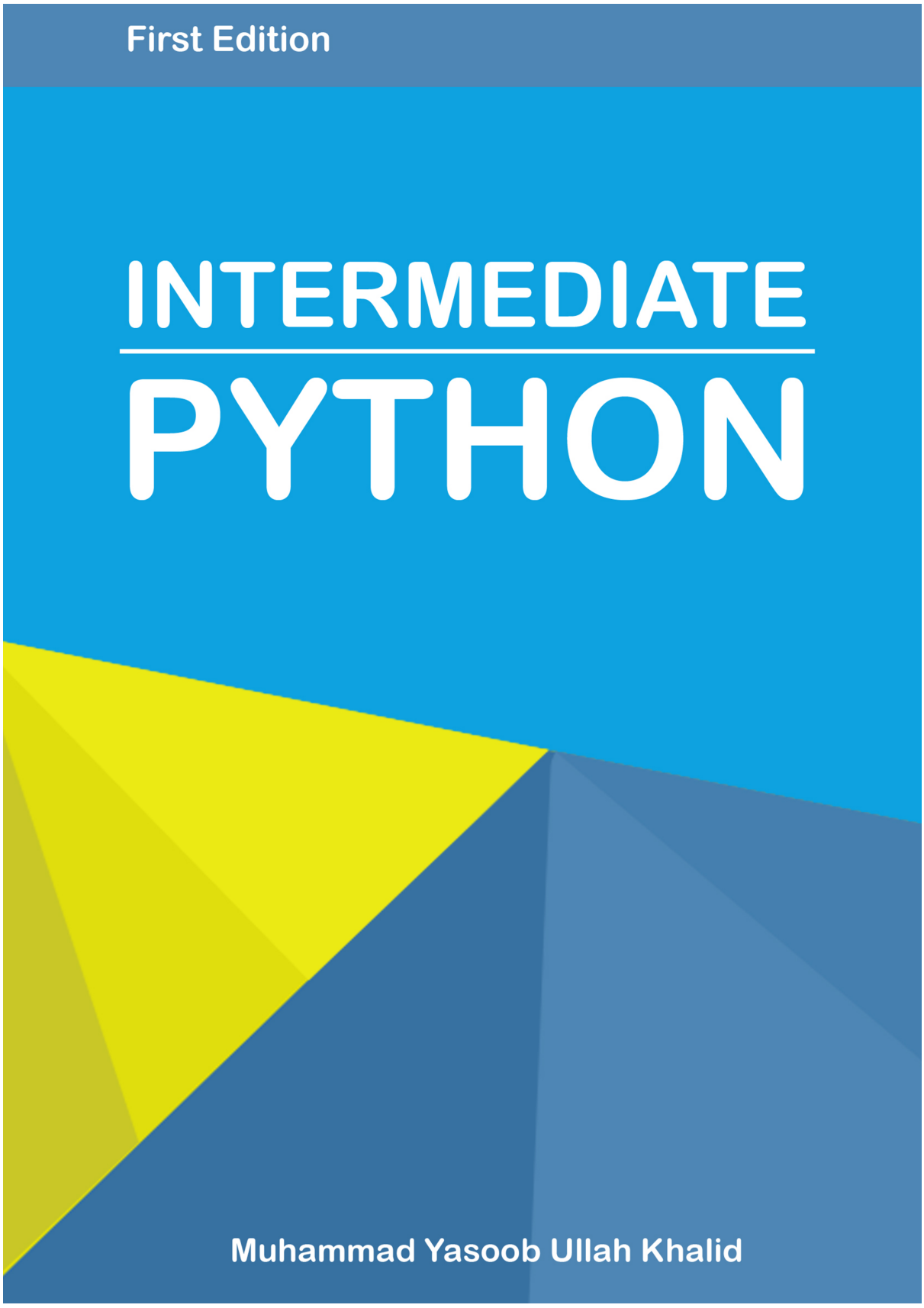


# Python Intermedio

*Versión 0.1*

Traducción: cursospython.com

**04 de junio de 2020**

# Contents


**1 Introducción** **2**

**2 Autor** **3**

**3 Tabla de Contenido** **4**

3.1 Uso de *args y **kwargs . . . 4
3.2 Depurando . . . 6
3.3 Generadores . . . 7
3.4 Map, Filter y Reduce . . . 11
3.5 Estructura de datos `set` . . . 13
3.6 Operadores ternarios . . . 14
3.7 Decoradores . . . 16
3.8 Global & Return . . . 25
3.9 Mutabilidad . . . 28
3.10 Método mágico __slots__ . . . 30
3.11 Entornos virtuales . . . 32
3.12 Colecciones . . . 33
3.13 Enumerados . . . 40
3.14 Introspección de objetos . . . 40
3.15 Comprensión . . . 42
3.16 Excepciones . . . 44
3.17 Clases . . . 46
3.18 Funciones Lambda . . . 50
3.19 Ejemplos en 1 línea . . . 51
3.20 `for/else` . . . 54
3.21 Extensiones C de Python . . . 55
3.22 Función `open` . . . 62
3.23 Usando Python 2+3 . . . 64
3.24 Corrutinas . . . 66
3.25 Caching de Funciones . . . 68
3.26 Gestores de Contexto . . . 69

**Nota:** Este documento contiene una serie de tutoriales y ejemplos en Python para un nivel intermedio. Ha sido traducido por la comunidad cursospython.com y todo el contenido ha sido donado bajo licencia de *Creative Commons*, por lo que siéntete libre para compartirlo, modificarlo o colaborar. Libro original: intermediatePython

CHAPTER 1

# Introducción

Python es un lenguaje de programación increíble con una comunidad de programadores de las más fuertes, y es cada vez más usado en gran cantidad de ámbitos e industrias. Sin embargo, muchas veces es difícil encontrar documentación, y sobre todo en Español. En cursospython.com estamos comprometidos con la enseñanza en Python, y colaboramos tanto con contenido propio como con traducciones. Es una pena que siendo el Español un idioma tan hablado, sea tan complicado encontrar documentación de calidad. No todo el mundo entiende Inglés perfectamente, y aunque ese sea el caso, ¿no resulta más fácil de leer el contenido escrito en tu lengua materna? El idioma nunca puede ser una barrera en la educación.

Este libro recopila diferentes conceptos de Python de un nivel intermedio, por lo que para poder leerlo fácilmente es necesario tener unas bases del lenguaje de programación. El libro no pretende ser una referencia, sino un complemento a la documentación oficial, que muchas veces resulta difícil de leer y poco didáctica, sobre todo para gente que está empezando.

Aún así, estoy seguro de que sea cual sea tu nivel, habrá algo para ti en este libro. Tanto si eres un principiante como si tu nivel es intermedio, verás nuevos conceptos que pronto podrás empezar a utilizar en tu día a día. Por otro lado si eres un experto, estoy seguro de que tal vez encuentres alguna forma de colaborar, por lo que estaríamos encantados de escucharte.

Este libro está en continua evolución, por lo que asegúrate de que tienes la última, y si tienes cualquier sugerencia estamos abiertos a ellas a través del repositorio de GitHub.

CHAPTER **2**

# Autor

Sobre el autor de la traducción: Este libro es una traducción en Español escrita por cursospython.com titulada *Python Intermedio*. Somos una comunidad de Python en español, comprometida a crear documentación y tutoriales sobre el lenguaje, accesibles de manera gratuita en nuestra web. Colaboramos con la comunidad *open source* y esta traducción ha sido donada al igual que el original bajo una licencia *Creative Commons* CC BY-NC-SA 4.0, lo que principalmente dice que puedes hacer lo que quieras con este libro siempre y cuando menciones la fuente y sea con fines no comerciales.

Sobre el autor original: Su versión original es de Muhammad Yasoob Ullah Khalid, un fan de Python con varios años de experiencia en el lenguaje, y conocido en la comunidad. Participó en 2014 en EuroPython en Berlin, una de las mayores conferencias de Python en Europa.

CHAPTER 3

# Tabla de Contenido

## 3.1 Uso de *args y **kwargs

La mayoría de los programadores nuevos en Python tienen dificultades para entender el uso de *args y **kwargs. ¿Para qué se usan? Lo primero de todo es que en realidad no tienes porque usar los nombres args o kwargs, ya que se trata de una mera convención entre programadores. Sin embargo lo que si que tienes que usar es el asterisco simple * o doble **. Es decir, podrías escribir *variable y **variables. Empecemos viendo el uso de *args.

### 3.1.1 Uso de *args

El principal uso de *args y **kwargs es en la definición de funciones. Ambos permiten pasar un número variable de argumentos a una función, por lo que si quieres definir una función cuyo número de parámetros de entrada puede ser variable, considera el uso de *args o **kwargs como una opción. De hecho, el nombre de args viene de argumentos, que es como se denominan en programación a los parámetros de entrada de una función.

A continuación te mostramos un ejemplo para que veas el uso de *args:

```
def test_var_args(f_arg, *argv):
    print("primer argumento normal:", f_arg)
    for arg in argv:
        print("argumentos de *argv:", arg)

test_var_args('python', 'foo', 'bar')
```

Y la salida que produce el código anterior al llamarlo con 3 parámetros es la siguiente:

```
primer argumento normal: python
argumentos de *argv: foo
argumentos de *argv: bar
```

Espero que esto haya aclarado el uso de *args, continuemos con **kwargs.

### 3.1.2 Uso de **kwargs

**kwargs permite pasar argumentos de longitud variable asociados con un nombre o **key** a una función. Deberías usar **kwargs si quieres manejar argumentos con nombre como entrada a una función. Aquí tienes un ejemplo de su uso.

```
def saludame(**kwargs):
    for key, value in kwargs.items():
        print("{0} = {1}".format(key, value))

>>> saludame(nombre="Covadonga")
nombre = Covadonga
```

Es decir, dentro de la función no solo tenemos acceso a la variable como con *args, sino que también tenemos acceso a un nombre o key asociado. A continuación veremos como se puede usar *args y **kwargs para llamar a una función con una lista o diccionario como argumentos.

### 3.1.3 Usando *args y **kwargs para llamar a una función

Ahora veremos como se puede llamar a una función usando *args y **kwargs. Consideremos la siguiente.

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

Ahora puedes usar *args o **kwargs para pasarle argumentos a la función. Se puede hacer de la siguiente manera:

```
# Primero con *args
>>> args = ("dos", 3, 5)
>>> test_args_kwargs(*args)
arg1: dos
arg2: 3
arg3: 5

# Ahora con **kwargs:
>>> kwargs = {"arg3": 3, "arg2": "dos", "arg1": 5}
```

(continues on next page)

(continued from previous page)

```
>>> test_args_kwargs(**kwargs)
arg1: 5
arg2: dos
arg3: 3
```

Por último, si quieres usar los tres tipos de argumentos de entrada a una función: normales, *args y **kwargs, deberás hacerlo en el siguiente orden.

```
funcion(fargs, *args, **kwargs)
```

### 3.1.4 ¿Cuándo usarlos?

Dependerá mucho de los requisitos de tu programa, pero uno de los usos más comunes es para crear decoradores para funciones (que veremos en otro capítulo). También puede ser usado para *monkey patching*, lo que significa modificar código en tiempo de ejecución. Considera por ejemplo que tienes una clase con una función llamada `get_info` que llama a una API que devuelve una determinada respuesta. Si quieres testearla, se puede reemplazar la llamada a la API por unos datos de test, como por ejemplo:

```
import someclass

def get_info(self, *args):
    return "Test data"

someclass.get_info = get_info
```

Estoy seguro de que se te ocurren otros usos.

## 3.2 Depurando

Depurar es una de las herramientas que mas nos pueden ayudar si tenemos un *bug* o fallo que necesitamos resolver. Mucha gente olvida la importancia del depurador de Python `pdb`. En esta sección veremos algunos de los comandos más importantes, por lo que si quieres entrar en detalle, no olvides entrar en la documentación oficial.

**Desde línea de comandos**

Puedes ejecutar un *script* desde la línea de comandos usando el depurador de Python. Se hace de la siguiente manera:

```
$ python -m pdb my_script.py
```

Esto hará que el depurador pare la ejecución del programa en la primera sentencia que encuentre. Su uso es muy útil cuando el *script* es corto. Puedes inspeccionar las variables y continuar con la ejecución línea por línea.

**Desde dentro del script**

También puedes asignar diferentes *break points* o puntos de ruptura para poder inspeccionar el contenido de las variables en determinados puntos del código. Esto se puede hacer con el método `pdb.set_trace()`. Vemos un ejemplo:

```
import pdb

def haz_algo():
    pdb.set_trace()
    return "No quiero"

print(haz_algo())
```

Intenta ejecutar el código anterior una vez guardado. Entrarás en el depurador en cuanto empieces a ejecutarlo. Visto esto, vamos a ver algunos de los comandos más útiles del depurador.

**Comandos:**

- `c`: continúa la ejecución
- `w`: muestra el contexto de la línea que se esta ejecutando.
- `a`: imprime la lista de argumentos para la función actual.
- `s`: ejecuta la primera línea y para en cuanto sea posible.
- `n`: continúa la ejecución hasta la siguiente línea en la función actual o hasta que se retorna.

La diferencia entre `n` y `s` se ve muy fácil en Inglés, ya que viene de **n**ext y **s**top. El uso de next ejecuta la función llamada prácticamente a velocidad normal, tan solo parando en la siguiente línea. Por lo contrario, stop para dentro de la función llamada.

Estos son sólo unos pocos comandos. `pdb` también soporta análisis *post mortem*, una de las características que te recomendamos que investigues un poco más a fondo.

**Nota:**

Puede no ser muy intuitivo usar `pdb.set_trace()` si eres nuevo en esto. Afortunadamente si usas Python 3.7+ puedes usar simplemente `breakpoint()` https://docs.python.org/3/library/functions.html#breakpoint. Automáticamente importa `pdb` y llama a `pdb.set_trace()`.

## 3.3 Generadores

Antes de nada, veamos un repaso de los iteradores. De acuerdo con Wikipedia, un iterador es un objeto que permite a un programador recorrer un contenedor, como podría ser una lista. Sin embargo, los iteradores recorren el contenedor y proveen acceso a los elementos del mismo, pero no realizan la iteración propiamente dicha. Hay tres conceptos que es necesario entender:

- Iterable
- Iterador
- Iteración

Todos ellos están relacionados entre sí. A continuación los explicaremos unos por uno.

### 3.3.1 Iterable

Un `iterable` es cualquier objeto en Python que implementa el método `__iter__` o `__getitem__`, es decir, que devuelve un iterador que puede ser indexado. Puedes leer más acerca de esto aquí. En otras palabras, un `iterable` es un objeto que nos puede proporcionar un **iterador**. ¿Pero qué es un **iterador**?

### 3.3.2 Iterador

Un iterador es cualquier objeto en Python que tenga definidos los métodos `next` (Python 2) o `__next__`. Sabido esto, vamos a ver ahora que es una **iteración**.

### 3.3.3 Iteración

La iteración es el proceso que seguimos cuando vamos tomando diferentes elementos de una lista, es decir, la vamos iterando. Cuando usamos un bucle para iterar sobre un elemento determinado, esto es una iteración. Es el nombre que se le da al proceso.

Una vez sabido esto, vamos a ver su relación con los **generadores**.

### 3.3.4 Generadores

Los generadores son en realidad iteradores, pero sólo permiten ser iterados una vez. Esto se debe a que no almacenan todos los valores en memoria, sino que los van generando al vuelo. Pueden ser usados de dos formas diferentes, iterándolos con un bucle `for` o pasándolos a una función como veremos a continuación.

