__', '__name__', '__package__', 'a']
>>> def f(x): x*x
...
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'a', 'f']
>>>
```

Si le pasamos un parámetros, devolverá una lista, en orden alfabético, de algunos de los atributos del objeto, o de atributos accesibles desde él. Su comportamiento por defecto es:

- Si le pasamos un módulo, devolverá una lista de los atributos definidos en el módulo.
- Si le pasamos una clase, devolverá una lista de sus atributos, más aquellos atributos definidos en las clases de las que herede.

- Para cualquier otro objeto, devolverá sus atributos, los atributos de su clase y los atributos definidos en las clases de las que herede su clase.

1.6.3 Entornos alternativos: IPython, bPython

Además del entorno REPL estándar de Python, existen varias versiones alternativas que añaden utilidades, interfaces o facilidades, manteniendo las capacidades del interprete Python. Entre los más interesantes cabe destacar IPython y bPython.

Día 1.- El Lenguaje de Programación Python

2.1 Nombres de variables

Los nombres de variables deben empezar con un caracter no numérico, el resto pueden ser letras, números y el caracter _. Python distingue entre mayúsculas y minúsculas, así que el nombre `a` es diferente de `A`.

Existen una serie de **palabras reservadas** por python, que no se pueden usar como nombres:

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

Además, hay ciertas variables y funciones "mágicas" usadas por python y que tienen significados especiales, estas son facilmente reconocibles porque siempre empiezan con dos caracteres _ y terminan igualmente por dos caracteres _. Ejemplos de estas variables son `__name__`, `__doc__` e `__init__`.

2.2 Tipos de datos simples

En Python hay tres tipos de datos básicos: Textos, números, y valores lógicos (Verdadero/Falso). Pero cada tipo básico es posible que se represente usando diferentes clases, por razones que veremos más adelante. Una definición de tipo la explicación de los valores que se pueden almacenar usando dichos tipos, así como los operadores que pueden ser usados con dichos valores.

2.2.1 Logicos (bool)

Los valores lógicos o *booleanos* son llamados así en honor a George Bool, inventor de la lógica booleana. Las variables booleanas solo pueden tener dos valores posibles: *True* (Verdadero) o *False* (Falso). Los operadores que trabajan con estos valores son *and*, *or* y *not*.

Los valores lógicos se utilizan sobre todo en condicionales. Cuando realizamos una comparación con uno de los operadores <, <=, >, >=, ==, != o <>, el resultado de la operación es un booleano. Si abrimos la shell de python y probamos a escribir simplmente:

```
>>> 7 > 2
True
>>> 2 <= -23
False
>>> 3 == 3
True
```

Existe una función *bool* que intenta convertir cualquier valor que se le pase a un valor booleano, siguiendo ciertas reglas, que se resumen en: la constante None, el número cero y las secuencias, tuplas, conjuntos y diccionarios vacíos se considera *False*. Cualquier otra cosa se considera True.

2.2.2 Números (int, long, float, decimal, complex)

En python, existen varias tipos de números. Cada uno de ellos almacena los valores de forma distinta, pero comparten el mismo conjunto de operadores, que es el que podemos esperar de cualquier numero:

Operador	Significado	Ejemplo
+	suma	12 + 23 -> 35
–	resta	12 - 6 -> 6
/	división	7 / 2 -> 3.5 (o 3 en Python 2.x)
//	division entera	7 / 2 -> 3
%	módulo	7 % 2 -> 1
*	multiplicación	2 * 10 -> 20
**	exponenciación	2 ** 10 <> 1024
&	AND a nivel de bits	26 & 1314 -> 2 (Solo para enteros)
|	OR a nivel de bits	1 | 2 -> 3 (Solo para enteros)
^	XOR a nivel de bits	3 ^ 5 -> 6 (Solo para enteros)

Tenemos los **números enteros**, que se corresponden con los números naturales, positivos y negativos. Una ventaja de Python sobre otros lenguajes es que no hay límite para el tamaño de los numeros enteros que podemos usar. Mejor dicho, el límite es la memoria RAM de la que dispongamos. Por ejemplo:

```
>>> 2**1024
179769313486231590772930519078902473361797697894230657273430081157732675805500963132708477322407536021120113879871393357658789768814416622492847430639474124377767893424865485276302219601246094119453082952085005768838150682342462881473913110540827237163350510684586298239947245938479716304835356329624224137216
```

Para crear una variable entera, solo hay que asignarle un valor que sea entero:

```
>>> a = 23
```

Ya que estamos, un valor se puede asignar a varias variables en una sola línea:

```
>>> a = b = c = d = 99
```

En python 2.7, había dos clases de enteros: la clase `Int` y la clase `Long`, y la conversión entre ambos tipos era automática. En python 3.0 hay un solo tipo. Es otro de los cambios introducidos para limpiar el polvo acumulado de los que hablaba antes.

Además de los enteros, tenemos también los **números reales** o en coma flotante, que en Python se corresponden con el tipo `float`. Para crear un numero en coma flotante solo tenemos que iniciar la variable con un valor que incluya

el punto decimal, o usar la notación cientifica `<coeficiente>e<exponente>`. Por ejemplo, todas las líneas siguientes crean variables en coma flotante:

```
>>> a = 23.0
>>> b = 3.141592653589793
>>> c = .23
>>> d = 1e-3
```

Abramos una terminal de python y probemos el siguiente código, pero antes de ejecutarlo, ¿Cuál creen que será el resultado, `True` o `False`?

```
>>> a = 0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1
>>> b = 1.0
>>> a == b
```

Implementación de float en Python

Python está escrito en C, e implemente los `float` con el tipo `double` o de doble precisión de C, es decir que los `float` utilizan 64 bits, siguiendo el estándar IEEE 754 para ese tamaño: 1 bit para signo, 11 para el exponente, y 52 para el coeficiente. Esto significa que los valores que podemos representar van desde $\pm 2{,}2250738585072020 \times 10^{-308}$ hasta $\pm 1{,}7976931348623157 \times 10^{308}$.

Python, como la mayoría de los lenguajes, utiliza el estándar de la IEEE para aritmética en coma flotante (IEEE 754). Con esta representación, algunos números no se pueden representar de forma exacta. Pasa lo mismo con la notación decimal: podemos representar con exactitud 1/4 como 0.25, pero 1/3 es 0.33333333333333... y así hasta el infinito. Como tenemos un espacio de memoria máximo para almacenar el número, esto implica que algunos de estos números se almacenarán con un error. Un error infinitesimal, pero error a fin de cuentas. El problema de estos errores viene cuando empezamos a acumular valores y, por tanto, a acumular errores. El valor 0.1 es uno de esos valores que no tiene representación exacta en IEEE 754, así que las repetidas sumas van acumulando el error. Si queremos saber el valor exacto del mismo podemos hacer:

```
>>> a = 0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1
>>> b = 1.0
>>> a - b
-1.1102230246251565e-16
```

Un valor insignificante, pero aun así suficiente para que los dos valores no sean iguales (1.0 si tiene representación exacta en IEEE 754)

Normalmente estos problemas se resuelven redondeando hasta la exactitud que necesitemos (por lo general muy por encima de 0.0000000000000001), pero Python introduce tambien un tipo de numero especial, **Decimal**, pensado especialmente para los calculos con un tamaño fijo decimal, normalmente esto significa que son los que usaremos para guardar información monetaria. Para poder usar estos números tenemos que hacer una operación especial antes, una **importación** de un módulo, que nos permite usarlos. La orden que realiza esta importación en concreto es:

```
from decimal import Decimal
```

Usando numeros decimales no incurrimos en ningún error de este tipo; hagamos la prueba anterior, pero ahora con números decimales:

```
>>> from decimal import Decimal
>>> a = Decimal('0.1')+Decimal('0.1')+Decimal('0.1')+Decimal('0.1') \
...     + Decimal('0.1')+Decimal('0.1')+Decimal('0.1')+Decimal('0.1') \
...     + Decimal('0.1')+Decimal('0.1')
>>> b = Decimal('1.0')
>>> a == b
True
```

Como curiosidad, comentar que, en modo interactivo, se crea una variable "mágica", _, que siempre contiene el resultado de la última expresion evaluada. Cuando usamos el interprete como calculadora, puede ser muy cómodo tener ese valor:

```
>>> impuestos = 17.5 / 100
>>> precio = 3.50
>>> precio * impuestos
0.6125
>>> precio + _
4.1125
```

Por último, Python incorpora también **números imaginarios** o **complejos**. Los números imaginarios se declaran añadiendo el sufijo `j` o `J` a la parte imaginaria. Para definir un número con parte real e imaginaria se usa la sintaxis (real+imagj), o bien los podemos crear usando la función `complex(real, imag)`:

```
>>> a = (3+4j)
>>> type(a)
>>> <type 'complex'>
```

La función `type` es muy util, le pasamos entre parentesis una variable y nos dice el tipo al que pertenece.

Se pueden extraer las partes reales e imaginarias de un número complejo `z` usando `z.real` y `z.imag`. La función `abs(z)` nos daría su magnitud:

```
>>> z = (3+4j)
>>> z.real
3.0
>>> z.imag
4.0
>>> abs(z)
5.0
```

2.2.3 Cadenas de Textos (string y unicode)

En Python las variables de texto se pueden expresar como literales usando comilla simples o dobles, indistintamente. Si tenenos la necesidad de incluir en el texto las propias comillas (simples o dobles, según corresponda) podemos hacerlo precediendo (En jerga de programadores, "escapando") a la comilla con el caracter \ (barra invertida). Si el texto ocupa varias líneas o queremos despreocuparnos de tener que escapar las comillas simples o dobles dentro del texto, podemos usar como delimitadores tanto tres comillas simples como tres comillas dobles. Por ejemplo, las siguientes declaraciones de variables tipo `String` son todas validas:

```
>>> a = 'Hola, mundo'
>>> b = 'It\'s seven o\'clock in the morning'
>>> c = "It's seventeen o'clock in the morning"
>>> d = "He said: \"Luke, I'm your father\""
>>> e = 'He said: "Luke, I\'m your father"'
>>> f = '''He said: "Frankly, my dear, I don't give a damn."'''
>>> g = """He said: "Frankly, my dear, I don't give a damn."""
>>> h = '''Vader: Obi-Wan never told you what happened to your father.
... Luke: He told me enough! He told me YOU killed him.
... Vader: No, I am your father.
... Luke: NOOOOOOOOOOOOOOOOOOooooo!!
... '''
```

En Python 2.7 y anteriores, se anteponía una `u` antes del delimitador del literal para indicar que el texto era unicode:

```
>>> # solo en python 2.7
>>> a = u'Cadena de texto unicode'
```

En Python 3.x, todas los literal son unicode por defecto, así que ya no tiene sentido esta notación; al contrario, tendremos que indicar cuando no es unicode, sino una string de texto codificado o códigos binarios, anteponiendo una b:

```
>>> solo en python 3.x
>>> a = b'Cadena de texto codificada. ¿Pero... con qué coded?'
```

Las operaciones que podemos hacer con las cadenas son muy variadas. Pueden ser unidas o concatenadas, con el operador +, y repetidas con *:

```
>>> saludo = "Hola," + "Mundo"
>>> saludo
'Hola,Mundo'
>>> linea = "-" * 22
>>> linea
'----------------------'
```

La función predeterminada `len` nos devuelve la longitud de una cadena, es decir, el número de caracteres:

```
>>> print(len('Hola, mundo'))
11
```

Las cadenas de texto permiten que se acceda a su contenido mediante índices, siendo el 0 la primera letra. Piensese en el índice no como una posición, sino como "El número de caracteres que hay antes del que me interesa". Ía forma de acceder es indicando el índice entre corchetes, después de la variable o string. Si usamos índices negativos, entonces la cuenta empieza desde la derecha, o sea, desde el final de la string:

```
>>> s = 'Con cien cañones por banda,'
>>> s[0]
'C'
>>> s[5]
'i'
>>> s[-1]
','
>>> s[0] != s[-1]
True
```

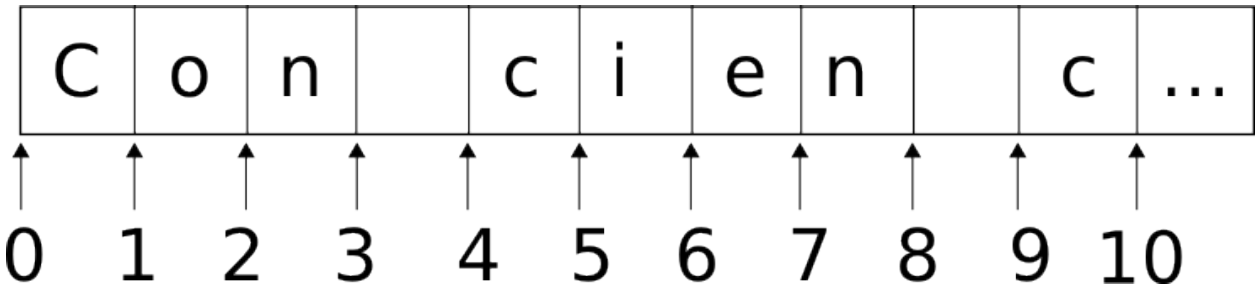
También podemos extraer subcadenas o **slices** a partir de una cadena mayor, usando la sintaxis [``<límite inferior>:<límite superior>]``. Si se omite el límite inferior, se supone un 0 (Es decir, desde el principio de la línea); si se omite el límite superior, se supone la longitud total de la cadena (es decir, hasta el final). Si se omiten los dos límites, obtendremos la cadena de texto original:

```
>>> s = 'Con cien cañones por banda,'
>>> s[0:3] # los  primeros tres caracteres
'Con'
>>> s[:8]  # los  primeros ocho caracteres
'Con cien'
>>> s[8:]  # todo, excepto los  primeros ocho caracteres
' cañones por banda,'
>>> s[4:8]
'cien'
>>> s[-6:]
'banda,'
>>> s2 = s[:]
>>> s == s2
True
```

Los excesos en los índices en estas operaciones se manejan con cierta indulgencia: Si un índice resulta demasiado grande, es reemplazado por la longitud de la cadena; si se especifica un límite inferior más grande que el límite superior se obtiene una string vacia:

```
>>> name = 'Anakin Skywalker'
>>> name[0:100]
'Anakin Skywalker'
>>> name[5:1]
''
```

Pera estas operaciones de "rebanado" o *slicing* resulta muy adecuada pensar que los índices apuntan a los espacios entre las letras, y no a las letras en sí, como se muestra en el siguiente gráfico:



no podemos modificar una parte de un texto usando estas expresiones, ni con índices ni con subcadenas:

```
>>> s = 'Con cien cañones por banda,'
>>> s[0] = 'P'
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> s[4:8] = 'doscientos'
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Pero si que podemos crear una nueva variable a partir de estas expresiones:

```
>>> s = 'Con cien cañones por banda,'
>>> s = s[:4] + 'doscientos' + s[8:]
>>> print(s)
'Con doscientos cañones por banda'
```

Esto es así porque las cadenas de texto, al igual que todos los tipos de datos que hemos visto hasta ahora, son **inmutables**. Esto significa que, una vez creada una variable de un tipo inmutable, esta *nunca* cambia de valor. Dentro de poco, veremos que hay tipos de datos que si son mutables; mientras tanto, lo que hay que explicar es: si las strings son inmutables, ¿Cómo es que el siguiente ejemplo no da error?

```
s1 = 'No por mucho madrugar'
s1 = s1 + ' amanece más temprano'
print(s1)
```

La respuesta es que Python crea una nueva string, uniendo las dos anteriores, y reasigna el nombre `s1` a la nueva String. Podemos comprobarlo usando la funcion `id()`, que nos devuelve un identificador de la variable que le pasemos; si dos objetos tienen el mismo identificador, entonces son en realidad el mismo objeto. Si probamos con esta segunda versión veremos que imprime dos números diferentes, es decir s1 apunta a una variable diferente la segunda vez:

```
s1 = 'No por mucho madrugar'
print(id(s1))
s1 = s1 + ' amanece más temprano'
print(id(s1))
print(s1)
```

2.2.4 None (TypeNone)

El valor especial **None** no es un tipo de dato, sino un valor constante especial, cuyo significado viene a ser "ausencia de valor" (Similar al valor especial `NULL` de SQL). Si una función no especifica un valor de retorno, este es *None*. *None* tiene su propio tipo de variable específica: *NoneType*.

Podemos comprobar si un valor es None con el operador `is`, o `is not`:

```
>>> a = 12
>>> a is None
False
>>> a is not None:
True
```

2.3 Tipos de datos compuestos

Los siguientes tipos de datos que veremos son llamados *compuestos*, porque sirven para agrupar distintas variables en una sola.

2.3.1 Listas

La lista es la estructura más habitual, consiste en una enumeración o lista de variables, siendo el orden de la misma importante, porque es precisamente usando ese orden como después podremos acceder a los valores individuales. En Python una lista se indica abriendo corchetes: `[`, a continuación las variables, valores o expresiones que formarán parte de la lista y acabamos cerrando los corchetes: `]`. Descrito así, parece mucho más complicado; veamos un ejemplo:

```
>>> a = ['Maria', 4, 723.4, None]
```

En otros lenguajes, la lista o `array` de datos solo puede contener un determinado tipo de datos; por ejemplo, una lista de enteros. En Python, como vemos, en la lista se pueden guardar cualquier tipo de datos.

Al igual que las cadenas de texto, las listas se acceden usando un índice, que de nuevo empieza por cero. También podemos usar "rebanadas" o *slices*, exactamente igual que con las strings, y tambien podemos concatenarlas usando el operador +:

```
>>> a = ['Maria', 4, 723.4, None]
>>> a[0]
'Maria'
>>> a[1:3]
[4, 723.4]
>>> a[-2:]
[723.4, None]
>>> a + [(6+7j), True]
['Maria', 4, 723.4, None, (6+7j), True
```

Tiene sentido, si pensamos que una cadena de textos al final viene a ser como una lista de caracteres.

Todas las operaciones de rebanado devuelven una nueva lista, que contiene los elementos indicados. Una forma habitual de obtener una copia de una lista es usando `[:]`; al omitir los límites inferior y superior estos son sustituidos por "desde el principio" y "hasta el final":

```
>>> a = ['Maria', 4, 723.4, None]
>>> b = a[:]
>>> a == b
True
```

```
>>> a is b
False
```

Pero, al contrario que las cadenas de texto, las listas si son *mutables*. Es posible cambier elementos individuales dentro de la lista:

```
>>> a = ['Maria', 4, 723.4, None]
>>> a[1] = a[1] + 6
```

Incluso es posible hacer aquello que no podiamos con las strings, asignar a una rodaja, aunque esto cambie el tamaño de la lista o incluso la deje totalmente vacia:

```
>>> a = [1,2,3,4]
>>> a[1:3] = [2.0, 2.1, 2.3, 2.5, 2.7, 2.9, 3.0]
>>> print(a)
[1, 2.0, 2.1, 2.3, 2.5, 2.7, 2.9, 3.0, 4]
>>> a[:] = [] # Borramos todo el contenido de la lista
>>> print(a)
[]
```

La función `len`, que en el caso de las cadenas de textos nos retornaba su longitud, aplicada a una lista nos devuelve el número de elementos de la lista.

```
>>> l = [1,2,3,4]
>>> print(len(l))
4
>>> s = '¡Es una trampa!'
>>> print(len(s))
16
```

Las listas pueden contener cualquier tipo de dato, no solo los datos simples que vimos al principio, tambien pueden contener otras listas, o tuplas, diccionarios (que veremos a continuación), etc... Por ejemplo podemos crear una matriz de 3x3 haciendo una lista de tres elementos, cada uno de los cuales es un una lista de tres elementos:

```
>>> a = [[1,2,3], [4,5,6], [7,8,9]]
>>> print(a[0][0])
1
>>> print(a[1][1])
5
```

Que las listas sean mutables es algo que no debemos olvidar, ya que puede causar muchos problemas en el programador principiante. Por ejemplo, dado el siguiente fragmento de código, ¿qué saldrá impreso por pantalla? ¿Por qué?:

```
a = [1,2,3,5]
b = a
b[3] = 'hola'
print(a)
```

Un ejemplo un poco más rebuscado. Podemos añadir un elamento a una lista usando el **método** append, de forma que:

```
>>> a = [1,2,3]
>>> a.append(4)
>>> print(a)
[1, 2, 3, 4]
```

Sabiendo esto, y dado el siguiente programa, ¿Cuál será la salida? ¿y por qué?:

```
q = ['a', 'b']
p = [1, q, 4]
```

```
q.append('extra')
print(p)
```

¿Qué operadores podemos usar con las listas? En primer lugar podemos compararlas con el operador ==; las listas a y b son iguales si lo son entre si cada uno de los elementos de que las componen:

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

Las comparaciones (<, <=, >, >= y !=) se realizan comparando los elementos en orden, a la primera discrepancia, se devuelve el resultado correspondiente. Si no se encuentra ninguna discrepacia, se considera que ambas secuencias son iguales:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 4]
>>> a > b
False
>>> a < b
True
```

Si una secuencia resulta ser la parte inicial de la otra, se considera que la secuencia más corta es la más pequeña. Si los elementos a comparar son cadenas de texto, se compararán caracter a caracter. Es legal comparar objetos de diferentes tipos, pero el resultado puede que le sorprenda: Los tipos se ordenan por su nombre (en inglés, claro). Por lo tanto, una *string* siempre será menor que una *tupla*, una *lista* siempre sera menor que una *string*, etc...

Tambien podemos sumar listas con el operador +:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print(c)
[1, 2, 3, 4, 5, 6]
```

Pero cuidado, el operador + siempre creará una nueva lista. Si queremos añadir una lista a la actual, sin crear una nueva, tenemos que usar una notación especial, +=:

```
>>> a = [1, 2, 3]
>>> a += [4, 5, 6]
>>> print(a)
[1, 2, 3, 4, 5, 6]
```

Aunque el resultado parezca el mismo, hay una sutil diferencia entre ampliar una lista o crear una nueva con el contenido ampliado, que puede causar problemas relativamente complejos. Por ejemplo, supongamos de nuevo la lista a:

```
>>> a = [1,2,3]
>>> b = a  # a y b son la misma lista
>>> a += [4]
>>> print b
[1, 2, 3, 4]
>>> # Es correcto, los cambios en a se reflejan en b, son la misma
>>> a is b
True
>>> a = a + [5]
>>> print(a) # 'a' es una nueva lista
[1, 2, 3, 4, 5]
>>> print(b) # 'b' sigue apuntado a la lista original
[1, 2, 3, 4]
```

```
>>> a is b
False
```

Evidentemente, si la lista es muy larga, es mucho más eficiente añadir un elemento a la lista que crear una nueva lista de cero, con el nuevo elemento añadido, más la sobrecarga de, probablemente, tener que liberar la memoria de las dos listas previas.

Las listas definen tambíen una serie de métodos, algunos de los cuales explicaremos brevemente aquí:

append(valor)

Añade el parámetro pasado, `valor`, al final de la lista.

count(valor) -> integer

Devuelve el número de veces que aparece `valor` en la lista.

extend(iterable)

Amplia la lista añadiendo los elementos que obtenga del iterable.

index(valor, [inicio, [final]]) -> integer

Retorna el índice de la primera aparición de `valor` en la lista; opcionalmente entre los límites indicados por `inicio` y `final`. Si `valor` no se encuentra en la lista, eleva una excepción de tipo `ValueError`.

insert(index, valor)

Inserta el valor en la lista en el índice `index`. `0` lo pondrá el primero de la lista. Tambien podemos usar el valor `-1` para ponerlo al final de la lista.

pop([index]) -> item

Extrae el elemento indicado por el índice `index`, (el último si de omite) de la lista y lo devuelve como resultado. Si la lista está vacia o el índice indicado cae fuera de rango, eleve una excepción de tipo `IndexError`.

remove(value)

Elimina la primera ocurrencia de `value` en la lista. Si `value` no está en la lista,e eleva una excepción de tipo `ValueError`.

reverse()

Invierte el orden la lista; el primero pasa a ser el último, el segundo pasa al antepenúltimo, etc... La operación modifica la lista actual, no crea una nueva.

sort(cmp=None, key=None, reverse=False)

Ordena los elementos de la lista, usando como criterio la función `cmp`, que debera aceptar dos parámetros y devolver -1, 0 o 1. Si no se especifica nada, se hara uso de la comparación normal. En ese caso, si la comparación no está definida, elevará una excapción de tipo `TypeError`. La operación modifica la lista actual, no crea una nueva.

Nota: Pilas y colas

Podemos usar la lista como una estructura de datos tipo pila o *stack* (LIFO: *Last In, First Out*) usando solo los métodos `append` y `pop` para introducir o extraer datos, o una estructura de datos cola o *queue* (FIFO: *First In, First Out*) si usamos solo `insert` (con index=0) y `pop`.

2.3.2 Tuplas

Hemos visto que las listas y los cadenas de texto tenian muchas cosas en comun, como el poder ser accedidas mediante índices y por rodajas. En realidad, hay más tipos de datos que comparten estas propiedades, todos ellos agrupados bajo el nombre genérico de **tipos de secuencias de datos**, como `bytes`, `bytearray` o el que nos ocupa ahora, tuplas (`tuple`). A medida que evoluciona Python, puede que se vayan añadiendo nuevos tipos de secuencia (De hecho, podemos escribir nuestros propios tipos de secuencia, tal y como veremos en la sección de programación orientada a objetos).

