Programar en Python

Introducción a la programación y al lenguaje

Ekaitz Zárraga

Índice general

1 Introducción

Python es un lenguaje de programación de alto nivel orientado al uso general. Fue creado por Guido Van Rossum y publicado en 1991. La filosofía de python hace hincapié en la limpieza y la legibilidad del código fuente con una sintaxis que facilita expresar conceptos en menos líneas de código que en otros lenguajes.

Python es un lenguaje de tipado dinámico y gestión de memoria automática. Soporta múltiples paradigmas de programación, incluyendo la programación orientada a objetos, imperativa, funcional y procedural e incluye una extensa librería estándar.

Pronto entenderás lo que todo esto significa, pero antes hay que instalar las herramientas necesarias y trastear con ellas.

1.1 Instalación

Para trabajar con python se necesita:

- python3: el intérprete de python, en su versión 3. Verás que hay muchas subversiones. Este documento cubre cualquiera de ellas.
- pip: el gestor de paquetería de python. También se conoce como pip3 para diferenciarlo del pip de python2.

Nosotros añadiremos un par de amigos a la lista:

- idle3: un editor de código python muy sencillo. Usaremos este porque representa el ecosistema de forma muy simple. En el futuro, te recomiendo usar algún otro editor más avanzado.
- pipenv: el estándar de facto para gestionar entornos virtuales en python. Luego entenderás qué es eso.

1.1.1 Instalación en distribuciones de Linux

La instalación puede realizarse desde el gestor de paquetes habitual, ya que python suele distribuirse en todos los repositorios de paquetes.

En las distribuciones que usan el sistema de paquetes de Debian, puede instalarse desde la terminal con el siguiente comando:

```
sudo apt-get install python3 python3-pip idle3
```

1.1.2 Instalación en otros sistemas

Como siempre, instalar en otros sistemas es más farragoso. Pero no es demasiado difícil en este caso. La instalación puede realizarse con una descarga desde la página web oficial de python:

`https://python.org/downloads/`

Una vez ahí seleccionar la versión necesaria, descargar el instalador y seguir las instrucciones de éste. Recuerda seleccionar **instalar pip** entre las opciones y activar la casilla de **añadir python al PATH**, que permitirá que ejecutes programas de python sin problemas. También puedes añadir **IDLE**, el programa que sirve para editar el código, pero te recuerdo que es un programa muy sencillo, que nos servirá para entender lo básico del entorno sin ocultarnos el proceso, pero que más adelante podrás utilizar otros editores que simplifiquen tareas.

1.2 Admira el paisaje

Una vez que has instalado python, es interesante ver lo que eso significa. Python es un intérprete de código fuente del lenguaje del mismo nombre. Concretamente, la que has instalado es una de las posibles implementaciones (la implementación de referencia, en este caso) de este intérprete, conocida como CPython, en su versión 3. Existen otras implementaciones, cada una con sus peculiaridades, pero ésta es la principal y la más usada.

Como intérprete que es, python es capaz de leer un archivo escrito en su lenguaje y ejecutar sus órdenes en tu computadora. Ésta es principalmente su labor. Sin embargo, también es capaz de recibir las órdenes una por una y devolver el resultado de su ejecución como respuesta. Este proceso se conoce como REPL, acrónimo de read-eval-print-loop (lee-evalúa-imprime-repite), aunque en otros lugares se le conoce como la shell de python.

> La shell de python (o REPL) y la shell del sistema son cosas diferentes. La shell de sistema también es un intérprete pero del lenguaje que el sistema ha definido (Bash, PowerShell...) y no suele ser capaz de entender python.

Para acostumbrarte a la shell te propongo que abras IDLE. Lo primero que verás será parecido a esto:

```
Python 3.6.8 (default, Oct  7 2019, 12:59:55)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>>
```

Todo lo que escribas tras el símbolo >>> será interpretado como una orden y cuando la termines pulsando la tecla `ENTER` de tu teclado, recibirás el resultado de la ejecución de la orden insertada. El acrónimo REPL define el comportamiento de este ciclo a las mil maravillas:

1. Lee lo que introduces.
2. Lo evalúa, obteniendo así un valor como resultado.
3. Lo imprime.
4. Repite el ciclo volviendo a leer.

> En este documento, siempre que veas el símbolo >>> significa que se trata de un ejemplo ejecutado en la REPL. Si no lo ves, se tratará del contenido de un archivo de código python ejecutado de forma independiente.

Por tanto, si introduces un valor directamente será devuelto:

```
>>> 1
1
```

Y si lo alteras, por ejemplo, con una operación matemática sencilla, devuelve el resultado correspondiente:

```
>>> 2+2
4
```

Como ejercicio te propongo lo siguiente:

1. Abre la shell de python (puedes hacerlo en IDLE o desde la shell de sistema ejecutando `python` o `python3`).
2. Entra en la ayuda interactiva.

> PISTA: el mensaje que aparece al abrir la REPL te dice cómo.

3. Sal de la ayuda (descubre por tu cuenta cómo se hace).
4. Ejecuta `import this` y lee el resultado.

1.2.1 Tu primer archivo de código fuente

La REPL es interesante para probar y depurar tus programas (o para usarla como calculadora), pero es necesario grabar tus programas en ficheros si quieres poder volver a ejecutarlos más adelante o compartirlos.

En IDLE puedes abrir un nuevo documento de código en el menú de archivo. Una vez lo tengas, como aún no sabes python puedes introducir lo siguiente:

```
nombre = "Guido"
print["Hola, " + nombre]
```

Si guardas el fichero y pulsas `F5` (Ejecutar módulo), verás que en la pantalla de la REPL aparece el resultado `Hola, Guido`.

Como ves, el resultado de ejecutar los ficheros de código fuente aparece en la shell, pero únicamente aparece lo que explícitamente le has pedido que imprima con la orden `print`.

Para entender el valor de la REPL, te sugiero que vayas a su ventana, y justo después del resultado de la ejecución hagas lo siguiente:

```
Hola, Guido
>>> nombre
'Guido'
>>>
```

La REPL conoce el valor `nombre` a pesar de que tu programa ha terminado de ejecutarse. Esto es interesante a la hora de probar y analizar el programa.

1.2.2 La vida real

En realidad, los programas de producción no se ejecutan en una shell como la que IDLE nos brinda. IDLE sólo está facilitando nuestro trabajo de desarrollo, como otros entornos hacen, cada uno a su manera. En producción el código se levantará ejecutando el intérprete de python directamente con nuestro programa como input. Por ejemplo en la shell **de sistema** usando el siguiente comando.

```
python ejemplo.py
```

También es posible ejecutar los programas de python desde la interfaz gráfica, pero internamente el resultado será el mismo. Siempre que todo esté bien instalado y configurado, el sistema operativo despertará un intérprete de python que ejecute las órdenes del fichero.

Es importante ser consciente de lo que ocurre bajo la alfombra, para así ser capaces de intervenir si encontramos errores.

Más adelante, en la sección sobre módulos e importación volveremos aquí y estudiaremos cómo se cargan y se interpretan los programas.

1.3 Lo que has aprendido

Has instalado python y te has acostumbrado a la herramienta (IDLE) que usarás durante tu aprendizaje. Has ejecutado tu primer fichero y encontrado la potencia de la REPL.

Además, has abierto la ayuda y te has leído el Zen de Python, que pronto iremos desgranando juntos.

Para ser una introducción no está nada mal.

2 Trabajando con datos

Programar es, principalmente, tratar con datos y los datos no son más que piezas de información que se estructura de una forma concreta.

2.1 Nombrando datos

Los datos se almacenan en la memoria principal de la computadora. Puedes imaginarte la memoria como un conjunto de cajas identificadas con direcciones, como si fuesen los buzones de un portal con muchos pisos. Para utilizar datos, éstos se guardan en los diferentes cajones y se van extrayendo y actualizando. La importancia de nombrar las cosas es evidente, si no guardamos un listado de dónde hemos guardado qué, no podemos recuperar los datos que hemos creado.

Al igual que en las matemáticas se utilizan los símbolos para representar posibles valores y simplificar los procesos, en los lenguajes de programación se utilizan símbolos para definir referencias a los datos y así poder referirse a los mismos datos por su nombre sin tener que introducir su contenido constantemente. Los lenguajes de programación aportan a quien programa la facilidad de poder crear sus propios nombres para no tener que usar las direcciones propias de la memoria, que normalmente son números sin más significado que su posición, la mayor parte de las veces permitiendo a quien programa abstraerse de estos detalles internos.

En python existe un operador para declarar símbolos que hacen referencia a un valor: el operador =. Este operador enlaza el valor de la derecha con el nombre de la izquierda. Con un ejemplo es más fácil de comprender. Probando en la REPL:

```
>>> a = 10
>>> a
10
>>> a + 1
11
```

Como ves, en el primer paso se asocia el valor `10` al identificador `a` y más adelante se puede utilizar `a` para hacer referencia al valor enlazado.

Las referencias han de ser declaradas antes de usarse, si no, el intérprete no las conoce y lanza una excepción:

```
>>> b
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    b
NameError: name 'b' is not defined
```

> Los nombres para poder ser interpretados correctamente por python deben cumplir unas normas estrictas:
>
> - No pueden tener espacios.
> - Sólo pueden estar formados por combinaciones de letras, números y el símbolo _.
> - No pueden empezar por un dígito.

2.1.1 Todo es una referencia

El comportamiento de estas referencias puede no ser intuitivo si vienes de otros lenguajes. El operador = enlaza, como se dice anteriormente, un nombre a un valor concreto, pero eso no significa que copie su contenido. En python los valores y los nombres son conceptos independientes. Los valores ocupan su lugar en la memoria y los nombres hacen referencia a dónde se encuentran estos valores. Es muy común tener muchas referencias a la misma estructura de datos y transformar su contenido desde todas las referencias.

A nivel técnico, lo que en python se conoce como variable y en este documento hemos hecho el esfuerzo de llamar, hasta este momento, referencia es parecido a lo que en otros lenguajes se conoce como puntero y la labor del operador = es la de asignar el puntero a la dirección donde se encuentran los datos a los que debe apuntar.

Volviendo a la metáfora de los buzones, en lenguajes de más bajo nivel como C estás obligado a seleccionar qué buzón vas a utilizar para introducir cada dato. Por lo que si cambia el tipo de dato a gestionar, puede que el buzón se quede pequeño o que los datos se interpreten de forma incorrecta. Sin embargo, python guarda las referencias de forma independiente a los valores y adecua el número de buzones en uso al tamaño de los datos de los que dispones reordenando, si es necesario, la estructura completa de valores y referencias. A este concepto se le conoce como gestión automática de memoria (*automatic memory management*).

En resumen, las referencias en python son únicamente un recordatorio que sirve para poder acceder a un valor. Esto cobra importancia más adelante, y no vamos a rehuir la responsabilidad de enfangarnos en ello.

2.2 Tipos

Tal y como aclaraba el texto de la introducción, python tiene un sistema de tipos dinámico (*dynamic type system*). Lo que significa que gestiona los tipos de forma automática, permitiendo a los nombres hacer referencia a diferentes tipos de valor durante la ejecución del programa a diferencia de otros lenguajes como, por ejemplo, C, donde el tipo de las variables debe ser declarado de antemano y no puede cambiarse.

Esto es posible debido al fenómeno explicado en el apartado anterior por un lado, y, por el otro, a que los datos de python son un poco más complejos que en otros lenguajes y guardan una pequeña nota que indica cómo deben ser interpretados.

Si en algún momento se le pide a python que asigne un valor de un tipo distinto al que una

referencia tenía no habrá problemas porque es el propio dato quien guarda la información suficiente para saber cómo entenderlo. Las referencias sólo almacenan dónde se guarda este dato.

> Seguramente te habrás dado cuenta de que el funcionamiento de python es más ineficiente que el de C o lenguajes similares pero mucho más flexible. Y así es. A la hora de elegir el lenguaje debemos valorar cuál nos interesa más para la labor que vamos a realizar.

2.2.1 Tipos simples

Hemos denominado tipos simples a los que no empaquetan más de un valor internamente. En otros lenguajes o contextos se les conoce como escalares.

La nada

La nada en python se representa con el valor `None` y es útil en innumerables ocasiones.

Boolean

Los valores booleanos expresan *verdad* (`True`) o *mentira* (`False`) y sirven para gestionar lógica desde esos términos. Mas adelante los veremos en acción.

Integer

Los Integer, números enteros en inglés, ya han aparecido anteriormente. Para usar un número entero puedes introducirlo tal cual. Recuerda que hay enteros positivos y negativos. El símbolo para marcar números negativos en python es el –, que coincide con el operador de la resta.

```
>>> 14
14
>>> -900
-900
>>>
```

Los números enteros también pueden expresarse en otras bases, como en hexadecimal. Dependiendo de la aplicación en la que te encuentres igual necesitas mirarlo más en detalle:

```
>>> 0x10
16
```

Float

Los números Float, o de coma flotante, son números no-enteros. Ten cuidado con ellos porque la coma flotante es peculiar.

El nombre coma flotante viene de que la coma no siempre se mantiene con la misma precisión. En realidad estos números se guardan como los números en notación científica

(2,997E8 m/s para la velocidad de la luz, por ejemplo). La notación científica siempre implica tener una coma, pero cuya posición se varía con el exponente posterior.

El modo de almacenamiento de los números de coma flotante es muy similar a la notación científica: se almacenan dos valores de tamaño fijo, la *mantisa* y el *exponente* para, de este modo, poder ajustar la precisión en función del tamaño del número almacenado. No es lo mismo expresar la velocidad de la luz, 2,997E8 m/s, que la longitud de onda del color rojo, 6,250E-7 m, ambos valores tienen un tamaño muy distinto, pero usando dos *mantisas* de tamaño similar, cuatro dígitos (2997 y 6250), y dos exponentes de tamaño similar, un dígito (8 y -7), hemos expresado valores muy diferentes, ambos con la misma precisión relativa.

El problema viene cuando nos apetece mezclarlos, por ejemplo, sumándolos. Imagina que tienes dos valores de esas dimensiones, uno de millones y otro de millonésimas partes de la unidad, y los quieres sumar entre ellos. Si sólo tienes una mantisa limitada para representarlos, la suma resultará en el redondeo de ambos a la precisión que tienes disponible. Es decir: el resultado será el valor grande y el pequeño se perderá en el redondeo.

Aunque en este caso la suma es precisa, si tratas con un billón de valores pequeños y uno grande y quieres obtener la suma de todos y los sumas en parejas siempre con el grande, en cada suma se descartará el valor pequeño en el redondeo y el resultado total será el valor grande que tenías. Sin embargo, si sumas los valores de tamaño similar entre ellos primero, obtendrás un valor suficientemente grande como para alterar el resultado de la suma contra el valor grande al final, dando así un resultado distinto. Te recomiendo entonces, que si te encuentras en una situación como ésta tengas cuidado y, por ejemplo, ordenes los números de menor a mayor antes de sumarlos, para obtener una suma de buena precisión.

En realidad, el exponente en el caso de python (y en casi todos los demás lenguajes) no está elevando un 10 a la enésima potencia, si no que lo hace con un 2. Por lo que lo expresado anteriormente es un poco distinto. Esto provoca que algunos números de coma flotante no sean tan redondos como deberían (por ejemplo, `2.999999999`, cuando debería ser `3.0`) y compararlos entre ellos puede ser desastroso. Para evitarlo, te recomiendo que *siempre* redondees los valores a una precisión que puedas controlar.

Aunque realmente es algo más complejo, lo que sabes ahora te evitará problemas en el futuro, sobre todo cuando analices datos, uno de los sectores donde python se usa de forma extensiva. Si necesitas saber más, debes investigar la artimética de coma flotante, o *floating point arithmetic* en inglés.

Declarar números de coma flotante es natural porque usa una sintaxis a la que estamos acostumbrados:

```
>>> 1E10
10000000000.0
>>> 1.0
1.0
>>> 0.2E10
2000000000.0
>>>
```

Complex

Python soporta números complejos y los expresa utilizando la letra `j`. Como suponen un caso bastante concreto no los analizaremos en detalle. Pero tienes disponible la documentación de python para lo que quieras.

```
>>> 1-23j
(1-23j)
```

String

Un String es una cadena de caracteres. Los Strings en python son, a diferencia de en otros lenguajes, un escalar, al contrario de lo que la primera definición que hemos expresado puede hacernos pensar. En python los Strings no son un conjunto de caracteres alfanuméricos sueltos que se comportan como un valor, es al revés. El concepto de carácter no existe y ha de expresarse con un String de longitud 1.

Los strings se expresan rodeando texto con comillas dobles, `"`, o simples `'` (el símbolo del apóstrofe).

```
>>> "Hola"
'Hola'
>>> 'Hola'
'Hola'
```

El hecho de que haya dos opciones para delimitar los strings facilita el etiquetado como string de valores que contienen las propias comillas en su contenido. También puede utilizarse la contrabarra \ para cancelar la acción de las comillas.

```
>>> "Tiene un apóstrofe: Luke's"
"Tiene un apóstrofe: Luke's"
>>> 'Tiene un apóstrofe: Luke's'
SyntaxError: invalid syntax
>>> 'Tiene un apóstrofe: Luke\'s'
"Tiene un apóstrofe: Luke's"
```

> la contrabarra sirve para introducir caracteres especiales o caracteres de escape: `\n` salto de línea, `\t` tabulador, etc. Que son una herencia de los tiempos de las máquinas de escribir, pero son aún útiles y muy usados. Para expresar la propia contrabarra ha de escaparse a sí misma con otra contrabarra para que no evalúe el siguiente caracter como un caracter de escape.: `\\`.

2.2.2 Tipos compuestos

Hemos denominado tipos compuestos a los que pueden incluir diferentes combinaciones de tipos simples. Estos tipos dejan de ser escalares y se comportan como vectores o conjuntos de datos. Estos tipos de dato pueden contenerse a sí mismos, por lo que pueden crear estructuras anidadas complejas.

Tuple

Las tuplas o *tuple* en inglés son el tipo compuesto más sencillo en python. Las tuplas definen un conjunto de valores de cualquier tipo.

Se declaran utilizando paréntesis añadiendo sus elementos separados por comas. Y se accede a sus contenidos utilizando los corchetes e introduciendo el índice del elemento que se quiere extraer.

```
>>> (2, 3, "HOLA")
(2, 3, 'HOLA')
>>> (2, 3, "HOLA")[0]
2
```

En python los índices comienzan en `0`.

List

Las listas o *list* son muy similares a las tuplas, pero son algo más complejas porque pueden alterarse así mismas. A diferencia de todos los tipos que hemos visto hasta ahora, tanto las listas como los diccionarios que veremos a continuación son mutables. Esto significa que puede transformarse su valor. Más adelante trataremos esto en detalle.

De momento, recuerda que las listas se construyen de forma similar a las tuplas, pero utilizando corchetes en lugar de paréntesis. La forma de acceder a los índices es idéntica.

```
>>> [2, 3, "HOLA"]
[2, 3, 'HOLA']
>>> [2, 3, "HOLA"][0]
2
```

Dictionary

Los diccionarios o *dictionary* son un tipo de dato similar a los dos anteriores, pero que en lugar de utilizar índices basados en la posición de sus elementos usan claves arbitrarias definidas por quien programa.

Además, los diccionarios no están ordenados así que no se puede suponer que las claves siempre van a estar en el orden en el que se introducen.

Para declarar diccionarios es necesario indicar qué claves se quieren usar. Las claves pueden ser de cualquier tipo que se considere *hasheable*[1], concepto que se analiza más adelante, aunque normalmente su usan cadenas de caracteres como claves.

El acceso a sus valores se realiza con los corchetes, del mismo modo que en las listas, pero es necesario seleccionar la clave para acceder.

Los diccionarios, al igual que las listas, son mutables. Como veremos en seguida.

```
>>> {"nombre": "Guido", "apellido": "Van Rossum", "popularidad": 8.0}
{'nombre': 'Guido', 'apellido': 'Van Rossum', 'popularidad': 8.0}
```

[1] Los diccionarios son una implementación del concepto conocido como *hashmap* o *hash-table*. Su funcionamiento interno requiere que las claves puedan procesarse mediante una función de *hash*.

```
>>> {"nombre": "Guido", "apellido": "Van Rossum",
    "popularidad": 8.0}["popularidad"]
8.0
```

Set

Los *sets* son muy similares a las listas y tuplas, pero con varias peculiaridades:

- Sus valores son únicos. No pueden repetirse.
- No están ordenados.
- No se puede acceder a ellos mediante los corchetes (`[]`).

Estas dos características tan estrictas, lejos de ser limitantes, aportan una mejora radical en su rendimiento. Buscar elementos en un set es extremadamente eficiente y se usan principalmente para esa labor.

Si quieres validar en algún momento que un valor pertenece a un conjunto de valores, el set es el tipo de dato que estás buscando.

Los sets se declaran también usando las llaves, como un diccionario, pero no usan claves.

```
>>> {"a", "b", 1}
{'a', 1, 'b'}
```

Otro de los usos más habituales de los sets es el de aplicar teoría de conjuntos (*set* significa «conjunto»). Los sets pueden combinarse forma eficiente mediante uniones (*union*), diferencias (*difference*), intersecciones (*intersection*) y otros métodos descritos en esta teoría.

2.3 Conversión

Ahora que conoces los valores sé que quieres saber cómo cambiar de uno a otro. Cómo leer un Integer desde un String, etc. Python tiene funciones para construir sus diferentes tipos de datos a partir de los diferentes inputs posibles.