La mayoría de las veces, los `generadores` son implementados como funciones. Sin embargo no devuelven los valores con `return` sino que lo hacen usando `yield`. Veamos un ejemplo sencillo de una función generadora.

```
def funcion_generadora():
    for i in range(10):
        yield i

for item in funcion_generadora():
    print(item)

# Salida: 0
```

(continues on next page)

(continued from previous page)

```
# 1
# 2
# 3
# 4
# 5
# 6
# 7
# 8
# 9
```

No se trata de un ejemplo demasiado práctico, pero sirve para ilustrar su funcionamiento. Los generadores son más útiles cuando es necesario realizar cálculos para un número muy elevado de elementos. Es decir, son útiles cuando no quieres tener en memoria todos los elementos a la vez, ya que sería demasiado. De hecho, muchas de las funciones de la librería estándar que devolvían listas en Python 2 han sido modificadas para devolver **generadores** en Python 3, porque se requiere de menos recursos.

Otro ejemplo con generadores para calcular la serie de fibonacci:

```
# Usando generadores
def fibon(n):
    a = b = 1
    for i in range(n):
        yield a
        a, b = b, a + b
```

Ahora podemos realizar lo siguiente:

```
for x in fibon(1000000):
    print(x)
```

De esta manera no nos tenemos que preocupar si usaremos demasiados recursos. Sin embargo, implementado de la siguiente forma, podríamos llegar a tener problemas:

```
def fibon(n):
    a = b = 1
    resultado = []
    for i in range(n):
        resultado.append(a)
        a, b = b, a + b
    return resultado
```

Si con el ejemplo anterior usáramos como entrada un número muy elevado, podríamos llegar a tener problemas.

Hasta ahora hemos explicado el uso de los `generators` pero no hemos llegado a probarlos. Antes de probarlos, es necesario saber un poco más acerca de la función `next()` de Python. Esta función nos permite acceder al siguiente elemento de una secuencia:

```
def funcion_generadora():
    for i in range(3):
        yield i

gen = funcion_generadora()
print(next(gen))
# Salida: 0
print(next(gen))
# Salida: 1
print(next(gen))
# Salida: 2
print(next(gen))
# Salida: Traceback (most recent call last):
#           File "<stdin>", line 1, in <module>
#         StopIteration
```

Como podemos ver, cuando se llega al final de la función, si se intenta llamar otra vez al `next()` tendremos un error `StopIteration`, ya que no hay más valores. Esto se debe a que la función no tiene más valores de los que hacer `yield`, es decir se ha llegado al final.

Tal vez te preguntes porque no pasa esto cuando usamos un bucle `for`. La respuesta es muy sencilla, el bucle `for` se encarga automáticamente de capturar este error y de no llamar más a `next`. ¿Sabías que algunas funciones que vienen por defecto también soportan ser iteradas? Vamos a verlo:

```
cadena = "Pelayo"
next(my_string)
# Salida: cadena (most recent call last):
#      File "<stdin>", line 1, in <module>
#    TypeError: str object is not an iterator
```

Tal vez no era eso lo que nos esperábamos. El error dice que `str` (la cadena) no es un elemento iterador. Bueno, eso es cierto, ya que se trata de un elemento iterable pero no es un iterador. Esto significa que soporta ser iterado pero que no puede ser iterado directamente. Entonces, ¿cómo lo iteramos? Veamos como usar la función `iter`, que devuelve un objeto iterador (`iterator`) de una clase iterable.

Entonces, el tipo numérico entero `int` no es iterable, pero una cadena si que lo es. Veamos el ejemplo para el `int`:

```
int_var = 1779
iter(int_var)
# Salida: Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# TypeError: 'int' object is not iterable
# Sucede ya que no es iterable

my_string = "Pelayo"
my_iter = iter(my_string)
```

(continues on next page)

(continued from previous page)

```
print(next(my_iter))
# Salida: 'P'
```

Podemos ver entonces como si llamamos a `iter` sobre un tipo entero, tendremos un error, ya que los enteros no son iterables. Sin embargo, si realizamos lo mismo con una cadena, nos devolverá un iterador sobre el que podemos usar `next()` para ir accediendo secuencialmente a sus valores hasta llegar al final.

Una vez explicado esto, esperamos que hayas entendido los `generators` y los conceptos asociados como el iterador o que una clase sea iterable. Los generadores son sin duda una herramienta muy potente, por lo que te recomendamos que tengas los ojos abiertos porque seguramente encontrarás alguna aplicación donde te ayuden a resolver un problema.

## 3.4 Map, Filter y Reduce

Estas tres funciones proporcionan un enfoque funcional a la programación. Si no sabes que es la programación funcional, te recomendamos que leas acerca de ello, ya que es un mundo muy interesante. A continuación explicamos `map`, `reduce` y `filter` con varios ejemplos.

### 3.4.1 Map

El uso de `map` aplica una determinada función a todos los elementos de una entrada o lista. Esta es su forma:

**Forma**

```
map(funcion_a_aplicar, lista_de_entradas)
```

Se trata de un caso de uso bastante recurrente. Imaginemos por ejemplo que tenemos una lista y queremos crear otra lista con todos sus elementos elevados al cuadrado. La primera forma que tal vez se nos ocurra, sería la siguiente:

```
lista = [1, 2, 3, 4, 5]
al_cuadrado = []
for i in lista:
    al_cuadrado.append(i**2)
```

Sin embargo, existe una forma más fácil de hacerlo con `map`. Es mucho más sencilla y corta:

```
lista = [1, 2, 3, 4, 5]
al_cuadrado = list(map(lambda x: x**2, lista))
```

La mayoría de las veces `map` es usado conjuntamente con funciones lambda. Si no sabes lo que son, las explicamos en otro capítulo.

Otra forma de usar `map` es teniendo una lista de funciones en vez de una en concreto. Veamos un ejemplo:

```
def multiplicar(x):
    return (x*x)
def sumar(x):
    return (x+x)

funcs = [multiplicar, sumar]
for i in range(5):
    valor = list(map(lambda x: x(i), funcs))
    print(valor)

# Salida:
# [0, 0]
# [1, 2]
# [4, 4]
# [9, 6]
# [16, 8]
```

Se puede ver como ahora para cada elemento (del 0 al 4) tenemos dos salida, la primera aplica la función `multiplicar` y la segunda `sumar`.

### 3.4.2 Filter

Como su nombre indica, `filter` crea una lista de elementos si usados en la llamada a una función devuelven `True`. Es decir, filtra los elementos de una lista usando un determinado criterio. Veamos un ejemplo:

```
lista = range(-5, 5)
menor_cero = list(filter(lambda x: x < 0, lista))
print(menor_cero)

# Salida: [-5, -4, -3, -2, -1]
```

La función `filter` es similar a un bucle, y de hecho podríamos conseguir lo mismo con un bucle y un `if`, pero su uso es más rápido.

**Nota:** Si no te gusta el uso de map y filter, echa un vistazo a las *list comprehensions* de las que hablamos en otro capítulo.

### 3.4.3 Reduce

Por último, `reduce` es muy útil cuando queremos realizar ciertas operaciones sobre una lista y devolver su resultado. Por ejemplo, si queremos calcular el producto de todos los elementos de una lista, y devolver un único valor, podríamos hacerlo de la siguiente forma sin usar `reduce`.

```
producto = 1
lista = [1, 2, 3, 4]
for num in lista:
    producto = producto * num

# producto = 24
```

Ahora vamos a hacerlo con `reduce`.

```
from functools import reduce
producto = reduce((lambda x, y: x * y), [1, 2, 3, 4])

# Salida: 24
```

## 3.5 Estructura de datos `set`

El `set` es una estructura de datos muy usada. Los `sets` se comportan como las listas, con la diferencia de que no pueden contener elementos duplicados. También son inmutables, y una vez son definidos sus elementos no pueden ser modificados. Tampoco son ordenadores, por lo que no respetan el orden en el que son definidos. Son útiles si por ejemplo quieres ver si en una lista hay duplicados o no. Tienes dos opciones de hacerlo, donde la primera usa un bucle `for`:

```
lista = ['a', 'b', 'c', 'b', 'd', 'm', 'n', 'n']

duplicados = []
for value in lista:
    if lista.count(value) > 1:
        if value not in duplicados:
            duplicados.append(value)

print(duplicados)
# Salida: ['b', 'n']
```

Pero hay una forma más simple y elegante de realizar la misma tarea usando los `sets`. Lo vemos a continuación:

```
lista = ['a', 'b', 'c', 'b', 'd', 'm', 'n', 'n']
duplicados = set([x for x in lista if lista.count(x) > 1])
print(duplicados)
# Salida: set(['b', 'n'])
```

Los sets también tienen otros métodos, vemos algunos a continuación.

**Intersección**

Podemos calcular la intersección entre dos sets de la siguiente manera.

```
set1 = set(['amarillo', 'rojo', 'azul', 'verde', 'negro'])
set2 = set(['rojo', 'marrón'])
print(set2.intersection(set1))
# Salida: set(['rojo'])
```

**Diferencia**

Con el método `difference` podemos calcular la diferencia entre dos sets. Es importante notar que no es lo mismo la diferencia `A-B` que `B-A`. En el siguiente caso se ve como la diferencia del set2 y el set1 son los elementos del set2 que no están presentes en el set1.

```
set1 = set(['amarillo', 'rojo', 'azul', 'verde', 'negro'])
set2 = set(['rojo', 'marrón'])
print(set2.difference(set1))
# Salida: set(['marrón'])
```

También puedes crear sets usando `{}` como se muestra a continuación.

```
set1 = {'red', 'blue', 'green'}
print(type(set1))
# Salida: <type 'set'>
```

Existen otros métodos del `set` muy bien explicados en este post.

## 3.6 Operadores ternarios

Los operadores ternarios son más conocidos en Python como expresiones condicionales. Estos operadores evalúan si una expresión es verdadera o no. Se añadieron a Python en la versión 2.4.

**Forma:**

```
condition_if_true if condition else condition_if_false
```

**Ejemplo:**

```
es_bonito = True
estado = "Es bonito" if es_bonito else "No es bonito"
```

Si te quedas con dudas te recomendamos este post donde se explican con más ejemplos.

Como se puede ver, permiten verificar de manera rápida una condición, y lo mejor de todo es que se puede hacer en una sola línea de código. Por lo general hacen que el código sea más compacto y fácil de leer.

Otra forma un tanto extraña y no demasiado usada es la siguiente:

**Forma:**

```
(if_test_is_false, if_test_is_true)[test]
```

**Example:**

```
es_bonito = True
apariencia = ("Feo", "Bonito")[es_bonito]
print("El gato es ", apariencia)
# Salida: El gato es bonito
```

Este ejemplo funciona ya que `True=1` y `False=0`, y puede ser usado también con listas. Es importante decir también que este ejemplo no es muy usado, y por lo general no gusta a los *Pythonistas*.

Otro de los motivos por los que no resulta del todo correcto su uso, es que ambos elementos son evaluados, mientras que en operador ternario if-else no.

**Ejemplo:**

```
condicion = True
print(2 if condition else 1/0)
#Salida is 2

print((1/0, 2)[condicion])
#Se lanza ZeroDivisionError
```

En este ejemplo la `condicion` es verdadera, por lo que tomaremos el segundo elemento (índice 1) de la lista. Sin embargo como podemos ver, el `1/0` es también evaluado, ya que se lanza una excepción. Esto sucede ya que primero la tupla es creada, y después se toma el elemento con el índice `[]`. Sin embargo el if-else ternario es igual que un if-else normal, por lo que sólo se evalúa una rama.

**Abreviación ternaria**

En Python existe también una forma acortada del operador ternario normal que hemos visto antes. Esta sintaxis fue introducida en Python 2.5, por lo que puede ser usada de ahí en adelante.

**Ejemplo**

```
>>> True or "Valor"
True
>>>
>>> False or "Valor"
'Some'
```

En el primer ejemplo *True or «Some»* devuelve `True`, mientras que en el segundo se devuelve `"Valor"`. Es una herramienta bastante útil cuando quieres verificar rápidamente el contenido de una variable, y mostrar un mensaje alternativo si está vacío.

```
>>> salida = None
>>> msg = salida or "No se devolvió nada"
```

(continues on next page)

(continued from previous page)

```
>>> print(msg)
No se devolvió nada
```

O también es una forma muy simple de definir parámetros con valores por defecto dinámicos. En el siguiente ejemplo vemos como se imprime el `nombre_real` por defecto, pero si se proporciona también un `nombre_opcional` se imprimirá este por pantalla en vez del anterior.

```
>>> def mi_funcion(nombre_real, nombre_opcional=None):
>>>     nombre_opcional = optional_display_name or nombre_real
>>>     print(nombre_opcional)
>>> mi_funcion("Pelayo")
Pelayo
>>> mi_funcion("Covadonga", "Cova")
Cova
```

## 3.7 Decoradores

Los decoradores son una funcionalidad relativamente importante en Python. Se podría decir que son funciones que modifican la funcionalidad de otras funciones, y ayudan a hacer nuestro código más corto y Pytónico o Pythonic. A continuación veremos lo que son, cómo se crean y cómo podemos usarlos.

### 3.7.1 Todo es un objeto en Python:

Antes de entrar en materia con los decoradores, vamos a entender bien las funciones.

```
def hola(nombre="Covadonga"):
    return "Hola " + nombre

print(hola())
# Salida: 'Hola Covadonga'

# Podemos asignar una función a una variable
saluda = hola
# No usamos () porque no la estamos llamando, sino que la estamos
# asignado a una variable

print(saluda())
# Salida: 'Hola Covadonga'

# También podemos eliminar la función asignada a la variable con del
del hola
print(hola())
#Salida: NameError
```

(continues on next page)

(continued from previous page)

```
print(saluda())
#Salida: 'Hola Covadonga'
```

### 3.7.2 Definir funciones dentro de funciones:

Vamos a ir un paso más allá. En Python podemos definir funciones dentro de otras funciones. Veamos un ejemplo:

```
def hola(nombre="Covadonga"):
    print("Estás dentro de la función hola()")

    def saluda():
        return "Estás dentro de la función saluda()"

    def bienvenida():
        return "Estás dentro de la función bienvenida()"

    print(saluda())
    print(bienvenida())
    print("De vuelta a la función hola()")

hi()
#Salida:Estas dentro de la función hola()
#       Estás dentro de la función saluda()
#       Estás dentro de la función bienvenida()
#       De vuelta a la función hola()

# Esto muestra como cada vez que llamas a la función hola()
# se llama en realidad también a saluda() y bienvenida()
# Sin embargo estas dos últimas funciones no están accesibles
# fuera de hola(). Si lo intentamos, tendremos un error.

saluda()
#Saluda: NameError: name 'saluda' is not defined
```

Ya hemos visto entonces como podemos definir funciones dentro de otras funciones. En otras palabras, podemos crear funciones anidadas. Pero para entender bien los decoradores, necesitamos ir un paso más allá. Las funciones también pueden devolver otras funciones.

### 3.7.3 Devolviendo funciones desde funciones:

No es necesario ejecutar una función dentro de otra. Simplemente podemos devolverla como salida:

```
def hola(nombre="Covadonga"):
    def saluda():
        return "Estás dentro de la función saluda()"

    def bienvenida():
        return "Estás dentro de la función bienvenida()"

    if nombre == "Covadonga":
        return saluda
    else:
        return bienvenida

a = hola()
print(a)
#Salida: <function saluda at 0x7f2143c01500>

#Es decir, la variable 'a' ahora apunta a la función
# saluda() declarada dentro de hola(). Por lo tanto podemos llamarla.

print(a())
#Salida: Estás dentro de la función saluda()
```

Echa un vistazo otra vez al código. Si te fijas en el if/else, estamos devolviendo `saluda` y `bienvenida` y no `saluda()` y `bienvenida()`. ¿A qué se debe esto? Se debe a que cuando usas paréntesis `()` la función se ejecuta. Por lo contrario, si no los usas la función es pasada y puede ser asignada a una variable sin ser ejecutada.