Las tuplas son, por tanto, un tipo de secuencia de dato. Se pueden escribir simplemente con una lista de valores separados con comas, o bien encerrandolas entre paréntesis. En general se recomienda, por claridad, incluir los paréntesis:

```
>>> t = 12.5, 9560 + 23, 'hola'
>>> t[0]
12.5
>>> t[1]
>>> 9583
>>> t
(12.5, 9583, 'hola')
>>> t == (12.5, 9560 + 23, 'hola')
True
```

Todo lo dicho para las listas es aplicable para las tuplas, con una diferencia: las tuplas no son mutables. Igual que con las *strings*, podemos crear nuevas tuplas cortando y rebanando de otras, pero no podemos modificar una determinada tupla una vez creada. Obsérvese que, aunque las tuplas sean inmutables, si que pueden contener en su interior objetos mutables, como una lista.

El interprete de python hace uso extensivo de las tuplas mediante un mecanismo llamada **empaquetado/desempaquetado automático de tuplas**. Esto es azucar sintáctico que nos permite expresiones como:

```
>>> a, b, c = 1, 2, 3
>>> print(a, b, c)
1 2 3
>>> a, b = b, a
>>> print(a, b)
2 1
```

Se plantea un problema con la construcción de tuplas vacias o con un único elemento: La sintaxis nos obliga a especificar una tupla vacia con `()`, lo cual parece bastante claro, pero para una tupla con un solo elemento debe incluir obligatoriamente una coma (no es suficiente con encerrar un valor entre paréntesis). La sintaxis es fea, hay que reconocerlo, pero funciona:

```
>>> vacia = ()
>>> singleton = ('hola',)     # <-- ojo a la coma
>>> len(vacia)
0
>>> len(singleton)
1
>>> singleton
('hola',)
```

Notese que la función `len` también funciona con las tuplas, y con un resultado equivalente al que producía para listas y cadenas de texto. Compacto.

También podemos comparar tuplas, siguiendo las mismas reglas que seguiamos para las listas. Las tuplas comparten métodos con las listas, pero no todos, ya que muchos no tienen sentido al ser las tuplas inmutables:

count(valor) -> integer

Devuelve el número de veces que aparece `valor` en la lista.

index(valor, [inicio, [final]]) -> integer

Retorna el índice de la primera aparición de `valor` en la lista; opcionalmente entre los límites indicados por `inicio` y `final`. Si `valor` no se encuentra en la lista, eleva una excepción de tipo `ValueError`.

2.3.3 Diccionarios

Los diccionarios son quizá la estructura de datos más potente de Python. Si solo pudieramos quedarnos con una, esta sería la prefererida de muchos.

Los dicionarios reciben también nombres como *memorias asociativas* o *arrays asociativos* en otros lenguajes. Es una estructura que nos permite almacenar valores, como lo es también una lista o una tupla, pero a diferencia de ellas, los valores que podemos usar para indexar -es decir, para acceder a los valores almacenados- no están limitados a un rango de números. Se accede a los contenidos de los diccionarios con claves o *keys*, que definimos nosotros a nuestro criterio. La unica condicion que tienen que tener un valor para poder ser usado como clave es que tiene que ser inmutable. Las cadenas de texto, como valores inmutables que son, resultan ideales para ser usadas como claves, pero también podemos usar enteros, tuplas (siempre y cuando contengan, a su vez, valores inmutables, claro), números complejos, etc...

Las listas y los propios diccionarios, al ser mutables, no pueden ser usadas como claves, pero si ser almacenadas dentro del diccionario.

La mejor manera de pensar en los diccionarios en como un montón de parejas clave: valor, donde las claves son únicas dentro del diccionario, y los valores pueden ser cualquier cosa. Podemos crear un diccionario vacio usando solo las llaves: `{}`. Si queremos inicializarlo con contenido, se añaden parejas con el formato `clave:valor`, separadas por comas, dentro de las llaves.

Veamos un ejemplo clásico, un diccionario que nos permite pasar de nombres de meses al número del mes:

```
>>> d = {
...     'enero': 1,
...     'febrero': 2,
...     'marzo': 3,
...     'abril': 4,
...     'mayo': 5,
...     'junio': 6,
...     'julio': 7,
...     'agosto': 8,
...     'septiembre': 9,
...     'octubre': 10,
...     'noviembre': 11,
...     'diciembre': 12,
...     }
>>> print('el mes de {} es el número {}'.format(
...     'octubre', d['octubre']
... ))
el mes de octubre es el número 10
```

Espero que no sorprenda a nadie la revelación de que la función `len()` también funciona con los diccionarios, y devuelve, por supuesto, el número de valores almacenados en el diccionario:

```
>>> print(len(d))
12
```

Las principales operaciones que podemos hacer con un diccionario son almacenar un valor con una determinada clave, o recuperar un valor a partir de la clave:

```
>>> d = {}
>>> d['hola'] = 'Mundo'
>>> print(d['hola'])
Mundo
```

Si se asigna un valor usando una clave que ya existe, se sobreescribe el valor nuevo y el antiguo, si no queda nada que lo referencia, desaparecerá. Si intentamos obtener un valor usando una clave que no existe en el diccionario, obtendremos una excepción de tipo `KeyError`.

Un método de uso habitual en los diccionarios es `keys()`, que devuelve una lista de todas las claves (En un orden sin determinar, lo que significa, en la práctica, en orden aleatorio). Tambien podemos determinar si una determinada clave existe en un diccionario usando la palabra reservada `in`. Siguiendo con el ejemplo de los meses:

```
>>> d.keys()
['marzo', 'abril', 'enero', 'febrero', 'noviembre',
'diciembre', 'mayo', 'octubre', 'julio', 'junio',
'septiembre', 'agosto']
>>> 'octubre' in d
True
>>> 'OCTUBRE' in d
False
```

Hay una función, `dict` que nos permite construir diccionarios de dos formas diferentes: o bien indicándole una secuencia de parejas clave, valor (por ejemplo, una lista de 2-tuplas):

```
>>> b = dict([
...     ('Jack Kirby', 1917),
...     ('Stan Lee', 1922),
...     ('Steve Ditko', 1927)
...     ])
>>> b['Stan Lee']
1922
```

O, si las claves son simples cadenas de texto, sin espacios, etc... puede ser más sencillo especificarlas mediante parámetros con nombre:

```
>>> n = dict(Au=79, Ag=47, Cu=29, Pb=82, Hg=89, Fe=26, H=1)
>>> print(n)
{'Pb': 82, 'Au': 79, 'Fe': 26, 'Ag': 47, 'H': 1, 'Hg': 89, 'Cu': 29}
```

Algunos de los métodos más importantes de los diccionarios son los siguientes:

clear()

Vacia el diccionario

get(key, [default_value]) -> item

Si `key` está en el diccionario, entonces devuelve el valor correspondiente, si no está, devuelve `default_value`, que por dede los mesesle `None`.

items() -> Lista de tuplas

Devuelve una lista de 2-tuplas, donde cada tupla esta constituida por una pareja clave, valor de cada entrada del diccionario.

keys() -> Lista

Devuelve una lista de todas las claves usadas en el diccionario.

pop(key, [default_value]) -> item

Devuelve el valor almacenado con la clave `key`, y borra la entrada del diccionario. Si `key` no está en el diccionario, devuelve el valor `default_value` si se ha especificado, si no, eleva la excepcion `KeyError`.

setdefault(key, [default_value]) -> item

Si `key` es una clave ya existente, entonces simplemente devuelve el valor que le corresponde. Si no, almacena `[default_value]` en la clave `key` y devuelve `[default_value]`. Nunca he podido comprender por qué no se llama `getdefault`.

update(d)

Actualiza el diccionario con los valores de `d`, que puede ser o bien otro diccionario, o un iterable que devuelve 2-tuplas, o bien pámetros por nombre.

values() -> List

Devuelve todos los valores almacenados en el diccionario.

2.3.4 Conjuntos (sets)

Los conjuntos son exactamente lo que su nombre indica: una implementación del concepto matemático de conjunto. Como tal, cumple con dos condiciones imprescindibles: sus elementos no mantienen ningún orden intrínseco, y no es posible que un elemento se repita dentro del conjunto. Las usos más habituales de los conjuntos son determinar si un objeto pertenece al conjunto o no, y eliminar duplicados.

Podemos crear un conjunto con la función `set`; normalmente le pasaremos una lista de elementos o un iterable a partir de los cuales crear el conjunto. Si hubiera duplicados en la lista, estos desaparecen en el conjunto. Para determinar si un valor pertenece a un conjunto podemos usar el operador `in`. La función `len`, como no podía ser de otra manera, también funciona con conjuntos:

```
>>> s = set(['a', 'e', 'i', 'o', 'u', 'a'])
>>> print(s)
>>> set(['a', 'i', 'e', 'u', 'o'])
>>> len(s)
5
>>> 'a' in s
True
>>> 'f' in s
False
```

Con los conjuntos es posible hacer cualquier operación del Algebra de Conjuntos: Unión, Intersección, Diferencia, Complemento y (con un poco de ayuda) Producto Cartesiano:

```
>>> a = set('PETER')
>>> a
set(['P', 'R', 'E', 'T'])
>>> b = set('PARKER')
>>> b
set(['A', 'P', 'K', 'R', 'E'])
>>> a - b # Letras en PETER, pero no en PARKER
set(['T'])
>>> b - a # Letras en PARKER, pero no en PETER
set(['A', 'K'])
>>> a | b # Letras en PETER o en PARKER (Unión)
set(['A', 'E', 'K', 'P', 'R', 'T'])
>>> a & b # Letras en PETER y en PARKER (Intersección)
set(['P', 'R', 'E'])
```

```
>>> a ^ b # Letras en PETER o en PARKER, pero no en los dos
set(['A', 'T', 'K'])
```

Para el producto tendremos que recurrir al módulo de la librería estándar `itertools`. No se preocupe si no se entiende por ahora:

```
>>> import itertools
>>> a = set('ABC')
>>> b = set('123')
>>> p = set(itertools.product(a, b))
>>> print(p)
set([('C', '1'), ('C', '2'), ('C', '3'),
     ('A', '2'), ('A', '3'), ('B', '3'),
     ('A', '1'), ('B', '2'), ('B', '1')])
```

Otros métodos interesantes de los conjuntos son los siguientes:

issubset(set) -> boolean

Indica si el conjunto es un subconjunto de otro mayor, que se pasa como parametro

issuperset(set) -> boolean

Indica si el el conjunto incluye al que se le pasa como parámetro.

isdisjoint(set) -> boolean

Indica si el subconjunto no tienen ningún elemento en común con el que se le pasa como parámetro.

2.4 Estructuras de control

2.4.1 if/else

Esta estructura de control seguramente es la más famosa y la más fácil de usar. Simplemente evalua una expresion, si el resultado es verdad (True) se ejecuta el bloque de código siguiente al `if`. Si es False, se ejecuta el bloque de código que sigue despues del `else`, si es que se ha incluido, ya que es opcional:

```
>>> if (7 > 3):
...     print('Siete es mayor que tres')
...     print('quien lo iba a pensar...')
... else:
...     print('Algo falla en las matemáticas...')
...
Siete es mayor que tres
quien lo iba a pensar...
>>>
```

Una cuestión importante es que tanto el bloque de código despues del `if` como el que va después del `else` está indentado [1]. Es es la forma que tiene Python de agrupar una serie de sentencias en un bloque. En otras palabras, la indentación, que en otros lenguajes es solo una opción estética destinada a mejorar la legibilidad, en Python si que tiene significado y es, por tanto, obligatoria.

En el modo interactivo, eso significa que tendremos que pulsar varias veces la barra de espacios o el tabulador, para cada línea dentro de un bloque. En la práctica, la mayoría de las veces escribiremos código Python usando algún editor

[1] Indentación: Anglicismo (de la palabra inglesa *indentation*) de uso común en informática que significa mover un bloque de texto hacia la derecha insertando espacios o tabuladores para separarlo del texto adyacente, lo que en el ámbito de la imprenta se ha denominado siempre como sangrado o sangría.

para programadores, todos los cuales tiene algún tipo de facilidad de auto-indentado. Otra pega del modo interactivo es que tendremos que indicar con una línea en blanco cuando hayamos acabado de introducir todas las líneas del bloque, ya que el analizador no tiene otra forma de saber si hemos acabado de introducir líneas o no.

Muchos programadores veteranos ven con cierto desagrado este aspecto de Python. En realidad, al poco de usarlo, la mayoría encuentra las ventajas de este sistema mucho mayores que las desventajas. Las ventajas son:

- El código es más legible y más corto.
- Permite reutilizar para otras funciones símbolos como `{` y `}`, usados en la mayoría de los lenguajes derivados de C, como C++, Java o C# para marcar el inicio y final de bloques, o reduce la lista de palabras reservadas del lenguaje, en casos como pascal (Donde se reservan las palabras BEGIN y END para esto).
- Evita ciertos casos de ambigüedad, donde la indentación indica una cosa pero el código realmente ejecuta otra. En estos casos o bien la identación es correcta, y el código esta mal, o viceversa. Ambos casos nos llevan a suponer que el código está haciendo una cosa, cuando realmente está haciendo otra. Este tipo de errores es relativamente frecuente, y difícil de detectar si hay muchos niveles de anidamiento. Con python no existe esta ambigüedad, ya que la unica referencia es el nivel de indentación.
- De todas formas, ibas a indentarlo.

Si queremos comprobar una serie de condiciones, y actuar de forma adecuada en cada caso, podemos encadenarlas usando la formula `if [elif ...] else`. `elif` es solo una forma abreviada de `else if`, apropiada para mantener la indentación de código a un nivel razonable.

Veamos un ejemplo. Importamos el modulo `random`, que nos permite trabajar con números aleatorios, y usamos su función `randint`, que nos devuelve un número al azar dentro del rango definido por los parámetros que le pasamos:

```
import random

n = random.randint(-10, 10)
print(n, 'es', end=' ')
if n == -10:
     print('el límite inferior')
elif -9 <= n < 0:
     print ('negativo')
elif n == 0:
     print('cero')
elif 0 < n <= 9:
     print ('positivo')
else:
    print('el límite superior')
```

En otros lenguajes se usa una estructura llamada `case` o `switch` para estas comprobaciones en serie, en python se prefiere la sintaxis de `if [elif...] [else]`. A nivel de rendimiento, no hay diferencia entre las dos sintaxis, ya quq ambas hacen exactamente lo mismo.

2.4.2 for

La estructura `for` nos permite repetir un trabajo varias veces. Su sintaxis es un poco diferente de la que podemos ver en otros lenguajes como Fortran o Pascal. En estos y otros lenguajes, esta construcción nos permite definir un rango de valores enteros, cuyos valores va adoptando sucesivamente una variable a cada vuelta o iteración dentro del bucle, y que a menudo se usan como índice para acceder a alguna estructura de datos.

En Python, por el contrario, el bucle `for`, está diseñado para que itera sobre cualesquiera estructuras de datos que sean "iterables"; por ejemplo, cadenas de texto, tuplas, listas o diccionarios. En cada vuelta o iteración tenemos en una variable, no el índice, sino el propio elemento dentro de la secuencia, en el orden correspondiente. Veamos algunos ejemplos:

```
>>> for letra in 'Texto':
...     print(letra)
T
e
x
t
o
>>> for word in ['Se', 'acerca', 'el', 'invierno']:
...      print(word, len(word))
Se 2
acerca 6
el 2
invierno 8
```

Como vemos, el bucle `for` funciona igual con una cadena de texto que con una lista, una tupla, etc... Repite el código en el bloque interno, tantas veces como elementos haya en la secuencia, asignado a una variable el elemento en cuestión. En el caso de iterar sobre un diccionario, la variable contendrá las distintas claves del mismo (En un orden indeterminado):

```
>>> casas = {
...     'Targaryen': 'Fuego y sangre',
...     'Stark': 'Se acerca el invierno',
...     'Baratheon': 'Nuestra es la Furia',
...     'Greyjoy': 'Nosotros no sembramos',
...     'Lannister': '¡Oye mi rugido!',
...     'Arryn': 'Tan alto como el honor',
...     'Martell': 'Nunca doblegado, Nunca roto',
...     }
>>> for clave in casas:
...      print('El lema de la casa', clave, 'es:', casas[clave])
El lema de la casa Arryn es: Tan alto como el honor
El lema de la casa Greyjoy es: Nosotros no sembramos
El lema de la casa Stark es: Se acerca el invierno
El lema de la casa Lannister es: ¡Oye mi rugido!
El lema de la casa Martell es: Nunca doblegado, Nunca roto
El lema de la casa Baratheon es: Nuestra es la Furia
El lema de la casa Targaryen es: Fuego y sangre
>>>
```

Si fuera necesario modificar la propia secuencia a medida que iteramos, por ejemplo para añadir o borrar elementos, es conveniente iterar sobre una copia; esto es muy fácil de hacer usando la notación de rodajas o *slices* `[:]`. Por ejemplo, el siguiente código borra las palabras de menos de tres letras de una lista:

```
>>> words = ['Se', 'acerca', 'el', 'invierno']
>>> for w in words[:]:
...     if len(w) < 4:
...          words.remove(w)
...
>>> print(words)
['acerca', 'invierno']
>>>
```

Si tenemos que iterar sobre un rango de números, la función predefinida `range` nos devuelve una secuencia iterable (En python 2.x nos devuelve una lista de números, en python 3.x devuelve una "lista virtual", que no consume tante memoria como la lista entera. Por ahora, podemos considerar ambas formas equivalentes, ya que son iterables las dos). La función `range` acepta entre uno y tres parámetros. Si solo se especifica uno, devuelve el rango empezando en 0 y acabando justo antes de llegar al valor del parámetro:

```
>>> for i in range(4): print(i)
...
0
1
2
3
>>>
```

Estos valores, `[0..n-1]` se corresponden exactamente con los índices válidos para una secuencia de `n` valores.

Si se le indican dos valores, devolverá el rango comprendido entre el primero y el inmediatamente anterior al segundo:

```
>>> for i in range(2, 5): print(i)
...
2
3
4
```

Si se indica un tercer parámetro, este se usará como incremento (`step`), en vez del valor por defecto, 1:

```
>>> for i in range(600, 1001, 100): print(i)
...
600
700
800
900
1000
```

Observese que, en todos los casos, el límite superior nunca se alcanza.

Si tenemos experiencia en otros lenguajes de programación, podemos sentir la tentación de seguir usando índices, y tirar de la función `range` cada vez que hagamos un `for`, quiza para sentirnos más cómodos, quizá por el temor de que en un futuro tengamos necesidad del índice. Es decir, en vez de hacer:

```
>>> for letra in 'ABCD':
...     print(letra, end=', ')
A, B, C, D,
```

Podríamos hacer:

```
>>> word = 'ABCD'
>>> for i in range(len(word)):
...     letra = word[i]
...     print(letra, end=', ')
A, B, C, D,
```

Esto no es nada recomendable, por varias razones:

- Es más difícil de leer
- Es más largo de escribir
- Creamos variables que no son necesarias (i)
- Es más lento (El recorrido del bucle `for` está optimizado en C)

¿Qué pasa si, por la razón que sea, necesito la variable `i`? ¿Es recomendable en ese caso usar la forma anterior? La respuesta es de nuevo no, existe una función, `enumerate`, que admite como parámetro cualquier secuencia y nos devuelve 2-tuplas, con el índice y el elemento de la secuencia:

```
>>> for i, letra in enumerate('ABCD'):
...     print(i, letra)
```

```
0 A
1 B
2 C
3 D
>>>
```

2.4.3 while

La sentencia `while` también nos permite ejecutar varias veces un bloque de código, pero en este caso se mantiene la repetición hasta que una determinada condición pasa a ser falsa. Veamos el siguiente ejemplo, que imprime el mayor de los números factoriales que sea menor que un millón:

```
acc = num = 1
while acc * (num+1) < 1000000:
    num = num + 1
    acc = num * acc

print('El mayor factorial menor que 1E6 es: ', num, '! = ', acc, sep='')
```

cuya salida es:

```
El mayor factorial menor que 1E6 es: 10! = 362880
```

El bucle se detiene cuando llegamos a 10, porque el factorial, 10! = 3.628.800, es mayor que un millón y, por tanto, la condición del `while` pasa a ser falsa (y hemos encontrado el número que buscábamos).

En casos como este, en que no sabemos a priori cuando debemos parar, la sentencia `while` encaja perfectamente. Podemos usar un bucle `while` para iterar cuando sepamos de antemano la cantidad de vueltas que tenemos que dar, pero parece más natural en ese caso usar el bucle `for`.

El error más común con un bucle de este tipo es olvidarnos de actualizar, dentro del código del bucle, la variable que es testeada en la condición `while`, lo que nos lleva a un bucle sin fin.

2.4.4 break, continue y else en bucles

La sentencia `break` fuerza la salida del bucle `for` o `while` en la que se encuentre (Si hay varios bucles anidados, solo saldrá del más interno). Esto puede ser muy útil para optimizar nuestro código. Por ejemplo, si estamos buscando dentro de una lista de números uno que sea múltiplo de 7, y simplemente nos interesa encontrar uno, el primero que encuentre, no tiene sentido seguir recorriendo el bucle hasta el final; podriamos hacer entonces:

```
>>> numeros = [15, 53, 98, 36, 48, 52, 27, 4, 29, 94, 13, 36]
>>> for n in numeros:
...       if n % 7 == 0:
...           print(n, 'es múltiplo de 7')
...           break
...
98 es múltiplo de 7
>>>
```

Los bucles en Python (tanto `for` como `while`) tienen una característica relativamente poco conocida, y es que dispone de una clausula `else`: El bloque de código especificado en el `else` solo se ejecuta si se cumplen estas dos condiciones:

el bucle ha llegado hasta el final

y no se ha salido de él mediante una clausula `break`.

En el caso de `for`, la cláusula `else`, si se ha especificado, solo se ejecutará si se han agotado todos los elementos de la secuencia y no se ha ejecutado ningún `break`. Por ejemplo, el código anterior no muestra ningún mensaje si no hubiera ningún número múltiplo de siete en la lista. Podemos arreglarlo así:

```
>>> numeros = [87, 39, 85, 72, 41, 95, 93, 65, 26, 11, 32, 17]
>>> for n in numeros:
...     if n % 7 == 0:
...         print(n, 'es múltiplo de 7')
...         break
... else:
...     print ('No hay ningún múltiplo de 7 en la lista')
...
No hay ningún múltiplo de 7 en la lista
```

Nota: Ejercicio

Usando la cláusula `else`, o la cláusula `break`, modificar el programa de cálculo de factoriales mostrado anteriormente para que muestre el **primer** factorial mayor que un millón (El mínimo factorial que sea mayor que 10^6).

2.5 Funciones

Una función no es más que un fragmento de código que queremos reutilizar. Para ello le damos un nombre que nos sirva para identificarla. También definimos unos nombres para las variables que servirán para pasar información a la función, si es que se le pasa alguna. Estas variables se llaman **parámetros** de la función.

Veamos con un ejemplo, una función que nos da el perímetro de una circuferencia, pasándole el radio de la misma:

```
import math

def perimetro(r):
    """Devuelve el perímetro de una circunferencia de radio r.
    """
    return 2 * math.pi * r
```

la palabra reservada `def` nos permite definir funciones. Después del `def` viene el nombre que le queremos dar a la función y luego, entre paréntesis, el parámetro o parámetros de entrada de la función, separados por comas. Si no hubiera ningún pafametro, aún así hemos de incluir los paréntesis. Finalmente, viene el signo de dos puntos. Todo el código que aparezca a continuación indentado a un nivel mayor que la palabra `def` forma parte del cuerpo o bloque de la función. Ahora podemos llamar o invocar a esta función desde cualquier parte de nuestro programa, simplemente usando su nombre seguido de los parámetros, entre paréntesis. Por ejemplo, el siguiente código imprime el resultado de calcular el perímetro de una circunferencia de radio 7:

```
import math

def perimetro(r):
    """Devuelve el perímetro de una circunferencia de radio r.
    """
    return 2 * math.pi * r

radio = 6
print('El perímetro de una circunferencia de radio', radio, 'es:', perimetro(radio))
```

cuyo resultado es:

```
El perímetro de una circunferencia de radio 6 es: 37.6991118431
```

Paso por referencia o por valor

Para el que esté interesado en las clasificaciones académicas, el paso de parámetros en Python no es ni por referencia, ní por valor. Para los que no, ignoren por favor esta nota con toda tranquilidad.
En Python, no podemos usar los términos *paso por valor* ni *paso por referencia* con el significado que tienen habitualmente.
En concreto, no se puede hablar de *paso por valor*, porque el código de la función puede, en determinados casos, modificar el valor de la variable que ve el código llamante.
Tampoco se le puede hablar de *paso por referencia*, porque no se le da acceso a las variables del llamador, sino solo a determinados objetos compartidos entre el código llamador y el código llamado. Pero el código llamado no tienen acceso a los nombres definidos en el espacio de nombres del llamador.
Este nuevo sistema se le conoce por varios nombres: Por objetos, compartido, o por referencia de objetos. Para quien esté interesado en los detalles, el siguiente enlace es un buen principio: Call by Object.