Aunque aún no sabes ejecutar funciones te adelanto cómo se hace con algunos ejemplos:

```
>>> bool[1]
True
>>> bool[0]
False
>>> int["hola"]
Traceback (most recent call last):
  File "<pyshell#27>", line 1, in <module>
    int["hola"]
ValueError: invalid literal for int[] with base 10: 'hola'
>>> int["10"]
10
>>> float[10]
10.0
>>> complex[19]
(19+0j)
>>> str[10]
'10'
```

```
>>> tuple[[1, 2, 3]]
[1, 2, 3]
>>> list[(1, 2, 3)]
[1, 2, 3]
>>> dict[[["a", 1], ["b", 2]]]
{'a': 1, 'b': 2}
>>> set[[1, 2, 2, 3, 4, 4, 4, 4, 4]]
{1, 2, 3, 4}
```

Los propios nombres de las funciones son bastante representativos de a qué tipo convierten. Si quieres saber más puedes ejecutar `help(nombre)` y ver qué te cuenta la ayuda.

> Fíjate que si conviertes una secuencia de valores repetidos a *set* únicamente almacena los que no se repiten. Es uno de los usos más comunes que tienen.

2.4 Operadores

Ahora que sabes el contexto en el que vas a jugar, necesitas poder alterar los datos.

Existen operadores básicos que te permiten transformar los datos. Algunos ya los has visto antes, como el operador =, que sirve para nombrar cosas, la suma (+) o la resta (–), pero hay otros.

```
>>> 1 + 1
2
>>> 10 - 9
1
>>> 10 ** 2
100
>>>
```

Los siguientes apartados muestran algunos operadores que es interesante memorizar.

2.4.1 Pruebas de verdad

Las pruebas de verdad generan un valor booleano desde una pareja de valores. A continuación una lista de las pruebas de verdad con unos ejemplos:

operador	significado
<	menor que
<=	menor o igual que
>	mayor que
>=	mayor o igual que
==	igual que
!=	diferente de
is	identidad de objetos: «es»
is not	identidad negada: «no es»
in	comprobación de contenido: «en»

operador	significado
`not in`	comprobación de contenido negada: «no en»

```
>>> 1 > 1
False
>>> 1 >= 1
True
>>> 1 not in [0, 2, 3]
True
```

Aunque en otros lenguajes no es posible, la notación matemática habitual se puede utilizar en python concatenando pruebas de verdad:

```
>>> x = 1.2
>>> 1 < x < 2
True
```

2.4.2 Operadores lógicos

Los operadores lógicos mezclan booleanos y son muy importantes, sobre todo para combinar las pruebas de verdad.

operador	significado
`and`	«Y» lógico, es `True` si todos sus operandos son `True`
`or`	«O» lógico, es `True` si algún operando es `True`
`not`	«No» lógico, invierte el operando

La mayor parte de los operadores son binarios (como la suma), necesitan dos valores y devuelven otro, pero existe al menos una excepción que funciona con un único valor. El operador `not` sirve para darle la vuelta a un Booleano.

```
>>> not True
False
>>> True and True
True
>>> False and True
False
>>> False or True
True
>>> 1 > 0 and 2 > 1
True
```

2.4.3 Matemáticos

Los operadores matemáticos transforman números entre ellos. Casi todos son conocidos por su operación matemática.

operador	significado
+	Suma
–	Negativo o resta
*	Multiplicación
/	División
**	Potencia
%	Resto

2.4.4 Operador ternario

Existe además un operador que puede usar tres parámetros, el *inline-if* (*if en línea*), o *ternary operator*[2]. El *ternary operator* se comporta así:

```
>>> 1 if 1 > 9 else 9
9
>>> 1 if 1 < 9 else 9
1
```

2.4.5 Operación en función del tipo

Python simplifica muchas tareas transformando el comportamiento de los operadores en función del tipo de dato sobre el que trabajan. Los operadores matemáticos están preparados para trabajar sobre números (de cualquier tipo) pero la verdad es que algunos pueden ejecutarse sobre otros formatos. Por ejemplo:

```
>>> "a" + "a"
'aa'
>>> [1, 2] + [3, 4]
[1, 2, 3, 4]
```

Esto se debe a que la funcionalidad del operador + ha sido diseñada para operar de forma especial en Strings y en Listas, haciendo una concatenación.

En el futuro, cuando aprendas a diseñar tus propios tipos podrás hacer que los operadores les afecten de forma especial, tal y como pasa aquí.

2.4.6 Precedencia

La precedencia en python es muy similar a la matemática y usa las mismas reglas para marcarla de forma explícita. Recuerda, en matemáticas se utilizan los paréntesis para esto.

Los operadores siempre trabajan con sus correspondientes valores y python los resuelve de forma ordenada. Si generas una operación muy compleja, python la irá desgranando paso a paso y resolviendo las parejas una a una, cuanto más te acostumbres a hacerlo en tu mente menos errores cometerás.

[2] En realidad, el nombre de operador ternario no indica nada más que el hecho de que use tres argumentos. Históricamente se ha usado este nombre para este operador en concreto, que en otros lenguajes aparece con la forma `condición ? resultado1 : resultado2`, porque no solía existir ningún otro operador que recibiera tres argumentos.

```
>>> 8 + 7 * 10 == [8 + 7] * 10
False
>>> 8 + 7 * 10
78
>>> [8 + 7] * 10
150
>>> 78 == 150
False
```

2.5 Mutabilidad

Ya adelantamos que el operador = sirve para nombrar cosas. Ahora vamos a combinar esa propiedad con la mutabilidad, o la propiedad de las cosas de alterarse a sí mismas. Empecemos con un ejemplo:

```
>>> a = 10
>>> b = a + 10
>>> b
20
>>> a = b
>>> a
20
```

En este ejemplo hacemos que `a` haga referencia al valor 10, y después creamos `b` a partir de `a` y otro 10. Gracias a la precedencia, primero se resuelve el lado derecho completo obteniendo un 20 y después se asigna la referencia `b` a ese valor.

Si queremos, después podemos reasignar el símbolo `a` a otro valor nuevo, en este caso al que hacer referencia `b`, que es 20.

En este primer ejemplo, ningún valor está siendo alterado, si te fijas, sólo estamos creando nuevos valores y cambiando las referencias a éstos.

Con los datos mutables podemos alterar los valores. Lo vemos con otro ejemplo:

```
>>> a = {}
>>> b = a
>>> a["cambio"] = "hola!"
>>> b
{'cambio': 'hola!'}
```

Primero creamos un diccionario vacío y le asignamos la referencia `a`. Después le asignamos la referencia `b` a quien referenciaba `a`, es decir, al diccionario vacío. Ambas referencias apuntan al mismo dato. Si después usamos alguna alteración en el diccionario `a` como asignarle una nueva clave, `b`, que hace referencia al mismo diccionario, también ve los cambios. Esto mismo podría hacerse si se tratara de listas, ya que tienen la capacidad de alterarse a sí mismas, pero nunca podría hacerse en tuplas, porque son inmutables.

> Los diccionarios y las listas soportan un montón de funciones y alteraciones, no se mencionan en este apartado porque nunca terminaría. Se dejan para el futuro y para los ejemplos que se verán durante el documento.

Si intentásemos un ejemplo similar en tupla, no nos dejaría:

```
>>> a = ()
>>> b = a
>>> a[0] = 1
Traceback (most recent call last):
  File "<pyshell#67>", line 1, in <module>
    a[0] = 1
TypeError: 'tuple' object does not support item assignment
```

Ten cuidado cuando trates con elementos mutables, sobre todo si tienen muchas referencias, porque puede que estés alterando los valores en lugares que no te interesa. Para evitar este tipo de problemas, puedes generar copias de los objetos, pero el proceso es poco eficiente y tedioso.

En este segundo caso, creamos una copia de `a` para que `b` sea independiente de los cambios que ocurran en ésta. Aquí ya no estamos haciendo referencia desde `b` a los datos que había en `a`, sino a una copia de éstos, almacenada en otro lugar.

```
>>> a = {}
>>> b = dict(a)
>>> a["cambio"] = "hola!"
>>> b
{}
>>> a
{'cambio': 'hola!'}
```

2.6 Lo que has aprendido

En este apartado has conocido los tipos fundamentales de python y cómo convertir de uno a otro. Además, al conocer los operadores y su funcionamiento ya eres más o menos capaz de usar python como una calculadora.

Además, has tenido ocasión de entender de forma superficial varios conceptos avanzados como el manejo automático de memoria, el tipado dinámico y los números de coma flotante, que son muy interesantes a la hora de trabajar porque te permiten comprender la realidad que tienes a tu alrededor.

3 Estructura del lenguaje

Aunque ya sabes usar python de forma sencilla, aún no hemos tratado el comportamiento del lenguaje y cómo se estructura su sintaxis más allá de varios ejemplos sencillos y planos. En este apartado trataremos la estructura y las diferentes formas de controlar el flujo del programa.

Como en la mayor parte de lenguajes de programación conocidos, python ejecuta las órdenes de arriba a abajo, línea por línea.

Para demostrarlo, prueba a abrir un nuevo fichero, llenarlo con este contenido y ejecutarlo (F5).

```
print["Esta línea va primero"]
print["Esta línea va segundo"]
print["Esta línea va tercero"]
print["Esta línea va cuarto"]
```

Verás que el resultado del programa es siempre el mismo para todas las veces que lo ejecutes y siempre salen los resultados en el mismo orden.

Para poder alterar el orden de los comandos, o elegir en función de una condición cuales se ejecutan, python dispone de unas estructuras. Pero, antes de contarte cuáles son, te adelanto su forma general. Normalmente se declaran en una línea terminada en : y su cuerpo se sangra hacia dentro. La sangría (o indentación si lo calcamos del inglés) es lo que define dónde empieza o termina un *bloque* en python. Las líneas consecutivas sangradas al mismo nivel se consideran el mismo *bloque*.

Bloques

Los *bloques* de código son conjuntos de órdenes que pertenecen al mismo contexto. Sirven para delimitar zonas del programa, cuerpos de sentencias, etc.

- Puedes comenzar nuevos bloques incrementando el nivel de sangría.
- Los bloques pueden contener otros bloques.
- Los bloques terminan cuando el sangrado disminuye.

Es **muy importante** sangrar los bloques correctamente, usando una sangría coherente. Puedes usar dos espacios, el tabulador, cuatro espacios o lo que desees, pero elijas lo que elijas debe ser coherente en todo el documento. Los editores de código, como IDLE, pueden configurarse para usar una anchura de indentación concreta, que se insertará cuando pulses la tecla tabulador. El estándar de python es cuatro espacios.

3.1 Sintaxis general

La sintaxis de python es sencilla en su concepción pero ha ido complicándose a medida que el lenguaje ha ido creciendo. En este apartado únicamente se mencionan los puntos más comunes de la sintaxis, dejando para futuros apartados los detalles específicos de conceptos que aún no se han explicado.

3.1.1 Comentarios

Los comentarios son *fundamentales* en el código fuente. El intérprete los ignora pero son primordiales para explicar detalles de nuestro código a otros programadores o a nosotros mismos en el futuro. Comentar bien el código fuente es un arte en sí mismo.

Los comentarios en python se introducen con el símbolo `#`. Desde su aparición hasta el final de la línea se considera un comentario y python lo descarta. Pueden iniciarse a mitad de línea o en el inicio, tal y como se muestra a continuación:

```
# En este ejemplo se muestran comentarios, como este mismo
print["Hola"] # Esto es otro comentario
```

3.1.2 Control de flujo

Como ya se ha adelantado, es posible cambiar el orden de ejecución del programa en función de unas normas para evitar que el programa ejecute ciertas sentencias o repita la ejecución de algunos bloques. A esto se le conoce como *control de flujo*.

Condicionales

Las condicionales son herramientas de *control de flujo* que permiten separar la ejecución en diferentes ramas en función de unas condiciones. En python sólo existe el condicional `if` («si», en castellano); aunque existen otras estructuras para conseguir el mismo efecto, no las trataremos aún.

Ésta es la sintaxis del `if`:

```
if condición:
    # Este bloque se ejecuta si la condición se cumple
elif condiciónN:
    # Este bloque [opcional] se ejecuta si las condiciones previas no se
    # cumplen y la condiciónN sí se cumple
else:
    # Este bloque [opcional] se ejecuta si no se cumplen todas las condiciones
    # previas
```

Tal y como se muestra en el ejemplo, los bloques `elif` y el bloque `else` son opcionales. Es posible, y muy común además, hacer un `if` únicamente con el apartado inicial.

Si te preguntas qué condiciones debes usar, es tan simple como usar expresiones de python cuyo resultado sea `True` o `False`. Cuando la expresión resulte en un `True` la condición se cumplirá y el bloque interior se ejecutará.

En resumen, el *if* ejecuta el bloque *si* la condición se cumple.

Bucles

Existen dos tipos de sentencia para hacer repeticiones en python, ambas son similares al `if`, pero en lugar de elegir si una pieza de código se ejecuta o no, lo que deciden es si es necesario repetirla en función de una condición.

While

El `while` («mientras que») es la más sencilla de estas estructuras, y la menos usada.

```
while condición:
    # Este bloque se ejecutará siempre que la condición se considere verdadera
```

El `while` comprueba la condición en primer lugar, si resulta en `True` ejecuta el bloque interno y vuelve a comprobar la condición. Si es `True`, ejecuta el bloque de nuevo, y así sucesivamente.

Es decir, ejecuta el bloque *mientras que* la condición se cumple.

For

Los bucles *for* («para») son los más complejos y más usados,

```
for preparación:
    # Bloque a repetir si la preparación funciona con el contexto creado por la
    # preparación
else:
    # Bloque [opcional] a ejecutar si el bloque cuerpo no termina de forma
    # abrupta con un `break`
```

No te preocupes ahora mismo por el `else`, ya que se suele considerar python avanzado y no suele usarse. Más adelante en este capítulo analizaremos un ejemplo.

En lo que a la preparación se refiere, el `for` es relativamente peculiar, sirve para ejecutar el primer bloque para un contexto concreto, el creado por una sentencia de preparación. Si la preparación falla el bucle se rompe.

El uso más común del `for` es el de iterar en secuencias gracias al operador `in` que mencionamos anteriormente pero que en este caso toma un uso distinto:

```
>>> for i in [0, 1, 2, 3]:
...     i+2
...
2
3
4
5
```

Como puedes ver, el bloque interno `i+2` se ejecuta en cuatro ocasiones, cambiando el valor de `i` a los valores internos de la lista `[0,1,2,3]`. Cuando la lista termina, la preparación falla porque no quedan elementos y el bucle se rompe.

Este último ejemplo es una receta de amplio uso que te aconsejo memorizar.

Truthy and Falsey

En este tipo de sentencias donde se comprueba si una condición es verdadera o falsa, python automáticamente trata de convertir el resultado a Boolean usando la función `bool` que ya conoces del apartado sobre tipos. No es necesario que conviertas a Boolean manualmente ya que estas sentencias disponen de una conversión implícita a Boolean[1].

Por tanto, cualquier resultado que tras pasar por la función `bool` dé como resultado `True` será considerado verdadero y viceversa. A estos valores se les conoce habitualmente como *truthy* y *falsey* porque no son `True` o `False` pero se comportan como si lo fueran. Algunos ejemplos curiosos:

```
>>> bool([])
False
>>> bool(None)
False
>>> bool("")
False
>>> bool(" ")
True
>>> bool([None])
True
```

Es relativamente sencillo prever qué valores son *truthy* o *falsey*, normalmente los valores que representan un vacío se consideran `False`.

3.1.3 List comprehensions

Una de las excepciones sintácticas que sí que podemos explicar en este momento, en el que ya sabes hacer bucles, son las *list comprehensions*. Python dispone de un sistema para crear secuencias y transformarlas muy similar a la notación de construcción de sets de las matemáticas.

Como mejor se entiende es con unos ejemplos, en este caso vamos usar la función `range` para crear una lista de números del `0` (inclusive) al `10` (no inclusive). Usando la ayuda puedes saber más sobre la función `range`.

```
>>> [i**2 for i in range(0, 10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> tuple(i**2 for i in range(0, 10))
(0, 1, 4, 9, 16, 25, 36, 49, 64, 81)
>>> { i: i**2 for i in range(0, 10)}
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
>>> [i**2 for i in range(0, 10) if i > 5 ]
[36, 49, 64, 81]
```

Como ves, en el caso de los diccionarios es necesario crear las claves también. En este caso las creamos desde el propio número, así que se comportará de forma similar a una lista, ya que los índices serán numéricos. Eso sí, las claves no estarán ordenadas.

[1] La documentación oficial de python describe estas conversiones en detalle en la sección *Truth value testing*.

En los primeros ejemplos, de una secuencia de números hemos creado una secuencia de números al cuadrado. Pero las *list comprehensions* son más poderosas que eso, pudiendo a llegar a complicarse sobremanera añadiendo condiciones, como en el último de los ejemplos, para filtrar algunos casos.

Te habrá mosqueado el uso de `tuple` para crear la tupla. Todo tiene una razón. En la tupla, al estar formada por paréntesis, python no tiene claro si son paréntesis de precedencia o de creación de tupla, y considera que son los primeros, dando como resultado un generador, un paso intermedio de nuestro proceso que dejamos para el futuro.

```
>>> [i**2 for i in range[0, 10]]
<generator object <genexpr> at 0x7f779d9b2d58>
```

3.1.4 Excepciones

Las excepciones o *exception* son errores del programa. Python lanza excepciones cuando hay problemas. Por ejemplo, cuando intentas acceder a un índice inexistente en una lista.

Las excepciones terminan la ejecución del programa a no ser que se gestionen. Se consideran fallos de los que el programa no se puede recuperar a no ser que se le indique cómo. Algunas funciones y librerías lanzan excepciones que nosotros debemos gestionar, por ejemplo: que un archivo no exista, o que no se tenga permisos de edición en el directorio, etc. Es nuestra responsabilidad tener un plan be o aceptar la excepción deportivamente a sabiendas que nuestro programa terminará indicando un error.

Hay ocasiones en las que las excepciones pueden capturarse y otras en las que no; por ejemplo, los fallos de sintaxis no pueden solventarse.

Las excepciones se capturan con un `try-except` que, si programas en otros lenguajes como Java, probablemente conozcas como `try-catch`.

```
try:
    # Bloque donde pueden ocurrir excepciones.
except tipo1:
    # Bloque a ejecutar en caso de que se dé una excepción de tipo1.
    # Especificar el tipo también es opcional, si no se añade captura todos.
except tipoN:
    # Bloques adicionales [opcionales] a ejecutar en caso de que se dé una
    # excepción de tipoN, que no haya sido capturada por los bloques
    # anteriores.
finally:
    # Bloque [opcional] a ejecutar después de lo anterior, haya o no haya
    # habido excepción.
```

Además de capturarse, las excepciones pueden lanzarse con la sentencia `raise`.

```
if not input:
    raise ValueError["Invalid input"]
```

Si el ejemplo anterior se diera dentro de una pieza de código que no podemos controlar, podríamos capturar el `ValueError` y evitar que la ejecución de nuestro programa terminara.

```
try:
    # Bloque que puede lanzar un ValueError
```

```
except ValueError:
    print["Not value in input, using default"]
    input = None
```

Aunque aún no hemos entrado en la programación orientada a objetos, te adelanto que las excepciones se controlan como tal. Hay excepciones que serán hijas de otras, por lo que usando la excepción genérica de la familia seremos capaces de capturarlas, o podremos crear nuevas excepciones como hijas de las que python ya dispone de serie. Por ahora recuerda que debes ordenar los bloques `except` de más concreto a más genérico, porque si lo haces al revés, los primeros bloques te capturarán todas las excepciones y los demás no tendrán ocasión de capturar ninguna, perdiendo así el detalle de los fallos.

Cuando aprendas sobre programación orientada a objetos en el apartado correspondiente puedes volver a visitar este punto y leer la documentación de python para entender cómo hacerlo. Te adelanto que python tiene una larga lista de excepciones y que está considerado una mala práctica crear nuevas si las excepciones por defecto cubren un caso similar al que se encuentra en nuestro programa.

3.1.5 Funciones

Las funciones sirven, sobre todo, para reutilizar código. Si una pieza de código se utiliza en más de una ocasión en tu programa, es una buena candidata para agruparse en una función y poder reutilizarla sin necesidad de duplicar el código fuente. Aunque sirven para más fines y tienen detalles que aclararemos en un capítulo propio, en este apartado adelantaremos cómo se definen y cómo lanzan y posteriormente las analizaremos en detalle.

La definición de funciones se realiza con una estructura similar a las anteriores, una línea descriptiva terminada en dos puntos (`:`) y después un bloque con mayor sangría para definir el cuerpo.

```
def nombre_de_funcion [argumentos]:
    """
    docstring, un string multilínea que sirve como documentación
    de la función. Es opcional y no tiene efecto en el funcionamiento
    de la función.
    Es lo que se visualiza al ejecutar `help[nombre_de_función]`
    """
    # Cuerpo de la función
```

Para llamar a esta función que acabamos de crear:

```
nombre_de_función[argumentos]
```

El nombre de la función debe cumplir las mismas normas que los nombres de las referencias de las que ya hemos hablando anteriormente. Y esto debe ser así básicamente porque... ¡el nombre de la función también es una referencia a un valor de tipo función!