Vamos a analizar el código paso por paso. Al principio usamos `a = hola()`, por lo que el parámetro para `nombre` que se toma es Covadonga ya que es el que hemos asignado por defecto. Esto hará que en el `if` se entre en `nombre == "Covadonga"`, lo que hará que se devuelva la función saluda. Si por lo contrario hacemos la llamada a la función con `a = hola(nombre="Pelayo")`, la función devuelta será `bienvenida`.

### 3.7.4 Usando funciones como argumento de otras:

Por último, podemos hacer que una función tenga a otra como entrada y que además la ejecute dentro de sí misma. En el siguiente ejemplo podemos ver como `hazEstoAntesDeHola()` es una función que de alguna forma encapsula a la función que se le pase como parámetro, añadiendo una determinada funcionalidad. En este ejemplo simplemente imprimimos algo por pantalla antes de llamar a la función.

```
def hola():
    return "¡Hola!"

def hazEstoAntesDeHola(func):
    print("Hacer algo antes de llamar a func")
    print(func())
```

(continues on next page)

(continued from previous page)

```
hazEstoAntesDeHola(hi)
#Salida: Hacer algo antes de llamar a func
#        ¡Hola!
```

Ahora ya tienes todas las piezas del rompecabezas. Los decoradores son funciones que decoran a otras funciones, pudiendo ejecutar código antes y después de la función que está siendo decorada.

### 3.7.5 Tu primer decorador:

Realmente en el ejemplo anterior ya vimos como crear un decorador. Vamos a modificarlo y hacerlo un poco realista.

```
def nuevo_decorador(a_func):

    def envuelveLaFuncion():
        print("Haciendo algo antes de llamar a a_func()")

        a_func()

        print("Haciendo algo después de llamar a a_func()")

    return envuelveLaFuncion

def funcion_a_decorar():
    print("Soy la función que necesita ser decorada")

funcion_a_decorar()
#Salida: "Soy la función que necesita ser decorada"

funcion_a_decorar = nuevo_decorador(funcion_a_decorar)
#Ahora funcion_a_decorar está envuelta con el decorador que hemos creado

funcion_a_decorar()
#Salida: Haciendo algo antes de llamar a a_func()
#        Soy la función que necesita ser decorada
#        Haciendo algo después de llamar a a_func()
```

Simplemente hemos aplicado todo lo aprendido en los apartados anteriores. Así es exactamente como funcionan los decoradores en Python. Envuelven una función para modificar su comportamiento de una manera determinada.

Tal vez te preguntes ahora porqué no hemos usado @ en el código. Esto es debido a que @ es simplemente una forma de hacerlo más corto, pero ambas opciones son perfectamente válidas.

```
@nuevo_decorador
def funcion_a_decorar():
```

(continues on next page)

(continued from previous page)

```
    print("Soy la función que necesita ser decorada")

funcion_a_decorar()
#Salida: Haciendo algo antes de llamar a a_func()
#        Soy la función que necesita ser decorada
#        Haciendo algo después de llamar a a_func()

#El uso de @nuevo_decorador es simplemente una forma acortada
#de hacer lo siguiente.
funcion_a_decorar = nuevo_decorador(funcion_a_decorar)
```

Una vez visto esto, hay un pequeño problema con el código. Si ejecutamos lo siguiente:

```
print(funcion_a_decorar.__name__)
# Output: envuelveLaFuncion
```

Nos encontramos con un comportamiento un tanto inesperado. Nuestra función es `funcion_a_decorar` pero al haberla envuelto con el decorador es en realidad `envuelveLaFuncion`, por lo que sobreescribe el nombre y el *docstring* de la misma, algo que no es muy conveniente. Por suerte, Python nos da una forma de arreglar este problema usando `functools.wraps`. Vamos a modificar nuestro ejemplo anterior haciendo uso de esta herramienta.

```
from functools import wraps

def nuevo_decorador(a_func):
    @wraps(a_func)
    def envuelveLaFuncion():
        print("Haciendo algo antes de llamar a a_func()")
        a_func()
        print("Haciendo algo después de llamar a a_func()")
    return envuelveLaFuncion

@nuevo_decorador
def funcion_a_decorar():
    print("Soy la función que necesita ser decorada")

print(funcion_a_decorar.__name__)
# Salida: funcion_a_decorar
```

Mucho mejor ahora. Veamos también unos fragmentos de código muy usados.

**Ejemplos:**

```
from functools import wraps
def nombre_decorador(f):
    @wraps(f)
    def decorada(*args, **kwargs):
        if not can_run:
```

(continues on next page)

(continued from previous page)

```
            return "La función no se ejecutará"
        return f(*args, **kwargs)
    return decorada

@nombre_decorador
def func():
    return("La función se esta ejecutando")

can_run = True
print(func())
# Salida: La función se esta ejecutando

can_run = False
print(func())
# Salida: La función no se ejecutará
```

Nota: `@wraps` toma una función para ser decorada y añade la funcionalidad de copiar el nombre de la función, el *docstring*, los argumentos y otros parámetros asociados. Esto nos permite acceder a los elementos de la función a decorar una vez decorada. Es decir, resuelve el problema que vimos con anterioridad.

Casos de uso:

A continuación veremos algunos áreas en las que los decoradores son realmente útiles.

Autorización

Los decoradores permiten verificar si alguien está o no autorizado a usar una determinada función, por ejemplo en una aplicación web. Son muy usados en *frameworks* como Flask o Django. Aquí te mostramos como usar un decorador para verificar que se está autenticado.

**Ejemplo :**

```
from functools import wraps

def requires_auth(f):
    @wraps(f)
    def decorated(*args, **kwargs):
        auth = request.authorization
        if not auth or not check_auth(auth.username, auth.password):
            authenticate()
        return f(*args, **kwargs)
    return decorated
```

Iniciar sesión

El inicio de sesión es otra de las áreas donde los decoradores son muy útiles. Vamos un ejemplo:

```
from functools import wraps

def logit(func):
    @wraps(func)
    def with_logging(*args, **kwargs):
        print(func.__name__ + " was called")
        return func(*args, **kwargs)
    return with_logging

@logit
def addition_func(x):
   """Función suma"""
   return x + x


result = addition_func(4)
# Salida: addition_func was called
```

### 3.7.6 Decoradores con argumentos

Hemos visto ya el uso de `@wraps`, y tal vez te preguntes ¿pero no es también un decorador? De hecho si te fijas acepta un parámetro (que en nuestro caso es una función). A continuación te explicamos como crear un decorador que también acepta parámetros de entrada.

Anidando un Decorador dentro de una Función

Vayamos de vuelta al ejemplo de inicio de sesión, y creemos un *wraper* que permita especificar el fichero de salida que queremos usar para el fichero de *log*. Si te fijas, el decorador ahora acepta un parámetro de entrada.

```
from functools import wraps

def logit(logfile='out.log'):
    def logging_decorator(func):
        @wraps(func)
        def wrapped_function(*args, **kwargs):
            log_string = func.__name__ + " fue llamada"
            print(log_string)
            # Abre el fichero y añade su contenido
            with open(logfile, 'a') as opened_file:
                # Escribimos en el fichero el contenido
```

(continues on next page)

(continued from previous page)

```
                opened_file.write(log_string + '\n')
            return func(*args, **kwargs)
        return wrapped_function
    return logging_decorator

@logit()
def myfunc1():
    pass

myfunc1()
# Salida: myfunc1 fue llamada
# Se ha creado un fichero con el nombre por defecto (out.log)

@logit(logfile='func2.log')
def myfunc2():
    pass

myfunc2()
# Salida: myfunc2  fue llamada
# Se crea un fichero func2.log
```

### Clases Decoradoras

Llegados a este punto ya tenemos el decorador *logit* creado en el apartado anterior funcionando en producción, pero algunas partes de nuestra aplicación son críticas, y si se produce un fallo este necesitará atención inmediata. Vamos a suponer que en determinadas ocasiones quieres simplemente escribir en el *log* (como hemos hecho), pero en otras quieres que se envíe un correo. En una aplicación como esta podríamos usar la herencia, pero hasta ahora sólo hemos usado decoradores.

Por suerte, las clases también pueden ser usadas para crear decoradores. Vamos a volver a definir *logit*, pero en este caso como una clase en vez de con una función.

```
class logit(object):

    _logfile = 'out.log'

    def __init__(self, func):
        self.func = func

    def __call__(self, *args):
        log_string = self.func.__name__ + " fue llamada"
        print(log_string)
        # Abre el fichero de log y escribe
        with open(self._logfile, 'a') as opened_file:
            # Escribimos el contenido
            opened_file.write(log_string + '\n')
        # Enviamos una notificación (ver método)
```

(continues on next page)

(continued from previous page)

```
        self.notify()

        # Devuelve la función base
        return self.func(*args)



    def notify(self):
        # Esta clase simplemente escribe el log, nada más.
        pass
```

Esta implementación es mucho más limpia que con la función anidada. Por otro lado, la función puede ser envuelta de la misma forma que veníamos usando hasta ahora, usando `@`.

```
logit._logfile = 'out2.log' # Si queremos usar otro nombre
@logit
def myfunc1():
    pass

myfunc1()
# Output: myfunc1 fue llamada
```

Ahora, vamos a crear una subclase de *logit* para añadir la funcionalidad de enviar un email. Enviaremos el email de manera ficticia.

```
class email_logit(logit):
    '''
    Implementación de logit con envío de email
    '''
    def __init__(self, email='admin@myproject.com', *args, **kwargs):
        self.email = email
        super(email_logit, self).__init__(*args, **kwargs)

    def notify(self):
        # Enviamos email a self.email
        # Código para enviar email
        # ...
        pass
```

Una vez creada la nueva clase que hereda de `logit`, si usamos `@email_logit` como decorador tendrá el mismo comportamiento, pero además enviará un email.

Si quieres saber más acerca de los decoradores, en este post tienes más información.

## 3.8 Global & Return

Estoy seguro de que por poco código en Python que hayas visto, te habrás encontrado la sentencia `return` al final de una función alguna vez. Al igual que en muchos lenguajes de programación, nos permite devolver valores a quien llama a la función. Veamos un ejemplo:

```
def suma(valor1, valor2):
    return valor1 + valor2

resultado = suma(3, 5)
print(resultado)
# Salida: 8
```

La función anterior toma dos argumentos de entrada y como salida devuelve su suma. Otra forma de conseguir el mismo resultado podría haber sido:

```
def suma(valor1, valor2):
    global resultado
    result = valor1 + valor2

suma(3,5)
print(resultado)
# Salida: 8
```

Vamos a analizar ambos ejemplos. Empecemos por el primero, en el que usábamos la sentencia `return`. Lo que esa función hace, es devolver el resultado, y ese resultado puede ser asignado como hemos visto a una variable con =. Esta es una de las formas más frecuentes de «sacar» contenido de la función, ya que de no hacerlo, la variable `resultado` se perdería.

Por otro lado, hemos visto como se puede hacer uso de `global`. Por norma general se podría decir que salvo que estemos muy seguros de lo que estamos haciendo, no es muy común hacer uso de variables globales. Lo que hace el modificador `global` es crear una variable global. Por lo tanto, dado que esa variable es global seguirá existiendo una vez se salga de la función, y por consiguiente puede ser accedida una vez la función ha terminado de ejecutarse. Vamos a ver un ejemplo.

```
# Sin usar global
def suma(valor1, valor2):
    resultado = valor1 + valor2

suma(2, 4)
print(resultado)

# Si intentamos acceder a la variable resultado
# tendremos un error ya que ha sido declarada dentro
# de la función, y por lo tanto no es accesible desde
# fuera
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
  File "", line 1, in
    result
NameError: name 'resultado' is not defined

# Ahora vamos a hacer lo mismo pero declarando la función
# como global.
def suma(valor1, valor2):
    global resultado
    resultado = valor1 + valor2

suma(2, 4)
print(resultado)
# Salida
# 6
```

Y como hemos explicado la segunda forma se ejecutará sin problemas. Sin embargo ten cuidado con el uso de `global`, ya que suele ser una buena práctica evitar su uso. No es muy recomendable tener variables globales salvo casos muy excepcionales.

### 3.8.1 Devolviendo múltiples valores

Tal vez quieras devolver más de una variable desde una función. Una primera forma de hacerlo sería la siguiente, pero de verdad, no te recomendamos que lo hagas así. Huye de esto y no mires atrás:

```
def perfil():
    global nombre
    global edad
    name = "Pelayo"
    age = 30

perfil()
print(nombre)
# Salida: Pelayo

print(edad)
# Salida: 30
```

**Nota:** No hagas esto. Tal vez te preguntes porqué mostramos código que no está bien. Pues bien, nos gusta mostrar también ejemplos de lo que está mal, ya que ayudan a entender lo que no se debe hacer.

Otra forma mucho mejor de hacer esto, es devolviendo los datos dentro de una estructura tipo `tuple`, `list` o `dict`. Una forma de hacerlo sería la siguiente:

```
def perfil():
    nombre = "Pelayo"
```

(continues on next page)

(continued from previous page)

```
    edad = 30
    return (nombre, edad)

datos_perfil = perfil()
print(datos_perfil[0])
# Salida: Pelayo

print(datos_perfil[1])
# Salida: 30
```

Y otra forma prácticamente igual pero más usada por convención sería la siguiente.

```
def perfil():
    nombre = "Pelayo"
    edad = 30
    return nombre, edad

nombre_perfil, edad_perfil = perfil()
print(nombre_perfil)
# Salida: Pelayo
print(edad_perfil)
# Salida: 30
```

Ten en cuenta que en el ejemplo anterior también se está devolviendo una tupla (aunque no haya paréntesis). Vistas estas formas, se podría decir que hay otra forma un poco más completa que tal vez te sea útil. Se trata del uso de namedtuple. Veamos un ejemplo:

```
from collections import namedtuple
def perfil():
    Persona = namedtuple('Persona', 'nombre edad')
    return Persona(nombre="Pelayo", edad=31)

# Usando el namedtuple
p = perfil()
print(p, type(p))
# Persona(nombre='Pelayo', edad=31) <class '__main__.Persona'>
print(p.nombre)
# Pelayo
print(p.edad)
#31

# Otra forma de usar la namedtuple
p = perfil()
print(p[0])
# Pelayo
print(p[1])
#31
```

(continues on next page)

(continued from previous page)

```
# También se puede hacer el unpacking
nombre, edad = profile()
print(nombre)
# Pelayo
print(edad)
#31
```

Esta forma es bastante útil sobre todo debido a que podemos acceder a los elementos de forma muy sencilla usando . y el argumento. Como hemos mencionado, otra forma de hacerlo sería con `lists` y `dicts`, pero como ya hemos comentado, intenta evitar `global` en la medida de lo posible.