La primera línea dentro de la definición de la función puede ser, opcionalmente, una cadena de texto. Este texto no tiene ningún efecto sobre el código, por lo que se puede considerar como un comentario, pero internamente ese texto se convierte en la documentación interna de la función. Es esta documentación interna (abreviada a `docstring`) la que muestra la función `help()`, ya que puede ser accedida en tiempo de ejecución. Es muy recomendable incluir este tipo de documentación, especificando al menos los parámetros que acepta y el resultado que devuelve.

La sentencia `return` permite especificar el valor o valores que devuelve la función. Si no se especifica ningun valor de retorno, aun así la función retornará un valor, el aburrido `None`.

El paso de parámetros también es interesante, y ofrece posibilidades en python que no todos los lenguajes soportan.

La forma habitual de asignación de parámetros es por posición, es decir, cuando llamemos a una funcion definida con varios parámetros, el primer dato que pongamos tras los paréntesis ocupará el lugar del primer parámetro, el segundo valor ocupará el segundo parámetro y así sucesivamente. En estos casos, hay que pasar tantos valores como parámetros se hayan definido en la función.

Python soporta también el paso de parámetros por nombre y los parámetros por defecto, el paso de parámetros por nombre y el paso de argumentos en número variable.

2.5.1 Parámetros con valores por defecto

Lo más habitual es especificar un valor por defecto a uno o varios de los parámetros. De este forma, la función puede ser llamada con menos parámetros de los que realmente soporta.

Por ejemplo, la siguiente función devuelve el texto que se le pasa, resaltándolo con una linea antes y otra después, compuesta de tantos caracteres como mida el texto, usando el carácter definido como segundo parámetro. Podemos no especificar el caracter, con lo que por defecto se usará un guión:

```
def resaltar(texto, mark_char='-'):
    size = len(texto)
    print(mark_char * size)
    print(texto)
    print(mark_char * size)

resaltar('Informe sobre probabilidad A')
resaltar('Informe sobre probabilidad A', '=')
```

Que produce la siguiente salida:

```
---------------------------
Informe sobre probabilidad A
---------------------------
===========================
Informe sobre probabilidad A
===========================
```

Los valores por defecto se evaluan en el momento y en el ámbito en que se realiza la definición de la función. Por eso, el siguiente código:

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

Imprimirá 5.

La evaluación del valor por defecto, por tanto, solo se hace una vez. Si el valor por defecto es inmutable, esto no tiene mayor importancia, pero si es mutable, como una lista, un diccionario o, como veremos más adelante, una instancia de la mayoría de las clases, puede que no se comporte como esperamos. Por ejemplo, la siguiente función acumula los parámetros con los que ha sido llamada, porque la lista `l` se crea durante la definición de la función y no en cada llamada:

```
>>> def f(a, l=[]):
...     l.append(a)
...     return l
...
>>> print(f(1))
[1]
>>> print(f(2))
[1, 2]
>>> print(f(3))
[1, 2, 3]
>>>
```

Si queremos evitar este comportamiento, la forma habitual es:

```
>>> def f(a, l=None):
...     if l is None:
...         l = []
...     l.append(a)
...     return l
>>> print(f(1))
[1]
>>> print(f(2))
[2]
>>> print(f(3))
[3]
>>> print(f(4, [1,2,3]))
[1, 2, 3, 4]
>>>
```

Es muy cómodo poder añadir parámetros con valores por defecto a una función ya existente y en uso, ya que nos permite ampliar las capacidades de la función sin romper el código existente. Por ejemplo, la función `resaltar` podría haberse definido inicialmente con un único parámetro, el texto, solo para darnos cuenta, después de usarlo en multitud de sitios, que necesitamos un carácter de resaltado diferente en un determinado caso.

Si no dispusieramos de parámetros por defecto, nuestas opciones pasaria por, o bien definir una nueva funcion, `resaltar2`, con lo que ello implica (código redundante, deplicación de funcionalidad, etc...) o bien tendríamos

que localizar todas las llamadas hechas a la función y reescribirlas para incluir los dos parámetros.

2.5.2 Parámetros por nombre

Podemos especificar los parámetros de una función por su nombre, en vez de por posicion. Supongamos que necesitamos escribir una función que calcule el área de un triángulo. ¿que parámetros necesita? Sabiendo que el área de un triángulo puede calcularse con la fórmula *base* por *altura* partido por dos, parece lógico suponer que necesitaremos dos parámetros, base y altura del triángulo. la función podría ser algo así:

```
def area_triangulo(base, altura):
    return (base * altura) / 2.0
```

Esta función puede invocarse de cualquiera de estas diferentes maneras:

```
>>> print(area_triangulo(3, 4))
6.0
>>> print(area_triangulo(3, altura=4))
6.0
>>> print(area_triangulo(base=3, altura=4))
6.0
>>> print(area_triangulo(altura=4, base=3))
6.0
>>>
```

Eso si, si se mezclan paso de parámetros por posición con paso de parámetros por nombre, los parámetros por posición siempre deben ir primero:

```
>>> print(area_triangulo(3, altura=4))
6.0
>>> print(area_triangulo(altura=4, 3))
  File "<stdin>", line 1
SyntaxError: non-keyword arg after keyword arg
>>>
```

El poder especificar los parámetros por su nombre, combinando con los valores por defecto, nos permite simplificar mucho la lectura del código, especialmente con funciones con multitud de parámetros, de los cuales normalmente el usuario está interesado solo en una pequeña parte.

Volviendo a nuestra función de cálculo del área de un triángulo, resulta que hay otras formas de calcular el área; por ejemplo, si conocemos las longitudes de los tres lados del triángulo: a, b y c, podemos usar la Formula de Herón:

> En un triángulo de lados a, b, c, y semiperímetro s=(a+b+c)/2, su área es igual a la raíz cuadrada de s(s-a)(s-b)(s-c).

Podemos modificar la función para que acepte todos estos parámetros, asignado valores predeterminados:

```
import math

def area_triangulo(base=0, altura=0, a=0, b=0, c=0):
        if base and altura:
            return (base * altura) / 2.0
        elif a and b and c:
            s = (a + b + c) / 2
            return math.sqrt(s*(s-a)*(s-b)*(s-c))
        else:
            raise ValueError('Hay que especificar base y altura, o los lados a,b,c')

print(area_triangulo(base=3, altura=4))
```

```
print(area_triangulo(a=3, b=4, c=5))
print(area_triangulo())
```

Cuyo resultado sería:

```
6.0
6.0
Traceback (most recent call last):
  File "ejemplos/area_triangulo.py", line 22, in <module>
    print(area_triangulo())
  File "ejemplos/area_triangulo.py", line 18, in area_triangulo
    raise ValueError('Hay que especificar base y altura, o los lados a,b,c')
ValueError: Hay que especificar base y altura, o los lados a,b,c
```

2.5.3 Parámetros arbitrarios

Por último, pero no menos importante, podemos especificar funciones que admitan cualquier número de parámetros, ya sea por posición o por nombre. Para ello se usan unos prefijos especiales en los parámetros a la hora de definir la función, $\star$ para obtener una tupla con todos los parámetros pasados por posición y $\star\star$ para obtener un diccionario con todos los parámetros pasados por nombre.

Por ejemplo, la siguiente función admite un parámetro inicial obligatorio y a continuación el número de argumentos que quiera; todos esos argumentos serán accesibles para el código de la función mediante la tupla `args`:

```
def cuenta_ocurrencias(txt, *args):
    result = 0
    for palabra in args:
        result += txt.count(palabra)
    return result

texto = """Muchos años después, frente al pelotón de fusilamiento,
el coronel Aureliano Buendía había de recordar aquella tarde remota
en que su padre le llevó a conocer el hielo."""

print(cuenta_ocurrencias(texto, 'coronel', 'el', 'tarde', 'fusilamiento'))
print(cuenta_ocurrencias(texto, 'remota', 'hielo'))
print(cuenta_ocurrencias(texto))
```

Que nos devuelve:

```
10
2
0
```

De igual forma, usando la sintaxis especial $\star\star$, podemos escribir una función que aceptará cualquier número de parámetros especificados por nombre. Los nombres y valores de los parámetros serán accesibles para el código de la función mediante la variable indicada, normalmente llamada `kwargs`. El siguiente ejemplo imprime los nombres y valores de los paramentros que se le pasen:

```
>>> def dump(**kwargs):
...     for name in kwargs:
...         print(name, kwargs[name])
...
>>> dump(a=1, b=2, c=3)
a 1
b 2
c 3
```

```
>>> dump(hola='mundo')
hola mundo
```

Podemos especificar una función que acepte un número arbitrario de parámetros por posición, seguidos por un número arbitrario de parámetros por nombre -siempre en este orden- de la siguiente manera:

```
def f(*args, **kwargs):
    ...
```

2.5.4 Listas, tuplas o diccionarios como parámetros

Puede que tengamos el caso contrario, una función que acepta `n` parámetros, y nosotros tenemos esos parámetros en una lista o tupla. Por ejemplo, la función `range` espera dos parámetros, de inicio y final. Si tenemos esos parámetros en una lista, en vez de desempaquetarlos a mano, podemos usar la sintaxis `*` para pasar la tupla directamente:

```
>>> range(3, 6)             # Llamada normal
[3, 4, 5]
>>> args = [3, 6]
>>> range(*args)            # Llamada con parámetros empaquetados
[3, 4, 5]
>>>
```

De la misma manera, podemos desempaquetar un diccionario para que sea aceptable como parámetros de una función usando `**`:

```
datos = {'base':3, 'altura': 4}
print(area_triangulo(**datos))
```

2.5.5 Lambda

Dentro del soporte para programación funcional de Python, se ha añadido la capacidad de crear pequeñas funciones anónimas, mediante la palabra reservada `lambda`. Por ejemplo, esta es la definición de una función que suma los dos parámetros que se le pasan: `lambda (x,y): x+y`.

No hace falta especificar la sentencia `return`, ya que sintácticamente las funciones lambda están limitadas a una única expresión, que será la resultante. Las funciones definidas con lambda pueden ser usadas de igual manera que cualquier otra función, son solo azucar sintáctico [2] para una definición de función normal. Las funciones lambda pueden referenciar variables en el ámbito en que se definan, por ejemplo:

```
>>> n = 42
>>> f = lambda x: x+n
>>> f(7)
49
>>> f(-12)
30
```

2.6 Módulos

Python tiene un sistema para poder almacenar nuestras definiciones de funciones, variables, clases y objetos en un fichero, desde el cual podemos usarlas en nuestros scripts o en el modo interactivo. Ese fichero se denomina **módulo**;y las definiciones hechas en ese módulo se deben **importar** para poder usarlas.

[2] Azúcar 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.

Un módulo, por tanto, es simplmente un fichero que contiene código python, y que (normalmente) tendra un nombre terminado con la extensión `.py`. Dentro del módulo, su propio nombre esta accesible, en forma de string, en la variable especial `__name__`.

Vamos a crear nuestro primer módulo: usando un editor de texto, vamos a crear un archivo llamado `fibo.py`, con el siguiente contenido:

```
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print(b, end=' ')
        a, b = b, a+b
    print()

def fib2(n): # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Ahora, si ejecutamos el interprete en modo interactivo en el mismo directorio donde tenemos el fichoro `fibo.py`, podemos ejecutar el siguiente comando:

```
>>> import fibo
>>>
```

Esta orden crea un nuevo nombre, `fibo`, en nuestro espacio de nombres local (podemos verlo haciendo `dir()`). Podemos acceder a las funciones definidas en el módulo mediante ese nombre:

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
>>>
```

Un módulo también puede contener sentencias ejecutables, no solo definiciones. En ese caso, dichas sentencias son ejecutadas la primera vez que se importa el módulo, por lo que suelen usarse para inicializarlo. También lo hacen si el módulo es ejecutado como un script independiente; de esta forma muchos módulos pueden tener una doble vida, como módulo y como script de utilidad. Veremos algún ejemplo de esto más adelante.

Cada módulo mantine su propia y privada tabla de símbolos, que a los efectos de todas las funciones definidas dentro del módulo actuan como variables globales. De esa forma podemos tener variables globales dentro del módulo que no afectan ni se ven afectadas por otras variables de otros módulos. Por otro lado, si sabes lo que estás haciendo (O no te importa) las variables "globales" del módulo pueden ser modificadas usando la misma sintaxis que la usada para las funciones: `modname.itemname`.

Los módulos pueden importar otros módulos. Se recomienda, aunque no es obligatorio, especificar todos las sentencias `import` al principio del módulo (o del programa, tanto da). Los nombres de los módulos importados se insertan en la tabla de símbolos del módulo importador.

Existe una variante en la sentencia `import```que nos permite importar solo lo que nos interese del módulo. En el caso anterior, por ejemplo, si solo estoy interesado en la función ``fib`, podría hacer:

```
>>> from fibo import fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
>>>
```

También hay una variante que nos permite importar todo el contenido de un módulo en nuestro propio espacio de nombres:

```
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
>>>
```

Esto importa todo el contenido del módulo excepto los nombres que empiecen por el caracter _. En general, esta forma de importación no se considera adecuada, ya que a menudo hace el código más ilegible, además del riesgo de colisión de nombres. Su uso más habitual es en la consola interactiva.

Si el fichero de un módulo combia, por ejemplo porque lo estamos editando, y lo tenemos cargado en una sesión interactiva, el interprete no se va a enterar de los cambios, tenemos que forzarlo a volver a leer el módulo con la función `reload(<nombre del modulo>)`.

El interprete de Python busca los módulos a importar buscando en una serie de directorios, empezando por el mismo directorio desde el que se ejecutó el programa principal. La lista de directorios donde busca esta definida en la variable `sys.path`.:

2.6.1 Ejecutando módulos como programas

Cuando ejecutamos un módulo python directamente, como:

```
python fibo.py
```

El código del módulo es ejecutado, igual que si lo hubieras importado, pero en este caso la variable `__name__` no contiene el nombre del módulo, sino la constante `__main__`. Por eso es habitual encontrarse al final del módulo algo como esto:

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

Con esa modificación conseguimos que el fichero sea utilizable como módulo y también como script independiente, porque el código que analiza la línea de comandos y llama a la función `fib` solo se ejecuta si se ejecuta como fichero "principal":

```
$ python fibo.py 50
1 1 2 3 5 8 13 21 34
```

Si el módulo es importado, ese código no se ejecuta:

```
>>> import fibo
>>>
```

Esto se usa a veces para proporcionar una interfaz de uso sencilla para el módulo o también para ejecutar código de pruebas; ejecutando el módulo como si fuera un script se ejecutan las baterias de pruebas.

2.6.2 Paquetes (Packages)

Los paquetes nos permiten organizar aun más nuestros programas, agrupando a los módulos siguiendo una estructura jerárquica. De la misma forma que los módulos permitian que diferente autores no tuvieran que preocuparse por si las

variables globales de uno entraban en conflicto con las de otro, los paquetes permiten a los desarrolladores de librerías complejas como NumPy o PIL (Python Image Library) dejar de preocuparse por posibles conflictos en los nombres de los módulos.

Para crear un paquete, lo unico que necesitamos en crear un directorio con su nombre, e incluir dentro los módulos que deseemos que formen parte del paquete. Además, debemos incluir un fichero con el nombre especial `__init__.py`.

Este fichero es muy importante, porque es la manera que tiene python de asegurarse de que el directorio es realmente un paquete. Podemos crear tantos directorios dentro del paquete como consideremos necesarios, pero todos ellos tandrán que tener su correspondiente fichero `__init__.py` para que formen parte del paquete.

Por ejemplo, si creamos la siguiente estructura:

```
paquete/
    __init__.py
    rama1/
        __init__.py
        modula_a.py
        modulo_b.py
    rama2/
        __init__.py
        modulo_c.py
```

Popdremos hacer las siguientes importaciones:

```
import paquete.rama1.modulo_a
from paquete.rama1 import modulo_a
from paquete.rama2.modulo_c import *
```

Al usar la forma `from package import item`, este `item` puede ser tanto un submódulo (o subpaquete), o algún otro nombre definido en el paquete, como una clase, una funcion o una variable. La sentencia `import` ' primero comprueba si es alguno de los nombres definidos en el paquete, y si no fuera así, intenta cargarlo como si fuera un módulo. Si esto falla, se eleva una excepción `ImportError`.

Por el contrario, cuando se usa la sintaxis `import item.subitem.subsubitem`, está claro que todos menos el último deben ser paquetes; el último puede ser un módulo o un paquete, pero no puede ser ni una función ni una clase ni una variable definida en el item previo.

2.6.3 Importar * de un paquete

¿Qué pasa si hacemos `from paquete.rama1 import *`? ¿Busca Python en el directorio cualquier cosa que parezca código python y lo importa? Pues la verdad que no. Por un lado, porque podría ser muy costoso en tiempo, y por otro, porque entra en conflicto con un principio de diseño de Python: **explícito mejor que implícito**. Es mejor que al autor indique explicitamente los módulos que deben importarse en estos casos.

Para ello se sigue la siguiente regla: si dentro del fichero `__init__.py` se define una variable de tipo lista llamada `__all__`, se consideran que esa lista es la de los nombres de los módulos que deben ser importados en caso de hacer una importación tipo `*`. Es responsabilidad del autor mantener esta lista actualizada.

Si no se define la variable `__all__`, Python simplemente importa y ejecuta el código definido en el módulo `__init__.py`.

En cualquier caso, las importaciones tipo `*` siguen desaconsejándose.

2.6.4 Módulos y paquetes incluidos en la Librería Estandar

Otra de las grandes ventajas de Python es la amplia librería que viene incluida con el lenguaje. La mayor parte de las librerias estandar están escritas en el propio Python, mientras que un pequeño número de ellas está escrito en C, por

razones de rendimiento. En algunos casos se incluyen dos versiones de la librería, una escrita en Python, más portable pero más lenta, y otra en C, más rápida. Es el caso, por ejemplo, de la librería StringIO, que tiene una versión en C más rápida y eficiente que podemos importar como cStringIO. Las dos implementan una clase cuyas instancias se comportan como si fueran ficheros, pero cuyo contenido está en memoria.

2.7 Objetos y Clases

Las clases (y los objetos) permiten que podamos definir nuestros propios tipos de datos en Python. Las Clases definen las propiedases (atributos) y las capacidades y comportamiento (métodos) general de los nuevos tipos. Un objeto es una variable creada a partir de una clase, o instanciada, y representa un caso particular dentro del conjunto de posibles instancias de una clase (De la misma forma que podemos considerar al número 7 como una instancia particular de la clase Numeros Enteros).

Las clases se definen en Python con la palabra reservada `class`. En teoría no hece falta nada más para crear una clase que su nombre; así, la clase más sencilla que podemos pensar es:

```
>>> class X:
...     pass
>>>
```

y, efectivamente, funciona. Pero no resulta demasiado útil. Podemos instanciar un objeto a partir de esta clase, simplemente hay que usar el nombre de la clase como si fuera una función:

```
>>> x = X()
>>> print(x)
<__main__.X instance at 0xb7268bac>
...
```

Podemos asignar dinámicamente atributos. Por ejemplo, imaginemos que necesitamos una nueva clase, que nos permita almacenar coordenadas geográficas. Podiamos crear atributos `latitud` y `longitud` dinámicamente en el objeto `x` creado anteriormente:

```
>>> x.latitud = 28.4779
>>> x.longitud = -16.3118
>>> print(x.latitud, x.longitud)
28.4779 -16.3118
>>>
```

Pero esto no es ni elegante, ni cómodo. En la mayor parte de los lenguajes orientados a objetos es obligatorio definir los atributos que puede tener un objeto. En PYthon no es necesario, pero si es conveniente tener centralizado la creación y asignacion de estos atributos. Para eso podemos definir un método con un nombre especial, `__init__`, que es invocada inmediatemente después de creado el objeto (Por lo que no es técnicamente el constructor, sino más bien inicializador, pero mucha gente se refiere a esta función como el constructor). Ya puestos, vamos a darle a nuestra clase un nuevo nombre y un poco más de empaque:

```
>>> class Point:
...     def __init__(self, lat, lng):
...         self.latitud = lat
...         self.longitud = lng
...
>>> x = Point(28.4779, -16.3118)
>>> print(x.latitud, x.longitud)
28.4779 -16.3118
>>>
```

Un poco por intuición, podemos suponer lo que hace Python cuando se crea la variable `x` a partir de la clase `Point` (A partir de ahora, *instanciar*). En primer lugar se crea el objeto. Inmediatamente a continuación, como hemos visto,

se llama al inicializador, y parece que tiene sentido suponer que los dos parámetros que usamos al crear al objeto son los mismos valores que se pasan al método inicializador con los nombres `lat` y `lng`, pero ¿De donde sale el primer parámetro, `self`? ¿Y qué representa?

Para programadores de C++ o Java

Los programadores de C++ o Java acceden a este mismo objeto propio usando la variable "magica" `this`. En Python se prefirió esta forma por considerarla más explicita. De igual manera, los atributos dentro de la función tienen que venir precedidos por el `self.`, no hay alias mágicos para los atributos de la instancia. En este caso, para evitar la ambigüedad.

Empezemos por la segunda pregunta: `self` representa al propio objeto recien creado. Además, podemos adelantar que este primer parámetro es lo que diferencia a las funciones que ya conociamos de los métodos; un método siempre tienen como primer parámetro la instancia sobre la que está siendo ejecutado.

Cuando definimos un método (en este caso `__init__`, pero vale para cualquiera), se generan en realidad dos versiones del mismo. La version que se asocia a la clase, y la versión que se asocia al objeto. Podemos verlo usando la funcion id, que nos muestra las identidades de los objetos, o la funcion `is`:

```
>>> class Point:
...     def __init__(self, lat, lng):
...         self.latitud = lat
...         self.longitud = lng
...
>>> x = Point(28.4779, -16.3118)
>>> print(x.latitud, x.longitud)
28.4779 -16.3118
>>>
>>> id(Point.__init__)
3073025724L
>>> id(x.__init__)
3072948500L
>>> Point.__init__ is x.__init__
False
>>>
```

Es más, si vemos la representacion de ambas funciones, ya vemos que son distintas:

```
>>> Point.__init__
<unbound method Point.__init__>
>>> x.__init__
<bound method Point.__init__ of <__main__.Point instance at 0xb726a18c>>
>>>
```

Una de ellas, la correspondiente a la clase `Point` se describe como *unbound method* (método no vinculado o libre), mientras que la correspondiente a la instancia se describe como *bound method* (método vinculado o ligado). ¿Que significa esto? Veamoslo con calma:

Cuando se intenta acceder a la función `__init__` desde la instancia, la clase prepara otra version especial de `__init__`, una en la que el primer parámetro ya ha sido "rellenado", por así decirlo, con la propia instancia, y es este el que se vincula a la instancia. De ahí que se pase de una método libre (El de la clase, que acepta tres parámetros) a un método vinculado (En la que el primer parámetro ya esta definido y acepta, por tanto, solo dos). Esta es la única diferencia entre los dos métodos.

Quizá ayude verlo con otro método más normal. Definamos un método que nos indique si un determinado punto está por encima o por debajo del Ecuador. Nuestro código queda así:

```
class Point:
    def __init__(self, lat, lng):
        self.latitud = lat
        self.longitud = lng

    def esta_en_hemisferio_sur(self):
        return self.latitud < 0
```

Si usamos esta definicion, vemos que podemos invocar cada una de las funciones, la de la clase y la de la instancia, si tenemos el cuidado de pasarle a cada una de ellas los parámetros que necesita:

```
x = Point(28.4779, -16.3118)
print(x, 'Está en el hemisferio sur', x.esta_en_hemisferio_sur())
print(x, 'Está en el hemisferio sur', Point.esta_en_hemisferio_sur(x))
```

con el mismo resultado (y correcto, por otro lado):

```
<__main__.Point instance at 0xb72f6cac> Está en el hemisferio sur False
<__main__.Point instance at 0xb72f6cac> Está en el hemisferio sur False
```

En resumen, cuando definimos métodos para una clase, tenemos que reservar el primer parámetro para el propio objeto. A la hora de llamar al método desde la instancia, los engranajes internos de Python ya se ocupan de poner el valor correcto como primer parámetro. La tradición y la costumbre marcan que este primer parámetro se llame `self`, pero en realidad no existe ninguna obligación de hacerlo impuesta por el lenguaje. Esta convención, sin embargo, es de las más respetadas dentro de la comunidad, y sería muy mala idea no seguirla. Haría nuestro código más extraño de leer para los demás, y a nosotros más haría más dificil leer el código de otros (Es la clasica situación *I loose-You loose*).

2.7.1 Herencia

Con esto tenemos un sistema bastante potente, una forma de agrupar en una sola entidad la estructura de datos y el código asociado con la misma, así que podemos tener nuestros propios tipos de datos. Pero en el mundo OOP pero para poder hablar de clases y objetos con propidad es necesario que haya algún tipo de herencia. La herencia nos permite definir una clase refinando o modificando otra (herencia simple) u otras (herencia múltiple). Veamos el caso de la herencia simple para empezar.

Si decimos que una clase A **deriva** o **hereda** de una clase B (o también que la clase B es una **superclase** de A), lo que queremos decir es que la clase A dispondrá de todos los atributos y métodos de la clase B, de entrada, aunque luego puede añadir más atributos o métodos y modificar (o incluso borrar, pero está muy mal visto) los atributos o métodos que ha heredado.