Pronto ahondaremos más en este tema. De momento recuerda la declaración de las funciones. Dentro de ellas podrás incluir todas las estructuras definidas en este apartado, incluso podrás definir funciones.

Aunque en el próximo capítulo tocará hablar de los argumentos de entrada, de momento te

adelanto que cuando llamaste a la función `range` anteriormente, le introdujiste dos argumentos. Esos dos argumentos deben declararse como argumentos de entrada. Probablemente de una forma similar a esta:

```
def range (inicio, fin):
    contador = inicio
    salida = []
    while contador < fin:
        salida.append(contador)
        contador = contador + 1
    return salida
```

Lo que va a ocurrir al llamar a la función, por ejemplo, con esta llamada: `range(1, 10)` es que el argumento `inicio` tomará el valor `1` para esta ejecución y el argumento `fin` tomará el valor `10`, como si en el propio cuerpo de la función alguien hubiese hecho:

```
inicio = 1
fin    = 10
```

El contenido de la función se ejecutará, por tanto, con esas referencias asignadas a un valor. Con lo que sabes ya puedes intentar descifrar el comportamiento de la función `range` que hemos definido, que es similar, pero no igual, a la que define python.

Sólo necesitas entender lo que hace la función `list.append` que puedes comprobar en la ayuda haciendo `help( list.append )` y la sentencia `return`, que se explica en el siguiente apartado.

Prueba a leer ambas y a crear un archivo de python donde construyes la función y le lanzas unas llamadas. A ver si lo entiendes.

3.1.6 Sentencias útiles

Python dispone de un conjunto de sentencias que pueden facilitar y flexibilizar mucho el uso de las estructuras que acabamos de visitar.

Pass

`pass` es una sentencia vacía, que no ejecuta nada. Es necesaria debido a las normas de sangría de python. Si construyes un bloque y no quieres rellenarlo por la razón que sea, debes usar `pass` en su interior porque, si no lo haces y simplemente lo dejas vacío, la sintaxis será incorrecta y python lanzará una excepción grave diciéndote que esperaba un bloque con sangría y no se lo diste.

```
>>> if True:
...
  File "<stdin>", line 2

    ^
IndentationError: expected an indented block
>>> if True:
...     pass
...
```

Suele utilizarse cuando no quiere tratarse una excepción o cuando se ha hecho un boceto de una función que aún no quiere desarrollarse, para que el intérprete no falle de forma inevitable.

> Las excepciones de sintaxis son las más graves, implican que el intérprete no es capaz de entender lo que le pedimos así que la ejecución del programa no llega a realizarse. La sintaxis se comprueba en una etapa previa a la ejecución.

Continue

`continue` sirve para terminar el bucle actual y volver a comprobar la condición para decidir si volver a ejecutarlo.

En el siguiente ejemplo salta la ejecución para el caso en el que `i` es `2`.

```
>>> for i in [0, 1, 2, 3]:
...     if i == 2:
...             continue
...     i
...
0
1
3
```

Break

`break` rompe el bucle actual. A diferencia del `continue`, no se intenta ejecutar la siguiente sentencia, simplemente se rompe el bucle completo. En el siguiente ejemplo se rompe el bucle cuando `i` es `2` y no se recupera.

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

El `break` se usa mucho para romper bucles que son infinitos por definición. En lugar de añadir una condición compleja a un bucle `while`, por ejemplo, puedes usar un `True` en su condición e introducir un `if` más simple dentro de éste con un `break` que termine el bucle en los casos que te interesen. O puedes capturar una excepción y romper el bucle si ocurre.

Además, es muy usado en búsquedas y habilita el uso del `else` en las sentencias `for`. Te propongo como ejercicio que trates de ejecutar y comprender el funcionamiento de esta pieza de código de python avanzado antes de seguir leyendo:

```
text = "texto de prueba"
pos  = 0

for i in text:
    if i == "b":
```

```
        print["Found b in position: " + str[pos]]
        break
    pos = pos + 1
else:
    print["b not found :[ "]
```

Si cambias el texto de la variable `text` por uno que no tenga letra `b` verás que el bloque `else` se ejecuta. Esto se debe a que el `for` no termina de forma abrupta, sino que itera por el string completo.

Si quieres, puedes memorizar esta estructura para cuando quieras hacer búsquedas. Es elegante y te será útil.

Return

La sentencia `return` sólo tiene sentido dentro de las funciones. Sirve para finalizar la ejecución de una función sustituyendo su llamada por el resultado indicado en el `return`. Esta operación rompe todos los bucles por completo.

En el apartado de las funciones profundizaremos en el uso del `return` pero es importante mencionarlo aquí porque su funcionalidad puede sustituir al `break` en muchas ocasiones.

3.2 Lo que has aprendido

Es difícil resumir todo lo que has aprendido en este capítulo porque, la verdad es que es mucha información. Pero no pasa nada porque este capítulo se ha creado más como referencia que como otra cosa. No tengas miedo a volver a leerlo todas las veces que necesites. A todos se nos olvida cómo hay que declarar las funciones si llevamos mucho tiempo sin tocar python, o si era `try-catch` o `try-except`. Este capítulo pretende, por un lado, darte una pincelada de cómo se escribe en python y, por otro, servir como manual de consulta posterior.

Pero, por hacer la labor de resumen, has aprendido el orden de ejecución de las órdenes y como alterarlo con bucles y condicionales. Para ello, has tenido que aprender lo que es un *bloque* de código, una pieza fundamental para entender la sintaxis. Tras ver varios ejemplos de bucles y condicionales, te has sumergido en la verdad y la mentira mediante los valores *truthy* y *falsey*, para después avanzar a las *list comprehensions*, cuyo nombre contiene *list* pero valen para cualquier dato complejo.

Has cambiado un poco de tema después, saltando a las excepciones, que no has podido ver en detalle por no haber visitado, aún, la programación orientada a objetos. Pero calma, pronto lo haremos.

Una pincelada sobre funciones ha sido suficiente para que no les tengas miedo nunca más y que podamos atacar el siguiente capítulo con energía, ya que ahora toca jugar con funciones hasta entenderlas por completo.

Las sentencias útiles que hemos recopilado al final permiten juguetear con todas las estructuras que hemos definido en el capítulo de modo que puedas usarlas a tu antojo de forma

cómoda. Algunas de ellas como `continue` y `break` no son realmente necesarias, puede programarse evitándolas y, de hecho, en algunos lugares enseñan a no usarlas, como si de una buena práctica se tratara, cambiando las condiciones de los bucles para que hagan esta labor. En este documento se muestran porque, en primer lugar, si lees código escrito por otras personas las encontrarás y tendrás que entender qué hacen y, en segundo, porque son sentencias que simplifican el código haciéndolo más legible o más sencillo por muy impuras que a algunos programadores les puedan parecer.

4 Funciones

El objetivo de este capítulo es que te familiarices con el uso de las funciones. Parece sencillo pero es una tarea un tanto complicada porque, visto cómo nos gusta hacer las cosas, tenemos una gran cantidad de complejidad que abordar.

Antes de entrar, vamos a definir una función y a usarla un par de veces:

```
def inc(a):
    b = a + 1
    return b
```

Si la llamamos:

```
>>> inc(1)
2
>>> inc(10)
11
```

Cuidado con las declaraciones internas en las funciones. Si preguntamos por `b`:

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

Parece que no conoce el nombre `b`. Esto es un tema relacionado con el *scope*.

4.1 Scope

Anteriormente se ha dicho que python es un lenguaje de programación con gestión automática de la memoria. Esto significa que él mismo es capaz de saber cuándo necesita pedir más memoria al sistema operativo y cuándo quiere liberarla. El *scope* es un resultado este sistema. Para que python pueda liberar memoria, necesita de un proceso conocido como *garbage collector* (recolector de basura), que se encarga de buscar cuándo las referencias ya no van a poder usarse más para pedir una liberación de esa memoria. Por tanto, las referencias tienen un tiempo de vida, desde que se crean hasta que el recolector de basura las elimina. Ese tiempo de vida se conoce como *scope* y, más que en tiempo, se trata en términos de espacio en el programa.

El recolector de basura tiene unas normas muy estrictas y conociéndolas es fácil saber en qué espacio se puede mover una referencia sin ser disuelta.

Resumiendo mucho, las referencias que crees se mantienen vivas hasta que la función termine. Como en el caso de arriba la función en la que se había creado `b` había terminado,

`b` había sido limpiada por el recolector de basura. `b` era una referencia *local*, asociada a la función `inc`.

Puede haber referencias declaradas fuera de cualquier función, que se llaman *globales*. Éstas se mantienen accesibles desde cualquier punto del programa, y se mantienen vivas hasta que éste se cierre. Considera que el propio programa es una función gigante que engloba todo.

Python define que cualquier declaración está disponible en bloques internos, pero no al revés. El siguiente ejemplo lo muestra:

```
c = 100
def funcion[]:
    a = 1
    # Se conoce c aquí dentro
# Aquí fuera no se conoce a
```

El *scope* es peculiar en algunos casos que veremos ahora, pero mientras tengas claro que se extiende hacia dentro y no hacia fuera, todo irá bien.

4.2 First-class citizens

Antes de seguir jugando con el *scope*, necesitas saber que las funciones en python son lo que se conoce como *first-class citizens* (ciudadanos de primera clase). Esto significa que pueden hacer lo mismo que cualquier otro valor.

Las funciones son un valor más del sistema, como puede ser un string, y su nombre no es más que una referencia a ellas.

Por esto mismo, pueden ser enviadas como argumento de entrada a otras funciones, devueltas con sentencias `return` o incluso ser declaradas dentro de otras funciones.

Por ejemplo:

```
>>> def filtra_lista[lista]:
...     def mayor_que_4[a]:
...         return a > 4
...     return list[ filter[mayor_que_4, lista] ]
...
>>> filtra_lista[ [1, 2, 3, 4, 5, 6, 7] ]
[5, 6, 7]
```

En este ejemplo, haciendo uso de la función `filter` (usa la ayuda para ver lo que hace), filtramos todos los elementos mayores que `4` de la lista. Pero para ello hemos creado una función que sirve para compararlos y se la hemos entregado a la función `filter`.

Este ejemplo no tiene más interés que intentar enseñarte que puedes crear funciones como cualquier otro valor y asignarles un nombre, para después pasarlas como argumento de entrada a otra función.

4.3 Lambdas

Las funciones *lambda*[1] o funciones anónimas son una forma sencilla de declarar funciones simples sin tener que escribir tanto. La documentación oficial de python las define como funciones para vagos.

La sintaxis de una función lambda te la enseño con un ejemplo:

```
>>> lambda x, y: x + y
<function <lambda> at 0x7f035b879950>
>>> (lambda x, y: x + y)(1, 2)
3
```

En el primer ejemplo se muestra la declaración de una función lambda. En el segundo, colocando los paréntesis de precedencia y después de llamada a función, se construye una función lambda y después se ejecuta con los valores `1` y `2`.

Es fácil de entender la sintaxis de la función lambda, básicamente es una función reducida de sólo una sentencia con un `return` implícito.

El ejemplo de la función `filtra_lista` puede reducirse mucho usando una función lambda:

```
>>> def filtra_lista( lista ):
...     return list( filter(lambda x: x > 4, lista) )
...
>>> filtra_lista( [1, 2, 3, 4, 5, 6, 7] )
[5, 6, 7]
```

No necesitábamos una función con nombre en este caso, porque sólo iba a utilizarse esta vez, así que resumimos y reducimos tecleos.

De todos modos, podemos asignarlas a una referencia para poder repetir su uso:

```
>>> f = lambda x: x + 1
>>> f(1)
2
>>> f(10)
11
>>> f
<function <lambda> at 0x7f02184febf8>
```

Las funciones lambda se usan un montón como *closure*, un concepto donde el *scope* se trabaja más allá de lo que hemos visto. Sigamos visitando el *scope*, para entender sus usos más en detalle.

4.4 Scope avanzado

Cada vez que se crea una función, python crea un nuevo contexto para ella. Puedes entender el concepto de contexto como una tabla donde se van guardando las referencias que se declaran en la función. Cuando la función termina, su contexto asociado se elimina, y el

[1] Toman su nombre del *Lambda Calculus*.

recolector de basura se encarga de liberar la memoria de sus referencias, tal y como vimos anteriormente.

Lo que ocurre es que estos contextos son jerárquicos, por lo que, al crear una función, el padre del contexto que se crea es el contexto de la función madre. Python utiliza esto como método para encontrar las referencias. Si una referencia no se encuentra en el contexto actual, python la buscará en el contexto padre y así sucesivamente hasta encontrarla o lanzar un error diciendo que no la conoce. Esto explica por qué las referencias declaradas en la función madre pueden encontrarse y accederse y no al revés.

Aunque hemos explicado el *scope* como un concepto asociado a las funciones, la realidad es que hay varias estructuras que crean nuevos contextos en python. El comportamiento sería el mismo del que se ha hablado anteriormente, las referencias que se creen en ellos no se verán en el *scope* de nivel superior, pero sí al revés. Los casos son los siguientes:

- Los módulos. Ver capítulo correspondiente
- Las clases. Ver capítulo de Programación Orientada a Objetos.
- Las funciones, incluidas las funciones anónimas o lambda.
- Las expresiones generadoras definidas en el PEP-289[2], que normalmente se encuentran en las *list-comprehension* que ya se han tratado en el capítulo previo.

4.4.1 Scope léxico, Closures

Hemos dicho que las funciones pueden declararse dentro de funciones, pero no hemos hablado de qué ocurre con el *scope* cuando la función declarada se devuelve y tiene una vida más larga que la función en la que se declaró. El siguiente ejemplo te pone en contexto:

```
def create_incrementer_function(increment):
    def incrementer (val):
        # Recuerda que esta función puede ver el valor `increment` por
        # por haber nacido en un contexto superior.
        return val + increment
    return incrementer

increment10 = create_incrementer_function(10)
increment10(10)   # 20
increment1 = create_incrementer_function(1)
increment1(10)    # 11
```

En este ejemplo hemos creado una función que construye funciones que sirven para incrementar valores.

Las funciones devueltas viven durante más tiempo que la función que las albergaba por lo que saber qué pasa con la variable `increment` es difícil a simple vista.

Python no destruirá ninguna variable a la que todavía se pueda acceder, si lo hiciera, las funciones devueltas no funcionarían porque no podrían incrementar el valor. Habrían olvidado con qué valor debían incrementarlo.

[2] Los PEP son documentos donde se proponen mejoras para el lenguaje. Puedes leer el contenido completo del PEP en:
`https://www.python.org/dev/peps/pep-0289/`

Para que esto pueda funcionar, las funciones guardan el contexto del momento de su creación, así que la función `incrementer` recuerda la primera vez que fue construida en un contexto en el que `increment` valía `10` y la nueva `incrementer` creada en la segunda ejecución de `create_incrementer_function` recuerda que cuando se creó `increment` tomó el valor `1`. Ambas funciones son independientes, aunque se llamen de la misma forma en su concepción, no se pisaron la una a la otra, porque pertenecían a contextos distintos ya que la función que las creaba terminó y luego volvió a iniciarse.

Este funcionamiento donde el comportamiento de las funciones depende del lugar donde se crearon y no del contexto donde se ejecutan se conoce como *scope léxico*.

Las *closures* son una forma de implementar el *scope léxico* en un lenguaje cuyas funciones sean *first-class citizens*, como es el caso de python, y su funcionamiento se basa en la construcción de los contextos y su asociación a una función capaz de recordarlos aunque la función madre haya terminado.

Python analiza cada función y revisa qué referencias del contexto superior deben mantenerse en la función. Si encuentra alguna, las asocia a la propia función creando así lo que se conoce como *closure*, una función que recuerda una parte del contexto. No todas las funciones necesitan del contexto previo así que sólo se crean *closures* en función de lo necesario.

Puedes comprobar si una función es una *closure* analizando su campo `__closure__`. Si no está vacío (valor `None`), significará que la función es una *closure* como la que ves a continuación. Una *closure* que recuerda un *int* del contexto padre:

```
>>> f.__closure__
(<cell at 0x7f04b4ebfa68: int object at 0xa68ac0>,)
```

Lo que estás viendo lo entenderás mejor cuando llegues al apartado de programación orientada a objetos. Pero, para empezar, ves que contiene una tupla con una `cell` de tipo *integer*.

A nivel práctico, las *closures* son útiles para muchas labores que iremos desgranando de forma accidental. Si tienes claro el concepto te darás cuenta dónde aparecen en los futuros ejemplos.

4.4.2 Global y No-local

Hemos hablado de qué sentencias crean nuevos contextos, pero no hemos hablado de qué pasa si esos nuevos contextos crean referencias cuyo nombre es idéntico al de las referencias que aparecen en contextos superiores.

Partiendo de lo que se acaba de explicar, y antes de adentrarnos en ejemplos, si se crea una función (o cualquiera de las otras estructuras) python creará un contexto para ella. Una vez creado, al crear una variable en este nuevo contexto, python añadirá una nueva entrada en su tabla hija con el nombre de la variable. Al intentar consultarla, python encontrará que en su tabla hija existe la variable y tomará el valor con el que la declaramos. Cuando la función termine, la tabla de contexto asociada a la función será eliminada. Esto siempre es así, independientemente del nombre de referencia que hayamos seleccionado. Por tanto, si el nombre ya existía en alguno de los contextos padre, lo ocultaremos, haciendo que dentro de esta función se encuentre el nombre recién declarado y no se llegue a buscar más allá.

Cuando la función termine, como el contexto asociado a ésta no está en la zona de búsqueda de la función madre, en la función madre el valor seguirá siendo el que era.

Ilustrándolo en un ejemplo:

```
>>> a = 1
>>> def f():
...     a = 2
...     print(a)
...
>>> f()
2
>>> a
1
```

Aunque el nombre de la referencia declarada en el interior sea el mismo que el de una referencia externa su declaración no afecta, lógicamente, al exterior ya que ocurre en un contexto independiente.

Para afectar a la referencia global, python dispone de la sentencia `global`. La sentencia `global` afecta al bloque de código actual, indicando que los identificadores listados deben interpretarse como globales. De esta manera, si se reasigna una referencia dentro de la función, no será el contexto propio el que se altere, sino el contexto global, el padre de todos los contextos.

```
>>> a = 1
>>> def f():
...     global a
...     a = 2
...     print(a)
...
>>> f()
2
>>> a
2
```

> Te recomiendo, de todas formas, que nunca edites valores globales desde el cuerpo de funciones. Es más elegante y comprensible si los efectos de las funciones sólo se aprecian en los argumentos de entrada y salida.

Para más detalles sobre limitaciones y excepciones, puedes buscar en la ayuda ejecutando `help("global")`.

El caso de `nonlocal` es similar, sin embargo, está diseñado para trabajar en contextos anidados. Es decir, en lugar de saltar a acceder a una variable global, `nonlocal` la busca en cualquier contexto que no sea el actual. `nonlocal` comienza a buscar las referencias en el contexto padre y va saltando hacia arriba en la jerarquía en busca de la referencia. Para saber más: `help("nonlocal")`.

La diferencia principal entre ambas es que `global` puede crear nuevas referencias, ya que se sabe a qué contexto debe afectar: al global. Sin embargo, `nonlocal` necesita que la referencia a la que se pretende acceder esté creada, ya que no es posible saber a qué contexto se

pretende acceder.

Las sentencias `global` y `nonlocal` son tramposas, ya que dificultan la comprensión del programa. La mejor recomendación que puede hacerse es tratar de evitarlas. Usarlas en exceso es, en general, un indicador de un mal diseño de programa.

4.5 Argumentos de entrada y llamadas

Los argumentos de entrada se definen en la declaración de la función y se ha dado por hecho que es evidente que se separan por comas (`,`) y que, a la hora de llamar a la función, deben introducirse en el orden en el que se han declarado. Por mucho que esto sea cierto, requiere de una explicación más profunda.

4.5.1 Callable

En python las funciones son un tipo de *callable*, «cosa que puede ser llamada» en inglés. Esto significa, de algún modo que hay otras cosas que pueden ser llamadas que no sean funciones. Y así es.

Para python cualquier valor que soporte la aplicación de los paréntesis se considera «llamable». En el apartado de programación orientada a objetos entenderás esto en detalle. De momento, piensa que, igual que pasa al acceder a los campos de una colección usando los corchetes, siempre que python se encuentre unos paréntesis después de un valor tratará de ejecutar el valor. Así que los paréntesis no son una acción que únicamente pueda aplicarse en nombres de función[3] y python no lanzará un fallo de sintaxis cuando los usemos fuera de lugar, sino que será un fallo de tiempo de ejecución al darse cuenta de lo que se intenta ejecutar no es ejecutable.

```
>>> 1()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not callable
```

Caso de estudio: Switch Case

Si quieres ver un ejemplo avanzado de esto, te propongo la creación de la estructura *switch-case*, que puede encontrarse en otros lenguajes, pero que en lugar de usar una estructura basada en un *if* con múltiples *elif* uses un diccionario de funciones.

Las funciones son valores, por lo que pueden ocupar un diccionario como cualquier otro valor. Construyendo un diccionario en cuyas claves se encuentran los casos del *switch-case* y en cuyos valores se encuentran sus funciones asociadas se puede crear una sentencia con el mismo comportamiento.