## 3.9 Mutabilidad

Los tipos mutables e inmutables en Python son conceptos que causan verdaderos quebraderos de cabeza a los programadores. En otras palabras, **mutable** significa «que puede cambiar» e **inmutable** significa «que es constante». ¿Quieres ver lo enrevesado que puede parecer si no se entiende correctamente? Veamos un ejemplo:

```
foo = ['hola']
print(foo)
# Salida: ['hola']

bar = foo
bar += ['adios']
print(foo)
# Output: ['hola', 'adios']
```

¿Que ha ocurrido? Hemos modificado la variable `bar`, pero la variable `foo` también ha sido modificada. Tal vez te esperabas algo como lo que mostramos a continuación:

```
foo = ['hola']
print(foo)
# Salida: ['hola']

bar = foo
bar += ['adios']

print(foo)
# Salida esperada: ['hola']
# Salida: ['hola', 'adios']

print(bar)
# Salida: ['hola', 'adios']
```

Evidentemente no se trata de un fallo, sino que lo que estamos viendo es la mutabilidad en acción. Cuando asignas una variable a otra variable que es de tipo mutable,

cualquier cambio que hagas sobre la segunda, afectará a la primera y viceversa. La variable nueva que hemos creado al hacer `bar = foo` es simplemente un *alias* de la primera. Por lo tanto cualquier modificación sobre `bar` afectará también a `foo`. No obstante, esto es solo cierto para los tipos mutables.

Veamos otro ejemplo:

```
def agrega(num, target=[]):
    target.append(num)
    return target

agrega(1)
# Salida: [1]

agrega(2)
# Salida: [1, 2]

agrega(3)
# Salida: [1, 2, 3]
```

Tal vez te esperabas otro comportamiento, ya que en cada llamada a `agrega` estamos creando una lista nueva vacía. Sería razonable esperar que la salida fuera la siguiente:

```
def agrega(num, target=[]):
    target.append(num)
    return target

agrega(1)
# Salida: [1]

agrega(2)
# Salida: [2]

agrega(3)
# Salida: [3]
```

Otra vez, estamos viendo la mutabilidad en acción. En Python, los argumentos por defecto se evalúan una vez que la función ha sido definida, no cada vez que la función es llamada. Por lo tanto, nunca deberías definir un argumento por defecto de un tipo mutable, a menos que realmente estés seguro de lo que estas haciendo. El siguiente ejemplo sería más correcto:

```
def agrega(element, target=None):
    if target is None:
        target = []
    target.append(element)
    return target
```

Ahora cada vez que llamamos a la función sin el argumento `target`, una nueva lista será creada. Por ejemplo:

```
agrega(42)
# Salida: [42]

agrega(42)
# Salida: [42]

agrega(42)
# Salida: [42]
```

## 3.10 Método mágico __slots__

En Python cualquier clase tiene atributos de instancia. Por defecto se usa un diccionario para almacenar los atributos de un determinado objeto, y esto es algo muy útil que permite por ejemplo crear nuevos atributos en tiempo de ejecución.

Sin embargo, para clases pequeñas con atributos conocidos, puede llegar a resultar un cuello de botella. El uso del diccionario `dict` desperdicia un montón de memoria RAM y Python no puede asignar una cantidad de memoria estática para almacenar los atributos. Por lo tanto, se come un montón de RAM si creas muchos objetos (del orden de miles o millones). Por suerte hay una forma de solucionar esto, haciendo uso de `__slots__`, que permite decirle a Python que no use un diccionario y que solo asigne memoria para una cantidad fija de atributos. Aquí mostramos un ejemplo del uso de `__slots__`:

**Sin usar** `__slots__`:

```
class MiClase(object):
    def __init__(self, nombre, identificador):
        self.nombre = nombre
        self.identificador = identificador
        self.iniciar()
    # ...
```

**Usando** `__slots__`:

```
class MiClase(object):
    __slots__ = ['nombre', 'identificador']
    def __init__(self, nombre, identificador):
        self.nombre = nombre
        self.identificador = identificador
        self.iniciar()
    # ...
```

El segundo código reducirá el uso de RAM. En alguna ocasiones se han reportado reducciones de hasta un 40 o 50% usando esta técnica.

Como nota adicional, tal vez quieras echar un vistazo a PyPy, ya que hace este tipo de optimizaciones por defecto.

En el siguiente ejemplo puedes ver el uso exacto de memoria con y sin "__slots__"hecho en IPython gracias a https://github.com/ianozsvald/ipython_memory_usage

```
Python 3.4.3 (default, Jun  6 2015, 13:32:34)
Type "copyright", "credits" or "license" for more information.

IPython 4.0.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: import ipython_memory_usage.ipython_memory_usage as imu

In [2]: imu.start_watching_memory()
In [2] used 0.0000 MiB RAM in 5.31s, peaked 0.00 MiB above current, total RAM␣
↪usage 15.57 MiB

In [3]: %cat slots.py
class MyClass(object):
        __slots__ = ['name', 'identifier']
        def __init__(self, name, identifier):
                self.name = name
                self.identifier = identifier

num = 1024*256
x = [MyClass(1,1) for i in range(num)]
In [3] used 0.2305 MiB RAM in 0.12s, peaked 0.00 MiB above current, total RAM␣
↪usage 15.80 MiB

In [4]: from slots import *
In [4] used 9.3008 MiB RAM in 0.72s, peaked 0.00 MiB above current, total RAM␣
↪usage 25.10 MiB

In [5]: %cat noslots.py
class MyClass(object):
        def __init__(self, name, identifier):
                self.name = name
                self.identifier = identifier

num = 1024*256
x = [MyClass(1,1) for i in range(num)]
In [5] used 0.1758 MiB RAM in 0.12s, peaked 0.00 MiB above current, total RAM␣
↪usage 25.28 MiB

In [6]: from noslots import *
In [6] used 22.6680 MiB RAM in 0.80s, peaked 0.00 MiB above current, total RAM␣
↪usage 47.95 MiB
```

Se puede ver una clara reducción en el uso de RAM 9.3008 MiB vs 22.6680 MiB.

## 3.11 Entornos virtuales

Los entornos virtuales o *virtual environments* son una herramienta muy potente que es parte de cualquier desarrollador de Python. Entonces, ¿qué son los `virtualenv`?

Se trata de una herramienta que permite crear entornos virtuales de Python totalmente aislados. Imagina que tienes una aplicación que requiere la versión 2 de Python, pero que también tienes otra que requiere Python 3. ¿Cómo puedes usar ambas aplicaciones? O también puedes tener diferentes aplicaciones que usan diferentes versiones de un determinado paquete. ¿Cómo podemos hacer? La respuesta son los `virtualenv`.

Si instalas todo en `/usr/lib/python2.7/site-packages` (o el directorio que tengas si usas Windows o cualquier otra plataforma) cualquiera de tus aplicaciones compartirá por defecto el contenido de esa carpeta, como por ejemplo las versiones de los paquetes que usas. El problema es que es normal acabar actualizando algún paquete, y esto puede ser algo que nos interese para una aplicación que tengamos, pero no para otra.

La solución a este problema es usar `virtualenv` para crear entornos completamente aislados unos de otros. Por lo tanto, tus aplicaciones usarán uno determinado, y los nuevos paquetes o actualizaciones que instales en uno no afectaran a otros.

Para instalar esta herramienta, basta con ejecutar el siguiente comando en el terminal:

```
$ pip install virtualenv
```

Los comandos más importantes son los siguientes:

- `$ virtualenv myproject`
- `$ source myproject/bin/activate`

El primero crea un entorno virtual en la carpeta `myproject`, y el segundo activa ese entorno.

Cuando creas un entorno virtual es necesario tomar una decisión sobre si quieres usar los paquetes que están por defecto en tu sistema en `site-packages`. Es importante notar que por defecto `virtualenv` no tiene acceso a `site-packages`.

Si quieres que tu entorno tenga acceso a los paquetes instalados en tu sistema en `site-packages` puedes usar `--system-site-packages`.

```
$ virtualenv --system-site-packages mycoolproject
```

Por otro lado, puedes desactivar el entorno de la siguiente manera:

```
$ deactivate
```

Ejecutando *python* después de desactivarlo, se usará el conjunto de paquetes que tengas instalado en el sistema (`site-packages`).

**Extra**

Puedes usar la librería `smartcd`, que permite que en *bash* y *zsh* puedas cambiar de entorno al hacer un cambio de directorio con `cd`. Puede ser realmente útil si tienes

varios proyectos con diferentes entornos y quieres navegar por ellos, ya que el entorno se irá activando o desactivando según el directorio en el que estés. Puedes leer más acerca de este proyecto en su GitHub.

Y hasta aquí esta rápida introducción de los `virtualenv`. Hay mucho más que esto, por lo que si quieres saber más, te recomendamos el siguiente enlace.

## 3.12 Colecciones

Python viene con un modulo que contiene varios contenedores de datos llamados colecciones o **collections** en Inglés. Hablaremos de algunos de ellos y de sus usos.

En concreto, hablaremos de los siguientes:

- `defaultdict`
- `OrderedDict`
- `counter`
- `deque`
- `namedtuple`
- `enum.Enum` (fuera del módulo; Python 3.4+)

### 3.12.1 `defaultdict`

Personalmente uso **defaultdict** bastante. A diferencia de `dict` con `defaultdict` no tienes que verificar que una llave o *key* este presente. Es decir, puedes hacer lo siguiente:

```
from collections import defaultdict

colours = (
    ('Asturias', 'Oviedo'),
    ('Galicia', 'Ourense'),
    ('Extremadura', 'Cáceres'),
    ('Galicia', 'Pontevedra'),
    ('Asturias', 'Gijón'),
    ('Cataluña', 'Barcelona'),
)

ciudades = defaultdict(list)

for name, colour in colours:
    ciudades[name].append(colour)

print(ciudades)
```

(continues on next page)

(continued from previous page)

```
# Salida
# defaultdict(<type 'list'>,
#    {'Extremadura': ['Cáceres'],
#     'Asturias': ['Oviedo', 'Gijón'],
#     'Cataluña': ['Silver'],
#     'Galicia': ['Ourense', 'Pontevedra']
# })
```

Una de las ocasiones en las que son más útiles, es si quieres añadir elementos a listas anidadas dentro e un diccionario. Si la llave o *key* no está ya presente en el diccionario, tendrás un error tipo `KeyError`. El uso de `defaultdict` permite evitar este problema. Antes de nada, vamos a ver un ejemplo con `dict` que daría un error `KeyError` como hemos mencionado, y después veremos la solución usando `defaultdict`.

**Problema:**

```
some_dict = {}
some_dict['region']['ciudad'] = "Oviedo"
# Raises KeyError: 'region'
```

**Solución:**

```
from collections import defaultdict
tree = lambda: defaultdict(tree)
some_dict = tree()
some_dict['region']['ciudad'] = "Oviedo"
# ¡Funciona!
```

Ahora podrías imprimir también el diccionario `some_dict` usando `json.dumps`. Aquí tienes un ejemplo:

```
import json
print(json.dumps(some_dict))
# Output: {"region": {"ciudad": "Oviedo"}}
```

### 3.12.2 `OrderedDict`

`OrderedDict` es un diccionario que mantiene ordenadas sus entradas según van siendo añadidas. Es importante saber también que sobreescribir un valor existente no cambia la posición de la llave o *key*. Sin embargo, eliminar y reinsertar una entrar mueve la llave al final del diccionario.

**Problema:**

```
colours =  {"Rojo" : 198, "Verde" : 170, "Azul" : 160}
for key, value in colours.items():
    print(key, value)
# Salida:
```

(continues on next page)

(continued from previous page)

```
#   Verde 170
#   Azul 160
#   Rojo 198
# Las entradas son recuperadas en un orden no predecible.
```

**Solución:**

```
from collections import OrderedDict

colours = OrderedDict([("Rojo", 198), ("Verde", 170), ("Azul", 160)])
for key, value in colours.items():
    print(key, value)
# Output:
#   Rojo 198
#   Verde 170
#   Azul 160
# El orden de inserción se mantiene.
```

### 3.12.3 counter

El uso de counter nos permite contar el número de elementos que una llave tiene. Por ejemplo, puede ser usado para contar el número de colores favoritos de diferentes personas.

```
from collections import Counter

colours = (
    ('Covadonga', 'Amarillo'),
    ('Pelayo', 'Azul'),
    ('Xavier', 'Verde'),
    ('Pelayo', 'Negro'),
    ('Covadonga', 'Rojo'),
    ('Amaya', 'Plata'),
)

favs = Counter(name for name, colour in colours)
print(favs)
# Salida: Counter({
#    'Covadonga': 2,
#    'Pelayo': 2,
#    'Xavier': 1,
#    'Amaya': 1
# })
```

También podemos contar las líneas más comunes de un fichero, como por ejemplo:

```
with open('nombre_fichero', 'rb') as f:
    line_count = Counter(f)
print(line_count)
```

### 3.12.4 deque

deque proporciona una cola con dos lados, lo que significa que puedes añadir y eliminar elementos de cualquiera de los lados de la cola. Primero debes importar el módulo de la librería de colecciones o *collections*:

```
from collections import deque
```

Una vez importado ya podemos crear el objeto:

```
d = deque()
```

Tienen un comportamiento relativamente similar a las conocidas listas de Python, y sus métodos son también similares. Puedes hacer lo siguiente:

```
d = deque()
d.append('1')
d.append('2')
d.append('3')

print(len(d))
# Salida: 3

print(d[0])
# Salida: '1'

print(d[-1])
# Salida: '3'
```

También puedes tomar elementos de los dos lados de la cola, una funcionalidad conocida como *pop*. Es importante notar que *pop* devuelve el elemento eliminado.

```
d = deque(range(5))
print(len(d))
# Salida: 5

d.popleft()
# Salida: 0

d.pop()
# Salida: 4

print(d)
# Salida: deque([1, 2, 3])
```

También podemos limitar la cantidad de elementos que la cola deque puede almacenar. Al hacer esto, simplemente quitará elementos del otro lado de la cola si el límite es superado. Se ve mejor con un ejemplo como se muestra a continuación:

```
d = deque([0, 1, 2, 3, 5], maxlen=5)
print(d)
# Salida: deque([0, 1, 2, 3, 5], maxlen=5)

d.extend([6])
print(d)
#Salida: deque([1, 2, 3, 5, 6], maxlen=5)
```

Ahora cuando insertamos valores después del 5, la parte más a la izquierda será eliminada de la lista. También puedes expandir la lista en cualquier dirección con valores nuevos.

```
d = deque([1,2,3,4,5])
d.extendleft([0])
d.extend([6,7,8])
print(d)
# Salida: deque([0, 1, 2, 3, 4, 5, 6, 7, 8])
```

### 3.12.5 namedtuple

Tal vez conozcas ya las tupas, que son listas inmutables que permiten almacenar una secuencia de valores separados por coma. Son simplemente como las listas pero con algunas diferencias importantes. La principal es que a diferencia de las listas **no puedes reasignar el valor de un elemento** una vez inicializada. Para acceder a un índice de la tupla se hace de la siguiente manera:

```
man = ('Pelayo', 30)
print(man[0])
# Output: Pelayo
```

Sabido esto, ¿qué son las namedtuples?. Se trata de un tipo que convierte las tuplas en contenedores bastante útiles para tareas simples. Con ellas, no necesitas usar índices enteros para acceder a los miembros de la misma. Puedes pensar en ellas como si fuesen diccionarios, con la salvedad de que son inmutables. Veamos un ejemplo.

```
from collections import namedtuple

Animal = namedtuple('Animal', 'nombre edad tipo')
perry = Animal(nombre="perry", edad=31, tipo="cat")

print(perry)
# Salida: Animal(nombre='perry', edad=31, tipo='cat')

print(perry.nombre)
# Salida: 'perry'
```

Puedes ver como es posible acceder a los elementos a través de su nombre, simplemente haciendo uso de `.`. Vamos a verlo con más detalle. Una `namedtuple` requiere de dos argumentos. Estos son, el nombre de la tupla y los campos de la misma. En el ejemplo anterior hemos visto como el nombre de la tupla era "Animal" y tenía tres atributos: "nombre", "edad" y "tipo".