La forma de expresar esta herencia en python es:

```
>>> class A(B):
...     pass
```

En el caso de que la clase modifique uno de los métodos que ha heredado, se dice que ha **reescrito** o **sobreescrito** (override) el método.

Como los objetos instanciados de `A` tienen los mismos atributos y métodos que `B`, pueden (o quizá deben, ya que nada lo impide excepto el sentido común) ser usados en cualquier sitio donde se use un objeto instaciado de `B`. Esto implica que entra `A` y `B` hay una relación de tipo *es un tipo de*, donde `B` es una generalidad de `A`, o quiza `A` sea una particularidad de `B`, depende del punto de vista. En otras palabras, todo objeto `a` instanciado de `A` es tambien un `B` (pero no necesariamente al reves).

Como hemos visto, `A` puede sobreescribir un metodo `f` de `B`, pero eso no afecta al resto del código. Si tenemos una lista con objetos de tipo `A` y de tipo `B` mezclados, podemos invocar sin miedo el método `f()` en cada uno de ellos, con la seguridad de que en cada caso se invocará al método adecuado. Esta capacidad se llama **polimorfismo** (del griego

Múltiples Formas. Quizá el nombre no sea el más adecuado, porque es más algo así como la misma forma, diferentes contenidos, pero es el que ha cuajado).

En Python no existe el concepto de métodos o atributos privados. En vez de eso, existe una convención de uso, por la cual si un atributo o método empieza con el carácter subrayado, ha de entenderse que es de uso interno, que no deberías jugar con él a no ser que sepas muy bien lo que estás haciendo, y que si en un futuro tu código deja de funcionar porque ese atributo o método ha desaparecido, no puedes culpar a nadie más que a ti mismo. En resumen, se supone que todos somos adultos responsables. Aunque parezca una locura, funciona.

¿Para que sirve la herencia? Por encima de todo, para reducir el tamaño del código evitando repeticiones. Si organizamos las herencias correctamente en jerarquias, de más genéricas a más específicas, podemos compatibilizar el código común de las primeras con el más específico de las últimas. Además, y no es un beneficio baladí, simplifica los usos de los objetos instanciados, de forma que podemos tratarlos a todos como si fueran del mismo tipo, aun siendo diferentes. En el ejemplo de la lista de objetos derivados de `A` y `B`, si no tuvieramos herencia (y polimorfismo) tendriamos que implementar un `if` para poder distinguir entre los dos tipos, y luego llamar a la versión correspondiente del código para cada tipo. además, tendremos que revisar la sentencia `if` si se nos ocurre incluir una nueva clase `C`.

En la herencia múltiple, una clase puede heredar de más de una clase Base. El problema en estos casos es que puede haber conflictos si se definen, en ramas distintas, un mismo método o atributo (llamado herencia en diamante). En python hay un sistema de ordenación de las clases, relativamente complicado, pero que asegura que no haya ambigüedad con respecto a cual de las dos versiones del atributo o método se va a usar.

¿Que pasa si la clase `A` sobreescribe un método de `B`, pero aun así ha de invocarlo? En realidad es un caso muy común, en el que la clase `A` quiere hacer lo mismo que la clase `B` y *un poquito más*. Desde Python 2.2 hay una función `super` que nos ayuda a invocar el código de la clase (o clases) de la que derivamos. En Python 2.7 podemos usarla así (Suponiendo que queremos llamar al método `f` de la clase de la que derivamos):

```
>>> class A(B):
...     def f(self, arg):
...         super(A, self).f(arg)
...
>>>
```

En Python 3.0 podemos usar una sintaxis mucho más limpia:

```
>>> class A(B):
...     def f(self, arg):
...         super().f(arg)
...
>>>
```

2.7.2 Funciones auxiliares

Si tenemos un objeto y queremos saber si es una instancia de una clase en particular, o de alguna de sus subclases , podemos usar la función `isinstance(objeto, clase)`, que nos devolverá verdadero si es así. Si queremos saber si un objeto es instancia de una subclase podemos usar la funcion `issubclass(objeto, clase)`.

2.7.3 Sobrecarga de operadores

Se puede, como en C++, sobreescribir los operadores (operadores aritméticos, acceso por índices, etc...) mediante una sintaxis especial.

Los métodos y atributos que empiezan y acaban con un doble signo de subrayado tiene por lo general un significado especial. En algunos casos es para ocultarlos o para marcarlos para uso privado, pero en otros son para usos especiales y tienen significados particulares.

Por ejemplo, si en nuestra clase definimos un método `__len__`, podemos hacer que las instancias de esa clase puedan ser usadas con la función `len()`:

```
class A:
    def __len__(self):
        return 7 # por la cara

a = A()
print(len(a))
```

De igual manera, si a esta clase le añadimos los métodos `__setitem__` y `__getitem__` podemos hacer que se comporte como si fuera una contenedor accesible mediante las operaciones de índices: El siguiente código muestra una clase que puede ser accedida como si fuera una lista; si accedemos a las posiciones 0 a la 6 devuelve una descripción del número en texto (tiene problemas para recordar el 6), pero si el índice cae fuera de rango, devuelve una descripción genérica: `Muchos`. Si intentamos modificar sus valores, no da error, porque hemos definido el método de asignacion correspondiente, `__setitem__`, pero como este no hace nada (la sentencia `pass`, como su nombre indica, no es muy activa), los intentos de modificarlo son fútiles.

```
class A:
    _Tabla = {
        0: 'ninguno',  1: 'uno',     2: 'dos',
        3: 'tres',     4: 'cuatro',  5: 'cinco',
        6: 'umm... seis',
    }

    def __len__(self):
        return 7 # por la cara

    def __getitem__(self, index):
        if 0 <= index < 7:
            return self._Tabla[index]
        else:
            return 'Muchos'

    def __setitem__(self, index, value):
        pass

a = A()
print(a[3])
print(a[25])
a[4] = 'IV'
print(a[4])
```

Podemos sobrecargar también operadores algebráicos; por ejemplo, Supongamos que queremos escribir un módulo de álgebra lineal y que definimos la clase `Vector`:

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return 'Vector({0}, {1})'.format(self.x, self.y)
```

Podríamos crear un método para sumar un vector a otro, o una función independiente para sumar vectores, algo como esto:

```
v1 = Vector(2, 3)
v2 = Vector(-4, 2)
```

```
v3 = suma_vector(v1, v2)
```

Pero es claramente mejor, más legible y bonito, poder hacer:

```
v1 = Vector(2, 3)
v2 = Vector(-4, 2)
v3 = v1 + v2
```

Para eso definimos los métodos especiales __add__ y __sub__, con lo cual podemos definir el comportamiento cuando se sumen o resten dos instancias de nuesta clase:

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return 'Vector({0}, {1})'.format(self.x, self.y)

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    def __sub__(self, other):
        return Vector(self.x - other.x, self.y - other.y)


v1 = Vector(2,34)
v2 = Vector(7, 2)
print(v1 + v2)
print(v1 - v2)
```

Existen muchos métodos especiales; si definimos el método __str__, por ejemplo, sera llamado cada vez que Python necesite convertir la instancia en una cadena de texto (Por ejemplo, para imprimirla por pantalla, o usando la función `str`). El resultado de este método, por supuesto, debe ser string. En el apartado Basic Customization del tutorial de Python se describen todos ellos.

2.8 Guía de estilo

Es recomendable seguir la guía de estilo definida en el PEP8, ya que ayudan a organizar y facilitan la distribución de código. Los puntos más importantes de esta guía son:

- Para la indentación usa 4 espacios para cada nivel, no tabuladores. Si te vez obligado a usar tabuladores, por la razón que sea, NUNCA mezcles espacios con tabuladores.
- Reformatea el código para que no haya líneas de más de 79 caracteres. Esto ayuda cuando las pantallas son pequeñas y en las grandes permite comparar dos secciones de código lado a lado.
- Usa líneas en blanco para separar funciones y métodos.
- Si es posible, añade comentarios
- Mejor todavía incluir *docstrings*.
- Usa espacios para separar los operadores y despues de las comas, pero no inmediatamente antes o despues de un paréntesis: `a = f(1, 2) + g(3, 4)`.
- Existen unas convenciones para llamar a funciones, métodos y clases. Las clases deberían seguir el formato `CamelCase` (letras iniciales en mayúsculas, resto en minúsculas, sin separadores) y los métodos y funciones

deberian usar `lower_case_with_underscores` (todo en minúsculas, usar el caracter _ como separador de palabras). El primer argumento de un método siempre debería llamarse `self`.

- Si alguna de estas reglas hace el código menos legible, rómpela. Pero asegurate de que sea realmente así.

Día 2.- Librerías estandar

3.1 Excepciones

Hay dos tipos de errores en Python: Errores sintácticos y excepciones. Los errores sintácticos se producen cuando escribimos algo que el interprete de Python no es capaz de entender; por ejemplo, crear una variable con un nombre no válido es un error sintáctico:

```
>>> 7a = 7.0
  File "<stdin>", line 1
    7a = 7.0
    ^
SyntaxError: invalid syntax
```

La información del error es todo lo completa que el interprete puede conseguir. Normalmente indica la línea e incluso con una flecha intenta señalar la posición más o menos exacta del error. No siempre lo consigue, no obstante, porque a lo mejor el error es detectado en un sitio distinto de donde es generado. También incluye el nombre del fichero fuente.

Las excepciones son errores de funcionamiento; el interprete ha entendido el código, por lo que es sintácticamente correcto, pero aun así, produce un error. Por ejemplo, si intentmos dividir por cero:

```
>>> a, b = 7, 0
>>> c = a / b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Las excepciones son errores que se producen en tiempo de ejecución, y tienen la ventaja de que pueden ser tratados, si nos preparamos para ello. Pero si la excepción no es tratada, inevitablemente conducirá al fin de la ejecución del programa.

La última línea del mensaje de error es la que resume lo que ha ocurrido. Las excepciones pueden ser de distintos tipos, y se informa del tipo en el mensaje de error; en el caso anterior, el tipo de la excepción es `ZeroDivisionError`. Otros tipos de excepciones, algunos de los cuales hemos visto ya, son por ejemplo `ValueError` o `TypeError`.

Si prevemos la posibilidad de que se produzca un error, podemos prepararnos para esta eventualidad con la estructura `try/except`. Por ejemplo, el siguiente fragmento de código:

```
try:
    a, b = 7, 0
    c = a / b
```

```
except ZeroDivisionError:
    print("No puedo dividir por cero")
```

Funciona así:

1. Se intentan ejecutar el bloque de código dentro de la sentencia `try`.
2. Si no se produce ningúna excepción mientras ejecuta ese código, se omite el código dentro del bloque `except` y seguimos con la ejecución del programa.
3. Si ocurre una excepcion en una de las líneas del código del `try`, el resto de las líneas no se ejecuta. Si el tipo de excepción coincide con el especificado en la clausula `except`, se ejecuta el bloque de código asociado y el programa continua ejecutándose.
4. Si el tipo de la excepción no coincide con el indicado en la cláusula `except`, entonces es una excepción no tratada, y provoca la parada del programa y el despliege del mensaje de error correspondiente

Una sentencia `try` puede tener más de una sentencia `except`, para aplicar diferentes tratamientos a diferentes tipos de excepciones. También podemos hacer que una sentencia `except` gestione más de un tipo de error usando paréntesis:

```
>>> try:

... except (RuntimeError, TypeError, NameError):
...     pass
```

Si incluimos una sentencia `except` sin especificar ningun tipo de excepción, trataremos todas las excepciones posibles. Esto ha de hacerse con cuidado, porque resulta muy fácil enmascarar así cualquier tipo de error, incluso aquellos en los que no estamos pensando. Un uso común es imprimir un mensaje de error y luego volver a elevar la excepción, con la sentencia `raise`, para que esta acabe la ejecución del programa (o bien sea tratada por un nivel superior).

3.1.1 La sentencia else en clausulas try/except

La sentencia `try/except` puede tener una cláusula `else`, de forma similar a los bucles `for` y `while`. Si incluimos la cláusula `else`, esta debe ir después de la o las cláusulas `except`. El codigo dentro del `else` se ejecuta si y solo si todas las líneas dentro del `try` se han ejecutado sin ninguna excepción.

3.1.2 Argumento de la excepción

Cuando ocure una excepción, tiene un valor asociado, al que llamamos *argumento* de la excepción. Tanto la presencia como el tipo del argumento depende de cada tipo de excepción. La sentencia `except` puede especificar una variable despues del tipo de excepción (o tupla de tipos). Si lo hacemos, dicha variable queda asociada al valor de la instancia de la excepción. Este objeto nos permite acceder a más información acerca del error que se ha producido, incluyendo los argumentos asociados con la excepción. la última línea impresa en el mensaje de error es precisamente la expresión en forma de cadena de texto de ese objeto, es decir, el resultado de la llamada a `__str__`.

Los manejadores de escepciones no se limitan a controlar los errores en las líneas dentro del try, tambien capturan y tratan errores que puedan ocurrir dentro de funciones o métodos llamados, ya sea directa o indirectamente, por el código dentro del `try`. Por ejemplo:

```
>>> def esto_falla():
...     x = 1/0
...
>>> try:
...     esto_falla()
... except ZeroDivisionError as detail:
...     print('Detectado error en tiempo de ejecución:', detail)
```

```
...
Detectado error en tiempo de ejecución: division by zero
>>>
```

3.1.3 Legibilidad del código con excepciones

Las excepciones nos permite aumentar la legibilidad del código separando la lógica de control de errores de la lógica principal del programa. En C, por ejemplo, los errores no se indican con excepciones, sino que las llamadas a una función puede que devuelvan un código especial para indicar un error. En consecuencia, los programas en C suelen consistir en una secuencia de llamadas a funciones intercaladas con código de comprobación de errores. El flujo principal se hace más difícil de leer con todas estas interrupciones. Las excepciones permiten tener el flujo principal del código completo y sin interrupciones dentro del `try`, y aun así, controlar las distintas posibilidades de error mediante cláusulas `except` separadas.

3.1.4 Elevar excepciones

Podemos provocar nosotros mismo excepciones -normalmente expresado como *elevar* una excepción- usando la sentencia `raise` que vimos antes. El único argumento de raise debe ser la propia excepción, o bien la clase de la que se instancia (La excepción es cuando intentamos volver a emitir la excepción que estamos tratando dentro de un `except`, ya vimos entonces que basta con poner `raise` sin parámtros). Veasmos un ejemplo:

```
>>> raise NameError('Hola')
Traceback (most recent call last):
File "<stdin>", line 1, in ?
NameError: Hola
>>>
```

3.1.5 Definir nuestras propia excepciones

También podemos definir nuestras propias excepciones, definiendo clases que deriven, directo o indirectamente de la clase `Exception`, que es la clase base de todas las excepciones (Es decir, que todas las excepciones son casos particulares de `Exception`).

Las Excepciones definidas por el usuario suelen ser relativamente simples, apenas un contenedor para los atributos que nos aporten información sobre el error producido. A la hora de crear un módulo, si en este vamos a definir varios tipos nuevos de excepciones, es una práctica común definir una base clase para ese tipo de excepciones, y a partir de esa clase base, derivar cada una de los casos particulares. Así obtenemos una organización jerarquica para nuestros tipos de errores que puede ser muy útil para los programadores que usan el módulo o paquete. Normalmente, los nombres de las nuevas excepciones se hacen terminar en `Error`, siguiendo la nomenclatura de las excepciones estándar.

3.1.6 La cláusula `finally`

Por último, la sentencia `try` puede tener una cláusula final, que se ejecutará siempre, se hayan producido o no excepciones en el código del `try`. El uso normal de `finally` es incluir código de liberación de recursos, operaciones de limpieza o cualquier otro tipo de código que tenga que ejecutarse "si ó si". Por ejemplo, si abrimos un fichero, podemos poner en la cláusula `finally` la operación de cierre, de forma que se gerantiza que, pase lo que pase, el fichero se cerrará.

El código de la sentencia `finally` se ejecuta siempre a continuación del código en la sentencia `try`:

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print "división por cero!"
...     else:
...         print "el resultado es", result
...     finally:
...         print "Ejecutando sentencia finally"
...
>>> divide(2, 1)
el resultado es 2
Ejecutando sentencia finally
>>> divide(2, 0)
division por cero!
Ejecutando sentencia finally
>>> divide("2", "1")
Ejecutando sentencia finally
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
>>>
```

El error de tipo `TypeError` (por intentar dividir dos cadenas de texto) no es tratado por ninguna cláusula `except`, así que se vuelve a elevar después de ejecutar el código de `finally`.

3.2 Gestores de contexto: La estructura de control with

La sentencia `with` nos permite "envolver" un bloque de código con operaciones a ejecutar antes y después del mismo. A menudo las operaciones tienen una cierta simetria; por ejemplo, la operación de abrir un archivo implica que en algún lado tiene que haber una operación de cierre. En lenguajes que operan directamente con la memoria, como C o C++, la petición para reservar un trozo de memoria (`malloc`) tiene su reflejo en la operación de liberado de la misma (`free`). Un error común en programación es olvidar esta simetría: abrir un fichero pero no cerrarlo, o reservar una parte de la memoria pero no liberarla [1], por ejemplo. Hemos visto que podemos resolver estos problemas con una clausula `try/finally`, pero la sentencia `with` (Disponible desde Python 2.5) es más potenta y permite "encapsular" este mecanismo.

Por ejemplo, los objetos de tipo fichero pueden trabajar con `with`, de forma que en vez de hacer esto:

```
>>> try:
...     f = open('fichero.datos', 'r')
...     # proceso el fichero
...     n = len(f.readlines())
... finally:
...     f.close()
...
>>>
```

Podemos hacer:

[1] Lo que conduce a un problema llamado "fuga de memoria" (*memory leaks*): un error de software que ocurre cuando un bloque de memoria reservada no es liberada en un programa de computación, habitualmente porque se pierden todas las referencias a esa área de memoria antes de haberse liberado. Dependiendo de la cantidad de memoria perdida y el tiempo que el programa siga en ejecución, este problema puede llevar al agotamiento de la memoria disponible en la computadora.

```
>>> with open('fichero.datos', 'r') as f:
...     # proceso el fichero
...     n = len(f.readlines())
```

Y en ambos casos está garantizado el cierre del fichero, se hayan producido o no errores durante el proceso.

Para conseguir esto, la sentencia `with` utiliza internamente lo que se denomina un *gestor de contexto* (*context manager*). Un gestor de contexto es un objeto que sabe lo que hay que hacer antes y lo que hay que hacer después de usar otro objeto. La clase `file`, en el ejemplo anterior, es capaz de suministrar un generador de contexto que sabe que, cuando termine, el fichero debe cerrarse; por eso en el segundo ejemplo no hay necesidad de poner un `close` explícito (Con lo que tampoco podemos olvidarnos de ponerlo).

El mecanismo interno de `with` es más o menos así:

1. La expresión que viene después del `with` es evaluada y se obtiene de ella un gestor de contexto
2. El método `__exit__()` del gestor de contexto es **cargado**
3. Se llama al método `__enter__()` del gestor de contexto
4. Si se ha incluido un destino en la sentencia (con la palabra reservada `as`), se le asigna el valor retornado por el método `__enter__`

1. Se ejecuta el bloque de sentencias dentro del `with`.
2. Se ejecuta el método `__exit__()`. El método acepta tres argumentos: Si se ha producido una excepción, se le pasan el tipo, valor y traza de ejecución de la misma. Si no se han producido errores, los tres parámetros se pasan cono `None`. Si ha habido una excepción y `__exit__()` returna `False`, la excepción se elevará de nuevo; si, por el contrario, retorna `True`, la excepción es suprimida. Si no ha habido ningún error, el resultado de `__exit__()` es indiferente.

Se pueden usar varias expresiones dentro del `with`, en ese caso, se considera como si estuvieran anidadas:

```
with A() as a, B() as b:
    # codigo
```

equivale a:

```
with A() as a:
    with B() as b:
        # codigo
```

3.3 Iteradores y generadores

Los iteradores y generadores nos permite crear nuestras propias variables iterables.

3.3.1 Iteradores (*iterators*)

Como hemos visto ya en muchas ocasiones, la mayoría de los objetos de tipo contenedor pueden ser usados dentro de un bloque `for`:

```
for element in [1, 2, 3]:
    print element
for element in (1, 2, 3):
    print element
for key in {'one':1, 'two':2}:
    print key
for char in "123":
```

```
    print char
for line in open("myfile.txt"):
    print line,
```

Lo que nos da un estilo limpio, conciso y fácil de leer. Este mecanismo de iteración permea y unifica todo el lenguaje. La sentencia `for` funciona internamente llamando a la función `iter()`, pasándole como parámetro el contenedor. La función retorna un objeto de tipo iterador, que se caracteriza por disponer de un método llamado `next()`, que es responsable de hacer lo siguiente:

1. Mientras queden valores en el contendor, `next()` nos devuelve cada uno de ello, unos tras otro, en cada invocación.
2. Cuando ya no quedan más valores, `next()` eleva la excepción `StopIteration`, de forma que el bucle `for` sabe que ha llegado al final.

Quizá con el siguiente ejemplo se vea más claro:

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<iterator object at 0x00A1DB50>
>>> it.next()
'a'
>>> it.next()
'b'
>>> it.next()
'c'
>>> it.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
    it.next()
StopIteration
```

Ahora que ya sabemos como funciona internamente el bucle `for`, vemos que es fácil añadir a nuestras clase un comportamiento similar a un contenedor, esto es, que sea iterable. Para ello debemos definir un método con el nombre `__iter__()` que devuelva un objeto. Ese objeto, el iterador, debe implementar un método `next()` que se comporte como se describió anteriomente. Un caso habitual es cuando la propia clase implementa el método `next()`, en ese caso, el método `__iter__()` simplemente devuelve `self`.

Vamos a implementar un objeto CuentaAtras de forma que sea iterable:

```
class CuentaAtras:
    def __init__(self, tope):
        self.tope = tope

    def __iter__(self):
        self.counter = self.tope
        return self

    def next(self): return self.__next__() # python 2.7

    def __next__(self):
        result = self.counter
        self.counter -= 1
        if result < 0:
            raise StopIteration
        return result
```

```
for i in CuentaAtras(12):
    print(i, end=', ')
print()
```

3.3.2 Generadores (*Generators*)

Los Generadores son una forma sencilla y potente de crear iteradores. Son exactamente como cualquier función, pero en vez de devolver el resultado con `return`, lo devuelven con la sentencia `yield`. Cada vez que se llama a `next()`, el generador continua a partir de donde se quedó (Recuerda todo su estado: los valores de las variables y que línea fue la última que se ejecutó). El siguiente es un generador trivial:

```
>>> def cuenta_atras(n):
...     while n >= 0:
...         yield n
...         n -= 1
...
>>> for i in cuenta_atras(5): print(i)
...
5
4
3
2
1
0
>>>
```

Cualquier cosa que se pueda hacer con generadores puede ser tambien implementada mediante clases que implementen los métodos de los iteradores que se describieron antes.

Lo cómodo de los generadores es que los métodos `__iter__()` y `next()` se generan automáticamente. Otra ventaja es la capacidad del generador de recordar todos los datos locales y el estado de ejecución entre llamadas. Gracias a esto es normalmente más fácil de escribir que su equivalente como clase. Además, cuando el generador termina, eleva automáticamente una excepción de tipo `StopIteration`. Con todas estas características, los generadores son la forma más fácil de implementar un iterador.

3.4 Programación funcional

La programación funcional parte de la premisa de que las funciones son solo otro tipo de variables; por tanto, todo lo que podemos hacer con una variable, lo debemos poder hacer con una función. Podemos pasar funciones como parámetros de otros funciones, las funciones nos pueden retornar otras funciones, las funciones se pueden almacenar en un diccionario, etc...

Esto se expresa normalmente con la frase: “Las funciones son objetos de primera clase”.

Los primeros ejemplos de programación funcional estaban en python desde la versión 1.0; se trata de las expresiones `lambda`, que ya vimos, y las funciones: `filter()`, `map()` y `reduce()`.

la función `filter` acepta como primer parámetro una función, y como segundo parámetro una secuencia. El resultado en otra secuencia en la que se estan sólo aquellos valores de la secuencia original para los que el resultado de aplicarles la función es `True`

Nota: Cambios en Python 2.7 / Python 3.x

En Python 2.7, si la secuencia es una string o una tupla, el resultado es del mismo tipo, si es una lista o alguna otra cosa, el resultado será una lista. En Python 3.0 siempre se devuelve un iterador; si se necesita una lista siempre se puede hacer `list(map(...))`.

Por ejemplo, la lista de los primeros 200 números que son divisibles por 5 y por 7:

```
>>> def es_divisible_por_5_y_7(x): return x % 5 == 0 and x % 7 == 0
...
>>> for i in filter(es_divisible_por_5_y_7, range(1, 201)): print(i)
...
35
70
105
140
175
>>>
```

la función `map` tambien acepta como primer parámetro una función, y como segundo, una secuencia. El resultado es otra secuencia, compuesta por los resultados de llamar a la función pasada en cada uno de los elementos de la secuencia original. Por ejemplo, para imprimir la lista de los cubos de los 10 primeros números:

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

Podemos pasar más de una secuencia; en ese caso, la función pasada como parámetro debe aceptar tantos parámetros como secuencias haya, y es invocada con los parámetros que correspondan de cada una de las secuencias (O con el valor `None`, si una de las secuencias es más corta que las otras). Por ejemplo, calculemos una lista con las medias de los datos de otras dos listas:

```
>>> l1 = [123, 45, 923, 2, -23, 55]
>>> l2 = [9, 35, 87, 75, 39, 7]
>>> def media(a, b): return (a + b) / 2
...
>>> map(media, l1, l2)
[66.0, 40.0, 505.0, 38.5, 8.0, 31.0]
>>>
```

la función `reduce`, para no variar, acepta una función y una secuencia, pero al contrario que las anteriores, devuelve un único valor. Ese valor se calcula de la siguiente manera: en primer lugar, la función que se pasa como primer parámetro tiene que aceptar dos valores, y retornar uno. Se calcula el resultado de aplicar la función a los dos primeros valores de la secuencia. A continuación, se aplica de nuevo la función, esta vez usando como parámetros el resultado calculado antes y al tercer elemento de la secuencia. Se prosigue así hasta agotar los valores de la secuencia original.

por ejemplo, para sumar los números del 1 al 10, podriamos (pero no deberíamos, vease la nota a continuación) hacer:

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

Nota: la función `sum`

No se debe usar este modo de realizar sumas, porque esta es una necesidad tan común que ya existe una función incorporada para ello: `sum(sequence)`, que funciona exactamente igual, pero más rápido al estár implementada en

C.

Si solo hay un elemento en la lista, se devuelve ese elemento. Si la lista esta vacia, sin embargo, se considera un error y se eleva una excepción de tipo `TypeError`.

Nota: Cambios en Python 2.7 / Python 3.x

En Python 3.x `reduce` ya no es una función incorporada por defecto, si se quiere utilizar, hay que importarla del módulo `functools`.

Se puede indicar también un tercer parámetro, que sería el valor inicial. En ese caso, la función se empieza aplicando como parámetros el valor inicial y el primer elemento, luego con el resultado previo y el segundo elemento, etc...

3.4.1 El módulo `itertools`

Este módulo implementa una serie de iteradores que se pueden usar como elementos básicos, inspirados por distintas construcciones que podemos encontrar en otros lenguajes como APL, Haskell o SML. Estas utilidades cuentan con la ventaja de ser estándar, eficientes y rápidas, al estar implementadas a bajo nivel. Con estas utilidades se puede formar una especie de *algebra de iteradores* que permite construir herramientas más especializadas de forma suscinta y eficiente.

Algunas de las funciones de este módulo son:

count(start, [step])

Iterador infinito. Devuelve la cuenta, empezando por `start` e incrementados por el valor opcional `step` (por omisión, 1):

```
>>> for i in itertools.count(10, -1):
...     print(i)
...     if i == 0: break;
...
10
9
8
7
6
5
4
3
2
1
0
```

cycle(s)

Iterador infinito. Empieza devolviendo los elementos de la secuencia `s`, y cuando termina, vuelve a empezar:

```
>>> color = itertools.cycle(['red', 'green', 'blue'])
>>> for i in range(7):
...     print(color.next())
...
red
green
blue
red
```

```
green
blue
red
>>>
```

chain(s1, s2, ... ,sn)

Encadena una secuencia detrás de otra:

```
>>> l = [c for c in itertools.chain('ABC', 'DEF')]
>>> l
['A', 'B', 'C', 'D', 'E', 'F']
>>>
```

groupby(s, f)

Agrupa los elementos de una secuencia `s`, por el procedimiento de aplicar la función `f` a cada elemento, asignado al mismo grupo a aquellos elementos que devuelven el mismo resultado. El resultado es un iterador que retorna duplas (tuplas de dos elementos) formadas por el resultado de la función y un iterador de todos los elementos correspondientes a ese resultado:

```
>>> l = ['Donatello', 'Leonardo', 'Michelangelo', 'Raphael']
>>> f = lambda x: x[-1]
>>> for (letra, s) in itertools.groupby(l, f):
...     print(letra)
...     for i in s: print(' -', i)
...
o
 - Donatello
 - Leonardo
 - Michelangelo
l
 - Raphael
>>>
```

product(p, q, ...)

Devuelve el proucto cartesiano de las secuencias que se la pasen como parámetros. Es equivalente a varios bucles for anidados; por ejemplo:

```
product(A, B)
```

devuelve el mismo resultado que:

```
((x,y) for x in A for y in B)
```

Ejemplo de uso:

```
>>> for (letra, numero) in itertools.product('AB', [1,2]):
...     print(letra, numero)
...
A 1
A 2
B 1
B 2
>>>
```

combinations(s, n)

Devuelve todas las combinaciones de longitud `n` que se pueden obtener a partir de los elementos de `s`. Los elementos serán considerados únicos en base a su posición, no por su valor, así

que si cada elemento es único, no habra repeticiones dentro de cada combinación. El número de combinaciones retornadas sera de `n! / r! / (n-r)!`, donde `r [0, 1, ..., n]`. Si `r` es mayor que `n`, no se devuelve ningún valor.

```
>>> for i in itertools.combinations('ABCD', 1): print(''.join(i))
...
A
B
C
D
>>> for i in itertools.combinations('ABCD', 2): print(''.join(i))
...
AB
AC
AD
BC
BD
CD
>>> for i in itertools.combinations('ABCD', 3): print(''.join(i))
...
ABC
ABD
ACD
BCD
>>> for i in itertools.combinations('ABCD', 4): print(''.join(i))
ABCD
>>>
```

3.4.2 Comprensión de listas

La *comprensión de listas* (del inglés *list comprehension*) nos proporcionan una forma muy expresiva de crear listas a partir de otras preexistentes. El uso habitual es crear una lista donde los elementos son resultado de aplicar una serie de operaciones con otra secuencia o iterable, o para crear una subsecuencia de aquellos elementos que cumplan una determinada condición (o ambas cosas). En resumen, nos permiten hacer lo mismo que `map` o `filter`, pero de forma más legible.

Por ejemplo, podemos crear una lista con los cuadrados de los 10 primeros así:

```
>>> squares = []
>>> for x in range(11):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
>>>
```

Con `map` conseguimos el mismo resultado con menos líneas:

```
>>> squares = map(lambda x: x**2, range(10))
>>>
```

pero obtendríamos el mismo resultado, aun más legible, con el siguiente código:

```
>>> squares = [x**2 for x in range(11)]
>>>
```

Una compresion de listas consiste en un corchete, a continuación una expresión, seguida de una clausula `for`, y seguida opcionalmente por una o más condiciones `if`, y finalmente cerrada por otro corchete. El resultado será una

nueva lista resultante de la evaluación de la expresion en el contexto del bucle `for` y de las clausulas `if`. Por ejemplo, el siguiente código encuentra cuales de los primeros 1500 números cumplen la condición de que su cubo acaba en 272:

```
>>> [x**3 for x in range(501) if str(x**3).endswith('272')]
[13481272, 116214272]
>>>
```

Expresiones generadoras

También tenemos una construcción muy similar, una **expresión generadora** (Disponible desde Python 2.4), que en vez de devolvernos una lista, nos permite obtener un generador. La sintaxis es idéntica a una comprensión de lista, pero sustituyendo los corchetes por paréntesis. Atendiendo al rendimiento, la diferencia puede ser muy importante, ya que con la lista obtenemos todos los elementos ya generados (y, por tanto, consumiendo memoria) mientras que un generador nos irá dando los valores de uno en uno (Lo que en informática se conoce como *evaluación perezosa* o *lazy evaluation*):

```
>>> s = [x**2 for x in range(11)]
>>> s # Es una lista
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
>>> s = (x**2 for x in range(11))
>>> s # es un generador
<generator object <genexpr> at 0xb74588ec>
>>> s.next()
0
>>> for i in s: print(i)
...
1
4
9
16
25
36
49
64
81
100
>>>
```

Comprensión de diccionarios

También tenemos a nuestra disposición (Desde python 2.7) la comprensión de diccionarios, es decir, poder crear diccionarios a partir de otras fuentes de datos. La sintaxis es similar, pero cambiando los corchetes/paréntesis por llaves, y la expresión tienen que tener la forma `<clave>:<valor>`:

```
>>> d = {str(x):x**2 for x in range(5)}
>>> d
{'1': 1, '0': 0, '3': 9, '2': 4, '4': 16}
>>> d = {str(x):x**2 for x in range(5) if x % 2 == 0}
>>> d
{'0': 0, '2': 4, '4': 16}
>>> print {i : chr(65+i) for i in range(4)}
{0 : 'A', 1 : 'B', 2 : 'C', 3 : 'D'}
>>>
```

Conprensión de conjuntos

Por último, también es posible definir un conjunto a partir de otros valores. La única forma de distinguir esta sintaxis de la usada para diccionarios es que la expresión no va en la forma <clave>:<valor>. Podemos crear un conjunto usando la forma más sencilla: una serie de valores separados por comas:

```
>>> s = {'a', 'b', 'c'}
>>> s
set(['a', 'c', 'b'])
>>>
```

O usar una comprensión, donde cada expresión obtenida pasará a formar parte del conjunto.

```
>>> s = {str(x**2) for x in range(7)}
>>> type(s)
<type 'set'>
>>> s
set(['25', '16', '36', '1', '0', '4', '9'])
>>>
```

3.4.3 Decoradores

Como deciamos en la sección de programación funcional, las funciones en Python son objetos de primera clase, es decir, son objetos como cualquier otro: Pueden tener atributos, se pueden pasar como parámetros, tienen una identidad, etc...

Hemos visto ya, con las funciones `map` y `filter`, por ejemplo, el paso de una función como parámetro. Algo que no hemos visto hasta ahora es que una función devuelva como resultado otra función. Parece un poco raro, pero si lo pensamos, no hay nada extraño. Veamos un ejemplo trivial:

```
>>> def dame_una_funcion_incremento(inc):
...     def funcion_a_retornar(x):
...         return x + inc
...     return funcion_a_retornar
...
>>> inc3 = dame_una_funcion_incremento(3)
>>> inc3(6)
9
>>> inc47 = dame_una_funcion_incremento(47)
>>> inc47(3)
50
```

Así, `dame_una_funcion_incremento` es una función que devuelve otra función. Sabiendo que esto es posible, veamos la definición de decorador:

> Un decorador es una función que acepta como parámetro una función, y devuelve otra función, que normalmente sustituirá a la original.

El uso de decoradores se enfoca a resolver el siguiente problema: Tenemos un conjunto de funciones, y queremos que todas ellas hagan una nueva cosa, algo por lo general ajeno al propio comportamiento de la función, pero que queremos que todas lo hagan por igual. En otras palabras, queremos añadir una funcionalidad horizontal. El ejemplo clásico es añadir información de auditoria a las funciones.

Supongamos que tenemos un conjunto de funciones `a()`, `b()`,..., `z()`, cada una de ellas con sus propios parámetros, comportamientos, particularidades, etc... Y queremos ahora, con el mínimo trabajo posible, que cada función escriba en un fichero log común cuando empieza a trabajar y cuanto termina.

La primera opción es sencilla, pero trabajosa: reescribir cada una de las funciones de forma que, por ejemplo, la función `a()` pasa de:

```
def a():
    # código de a
```

a:

```
def a():
    with open('/tmp/log.txt', 'a') as log:
        log.write('Empieza la función a\n')
    # codigo de a
    with open('/tmp/log.txt', 'a') as log:
        log.write('Acaba la función a\n')
```

Y así con todas las funciones. Los problemas de este enfoque son:

- Hay que reescribir un montón de código
- Hay muchísimo código repetido (Todas esas escrituras al log)
- El tamaño del código aumenta bastante
- La lógica de todas las funciones queda difuminada con todas las llamadas al log, que no son parte del problema que soluciona *a()*
- Si hubiera que cambiar la información del log, por ejemplo, para incluir la fecha y hora, tendriamos que volver a modificar todas las funciones

Código fuente

Se puede ver el código de este ejemplo en `ejemplos/decoradores.py`

El decorador intenta solucionar estos problemas. Lo que hace un decorador normalmente es coger la función original (En nuestro caso, las funciones `a()`, `b()`,..., `z()`), modificarlas y sustituirlas, de forma que ahora, cuando se llama a `a()`, se invoca en realidad a nuestra versión modificada, que a su vez invocará (o no, según el caso) a la `a()` original.

Para el ejemplo de log, primero creamos una función decoradora, que llamaramos `logged` en un derroche de originalidad. Para simplificar, en vez de escribir a un fichero log nos limitaremos a hacer dos prints, uno antes de que empieze la función y otro después:

```
def logged(func):
    def inner(* args, **kwargs):
        print('Empieza la función {}'. format(func.__name__))
        func(*args, **kwargs)
        print('Termina la función {}'. format(func.__name__))

    return inner
```

Ahora podemos sustituir, pongamos por ejemplo, la funcion `b()` por su version decorada, haciendo:

```
b = logged(b)
```

```
@logged
```

O podemos usar el simbolo `@` y la siguiente sintaxis, que hacen exactamente lo mismo (En este caso , para la funcion `c()`:

```
@logged
def d(msg): print('Soy d y digo: {}'.format(msg))
```

Con los decoradores hemos resuelto los problemas anteriores. Hay que tocar el código de cada función, si, pero el cambio es mínimo: añadir el decorador con el simbolo `@`. El código no se repite. No hay aumento apreciable de

tamaño del mismo y el código interno de las funciones `a()`, `b()`,..., `z()` no se ve perturbado por la funcionalidad del log. Además, podemos añadir nuevas características a las funciones "logeadas" modificando solo una cosa: el decorador `logged`.

3.5 Módulos de la librería estándar

3.5.1 Optimización de rendimiento

Para algunos usuarios de Python el rendimiento es una preocupación fnudamental, y están interesados es poder medir la diferencia de rendimiento entre opciones alternativas. Python proporciona varias maneras de poder resolver estas preguntas, siendo una de las más sencillas el módulo `timeit`.

pero antes, un consejo:

> **La optimización prematura es la raíz de todo mal**
>
> Donald Knuth "Structured Programming with go to Statements".

Hay que entender que sin medidas precisas y globales del rendimiento de la aplicación, no debemos optimizar. Hay optimizaciones triviales, como por ejemplo no concatenar cadenas de texto (mejor ir añadiendolas a una lista y usar `join()` para obtener la cadena de texto final), pero a no ser que sean realmente obvias, las optimizaciones deben esperar hasta que podamos medir el desempeño en un contexto global. La herramienta que realiza estas medidas se conocen como *profiler*.

El peligro de optimizar sin este conocimiento es que no sabemos que parte del código es la que realmente demanda nuestros esfuerzos de mejora. Podemos acabar complicando una parte del código que, en realidad, solo es responsable del 1 % del tiempo de ejecución. Eso significa que si consiguieramos, por ejemplo, optimizar esa parte del código para que vaya el doble de rápido (Una optimización del 200 %, realmente impresionaante y nada habitual) el resultado neto seria una mejora de un 0.005 % en el rendimiento total. De hecho, si consiguieramos una mejora espectacular, que ese código se ejecute en, digamos, 0.001 nanosegundos, aun así, la mejora en el redimiento total nunca superará el 1 %. A cambio, tenemos software más complejo, más difícil de leer y con mayor capacidad potencial de errores.

timeit

timeit permite obtener una medida fiable de los los tiempos de ejecución de un fragmento de código Python.

Vimos en su día que había dos formas de intercambiar los valores de dos variables, usando una variable auxiliar o usando el mecanismo de empaquetado de tuplas:

```
>>> # Usando una variable auxiliar
>>> a = 7; b = 12
>>> temp = a; a = b; b = temp
>>>
>>> # Usando tuplas
>>> a = 7; b = 12
>>> a, b = b, a
```

Pero ¿cuál será más rápida? Usando el módulo `timeit` podemos salir de dudas:

```
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=7; b=12').timeit()
0.0470120906829834
>>> Timer('a,b = b,a', 'a=7; b=12').timeit()
0.04259920120239258
```

El intercambio por medio de tuplas es -ligeramente- más rápido.

profile

Los módulos `profile` y `pstats` son más avanzados que `timeit`. Además de medir tiempos de ejecución, nos darán herramienta para poder identificar las secciones críticas, es decir, nos permiten identificar aquellas partes del código que mas tiempo/ciclos de CPU consumen, de forma que podemos concentrarnos es optimizar las partes adecuadas.

Los módulos `profile` y `cProfile` (`cProfile` es simplemente una version de `profile` escrita en C para mejorar su rendimiento) nos dan un medio para recolectar y analizar estadísticas acerca del consumo del procesador que hace nuestro código Python.

El método más directo de analisis en el módulo `profile` es la función `run()`. Acepta como parámetro una cadena de texto con código fuente Python y crea un informe del tiempo gastado en cada una de las diferentes líneas de código, a medida que se ejecutan las sentencias.

Nuestra primera versión recursiva de Fibonacci nos será muy útil para comprobar el uso de estas herramientas, ya que su rendimiento se puede mejorar significativamente.

```
import profile

def fib(n):
    # from literateprograms.org
    # http://bit.ly/hlOQ5m
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

def fib_seq(n):
    seq = [ ]
    if n > 0:
        seq.extend(fib_seq(n-1))
    seq.append(fib(n))
    return seq

profile.run('print(fib_seq(20)); print()')
```

El informe estandar muestra primero una línea de resumen y luego un listado de detalle para cada una de las funciones ejecutadas:

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765]
()
         57356 function calls (66 primitive calls) in 0.376 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
       21    0.000    0.000    0.000    0.000 :0(append)
       20    0.000    0.000    0.000    0.000 :0(extend)
        1    0.000    0.000    0.000    0.000 :0(setprofile)
        1    0.000    0.000    0.376    0.376 <string>:1(<module>)
        1    0.000    0.000    0.376    0.376 profile:0(print(fib_seq(20)); print())
        0    0.000             0.000          profile:0(profiler)
 57291/21    0.376    0.000    0.376    0.018 profile_01.py:11(fib)
     21/1    0.000    0.000    0.376    0.376 profile_01.py:21(fib_seq)
```

Esta primera versión se toma 57.356 llamadas a diferentes funciones, tomando en total su ejecución más o menos un

tercio de segundo. El hecho de que solo haya 66 llamadas primitivas indica que el resto, es decir, la mayor parte de las llamadas, son llamadas recursivas. Los detalles sobre como se gasta el tiempo en cada función vienen dados por los siguientes valores, correspondientes a las columnas del informe: Número de llamadas (`ncalls`), el tiempo total gastado en la función (`tottime`), el tiempo medio consumido por cada llamada (`percall = ncalls/tottime`), el tiempo acumuado (`cumtime`) y la proporción entre tiempo acumulado y llamadas a primitivas.

Cuando hay dos números en la primera columna, indica que la llamada es recursiva. El primer número es el total de llamadas y el segundo el número de ellas que no son primitivas (Es decir, que se invocaron recursivamente). Por tanto, cuando una función no es recursiva, ambos números tienen que ser iguales.

Vemos claramente que la mayor parte del tiempo, 0,376 segundos, se ha gastado en llamadas a la función `fib`. Si añadimos un decorador de tipo :term:memoize eliminaríamos la mayoría de esas llamadas, con lo que parece que podriamos obtener una mejora de rendimiento notable:

```
import profile

class memoize:
    # from Avinash Vora's memoize decorator
    # http://bit.ly/fGzfR7

    def __init__(self, function):
        self.function = function
        self.memoized = {}

    def __call__(self, *args):
        try:
            return self.memoized[args]
        except KeyError:
            self.memoized[args] = self.function(*args)
            return self.memoized[args]

@memoize
def fib(n):
    # from literateprograms.org
    # http://bit.ly/hlOQ5m
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

def fib_seq(n):
    seq = [ ]
    if n > 0:
        seq.extend(fib_seq(n-1))
    seq.append(fib(n))
    return seq

if __name__ == '__main__':
    profile.run('print(fib_seq(20)); print()')
```

Efectivamante, como recordamos el valor de fibbonaci en cada nivel, evitamos la mayoría de las llamadas recursivas, y el número de llamadas descienda hasta 145, mientras que el tiempo total baja a 0.004 segundos:

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765]
()
         145 function calls (87 primitive calls) in 0.004 seconds
```

```
   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
       21    0.000    0.000    0.000    0.000 :0(append)
       20    0.000    0.000    0.000    0.000 :0(extend)
        1    0.000    0.000    0.000    0.000 :0(setprofile)
        1    0.000    0.000    0.004    0.004 <string>:1(<module>)
        1    0.000    0.000    0.004    0.004 profile:0(print(fib_seq(20)); print())
        0    0.000             0.000          profile:0(profiler)
    59/21    0.000    0.000    0.004    0.000 profile_02.py:19(__call__)
       21    0.004    0.000    0.004    0.000 profile_02.py:26(fib)
     21/1    0.000    0.000    0.004    0.004 profile_02.py:37(fib_seq)
```

Si el código que queremos probar requiere numerosas varibles, la cadena de texto con las inicializaciones puede ser incomoda. Podemos, en cambio, definir un contexto que almacene esas variables, y usar `runctx` en vez de `run`. en el siguiente ejemplo, el valor de `n` se toma directamente del contexto que hemos pasado como tercer parámetro.:

```
import profile
from profile_fibonacci_memoized import fib, fib_seq

if __name__ == '__main__':
    profile.runctx('print fib_seq(n); print', globals(), {'n':20})
```

pstats: Almacenando y trabajando con estadísiticas

Podemos ejecutar una análisis con `profile` y almacenarlo en un fichero, en vez de imprimirlo, para eso indicamos el nombre del fichero en la llamada a `run|runctx`. Con la clase `Stats`, definida en el módulo `pstats`, podemos recuperar la información deesos ficheros y formatearla de diferentes maneras. También podemos combiar varios conjuntos de resultados

La clase Stats nos permite reordenar y analizar los datos de rendimiento capturados por `profile`. Por ejemplo, podemos ordenar los resultados por orden de tiempo acumulado, y luego listar solo las 10 primeras de la lista, que nos daría unos buenos candidatos para optimizar:

```
import profile
import pstats
import re

profile.run('re.compile("foo|bar")', 'profile_pruebas_re')
p = pstats.Stats('profile_pruebas_re')
p.sort_stats('cumulative')
p.print_stats(10)
```

3.5.2 Debugging: Búsqueda de errores

Varios módulos nos ayudan a encontrar los errores en el programa. Los más importantes son `logging`, `traceback`, `pdb` y `trace`.

logging

El módulo `logging` define un sistema flexible y homogeneo para añadir un sistema de registro de eventos o :term:log a nuestras aplicaciones o librerías. Crear un log es relativamente fácil, pero la ventaja de usar el API definido en las librerías estándar es que todos los módulos pueden participar en un log común, de forma que podamos integrar nuestros mensajes con los de otros módulos de terceros.

El módulo define una serie de funciones habituales es sistemas de *logging*: `debug()`, `info()`, `warning()`, `error()` y `critical()`. Cada función tiene un uso dependiendo de la gravedad del mensaje a emitir; estos niveles, de menor a mayor severidad, se describen en la siguiente tabla:

Nivel	A usar para
DEBUG	Información muy detallada, normalmente de interes sólo para diagnosticar problemas y encontrar errores.
INFO	Confirmación de que las cosas están funcionando como deben.
WARNING	Una indicación de que ha pasado algo extraño, o en previsión de algún problema futuro (Por ejemplo, "No queda mucho espacio libre en disco"). El programa sigue funcionando con normalidad.
ERROR	Debido a un problema más grave, el programa no has sido capaz de realizar una parte de su trabajo.
CRITICAL	Un error muy grave, indica que el programa es incapaz de continuar ejecutándose.

Un ejemplo muy sencillo:

```
import logging
logging.warning('¡Cuidado!') # el mensaje sale por pantall
logging.info('Mira que te lo dije...') # este no aparecerá
```

Si ejecutamos este código, veremos que solo se imprime el primer mensaje:

```
WARNING:root:¡Cuidado!
```

Esto es porque el nivel por defecto es `WARNING`, es decir, que solo se emiten los mensajes de ese nivel o superior. La idea de usar niveles es precisamente para poder centrarnos en los mensajes que nos afectan en un determinado momento.

El mensaje impreso incluye el nivel y la descripción que incluimos en la llamada. También incluye una referencia a `root`, que se explicará más tarde. El formato del mensaje también es modificable, si queremos.

Lo más habitual es crear el log usando un ficharo de texto:

```
import logging
logging.basicConfig(filename='ejemplo.log', level=logging.DEBUG)
logging.debug('Este mensaje debería ir al log')
logging.info('Y este')
logging.warning('Y este también')
```

Si abrimos el fichero deberíamos ver:

```
DEBUG:root:Este mensaje debería ir al log
INFO:root:Y este
WARNING:root:Y este también
```

Al configurar el nivel como `DEBUG` vemos que se han grabado todos los mensajes. Si subieramos a `ERROR`, no aparecería ninguno.

El formato por defecto es `severity:logger name:message`. Podemos cambiar también el formato de los mensajes, usando el parámetro `format` en la llamada a `basicConfig`:

```
 import logging

logging.basicConfig(
    filename='ejemplo.log',
    level=logging.DEBUG,
    format='%(asctime)s -%(name)s -%(levelname)s -%(message)s'
    )
```

Podemos definir distintas instancias de loggers (las funciones que hemos visto hasta ahora usan el logger por defecto, de nombre `root`), y usar sus nombres para organizarlos en una jerarquía, usando puntos `.` como separadores, de forma similar a como organizamos los paquetes. Los nombres pueden ser lo que queramos, pero es una práctica habitual usar como nombre el del módulo:

```
import logging
logger = logging.getLogger(__name__)
```

De esta forma el nombre del logger refleja la estructura de paquetes y módulos que estemos usando, y es muy sencillo de usar.

Tambien podemos usar diferentes gestionadores para notificarnos, aparte de la consola y el fichero de textos, tenemos notificacines vía sockets, datagramas UDP, por correo, envios a un demonio syslog, a un buffer en memoria y, por supuesto, la posibilidad de crear nuestros propios manejadores.

traceback

Este módulo nos permite extraer, formatear e imprimir la traza de ejecución (El "camino" que ha seguido el programa hasta llegar a un determinado punto). Esta traza es la misma que se muestra cuando se eleva una excepción y nadie la captura. Es muy util cuando se quieren mostrar estas trazas y mensajes de error de forma controlada, por ejemplo es lo que hace iPython para mostrar mensajes de error con coloreado de sintaxis e información adicional.

El módulo trabaja con objetos de tipo `traceback`, que son los objetos que podemos encontrar en `sys.last_traceback`, o devueltos en tercer lugar en la tupla que retorna la función `sys.exc_info()`.

Algunos métodos utiles en este módulo son:

traceback.print_exc([limit[, file]])

Esta llamada obtiene la traza actual y la imprime en pantalla (o en un fichero, si se especifica), usando el mismo formato que usaría por defecto Python si la excepción no se captura.

traceback.format_exc([limit])

Es como `print_exc()`, pero devuelve una cadena de texto en vez de imprimirla.

pdb

El módulo `pdb` define un debugger interactivo para programas Python. Soporta *breakpoints* y *breakpoints* condicionales, ejecución paso a paso, inspección de la traza, listado del codigo fuente y evaluación de código Python arbitrario en el contexto del programa. Tambien puede ser llamada bajo el control del programa.

Podemos invocarlo desde la línea de comandos con:

```
$ pdb <script.py>
```

Cuando se sjecuta el debugger, el prompt cambia a `(Pdb)`. Podemos consultar una breve ayuda pulsando `help`. Los comandos más útiles puende ser:

h(elp)

Sin argumentosm imprime la lista de posibles ordener. Si la pasamos una orden como parámetro, ampliará la información sobre el mismo.

w(here)

Imprime la traza, con la actuividad más reciente al final. Una flecha indica en entorno actual

s(tep)

Ejecuta la línea actual, parandose en la primera ocasión que pueda: O bien en la primera línea de una función que se ha llamado o en la siguiente línea.

n(ext)

Continua la ejecución hasta que se alcanza la siguiente línea en el bloque actual o retorne de una función. La diferencia con `step` es que `step` entrará dentro del cuerpo de una función, mientras que next la ejecutará y seguirá hasta la siguiente línea.

r(eturn)

Ejecuta el resto de la función y retorna.

c(ont(inue))

Continua la ejecución. Solo se para si encuentra un *breakpoint* o si termina el programa.

l(ist) [first[, last]]

Lista el código fuente.

Podemos usar el debugger desde dentro del programa; lo habitual es ejecuta la siguiente línea antes de llegar al código problemático:

```
import pdb; pdb.set_trace()
```

Esto arrancará en modo debugger justo en esa línea. A partir de hay se puede avanzar a traves del código con `s` o `n`, o seguir la ejecucion con `c`.

Hay muchas más ordenes y usos disponibles. Consulta la documentación oficial de Python para ver todos las opciones.

3.5.3 De uso general

Expresiones regulares (re)

Este módulo permite trabajar con expresiones regulares. Una expresión regular viene a definir por un conjunto de cadenas, que cumplen ciertas condiciones. Si una cadena de texto pertenece al conjunto de posibles cadenas definidas por la expresión, se dice que *casan* o que ha habido una coincidencia (*match*).

Las expresiones regulares se crean combinando expresiones regulares más pequeñas (o primitivas). La cadena que define una expresión regular puede incluir caractereres normales o especiales. Los caracteres normales solo casan consigo mismo. Por ejemplo, la expresion regulas `a` solo casaria con una `a`. Los especiales, como `|` o `.`, tienen otros significados; o bien definen conjuntos de caracteres o modifican a las expresiones regulares adyacentes.

Algunos caracteres especiales son:

`.`

Punto. casa con cualquier caracter.

`^`

Casa con el principio de una string

`$`

Casa con el final de una string

`*`

Afecta a la re anterior. Casa con cualquier repetición (incluyendo el cero, es decir, ninguna) de la re. `ab*` casará con las cadenas `a`, `ab`, `abb`, `abbb`, ...

`+`

Afecta a la re anterior. Casa con una o más repeticiones de la re anterior. `ab+` casará con las cadenas `ab`, `abb`, `abbb`, ..., pero no con `a`.

`?`

Afecta a la re anterior. Casa con una o ninguna vez el patrón. `ab?` casará con `a` o con `ab`.

`{m}`

Afecta a la re anterior. Casa con exactamente `n` repeticiones del patron anterior. `a{6}` casará con `aaaaaa`.

`{m,n}`

Afecta a la re anterior. Casa con entre `m` y `n` repeticiones del patron anterior. `a{3:5}` casará con `aaa`, `aaaa` o `aaaa`

`\`

Normalmente "escapa" el significado del caracter a continuación, permitiendo asi incluir caracteres como `{` o `*` literales.

`|`

Alernancia entre dos patrons: A|B significa que casa con cuanlquir cadena que case con A o con B. Se pueden usar multiples patrones separados con `|`. `a|b|c` casa con `a`, `b` o `c`.

`[]`

Sirve para indicar un conjunto de caracteres. En un conjunto:

Los caracteres se pueden listar individualmente, como por ejemplo, `[abc]`, que casa con el caracter `a`, con `b` o con `c`.

También se pueden espcificar rangos de caracteres, usado el guión para separar los límites, por ejemplo `[a-z]` casa con cualquier letra minúsucula, `[0-9]` casa con cualquier dígito.

Los caracteres especiales pierden su significado dentro de los corchetes, no hace falta escaparlos.

Se puede definir el complemento del conjunto incluyendo como primer caracter `^`: `[^5]` casa con cualquier caracter, excepto el el cinco.

El uso de expresiones regulares es tremendamente potente y complejo, y hay varios libros dedicados al tema.

Ejercicio: Expresiones regulares para encontrar matrículas de coche.

Veamos el siguiente ejemplo, que busca matrículas de vehículos en un determinado texto, tal y como se describen en el sistema de matriculación vigente actualmente en España:

> El 18 de septiembre del año 2000 entró en vigor el nuevo sistema de matriculación en españa, introduciendo matrículas que constan de cuatro dígitos y tres letras consonantes, suprimiéndose las cinco vocales, y las letras Ñ, Q, CH y LL. [...] Si el vehículo es histórico, y se ha matriculado con una placa de nuevo formato, aparece primero una letra H en la placa.

El siguiente código lista las matrículas encontradas en el texto:

```
import re

Texto = '''INSTRUIDO por accidente de circulación ocurrido a las 09:43
entre la motocicleta HONDA 500, matrícula 0765-BBC  y la
motocicleta HARLEY-DAVIDSON , matrícula 9866-LPX, en el punto
kilométrico 3.5 de la carretera general del sur, término municipal de
Arona, Tenerife, y bla, bla, bla...'''
```

```
patron = re.compile('[H]?[0-9]{4}-?[BCDFGHJKLMNPRSTVWXYZ]{3}', re.IGNORECASE)
for matricula in patron.findall(Texto):
    print(matricula)
```

os: Acceso a llamadas del sistema operativo

Este módulo da acceso a llamadas del sistema operativo. A nivel de diseño, las llamadas que funcionana en todos los sistemas usan y devuelven la misma interfaz, independiente del S.O. Por ejemplo, la función `stat` devuelve información sobre un fichero en el mismo formato, independientemente de la plataforma.

Las funciones que solo están disponibles para un determinado sistema operativo estan en submódulos aparte.

El submodulo os.path (cargado automáticamente) incluye funciones de ayuda para trabajar con rutas de archivos.

Algunas de las funciones de este módulo son:

os.name¶

> El nombre del sistema operativo sobre el que se está ejecutando python. Actualmente existen los siguientes nombres posibles: 'posix', 'nt', 'os2', 'ce', 'java', 'riscos'.

os.environ

> Un diccionario que contiene las variables de entorno definidas en el sistema operativa. Los valores se obtiene la primera vez que se importa el modulo, por lo que no reflejan cambios hechos después, a no ser que se fuerze la recarga del módulo

os.walk(top, topdown=True, onerror=None, followlinks=False)

> Devuelve un iterador que nos permite examinar todo un sistema de archivos. Para cada directorio y subdirectorio en la raiz (indicada por `top`), incluyendo la propia raiz, el iterador devuelte una tupla de tres elementos (`dirpath`, `dirnames`, `filenames`). `dirpath` es una cadena de texto, la ruta del direcotorio. `dirnames` es una lista con los nombres de los subdirectorios dentro de `dirpath` (excluyendo los nombres especiales `.` y `..`). `filenames` es una lista de nombres de los ficheros que no son directorio en `dirpath`. En cualquier momento podemos tener una ruta absoluta a un archivo `f` en `filenames` haciendo `os.path.join(top, dirpath, f)`.

os.path.getsize(path)

> devuelve el tamaño, en bytes, del fichero

os.path.getmtime(path)

> devuelve el tiempo de la ultima modificación del archivo. El valor es en tiempo unix: el número de segundos desde la medianoche UTC del 1 de enero de 1970. Vease módulo `time`.

os.path.splitext(path)

> devuelve una tupla de dos elementos (`root`, `ext`). En la primera posición va la ruta completa del fichero, sin extensión, y en la segunda va la extension, de forma que `path == root + ext`.

Ejercicio: Calcular cuanto ocupan todos los ficheros de tipo PDF en un determinado directorio, incluyendo sus subdirectorios, si los hubiera. Listar los nombres absolutos, es decir, incluyendo la ruta desde la raiz.

sys: configuración específica del sistema

Este módulo porporciona acceso a algunas variables usadas o mantenidas por el propio interprete de Python. Siempre está disponible:

sys.argv

la lista de argumentos pasados al script de python. En la posicion o (sys.argv[0]) siempre va el nombre del script (depende del S.O. subyacente si incluye el nombre completo, incluyendo la ruta, o no).

sys.exc_info()

Esta función devuelve una tupla de tres valores con información sobre el error que está siendo tratado: Tipo de la excepcion, valor de la misma y traza de ejecución. Podemos usarla en una clausula `except` para obtener más información del error. Si se llama cuando no hay ninguan excepción enmarcha, devuelve una tupla de tres valores `None`.

sys.path

Una lista de cadenas de texto que especifican las rutas de búsqueda para los módulos y paquetes de python.

sys.platform

Un identificador de la version de Python en ejecución

Ejercicio: Modificar el ejercicio anterior para que busque ficheros con una extension que determinaremos nosotros, como un argumento del script.

El módulo difflib: Buscar las diferencias entre secuencias

Este módulo proporciona clases y funciones que nos permite comparar secuencias. Se puede usar, por ejemplo, para compara ficheros, considerados como secuencias de líneas, o textos, considerados como secuencias de caracteres. Las diferencias se pueden analizar e imprimir en diferentes formatos, incluyendo HTML y parches.

La clase `difflib.SequenceMatcher` permite comparar cualquier tipo de secuencia. El algoritmo que usa intenta conseguir la subcadena más largas de coincidencias consecutivas. A partir de ahi se aplica recursivamente tanto a la derecha com a la izquierda de dicha subcadena. No produce secuencias de ediciones mínimas, pero son más comprensibles para los humanos.

Ejercicio: Dada una secuencia de lineas de texto, encontrar la más parecia a un texto original.

El módulo collections: Otras estructuras de datos

Este módulo implemente ciertos contendores especializados a partir de los básicos: dicionarios, listas, conjunto y tuplas.

`namedtuple`

Tuplas cuyo contenido puede ser accedido tanto por posición como por nombre.

`deque`

Una doble lista encadenada especializada en realizar operaciones de añadir o quitar de los extremos con gran rapidez.

`Counter`

Un diccionarios especializado en llevar cuentas; asocia a cada clave un contador y tienen métodos adicionales para este tipo de estructura de datos. Es equivalente en otros lengajes al concepto de `multisets`.

`OrderedDict`

Una subclase del diccionario que recuerda el orden en que se han añadido sus elementos.

`defaultdict`

Una subclase del diccionario que llama a una función definida por nosotros cuando no encuentra una clave.

random

Este módulo implementa generadores de números pseudo-aleatorios para distintas distribucines. Para enteros, podemos hacer una selección uniforme dentro de un rango; para secuencias, una selección uniforme de un elemento. Podemos reordenar al azar -barajar- una secuencia y obtener una muestra al azar. También podemos trabajar con distribucions uniformes, normales (Gauss), logaritmica normal, exponencial negativa, y distribuciones gamma y beta.

Casi todas las funciones dependen de la función básica `random()`, que genera un numero al azar en el intervalo semiabierto [0.0, 1.0).

`random.seed([x])`

Inicializa el generador de números con un determinado valor. Si se omite, se usa un valor obtenido a partir de la fecha y hora actual

`random.random()`

Devuelve un número al azar en coma flotante en el intervalo [0.0, 1.0).

`random.randint(a, b)`

Genera un entero N al azar tal que a <= N <= b.

`random.choice(seq)`

Devuelve un elemento al azar de los perteneciente a la secuencia `seq` Si `seq` está vacio, eleva una excepción `IndexError`.

`random.shuffle(x[, random])`

Baraja la secuencia (internamente, es decir, no genera una nueva secuencia). El argumento opcional `random` es una función sin argumentos que devuelve un número en coma flotante en el intervalo [0.0, 1.0); por defecto es la función `random()`.

`random.gauss(mu, sigma)`

Distribución normal o de Gauss. `mu` es la media, `sigma` es la desviación estandar.

3.5.4 Trabajar con fechas y tiempos: time y datetime

time

Esté modulo proporciona funciones para trabajar con tiempos y fechas. La mayoría de las funciones realizan llamadas al S.O. subyacente.

Algunas consideraciones y terminología:

- UTC es el tiempo coordinado Universal, anteriormente conocido como GMT o Hora de Greenwich (El acrónimo UTC es un comprimiso entre el inglés y el francés)
- DST es el ajuste de horario de verano (*Daylight Saving Time*) una modificacion de la zona horaria, normalmente de una hora, que se realiza durante parte del año. las reglas de los DST son, en la práctica, pura magia (dependen de las leyes locales) y pueden cambiar de año a año,

Los valores de tiempo devueltos por `gmtime()`, `localtime()` y `strptime()`, y aceptados por `asctime()`, `mktime()` y `strftime()` son tuplas (En realidad, `namedtuple`) de 9 enteros: año, mes, dia, horas, minutos, segundos, día de la semana, día dentro del año y un indicador de si se aplica o no el horario de verano.

Algunas funciones definidas en este módulo:

`time.time()`

Devuelve el tiempo en segundos, en forma de número de coma flotante.

`time.gmtime([secs])`

Convierte un tiempo en segundos en una tupla de nueve elementos, en los cuales el flag final es siempre 0. Si no se indica el tiempo , se tomará el momento actual.

`time.localtime([secs])`

Como `gmtime()`, pero convertido a tiempo local. El indicador final se pone a uno si en ese momento estaba activo el horario de verano.

`time.mktime(t)`

La inversa de `localtime()`. Su argumento es una tupla de 9 elementos (Como el flag final es obligatorio, se puede poner -1 para indicar que no lo sabemos). Devuelve un número de segundos unix.

`time.sleep(secs)`

Suspender la ejecución del programa durante el tiempo en segundos indicado como parámetro.

Ejercicio: Averiguar el día de la semana en que nacieron -O cualquier otra fecha que les interese-.

datetime

El módulo `datetime` continua donde lo deja `time`. Proporciona clases para trabajar con fechas y tiempos, soportando por ejemplo aritmética de fechas.

La clase `datetime` sirva para trabajar con fechas y horas. Para trabajar con estos objetos hay que saber que podemos tener derivar dos tipos distintos de fechas/horas a partir de esta clase: las fechas absolutas o relativas.

Una fecha absoluta dispone de toda la información necesaria para poder determinar, sin ninguna ambigüedad, su valor. Sabe por tanto en que zona horaría está y, lo que es más complicado, si está activo o no el horario de verano. El horario de verano es un acuerdo político, administrado por cada país, por lo que suele ser cambiante, difícil de entender y, en general, caótico. La ventaja de este tipo de fecha/hora es que no está sujeta a interpretación.

Una fecha relativa, por el contrario, no tienen toda la información necesaria para que su valor sea indiscutible, lo que dificulta, por ejemplo, hacer comparaciones. Determinar si una fecha relativa está referida al Tiempo Coordinado Universal (UTC), la fecha y hora local o la fecha y hora en alguna otra zona horaria depende por entero del programa, de la misma forma que es responsabilidad del programa determinar si un número representa metros, micras o litros. Las fechas/tiempo locales son fáciles de entender y de usar, pero tenemos que pagar el coste que supone ignorar ciertos aspectos de la realidad.

Los tipos disponibles en este módulo son:

`class datetime.date`

Una fecha local, que asume que el *Calendario Gregoriano* siempre ha estado y siempre estará vigente. Tiene los atributos: `year`, `month` y `day`.

`class datetime.time`

Una marca de tiempo ideal, no sujeta a ninguna fecha en particular, y que asume que cada día tiene exactamente 24*60*60 segundos. Tiene los atributos: `hour`, `minute`, `second`, `microsecond` y `tzinfo`.

`class datetime.datetime`

Combinación de fecha y hora, con los atributos: `year`, `month`, `day`, `hour`, `minute`, `second`, `microsecond` y `tzinfo`

`class datetime.timedelta`

Representa una duración: La diferencia entre dos objetos de tipo `date` o `datetime`.

Estos tipos de datos son todos inmutables.

3.5.5 Módulos para trabajar con ficheros xml y csv

xml

Los diferentes módulos de Python para trabajar con ficheros XML están agrupados en el paquete `xml`. Las dos formas más habituales de trabajar con un fichero XML son DOM (*Document Object Model*) y SAX (*Simple API for XML*). Ambas están disponibles en los módulos `xml.dom` y `xml.sax` respectivamente. Usando el modelo DOM tenemos acceso a todo el árbol de una sola vez, lo que puede ser costoso en terminos de almacenamiento en memoría. Con SAX procesamos el árbol paso a paso, respondiendo ante ciertos eventos, a medida que se van abriendo y cerrando los nodos. Con esta segunda forma perdemos cierta flexibilidad pero no tenemos el problema del almacenamiento completo del árbol en memoria.

XML es un formato de datos jerárquico, con lo que la forma maás habitual de representarlo es un árbol. Para eso se definen las clases `ElementTree`, que representa todo el documento XML a tratar, y `Element`, que representa a un nodo dentro del árbol. Las interacciones con el documento como un todo, como por ejemplo leerlo o guardarlo en un fichero en disco, se hacen normalmente a nivel de `ElementTree`. Las interacciones con un elemento XML o sus subelementos se realizan en el nivel de `Element`.

Usaremos para explicar estos módulos el siguiente documento XML:

```
<?xml version="1.0"?>
<data>
    <country name="Liechtenstein">
        <rank>1</rank>
        <year>2008</year>
        <gdppc>141100</gdppc>
        <neighbor name="Austria" direction="E"/>
        <neighbor name="Switzerland" direction="W"/>
    </country>
    <country name="Singapore">
        <rank>4</rank>
        <year>2011</year>
        <gdppc>59900</gdppc>
        <neighbor name="Malaysia" direction="N"/>
    </country>
    <country name="Panama">
        <rank>68</rank>
        <year>2011</year>
        <gdppc>13600</gdppc>
        <neighbor name="Costa Rica" direction="W"/>
        <neighbor name="Colombia" direction="E"/>
    </country>
</data>
```

Lo más básico es importar y leer estos datos desde un fichero. Lo podemos hacer con el siguiente código:

```
import xml.etree.ElementTree as ET
tree = ET.parse('country_data.xml')
root = tree.getroot()
```

O también podemos leer los datos a partir de una variable de tipo string:

```
root = ET.fromstring(country_data_as_string)
```

Como `root` es un elemento (un objeto de la clase `Element`), tiene una etiqueta (`tag`) y un conjunto de atributos, en forma de diccionario:

```
>>> root.tag
'data'
>>> root.attrib
{}
>>>
```

Tambien tiene una serie de hijos, sobre los que podemos iterar:

```
>>> for child in root:
...   print(child.tag, child.attrib)
...
country {'name': 'Liechtenstein'}
country {'name': 'Singapore'}
country {'name': 'Panama'}
>>>
```

También podemos acceder a los hijos usando índices:

```
>>> root[0][1].text
'2008'
>>>
```

La clase `Element` define una serie de método que nos ayudan a recorrer recursivamente todo el subárbol que haya debajo de él (Sus hijos, nietos, etc...). Por ejemplo, el método `iter()`:

```
>>> for neighbor in root.iter('neighbor'):
...   print(neighbor.attrib)
...
{'name': 'Austria', 'direction': 'E'}
{'name': 'Switzerland', 'direction': 'W'}
{'name': 'Malaysia', 'direction': 'N'}
{'name': 'Costa Rica', 'direction': 'W'}
{'name': 'Colombia', 'direction': 'E'}
>>>
```

El método `Element.findall()` localiza sólo los elementos de una determinada etiqueta que son hijos directos del nodo actual. El método `Element.find()` encuentra el primer hijo que cumpla esta misma condición. Con `Element.text` podemos acceder al contenido textual del elemento, y con `Element.get` podemos acceder a los valores de sus atributos:

```
>>> for country in root.findall('country'):
...   rank = country.find('rank').text
...   name = country.get('name')
...   print(name, rank)
...
Liechtenstein 1
Singapore 4
```

```
Panama 68
>>>
```

Se pueden hacer operaciones de búsqueda aun más sofisticadas usando Xpath.

csv

El formato de fichero llamado :term:CSV (*Comma Separated Values* o Valores separados por comas) es uno de los más habitualmente usados para el intercambio de información de hojas de cálculo o bases de datos. A pesar de eso, no hay ningún estandar ni norma escrita, así que el formato esta definido de forma más o menosinformal por el conjunto de aplicaciones que pueden leerlo o escribirlo.

Esta carencia de estandares provoca que haya multiples, variadas y pequeñas diferencias entre los datos producidos o consumidos por diferentes aplicaciones. Por esta razón, trabajar con distinto ficheros CVS provinientes de distintas fuentes suele dar más de un dolor de cabeza. A pesar de estas divergencias (empezando por que caracter usar como separador de campos), es posible escribir un módulo que pueda maniputar de forma eficiente estos datos, ocultado al programador los detalles específicos de leer o escribir estos ficheros,

El módulo csv permite escribir y leer estos archivos. El programador puede especificar, por ejemplo, "escribe este archivo en el formato preferido por excel", o "lee este fichero como fuera de excel, pero usando el carácter `:` como separador de campos". También nos permite definir nuestros propios formatos de uso particular, que el módulo denomina "dialectos".

Las funciones `reader()` y `writer()` leen y escriben secuencias.

Un ejemplo sencillo de lectura:

```
import csv
with open('some.csv', 'r') as f:
    reader = csv.reader(f)
    for row in reader:
        print row
```

Y uno de escritura:

```
import csv
datos = [
    ('Leonardo', 'Azul', 1452),
    ('Raphael', 'Rojo', 1483),
    ('Michelangelo', 'Naranja', 1475),
    ('Donatello', 'Violeta', 1386),
    ]
with open('some.csv', 'wb') as f:
    writer = csv.writer(f)
    writer.writerows(datos)
```

3.5.6 Módulos para trabajar con ficheros comprimidos

zipfile — Soporte para archivos ZIP

El archivo ZIP es un formato estándar de archivado y compresión de archivos. Este módulo proporciona mecanismos para crear, leer, escribir, modificar y listar archivos ZIP. Soporta trabajar con ficheros ZIP cifrados, pero por el momento no puede crearlos. El descifrado es particularmente lento, porque no está implementado en C.

La función `is_zipfile()` devuelve un booleno indicando si el fichero que se le pasa como parámetro es un archivo ZIP o no.

La clase `ZipFile` nos permite trabajar directamente con un archivo ZIP. Tiene métodos para obtener información sobre los ficheros contenidos en el archivo, así como para añadir nuevos ficheros a un archivo.

Por ejemplo, para leer los nombres de los ficheros contenidos deontr de un archivo ZIP, podemos hacer:

```
import zipfile

zf = zipfile.ZipFile('example.zip', 'r')
print zf.namelist()
```

gzip - Soporte para ficheros gzip

El módulo `gzip` nos proporciona los medios para comprimir o descomprimir ficheros igual que lo hacen los programas unix `gzip` y `gunzip`. Al contrario que con el formato ZIP, el formato gzip solo permite comprimir y descomprimir un fichero, porque no tiene capacidad de archivado (Es decir, la posibilidad de añadir varios ficheros dentro del archivo).

El módulo `gzip` proporciona la clase `GzipFile`, que imita a un objeto de tipo `file` de Python. Los objetos instanciados de esta clase leen y escriben ficheros con el formato gzip. La compresión y descompresión es realizada automáticamente, por lo que el programandor puede trabajar con el fichero como si fuera un fichero normal.

Un ejemplo de como leer un fichero comprimido:

```
import gzip
f = gzip.open('file.txt.gz', 'rb')
file_content = f.read()
f.close()
```

Como crear un fichero comprimido con gzip:

```
import gzip
content = "Lots of content here"
f = gzip.open('file.txt.gz', 'wb')
f.write(content)
f.close()
```

Como comprimir un fichero ya existente:

> import gzip f_in = open('file.txt', 'rb') f_out = gzip.open('file.txt.gz', 'wb') f_out.writelines(f_in) f_out.close() f_in.close()

3.5.7 Internet

urllib2 — Librería para abrir URLs

Nota: Diferencias entre Python 2.x / Python 3.x

El módulo `urllib2` ha sido dividido en dos en Python 3.x: `urllib.request` y `urllib.error`. La herramienta `2to3` adapta automáticamente estos imports.

Este módulo nos permite abrir y trabajar con direcciones de internet (URLs). La función más usada del módulo es la siguiente:

```
urllib2.urlopen(url[, data][, timeout])
```

Que abre la url indicada, dandonos un objeto similar a un fichero.

En el parámetro opcional `data` podemos incluir información adicional que requieren ciertas peticiones web, especialmente `POST`. Si se incluye, `data` debe estar formateada con el estándar `application/x-www-form-urlencoded`, algo que podemos conseguir usando la función `urllib2.urlencode()`, que acepta como parámetro un diccionario o una secuencia de parejas (2-tuplas), y deveulve una string en dicho formato.

El otro parámetro opcional, `timeout`, indica el tiempo en segundos que debemos esperar antes de descartar por imposible una conexión.

El objeto devuelto, además de comportarse como un archivo, dispone de tres métodos adicionales:

`geturl()`

Devuelve la URL del recurso recuperado. Esto se utiliza noprmalmente para determinar si ha habido alguna clase de recirección.

`info()`

Devuelve la meta-información sobre el recurso solicitado, como las cabeceras, en forma de una instancia de la clase `mimetools.Message`.

`getcode()`

Devuelve el código de estado del protocolo HTTP de la respuesta.

Ejemplo: Salvar una página de Internet en un fichero local:

```
import urllib2

url = 'http://www.python.org/'
f = urllib2.urlopen(url)
with open('python.html', 'w') as salida:
    for linea in f.readlines():
        salida.write(linea)
f.close()
```

smtplib — cliente de protocolo SMTP

El módulo `smtplib` define un cliente del protocolo :term:SMTP (*Simple Mail TRansfer Protocol*), que puede ser usado para enviar correo electrónico a cualquier ordenador en Internet que esté ejecutando un demonio SMTP o ESMTP.

El siguiente ejemplo compone un mensaje, ayudándose de la clase `Message` definido en `email.message`. Las variables `gmail_user` y `gmail_password` están definidas en el código, lo que quizá no sea la mejor de las ideas posibles. Una vez creado el mensaje, se realiza la conexión al servidor de correo, que en este caso es el de Google Mail. La conexión en este caso es un poco más complicada de lo que sería con un servidor SMTP local, en la que la seguridad a lo mejor es un poco más laxa:

```
from email.message import Message
from smtplib import SMTP

gmail_user = 'tuusuario@gmail.com'
gmail_password = 'tu contraseña'

# Creamos el mensaje
msg = Message()
msg['to'] = 'euribates+test@gmail.com'
msg['from'] = 'euribates@gmail.com'
msg['subject'] = 'Esto es una prueba!'
msg.set_payload('Hola, mundo\n\n-- Juan')
```

```
# Lo enviamos
print('Enviando correo', end=' ')
smtpserver = SMTP("smtp.gmail.com", 587)
smtpserver.ehlo()
smtpserver.starttls()
smtpserver.ehlo
smtpserver.login(gmail_user, gmail_password)
smtpserver.sendmail(gmail_user, msg['to'], msg.as_string())
smtpserver.close()
print('[OK]')
```

Aunque el formato de los mensajes es realmente sencillo, usar la clase `Message` nos permite incluir de forma rápida y sencilla funcionalidades más elaboradas, como anexar ficheros o enviar múltiples versiones del mismo contenido.

SimpleHTTPServer — Simple HTTP request handler

Nota: Diferencias entre Python 2.x / Python 3.x

En Python3.x el módulo SimpleHTTPServer ha sido asimilado por `http.server`. La herramienta `2to3` adapta automáticamente estos imports.

Este módulo define una serie de clases que nos permiten implementar nuestros propios servidores web. La clase SimpleHTTPRequestHandler (Definida en el módulo `SimpleHTTPServer` en Python 2.x y en `http.server` en Python 3.x) es un servidor de ejemplo básico que sirve los ficheros del directorio donde se ha ejecutado, mapeando la estrucura de directorios como páginas web.

La mayor parte del trabajo, como analizar las peticiones, por ejemplo, lo hace la clase de la que deriva, `BaseHTTPServer`, la clase de ejemplo solo tienen que sobreescribir los métodos `do_GET()` y `do_HEAD()`.

El siguiente programa usa la clase de ejemplo para arrancar un servidor web básico, escuchando en la máquina local y en el puerto 8000:

```
import SimpleHTTPServer
import SocketServer

PORT = 8000

Handler = SimpleHTTPServer.SimpleHTTPRequestHandler
httpd = SocketServer.TCPServer(("", PORT), Handler)
print("serving at port", PORT)
httpd.serve_forever()
```

Pero puede ser aun más fácil, usando la opcion `-m` en el interprete para que ejecute el módulo como si fuera el programa principal, y opcionalmente indicando el número de puerto al que se vincula el servidor.

Para Python 2.x:

```
$ python -m SimpleHTTPServer 8000
```

Para Python 3.x:

```
$ python3 -m http.server 8000
```

hashlib - hashes y códigos de verificación e integridad

El módulo `hashlib` define una interfaz común a una serie de algoritmos conocidos como *funciones de hash criptográficos* o *funciones resumen*: SHA1, SHA224, SHA256, SHA384 y SHA512, así como el algoritmo MD5 de RSA (Definido como estándar en el RFC 1321).

Su uso en muy sencillo: Por ejemplo, usamos `md5()` para crear un objeto. A partir de ahi, podemos ir actualizando los datos sobre los que se tienen que hacer el *hash* con sucesivas llamadas a su método `update()`. Hacer una serie de llamadas sucesivas con partes del texto es equivalente a hacer un solo `update()` con todo el texto concatenado en un único valor; en otras palabras:

```
m.update(a); m.update(b)
```

es equivalente a:

```
m.update(a+b)
```

Durante cualquier momento del proceso se puede pedir el código de *hash*. Por ejemplo, para obtener el *hash* criptográfico de la frase "Su teoría es descabellada, pero no lo suficente para ser correcta.", podemos hacer:

```
import hashlib

m = hashlib.md5()
m.update("Su teoría es descabellada".encode('utf-8'))
m.update(", pero no lo suficente".encode('utf-8'))
m.update(" para ser correcta.".encode('utf-8'))
print(m.hexdigest()) # 46c8a761de36c7306532ae6f1013164c
m = hashlib.md5()
m.update('Su teoría es descabellada, pero no lo suficente para ser correcta.'.encode('utf-8'))
print(m.hexdigest())  # 46c8a761de36c7306532ae6f1013164c
```

El código obtenido depende de los datos suministrados, de forma que cualquier alteración, por mínima que sea, en el texto original, provocará una alteración enorme en el código de salida. Por ejemplo, veamos como cambia el resultado simplemente cambiando una coma de lugar [2]:

```
>>> from hashlib import md5
>>> print(md5('Perdón imposible, ejecutar prisionero').hexdigest())
eafd88022b53be13af86520a6a221024
>>> print(md5('Perdón, imposible ejecutar prisionero').hexdigest())
2b4360dbca5fd7b7b5df3fc4af7bab24
```

[2] El ejemplo se basa en una anécdota apócrifa atribuida al emperador Carlos V, de la que circulan varias versiones. Ésta es una de ellas:
Estando el rey en el teatro, le recordaron que tenía que decidir si indultaba o no a un condenado a muerte, decisión que había aplazado en su última audiencia, pero que ahora corría prisa, pues la ejecución estaba prevista para el día siguiente. Como respuesta, escribió en un billete: "Perdón imposible ejecutar al reo". El secretario que llevaba el papel se dió cuenta de que la vida del prisionero estaba en sus manos, y dependía de dónde se añadiese la coma que, evidentemente, faltaba. Si se decía "Perdón imposible, ejecutar al reo", el condenado era hombre muerto, pero si se escribía "Perdón, imposible ejecutar al reo", se salvaba.

Día 3.- Python y Librerías externas

4.1 Librerías externas

4.1.1 iPython

iPython es un interprete interactivo de Python, similar al interprete que viene instalado por defecto, pero con capacidades ampliadas y mejoradas, pensando especialmente en la comunidad científica. La idea es conseguir un entorno para cálculos y operaciones que nos permitan explorar ideas, investigar posibilidades y modificar modelos de forma interactiva. Para ello, iPython se basa en dos componentes:

- Un entorno de Python interactivo mejorado
- Una arquitectura para soportar computación paralela interactiva

Las caracteristicas más detacadas de iPython son las siguientes:

Autonumeración

iPython va numerando sucesivamente tanto las entradas y las salidas, al estilo de herramientas como **Matlab_** o **Octave_**. Esta numeración facilita el poder luego referirnos a nuestros pasos previos.

Completado automático

Usando la tecla TAB, obtenemos completado automático de código, normalmente dependiente del contexto, es decir, que nos muestra el completado de código dependiendo de lo escrito antes. Por ejemplo si tenemos un objeto en memoria `o`, simplemente escribiendo `o.<tab>` obtenemos un listado de sus métodos y atributos. Pero si estamos escribiendo una ruta dentro del sistema de ficheros -incluso aunque sea dentro de una string- el autocompletado mostrara los ficheros o directorios correspondientes. De igual manera, si importamos un módulo o paquete podemos examinar rápidamente su contenido usando el autocompletado.

Explorar objetos

Tecleando `<objeto>?` accedemos a un resumen de las características, métodos, atributos y documentación del objeto. Como en Python prácticamente todo es un objeto, podemos usar este sistema para obtener información instantanea de casi todo. Para salir de este sistema de ayuda, si fuera necesario, pulsar `q`.

Funciones mágicas

Python define una serie de funciones mágicas que se pueden invocar de la misma manera que si fueran ordenes de la línea de comandos. Estas funciones u ordenes mágicas empiezan con el caracter especial `%` seguido de su nombre, y pueden tambien aceptar argumentos, separados por espacios, igual que si fuera una orden de línea de comandos. Hay dos tipos de funciones, las que implican solo una linea (En cuyo caso la orden viene precedida por un solo caracter `%`) y las que implican una "celula", (Que vienen precedidas por dos caracteres `%`). Estas últimas se aplican a un bloque de código.

Por ejemplo, la funcion mágica `#timeit`, similar a las funciones de rendimiento que vimos en el módulo de la librería estándar del mismo nombre, puede ser usada para que afecte a una sola línea:

```
In [1]: import re
In [2]:%timeit re.compile('foo|bar')
1000000 loops, best of 3: 852 ns per loop

In [3]:
```

O a un bloque:

```
In [14]:%%timeit
...: x = range(1000)
...: max(x)
...:
10000 loops, best of 3: 32.2 us per loop

In [15]:
```

Algunas de las funciones mágicas disponibles son:

- Funciones que actuan con al código:: `%run`, `%edit`, `%save` y `%macro`, entre otras.
- Funciones que afectan al interprete: `%colors`, `%xmode`, `%autoindent`, etc.
- Otras funciones, como `%reset`, `%timeit` o `%paste`.

Se puede obtener una explicacion del systema de llamadas mágicas con `%magic`, o usar tambien el mecanismo de ayuda `?`, por ejemplo:

```
In [8]%run?
```

Podemos obtener un listado de todas las funciomes mágicas disponibles con `%lsmagic`.

Ejecutar y editar código

La orden mágica `%run` nos permite ejecutar cualquier script de Python y cargar todos sus datos directamente en nuestro interprete, como si se hubiera tecleado a mano. Como el fichero es releido de disco cada vez que hagamos un `%run`, los cambios que se hagan en el archivo se pueden ver reflejados inmediatemente, al contrario que los módulos, que tiene que ser recargados de forma especifica con `reload`. iPython también incluye la función `dreload()`, una versión recursiva de `reload`.

La orden %run tiene unos indicadores especiales para medir el tiempo de ejecución del script: `-t` o para ejecutarlo bajo el control de un debugger: `-d` o del profiler: `-p`.

La orden %edit abre nuestro editor favorito (Hay que especificarlo en los ficheros de configuración de iPython, en `~/.ipython/profile_default` o `~/.config/ipython/profile_default`). Cuando salimos del editor, el código se ejecuta como si lo hubieramos escrito directamente.

These will be placed in ~/.ipython/profile_default or ~/.config/ipython/profile_default, and contain comments explaining what the various options do.

Debugging

Después de que ocurra una excepción, se puede llamar a `%debug` para ejecutar de inmediato el debugger (`pdb`) en modo análisis forense y examinar el problema. Si ejecutamos `%pdb`, entraremos en el debugger en cuanto se produzca la primera excepción no capturada.

Historial

iPython almacena todas las ordenes que recibe, así como sus resultados. Se puede acceder facilmente a este histórico de comando pulsando las teclas de fecha arriba o abajo.

El historial se almacena en una variables llamadas `In` y `Out`, que pueden ser accedidas usando como índices los números de línea que iPython proporciona automáticamente. Podemos acceder a las tres últimas salidas usando las variables llamadas _, __ y ___.

Se puede usar la función mágica `%history` para examinar el historico. Hay otras funciones mágicas que también acceden a este histórico: `%edit`, `%rerun`, `%recall`, `%macro`, `%save` y `%pastebin`. Podemos usar el formato habitual para referirnos a un conjunto de líneas, e incluso referirnos a sesiones anteriores:

```
%pastebin 3 18-20 ~1/1-5
```

La orden anterior tomará la línea 3 y las líneas 18 a la 20 de la sesión actual, y las lineas 1 a la 5 de la sesión anterior, y las subirá al servicio pastebin, devolviendo la URL correspondiente.

Comandos del sistema

Se puede ejecutar cualquier comando de la shell, simplemente añadiendo el caracter `!` como prefijo, por ejemplo:

```
In [1]: !uname
Linux

In [2]:
```

Se puede capturar la salida, por ejemplo, para obtener una lista de los ficheros en el directorio actual podemos hacer `files = !ls`. Para pasar valores almacenados en variables python debemos prefijarlas con `$`, por ejemplo:

```
In [1]: patron = "*.py"
In [2]: files = !ls $patron
In [3]: files
Out[3]:
['comentarios.py',
 'Database.py',
 'endirecto.borrame.py',
 'en_directo.py',
 'Filters.py',
 'Form.py',
 'test_json.py']
In [4]:
```

Con un doble dolar `$$` podemos pasar el símbolo de `$` literal a la shell, para poder accedr a las variables de entorno como `PATH`.

4.1.2 iPython notebook

IPython notebook permite usar todas las capacidades de IPython pero sustitutendo el entorno en consola con una página web acceisble desde cualquier browser.

La nueva interface permite, además de usar todas las capadicades de iPython, incorporar al flujo de trabajo, ademas del código python, textos, expresiones matemáticas, gráficos, vídeos y practicamente cualquier contenido que un navegador modernos sea capaz de mostrar.

Podemos arrancar este entorno con la orden:

```
$ ipython notebook
```

Se pueden salvar las sesiones de trabajo como documentos, que mantienen todos estos elementos y que pueden ser almacenados en sistemas de control de veriones, o enviados por correo electrónico osalvados como ficheros HTML o PDF para imprimir o publicar estáticamente en la web. El formato interno de almacenamiento es json, que puede ser manipulado con facilidad para exportar a otros formatos.

- Crear un notebook
- Ejecutar un código Python
- Introducor texto
- Formulas matemáticas (http://www.mathjax.org/)
- Salvar como HTML/PDF (Estático)
- Salvar como notebook

4.1.3 Python Image Library (PIL) Procesado de imágenes

4.1.4 numPy Trabajndo con datos numéricos

4.1.5 Matpltlib

4.1.6 Scipy

4.1.7 Panda

4.1.8 Networkx

4.1.9 Scrapy

4.1.10 PyX

4.1.11 Scikit Machine Learning con Python

4.1.12 Interfaz con C

4.1.13 Waitress

Waitress is meant to be a production-quality pure-Python WSGI server with very acceptable performance. It has no dependencies except ones which live in the Python standard library. It runs on CPython on Unix and Windows under Python 2.6+ and Python 3.2+. It is also known to run on PyPy 1.6.0+ on UNIX. It supports HTTP/1.0 and HTTP/1.1.

For more information, see the "docs" directory of the Waitress package or http://docs.pylonsproject.org/projects/waitress/en/latest/ .

4.2 Ejemplo de uso de python embebido

4.2.1 Gimp

4.2.2 Blender

4.2.3 Inkscape

4.3 Desarrollo dirigido por pruebas: TTD

4.4 Python one liners

Apéndice 1: Para programadores con experiencia en otros lenguajes

Python, como cualquier otro lenguaje, tiene sus formas particulares de realizar algunas tareas, algunas de ellas pueden ser más sorprendentes para programadores que provengan de otros lenguajes que para una persona que empieze de cero. [Escena del titanic]

Vamos a ver algunas de estas modismos o costumbres que pueden sorprender a los más experimentados.

5.1 Para definir bloques de código se usa el sangrado

Es decir, no hay marcas de principio y fin de bloque, como en Pascal, Delphi (BEGIN, END) o C, Java, C# (Llaves de apertura y cierre { y }). La indentación del código marca el principio del bloque (cuando aumenta) y el fin del bloque (cuando desminuye). Esto parecerá extraño a la mayoría de los programadores, que están acostumbrados a que los espacios no sean significativos.

Sin embargo, tiene muchas ventajas:

- La mayoría de los desarrolladores ya indentan el código de todas maneras. Usar el indentado para marcar los límites de los bloques de código simplifica la escritura y, sobre todo, la lectura del mismo. Presentele un trozo de código java relativamente complejo y sin ninguna indentación a un programador y lo primero que hará este, en un 99 % de los casos [1], es indentar el código a su gusto mientras lo lee para entender como funciona.
- En otros lenguajes, el indentado solo tiene una función decorativa, es una forma de simplificar la lectura del mismo, pero no tiene ningún significado real; la estructura será la que indiquen los marcadores de principio y fin de código. Muchos programadores se han dejdo las pestañas intentando encontrar un error en el flujo del programa porque ha indentado mal (o ha indentado bien, pero se le han escapado un par de llaves, por ejemplo). Si el indentado y las marcas no concuerdan, puede ser un problema, porque es mucho más fácil leer el indentado que las marcas, sobre todo si el código es extenso.
- No hay distintas formas de indentar código. En C y sus derivados hay tantas formas que incluso se agrupan por familias, según su semejanza. Casi podriamos decir que hay tantos estilos de indentación como desarrolladores. En Python solo hay que limitarse a decidir entre espacios y tabuladores –lo recomendado son espacios– y en su caso, cuantos espacios usar para cada nivel de indentación –lo recomendado son 4 espacios–.
- Además, nos ahorramos dos caracteres o palabras reservadas, que se pueden usar en otras partes.

[1] Si perteneces al 99 % te extrañará que exista siquiera ese 1 %. Hay gente para todo.

- De todas formas, ibas a indentar el código, ¿no?

5.2 No hay métodos ni propiedades privadas

En otros lenguajes orientados a objetos como C++ o java es posible proteger determinados métodos o atributos de nuestras clases, de forma que sea imposible usarlas y/o modificarlas. En Python no se puede [2], todos los métodos y atributos son públicos. No existe nada que sea "privado" en el sentido de Java o C++.

Existe la convenión de marcar los atributos y métodos internos con el caracter subrayado. Esto no tiene ningún significado especial para el lenguaje (Excepto en el caso de los módulos cuando se importa todo el contenido con `from module import *`, en cuyo caso no se importan ningún nombre que empiece por el caracter subrayado). Nada impide que accedas a un attributo o nombre que empiece por _, pero ha de entenderse que es de uso interno, que no deberías jugar con él a no ser que sepas muy bien lo que estás haciendo, y que si en un futuro tu código deja de funcionar porque ese atributo o método ha desaparecido, no podrás culpar a nadie más que a ti mismo.

Eso si, no es un fallo en el lenguaje, es una decisión consciente y forma parte del diseño del lenguaje. La documentación de Python lo resume de la siguiente manera: "Aquí somos todos adultos". Algunos consideramos que la misma idea de ocultar o esconder parte del código es "poco pythonico". Así, ninguna clase ni ningún objeto puede mantener sus mecanismos internos ajenos al resto de los desarrolladores. Esto hace que la introspección sea sencilla y potente.

La filosofía es que Python confía en ti y en tus habilidades. Viene a ser algo así: "Si consideras necesario meterte por los recovecos y usar métodos que no están diseñados para el usuario final, adelante, pensaremos que tienes una buena razón para hacerlo, pero no digas luego que la culpa es nuestra. Aquí somos todos adultos y todos conocemos las reglas del juego".

Perl tiene una filosofía similar que expresa de la siguiente forma: "[Los modulos] de Perl prefieren que te mantengas fuera de su sala de estar, pero que lo hagas porque no estás invitado, no porque tengas una escopeta de cañones recortados."

5.3 Estructuras de datos integradas en el lenguaje

En otros lenguajes, hay estructuras de datos como pilas, colas, mapas (hash), tuplas, etc... que, por su gran utilidad, están implementadas como librerías. Python da un paso más alla, y estas estructuras de datos, entre otras, forman parte nativa del lenguaje. Esto permite que el lenguaje interactue con estas estructuras de forma mucho más fluida.

El bucle `for`, por ejemplo, está diseñado nativamente para que itere sobre aquellas estructuras de datos que sean "iterables", de las cuals la lista es el más habitual, pero no el único ni mucho menos. Veamos lo que significa esto con un ejemplo: Para imprimir una lista de nombres guardados en la variable `lista`, en C, haríamos:

```
include <stdio.h>

void main(int argc, char *argv[]) {
    char * lista[] = {"hola", "mundo", "cruel"};
    int i, n = sizeof(l)/sizeof(char *);
    for (i=0; i<n; i++) {
        puts(l[i]);
        }
    }
```

en Python, sería:

[2] En realidad si se puede, porque en Python se puede hacer casi todo, pero es poco pythonico, la sintaxis es confusa y las razones de uso casi siempre inexistentes.