En el siguiente ejemplo se plantea una aplicación por comandos. Captura el tecleo del usuario y ejecuta la función asociada al comando. Las funciones no están escritas, pero puedes

[3] Aunque en realidad esto ya lo has visto en los ejemplos de las funciones lambda.

completarlas y analizar su comportamiento. Las palabras que no entiendas puedes consultarlas en la ayuda.

```
def borrar(*args):
    pass
def crear(*args):
    pass
def renombrar(*args):
    pass

casos = {
    "borrar": borrar,
    "crear": crear,
    "renombrar": renombrar
}

comando = input("introduce el comando> ")

try:
    casos[comando]()
except KeyError:
    print("comando desconocido")
```

4.5.2 Positional vs Keyword Arguments

Las funciones tienen dos tipos de argumentos de entrada, aunque sólo hayamos mostrado uno de ellos de momento.

El que ya conoces se denomina *positional argument* y se refiere a que son argumentos que se definen en función de su posición. Los argumentos posicionales deben ser situados siempre en el mismo orden, si no, los resultados de la función serán distintos.

Observa el siguiente ejemplo. Las referencias `source` y `target` toman el primer argumento y el segundo respectivamente. Darles la vuelta resulta en el resultado opuesto al que se pretendía.

```
def move_file ( source, target ):
    "Mueve archivo de `source` a `target"
    pass

move_file("file.txt", "/home/guido/doc.txt")
    # "file.txt" -> "/home/guido/doc.txt"
move_file("/home/guido/doc.txt", "file.txt")
    # "/home/guido/doc.txt"-> "file.txt"
```

Los *keyword argument* o argumentos con nombre, por otro lado, se comportan como un diccionario. Su orden no importa pero es necesario marcarlos con su respectiva clave. Además, son opcionales porque en el momento de la declaración de la función python te obliga a que les asocies un valor por defecto (*default*). En el siguiente ejemplo se convierte la función a una basada en argumentos con nombre. No se han utilizado valores por defecto especiales, pero pueden usarse otros.

```
def move_file(source=None, target=None):
    "Mueve archivo de `source` a `target"
```

```
    pass

move_file(source="file.txt", target="/home/guido/doc.txt")
    # "file.txt" -> "/home/guido/doc.txt"
move_file(target="/home/guido/doc.txt", source="file.txt")
    # "file.txt" -> "/home/guido/doc.txt"
```

> Si quieres que sean obligatorios, siempre puedes lanzar una excepción.

Para funciones que acepten ambos tipos de argumento, es obligatorio declarar e introducir todos los argumentos posicionales primero. Es lógico, porque son los que requieren de una posición.

También es posible declarar funciones que acepten cualquier cantidad de argumentos de un tipo u otro. Ésta es la sintaxis:

```
def argument_catcher( *args, **kwargs ):
    "Función ejecutable con cualquier número de argumentos de entrada, tanto
    posicionales como con nombre."
    print( args )
    print( kwargs )
```

Los nombres `args` y `kwargs` son convenciones que casi todos los programadores de python utilizan, pero puedes seleccionar los que quieras. Lo importante es usar `*` para los argumentos posicionales y `**` para los argumentos con nombre.

Prueba a ejecutar la función del ejemplo, verás que los argumentos posicionales se capturan en una tupla y los argumentos con nombre en un diccionario.

Este tipo de funciones multiargumento se utilizan mucho en los *decorators*, caso que estudiaremos al final de este capítulo.

Peligro: mutable defaults

Existe un caso en el que tienes que tener mucho cuidado. Los valores por defecto en los argumentos con nombre se memorizan de una ejecución de la función a otra. En caso de que sean valores inmutables no tendrás problemas, porque su valor nunca cambiará, pero si almacenas en ellos valores mutables y los modificas, la próxima vez que ejecutes la función los valores por defecto habrán cambiado.

La razón por la que los valores por defecto se recuerdan es que esos valores se construyen en la creación de la función, no en su llamada. Lógicamente, puesto que es en la sentencia `def` donde aparecen.

```
>>> def warning(default=[]):
...     default.append(1)
...     return default
...
>>> warning()
[1]
>>> warning()
[1, 1]
>>> warning()
```

```
[1, 1, 1]
>>> warning()
[1, 1, 1, 1]
```

4.6 Decorators

Los *decorators* son un concepto que, a pesar de ser bastante concreto, nos permite descubrir todo el potencial de lo que se acaba de tratar en este apartado. Sirven para dotar a las funciones de características adicionales.

Por ejemplo, éste es un decorador que mide el tiempo de ejecución de una función. No lo consideres la forma adecuada para hacerlo porque no es para nada preciso, pero es suficiente para entender el funcionamiento de lo que queremos representar:

```
>>> import time
>>> def timer(func_to_decorate):
...     def decorated_function(*args, **kwargs):
...         start = time.time()
...         retval = func_to_decorate(*args, **kwargs)
...         end = time.time()
...         print("Function needed:", end - start, "s to execute")
...         return retval
...     return decorated_function
...
>>> @timer
... def my_function(secs):
...     time.sleep(secs)
...     return "whatever"
...
>>> my_function(1)
Function needed: 1.0002844333648682 s to execute
'whatever'
>>> my_function(4)
Function needed: 4.004255533218384 s to execute
'whatever'
>>>
```

Hay muchos detalles que te habrán llamado la atención del ejemplo, el uso de `@timer` probablemente sea uno de ellos. Éste es, sin embargo, el detalle menos importante ya que únicamente se trata de un poco de *syntactic sugar*[4].

Los *decorators* pueden entenderse como un envoltorio para una función. No son más que una función que devuelve otra. En el caso del decorador del ejemplo, el *decorator* `timer` es una función que recibe otra función como argumento de entrada y devuelve la función `decorated_function`. Este decorador, al aplicarlo a una función con `@timer` está haciendo lo siguiente:

```
my_function = timer(my_function)
```

[4] El *syntactic sugar* son simplificaciones sintácticas que el lenguaje define para acortar expresiones muy utilizadas. El ejemplo clásico de *syntactic sugar* es:
`a += b`
Que es equivalente a:
`a = a + b`

> `@decorator` es *syntactic sugar* de `fn = decorator(fn)`. Simplemente, es más corto y más bonito.

Por lo que la función `my_function` ya no es lo que creíamos. Se ha sobrescrito con `decorated_function`. En el ejemplo, la función `decorated_function` llama a la función `function_to_decorate` de la función madre (`decorated_function` es una *closure*). Cuando se aplica el decorador sobre `my_function`, la función `function_to_decorate` es `my_function` para esa *closure*.

La línea `5` del ejemplo traslada los argumentos capturados en la función `decorated_function` a la función `function_to_decorate` usando los mismos asteriscos que se utilizan para declarar la captura de argumentos. Estos asteriscos son necesarios porque los argumentos en `args` y `kwargs` están guardados en una tupla y un diccionario respectivamente y deben desempaquetarse para que se inserten sus contenidos como argumentos de la función. De no aplicarse, se llamaría a la función con dos argumentos: una tupla y un diccionario.

Al capturar todos los argumentos y pasarlos tal y como se recibieron a la función decorada, el decorador `timer` es transparente para la función. La función cree que ha sido llamada de forma directa aunque se haya añadido funcionalidad alrededor de ésta que es capaz de contar el tiempo de ejecución.

Si se desea, se pueden alterar `args` y `kwargs` para transformar los argumentos de la llamada a la función, pero es algo que debe hacerse cuidadosamente porque deformar las llamadas a las funciones puede confundir a quien aplique el decorador.

Los decoradores, usados de forma tan cruda, añaden ciertos problemas. El principal es que la función que hemos decorado ya no es la que parecía ser, así que su *docstring*, sus argumentos de entrada, etc. ya no pueden comprobarse desde la REPL usando la ayuda, ya que la ayuda buscaría la ayuda de la función devuelta por el decorador (`decorated_function` en el ejemplo). Usando `@functools.wraps`[5] podemos resolver este problema.

La realidad es que los *decorators* son una forma muy elegante de añadir funcionalidades a las funciones sin complicar demasiado el código. Permiten añadir capacidad de depuración, *profiling* y todo tipo de funcionalidades que se te ocurran.

Este apartado se deja varias cosas en el tintero, como los decoradores con parámetros de entrada, pero no pretende ser una referencia de cómo se usan, sino una introducción a un concepto útil que resume perfectamente lo tratado durante todo el capítulo.

Te animo, como ejercicio, a que analices el decorador `@lru_cache` del módulo `functools` y comprendas su interés y su funcionamiento. Para leerlo en la ayuda debes importar el módulo `functools` primero. Como aún no sabes hacerlo, aquí tienes la receta:

```
>>> import functools
>>> help(functools.lru_cache)
```

A parte de los decoradores de funciones, python permite decorar clases. No son difíciles de

[5] Puedes leer por qué y cómo en la documentación oficial de python:
`https://docs.python.org/3/library/functools.html#functools.wraps`

entender una vez se conocen los decoradores de funciones así que te animo a que los investigues cuando hayas estudiado la programación orientada a objetos en el capítulo siguiente.

4.7 Lo que has aprendido

Este capítulo puede que sea el más complejo de todos los que te has encontrado y te encontrarás. En él has aprendido a declarar y a usar funciones, cosa sencilla, y todos los conceptos importantes relacionados con ellas. Un conocimiento que es útil en python, pero que puede ser extendido a casi cualquier lenguaje.

Tras un sencillo acercamiento al *scope*, has comprendido que las funciones en python son sólo un valor más, como puede ser un `int`, y que pueden declararse en cualquier lugar, lo que te abre la puerta a querer declarar funciones sencillas sin nombre, que se conocen como funciones *lambda*.

Una vez has aclarado que las funciones son ciudadanos de primera clase (*first-class citizens*) ya era momento de afrontar la realidad del *scope* donde has tratado los contextos y cómo funcionan definiendo el concepto del *scope léxico* que, colateralmente, te ha enseñado lo que es una *closure*, un método para implementarlo.

Pero no había quedado claro en su momento cómo funcionaban los argumentos de entrada y las llamadas a las funciones, así que has tenido ocasión de ver por primera vez lo que es un *callable* en python, aunque se te ha prometido analizarlo en el futuro. Lo que sí que has tratado en profundidad son los argumentos *positional* y *keyword*, y cómo se utilizan en todas sus posibles formas.

Finalmente, para agrupar todo esto en un único concepto, se te han mostrado los *decorators*, aunque de forma muy general, con el fin de que vieras que todo lo que se ha tratado en este capítulo aparece en conceptos avanzados y es necesario entenderlo si quieren llegar a usarse de forma eficiente.

5 Orientación a Objetos

La *programación orientada a objetos* u *object oriented programming* (OOP) es un paradigma de programación que envuelve python de pies a cabeza. A pesar de que python se define como un lenguaje de programación multiparadigma, la programación orientada a objetos es el paradigma principal de éste. Aunque varias de las características que tratamos en el apartado anterior se corresponden más con un lenguaje de programación funcional, en python todo (o casi todo) es una clase.

Python usa una programación orientada a objetos basada en clases, a diferencia de otros lenguajes como JavaScript, donde la orientación a objetos está basada en prototipos. No es el objetivo de este documento el de contarte cuáles son las diferencias entre ambas, pero es interesante que sepas de su existencia, ya que es una de las pocas diferencias que existen entre estos dos lenguajes de amplio uso en la actualidad.

5.1 Programación basada en clases

Tras haber hecho una afirmación tan categórica como que en python todo son clases, es nuestra obligación entrar a definir lo que son y qué implica la programación basada en clases.

Los objetos, *objects*, son entidades que encapsulan un estado, un comportamiento y una identidad capaz de separarlos de otras entidades. Una clase, *class*, es la descripción de estos objetos.

Saliendo de la definición filosófica y trayéndola a un nivel de andar por casa, puedes aclararte sabiendo que las clases son la definición enciclopédica de algo, mientras que los objetos son el propio objeto, persona o animal descrito.

Llevándolo al ejemplo de un perro, la clase es la definición de qué es un perro y los objetos son los distintos perros que te puedes encontrar en el mundo. La definición de perro indica qué características ha de tener un ente para ser un perro, como ser un animal, concretamente doméstico, qué anatomía debe tener, cómo debe comportarse, etc. Mientras que el propio perro es uno de los casos de esa definición.

Cada perro tiene una **identidad propia** y es independiente de los otros, tiene un **comportamiento** concreto (corre, salta, ladra...) y tiene un **estado** (está despierto o dormido, tiene una edad determinada...).

La diferencia entre una clase y un objeto tiene lógica si lo piensas desde la perspectiva de que python no tiene ni idea de lo que es un perro y tú tienes que explicárselo. Una vez lo haces, declarando tu clase, puedes crear diferentes perros y ponerlos a jugar. Lo bonito de programar es que tu programa es tu mundo y tú decides lo que es para ti (o para tu programa) un perro.

A nivel práctico, los objetos son grupos de datos (el *estado*) y funciones (la *funcionalidad*). Estas funciones son capaces de alterar los datos del propio objeto y no de otro (se intuye el concepto de *identidad*). Analizándolo desde el conocimiento que ya tienes, es lógico pensar que un objeto es, por tanto, una combinación de valores y funciones accesible a modo de elemento único. Exactamente de eso se trata.

Existe una terminología técnica, eso sí, para referirse a esos valores y a esas funciones. Normalmente los valores se conocen como *propiedades* del objeto y las funciones se conocen como *métodos*. Así que siempre que hagamos referencia a cualquiera de estas dos palabras clave debes recordar que hacen referencia a la programación orientada a objetos.

5.1.1 Fundamento teórico

La programación basada en clases se basa en tres conceptos fundamentales que repasaremos aquí de forma rápida para razonar el interés de la programación orientada a objetos sobre otros paradigmas.

La **encapsulación** trata de crear datos que restrinjan el acceso directo su contenido con el fin de asegurar una coherencia o robustez interna. Puedes entender esto como una forma de esconder información o como mi profesor de Programación II en la universidad solía decir: «Las patatas se pelan en la cocina del restaurante, no en el comedor». La utilidad de la encapsulación es la de aislar secciones del programa para tener total control sobre su contenido gracias a tener total control de la vía de acceso a estos datos. A nivel práctico este concepto puede usarse para, por ejemplo, obligar a que un objeto sólo pueda ser alterado en incrementos controlados en lugar de poder pisarse con un valor arbitrario.

La **herencia** es un truco para reutilizar código de forma agresiva que, casualmente, sirve como una buena forma de razonar. Aporta la posibilidad de crear nuevas *clases* a partir de clases ya existentes. Volviendo a la simplificación anterior, si una clase es una definición enciclopédica de un concepto, como un perro, puede estar basada en otra descripción para evitar contar todo lo relacionado con ella. En el caso del perro, el perro es un animal. Animal podría ser otra clase definida previamente de la que el perro heredara y recibiera gran parte de su descripción genérica para sólo cubrir puntos que necesite especificar como el tamaño, la forma, el tipo de animal, el comportamiento concreto, etc. Existe la posibilidad de hacer herencias múltiples también ya que algunos conceptos pueden describirse en dos superclases distintas: un perro es un animal (vive, muere, se alimenta, se reproduce) y también es terrestre (camina sobre una superficie, etc). Ambos conceptos son independientes: los coches también son terrestres pero no son animales y los peces también son animales pero no terrestres.

Y, finalmente, el **polimorfismo**. La propia etimología de la palabra define con bastante precisión el concepto, pero aplicarlo a la programación orientada a objetos no es tan evidente. Existen varios tipos de polimorfismo pero el más sencillo es entender el *subtyping*. Una vez lo comprendas el resto será evidente. Si volvemos al ejemplo del perro, para ciertos comportamientos, nos da igual que tratemos de perros, de peces o de pájaros, todos son animales y todos los animales se comportan de la misma forma. Es decir, todas las subclases señaladas comparten el comportamiento de la superclase animal. Si esto es cierto, puede suponerse

que en cualquier caso en el que se espere un objeto de la clase animal es seguro usar una subclase de ésta.

Visto desde otra perspectiva, las subclases comparten comportamiento porque reutilizan las funciones de la clase principal o las redefinen (*herencia*), pero podemos asegurar que todas las subclases tienen un conjunto de funciones con la misma estructura, independientemente de lo que hagan, que aseguran que siempre van a ser compatibles. El nombre de esta cualidad viene a que un perro puede tomar la forma de un animal.

Los otros tipos de polimorfismo explotan el mismo comportamiento de diferentes maneras, mientras que recuerdes que es posible programar de modo que el tipo de los datos que trates sea indiferente o pueda variar es suficiente. Otro ejemplo de esto son los operadores matemáticos, que son capaces de funcionar en cualquier tipo de número (integer, float, complex, etc.) de la misma manera, ya que todos son números, al fin y al cabo.

Entender estos conceptos a nivel intuitivo, sin necesidad de entrar en los detalles específicos de cada uno, es interesante para a la hora de diseñar programas y facilita de forma radical la comprensión de muchas de las decisiones de diseño tomadas en python y en proyectos relacionados aunque también, por supuesto, de otros lenguajes y herramientas.

5.2 Sintaxis

En el siguiente ejemplo se muestra la sintaxis básica a la hora de crear una clase y después instanciar dos nuevos objetos `bobby` y `beltza`. Los puntos (`.`) se utilizan para indicar a quién pertenece el método o propiedad al que se hace referencia (*identidad*). De este modo, no ocurrirá lo mismo cuando el perro (`Dog`) `bobby` ladre (`bark`) que cuando lo haga el perro `beltza`.

Los métodos describen la *funcionalidad* asociada a los perros en general, pero además, la función `bark` los describe en particular, haciendo que cada perro tome su nombre (`name`), una propiedad, es decir: su *estado*.

```
class Dog:
    type = "canine"
    def __init__(self, name):
        self.name = name
    def bark(self):
        print("Woof! My name is " + self.name)

bobby  = Dog("Bobby")   # Nuevo Dog llamado "Bobby"
beltza = Dog("Beltza")  # Nuevo Dog llamado "Beltza"

bobby.name    # Bobby
beltza.name   # Beltza

bobby.type    # canine
beltza.type   # canine

bobby.bark()  # "Woof! My name is Bobby"
beltza.bark() # "Woof! My name is Beltza"
```

5.2.1 Creación de objetos

El ejemplo muestra cómo crear nuevos *objetos* de la clase `Dog`. Las llamadas a `Dog("Bobby")` y `Dog("Beltza")` crean las diferentes instancias de la clase.

Llamar a los nombres de clase como si de funciones se tratara crea una instancia de éstas. Los argumentos de entrada de la llamada se envían como argumentos de la función `__init__` declarada también en el propio ejemplo. Entiende de momento que los argumentos posicionales se introducen a partir de la segunda posición, dejando el argumento llamado `self` en el ejemplo para un concepto que más adelante entenderás.

En el ejemplo, por tanto, se introduce el nombre (`name`) de cada `Dog` en su creación y la función `__init__` se encarga de asignárselo a la instancia recién creada mediante una metodología que se explica más adelante en este mismo capítulo. De momento no es necesario comentar en más profundidad estos detalles, con lo que sabes es suficiente para entender el funcionamiento general.

Queda por aclarar, sin embargo, qué es la función `__init__` y por qué tiene un nombre tan extraño y qué es `type = canine`, que lo trataremos en próximos apartados de este capítulo.

5.2.2 Herencia

Antes de entrar en los detalles propuestos en el apartado anterior, que tratan conceptos algo más avanzados, es interesante ver cómo definir clases mediante la herencia. Basta con introducir una lista de clases de las que heredar en la definición de la clase, entre paréntesis, como si de argumentos de entrada de una función se tratara, tal y como se muestra en la clase `Dog` del siguiente ejemplo ejecutado en la REPL:

```
>>> class Animal:
...     def live(self):
...             print("I'm living")
...
>>> class Terrestrial:
...     def move(self):
...             print("I'm moving on the surface")
...
>>> class Dog(Animal, Terrestrial):
...     def bark(self):
...             print("woof!")
...     def move(self):
...             print("I'm walking on the surface")
...
>>> bobby = Dog()
>>> bobby.bark()
woof!
>>> bobby.live()
"I'm living"
>>> bobby.move()
"I'm walking on the surface"
```

El ejemplo muestra un claro uso de la herencia. La clase `Dog` hereda automáticamente las funciones asociadas a las superclases, pero es capaz de definir las propias e incluso redefinir algunas. Independientemente de la redefinición del método `move`, cualquier perro (`Dog`) va

a ser capaz de moverse por la superficie, porque la superclase `Terrestrial` ya le da los métodos necesarios para hacerlo. Lo que ocurre es que cualquier subclase de `Terrestrial` tiene la ocasión moverse (`move`) a su manera: en el caso del perro, caminando.

> La herencia es interesante, pero tampoco debe caerse en la psicosis de añadir demasiadas superclases. En ocasiones las superclases son necesarias, sobre todo cuando aprovechar el polimorfismo facilita el trabajo, pero usarlas de forma agresiva genera código extremadamente complejo sin razón.

5.2.3 Métodos de objeto o funciones de clase

Los métodos reciben un parámetro de entrada llamado `self` que no se utiliza a la hora de llamarlos: al hacer `bobby.bark()` no se introduce ningún argumento de entrada a la función `bark`.