Las `namedtuple` son muy útiles ya que hacen que las tuplas tengan una especie de documentación propia, y apenas sea necesaria una explicación de como usarlas, ya que puedes verlo con un simple vistazo al código. Además, dado que no es necesario usar índices, hace que sea más fácil de mantener.

Otra de las ventajas es que son bastante ligeras, y no necesitan mas memoria que las tuplas normales. Esto hace que sean mas rápidas que los diccionarios. Sin embargo, recuerda que los atributos de las tuplas son inmutables, por lo que no pueden ser modificados. El siguiente ejemplo no funcionaría:

```
from collections import namedtuple

Animal = namedtuple('Animal', 'nombre edad tipo')
perry = Animal(nombre="perry", edad=31, tipo="cat")
perry.edad = 42

# Salida: Traceback (most recent call last):
#           File "", line 1, in
#         AttributeError: can't set attribute
```

Deberías usar las `namedtuple` si quieres que tu código sea autodocumentado. Lo mejor de todo es que ofrecen compatibilidad con las tuplas, por lo que **puedes indexarlas como si de una tupla normal se tratase**. Veamos un ejemplo:

```
from collections import namedtuple

Animal = namedtuple('Animal', 'nombre edad tipo')
perry = Animal(nombre="perry", edad=31, tipo="cat")
print(perry[0])
# Salida: perry
```

Por último, aunque no por ello menos importante, puedes convertir una namedtuple en un diccionario. Se puede hacer de la siguiente manera:

```
from collections import namedtuple

Animal = namedtuple('Animal', 'nombre edad tipo')
perry = Animal(nombre="Perry", edad=31, tipo="cat")
print(perry._asdict())
# Salida: OrderedDict([('nombre', 'Perry'), ('edad', 31), ...
```

### 3.12.6 `enum.Enum` (Python 3.4+)

Otra de las colecciones más útiles de Python es el tipo **enum**, que se encuentra disponible en el módulo `enum` desde Python 3.4 en adelante (también está disponible como *backport* en PyPI bajo el nombre `enum32`). Los enums (enumerated type) son básicamente una forma de organizar aquellos nombres que puedan tomar un determinado número de estados limitados y claramente definidos.

Vamos a considerar el ejemplo anterior en namedtuples del Animal. Si recuerdas, había un campo denominado `tipo`. El problema de este tipo es que era una cadena. ¿Qué pasaría si escribimos `Gato` o `GATO`?

El uso de enum nos puede ayudar a resolver este problema, evitando por lo tanto usar cadenas. Veamos el siguiente ejemplo:

```
from collections import namedtuple
from enum import Enum

class Especies(Enum):
    gato = 1
    perro = 2
    caballo = 3
    lobo = 4
    mariposa = 5
    buho = 6
    # ¡Y muchos más!

    # Se pueden usar también alias
    gatito = 1
    perrito = 2

Animal = namedtuple('Animal', 'name age type')
perry = Animal(name="Perry", age=31, type=Especies.gato)
caballo = Animal(name="HorseLuis", age=4, type=Especies.caballo)
tom = Animal(name="Tom", age=75, type=Especies.lobo)
luna = Animal(name="Luna", age=35, type=Especies.gatito)

# Y un ejemplo
>>> perry.type == luna.type
True
>>> luna.type
<Especies.gato: 1>
```

Un código así es mucho menos propenso a tener fallos. Si necesitamos ser específicos, deberíamos usar sólo los tipos enumerados.

Por último, existen tres formas de acceder a los enum. Sigamos con el ejemplo anterior de las especies. Vamos a acceder a **gato**:

```
Especies(1)
Especies['cat']
```

(continues on next page)

(continued from previous page)

```
Especies.cat
```

Con esto finalizamos una breve introducción al módulo de `collections` de Python. Si quieres saber más, te recomendamos que leas la documentación oficial de Python, que aunque pueda ser un poco más técnica y menos didáctica, con esta introducción ya deberías estar list@ para entenderla.

## 3.13 Enumerados

Python viene con un tipo por defecto denominado Enumerate. Permite asignar índices a elementos de, por ejemplo una lista. Veamos un ejemplo:

```
for contador, valor in enumerate(lista):
    print(contador, valor)
```

También acepta un parametro opcional que que lo hace aún más útil.

```
mi_lista = ['Ibias', 'Pesoz', 'Tineo', 'Boal']
for c, valor in enumerate(mi_lista, 1):
    print(c, valor)

# Salida:
# 1 Ibias
# 2 Pesoz
# 3 Tinero
# 4 Boal
```

Este argumento opcional nos permite decirle al `enumerate` el primer elemento del índice. También puedes creas tuplas que contengan el índice y la lista. Por ejemplo:

```
mi_lista = ['Ibias', 'Pesoz', 'Tineo', 'Boal']
lista_contador = list(enumerate(mi_lista, 1))
print(lista_contador)
# Salida: [(1, 'Ibias'), (2, 'Pesoz'), (3, 'Tineo'), (4, 'Boal')]
```

## 3.14 Introspección de objetos

En el mundo de la programación, la **instrospección** es la habilidad para determinar el tipo de un objeto en tiempo de ejecución, y se trata de una de las mejores características de Python. En Python todo es un objeto, y podemos examinarlos de manera muy sencilla con las funciones por defecto que se nos proporcionan.

### 3.14.1 `dir`

A continuación explicaremos el uso de `dir` y como podemos usarla. Se trata de una de las funciones clave para la introspección de objetos en Python. Nos devuelve una lista con todos los atributos y métodos que un determinado objeto tiene.

```
mi_lista = [1, 2, 3]
dir(mi_lista)
# Salida: ['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
# '__delslice__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
# '__getitem__', '__getslice__', '__gt__', '__hash__', '__iadd__', '__imul__',
# '__init__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__',
# '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__
↪',
# '__setattr__', '__setitem__', '__setslice__', '__sizeof__', '__str__',
# '__subclasshook__', 'append', 'count', 'extend', 'index', 'insert', 'pop',
# 'remove', 'reverse', 'sort']
```

Como podemos ver, se nos devuelven todos los atributos y métodos en una lista. Esto puede ser útil si no recuerdas el nombre de un método y no tienes la documentación a mano. Si ejecutamos `dir()` sin ningún argumento, se nos devolverá todos los nombres en el *scope* actual.

### 3.14.2 `type` y `id`

La función `type` nos devuelve el tipo de un objeto:

```
print(type(''))
# Salida: <type 'str'>

print(type([]))
# Salida: <type 'list'>

print(type({}))
# Salida: <type 'dict'>

print(type(dict))
# Salida: <type 'type'>

print(type(3))
# Salida: <type 'int'>
```

Por otro lado, `id` devuelve un *id* o identificador único para cada objeto.

```
nombre = "Pelayo"
print(id(nombre))
# Salida: 139972439030304
```

### 3.14.3 Módulo `inspect`

El módulo `inspect` nos proporciona diferentes funciones para consultar información de objetos. Por ejemplo, puedes consultar los miembros de un objeto ejecutando el siguiente código:

```
import inspect
print(inspect.getmembers(str))
# Salida: [('__add__', <slot wrapper '__add__' of ... ...
```

Existen también otros métodos para realizar introspección sobre objetos. Te recomendamos que consultes la documentación oficial y leas sobre ellos.

## 3.15 Comprensión

La comprensión o *comprehensions* en Python son una de las características que una vez sabes usarlas, echarias mucho de menos si las quitaran. Se trata de un tipo de construcción que permite crear secuencias a partir de otras secuencias. Existen diferentes *comprehensions* soportadas en Python 2 y Python 3:

- Comprensión de listas
- Comprensión de diccionarios
- Comprensión de sets
- Comprensión de generadores

A continuación las explicaremos una por una. Una vez que entiendes el uso con las listas, cualquiera de las otras será entendia my fácilmente.

### 3.15.1 Comprensión de `list`

Las comprensiones de listas nos proporcionan una forma corta y concisa de crear listas. Se usan con corchetes `[]` y en su interior contienen una expresión seguida de un bucle `for` y cero o más sentencias `for` o `if`. La expresión puede ser cualquier cosa que se te ocurra, lo que significa que puedes usar cualquier tipo de objetos en la lista. El resultado es una nueva lista creada tras evaluar las expresiones que haya dentro.

**Uso**

```
variable = [out_exp for out_exp in input_list if out_exp == 2]
```

Aquí mostramos un ejemplo:

```
multiples = [i for i in range(30) if i % 3 == 0]
print(multiples)
# Salida: [0, 3, 6, 9, 12, 15, 18, 21, 24, 27]
```

Esto puede ser realmente útil para crear listas de manera rápida. De hecho hay gente que las prefiere sobre el uso de la función `filter`. Las comprensiones de listas son la mejor opción si por ejemplo quieres añadir elementos a una lista fruto de un bucle `for`. Si queremos hacer algo como lo siguiente:

```
squared = []
for x in range(10):
    squared.append(x**2)
```

Se podría simplificar en una línea de código con el uso de las comprensiones de listas:

```
squared = [x**2 for x in range(10)]
```

### 3.15.2 `dict` comprehensions

Los diccionarios se usan de una manera muy similar. Aquí vemos un ejemplo:

```
mcase = {'a': 10, 'b': 34, 'A': 7, 'Z': 3}

mcase_frequency = {
    k.lower(): mcase.get(k.lower(), 0) + mcase.get(k.upper(), 0)
    for k in mcase.keys()
}

# mcase_frequency == {'a': 17, 'z': 3, 'b': 34}
```

En el ejemplo anterior combinamos los valores de las llaves o *keys* del diccionario que sean las mismas pero en mayúsculas o minúsculas. Es decir, el contenido de "a" y "A" se juntaría.

Otro ejemplo podría ser invertir las llaves y valores de un diccionario como se muestra a continuación:

```
{v: k for k, v in some_dict.items()}
```

### 3.15.3 `set` comprehensions

Las comprensiones en los sets son muy similares a las listas. La única diferencia es que es necesario hacer uso de llaves `{}` en vez de corchetes.

```
squared = {x**2 for x in [1, 1, 2]}
print(squared)
# Output: {1, 4}
```

### 3.15.4 `generator` comprehensions

Por último, tenemos los generadores. La única diferencia es que no asignan memoria para toda la lista, sino que la asignan elemento a elemento, lo que las hace mas eficientes desde el punto de vista del uso de la memoria.

```
multiples_gen = (i for i in range(30) if i % 3 == 0)
print(multiples_gen)
# Salida: <generator object <genexpr> at 0x7fdaa8e407d8>
for x in multiples_gen:
  print(x)
  # Salida numbers
```

## 3.16 Excepciones

El manejo de excepciones es un arte que una vez que entiendes resulta de lo mas útil. Vamos a ver como pueden ser manejadas en Python.

**Nota**: Si aún no sabes muy bien lo que son las excepciones, te recomendamos empezar por este post y este otro, donde se explican de manera muy sencilla y didáctica.

Empecemos con el uso de `try/except`. El código que puede causar una excepción se pone en el `try` y el código que maneja esa excepción se ubica en el bloque `except`. Veamos un ejemplo:

```
try:
    file = open('test.txt', 'rb')
except IOError as e:
    print('Ocurrió un IOError {}'.format(e.args[-1]))
```

En ejemplo anterior estamos manejando la excepción `IOError`. Otra cosa que veremos a continuación es que en realidad podemos manejar varias excepciones.

### 3.16.1 Manejando múltiples excepciones:

Podemos manejar las excepciones de tres maneras distintas. La primera consiste en poner todas las excepciones que puedan ocurrir separadas por coma, en una tupla. Se muestra a continuación:

```
try:
    file = open('test.txt', 'rb')
except (IOError, EOFError) as e:
    print("Ocurrió un error. {}".format(e.args[-1]))
```

Otra forma es manejar las excepciones de manera individual, creando un bloque `except` para cada una. Veamos un ejemplo:

```
try:
    file = open('test.txt', 'rb')
except EOFError as e:
    print("Ocurrió un EOFError")
except IOError as e:
    print("Ocurrió un IOError")
```

De esta manera, si la excepción no es manejada en el primer bloque, lo será en el segundo o en alguno de los sucesivos. Aunque también puede pasar que no llegue a manejarse en ninguno.

Y por último, el siguiente método permite manejar todas las excepciones con un solo bloque.

```
try:
    file = open('test.txt', 'rb')
except Exception as e:
    # Puedes añadir algún tipo de información extra
    pass
```

Esto puede ser útil cuando no se sabe con certeza que excepciones pueden ser lanzadas por el programa.

#### Uso de `finally`

Ya hemos visto que debemos ubicar el código que pueda causar una excepción en el `try`, y que en el `except` podemos tratar lo que hacer en el caso de que se produzca una excepción determinada. A continuación veremos el uso del `finally`, que permite ejecutar un determinado bloque de código siempre, se haya producido o no una excepción. Se trata de un bloque muy importante, y que suele ser usado para ejecutar alguna tarea de limpieza. Veamos un ejemplo:

```
try:
    file = open('test.txt', 'rb')
except IOError as e:
    print('Ocurrió un IOError. {}'.format(e.args[-1]))
finally:
    print("Se entra aquí siempre, haya o no haya excepción")

# Salida: Ocurrió un IOError. No such file or directory
# Se entra aquí siempre, haya o no haya excepción
```

#### Uso de `try/else`

Puede ser también útil tener una determinada sección de código que sea ejecutada si **no** se ha producido ninguna excepción. Esto se puede realizar con el uso de `else`. Se trata de algo bastante útil porque puede haber determinadas secciones de código que sólo tengan sentido ejecutar si el bloque completo `try` se ha ejecutado correctamente.

Si bien es cierto que no es muy habitual ver su uso, es una herramienta a tener en cuenta.

```
try:
    print('Estoy seguro de que no ocurrirá ninguna excepción')
except Exception:
    print('Excepción')
else:
    # El código de esta sección se ejecutará si no se produce
    # ninguna excepción. Las excepciones producidas aquí
    # tampoco serán capturadas.
    print('Esto se ejecuta si no ocurre ninguna excepción')
finally:
    print('Esto se imprimirá siempre')

# Salida: Estoy seguro de que no ocurrirá ninguna excepción
#         Esto se ejecuta si no ocurre ninguna excepción
#.        Esto se imprimirá siempre
```

El contenido del `else` sólo se ejecutará si no se ha producido ninguna excepción, y será ejecutada antes del `finally`.

## 3.17 Clases

Las clases son el núcleo de Python. Nos dan un montón de poder, pero es muy fácil usarlo de manera incorrecta. En esta sección compartiremos algunos de los trucos relacionados con las clases en Python. ¡Vamos a por ello!

**Nota**: Si aún no entiendes bien la Programación Orientada a Objetos, te recomendamos que empieces antes por este post dónde se explica de manera muy fácil la POO y conceptos relacionados como la herencia y los métodos estáticos y de clase.

### 3.17.1 1. Variables de instancia y clase

La mayoría de principiantes o incluso algunos programadores avanzados de Python, no entienden la diferencia entre instancia y clase. Dicha falta de conocimiento, les fuerza a hacer un uso incorrecto de los mismos. Vamos a explicarlos.