```
lista = ("hola", "mundo", "cruel")
for s in lista:
    print s
```

El resultado es el mismo en los dos casos, pero la legibilidad es mucho mayor en el segundo. No hay ni cálculo de tamaño, ni comprobaciones para no superar el límite, ni incremento de variables auxiliares ni, ya puestos, variables auxiliares. La magia no existe, la operaciones siguen siendo necesarias, pero se hacen internamente, con más rápidez y menos posibilidad de error [3].

5.4 Las Funciones pueden devolver más de resultado

En otros lenguajes, las funciones solo pueden devolver un único resultado. En python, gracias al empaquetado y desempaquetado automático de tuplas, las funciones pueden devolver más de una variable. Veamos un ejemplo:

```
def division_y_resto(dividendo, divisor):
    return dividendo // divisor, dividendo % divisor

cociente, resto = division_y_resto(47, 9)
print 'cociente:', cociente
print 'resto:', resto
```

Este pequeño programa nos informa de que 47 dividido por 49 da cinco, con resto dos, o dicho de otra manera, que (9 * 5) + 2 = 47

5.5 Las asignaciones pueden encadenarse

Gracias a la magia de las tuplas y al empaquetado y desempaquetado automatico de las mismas, junto con algún que otro truco, las expresiones siguientes son posibles:

```
>>> a = b = c = d = 0
```

Y significan lo que uno podría esperarse, las variables `a`, `b`, `c` y `d` se inicializan a cero.

También podemos intercambiar los valores de dos variables sin necesidad de recurrir a variables intermedias:

```
>>> a,b = b,a
```

Las comparaciones también pueden escribirse de forma más legigle que en otros lenguajes, por ejemplo, para comprobar que la variable `a` está entre cero y cien, podemos expresarlo así:

```
>>> if a > 0 and a < 100:
...     print 'OK'
```

o, más legible:

```
>>> if 0 < a < 100:
...     print 'OK'
```

[3] En C uno de los errores más frecuentes era acceder con un puntero a direcciones de memoria posteriores a las que ocupaba una variable, provocando todo tipo de fallos. Eran tantos los errores de este tipo que incluso recibieron un nombre: *buffer overrun* o desbordamiento de buffer.

5.6 No hay necesidad de implementar funciones getters/setters

En algunos lenguajes, especialmente C++ y derivados, se considera una mala práctica el acceder directamente a los atributos de una clase. Se recomienda siempre definir, para cada atributo, dos funciones: una se usará para asignar valores al atributo (*setter*) y la otra para leer el valor del atributo (*getter*). La mayor parte de los IDE permiten incluso generar automáticamente estos métodos para simplificar el trabajo al programador.

El razonamiento detrás de esta práctica es que no podemos predecir, a medida que el programa crece o evoluciona, si un atributo no podrá convertirse en algo más complejo; algo que requiera determinados cálculos o que produzca efectos paralelos al ser leido o modificado. En este caso, habría que buscar las referencias al atributo y cambiarlas por las correspondientes llamadas a métodos, algo que puede convertirse en una tarea titánica si el código es muy grande. Curándose es salud, los programadores en estos lenguajes nunca acceden al atributo directamente, sino que usan siempre los métodos correspondientes.

Supongamos, por ejemplo, que hemos definido una clase para guardar información de los empleados, y que hemos reservado el atributo edad para almacenar cuantos años tiene cada empleado. Un ejemplo de esta clase podría ser:

```
class Empleado:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    def __str__(self):
        return self.nombre

tyson = Empleado('Neil deGrasse Tyson', 54)
```

Nuestra clase es un éxito, y se empieza a usar en diferentes partes del programa; por ejemplo, alguien de recursos humanos, seguramente con aviesas intenciones, decide escribir una función para saber si un empleado está cerca de su jubilación:

```
def es_jubilable(empl):
    return empl.edad > 65

es_jubilable(tyson) #--> False
```

Sin embargo, nuestro diseño tiene un fallo: es complicado mantener el campo *edad*. Es mucho más práctico almacenar la fecha de nacimiento: contamos con más información -podemos felicitar automáticamente al empleado en su cumpleaños, por ejemplo- y no tenemos que actualizar el campo edad de todos los empleados cada año. Manos a la obra:

import datetime

class Empleado:

def __init__(self, nombre, f_nacimiento): self.nombre = nombre self.f_nacimiento = f_nacimiento

def get_edad(self): delta = datetime.date.today() - self.f_nacimiento return int(data.days // 365.25)

tyson = Empleado('Neil deGrasse Tyson', datetime.date(1958, 10, 5))

Sin embargo, esto rompe el código recursos humanos, ya que el atributo edad ha desaparecido.

Para evitar esto, un programar con un *background* de Java podría haber previsto este problema, y creado la clase `Empleado` con sus correspondientes *setters* y *getters*:

```
class Empleado:
    def __init__(self, nombre, edad):
        self.set_nombre(nombre)
```

```
        self.set_edad(edad)

    def __str__(self):
        return self.get_nombre()

    def set_edad(self, edad):
        self._edad = edad

    def get_edad(self):
        return self._edad

    def set_nombre(self, nombre):
        self._nombre = nombre

    def get_nombre(self):
        return self._nombre

tyson = Empleado('Neil deGrasse Tyson', 54)
```

El empleado de recursos humanos no habría accedido nunca a la valiable interna `_edad`, habría usado los métodos de acceso correspondientes:

```
def es_jubilable(empl):
    return empl.get_edad() > 65

es_jubilable(tyson) #--> False
```

Con lo que, al hacer el cambio de edad for fecha de nacimiento, no se rompe ningún código:

```
class Empleado:
    def __init__(self, nombre, f_nacimiento):
        self.set_nombre(nombre)
        self.set_f_nacimiento(f_nacimiento)

    def __str__(self):
        return self.nombre

    def set_edad(self, edad):
        raise ValueError('No se puede asignar la edad; modifique la fecha de nacimiento')

    def get_edad(self):
        delta = datetime.date.today() - self.f_nacimiento
        return int(data.days // 365.25)

    def set_nombre(self, nombre):
        self._nombre = nombre

    def get_nombre(self):
        return self._nombre

    def set_f_nacimiento(self, f_nacimiento):
        self.f_nacimiento = f_nacimiento

    def get_f_nacimiento(self):
        return self.f_nacimiento

tyson = Empleado('Neil deGrasse Tyson', datetime.date(1958, 10, 5))
```

El problema es que la clase se nos ha complicado innecesariamente; los metodos get_nombre, *set_nombre*,

set_f_nacimiento y *get_f_nacimiento* no aportan nada. Es verdad que segurmanete el IDE las ha generado automáticamente por nosotros, pero recuerdese que uno de los principios de diseño de Python era que es preferible que el código sea fácil de leer, y no es el caso.

La solución "pythonica" pasa por el uso de propiedades, property. Esta funcion nos permite redefinir como se accede, modifica o borra un atributo de un objeto, sin que el resto del código sea consciente de que el atributo al que estaba accediendo ya no es un simple campo. En el caso anterior, reescribiriamos la clase *Empleado* así:

```
import datetime

class Empleado:
    def __init__(self, nombre, f_nacimiento):
        self.nombre = nombre
        self.f_nacimiento = f_nacimiento

    def __str__(self):
        return self.nombre

    def set_edad(self, edad):
        raise ValueError('No se puede asignar la edad; modifique la fecha de nacimiento')

    def get_edad(self):
        delta = datetime.date.today() - self.f_nacimiento
        return int(delta.days // 365.25)

    edad = property(get_edad, set_edad)

tyson = Empleado('Neil deGrasse Tyson', datetime.date(1958, 10, 5))
```

5.7 Las funciones son objetos

Las funciones son objetos en si mismo, es decir, que podemos hacer con ellas cosas que en otros lenguajes serían imposibles. Por ejemplo, podemos tener un array de funciones, o podemos pasar una función –ojo, no el resultado de una función, sino la función en si– como parámetro de otra función. Esto no sorprenderá en absoluto a aquellos que hayan tenido experiencia con lenguajes funcionales, pero si a aquellos que sólo estén acostumbrados a lenguajes imperativos.

Apéndice 2: “One liners”

6.1 Codificar/decodificar un texto o fichero con base64::

Base 64 es un sistema de numeración posicional que usa 64 como base. Esta es la mayor potencia de dos que puede ser representada usando únicamente los caracteres imprimibles de ASCII puro. Se usa para codificación de correos electrónicos, PGP y otras aplicaciones:

```
$ python -m base64 -e <input
$ echo "hola, mundo" | python -m base64 -e

$ python -m base64 -d <input
$ echo "aG9sYSwgbXVuZG8K" | python -m base64 -d
```

6.2 “Cifrar” texto con ROT-13

Cifra un texto en ROT-13 (Rotar 13 veces). Este cifrado es demasiado sencillo para que sea utilizable, excepto para evitar una lectura casual: ROT-13 se ha descrito como el “equivalente en Usenet de una revista que imprime bocabajo la respuesta a un pasatiempo”:

```
$ echo "Hola, Cesar" | python -m encodings.rot_13
```

Se codifica y descodifica igual:

```
$ echo "Ubyn, Prfne" | python -m encodings.rot_13
```

6.3 Validar y reformatear json::

A partir de texto json, lo valida y lo imprime formateado de forma más agradable para el lector humano:

```
$ python -m json.tool <input
```

Se complementa muy bien con la utilidad curl, de forma que podemos obtener json desde una direccion y formatearla con la siguiente línea:

```
$ curl <URL> | python -m json.tool
```

6.4 Servidor de correo de pruebas

La siguiente línea crea un servidor smtp que imprime los correos que recibe en la línea estandar (Y no los envia, claro). Es útil para desarrollo web y para crear baterias de test que comprueben nuestros envios de correo.

python -m smtpd -n -c DebuggingServer localhost:1025

6.5 Servidor web

Las siguientes líneas arrancan un servidor web, que monstrara los contenidos del directorio actual, y que permitira descargar los ficheros que se encuentra en el directorio actual o es sus descendientes.

Para Python 2.x:

```
$ python2 -m SimpleHTTPServer [port#]
```

Para Python 3.x:

```
$ python3 -m http.server [--cgi] [port#]
```

Véase Waitress para un ejemplo de servidor web más orientado a producción en Python.

6.6 Compresión y decompresion de archivos con gzip/zip

Para comprimir o descomprimir ficheros con gzip:

```
$ python -m gzip [file] # comprimir
$ python -m gzip -d [file] # descomprimir
```

Para gestionar archivadores .ZIP:

```
$ python -m zipfile -l <file> # listar contenidos
$ python -m zipfile -t <file> # test
$ python -m zipfile -e <file> <dir> # extraer
$ python -m zipfile -c <file> sources... # crear
```

6.7 Cliente sencillo ftp

Permite recuperar rapidamente un archivo desde un servidor FTP, usando un usuario anónimo (o, si está definido, leyendo usuario y password del fichero `.netrc`):

```
$ python -m ftplib host -p <ruta al fichero>
```

6.8 Extraer el texto de un fichero en HTML

$ python -m htmllib <archivo>

6.9 Listar el contenido de un buzón POP3

$ python -m poplib <server> <username> <password>

6.10 MIME type/extension database

Consultar la base de datos de tipos MIME. Para ver el tipo MIME que le correspondería a un fichero, basándonos en su extensión:

```
$ python -m mimetypes file.ext
```

Para ver las extensiones aosciadas con un determinado tipo MIME:

```
$ python -m mimetypes -e mime/type
```

6.11 Abrir un navegador

Para abrir una página web en un navegador:

```
$ python -m webbrowser -n <url>
```

Para abrir una página web en un navegador, pero en una nueva pestaña si el navegador:

```
$ python -m webbrowser -t <url> # new tab
```

6.12 Antigravedad

$ python -m antigravity

6.13 Navegador Documentacion python

Para abrilo en modo consola (parecido a man):

```
$ python -m pydoc <topic>
```

Para abrilo en modo gráfico:

```
$ python -m pydoc -g
```

Para abrirlo como un servidor web en un puerto determinado:

```
$ python -m pydoc -p <port> # star
```

6.14 Comparar directorios:

Comparar el contenido de dos directorios:

```
python -m filecmp dir1 dir2
```

Si queremos que incluya recursivamente todos los directorios:

```
python -m filecmp -r dir1 dir2
```

6.15 Varios

Imprime un calendario (como cal) pero puede sacar html y tienen unas cuantas opciones de formato:

```
$python -m calendar
```

Para ver las opciones:

```
$ python -m calendar --help
```

Formatear párrafos de un fichero de texto:

```
python -m formatter [file]
```

Mostrar la plataforma actual (como `uname` pero más sencillo):

```
$ python -m platform
```

CAPÍTULO 7

Apéndice 3: Version 2.7 frente a Versión 3.x

La última versión de Python, a la hora de escribir esto es la 3.3.1, estando en desarrollo la 3.4. Los cambios en la rama 3.x, como ya se explicó, producen incompatibilidad hacia atrás; es decir, el código escrito para las versiones anteriores no es directamente compatible con la nueva versión. Esto no es tan grave como parece.

En primer lugar, los cambios no son tan dramáticos. Es verdad que hay muchos más cambios de lo habitual en una nueva versión, y que algunos de esos cambios son realmente importantes. Pero una vez analizados, vemos que a fin de cuentas el lenguaje en si no ha cambiado tanto; la mayor parte de los cambios se dirigen a eliminar pequeños defectos bien conocidos, a la vez que se eliminaba algo del "polvo acumulado" en el código.

7.1 Importar desde el futuro (con python se puede)

Python 2.7.x es la forma natural de migrar a Python 3k, ya que, con ciertos mínimos ajustes, podemos conseguir que ejecute tanto código Python 2.7 como Python 3.0. Para ello se usa una convención muy útil que la comunidad Python ha venido usando desde hace tiempo: Importar desde el futuro.

La costumbre es que cuando se prepara una nueva funcionalidad, antes de publicarla oficialmente en una versión determinada, se incluye como parte del modulo `__future__`. De esta forma, los programadores pueden aprender a usarla, advertir sobre errores, familiarizarse con ella, etc... simplemente haciendo un `import`.

Veremos algunos de las importaciones que tendremos que hacer en Python 2.7 para obtener un entorno similar a Python 3

Nota: Todo o Nada

Las sentencias `from __future__ import xxx` deben ser las primeras que se ejecutan dentro de un *script*; no está permitido ejecutarlas en medio del programa. Es una de esas situaciones todo o nada.

7.2 La función `print`

Este cambio es uno de los más drásticos, a nivel de sintaxis del lenguaje, pero veremos que no es tan grave, porque los cambios que hay que hacer para adaptar nuestro programas son triviales (Aunque, eso si, puede que tengamos que

cambiar muchas líneas de código).

Básicamente, se ha convertido la orden `print`, que técnicamente era una sentencia, en una llamada a una función. De forma que, siguiendo las convenciones de llamada de funciones de Python, es obligatorio usar paréntesis. En otras palabras, donde antes escribíamos:

```
print "Hola, Mundo"
```

Ahora tenemos que escribir:

```
print("Hola, Mundo")
```

Se usan parámetros por nombre para reemplazar la sintaxis especial de la antigua sentencia *print*, veamos algunos ejemplos:

```
Antes: print "La respuesta es:", 2*2
Ahora: print("La respuesta es:", 2*2)

Antes: print x,           # La coma final suprimía el salto de línea
Ahora: print(x, end=" ")  # y lo sustituía por un espacio

Antes: print              # Imprime un salto de línea
Ahora: print()            # ¡Pero ahora es una función!

Antes: print >>sys.stderr, "Error fatal"
Ahora: print("Error fatal", file=sys.stderr)
```

Ahora también se puede definir el separador a usar, con el parámetro `sep`, cuando imprimimos varios elementos (Si no se indica nada, se asumirá que el separador es la cadena vacia, es decir, que no hay separador):

```
>>> print("There are <", 2**32, "> possibilities!", sep="")
There are <4294967296> possibilities!
```

La herramienta de conversión de código `2to3` realiza automáticamente la transformación de sentencias `print` en llamadas a la función `print()`, para que este cambio no sea crítico para proyectos grandes.

Para que nuestro codigo 2.7 utilice la función `print`, pondremos al principio la línea:

```
from __future__ import print_function
```

7.3 Vistas e iteradores en vez de listas

Algunas de las API más usadas ya no devuelven listas:

- Los métodos de los diccionarios `keys()`, `items()` y `values()` devuelven "vistas" en vez de listas. (Una vista es como un iterador especial que sigue vinculada con el objeto a partir del cual se creó, de forma que las modificaciones en el objeto original afectan a la vista). Por ejemplo, el siguiente código fallará:

  ```
  >>> d = {'uno':1, 'dos':2}
  >>> k = d.keys()
  >>> k.sort()
  >>>
  ```

  Mejor usar `sorted` (Funciona desde la versión 2.5 y es más eficiente):

  ```
  >>> d = {'uno':1, 'dos':2}
  >>> k = sorted(d)
  >>>
  ```

Los métodos `iterkeys()`, `iteritems()` e `itervalues()` ya no son soportados.

- `map()` y `filter()` devuelven iteradores. Si se necesita una lista, la solución más rápida es:

```
>>> list(map(...))
>>>
```

  Pero quizá convenga ver si se pueden sustituir todo el código con comprensión de listas, especialmente si en el código original había expresiones lambda.

- la función `range()` se comporta como solia hacerlo `xrange()`. Esta última, por tanto, desaparece.
- La función `zip()` devuelve un iterador.

7.4 La división de enteros

En python 2.7, la división de un entero por un entero produce, a su vez, un entero. Podemos probarlo en Python 2.7:

```
>>> print 7/2
3
```

En python 3k, la división de un entero por un entero produce un número en coma flotante, más parecido a lo que estamos acostumbrados. Podemos probarlo en Python 3:

```
>>> print(7/2)
3.5
```

Podemos usar una doble barra de división para obtener un resultado entero, truncado. Esta capacidad existe desde hace años, al menos desde la versión 2.2:

```
>>> print(7//2)
3
```

Para que nuestro codigo 2.7 utilice la nueva división, pondremos al principio la línea:

```
from __future__ import division
```

7.5 Unicode por defecto

Este es uno de los cambios más importantes. Prácticamente todo lo que se refiere a variables de texto cambia. Python 3.0 trabaja con dos conceptos: Cadenas de texto unicode y datos binarios (mientras que python 2.7 trabaja con cadenas de texto unicode y cadenas de texto de 8 bits).

En 3.x todo el texto es Unicode; cualquier texto codificado se considera ahora datos binarios. El Tipo de datos usado para texto es `str`, y el usado para datos es `byte`. La mayor diferencia con la versión 2.x es que cualquier intento de mezclar texto y datos provoca un error de tipo `TypeError`. En la versión anterior, la mezcla de los dos tipos podía funcionar o no: si teníamos la suerte de que las cadenas de texto de 8 bits solo contuvieran texto ASCII de 7 bits (Tambien llamado ASCII puro, básico o estándar) entonces funcionaba. Si no, nos lanzaba un `UnicodeDecodeError`. Este tipo de error ha repartido mucha tristeza en estos años.

A consecuencia de estos cambios, casi todo el código que trabaje con unicode, texto codificado y datos binarios tendrá que cambiar. El cambio es para mejor, ya que el el mundo 2.x muchisimos errores tenían que ver con estas mezclas de texto codificado con texto sin codificar. Para prepararse para el paso a Python 3k, lo mejor es empezar a usar texto unicode para cualquier texto que no tenga que estar codificado, y reservar el tipo `str```solo para textos codificados y datos binarios. Haciendolo así la herramienta automática ``2to3` podrá realizar la mayor parte del trabajo por nosotros.

Ya no se podrá usar la forma u"..." para indicar literales en unicode, porque ya no hará falta. Al contrario, tendremos que marcar con b"..." para inidicar texto codificado o datos binarios.

Como los tipos `str` y `bytes` ya no se pueden mezclar, habrá que realizar las conversiones explicitamente, ya sea usando los métodos `encode()` ``y `` `decode()` de los tipos `str` y `bytes` respectivamente, o usando las funciones `bytes(s, encoding=...)` o `str(b, encoding=...)`.

Tanto los tipos `str` como `bytes` son inmutables. Hay un tipo diferente, llamado `bytearray` que si es mutable, y que podemos usar para para mantener buffers de datos binarios.

Los archivos abiertos en modo texto (Que sigue siendo el modo por defecto de `open()`) siempre utilizarán algún tipo de codificación para mapear entre los datos (bytes) guardados en disco y las cadenas de texto (`string`) en memoria. Esto implica que si se abre un archivo con un modo incorrecto o una codificacion erronea, se producirá un error y se elevará la correspondiente excepción, que siempre es mejor que empezar a emitir datos incorrectos como si no pasara nada (Los errores nunca deberían dejarse pasar silenciosamente). También significa que los usuarios de unix tendrán que empezar a abrir los archivos con los que trabajen con el modo correcto: texto o binario. Hay una codificación por defecto, que enlas plataformas unix se determinara por la variable de entorno `LANG`. En muchos casos, pero no siempre, la codificación por defecto será `utf-8`.

Como efecto adicional, los nombres de las variables tambien se codificarán en python 3.x con unicode; esto significa que podemos tener variables con nombres como `árbol`, `temporada_otoño` e incluso . Yo personalmente seguire usando ASCII puro para mis variables, ya que no creo que esto aporte muchas más legibilidad y si creo que puede aumentar los errores.

Para que nuestro codigo 2.7 utilice literales unicode, pondremos al principio la línea:

```
from __future__ import unicode_literals
```

7.6 Importaciones relativas / absolutas

La importación de módulos funciona buscando, en una serie de directorios habilitados para ello, un fichero que corresponda con el nombre del modulo a importar. Para estructurar mejor el código, los paquetes permiten definir una estructura de módulos, organizados en forma de árbol. Esto funciona muy bien, pero presentaba dos problemas:

 # En los paquetes con muchos niveles, las sententcias `import` podian acabar siendo bastante largas

 # Las importaciones podían ser ambigüas en combinación con los paquetes; dentro de un paquete, no quedaba claro si, al hacer `import foo`, el programador se refería a un módulo dentro del paquete o a un módulo fuera de este.

La solución para este segundo problema paso por hacer que todos las importaciones sean absolutas por defecto (Es decir, que se buscará solo en los directorios indicados por `sys.path`), y se usará una sintaxis especial (anteponiendo puntos) para acceder a los módulos internos del paquete. Un único punto precediendo al nombre del módulo significa una importación relativa, en el mismo nivel que el actual. Dos o más puntos indican una importación relativa con respecto a los padres del paquete actual, un nivel por cada punto después del primero.

Veamos un ejemplo, si tenemos la siguiente estructura dentro de un paquete:

```
paquete/
__init__.py
subpaquete1/
    __init__.py
    moduloX.py
    moduloY.py
subpaquete2/
    __init__.py
    moduloZ.py
moduloA.py
```

Si suponemos que estamos trabajando con `moduloX` dentro del `subpaquete1`, las siguientes importaciones relativas serían todas válidas:

```
from . import moduloY
from .moduloY import spam
from ..subpaquete1 import moduloY
from ..subpackage2 import moduloZ
from ..subpaquete2.moduloZ import eggs
from .. import moduloA
```

las importaciones relativas siempre tienen que ser de la forma `from <> import <>`; la forma `import <>` siempre será absoluta.

Para que python 2.7 funcione con este nuevo sistema de importaciones, tenemos que incluir al principio de nuestro programa la línea:

```
from __future__ import absolute_import
```

glosario de términos

Calendario Gregoriano El calendario gregoriano es un calendario originario de Europa, actualmente utilizado de manera oficial en casi todo el mundo. Se denomina así por haber sido su promotor el Papa Gregorio XIII. Vino a sustituir en 1582 al Calendario Juliano, utilizado desde que Julio César lo instaurara en el año 46 A. C.

http://en.wikipedia.org/wiki/Gregorian_calendar

comprensión de listas, list comprehension La comprensión de listas en una estructura sintáctica disponible en algunos lenguajes de programación que permite definir una lista en base a otras listas preexistentes. Sigue la misma formulación que el concepto matemático de definición intensiva de un conjunto.

http://en.wikipedia.org/wiki/List_comprehension

CSV, Comma Separted Values, Valores Separados por Comas Los ficheros CSV (del inglés comma-separated values) son un tipo de documento en formato abierto sencillo para representar datos en forma de tabla, en las que las columnas se separan por comas (o punto y coma en donde la coma es el separador decimal: España, Francia, Italia...) y las filas por saltos de línea. Los campos que contengan una coma, un salto de línea o una comilla doble deben ser encerrados entre comillas dobles.

http://en.wikipedia.org/wiki/Comma-separated_values

DOM, Document Object Model, Modelo de Objetos del Documento Una interfaz de programación de aplicaciones (API) que proporciona un conjunto estándar de objetos para representar documentos HTML y XML, un modelo estándar sobre cómo pueden combinarse dichos objetos, y una interfaz estándar para acceder a ellos y manipularlos. A través del DOM, los programas pueden acceder y modificar el contenido, estructura y estilo de los documentos HTML y XML, que es para lo que se diseñó principalmente.

http://en.wikipedia.org/wiki/Document_Object_Model

gestor de contexto Un gestor de contexto o *context manager* es un objeto que define el contexto de ejecución que debe ser establecido al ejecutar una sentencia `with`. Es el encargado de gestiona la entrada y la salida dentro del contexto definido para la ejecución del bloque de código.

intérprete Programa informático capaz de analizar y ejecutar otros programas, escritos en un lenguaje de alto nivel. Los intérpretes se diferencian de los compiladores en que mientras estos traducen un programa desde su descripción en un lenguaje de programación al código de máquina del sistema, los intérpretes sólo realizan la traducción a medida que sea necesaria, típicamente, instrucción por instrucción, y normalmente no guardan el resultado de dicha traducción.

lazy evaluation, evaluación perezosa En la teoría de lenguajes de programación, La evaluación perezosa (del inglés *lazy evaluation*) o llamada por necesidad es una estrategia de evaluación que retrasa el cálculo de una expresión hasta que su valor sea necesario, y que también evita repetir la evaluación en caso de ser necesaria en posteriores

ocasiones. Esta compartición del cálculo puede reducir el tiempo de ejecución de ciertas funciones de forma exponencial, comparado con otros tipos de evaluación.

http://en.wikipedia.org/wiki/Lazy_evaluation

log Log es un anglicismo, equivalente a la palabra bitácora en español. En la jerga informática, un log es un registro de eventos que suceden durante un rango de tiempo en particular. Para los profesionales en seguridad informática es usado para registrar datos o información sobre quién, qué, cuándo, dónde y por qué (who, what, when, where y why) un evento ocurre para un dispositivo en particular o aplicación.

memoize Técnica de optimización, utilizada principalmente para acelerar la velocidad del programa. Para ello evita la ejecución repetida de las llamadas que tengan los mismos parámetros de entrada, mediante el sistema de almacenar los resultados de los calculos realizados previamente.

SAX, Simple API for XML Originalmente, una API únicamente para el lenguaje de programación Java, que después se convirtió en la API estándar de facto para usar XML en JAVA. Existen versiones de SAX no sólo para JAVA, si no también para otros lenguajes de programación, como Python.

http://en.wikipedia.org/wiki/Simple_API_for_XML

SMTP, Simple Mail Transfer Protocol Es un protocolo de red de la capa de aplicación, basado en texto, utilizado para el intercambio de mensajes de correo electrónico entre ordenadores. Es un estándar oficial de Internet definido en el RFC 2821.

http://tools.ietf.org/html/rfc2821 http://en.wikipedia.org/wiki/Simple_Mail_Transfer_Protocol

Indices and tables

- *genindex*
- *modindex*
- *search*