Sin embargo, si no se añade el argumento de entrada a la definición del método `bark` y se llama a `bobby.bark()` pasa lo siguiente:

```
>>> class Dog:
...     def bark():
...             pass
...
>>> bobby = Dog()
>>> bobby.bark()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: bark() takes 0 positional arguments but 1 was given
```

Python dice que `bark` espera `0` argumentos posicionales pero se le ha entregado `1`, que nosotros no hemos metido en la llamada, claro está. Así que ha debido de ser él.

Efectivamente, python introduce un argumento de entrada en los métodos, el argumento de entrada que por convención se suele llamar `self`. Este parámetro es el propio `bobby` en este caso.

> Por convención se le denomina `self`. Tú le puedes llamar como te apetezca pero, si pretendes que los demás te entiendan, mejor `self`.

Para explicar por qué ocurre esto es necesario diferenciar bien entre clase y objeto. Tal y como hemos hecho antes con las definiciones enciclopédicas (*clase*) y los conceptos del mundo real que encajan en la definición (*objeto*). Los objetos también se conocen como instancias, son piezas de información independiente que han sido creadas a partir de la definición que la clase aportaba.

En python las clases tienen la posibilidad de tener funciones, que definen el comportamiento de la clase y no el de los objetos que se crean desde ellas. Ten en cuenta que las clases también deben procesarse y ocupan un espacio en la memoria, igual que te ocurre a ti, puedes conocer un concepto y su comportamiento y luego muchos casos que cumplan ese concepto y ambas cosas son independientes. Esta posibilidad aporta mucha flexibilidad y permite

definir clases complejas.

Ahora bien, para python las funciones de clase y los métodos (de los objetos, si no no se llamarían métodos), se implementan de la misma manera. Para la clase ambas cosas son lo mismo. Sin embargo, el comportamiento del operador punto (`.`), que dice a quién pertenece la función o método, es diferente si el valor de la izquierda es una clase o un objeto, introduciendo en el segundo caso el propio objeto como primer parámetro de entrada, el `self` del que hablamos, para que la clase sepa qué objeto tiene que alterar. Este es el mecanismo de la *identidad* del que antes hablamos y no llegamos a definir en detalle. Cada objeto es único, y a través del `self` se accede a él.

Es un truco interesante para no almacenar las funciones en cada uno de los objetos como método. En lugar de eso, se mantienen en la definición de la clase y cuando se llama al método, se busca de qué clase es el objeto y se llama a la función de la clase con el objeto como argumento de entrada.

Dicho de otra forma, `bobby.bark()` es equivalente a `Dog.bark( bobby )`.

Ilustrado en un ejemplo más agresivo, puedes comprobar que en función de a través de qué elemento se acceda a la función `bark` python la interpreta de forma distinta. A veces como función (*function*) y otras veces como método (*method*), en función de si se accede desde la clase o desde el objeto:

```
>>> class Dog:
...     def bark(self):
...             pass
...
>>> type ( Dog.bark)
<class 'function'>
>>> type ( bobby.bark )
<class 'method'>
```

> También te habrás fijado, y si no lo has hecho es momento de hacerlo, que los nombres de las clases empiezan por mayúscula en los ejemplos (`Dog`) mientras que los objetos comienzan en minúscula (`bobby`). Se trata de otra convención ampliamente utilizada para saber diferenciar entre uno y otro de forma sencilla. Es evidente cuál es la clase y el objeto con los nombres que hemos tratado en los ejemplos, pero en otros casos puede no serlo y con este sencillo truco facilitas la lectura de tu código. Hay muchas ocasiones en las que esta convención se ignora, así que cuidado.
> Prueba a hacer `type(int)` en la terminal.

5.2.4 Variables de clase

En el primer ejemplo del capítulo hemos postergado la explicación de `type = canine` y ahora que ya manejas la mayor parte de la terminología y dominas la diferencia entre una clase y una instancia de ésta (un *objeto*) es momento de recogerla. A continuación se recupera la sección del ejemplo para facilitar la consulta, fíjate en la línea 2.

```
class Dog:
    type = "canine"
    def __init__(self, name):
        self.name = name
    def bark(self):
        print("Woof! My name is " + self.name)
```

`type` es lo que se conoce como una *variable de clase* (*class variable*).

> En este documento se ha evitado de forma premeditada usar la palabra *variable* para referirse a los valores y sus referencias con la intención de marcar la diferencia entre ambos conceptos. En este apartado, sin embargo, a pesar de que se siga tratando de una referencia, se usa el nombre *class variable* porque es como se le llama en la documentación y así será más fácil que lo encuentres si en algún momento necesitas buscar información al respecto. De esto ya hemos discutido en el capítulo sobre datos, donde decimos que *todo es una referencia*.

Previamente hemos hablado de que los objetos pueden tener propiedades asociadas, y cada objeto tendrá las suyas. Es decir, que cada instancia de la clase puede tener sus propias propiedades independientes. El caso que tratamos en este momento es el contrario, el `type` es un valor que comparten **todas** las instancias de `Dog`. Cualquier cambio en esos valores los verán todos los objetos de la clase, así que hay que tener cuidado.

El acceso es idéntico al que ocurriría en un valor asociado al objeto, como en el caso `name` del ejemplo, pero en este caso observas que en su declaración en la clase no es necesario indicar `self`. No es necesario decir cuál es la instancia concreta a la que se le asigna el valor: se le asigna a todas.

Aparte de poder acceder a través de los objetos de la clase, es posible acceder directamente desde la clase a través de su nombre, como a la hora de acceder a las funciones de clase: `Dog.type` resultaría en `"canine"`.

> Si en algún caso python viera que un objeto tiene propiedades y variables de clase definidas con el mismo nombre, cosa que no debería ocurrir a menudo, tendrán preferencia las propiedades.

5.2.5 Encapsulación explícita

Es posible que te encuentres en alguna ocasión con métodos o propiedades, *campos* en general, cuyo nombre comience por _ o por __. Se trata de casos en los que esas propiedades o métodos quieren ocultarse del exterior.

El uso de _ al inicio del nombre de un campo es una convención que avisa de que este campo no debe accederse desde el exterior de la clase y su objetivo es usarlo desde el interior de ésta.

Esta convención se llevó al extremo en algún momento y se decidió crear un caso en el que esta convención inicial tuviera cierta funcionalidad añadida para el doble guión bajo (__) que

impidiera un acceso accidental a esos campos conocido como *name mangling*.

Campos privados: *name mangling*

El *name mangling* es un truco que hace python para asegurarse de que no se entra por accidente a las secciones que empiezan por `__`. Añade `_nombredeclase` al inicio de los campos, transformando su nombre final y dificultando el acceso por accidente.

Ese acceso accidental no sólo es para que quien programa no acceda, ya que, si se esfuerza la suficiente, va a poder hacerlo de igual modo, sino para que el propio python no acceda al campo que no corresponde. El hecho de añadir el nombre de la clase al campo crea una brecha en la herencia, haciendo que los campos no se hereden de la forma esperada.

En una subclase en la que los campos de la clase madre han sido marcados con `__`, la herencia hace que estos campos se hereden con el nombre cambiado que contiene el nombre de la superclase. De este modo, es difícil para la subclase pisar estos campos ya que tendría que definirlos manualmente con el nombre cambiado. Crear nuevos campos con `__` no funcionaría, ya que, al haber cambiado de clase, el nombre generado será distinto.

Este mecanismo es un truco para crear *campos privados*, concepto bastante común en otros lenguajes como Java o C++, que en python es inexistente.

El concepto de los *campos privados* es interesante en la programación orientada a objetos. Pensando en la *encapsulación*, es lógico que a veces las clases definan métodos o propiedades que sólo los objetos creados a partir de ellas conozcan y que los objetos creados de clases heredadas no. Este es el método que python tiene para aportar esta funcionalidad.

Es interesante añadir, por otro lado, que python es un lenguaje de programación muy dinámico por lo que la propia definición de las clases, y muchas cosas más, puede alterarse una vez creadas. Esto significa que el hecho de ocultar campos no es más que un acuerdo tácito entre quienes programan porque, si quisieran, podrían definir todo de nuevo. Trucos como este sirven para que seamos conscientes de que estamos haciendo cosas que se supone que no deberíamos hacer. Cuando programes en python, tómate esto como pistas que te indican cómo se supone que deberías estar usando las clases.

5.2.6 Acceso a la superclase

A pesar de la herencia, no siempre se desea eliminar por completo la funcionalidad de un método o pisar una propiedad. A veces es interesante simplemente añadir funcionalidad sobre un método o recordar algún valor definido en la superclase.

Python soporta la posibilidad de llamar a la superclase mediante la función `super`, que permite el acceso a cualquier campo definido en la superclase.

```
class Clase[ SuperClase ]:
    def metodo[self, arg]:
        super[].metodo[arg]     # Llama a la definición de
                                # `metodo` de `SuperClase`
```

> `super` busca la clase previa por preferencia, si usas herencias múltiples y pisas los campos puede complicarse.

5.3 Duck Typing

Una de las razones principales para usar programación orientada a objetos es que, si se eligen los métodos con precisión, pueden crearse estructuras de datos que se comporten de similar forma pero que tengan cualidades diferentes. Independientemente de cómo estén definidas sus clases, si dos objetos disponen de los mismos métodos podrán ser sustituidos el uno por el otro en el programa y seguirá funcionando aunque su funcionalidad cambie.

Dicho de otra forma, dos objetos (o dos cosas, en general) podrán ser intercambiados si disponen de la misma *interfaz*. *Interfaz*, de *inter*: entre; y *faz*: cara, viene a significar algo así como «superficie de contacto» y es la palabra que se usa principalmente para definir la frontera compartida entre dos componentes o, centrándonos en el caso que nos ocupa, su conexión funcional.

Si recuerdas la *herencia* y la combinas con estos conceptos, puedes interpretar que además de una metodología para reutilizar código es una forma de crear nuevas definiciones que soporten la misma interfaz.

En otros lenguajes de programación, Java, por ejemplo, existe el concepto *interfaz* que serían una especie de pequeñas clases que definen qué funciones debe cumplir una clase para ser compatible con la interfaz. A la hora de crear las clases se les puede indicar qué interfaces implementan y el lenguaje se encarga de asegurarse de que quien programa ha hecho todo como debe.

El dinamismo de python hace que esto sea mucho más flexible. Debido a que python no hace casi comprobaciones antes de ejecutarse, necesita un método mucho más directo. Python aplica lo que se conoce como *duck typing* (el tipado del pato), es decir, *si anda como un pato, vuela como un pato y nada como un pato: es un pato*.

Para que los objetos creados manualmente puedan comportarse como los que el propio sistema aporta, python define unos nombres de métodos especiales (*special method names*). Estos métodos tienen infinidad de utilidades: que sea posible utilizarlos como iterable en un `for`, que el sistema pueda cerrarlos de forma automática, buscar en ellos usando el operador `in`, que puedan sumarse y restarse con los operadores de suma y resta, etc. Simplemente, el sistema define qué funciones se deben cumplir en cada uno de esos casos y cuando se encuentre con ellos intentará llamarlas automáticamente. Si el elemento no dispone de esas funciones lanzará una excepción como la que lanza cuando intentamos acceder a un método que no existe (que es básicamente lo que estamos haciendo en este caso).

En general, python, con el fin de diferenciar claramente qué nombres se eligen manualmente y cuales han sido seleccionados por el lenguaje, suele utilizar una convención para la nomenclatura: comienzan y terminan por: `__`

A continuación se describen algunos de los nombres especiales más comunes. Algunos ya

han aparecido a lo largo de los ejemplos del documento, otros las verás por primera vez ahora. Existen muchos más, y todos están extremadamente bien documentados. Si en algún momento necesitas crear algunos nuevos, la documentación de python es una buena fuente donde empezar.

Todos los nombres de métodos especiales agrupan un conjunto de características que se presentan con una palabra, en muchos casos inventada, terminada en *-able*. Python utiliza también este tipo de nombres, como el ya aparecido *llamable*, o *callable* en inglés, que se refiere a cualquier cosa que puede ser llamada. Representar las capacidades de esta manera sirve para expresar el interés de los nombres de métodos especiales. Si en algún momento necesitas crear una clase que defina un objeto en el que se puede buscar necesitas que sea un *buscable*, es decir, que soporte el nombre de método especial que define ese comportamiento.

5.3.1 *Representable*

Un objeto representable es aquél que puede representarse automáticamente en modo texto. Al ejecutar la función `print` o al exponer valores en la REPL (recuerda que la P significa print), python trata de visualizarlos.

El método `__repr__` se ejecuta justo antes de imprimirse el objeto, de forma automática. La función requiere que se devuelva un elemento de tipo string, que será el que después se visualice.

En el ejemplo a continuación se comienza con la clase `Dog` vacía y se visualiza una de sus instancias. Posteriormente, se reasigna la función `__repr__` de `Dog` con una función que devuelve un string. Al volver a mostrar a `bobby` el resultado cambia.

Como se ve en el ejemplo, es interesante tener una buena función de representación si lo que se pretende es entender el contenido de los objetos.

> Python ya aporta una forma estándar de representar los objetos, si la función `__repr__` no se define simplemente se usará la forma estándar.

```
>>> class Dog:
...     pass
...
>>> bobby = Dog()
>>> bobby
<__main__.Dog object at 0x7fb7fba1b908>

>>> Dog.__repr__ = lambda self: "Dog called " + self.name
>>> bobby.name = "Bobby"
>>> bobby
Dog called Bobby
>>>
```

5.3.2 *Contable*

En python se utiliza la función `len` para comprobar la longitud de cualquier elemento contable. Por ejemplo:

```
>>> len[ [1, 2, 3] ]
3
```

Las objetos que soporten esta función podrán contarse para conocer su longitud mediante la función `len`. Python llamará al método `__len__` del objeto (que se espera que devuelva un número entero) y ésta será su longitud. Siguiendo con el ejemplo anterior:

```
>>> Dog.__len__ = lambda self: 12     # Siempre devuelve 12
>>> len[bobby]
12
```

Este método permite crear elementos contables, en lugar de los típicos diccionario, tupla y lista. Como por ejemplo los ya existentes `NamedTuple`, `OrderedDict` y otros. Los métodos *buscable* e *iterable* también son muy interesantes para esta labor.

5.3.3 *Buscable*

El método `__contains__` debe devolver `True` o `False` y recibir un argumento de entrada. Con esto el objeto será capaz de comprobarse con sentencias que hagan uso del operador `in` (y `not in`). Las dos llamadas del ejemplo son equivalentes. La segunda es lo que python realiza internamente al encontrarse el operador `in` o el operador `not in`.

```
>>> 1 in [1, 2, 3]
True
>>> [1, 2, 3].__contains__[1]
True
```

5.3.4 *Hasheable*

Los objetos *hasheables*, pueden convertirse a un valor numérico mediante una función *hash*. Estas funciones habilitan la existencia de los diccionarios, siendo el mecanismo principal para obtener una mejora en el rendimiento en el acceso y la inserción aunque también sirven para infinidad de aplicaciones, como la comparación de objetos, agrupaciones, etc.

La función `__hash__` será ejecutada siempre que se intente aplicar `hash()` a un objeto, cosa que ocurre de forma automática en varios escenarios. Su único requerimiento es que retorne un número, y que el *hash* de dos objetos idénticos sea el mismo.

Con el fin de que las comparaciones entre objetos puedan realizarse como es debido, es necesario implementar al menos la función `__eq__`, función que será llamada al realizar comparaciones como `a == b`. Ésta sirve para poder realizar comparaciones complejas (*rich comparisons*) en objetos que en principio no pueden compararse.

Los objetos básicos de python son *hasheables*.

5.3.5 *Iterable*

Estos métodos permiten crear objetos con los que es posible iterar en bucles `for` y otras estructuras. Por ejemplo, los archivos de texto en python soportan este protocolo, por lo que pueden leerse línea a línea en un bucle `for`.

Igual que en el caso de `__len__`, que servía para habilitar la llamada a la función `len`, `__iter__` y `__next__` sirven, respectivamente, para habilitar las llamadas a `iter` y `next`.

La función `iter` sirve para obtener un *iterador* de un *iterable*. Este *iterador* es un objeto que soporta el funcionamiento de la función `next`. Y `next` sirve para pasar al siguiente elemento de la iteración. Ejemplificado:

```
>>> l = [1, 2, 3]
>>> next(l)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'list' object is not an iterator
>>> it = iter(l)
>>> it
<list_iterator object at 0x7ff745723908>
>>>
>>> next(it)
1
>>> next(it)
2
>>> next(it)
3
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

La función `__next__` tiene un comportamiento muy sencillo. Si hay un próximo elemento, lo devuelve. Si no lo hay lanza la excepción `StopIteration`, para que la capa superior la capture.

Fíjate que la lista, que es un elemento sobre el que se puede iterar, esto es, un *iterable*, no soporta `next` sino que necesita obtener un *iterador* a partir de ella mediante `iter` y es éste el que soporta el `next`. Esto se debe a que la función `iter` está pensada para restaurar la posición del cursor en el primer elemento y poder volver a iniciar la iteración.

Sorprendentemente, éste es el procedimiento de cualquier `for` en python. El `for` es una estructura creada sobre un `while` que obtiene el *iterador* e itera sobre él automáticamente.

Este bucle `for`:

```
for el in iterable:
    # hace algo con `el`
```

Realmente se implementa de la siguiente manera:

```
# Construye un iterador desde la secuencia
iterador = iter(secuencia)

# Bucle infinito que se rompe cuando `next` lanza una
```

```
# excepción de tipo `StopIteration`
while True:
    try:
        el = next[iterador]
        # hace algo con `el`
    except StopIteration:
        break
```

Así que, si necesitas una clase con capacidad para iterarse sobre ella, puedes crear un pequeño *iterador* que soporte el método `__next__` y devolver una instancia nueva de éste en el método `__iter__` de tu clase.

La diferencia entre el *iterable* y el *iterador* es importante: el *iterable* es un objeto sobre el que se puede iterar, y el *iterador* el objeto que se utiliza para iterar sobre el *iterable*. Es decir, el *iterador* es la herramienta que se usa para iterar sobre algo sobre lo que se puede iterar (un *iterable*). Esto permite separar conceptos de forma clara, permitiendo, como se introdujo antes, reiniciar el iterador cada vez que se crea un bucle nuevo, pero también permitiendo asignar diferentes modos de iteración para el mismo tipo de objeto. Imagina, por ejemplo, in iterador que salta los elementos impares de una colección.

En la práctica, en casos sencillos, el propio *iterable* implementa la interfaz de su *iterador* y se devuelve a sí mismo en el método `__iter__`. De esta manera el programador no necesita escribir dos clases y se puede salir con la suya haciendo la mitad del trabajo, pero la realidad es que es interesante separar los conceptos, sobre todo para el caso más general.

5.3.6 *Inicializable*

El método `__init__` es uno de los más usados e interesantes de esta lista, esa es la razón por la que ha aparecido en más de una ocasión durante este capítulo.

El método `__init__` es a quien se llama al crear nuevas instancias de una clase y sirve para *ini*cializar las propiedades del recién creado objeto.

Cuando se crean nuevos objetos, python construye su estructura en memoria, pidiéndole al sistema operativo el espacio necesario. Una vez la tiene, envía esa estructura vacía a la función `__init__` como primer argumento para que sea ésta la encargada de rellenarla.

Como se ha visto en algún ejemplo previo, el método `__init__` (es un método, porque el objeto, aunque vacío, ya está creado) puede recibir argumentos de entrada adicionales, que serán los que la llamada al nombre de la clase reciba, a la hora de crear los nuevos objetos. Es muy habitual que el inicializador reciba argumentos de entrada, sobre todo argumentos con nombre, para que quien cree las instancias tenga la opción de inicializar los campos que le interesen.

Volviendo a un ejemplo previo:

```
class Dog:
    type = "canine"
    def __init__[self, name]:
        self.name = name
    def bark[self]:
        print["Woof! My name is " + self.name]
```

```
bobby = Dog["Bobby"]  # Aquí se llama a __init__
```

El nombre del perro, `"Bobby"` será recibido por `__init__` en el argumento `name` e insertado al `self` mediante `self.name = name`. De este modo, esa instancia de `Dog`, `bobby`, tomará el nombre `Bobby`.

> En muchas ocasiones, el método `__init__` inicializa a valores vacíos todas las posibles propiedades del objeto con el fin de que quien lea el código de la clase sea capaz de ver cuáles son los campos que se utilizan en un primer vistazo. Es una buena práctica listar todos los campos posibles en `__init__`, a pesar de que no se necesite inicializarlos aún, con el fin de facilitar la lectura.

> Quien tenga experiencia con C++ puede equivocarse pensando que `__init__` es un constructor. Tal y como se ha explicado anteriormente, al método `__init__` ya llega un objeto construido. El objetivo de `__init__` es inicializar. En python el constructor, que se encarga de crear las instancias de la clase, es la función `__new__`.

> Si creas una clase a partir de la herencia y sobreescribes su método `__init__` es posible que tengas que llamar al método `__init__` de la superclase para inicializar los campos asociados a la superclase. Recuerda que puedes acceder a la superclase usando `super`.

Los tipos básicos de python están definidos en clases también, pero su nombre puede usarse para hacer conversiones.