La diferencia es la siguiente:

- Las variables de instancia son usadas para almacenar datos que son únicos para cada objeto.
- Por lo contrario, las variables de clase son compartidas entre diferentes instancias de la clase.

Vamos a ver un ejemplo:

```
class Cal(object):
    # pi es una variable de clase
    pi = 3.142

    def __init__(self, radio):
        # self.radio es una variable de instancia
        self.radio = radio

    def area(self):
        return self.pi * (self.radio ** 2)

a = Cal(32)
a.area()
# Salida: 3217.408
a.pi
# Salida: 3.142
a.pi = 43
a.pi
# Salida: 43

b = Cal(44)
b.area()
# Salida: 6082.912
b.pi
# Salida: 3.142
b.pi = 50
b.pi
# Salida: 50
```

En el ejemplo anterior no hay demasiados problemas al estar usando variables de clase que son inmutables, es decir que no son modificadas. Esta es una de las principales razones por la que ciertos programadores no intentan aprender mas acerca de ellas, ya que no se suelen enfrentar a ningún problema. En el siguiente ejemplo vemos como un mal uso de las variables de clase e instancia pueden causar problemas.

```
class SuperClass(object):
    superpowers = []

    def __init__(self, name):
        self.name = name

    def add_superpower(self, power):
        self.superpowers.append(power)

foo = SuperClass('foo')
bar = SuperClass('bar')
foo.name
# Salida: 'foo'
```

(continues on next page)

(continued from previous page)

```
bar.name
# Salida: 'bar'

foo.add_superpower('fly')
bar.superpowers
# Salida: ['fly']

foo.superpowers
# Salida: ['fly']
```

Esto es un mal uso de las variables de clase. Si te das cuenta la llamada `add_superpower` sobre el objeto `foo` modifica la variable de clase `superpowers`, y dado que es compartida por todos los objetos de la clase, hace que `bar` también cambie. Por lo tanto es importante tener cuidado con esto, y salvo que realmente sepas lo que estás haciendo, no es muy recomendable usar variables de clase mutables.

### 3.17.2 2. Nuevo estilo de clases

Un nuevo estilo de clases fue introducido en Python 2.1, pero mucha gente aún no sabe de ello. Puede ser en parte porque Python sigue manteniendo el antiguo estilo para mantener lo que se llama compatibilidad hacia atrás o *backward compatibility*. Veamos las diferencias:

- En el estilo antiguo, las clases no heredan de nada.
- En el nuevo estilo las clases heredan de `object`.

Un ejemplo muy sencillo podría ser:

```
class OldClass():
    def __init__(self):
        print('I am an old class')

class NewClass(object):
    def __init__(self):
        print('I am a jazzy new class')

old = OldClass()
# Salida: I am an old class

new = NewClass()
# Salida: I am a jazzy new class
```

Esta herencia de `object` permite que las clases pueden utilizar cierta *magia*. Una de las principales ventajas es que puedes hacer uso de diferentes optimizaciones como `__slots__`. También puedes hacer uso de `super()` o de descriptores. ¿Conclusión? Intenta usar el nuevo estilo de clases.

**Nota:** Python 3 solo tiene el estilo nuevo de clases. No importa si heredas de `object` o

no. Sin embargo es recomendable que heredes de `object`, aunque tal vez en la práctica tampoco se hace.

### 3.17.3 3. Métodos mágicos

Las clases en Python son famosas por sus métodos mágicos, comúnmente referidos con **dunder** que viene del Inglés y significa *double underscore*. Es decir, son métodos definidos con doble barra baja, tanto al principio con al final del nombre del mismo. Vamos a explicar algunos de ellos.

- `__init__`

Se trata de un inicializador de clase o también conocido como constructor. Cuando una instancia de una clase es creada, el método `__init__` es llamado. Por ejemplo:

```
class GetTest(object):
    def __init__(self):
        print('Saludos!!')
    def another_method(self):
        print('Soy otro método que no es llamado'
              ' automáticamente')

a = GetTest()
# Salida: Saludos!!

a.another_method()
# Salida: Soy otro método que no es llamado automáticamente
# called
```

Puedes ver como `__init__` es llamado inmediatamente después de que la instancia haya sido creada. También puedes pasar argumentos en la inicialización, como se muestra a continuación.

```
class GetTest(object):
    def __init__(self, name):
        print('Saludos!! {0}'.format(name))
    def another_method(self):
        print('Soy otro método que no es llamado'
              ' automáticamente')

a = GetTest('Pelayo')
# Salida: Saludos!! Pelayo

# Si intentas crear el objeto sin ningún argumento, da error.
b = GetTest()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __init__() takes exactly 2 arguments (1 given)
```

Estoy seguro de que con esto ya entiendes perfectamente el método `__init__`.

- `__getitem__`

Implementar el método `__getitem__` en una clase permite a la instancia usar `[]` para indexar sus elementos. Veamos un ejemplo:

```
class GetTest(object):
    def __init__(self):
        self.info = {
            'name':'Covadonga',
            'country':'Asturias',
            'number':12345812
        }

    def __getitem__(self,i):
        return self.info[i]

foo = GetTest()

foo['name']
# Output: 'Covadonga'

foo['number']
# Output: 12345812
```

Sin implementar el método `__getitem__` tendríamos un error si intentamos hacerlo:

```
>>> foo['name']

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'GetTest' object has no attribute '__getitem__'
```

## 3.18 Funciones Lambda

Las funciones lambda son funciones que se definen en una línea, y son conocidas en otros lenguajes como funciones anónimas. Uno de sus usos es cuando tienes una determinada función que sólo vas a llamar una vez. Por lo demás, su uso y comportamiento es muy similar a las funciones «normales».

**Forma**

```
lambda argumentos: manipular(argumentos)
```

**Ejemplo**

```
suma = lambda x, y: x + y

print(suma(3, 5))
# Salida: 8
```

A continuación mostramos otras formas de usar las funciones lambda:

**Ordenar una lista**

```
a = [(1, 2), (4, 1), (9, 10), (13, -3)]
a.sort(key=lambda x: x[1])

print(a)
# Salida: [(13, -3), (4, 1), (1, 2), (9, 10)]
```

**Ordenar listas paralelamente**

```
datos = zip(lista1, lista2)
datos.sort()
lista1, lista2 = map(lambda t: list(t), zip(*datos))
```

Si quieres saber más acerca de las funciones lambda, puedes encontrar más información en este post post.

## 3.19 Ejemplos en 1 línea

En este capítulo veremos algunos ejemplos en Python que pueden ser escritos en una sola línea de código.

**Servidor Web**

¿Alguna vez has querido enviar un fichero a través de la red? En Python se puede hacer de manera muy fácil de la siguiente forma. Vete al directorio donde tengas el fichero, y escribe el siguiente código.

```
# Python 2
python -m SimpleHTTPServer

# Python 3
python -m http.server
```

**Prints Organizados**

Algo muy común a lo que a veces nos enfrentamos, es tener que imprimir un determinado tipo con `print()`, pero a veces nos encontramos con un contenido que es prácticamente imposible de leer. Supongamos que tenemos un diccionario. A continuación mostramos como imprimirlo de una manera más organizada. Para ello usamos `pprint()` que viene de *pretty* (bonito).

```
from pprint import pprint

my_dict = {'name': 'Pelayo', 'age': 'undefined', 'personality': 'collaciu'}
print(dir(my_dict))
# ['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__
↪', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__
↪', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__init_subclass__
↪', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__', '_
↪reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr_
↪_', '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'append', 'clear
↪', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse',
↪'sort']
```

(continues on next page)

(continued from previous page)

```
pprint(dir(my_dict))
# ['__add__',
#  '__class__',
#  '__contains__',
#  '__delattr__',
#  '__delitem__',
#  '__dir__',
#  '__doc__',
#  '__eq__',
#  '__format__',
#  '__ge__',
#  '__getattribute__',
#  '__getitem__',
#  '__gt__',
#  '__hash__',
#  '__iadd__',
#  '__imul__',
#  '__init__',
#  '__init_subclass__',
#  '__iter__',
#  '__le__',
#  '__len__',
#  '__lt__',
#  '__mul__',
#  '__ne__',
#  '__new__',
#  '__reduce__',
#  '__reduce_ex__',
#  '__repr__',
#  '__reversed__',
#  '__rmul__',
#  '__setattr__',
#  '__setitem__',
#  '__sizeof__',
#  '__str__',
#  '__subclasshook__',
#  'append',
#  'clear',
#  'copy',
#  'count',
#  'extend',
#  'index',
#  'insert',
#  'pop',
#  'remove',
#  'reverse',
#  'sort']
```

Usado en diccionarios anidados, resulta incluso más efectivo. Por otro lado, también

puedes imprimir un fichero *json* con el siguiente comando.

```
cat file.json | python -m json.tool
```

### Profiling de un script

Esto puede ser realmente útil para ver donde se producen los cuellos de botella de nuestro código. Se entiende por hacer *profiling* de un código, al analizar los tiempos de ejecución de sus diferentes partes, para saber dónde se pierde más tiempo y actuar en consecuencia.

```
python -m cProfile mi_script.py
```

Nota: `cProfile` es una implementación más rápida que `profile` ya que está escrito en C.

### Convertir CSV a json

Si ejecutas esto en el terminal, puedes convertir un CSV a json.

```
python -c "import csv,json;print json.dumps(list(csv.reader(open('csv_file.csv
↪')))))"
```

Asegúrate de que cambias `csv_file.csv` por tu fichero.

### Convertir una Lista anidada

Puedes convertir una lista con elemento anidados a una única lista de una dimensión con `itertools.chain.from_iterable` del paquete `itertools`. Veamos un ejemplo:

```
lista = [[1, 2], [3, 4], [5, 6]]
print(list(itertools.chain.from_iterable(lista)))
# Salida: [1, 2, 3, 4, 5, 6]

# Otra forma
print(list(itertools.chain(*lista)))
# Salida: [1, 2, 3, 4, 5, 6]
```

### Construcciones en 1 línea

Otro código bastante interesante y que nos puede ahorrar varias líneas es el siguiente. Tenemos el constructor de una clase con un determinado número de parámetros. En vez de hacer `self.nombre = nombre` uno a uno, podemos reemplazarlo por la siguiente línea.

```
class A(object):
    def __init__(self, a, b, c, d, e, f):
        self.__dict__.update({k: v for k, v in locals().items() if k != 'self'})
```

Si quieres ver más construcciones de una línea, te recomendamos que leas el siguiente enlace.

## 3.20 for/else

Los *loops* o bucles son una parte muy importante de cualquier lenguaje de programación, y por supuesto también existen en Python. Sin embargo, tienen algunas particularidades que mucha gente no conoce. A continuación las explicaremos.

**Nota:** Si buscas una explicación más completa de los bucles `for` en Python te recomendamos este post sobre el uso del for y este otro para el while.

Empecemos con un ejemplo básico de `for`, nada nuevo:

```
frutas = ['manzana', 'plátano', 'mango']
for fruta in frutas:
    print(fruit.capitalize())

# Output: Manzana
#         Plátano
#         Mango
```

Un ejemplo sencillo en el que iteramos una lista que almacena diferentes cadenas con for, y cambiamos su primera letra con una mayúscula. Veamos ahora otras de las funcionalidades que tal vez no sean tan conocidas.

### 3.20.1 Uso del else

Los bucles `for` también tienen una cláusula `else`, y puede ser usada para ejecutar un determinado fragmento de código cuando el bucle termina de manera natural. Por manera natural se entiende que el bucle ha sido ejecutado tantas veces como había sido planeado en su definición, y no termina por la sentencia `break`. Por lo tanto, si un `break` rompe la ejecución del bucle, la cláusula `else` no será ejecutada.

Un ejemplo muy clásico en el uso de bucles, es iterar una determinada lista buscando un elemento concreto. Si el elemento se encuentra, es habitual usar `break` para dejar de buscar, ya que una vez hayamos encontrado lo que buscábamos, no tendría mucho sentido seguir buscando.

Por otro lado, podría ocurrir también que se acabara de iterar la lista y que no se hubiera encontrado nada. En este caso, el bucle terminaría sin pasar por la sentencia `break`. Por lo tanto, una vez sabidos estos dos posibles escenarios, uno podría querer saber cual ha sido la causa por la que el bucle ha terminado, si ha sido porque se ha encontrado el elemento que se buscaba, o si por lo contrario se ha terminado sin encontrar nada.

Veamos un ejemplo de la estructura del `for/else`:

```
for item in container:
    if busca_algo(item):
        # Se ha encontrado
        procesa(item)
```

(continues on next page)

(continued from previous page)

```
        break
else:
    # No se encontró nada
    no_encontrado()
```

Veamos un ejemplo en concreto, tomado de la documentación oficial.

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(n, 'igual', x, '*', n/x)
            break
```

Este ejemplo itera números de 2 a 10, y para cada uno busca un número que divida de manera entera a cada uno de ellos. Si se encuentra, se rompe el primer bucle y se continúa con el siguiente número.

Al ejemplo anterior podemos añadir un bloque `else`, para mostrar determinada información al usuario. Por ejemplo, si el número no es divisible por ninguno de sus antecesores, significará que es un número primo.

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print( n, 'igual', x, '*', n/x)
            break
    else:
        # Si no se llama a break, se entra al else
        print(n, 'es un número primo')
```

## 3.21 Extensiones C de Python

Una característica muy interesante de la que disponemos en Python es tener un interfaz para interactuar con código escrito en el lenguaje de programación C. Existen varios métodos, pero los que cubriremos en este capítulo serán tres: `ctypes`, `SWIG` y `Python/C API`. Veremos las ventajas y desventajas de cada uno con algunos ejemplos de como pueden ser usados.

Pero antes tal vez te preguntes ¿y para qué quiero yo usar código C en Python? Pues bien, existen varias razones:

- Si necesitas velocidad, C es mucho más rápido que Python. En determinadas ocasiones puede ser del orden de decenas de veces más rápido, por lo que si buscamos velocidad, esta puede ser una buena opción.
- Algunas librerías antiguas de C funcionan perfectamente, por lo que podría ser conveniente poder usarlas. Escribirlas otra vez en Python costaría tiempo y dinero.

- Tal vez quieras tener ciertos accesos a recursos muy a bajo nivel.
- O tal vez simplemente porque quieres hacerlo.

Sabido esto, vamos a ver tres formas de usar código C en Python.

### 3.21.1 CTypes

El módulo ctypes de Python es la forma más sencilla de llamar a funciones escritas en C desde Python. Este módulo nos proporciona tipos de datos compatibles con C, y funciones para cargar las librerías DLL de tal forma que puedan ser llamadas. La principal ventaja es que el código escrito en C **no necesita ser modificado**.

**Ejemplo**

Tenemos un código muy sencillo escrito en C que realiza la suma de dos números. Lo guardamos en `suma.c`.

```
//Código C para sumar dos números, enteros y floats.

#include <stdio.h>

int suma_int(int, int);
float suma_float(float, float);

int suma_int(int num1, int num2){
    return num1 + num2;
}

float suma_float(float num1, float num2){
    return num1 + num2;
}
```

Ahora compilamos el código C en un archivo `.so` (DLL para Windows). Esto generará el fichero `adder.so`.

```
#Para Linux
$ gcc -shared -Wl,-soname,adder -o adder.so -fPIC suma.c

#Para macOS
$ gcc -shared -Wl,-install_name,adder.so -o adder.so -fPIC suma.c
```

Ahora en el código Python, podemos hacer lo siguiente

```
from ctypes import *

#Cargamos la librería
adder = CDLL('./adder.so')

#Realizamos la suma entera
res_int = adder.suma_int(4,5)
```

(continues on next page)

(continued from previous page)

```
print "La suma de 4 y 5 es = " + str(res_int)

#Realizamos la suma float
a = c_float(5.5)
b = c_float(4.1)

suma_float = adder.suma_float
suma_float.restype = c_float
print "La suma de 5.5 y 4.1 es = ", str(suma_float(a, b))
```

Y tras ejecutarlo nos encontraríamos con la siguiente salida:

```
La suma de 4 y 5 es = 9
La suma de 5.5 y 4.1 es = 9.60000038147
```

Vamos a explicar el ejemplo paso por paso. Por un lado tenemos el fichero de C, que tampoco necesita explicación. Simplemente tenemos un par de funciones que suman dos valores, la primera enteros y la segunda float.

Por otro lado, en el fichero de Python importamos el módulo `ctypes`. Después importamos la *shared library* que hemos creado usando la función `CDLL`. Una vez hayamos hecho esto, las funciones definidas en la librería de C estarán disponibles en Python a través de la variable `adder`. Por lo tanto, podemos por ejemplo llamar a `suma_int` usando `adder.suma_int()` y pasando dos enteros como entrada. Esto producirá que la función de C sea llamada con esos parámetros que hemos proporcionado y se nos devuelva la salida. Es importante notar que podemos usar por defectos los tipos enteros y cadenas.

Para otros tipos como booleanos o float, tenemos que especificarlo nosotros. Esto es por lo que cuando pasamos los parámetros a la otra función que hemos definido que usaba floats `adder.suma_float()`, tenemos que especificar el tipo con `c_float`. Como se puede ver esta forma es relativamente sencilla de implementar, pero tiene limitaciones. Por ejemplo, no sería posible manipular objetos en C.

### 3.21.2 SWIG

Otra de las formas que existen para usar código C desde Python, es usando SWIG, que viene de *Simplified Wrapper and Interface Generator*. En este método, es necesario crear un nuevo interfaz (un fichero), que es usado como entrada para SWIG.

Es un método no muy conocido ya que en la mayoría de los casos es innecesariamente complejo. Sin embargo, es un método bastante útil cuando tenemos cierto código en C/C++ y queremos usarlo en diferentes lenguajes (no sólo en Python).

**Ejemplo** (De la web de SWIG )

Por un lado tenemos el código en C guardado en `ejemplo.c`, que tiene diferentes funciones y variables.

```
#include <time.h>
double My_variable = 3.0;

int fact(int n) {
    if (n <= 1) return 1;
    else return n*fact(n-1);
}

int my_mod(int x, int y) {
    return (x%y);
}

char *get_time()
{
    time_t ltime;
    time(&ltime);
    return ctime(&ltime);
}
```

Por otro lado tenemos el fichero que actúa de interfaz, y que será el mismo para cualquier lenguaje de programación, por lo que se puede reusar.

```
/* ejemplo.i */
 %module ejemplo
 %{
 /* Pon aquí las cabeceras o las declaraciones como se muestra a continuación */
 extern double My_variable;
 extern int fact(int n);
 extern int my_mod(int x, int y);
 extern char *get_time();
 %}

 extern double My_variable;
 extern int fact(int n);
 extern int my_mod(int x, int y);
 extern char *get_time();
```

Lo compilamos.

```
unix % swig -python ejemplo.i
unix % gcc -c ejemplo.c ejemplo_wrap.c \
        -I/usr/local/include/python2.1
unix % ld -shared ejemplo.o ejemplo_wrap.o -o _ejemplo.so
```

Y ahora en la parte de Python.

```
>>> import ejemplo
>>> ejemplo.fact(5)
120
>>> ejemplo.my_mod(7,3)
```

(continues on next page)

(continued from previous page)

```
1
>>> ejemplo.get_time()
'Sun Feb 11 23:01:07 1996'
>>>
```

Como podemos ver, el resultado que conseguimos con SWIG es el mismo, pero requiere de un poco más des esfuerzo al tener que crear un fichero nuevo. Sin embargo tal vez merezca la pena si queremos compartir código C con más de un lenguaje, ya que este fichero de interfaz que hemos visto sólo necesitaría ser creado una vez.

### 3.21.3 Python/C API

Por último, la CPython API es una de las opciones más usadas, aunque no por su simplicidad. Esto se debe aunque a pesar de ser más compleja que las anteriores vistas, nos permite manipular objetos de Python en C.

Este método requiere que el código C sea escrito de manera específica para poder ser usado desde Python. Todos los objetos de Python se representan como un `PyObject`, y la cabecera `Python.h` nos proporciona diferentes funciones para manipularlos. Por ejemplo, si el `PyObject` es un `PyListType` (es decir, una lista), podemos usar `PyList_Size()` para calcular su longitud. Sería el equivalente a usar `len(list)` en Python. En general, la mayoría de las funciones de Python están disponibles en `Python.h`.

**Ejemplo**

Vamos a ver como escribir una extensión en C, que toma una lista y suma todos sus elementos. Vamos a asumir que todos los elementos son números, como resulta evidente.

Empecemos viendo la forma en la que nos gustaría poder usar la extensión de C que vamos a crear.

```
#A pesar de que parece un import normal, en realidad
#importa la extensión en C que definiremos a continuación.
import sumaLista

l = [1,2,3,4,5]
print "La suma de la lista es " + str(l) + " = " +  str(sumaLista.add(l))
```

El ejemplo anterior podría parecer un fichero normal y corriente de Python, que importa y usa otro módulo llamado `sumaLista`. Sin embargo este módulo no está escrito en Python sino en C. Esta es una de las ventajas principales, ya que en la parte de Python no nos tenemos que preocupar de aprender nada nuevo o usar funciones extra. Se nos abstrae la librería de C como si fuera un módulo Python normal.

Lo siguiente es escribir el código `sumaLista` que será usado como hemos visto antes en Python. Puede parecer un poco complicado pero ya verás como no lo es tanto.

```
//Fichero: adder.c

//Python.h proporciona funciones para manipular los objetos de Python
#include <Python.h>

//Esta es la función que es llamada desde Python
static PyObject* sumaLista_add(PyObject* self, PyObject* args){

  PyObject * listObj;

  //Los argumentos de entrada son proporcionados como una tupla, los parseamos␣
↪para
  //obtener las variables. En este caso sólo se trata de una lista, que será
  //referenciada por listObj.
  if (! PyArg_ParseTuple( args, "O", &listObj))
    return NULL;

  //Devuelve la longitud de la lista
  long length = PyList_Size(listObj);

  //Iteramos todos los elementos
  long i, sum =0;
  for(i = 0; i < length; i++){
    //Tomamos un elemento de la lista (también es un objeto Python)
    PyObject* temp = PyList_GetItem(listObj, i);
    //Sabemos que es un entero, por lo que lo convertimos a long
    long elem = PyInt_AsLong(temp);
    //Realizamos la suma y guardamos el resultado
    sum += elem;
  }

  //Devolvemos a Python otro objeto Python
  //Convertimos el long que teníamos en C a entero en Python
  return Py_BuildValue("i", sum);
}

//Este es un docstring (documentación) de nuestra función suma.
static char sumaLista_docs[] =
"add( ): Suma todos los elementos de la lista\n";

/* Esta tabla relaciona la siguiente información -
  <function-name del módulo Python>, <actual-function>,
  <type-of-args que la función espera>, <docstring asociado a la función>
*/
static PyMethodDef sumaLista_funcs[] = {
  {"add", (PyCFunction)sumaLista_add, METH_VARARGS, sumaLista_docs},
  {NULL, NULL, 0, NULL}
};

/*
```

(continues on next page)

(continued from previous page)

```
  sumaLista es el nombre del módulo, y esto es la inicialización.
  <desired module name>, <the-info-table>, <module's-docstring>
*/
PyMODINIT_FUNC initsumaLista(void){
  Py_InitModule3("sumaLista", sumaLista_funcs,
   "Suma todos los elementos");
}
```

Veamos una explicación paso por paso:

- El fichero `<Python.h>` proporciona todos los tipos que son usados para representar objetos en Python, además de funciones para operar con ellos, como por ejemplo la lista que hemos visto y su función para calcular la longitud.
- A continuación escribimos la función que vamos a llamar desde Python. Por convención, se usa {nombre-módulo}_{nombre-función}. En nuestro caso es `sumaLista_add`.
- Después añadimos a la tabla la información sobre esa función, como el nombre (tanto en C como en Python). En esta tabla hay una entrada por cada función que tengamos, y tiene que ir terminada por lo que se conoce como valor *sentinel*, que es una fila con elementos nulos.
- Finalmente, inicializamos el módulo con `PyMODINIT_FUNC`.

Como podemos ver, la función `sumaLista_add` acepta argumentos de entrara que son del tipo `PyObject` (args es también del tipo tupla, pero como en Python todo es un objeto, usaremos la notación de PyObject). Por otro lado, los argumentos de entrada se parsean usando `PyArg_ParseTuple()`. El primer parámetro es el argumento variable a ser parseado. El segundo es una cadena que nos dice como parsear cada elemento de la tupla. La letra en la posición `n` de la cadena indica el tipo del elemento en la posición ‘‘n’’de la tupla. Por ejemplo, “i” significa entero (*integer*), “s” cadena (*string*) y “0” significa objeto Python.

Por otro lado tenemos la función `PyArg_ParseTuple()` que merece una explicación por separado. Esta función permite almacenar los elementos que se han parseado en variables separadas. Su número de argumentos es igual al número de argumentos que la función espera recibir. Veamos un ejemplo. Si nuestra función recibiera una cadena, un entero y una lista de Python en ese orden, la función se llamaría de la siguiente forma. Una vez llamada, tendríamos en las variables `n`, `s` y `list` los valores ya parseados y listos para ser usados.

```
int n;
char *s;
PyObject* list;
PyArg_ParseTuple(args, "siO", &s, &n, &list);
```

Sin embargo en nuestro ejemplo simplemente extraemos la lista, y almacenamos su contenido en `listObj`. Por otro lado, se puede ver como hacemos uso de la función `PyList_Size()`, lo que nos devuelve la longitud de la lista, el equivalente a `len(list)` que conocemos de Python.

Más adelante, iteramos la lista y tomamos cada elemento con la función `PyList_GetItem(list, index)`. Esto nos devuelve un PyObject*, pero como sabemos que ese elemento es en realidad un entero `PyIntType`, podemos usar la función `PyInt_AsLong(PyObj *)` para obtener el valor. Como se puede ver, realizamos esto para cada elemento calculando la suma.

La suma es por lo tanto convertida a un objeto de Python y devuelta, con ayuda de la función `Py_BuildValue()`. Como podemos ver, usamos "i", lo que indica que queremos convertir un valor que es un entero (*integer*).

Una vez entendido esto, podemos compilar el módulo C. Guarda el siguiente fichero como `setup.py`.

```
#Compila los módulos

from distutils.core import setup, Extension

setup(name='sumaLista', version='1.0',  \
      ext_modules=[Extension('sumaLista', ['adder.c'])])
```

Y ejecuta el siguiente comando en el terminal.

```
python setup.py install
```

Una vez realizado esto, ya podríamos usar el módulo que hemos creado en Python como si de un módulo normal se tratase. Veamos como funciona:

```
#Importamos el módulo que "habla" con C
import sumaLista

l = [1,2,3,4,5]
print "La suma dela lista - " + str(l) + " = " +  str(sumaLista.add(l))
```

Y aquí tenemos la salida.

```
La suma de la lista - [1, 2, 3, 4, 5] = 15
```

Hemos explicado como crear tu primera extensión de C para Python usando la API `Python.h`. Se trata de un método que puede parecer un poco complejo inicialmente, pero una vez te acostumbras a el, puede ser realmente útil.

Existen otras formas de usar código C desde Python, como puede ser usar Cython, pero se trata de un lenguaje un tanto diferente al típico Python, por lo que no lo cubriremos aquí. No obstante te recomendamos que le eches un vistazo.

## 3.22 Función open

La función open simplemente abre un determinado fichero. Puede parecer sencillo pero en gran cantidad de ocasiones es usada de manera incorrecta. Por ejemplo:

```
f = open('foto.jpg', 'r+')
jpgdata = f.read()
f.close()
```

Una de las razones por las que creemos conveniente explicar `open()` es por que es habitual encontrarse el código anterior. Pues bien, hay un total de tres errores (o más bien malas practicas). Al final de este capítulo entenderás porqué. Empecemos por lo básico.

La función `open` devuelve lo que se conoce como *file handle*, y es dado por el sistema operativo a tu aplicación de Python. Una vez has terminado de usar este *file handle* (que te permite acceder al fichero) es importante devolverlo y cerrarlo. Esto se debe en parte a que el sistema operativo tiene un número máximo de ficheros que puede tener abiertos, y no tendría mucho sentido mantener uno abierto si ya no se está usando.

En el código anterior podemos ver como existe la llamada `close()`. La intención de este código es buena, porque se cierra el fichero abierto, pero el problema es que sólo se cerrará si `f.read()` funciona correctamente. Es decir, si existe un error en la función `f.read()`, el programa terminará y el cierre del fichero no se producirá.

Por lo tanto, una de las mejores formas de asegurarnos de que el fichero se cierra correctamente, pase lo que pase, es la siguiente haciendo uso de `with`.

```
with open('foto.jpg', 'r+') as f:
    jpgdata = f.read()
```

El primer argumento de `open` es el nombre del fichero. El segundo es el modo de apertura, que indica cómo se abrirá el fichero:

- `r`: Abre el fichero en modo lectura.
- `r+`: Si quieres leer y escribir en el fichero.
- `w`: Para sobreescribir el contenido.
- `a`: Para añadir al final del fichero en el caso de que ya exista.

Existe algún otro modo de apertura, pero estos son los más comunes. El modo es muy importante ya que cambia el comportamiento, y podríamos llegar a encontrarnos con un error si abrimos un fichero con `w` del que no tenemos permiso de escritura.

Existe un modo más de apertura que merece una mención especial. Se trata del **modo binario** `b`. Es un modo muy usado cuando abrimos ficheros que realmente no tienen contenido legible por los humanos, como podría ser una imagen. Una imagen puede ser vista una vez interpretada y representada por el ordenador, pero si abrimos su contenido con Python, no veremos ninguna información útil. Este tipo de ficheros es común abrirlos en modo binario con `b`.

Por otro lado, un fichero abierto en **modo texto** necesita saber de su *encoding*. Es decir, en que forma está almacenado el texto. Esto es muy importante ya que al final y al cabo, para Python todos los ficheros tienen contenido binario, solo que si lo abrimos en modo texto, lo interpreta de una manera determinada para mostrárnoslo.

Por desgracia, `open()` no soporta especificar el *encoding* en Python 2.x. Sin embargo, la función io.open está disponible tanto en Python 2.x como 3.x y nos lo permite hacer. Puedes pasar el tipo de *encoding* con la palabra `encoding`. Si no pasas ningún argumento, se tomará el *encoding* por defecto. Suele ser una buena práctica indicar un *encoding* específico. El `utf-8` es uno de los más usados y con mayor soporte en navegadores y lenguajes de programación. Por último, de la misma manera que se elige *encoding* para leer, también se puede seleccionar para escribir un fichero.

**Nota**: Si usas `utf-8` no deberías tener ningún problema con las ñ u otras letras como á o ó. Sin embargo con otros *encodings* podrías tenerlos.