```
>>> int["1"]
1
```

Para entender el funcionamiento de esa llamada, no hay más que recordar el que el aplicar el nombre de la clase como una llamada a función sirve para crear un objeto del tipo indicado, enviando los argumentos a su inicializador. Es decir, simplemente soportan el método `__init__` y construyen un nuevo objeto del tipo indicado a partir de lo que se les envía como argumento.

5.3.7 *Abrible* y *cerrable*

Los objetos *abribles* y *cerrables* puedan ser abiertos y cerrados de forma segura y con una sintaxis eficiente. Aunque no se van a detallar en profundidad, el objetivo de este punto es mostrar la sentencia `with` que se habilita gracias a estos métodos y mostrar cómo facilitan la apertura y cierre.

El PEP 343[1] muestra en detalle la implementación de la sentencia `with`. Simplificándolo y

[1] Puedes leer el contenido completo del PEP en:
`https://www.python.org/dev/peps/pep-0343/`

resumiéndolo, `with` sirve para abrir elementos y cerrarlos de forma automática.

> Los PEP (*Python Enhancement Proposals*) son propuestas de mejora para el lenguaje. Puedes consultar todos en la web de python. Son una fuente interesante de información y conocimiento del lenguaje y de programación en general.
> `https://www.python.org/dev/peps/`

Pensando en, por ejemplo, la lectura de un archivo, se requieren varias etapas para tratar con él, por ejemplo:

```
f = open["file.txt"]    # apertura del fichero
f.read[]                # lectura
f.close[]               # cierre
```

Este método es un poco arcaico y peligroso. Si durante la lectura del fichero ocurriera alguna excepción el fichero no se cerraría, ya que la excepción bloquearía la ejecución del programa. Para evitar estos problemas, lo lógico sería hacer una estructura `try-except` y añadir el cierre del fichero en un `finally`.

La sentencia `with` se encarga básicamente de hacer eso y facilita la escritura de todo el proceso quedándose así:

```
with f as open["file.txt"]: # apertura
    f.read[]                # en este cuerpo `f` está abierto

# Al terminar el cuerpo, de forma normal o forzada,
# `f` se cierra.
```

Ahora bien, para que el fichero pueda ser abierto y cerrado automáticamente, deberá tener implementados los métodos `__enter__` y `__exit__`. En el PEP 343 se muestra la equivalencia entre la sentencia `with` y el uso de `__enter__`, `__close__` y el `try-except`.

5.3.8 *Llamable*

Queda pendiente desde el capítulo sobre funciones responder a qué es un *callable* o *llamable*. Una vez llegados a este punto, tiene una respuesta fácil: un *llamable* es un objeto que soporta el método `__call__`.

Aunque pueda parecer sorprendente, las funciones en python también se llaman de este modo, así que realmente son objetos que se llaman porque soportan este método. Es lógico, porque las funciones, recuerda el capítulo previo, pueden guardar valores, como el contexto en el que se crean (la *closure*). Las funciones son meros *llamables* y como tales se comportan.

```
>>> class Dog:
...     def __call__[self]:
...         print["Dog called"]
...
>>> dog = Dog[]
>>> dog[]
Dog called
```

Ten en cuenta que el método `__call__` puede recibir cualquier cantidad de argumentos

como ya hemos visto en apartados anteriores, pero el primero será el propio objeto que está siendo llamado, el `self` que ya conocemos.

Resumiendo, el método `__call__` describe cómo se comporta el objeto cuando se le aplican los paréntesis.

5.3.9 *Subscriptable*

Tal y como el método anterior describía cómo se aplican los paréntesis a un objeto, los métodos que se muestran en este apartado describen el comportamiento del objeto cuando se le aplican los corchetes. Recordando el capítulo sobre datos, los corchetes sirven para acceder a valores de las listas, tuplas y diccionarios.

Cuando python encuentra que se está tratando de acceder a un campo de un objeto mediante los corchetes llama automáticamente al método `__getitem__` y cuando se intenta asociar un campo a un valor llama al método `__setitem__` del objeto. Al pedir la eliminación de un campo del objeto con la sentencia `del`, se llama al método `__delitem__`.

Aunque en otros conjuntos de métodos aquí descritos hemos inventado un nombre para este documento, Python denomina a este comportamiento *subscriptable* así que cuando intentes acceder usando corchetes a un objeto que no soporta esta funcionalidad, el error que saltará utilizará la misma nomenclatura que nosotros.

El siguiente ejemplo muestra el funcionamiento en una clase que en lugar de usar los métodos, imprime en pantalla. Lo lógico y funcional sería utilizar estos dos métodos para facilitar el acceso a campos de estas clases o para crear clases que pudiesen sustituir a listas, tuplas o diccionarios de forma sencilla, pero puede hacerse cualquier porque son métodos normales.

```
>>> class Dog:
...     def __getitem__(self, k):
...             print(k)
...     def __setitem__(self, k, v):
...             print(k, v)
...
>>> bobby = Dog()
>>> bobby["field"]
field
>>> bobby["field"] = 10
field 10
```

Fíjate en que reciben diferente cantidad de argumentos de entrada cada uno de los métodos. El método `__setitem__` necesita indicar no sólo qué *item* desea alterarse, sino su también su valor.

Slice notation

Se trata de una forma avanzada de seleccionar las posiciones de un objeto, el nombre viene de *slice*, rebanada, y significa que puede coger secciones del objeto en lugar de valores únicos. Piénsalo como en una barra de pan cortada en rebanadas de la que quieres seleccionar qué rebanadas te interesan en bloque.

No todos los objetos soportan *slicing*, pero los que lo hacen permiten acceder a grupos de valores en el orden en el que están indicando el inicio del grupo (inclusive), el final (no inclusive) y el salto de un elemento al siguiente.

Además, los valores del *slice* pueden ser negativos. Añadir un número negativo al salto implica que el salto se hace hacia atrás. Añadirlo en cualquier de los otros dos valores, inicio o final de grupo, implica que se cuenta el elemento desde el final de la colección en dirección opuesta a la normal.

La sintaxis de los *slice*s es la siguiente: `[inicio:fin:salto]`. Cada uno de los valores es opcional y si no se añaden se comportan de la siguiente manera:

- Inicio: primer elemento
- Fin: último elemento inclusive
- Salto: un único elemento en orden de cabeza a cola

> El índice para representar el último elemento es -1, pero si se quiere indicar como final, usar -1 descartará el último elemento porque el final no es inclusivo. Para que sea inclusivo es necesario dejar el campo fin vacío.

Dada una lista de los números naturales del 1 al 99, ambos incluidos, de nombre `l` se muestran unos casos de *slicing*.

```
>>> l[-5:]
[95, 96, 97, 98, 99]
>>> l[6:80:5]
[6, 11, 16, 21, 26, 31, 36, 41, 46, 51, 56, 61, 66, 71, 76]
>>> l[60::-5]
[60, 55, 50, 45, 40, 35, 30, 25, 20, 15, 10, 5]
```

La sintaxis de los *slice*s mostrada sólo tiene sentido a la hora de acceder a los campos de un objeto, si se trata de escribir suelta lanza un error de sintaxis. Para crear *slice*s de forma separada se construyen mediante la clase `slice` de la siguiente manera: `slice(inicio, fin, salto)`.

En los métodos que permiten a un objeto ser *subscriptable* (`__getitem__`, `__setitem__` y `__delitem__`) a la hora de elegir un *slice* se recibe una instancia del tipo *slice* en lugar de una selección única como en el ejemplo previo:

```
>>> class Dog:
...     def __getitem__(self, item):
...             print(item)
...
>>> bobby = Dog()
>>> bobby[1:100]
slice(1, 100, None)
>>> bobby[1:100:9]
slice(1, 100, 9)
>>> bobby[1:100:-9]
slice(1, 100, -9)
```

Por complicarlo todavía más, los campos del *slice* creado desde la clase `slice` pueden ser del tipo que se quiera. El formato de los `:` es únicamente *sintactic sugar* para crear *slices* de

tipo integer o string. Aunque después es responsabilidad del quien implemente soportar el tipo de *slice* definido, es posible crear *slices* de lo que sea, incluso anidarlos.

Como ejemplo de un caso que utiliza *slices* no integer, los tipos de datos como los que te puedes encontrar en la librería `pandas` soportan *slicing* basado en claves, como si de un diccionario se tratara.

5.3.10 Ejemplo de uso

Para ejemplificar varios de estos métodos especiales, tomamos como ejemplo una pieza de código fuente que quien escribe este documento ha usado en alguna ocasión en su trabajo como desarrollador.

Se trata de un iterable que es capaz de iterar en un sistema de ficheros estructurado en carpetas *año-mes-día* con la estructura `AAAA/MM/DD`. Este código se creó para analizar datos que se almacenaban de forma diaria en carpetas con esta estructura. Diariamente se insertaban fichero a fichero por un proceso previo y después se realizaban análisis semanales y mensuales de los datos. Esta clase permitía buscar por las carpetas de forma sencilla y obtener rápidamente un conjunto de carpetas que procesar.

El ejemplo hace uso del módulo `datetime`, un módulo de la librería estándar que sirve para procesar fechas y horas. Por ahora, puedes ver la forma de importarlo como una receta y en el siguiente capítulo la entenderás a fondo. El funcionamiento del módulo es sencillo y puedes usar la ayuda para comprobar las funciones que no conozcas.

Te animo a que analices el comportamiento del ejemplo, viendo en detalle cómo se comporta. Como referencia, fuera de la estructura de la clase, en las últimas líneas, tienes disponible un bucle que puedes probar a ejecutar para ver su comportamiento.

```
from datetime import timedelta
from datetime import date

class dateFileSystemIterator:

    """
    Iterate over YYYY/MM/DD filesystems or similar.
    """
    def __init__( self, start = date.today(), end = date.today(),
                  days_step = 1, separator = '/' ):
        self.start = start
        self.current = start
        self.end = end
        self.separator = separator
        self.step = timedelta( days = days_step )

    def __iter__( self ):
        self.current = self.start
        return self

    def __next__( self ):
        if self.current >= self.end:
            raise StopIteration
        else:
```

```
25              self.current += self.step
26              datestring = self.current - self.step
27              datestring = datestring.strftime( "%Y" \
28                                  + self.separator \
29                                  + "%m"+self.separator \
30                                  +"%d")
31              return datestring
32
33      def __repr__( self ):
34          out = self.current - self.step
35          tostring = lambda x: x.strftime("%Y" \
36                          + self.separator \
37                          + "%m" \
38                          + self.separator + "%d")
39          return "<dateFileSystemIterator: <Current: " \
40                  + tostring[self.current] + ">" \
41                  + ", <Start: " + tostring[self.start]  + ">" \
42                  + ", <End: " + tostring[self.end]  + ">" \
43                  + ", <Step: " + str[self.step] +  ">"
44
45
46  it = dateFileSystemIterator[start = date.today[] - timedelta[days=30]]
47  print[it]
48  for i in it:
49      print[i]
```

> Esta clase implementa tanto el iterable como su propio iterador, pero fíjate que cuando se llama a `__iter__` reinicia la cuenta. Otra forma de hacer esto podría ser con dos clases que diferencien bien los conceptos (*iterable* e *iterador*), pero es quizás demasiado escribir para un caso tan sencillo.
>
> El método `__repr__` hace mucho uso de la concatenación de strings y de la conversión de valores a string. Históricamente en python siempre ha habido formas más elegantes de hacer esto. La más reciente (a partir de python 3.6) es utilizar lo que se conoce como *formatted string literals* o *f-strings*. Puedes leer más sobre ellos en el PEP 498[2]

Ejercicio libre: generadores

La parte de la iteración del ejemplo previo puede realizarse forma más breve mediante el uso de la sentencia `yield`. Aunque no la trataremos, `yield` habilita muchos conceptos interesantes, entre ellos los *generadores*.

A continuación tienes un ejemplo de cómo resolver el problema anterior mediante el uso de esta sentencia. Te propongo como ejercicio que investigues cómo funciona buscando información sobre los *generadores* (*generator*) y la propia sentencia `yield`.

```
from datetime import datetime, timedelta

def iterate_dates[ date_start, date_end=datetime.today[],
                   separator='/', step=timedelta[days=1] ]:
    date = date_start
```

```
    while date < date_end:
        yield date.strftime('%Y'+separator+'%m'+separator+'%d')
        date += step
```

`yield` tiene mucha relación con las *corrutinas* (*coroutine*) que, aunque no se tratarán en este documento, son un concepto muy interesante que te animo a investigar. Si lo haces, verás que los generadores son un caso simple de una corrutina.

5.4 Lo que has aprendido

Este capítulo también ha sido intenso como el anterior, pero te prometo que no volverá a pasar. El interés principal de este capítulo es el de hacerte conocer la programación orientada a objetos y enseñarte que en python lo inunda todo. Todo son objetos.

Para entenderlo has comenzado aprendiendo lo que es la programación orientada a objetos, concretamente la orientada a clases, donde has visto por primera vez los conceptos de identidad propia, comportamiento y estado.

Desde ahí has saltado al fundamento teórico de la programación orientada a objetos y has visitado la encapsulación, la herencia y el polimorfismo para luego, una vez comprendidos, comenzar a definir clases en python.

Esto te ha llevado a necesitar conocer qué es el argumento que suele llamarse `self`, una excusa perfecta para definir qué son las variables y funciones de clase y en qué se diferencian de las propiedades y métodos.

Como la encapsulación no se había tratado en detalle aún, lo próximo que has hecho ha sido zambullirte en los campos privados viendo cómo python los crea mediante un truco llamado *name mangling* y su impacto en la herencia.

Aunque en este punto conocías el comportamiento general de la herencia hacia abajo, necesitabas conocerlo hacia arriba. Por eso, ha tocado visitar la función `super` en este punto, función que te permite acceder a la superclase de la clase en la que te encuentras. En lugar de contártela en detalle, se te ha dado una pincelada sobre ella para que tú investigues cuando lo veas necesario, pero que sepas por dónde empezar.

Para describir más en detalle lo calado que está python de programación orientada a objetos necesitabas un ejemplo mucho más agresivo: los métodos con nombres especiales. A través de ellos has visto cómo python recoge las funcionalidades estándar y te permite crear objetos que las cumplan. Además, te ha servido para ver que **todo** en python es un objeto (hasta las clases lo son[3]) y para ver formas elegantes de resolver problemas comunes, como los iteradores, `with` y otros.

[3] Puedes preguntárselo a python:

```
>>> class C: pass
...

>>> isinstance(C, object)
True
```

También, te recuerdo que, aunque sea de forma colateral y sin prestarle demasiada atención, se te ha sugerido que cuando programamos no lo hacemos únicamente para nosotros mismos y que la facilidad de lectura del código y la preparación de éste para que otros lo usen es primordial. Los próximos capítulos tratan en parte de ésto: de hacer uso del patrimonio tecnológico de la humanidad, y de ser parte de él.

6 Módulos y ejecución

Hasta ahora, has ejecutado el código en la REPL y de vez en cuando has usado `F5` en IDLE para ejecutar. Aunque te ha permitido salir del paso, necesitas saber más en detalle cómo funciona la ejecución para empezar a hacer tus programas. Además, es absurdo que te pelees contra todo, hay que saber qué batallas librar, así que necesitarás aprender a importar código realizado por otras personas para poder centrarte en lo que más te interesa: resolver tu problema.

Este capítulo trata ambas cosas, que están muy relacionadas, y sirve como trampolín para el siguiente, la instalación de nuevos paquetes y la gestión de dependencias, y los posteriores sobre librerías interesantes que te facilitarán el desarrollo de tus proyectos.

Este capítulo ya te capacita casi al cien por cien para la programación aunque aún no hemos trabajado su utilidad, pero seguro que alguna idea se te habrá ocurrido, si no no estarías leyendo este documento.

6.1 Terminología: módulos y paquetes

En python a cualquier fichero de código (extensión `.py`) se le denomina *módulo* (*module*). A cualquier directorio que contenga módulos de código python y un fichero llamado `__init__.py`, que puede estar vacío, se le denomina *paquete* (*package*). El uso del fichero `__init__.py` permite que python busque módulos dentro del directorio.

Piensa en los paquetes como grupos de módulos que incluso pueden anidarse con subpaquetes. En la documentación oficial verás en más de una ocasión que se trata a los paquetes como si fueran módulos, y tiene cierto sentido, porque, normalmente, existe un módulo principal que permite el acceso a un subpaquete. Así que, principalmente estás trabajando con un único módulo de un paquete, que contiene subpaquetes a los que este módulo hace referencia. Aunque pueda sonar algo enrevesado, no te preocupes por ahora: los módulos son ficheros únicos y los paquetes conjuntos de ellos.

6.2 Ejecución

Ya conoces un par de maneras de ejecutar tus módulos de python. Usar la REPL, introduciéndole el código que quieres ejecutar, llamar a la función «ejecutar módulo» de IDLE con la tecla `F5` e incluso llamar a la shell de sistema con un comando similar a este:

```
python mi_archivo.py
```

Normalmente, los ficheros de python se ejecutan de este último modo en producción, mientras que los dos anteriores son más usados a la hora de desarrollar. Realmente son métodos

similares, en todos ellos el intérprete accede al fichero o contenido que recibe y ejecuta las líneas una a una.

Existe también una opción adicional muy usada que sirve para ejecutar módulos que el sistema sea capaz de encontrar por sí mismo, en lugar de indicarle la ruta, se le puede indicar simplemente el nombre del módulo usando la opción `-m`:

```
python -m nombre_de_modulo
```

6.3 Importación y *namespaces*

Anteriormente hemos pasado sobre la sintaxis de la importación de forma muy superficial pero tampoco es mucho más compleja a lo que ha aparecido. La sentencia `import` permite importar diferentes módulos a nuestro programa como en el siguiente ejemplo:

```
>>> import datetime
>>> datetime
<module 'datetime' from '/usr/lib/python3.6/datetime.py'>
```

Han pasado, sin embargo, muchas cosas interesantes en el ejemplo. En primer lugar, python ha buscado y encontrado el módulo `datetime` en el sistema y, en segundo lugar, ha creado un objeto módulo llamado `datetime` que atesora todas las definiciones globales del módulo `datetime`.

Empezando por el final, python usa lo que se conoce como *namespace* de forma muy extendida. Los *namespaces*, de nombre (*name*) y espacio (*space*), son una herramienta para separar contextos ampliamente usada. Los objetos, en realidad, son una caso de *namespace* ya que cuando se llama a un método se le dice cuál es el contexto de la llamada, es decir: al método de qué objeto se llama.

Para los módulos el proceso es el mismo. La sentencia `import` trae un módulo al programa pero lo esconde tras su *namespace*, de este modo, para acceder a algo definido en el recién importado módulo es necesario indicarle el nombre de éste de la siguiente manera:

```
>>> import datetime
>>> datetime.date.today()
datetime.date(2019, 12, 3)
```

En el ejemplo se accede a la clase `date` dentro del módulo `datetime`, y se lanza su función `today`, que indica el día de hoy. Como puedes apreciar, el operador `.` se utiliza del mismo modo que en las clases y objetos, y en realidad es difícil saber cuándo se está accediendo a una clase y cuándo a un módulo, aunque tampoco es necesario saberlo.

Para no tener que escribir el nombre del módulo completo, existe otra versión de la sentencia `import` que tiene un comportamiento muy similar:

```
>>> import datetime as dt
>>> dt.date.today()
datetime.date(2019, 12, 3)
```

En este ejemplo, se ha cambiado el nombre del módulo a uno más corto decidido por el programador, `dt`. El funcionamiento es el mismo, simplemente se ha cambiado el nombre

para simplificar. Este cambio de nombre también es útil cuando se va a importar un módulo cuyo nombre es igual que alguna otra definición. Cambiando el nombre se evitan colisiones.

Existen versiones además, que permiten importar únicamente las funciones y clases seleccionadas, pero que las añaden al *namespace* actual, para evitar tener que usar el prefijo.