Llegados a este punto, tal vez te preguntes ¿y cómo se yo el *enconding* de un fichero?. Bueno, existen varias maneras de hacerlo. Algunos editores de texto como Sublime Text te lo suelen decir. Muchas veces los ficheros vienen con unos metadatos que indican el *encoding* que es usado (como por ejemplo en las cabeceras HTTP).

Una vez sabido esto, vamos a escribir un programa que lee un fichero, y determina si es una imagen JPG o no. (pista: Los ficheros JPG empiezan con la siguiente secuencia de bits `FF D8`).

```
import io

with open('foto.jpg', 'rb') as inf:
    jpgdata = inf.read()

if jpgdata.startswith(b'\xff\xd8'):
    text = u'Es una imagen JPEG (%d bytes long)\n'
else:
    text = u'No es una imagen JPEG (%d bytes long)\n'

#Escribimos también el resultado en un fichero
with io.open('resumen.txt', 'w', encoding='utf-8') as outf:
    outf.write(text % len(jpgdata))
```

Con esto ya hemos visto como abrir ficheros en diferentes modos, asegurándonos de que son cerrados al terminar con ellos con `with open`. Hemos visto también el uso del *encoding* y como podemos usar los metadatos de un fichero para saber si un archivo contiene o no una imagen en JPEG.

Si te quedas con dudas, en estos post puedes leer más acerca de escribir ficheros y leer ficheros en Python.

## 3.23 Usando Python 2+3

En algunas ocasiones puede ser normal querer desarrollar programas que puedan funcionar en Python 2+ y Python 3+. Imagínate por ejemplo que has credo un módulo de Python usado por cientos de personas, pero no todos tienen Python 2 o 3. En este caso podrías tener dos opciones. La primera sería distribuir dos módulos, uno para Python 2 y otro para Python 3. La segunda sería modificar tu código para que funcionara con

ambas versiones.

En esta sección vamos a ver algunos de los trucos que puedes usar para que tus programas sean compatibles con ambas versiones.

**Imports del futuro**

El primer y más importante método es usar los *imports* con `__future__`. Permite importar funcionalidades de Python 3 en Python 2. Veamos un par de ejemplos:

Por ejemplo, los gestores de contexto o *context managers* se introdujeron en Python 2.6+. Para usarlos en Python 2.5 podrías hacer lo siguiente.

```
from __future__ import with_statement
```

Por otro lado, `print` fue cambiado a una función en Python 3. Si quieres usarlo en Python 2, podrías hacer lo siguiente haciendo uso de `__future__`:

```
print
# Salida:

from __future__ import print_function
print(print)
# Salida: <built-in function print>
```

**Gestionando cambios de nombre en los módulos**

Antes de nada, veamos como podemos importar módulos en Python. Es muy común hacerlo de la siguiente manera.

```
import foo
# o también
from foo import bar
```

¿Sabes que otra cosa puedes hacer? Es posible también realizar lo siguiente:

```
import foo as foo
```

La funcionalidad es la misma que la anterior, pero es vital para hacer tu programa compatible con Python 2 y Python 3. Ahora examinemos el siguiente código:

```
try:
    import urllib.request as urllib_request  # Para Python 3
except ImportError:
    import urllib2 as urllib_request  # Para Python 2
```

Lo primero, estamos realizando los *import* dentro de un `try/excep` lee este post si tienes dudas sobre try o except.

Hacemos esto ya que en Python 2 no existe el módulo `urllib.request`, por lo que si intentamos importar tendremos un `ImportError`. La funcionalidad de `urllib.request` es proporcionada por `urllib2` en Python 2. Por lo tanto, si usamos Python 2 intentaremos importar `urllib.request` y como dará un error, importaremos `urllib2`.

También es importante mencionar el uso de la palabra clase as. Es una forma de asignar un nombre al módulo importado, en nuestro caso `urllib_request`. Por lo tanto si realizamos esto, todas las clases y métodos de `urllib2` estarán disponibles con el alias `urllib_request`.

**Funciones obsoletas de Python 2**

Otra cosa muy importante a tener en cuenta es que hay un total de 12 funciones de Python 2 que han sido eliminadas de Python 3. Es importante asegurarse de que no se usan en Python 2, para hacer que el código sea compatible con Python 3. A continuación mostramos una forma que nos permite asegurarnos de que estas 12 funciones no son usadas.

```
from future.builtins.disabled import *
```

Ahora cada vez que usas una de las funciones que han sido eliminadas de Python 3, tendrás un error `NameError` como el que se muestra a continuación.

```
from future.builtins.disabled import *

apply()
# Salida: NameError: obsolete Python 2 builtin apply is disabled
```

**Librerías externas (backports)**

Existen algunos paquetes que proporcionan determinadas funcionalidades de Python 3 en Python 2. Tenemos por ejemplo las siguientes:

- enum `pip install enum34`
- singledispatch `pip install singledispatch`
- pathlib `pip install pathlib`

Para más información, te recomendamos la documentación oficial de Python con los pasos que tienes que seguir para hacer tu código Python compatible entre las versiones 2 y 3.

## 3.24 Corrutinas

Las corrutinas son similares a los generadores pero tienen ciertas diferencias. Las principales son las siguientes:

- Los generadores son productores de datos
- Las corrutinas son consumidores de datos

Antes de nada, vamos a revisar como se creaba un generador. Podemos hacerlo de la siguiente manera:

```
def fib():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a+b
```

A modo de breve recordatorio, el uso de `yield` retorna de la función a donde fue llamada, pero si es vuelta a llamar continúa su ejecución inmediatamente después del `yield`.

Ahora podemos usarlo en un bucle `for` como se muestra a continuación:

```
for i in fib():
    print(i)
```

Es rápido y no consume demasiada memoria ya que genera los valores al vuelo (uno a uno) en vez de almacenarlos todos en una lista. Ahora, si usamos `yield` en el anterior ejemplo, tendremos una corrutina. Las corrutinas consumen los valores que le son enviados. Un ejemplo muy sencillo sería un `grep` en Python:

```
def grep(pattern):
    print("Buscando", pattern)
    while True:
        line = (yield)
        if pattern in line:
            print(line)
```

Pero espera, ¿qué es lo que devuelve `yield`? Bueno, en realidad lo que hemos hecho es convertirlo en una corrutina. No contiene ningún valor inicialmente, sino que proporcionamos esos valores externamente. Los valores son proporcionados usando el método `.send()`. Aquí podemos ver un ejemplo:

```
search = grep('coroutine')
next(search)
# Salida: Buscando coroutine
search.send("I love you")
search.send("Don't you love me?")
search.send("I love coroutines instead!")
# Salida: I love coroutines instead!
```

Por lo tanto cuando enviamos una línea con *.send()*, si cumple con el *pattern* o patrón que hemos indicado al llamar a la función `grep()` será impresa por pantalla.

Los valores enviados son accedidos por `yield`. Tal vez te preguntes sobre el uso de `next()`. Es requerido para empezar la corutina. Al igual que los `generators`, las corutinas no empiezan inmediatamente, sino que se ejecutan en respuesta a los métodos `__next__()` y `.send()`. Por lo tanto tienes que ejecutar `next()` para que la ejecución avance hasta la expresión `yield`.

Por otro lado, podemos cerrar la corrutina llamando al método `.close()` como se muestra a continuación:

```
search = grep('coroutine')
# ...
search.close()
```

Las coroutines van mucho más allá de lo que hemos explicado, por lo que te sugerimos que eches un vistazo a estar increíble presentación http://www.dabeaz.com/coroutines/Coroutines.pdf de David Beazley (en Inglés).

## 3.25 Caching de Funciones

El *caching* de funciones permite almacenar el valor de retorno de una función dependiendo de los argumentos de entrada. Puede ahorrar tiempo cuando una determinada función es llamada con los mismos argumentos de entrada una y otra vez. En versiones anteriores a Python 3.2, teníamos que implementarlo a mano, pero de Python 3.2 en adelante, tenemos un decorador `lry_cache` que permite almacenar y eliminar el caché de retorno de una determinada función.

**Nota**: Si tienes alguna duda sobre el uso de las funciones en Python, te recomendamos este post

Veamos como se puede hacer en diferentes versiones de Python.

### 3.25.1 Python 3.2+

Vamos a implementar una función que calcule la sucesión de Fibonacci usando `lru_cache`.

```
from functools import lru_cache

@lru_cache(maxsize=32)
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)

>>> print([fib(n) for n in range(10)])
# Salida: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

El argumento `maxsize` indica a `lru_cache` el número de valores de retorno a almacenar en caché.

Podemos fácilmente limpiar el caché usando:

```
fib.cache_clear()
```

### 3.25.2 Python 2+

En Python 2+, existen un par de formas de conseguir el mismo efecto. Puedes crear crear tú mismo el mecanismo de caché, que dependerá de tus necesidades. Aquí te mostramos un ejemplo genérico. Se crea un decorador que aplicado a una función hace que al llamarla se busque en `memo` por los argumentos de entrada. Si no se encuentran, llama a la función y los almacena.

```
from functools import wraps

def memoize(function):
    memo = {}
    @wraps(function)
    def wrapper(*args):
        try:
            return memo[args]
        except KeyError:
            rv = function(*args)
            memo[args] = rv
            return rv
    return wrapper

@memoize
def fibonacci(n):
    if n < 2: return n
    return fibonacci(n - 1) + fibonacci(n - 2)

fibonacci(25)
```

**Nota:** memoize no funcionará con tipos mutables (o *unhashable*) (diccionarios, listas, etc...). Sólo funcionará con tipos inmutables, así que ten esto en cuenta.

En el siguiente enlace tienes un artículo de Caktus Group en el que encuentran un fallo en Django debido al uso de `lru_cache`. Es bastante interesante, échale un vistazo.

## 3.26 Gestores de Contexto

Los gestores de contexto o *context managers* permiten asignar o liberar recursos de una forma expresa. El ejemplo más usado es el `with`. Imagínate que tienes dos operaciones relacionadas que te gustaría ejecutar con un determinado código de por medio. Los gestores de contexto te permiten hacer precisamente esto. Veamos un ejemplo:

```
with open('fichero', 'w') as opened_file:
    opened_file.write('Hola!')
```

En el ejemplo anterior se abre el fichero, se escriben unos datos y se cierra automáticamente. Si se produce un error al intentar abrir el fichero o al intentar escribir contenido en el, el fichero se cierra al final. El siguiente código sería el equivalente con manejo de excepciones.

```
file = open('fichero', 'w')
try:
    file.write('Hola!')
finally:
    file.close()
```

Al comparar los ejemplos anteriores podemos ver que gran cantidad de código repetido es eliminado al usar `with`. La principal ventaja del uso de `with` es que se asegura que el fichero se cierra, sin importar lo que hay en el bloque de código.

En general, los usos más comunes de los gestores de contexto son bloquear y liberar recursos, como en el ejemplo que acabamos de ver con un fichero.

Vamos a ver como podemos implementar nuestro propio gestor de contexto. Esto sin duda te permitirá entender que es lo que pasa por debajo.

### 3.26.1 Implementando un Gestor de Contexto I

Todo gestor de contextos tiene que tener al menos unos métodos `__enter__` y un `__exit__` definidos. Vamos a crear nuestro propio gestor de contextos para abrir un fichero:

```
class File(object):
    def __init__(self, file_name, method):
        self.file_obj = open(file_name, method)
    def __enter__(self):
        return self.file_obj
    def __exit__(self, type, value, traceback):
        self.file_obj.close()
```

Una vez definidos los métodos `__enter__` y `__exit__` en nuestra clase ya podemos hacer uso del `with` de la misma forma que vimos anteriormente. Vamos a probarlo:

```
with File('demo.txt', 'w') as opened_file:
    opened_file.write('Hola!')
```

Nuestro método `__exit__` acepta tres argumentos, más adelante veremos porqué.

Pero antes, analicemos lo que pasa por debajo:

1. La sentencia `with` almacena el método `__exit__` de la clase `File`.
2. Llama al método `__enter__` de la clase.
3. El método `__enter__` abre el fichero y lo devuelve.
4. El fichero abierto es pasado a `opened_file`.
5. Escribimos en él usando `.write()`.
6. La sentencia `with` llama al método `__exit__`.
7. Por último el método `__exit__` cierra el fichero.

### 3.26.2 Manejando Excepciones

En el ejemplo anterior no hemos hablado sobre los argumentos `type`, `value` y `traceback` que tenía el método `__exit__`. Entre los pasos 4 y 6 anteriores, si ocurre una excepción, Python pasa estas tres variables al método `__exit__`. Esto es lo que permite a `__exit__` decidir como cerrar el fichero y si realizar algún otro tipo de acción.

¿Que pasaría si tuviéramos una excepción? Por ejemplo, tal vez podríamos estar accediendo a a un método que no existe:

```
with File('demo.txt', 'w') as opened_file:
    # Este método no existe.
    opened_file.undefined_function('Hola!')
```

Veamos ahora todo lo que ocurre cuando `with` se encuentra con una excepción.

1. Se pasa el type, value y traceback del error al método `__exit__`.
2. Se delega en el `__exit__` la gestión de la excepción.
3. Si `__exit__` devuelve `True`, significa que la excepción ha sido manejada correctamente.
4. Si algo diferente a *True* es devuelto, una excepción es lanzada por la sentencia *with*.

En nuestro caso el método `__exit__` devuelve `None` (ya que no hemos especificado ningún valor de retorno). Por lo tanto y como hemos explicado, `with` lanzará la siguiente excepción:

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
AttributeError: 'file' object has no attribute 'undefined_function'
```

Vamos a dar un paso más y manejar la excepción en el método `__exit__`, ademas de devolver `True`:

```
class File(object):
    def __init__(self, file_name, method):
        self.file_obj = open(file_name, method)
    def __enter__(self):
        return self.file_obj
    def __exit__(self, type, value, traceback):
        print("La excepción fue manejada")
        self.file_obj.close()
        return True

with File('demo.txt', 'w') as opened_file:
    opened_file.undefined_function()

# Output: La excepción fue manejada
```

Podemos ver ahora como `__exit__` devuelve *True*, por lo tanto `with` ya no lanza ninguna excepción.

Esta no es la única forma de implementar Gestor de Contexto. Existe otra forma que explicaremos en la siguiente sección.

### 3.26.3 Implementando un Gestor de Contexto II

También podemos implementar un gestor de contexto usando decoradores y generadores. Python viene con un módulo llamado `contextlib` para este propósito. En vez de crear una clase, podemos usar una función genérica. Veamos un ejemplo sencillo, aunque tal vez no muy útil.

```
from contextlib import contextmanager

@contextmanager
def open_file(name):
    f = open(name, 'w')
    try:
        yield f
    finally:
        f.close()
```

La verdad que esta forma de implementar el gestor de contexto parece mucho más fácil e intuitiva. Sin embargo esta forma requiere de algo de conocimiento previo acerca de los generadores, decoradores y la sentencia *yield*. En este ejemplo no hemos capturado ninguna excepción que pueda ocurrir.

Vamos a verlo parte por parte:

1. Python se encuentra con la palabra `yield`, por lo que crea un generador en vez de una función normal.
2. Debido al uso del decorador, `contexmanager` es llamado con la función `open_file` como argumento.
3. El decorador `contextmanager` devuelve el generador envuelto con el objeto `GeneratorContextManager`.
4. El `GeneratorContextManager` es asignado a la función `open_file`. Por lo tanto, cuando llamamos a la función `open_file` estamos en realidad usando un objeto de la clase `GeneratorContextManager`.

Ahora que ya sabemos esto, podemos usar nuestro nuevo gestor de contexto de la siguiente forma:

```
with open_file('some_file') as f:
    f.write('hola!')
```