```
>>> from datetime import date
>>> date.today()
datetime.date(2019, 12, 3)
```

En este último ejemplo, se trae la clase `date` al contexto actual. También existe la posibilidad de importar más de una definición del módulo, usando la coma para separarlas, o todo lo que el módulo exponga mediante el símbolo `*`. Es peligroso, sin embargo, traer definiciones al namespace actual de forma descuidada, sobre todo con la última opción, porque es posible que se repitan nombres por accidente y se pisen definiciones. Los namespaces se inventan con el fin de separar las definiciones y evitar colisiones de este tipo.

6.3.1 Búsqueda

Una vez descrito cómo se interactúa con los módulos importados, es necesario describir dónde se buscan estos módulos.

Los módulos se buscan en los siguientes lugares:

1. El directorio del fichero ejecutado o el directorio de trabajo de la REPL
2. Los directorios indicados en el entorno
3. La configuración por defecto (depende de la instalación)

Esto significa que si guardas un archivo de python en IDLE y guardas otro más en el mismo directorio con el nombre `modulo.py` podrás importarlo usando `import modulo` en el primero, ya que comparten directorio. Lo mismo ocurre con los paquetes, crear un directorio con nombre `paquete` y añadirle un fichero vacío llamado `init.py` te permitirá hacer `import paquete`. Si añadieras más módulos dentro del paquete, podrías importar cada uno de ellos mediante `paquete.modulo`.

> Los nombres de los ficheros deben coincidir con el el nombre del módulo más la extensión `.py`. En el caso de los directorios, saltar a un subdirectorio implica acceder a un paquete, por lo que se añadirá un punto (`.`).

El primer punto sirve para facilitar que organices tu proyecto de python en varios módulos, separando así la funcionalidad en diferentes archivos.

Los últimos dos puntos son los que permiten a python encontrar su librería estándar y los módulos de sistema. El tercero depende de la instalación y del formato de ésta: si python está instalado como portable no será igual que si se instala en el sistema del modo habitual. El segundo punto también puede variar de un sistema a otro, pero en resumen se trata de varias variables de entorno de sistema que le indican a python dónde buscar (normalmente toman el nombre `PYTHONPATH`, pero no es siempre así). El segundo punto puede alterarse de modo que en función de lo que se le indique, se puede pedir a python que busque los módulos en un lugar u otro.

Estos lugares de búsqueda se pueden mostrar de la siguiente manera:

```
>>> import sys
>>> print(sys.path)
['',
  '/usr/lib/python36.zip',
  '/usr/lib/python3.6',
  '/usr/lib/python3.6/lib-dynload',
  '/usr/local/lib/python3.6/dist-packages',
  '/usr/lib/python3/dist-packages']
```

En función del sistema en el que te encuentres y la configuración que tengas, python mostrará diferente resultado.

Rescatando un ejemplo previo:

```
>>> import datetime
>>> datetime
<module 'datetime' from '/usr/lib/python3.6/datetime.py'>
```

Ahora entiendes por qué es capaz de encontrar `datetime` en `/usr/lib/python3.6`, carpeta listada en `sys.path`, bajo el nombre `datetime.py`.

6.4 Ejecución e importación

A la hora de importar un módulo, python procesa el contenido de éste ya que necesita definir las funciones, clases, valores, etc. a exportar. Para poder hacerlo, python necesita ejecutar el módulo.

Python define una forma de separar la funcionalidad del código de sus definiciones con el fin de poder crear código cuyas definiciones sean reutilizables mediante la importación en otro módulo, sin que tenga ninguna funcionalidad cuando esto ocurra, pero habilitando que tenga funcionalidades cuando sea llamado directamente.

Un ejemplo de uso de esto puede ser un módulo de acceso a ficheros, por ejemplo, que visualice el contenido del fichero cuando se llame de forma directa pero que cuando se importe únicamente aporte las funciones de lectura y escritura sin leer y mostrar ningún fichero.

Para que los módulos puedan tener esta doble vida, python define la variable `__name__` que representa en qué nivel del *scope* se está ejecutando el módulo actual. La variable `__name__` toma el valor del nombre del módulo cuando éste está siendo importado y el valor `__main__` cuando ha sido llamado de forma directa o está siendo ejecutado en la REPL. `__main__` es el *scope* global de los programas, por lo que cuando algo se declara en él, implica que es el programa principal.

Para poder diferenciar cuándo se ha ejecutado un módulo de forma directa y cuando se ha importado se utiliza lo que se conoce como `__main__` *guard*:

```
if __name__ == "__main__":
    # Este bloque sólo se ejecuta cuando el módulo es el principal
```

Aunque igual es un poco incómodo de entender de primeras, encontrarás esta estructura en casi cualquier módulo de código python. Se utiliza constantemente, incluso para los casos

en los que no se pretende que el código pueda importarse. Es una buena práctica incluir el *guard* para separar la ejecución de las definiciones, de este modo, quien quiera saber cuál es la funcionalidad del módulo tendrá mucho más fácil la búsqueda.

Puedes leer más sobre este tema en la documentación de python[1].

Siguiendo este concepto, también existe un estándar de nomenclatura de ficheros. El nombre `__main__.py` hace referencia al fichero que contiene el código principal del programa y será el fichero que python buscará ejecutar siempre que se le pida ejecutar un paquete o un directorio sin especificar qué módulo debe lanzar. Por ejemplo, ejecutar `python .`[2] en la shell de sistema es equivalente a ejecutar `python __main__.py`.

6.5 Lo que has aprendido

En este capítulo corto has aprendido lo necesario sobre importación de módulos y ejecución de código. Conocer en detalle el patrón de búsqueda de módulos de python es primordial para evitar problemas en el futuro y organizar los proyectos de forma elegante.

Además, el `__main__` *guard* era una de las últimas convenciones de uso común que quedaban por explicar y una vez vista ya eres capaz de leer proyectos de código fuente que te encuentres por ahí sin demasiados problemas.

Los próximos capítulos, basándose en lo aprendido en éste, te mostrarán cómo instalar nuevas dependencias y cómo preparar tu propio código para que pueda ser instalado de forma limpia y elegante.

[1] `https://docs.python.org/3/library/__main__.html`

[2] `.` significa directorio actual en cualquiera de los sistemas operativos comunes.

7 Instalación y dependencias

Ahora que sabes lidiar con módulos, necesitas aprender a instalarlos en tu propio sistema, porque es bastante tedioso que tengas que copiar el código de todas tus dependencias en la carpeta de tu proyecto.

En la introducción nos aseguramos de instalar con python la herramienta `pip`. Que sirve para instalar paquetes nuevos en el sistema de forma sencilla.

7.1 Sobre PIP

`pip` es una herramienta extremadamente flexible, capaz de instalar módulos de python de diferentes fuentes: repositorios de git, carpetas de sistema o, la más interesante quizás de todas, el *Python Package Index* (*PyPI*)[1].

`pip` buscará la descripción del paquete en la fuente que se le indique y, de esta descripción, obtendrá las dependencias necesarias y las indicaciones de cómo debe instalarlo. Una vez procesadas las normas, procederá a instalar el paquete en el directorio de sistema con todas las dependencias de éste para que funcione correctamente.

Una vez instalado el paquete en el directorio de sistema está listo para ser importado.

7.1.1 PyPI

El *Python Package Index* o *PyPI* es un repositorio que contiene software programado en python. En él se listan miles de librerías independientes para que cualquiera pueda descargarlas e instalarlas. Más adelante veremos algunas de ellas y nos acostumbraremos a usar PyPI como recurso.

Ahora que sabes programar en python tú también puedes publicar tus proyectos ahí para que otras personas los usen para crear los suyos.

7.1.2 Reglas de instalación

Para que `pip` pueda hacer su trabajo correctamente hay que indicarle cómo debe hacerlo, ya que cada paquete es un mundo y tiene necesidades distintas. El módulo `setuptools` permite crear un conjunto de reglas comprensible por `pip` que facilita la distribución e instalación.

Normalmente este conjunto de reglas suele almacenarse en un fichero de nombre `setup.py`. Como ejemplo de `setup.py` puedes ver el de la librería `BeautifulSoup4`[2] que se adjunta a

[1] `https://pypi.org/`
[2] `https://www.crummy.com/software/BeautifulSoup/bs4/doc/`

continuación con ligeras alteraciones para que encaje en la página:

```
from setuptools import (
    setup,
    find_packages,
)

with open("README.md", "r") as fh:
    long_description = fh.read()

setup(
    name="beautifulsoup4",
    # NOTE: We can't import __version__ from bs4 because bs4/__init__.py
    # is Python 2 code, and converting it to Python 3 means going through
    # this code to run 2to3.
    # So we have to specify it twice for the time being.
    version = '4.8.1',
    author="Leonard Richardson",
    author_email='leonardr@segfault.org',
    url="http://www.crummy.com/software/BeautifulSoup/bs4/",
    download_url = "http://www.crummy.com/software/BeautifulSoup/bs4/download/",
    description="Screen-scraping library",
    install_requires=["soupsieve>=1.2"],
    long_description=long_description,
    long_description_content_type="text/markdown",
    license="MIT",
    packages=find_packages(exclude=['tests*']),
    extras_require = {
        'lxml' : [ 'lxml'],
        'html5lib' : ['html5lib'],
    },
    use_2to3 = True,
    classifiers=[
        "Development Status :: 5 - Production/Stable",
        "Intended Audience :: Developers",
        "License :: OSI Approved :: MIT License",
        "Programming Language :: Python",
        "Programming Language :: Python :: 2.7",
        'Programming Language :: Python :: 3',
        "Topic :: Text Processing :: Markup :: HTML",
        "Topic :: Text Processing :: Markup :: XML",
        "Topic :: Text Processing :: Markup :: SGML",
        "Topic :: Software Development :: Libraries :: Python Modules",
    ],
)
```

En el ejemplo se aprecia a la perfección el tipo de información que es necesario aportarle a `setuptools`. Dependiendo del proyecto, esta configuración puede ser más compleja o más sencilla, y tendrás que indagar a través de la configuración de `setuptools` para ajustar la herramienta a tu proyecto.

7.2 Entornos virtuales

La herramienta `pip` es interesante para instalar herramientas en el sistema pero tiene ciertas carencias. La primera, que no es capaz de resolver las dependencias a la hora de desinstalar paquetes, por lo que, si se instala un paquete que dependa de otro paquete, `pip` instalará todas las dependencias necesarias, pero por miedo a romper paquetes, no las desinstalará si se le pide desinstalar el paquete que las arrastró.

Por otro lado, si quieres trabajar en proyectos de desarrollo, probablemente tengas que instalar sus dependencias. Si tienes varios proyectos en marcha simultáneamente o si tu sistema necesita de alguna herramienta escrita en python, es posible que tengas colisiones.

Imagina que dos de los proyectos, por ejemplo, usan versiones diferentes de una de sus librerías. Si instalas sus dependencias usando `pip`, se mezclaran en tu sistema y no podrán coexistir. Además, cuando termines los proyectos o abandones su desarrollo, te interesará limpiar sus dependencias de tu sistema, cosa complicada si `pip` no gestiona la liberación de paquetes de forma correcta.

Para evitar estos problemas y algún otro adicional, existen herramientas que alteran el comportamiento de `pip`, y del propio python, creando lo que se conoce como *entornos virtuales* (*virtual environments*) que quedan aislados entre ellos y el sistema.

El funcionamiento de los entornos virtuales es muy sencillo. Cuando se activan, crean un nuevo contexto en el que alteran las variables de entorno que describen dónde debe buscarse el intérprete de python, dónde busca éste los módulos y dónde instala `pip` los paquetes. Cuando se desactivan, se restaura el entorno por defecto. De este modo, cada entorno virtual queda perfectamente aislado de otros o incluso de la instalación del sistema, permitiéndote hasta tener diferentes versiones de python en tu sistema y que no colisionen entre ellas. Cuando has terminado con tu entorno virtual puedes borrarlo de forma segura sabiendo que no va a afectar a tu sistema y que va a llevarse todas las dependencias con él.

Históricamente se han utilizado varias herramientas para esta labor, como `virtualenv`, que como era poco amigable se simplificaba con `virtualenv-wrapper`, u otras. Hoy en día `pipenv` es la herramienta recomendada.

`pipenv` es una combinación de `virtualenv` y `pip` creada para gestionar entornos virtuales y dependencias de desarrollo. Puedes considerarla un gestor de paquetes de desarrollo como `npm` en JavaScript, `composer` en PHP o cualquier otro que conozcas. Aporta la mayor parte de funcionalidades habituales como ficheros de dependencias, lockfiles etc. mientras que expone una interfaz de comandos sencilla y bien documentada.

7.2.1 Instalación

Para instalar pipenv, podemos usar `pip`, que instalamos en la introducción. En la shell de sistema ejecutando:

```
pip install pipenv
```

> En función del sistema que utilices, puede que `pip` se llame `pip3`. El funcionamiento es idéntico.

7.2.2 Uso

Una vez instalado `pipenv`, puedes usarlo en el directorio que desees para crear un conjunto de nuevas dependencias pidiéndole que instale un nuevo paquete lanzando la orden `pipenv install` en la shell de sistema.

Para ejecutar módulos en el entorno virtual recién creado dispones de dos opciones: `pipenv shell` que prepara una shell de sistema en el entorno o `pipenv run` que ejecuta el comando que se le indique en el entorno.

Puedes seguir añadiendo dependencias al proyecto con `pipenv install` y eliminar las que no te gusten con `pipenv uninstall`. Además, dispones de muchas opciones adicionales que te animo que ojees ejecutando `pipenv --help`.

7.2.3 Usar IDLE desde un entorno virtual

Si utilizas entornos virtuales deberás preparar IDLE para verlos y que su REPL y su intérprete encuentren los paquetes del entorno virtual.

Puedes lanzar IDLE desde el entorno virtual usando el siguiente truco en la shell de sistema:

```
pipenv run python -m idlelib.idle
```

Recordando el capítulo anterior y lo descrito en este, ese comando ejecuta `python -m idlelib.idle` en el entorno virtual en el que te encuentres. El comando, por su parte le pide a python que ejecute el módulo `idle` del paquete `idlelib`, que contiene el propio programa IDLE.

Si trabajas con otros editores integrados de código tendrás que aprender a hacer que sus intérpretes busquen en el entorno virtual actual, pero casi todos los editores actuales soportarán esta opción de una forma u otra.

7.3 Otras herramientas

La realidad es que las herramientas propuestas no son las únicas que existen para estas tareas. Históricamente, python ha tenido otras herramientas con comportamientos similares a `pip`, como `easy_install` y otras con parecidas a `setuptools` como `distutils` y otras que asemejan a `pipenv`. Este capítulo trata únicamente las herramientas que se consideran las recomendadas en el momento en el que se está escribiendo[3], aunque los conceptos descritos son extrapolables, no sólo a otras herramientas sino también a otros lenguajes y entornos.

[3] Para un listado más extenso, visitar:
`https://packaging.python.org/guides/tool-recommendations/`

7.4 Lo que has aprendido

En este capítulo has aprendido lo necesario sobre las herramientas que rodean a python y su uso. De este modo, no te vas perder en un maremágnum de nombres extraños y comandos cuando trabajes en proyectos que ya las usan.

Más que convertirte en un experto de cómo trabajar con estas herramientas, cosa que te dará la práctica, este episodio te ha dado las referencias que necesitas para ir creciendo por tu cuenta en su uso, explicándote el interés de cada una de ellas.

Te encontrarás, probablemente, proyectos que utilicen las herramientas de un modo peculiar, usen otras herramientas o hasta proyectos que no las usen correctamente. Este es un mundo complejo, y la historia de python no facilita la elección. Durante mucho tiempo la comunidad creó nuevas herramientas para suplir las carencias de otras y el ecosistema se complicó. A pesar de que hoy en día se recomiende el uso de unas herramientas sobre otras, no siempre fue así y muchos proyectos se han quedado obsoletos en este sentido.

Para evitar atarte a las herramientas que aquí se muestran, se ha preferido darte una ligera noción, haciéndote recordar que las herramientas no son realmente lo importante, sino el conocimiento subyacente.

Con la herramienta principal que has conocido en este apartado, `pipenv`, podrás instalar los paquetes que quieras de forma aislada y jugar con ellos todo lo que desees sin manchar tu sistema por lo que podrás adentrarte en los próximos capítulos sin miedo a estropear tu pulcra configuración inicial.

8 La librería estándar

La librería estándar se refiere a todas las utilidades que un lenguaje de programación trae consigo. Los lenguajes de programación, a parte de aportar la propia funcionalidad del lenguaje en sí mismo, que simplemente sería la ejecución del código fuente que se le indique, suelen incluir funcionalidades que no están necesariamente relacionadas con ese proceso.

El motivo de esto es facilitar el uso del lenguaje para las labores más comunes, incluyendo el código necesario para realizarlas sin necesitar añadidos externos.

En lenguajes de programación de dominio específico la librería estándar suele contemplar muchos casos de ese dominio concreto. Por ejemplo, el lenguaje de programación Julia, aunque esté diseñado con el objetivo de ser un lenguaje válido para cualquier uso, su área de trabajo se centra en el entorno científico. Es por eso que incluye en su librería estándar paquetes de álgebra lineal, estadística y otros.

Existen muchas diferentes aproximaciones a la librería estándar. Algunos lenguajes las mantienen extremadamente reducidas con el fin de que la implementación del lenguaje sea más liviana, con la contrapartida de forzar a quien los use a tener que utilizar herramientas externas o a desarrollarlas.

Python es un lenguaje de propósito general con una extensísima librería estándar. Esto dificulta la selección de apartados a mostrar en este capítulo, pero facilita la programación de cualquier aplicación en éste lenguaje.

Los paquetes estándar de python facilitan la lectura de infinidad de tipos de fichero, la conversión y el tratamiento de datos, el acceso a la red, la ejecución concurrente, etc. por lo que librería estándar es más que suficiente para muchas aplicaciones.

Conocer la librería estándar y ser capaz de buscar en ella los paquetes que necesites te facilitará mucho la tarea, evitando, por un lado, que dediques tiempo a desarrollar funcionalidades que el propio lenguaje ya aporta y, por otro, que instales paquetes externos que no necesitas.

Todos los apartados aquí listados están extremadamente bien documentados en la página oficial de la documentación de python y en la propia ayuda. Eso sí, tendrás que importarlos para poder leer la ayuda, pero ya sabes cómo se hace.

A continuación se recogen los módulos más interesantes, aunque en función del proyecto puede que necesites algún otro. Puedes acceder al listado completo en la página oficial de la documentación[1].

[1] `https://docs.python.org/3/library/index.html#library-index`

8.1 Interfaz al sistema operativo

Ya definimos previamente el concepto *interfaz* como superficie de contacto entre dos entidades. El módulo `os` facilita la interacción entre python y el sistema operativo.

La comunicación con el sistema operativo es primordial para cualquier lenguaje de programación de uso general, ya que es necesario tener la capacidad de hacer peticiones directas al sistema operativo. Por ejemplo, cambiar de directorio actual para facilitar la inclusión rutas relativas en el programa, resolver rutas a directorios y un largo etc.

En capítulos previos hemos hablado de las diferencias entre sistemas operativos. Esta librería, además, facilita el trabajo para esos casos. Por ejemplo, que el separador de directorios en UNIX es / y en Windows \, si quisiéramos programar una aplicación multiplataforma, nos encontraríamos con problemas. Sin embargo, el módulo `os.path` dispone de herramientas para gestionar las rutas de los directorios por nosotros, por ejemplo, la variable `os.path.sep` (copiada en `os.sep` por abreviar), guarda el valor del separador del sistema en el que se esté ejecutando la aplicación: / en UNIX y \ en Windows.

Este paquete es muy interesante para desarrollar código portable entre las diferentes plataformas.

8.2 Funciones relacionadas con el intérprete

Aunque el nombre de este módulo, `sys`, suene complicado, su uso principal es el de acceder a funcionalidades que el propio python controla. Concretamente, se usa sobre todo para la recepción de argumentos de entrada al programa principal, la redirección de entradas y salidas del programa y la terminación del programa.

8.2.1 Salida forzada

Para salir de forma abrupta del programa y terminar su ejecución, python facilita la función `sys.exit()`.

8.2.2 *Standard streams*

Cuando se trabaja en programas que funcionan en la terminal se pueden describir tres vías de comunicación con el usuario:

1. El teclado: de donde se obtiene lo que el usuario teclee mediante la función `input`.
2. La pantalla: a donde se escribe al ejecutar la función `print`.
3. Y la salida de errores: Una salida especial para los fallos, similar a la anterior, pero que se diferencia para poder hacer un tratamiento distinto y porque tiene ciertas peculiaridades distintas. Los mensajes de error de excepciones se envían aquí.

Estas tres vías se conocen comúnmente como entrada estándar (*standard input*), `stdin`, salida estándar (*standard output*), `stdout`, y error estándar (*standard error*), `stderr`. Este concepto responde al nombre de *standard streams* y es una abstracción de los dispositivos físicos que antiguamente se utilizaban para comunicarse con una computadora. Hoy en día, estos tres

streams son una abstracción de esa infraestructura y se comportan como simples ficheros de lectura en el primer caso y de escritura en los otros dos.

Los diferentes sistemas operativos los implementan a su modo, pero desde el interior de python son simplemente ficheros abiertos, como si se hubiese ejecutado la función `open` en ellos y ellos se encargan de leer o escribir a la vía de interacción con el usuario correspondiente. Es decir, en realidad `print`, `input`, etc. son únicamente funciones que facilitan la labor de escribir manualmente en estos *streams*:

```
>>> import sys
>>> chars = sys.stdout.write["Hola\n"]
Hola
>>> chars     # Recoge el número de caracteres escritos
5
```

La realidad es que estos *streams* dan una flexibilidad adicional muy interesante. Como son únicamente variables de ficheros abiertos pueden cerrarse y sustituirse por otros, permitiéndote redireccionar la entrada o la salida de un programa a un fichero.

El siguiente ejemplo redirecciona la salida de errores a un fichero llamado `exceptions.txt` y trata de mostrar una variable que no está definida. Python en lugar de mostrar el mensaje de una excepción, la escribe en el fichero al que se ha redireccionado la salida:

```
>>> sys.stderr = open["exceptions.txt", "w"]
>>> aasda   # No muestra el mensaje de error
>>>
>>> with open["exceptions.txt"] as f:
...     f.read[]    # Muestra el contenido del archivo
...
'Traceback (most recent call last):\n  File "<stdin>", line 1, in \
<module>\nNameError: name \'aasda\' is not defined\n'
```

8.2.3 Argumentos de entrada

Es posible añadir argumentos de entrada a la ejecución de los programas. Piensa en el propio programa llamado python, al ejecutarlo en la shell de sistema se le pueden mandar diferentes opciones que, por ejemplo, digan qué módulo debe ejecutar:

```
python test.py
```

Para la shell de sistema, todo lo que se escriba después de la primera palabra es considerado un argumento de entrada.

Cuando se ejecutan nuestros programas escritos en python, es posible también enviarles argumentos de entrada:

```
python test.py argumento1 argumento2 ...
```

Lo que el python reciba después del nombre del módulo a ejecutar lo considerará argumentos de entrada de nuestro módulo y nos lo ordenará y dejará disponible en la variable `sys.argv`, una lista de todos los argumentos de entrada.

Si muestras el contenido de la variable en la REPL, te responderá `['']` ya que la REPL se ejecuta sin argumentos. Sin embargo, si creas un módulo y le añades este contenido:

```
import sys
print(sys.argv)
```

Verás que se imprime una lista con el nombre del archivo en su primer elemento. Esta diferencia sirve para que desde el propio programa se conozca también cómo se le llamó. El uso más obvio de esto es poder mostrar una ayuda coherente con el nombre del programa.

Si ejecutas el módulo desde la shell de sistema añadiéndole argumentos de entrada:

```
python modulo.py arg1 arg2 arg3
```

La lista `sys.argv` recibirá todos, siempre a modo de string.

Los argumentos de entrada son extremadamente interesantes para permitir que los programas sean configurables por quien los use sin necesidad de tener que editar el código fuente, cosa que nunca debería ser necesaria a menos que exista un error en éste o quiera añadirse una funcionalidad.

8.3 Procesamiento de argumentos de entrada

Como la variable `sys.argv` entrega los argumentos de entrada tal y como los recibe y no comprueba si son coherentes, python dispone de una librería adicional para estas labores. El módulo `argparse` permite definir qué tipo de argumentos de entrada y opciones tiene tu programa y qué reglas deben seguir. Además, facilita la creación de ayudas como la que se muestra cuando ejecutas `python -h` o `pip -h` en la shell de tu sistema.

Es una librería bastante compleja con infinidad de opciones que es mejor que leas en la propia documentación cuando necesites utilizarla.

8.4 Expresiones regulares

Las expresiones regulares (*regular expression*, también conocidas como *regexp* y *regex*) son secuencias de caracteres que describen un patrón de búsqueda.

En python se soportan mediante el módulo `re` de la librería estándar.

8.5 Matemáticas y estadística

El módulo `math` soporta gran cantidad de operaciones matemáticas avanzadas para coma flotante. En él puedes encontrar logaritmos, raíces, etc.

El módulo `statistics` soporta estadística básica como medias, medianas, desviación típica, etc.

Ambos módulos tienen mucho interés ya que python se usa extensivamente en el análisis de datos. Aunque tiene librerías de terceros mucho más adecuadas para esta labor, para proyectos pequeños puede que sea suficiente con estos módulos.

8.6 Protocolos de internet

`urllib` es un conjunto de paquetes que permiten seguir URLs o enlaces, principalmente para HTTP y su versión segura HTTPs. Soporta cookies, redirecciones, autenticación básica, etc.

El siguiente ejemplo te muestra cómo descargar el primer boceto del estándar del protocolo HTTP de la página web del IETF. Recordando los apartados previos, fíjate en el uso de la sentencia `with` y en el uso de los corchetes para obtener un número limitado de caracteres de la recién decodificada respuesta.

```
>>> from urllib.request import urlopen
>>> with urlopen("https://tools.ietf.org/rfc/rfc2068.txt") as resp:
...     print( resp.read().decode("utf-8")[:1750] )
...
'
```

```
Network Working Group                                          R. Fielding
Request for Comments: 2068                                       UC Irvine
Category: Standards Track                                        J. Gettys
                                                                  J. Mogul
                                                                       DEC
                                                                H. Frystyk
                                                           T. Berners-Lee
                                                                  MIT/LCS
                                                              January 1997


                Hypertext Transfer Protocol -- HTTP/1.1

Status of this Memo

   This document specifies an Internet standards track protocol for the
   Internet community, and requests discussion and suggestions for
   improvements.  Please refer to the current edition of the "Internet
   Official Protocol Standards" (STD 1) for the standardization state
   and status of this protocol.  Distribution of this memo is unlimited.

Abstract

   The Hypertext Transfer Protocol (HTTP) is an application-level
   protocol for distributed, collaborative, hypermedia information
   systems. It is a generic, stateless, object-oriented protocol which
   can be used for many tasks, such as name servers and distributed
   object management systems, through extension of its request methods.
   A feature of HTTP is the typing and negotiation of data
   representation, allowing systems to be built independently of the
   data being transferred.

   HTTP has been in use by the World-Wide Web global information
   initiative since 1990. This specification defines the protocol
```

```
   referred to as "HTTP/1.1".

'
>>>
```

8.7 Fechas y horas

`datetime`, que ya ha aparecido anteriormente, es un módulo para gestión de fecha y hora. `datetime` soporta gran cantidad de operaciones y tipos de dato. Entre ellos los `timedeltas`, diferencias temporales que te permiten sumar y restar fechas con facilidad con los operadores habituales.

Con las facilidades de `datetime`, es poco probable que necesites importar una librería de gestión de fecha y hora independiente.

8.8 Procesamiento de ficheros

Python habilita gran cantidad de procesadores de formatos de fichero, los dos que se lista en este apartado tienen especial interés.

El primer módulo, `json`, sirve para manipular datos en formato JSON. Sus dos funciones principales `dumps` y `loads` vuelcan o cargan datos de JSON a diccionarios o listas en una sola orden debido que la propia estructura de JSON está formada por parejas clave valor o listas de valores ordenadas por índices. Esta es la razón por la que en muchas ocasiones se utilizan ficheros de formato JSON para intercambiar información entre aplicaciones de python: su lectura y escritura es extremadamente sencilla.

El segundo módulo, `sqlite3`, facilita el acceso a ficheros en formato SQLite3, un formato binario con una interfaz de acceso que permite consultas SQL. El módulo `sqlite3` es capaz de convertir las tablas que SQLite3 retorna a estructuras de python de forma transparente y cómoda por lo que es un aliado interesante para aplicaciones que requieren una base de datos pequeña y resistente a las corrupciones.

8.9 Aritmética de coma flotante decimal

En el apartado sobre datos tratamos la complejidad de los números de coma flotante y que su representación binaria puede dar lugar a problemas. Este módulo de aritmética decimal aporta una solución rigurosa a este problema con el fin de facilitar el uso de python en entornos que requieren precisión estricta que incluso puede requerir cumplir con normativas, como puede ser la banca.

La documentación del módulo muestra un par de ejemplos muy interesantes usando la clase `Decimal` aportada por éste. Se adjuntan a continuación para que los estudies y los diseccione en busca de las diferencias con el uso de los números de coma flotante normales:

```
>>> from decimal import *
>>> round[Decimal['0.70'] * Decimal['1.05'], 2]
```

```
Decimal['0.74']
>>> round[.70 * 1.05, 2]
0.73

>>> Decimal['1.00'] % Decimal['.10']
Decimal['0.00']
>>> 1.00 % 0.10
0.09999999999999995

>>> sum[[Decimal['0.1']]*10] == Decimal['1.0']
True
>>> sum[[0.1]*10] == 1.0
False
```

8.10 Lo que has aprendido

Más que aprender, en este apartado has sobrevolado la librería estándar de python desde la distancia y te has parado a observar qué problemas comunes ya están resueltos en python.

La realidad es que la librería estándar es mucho más extensa que lo listado en este apartado, pero aquí se han mencionado los módulos que más frecuentemente se suelen usar en entornos normales y algunos otros que tienen interés a la hora de afianzar conocimiento que has obtenido en capítulos previos, como es el caso del módulo `decimal`.

Este capítulo empieza a mostrarte el interés de programar en python y las posibilidades que tienes con él, únicamente con la librería estándar. En el próximo verás la cantidad de funcionalidad adicional que le puedes añadir con un par de librerías escritas por terceros.

Sirva también este capítulo como recordatorio de lo extensa que es la librería estándar de python. En el futuro, cuando tengas intención de buscar una librería para resolver un problema que te despista de trabajar en la funcionalidad principal de tu aplicación, empieza por la librería estándar.

9 Librerías útiles

Ahora que ya sabes cómo instalar librerías y que has visto que muchas funcionalidades están contenidas en la librería estándar de python, es un buen momento para que visites varios proyectos que aportan recursos muy interesantes a la hora de resolver problemas. Debido al carácter de uso general de python, estas librerías aportan facilidades muy diversas. El criterio para escogerlas parte de la experiencia personal del autor de este documento y añade algunas librerías y herramientas que pueden ser interesantes debido a su amplio uso en la industria.

9.1 Librerías científicas: ecosistema SciPy

SciPy es un ecosistema de librerías que tiene como objetivo facilitar el cálculo en ingeniería, matemática y ciencia en general.

Además de ser el nombre del ecosistema, comparte nombre con una de las librerías fundamentales de éste. El ecosistema está formado por varias librerías, entre ellas se encuentran:

- Numpy: un paquete de computación científica para python. Soporta matrices multidimensionales, funciones sofisticadas y álgebra lineal, entre otras.
- SciPy: librería basada en Numpy que aporta rutinas numéricas eficientes para interpolación, optimización, algebra lineal, estadística y otros.
- SymPy: una librería de matemática simbólica para python que complementa el resto de librerías del ecosistema, ya que casi todas están orientadas al análisis numérico.
- Matplotlib: librería para representar gráficos y figuras científicas en 2D.
- Pandas: aporta unas estructuras de datos muy potentes basadas en tablas, su objetivo es reforzar a python a la hora de tratar datos. El área de esta librería es el del análisis de datos, pero puede combinarse con otras áreas de estudio como la econometría (ver el proyecto statsmodels). Esta librería dispone de un ecosistema muy poderoso a su alrededor debido al auge del análisis de datos en la industria del software. Muchas librerías comparten la interfaz de Pandas por lo que tener nociones de su comportamiento abre muchas puertas en el sector del análisis de datos. Al igual que ocurre en SciPy, las estructuras de datos de Pandas también se basan en Numpy.
- IPython: más que una librería, IPython es una herramienta. Se trata de una REPL interactiva (su propio nombre viene de *Interactive Python*) que añade diversas funcionalidades sobre la REPL de python habitual: histórico de comandos, sugerencias, comandos mágicos (*magic*) que permiten alterar el comportamiento de la propia interfaz y un larguísimo etcétera. IPython, además, sirve como núcleo de los cuadernos Jupyter que

integran una shell de python con visualización de tablas y gráficos para generar documentos estilo *literate programming*.

9.2 Machine Learning: ScikitLearn

ScikitLearn es una librería de machine learning muy potente, implementa gran cantidad de algoritmos y tiene una documentación extensa, sencilla y educativa.

Encaja a la perfección con el ecosistema SciPy, ya que se basa en NumPy, SciPy y Matplotlib.

9.3 Peticiones web: Requests

Requests es una librería alternativa al módulo `urllib` aportado por la librería estándar de python. Se describe a sí misma como «HTTP para seres humanos», sugiriendo que `urllib` no es cómoda de usar.

Requests gestiona automáticamente las URL-s de las peticiones a partir de los datos que se le entreguen, sigue redirecciones, almacena cookies, descomprime automáticamente, decodifica las respuestas de forma automática, etc. En general es una ayuda cuando no se quiere dedicar tiempo a controlar cada detalle de la conexión de modo manual.

9.4 Manipulación de HTML: Beautifulsoup

Beautifulsoup es una librería de procesado de HTML extremadamente potente. Simplifica el acceso a campos de ficheros HTML con una sintaxis similar a los objetos de python, permitiendo acceder a la estructura anidada mediante el uso del punto en lugar de tener que lidiar con consultas extrañas o expresiones regulares. Esta funcionalidad puede combinarse con selectores CSS para un acceso mucho más cómodo.

9.5 Tratamiento de imagen: Pillow

Pillow es un fork de la librería PIL (*Python Imaging Library*) cuyo desarrollo fue abandonado. En sus diversos submódulos, Pillow permite acceder a datos de imágenes, aplicarles transformaciones y filtros, dibujar sobre ellas, cambiar su formato, etc.

9.6 Desarrollo web: Django, Flask

Existen infinidad de frameworks y herramientas de desarrollo web en python. Así como en el caso del análisis de datos la industria ha convergido en una herramienta principal, Pandas, en el caso del desarrollo web hay muchas opciones donde elegir.

Los dos frameworks web más usados son Django y Flask, siendo el segundo menos común que el primero pero digno de mención en conjunción con el otro por varias razones.

La primera es la diferencia en la filosofía: así como Django decide con qué herramientas se debe trabajar, Flask, que se define a sí mismo como microframework, deja en manos de quien lo usa la elección de qué herramientas desea aplicarle. Cada una de las dos filosofías tiene ventajas y desventajas y es tu responsabilidad elegir las que más te convengan para tu proyecto.

La segunda razón para mencionar Flask es que su código fuente es uno de los más interesantes a la hora de usar como referencia de cómo se debe programar en python. Define su propia norma de estilo de programación, basada en la sugerencia de estilo de python[1] y su desarrollo es extremadamente elegante.

Django, por su parte, ha sido muy influyente y muchas de sus decisiones de diseño han sido adoptadas por otros frameworks, tanto en python como en otros lenguajes. Lo que sugiere que está extremadamente bien diseñado.

A pesar de las diferencias filosóficas, existen muchas similitudes entre ambos proyectos por lo que aprender a usar uno de ellos facilita mucho el uso del otro y no es aprendizaje perdido. No tengas miedo en lanzarte a uno.

9.7 Protocolos de red: Twisted

Twisted es motor de red asíncrono para python. Sobre él se han escrito diferentes librerías para gestión de protocolos de Internet como DNS (via Twisted-Names), IMAP y POP3 (via Twisted-Mail), HTTP (via Twisted-Web), IRC y XMPP (via Twisted-Words), etc.

El diseño asíncrono del motor facilita sobremanera las comunicaciones eficientes. Programar código asíncrono en python es relativamente sencillo, pero ha preferido dejarse fuera de este documento por diversas razones. Te animo a indagar en esta librería para valorar el interés del código asíncrono.

9.8 Interfaces gráficas

A pesar de que python dispone de un módulo en su librería estándar para tratar interfaces gráficas llamado TKinter, es recomendable utilizar librerías más avanzadas para esto. TKinter es una interfaz a la herramienta Tk del lenguaje de programación Tcl y acompaña a python desde hace años.

En programas simples TKinter es más que suficiente (IDLE, por ejemplo, está desarrollado con TKinter) pero a medida que se necesita complejidad o capacidad del usuario para configurar detalles de su sistema suele quedarse pequeño.

Para programas complejos se recomienda usar otro tipo de librerías más avanzadas como PyQt, PyGTK o wxPython, todas ellas interfaces a librerías escritas en C/C++ llamadas Qt, GTK y wxWidgets respectivamente. Estas librerías aportan una visualización más elegante, en algunos casos usando widgets nativos del sistema operativo en el que funcionan.

[1] `https://www.python.org/dev/peps/pep-0008/`

Debido a la complejidad del ecosistema nace el proyecto PySimpleGUI, que pretende aunar las diferentes herramientas en una sola, sirviendo de interfaz a cualquiera de las anteriores y alguna otra. Además, el proyecto aporta gran cantidad de ejemplos de uso. PySimpleGUI aún está en desarrollo y el soporte de algunos de los motores no está terminado, pero es una fuente interesante de información y recursos.

10 Lo que has aprendido

Rescatando la definición de la introducción:

> Python es un lenguaje de programación de alto nivel orientado al uso general. Fue creado por Guido Van Rossum y publicado en 1991. La filosofía de python hace hincapié en la limpieza y la legibilidad del código fuente con una sintaxis que facilita expresar conceptos en menos líneas de código que en otros lenguajes.
>
> Python es un lenguaje de tipado dinámico y gestión de memoria automática. Soporta múltiples paradigmas de programación, incluyendo la programación orientada a objetos, imperativa, funcional y procedural e incluye una extensa librería estándar.

Ahora sí que estás en condición de entenderla no sólo para python sino para cualquier otro lenguaje que se te presente de este modo. Ahora tienes la habilidad de poder comprender de un vistazo qué te aporta el lenguaje que tienes delante únicamente leyendo su descripción.

Desgranándola poco a poco, has conocido la sintaxis de python en bastante detalle y has visto cómo hace uso de las sangrías para delimitar bloques de código, cosa que otros lenguajes hacen con llaves (`{}`) u otros símbolos.

La facilidad de expresar conceptos complejos en pocas líneas de código puede verse en las *list comprehensions*, la sentencia `with` y muchas otras estructuras del sistema. Python es un lenguaje elegante y directo, similar al lenguaje natural.

El tipado dinámico trata lo que estudiaste en el apartado sobre datos, donde se te cuenta que las referencias pueden cambiar de tipo en cualquier momento ya que son los propios valores los que son capaces de recordar qué tipo tienen.

La gestión de memoria automática también se presenta en el mismo apartado, contándote que python hace uso de un *garbage collector* o recolector de basura para limpiar de la memoria los datos que ya no usa.

Los diferentes paradigmas de programación no se han tratado de forma explícita en este documento, más allá de la programación orientada a objetos, que inunda python por completo. Sin embargo, el apartado sobre funciones adelanta varios de los conceptos básicos del paradigma de programación funcional: que las funciones sean ciudadanos de primera clase (*first-class citizens*), el uso de funciones anónimas (*lambda*) y las *closures*.

Los paradigmas procedural e imperativo son la base para los dos paradigmas de los que hemos hablado. La programación imperativa implica que se programa mediante órdenes (el caso de python, recuerda) en lugar de declaraciones (como puede ser la programación lógica,

donde se muestran un conjunto de normas que el programa debe cumplir). La programación procedural es un paradigma cuyo fundamento es el uso de bloques de código y su *scope*, creando funciones, estructuras de datos y variables aunque, a diferencia de la programación funcional, en la programación procedural no es necesario que las funciones sean ciudadanos de primera clase y pueden tener restricciones.

Estos dos últimos paradigmas, en realidad, se soportan casi por accidente al habilitar los dos anteriores.

En muchas ocasiones, te encontrarás escribiendo pequeñas herramientas y no necesitarás mucho más que usar las estructuras básicas de python y varias funciones para alterarlas, por lo que estarás pensando de forma procedural accidentalmente.

Los paradigmas no son más que patrones de diseño que nos permiten clasificar los lenguajes y sus filosofías, pero son muy interesantes a la hora de diseñar nuestras aplicaciones.

Además de todo esto, has tenido ocasión de conocer de forma superficial la librería estándar del lenguaje y un conjunto de librerías adicionales que te aportan los puntos de los que la librería estándar carece. Ahora sabes instalar dependencias y usarlas en entornos virtuales (*virtual environments*) para mantener limpia tu instalación.

A parte de lo mencionado en la definición del lenguaje, has aprendido a ejecutar, cargar y distribuir módulos de python, algo primordial si pretendes crear paquetes o nuevas librerías y usar las de terceros.

Con todo esto, tienes una visión general pero bastante detallada a nivel técnico de lo que python aporta y cómo. Lo que necesitas para compensarla es trabajar con él, acostumbrarte a su ecosistema y leer mucho código de buena calidad para acostumbrarte a seguir las convenciones y recetas habituales.

10.1 El código pythónico

A lo largo del documento se tratan temas que puede que no te esperases encontrar al leer sobre programación, ya que tu interés principal es resolver tus problemas de forma efectiva y construir aplicaciones. Hacer robots que te hagan la vida más fácil, en definitiva.

Sin embargo, quien se dedica a la programación tiene una vida muy ligada a la vida de quien se dedica a la filosofía o al diseño y es por eso que esas dos disciplinas aparecen de vez en cuando en cualquier conversación un poco seria sobre el trabajo con software.

Las tres disciplinas, en primer lugar, ocurren en la mente de las personas y no en sus manos. Es por eso que los patrones mentales y los modos de pensamiento son parte fundamental de ellas. Ninguno de ellos son trabajos para los que se pueda entrenar una memoria muscular. Es necesario pensar. Y es necesario pensar de forma consciente y premeditada.

Ver cómo desarrollan otras personas su actividad es valioso para realizar tu tarea con elegancia.

Otro detalle que has debido de observar, sobre todo porque acaba de aparecer, es la *elegancia*. La elegancia es subjetiva y depende del gusto de quien la mira. Sin embargo, esto sólo

es así hasta cierto punto. La realidad es que alguien puede considerar algo elegante y aun así no gustarle. Python es un ejemplo de algo así. Guste o no guste, python es un lenguaje de programación elegante, cuya elegancia forma parte primordial de la filosofía del lenguaje.

El autor de este documento, por ejemplo, no es un entusiasta de python, pero a lo largo de la travesía de escribir este documento ha podido reencontrarse, una vez más, con su elegancia.

Cuando se habla de código pythónico (*pythonic code*), se habla de un código que sigue los estándares de elegancia de python. Que es bonito, comprensible y claro. Un código que la comunidad de desarrollo de python aprobaría.

Cuando programes en python, trata de programar código pythónico, pues es esa la verdadera razón por la que se creó el lenguaje y es la forma en la que el lenguaje más fácil te lo va a poner.

Anexo I: Herramientas

IDLE es una herramienta de desarrollo muy limitada, suficiente para seguir los ejemplos que se recogen en este documento pero insuficiente para desarrollar aplicaciones avanzadas.

10.2 Desarrollo de código fuente

Existen gran cantidad de entornos de desarrollo (IDE) avanzados y editores que pueden ser recomendables, aunque es cuestión de gusto personal decantarse por uno de ellos o por otro.

La diferencia entre un entorno de desarrollo integrado y un editor es la siguiente: los entornos de desarrollo cumplen varias funciones adicionales, como, en el caso de IDLE, dar acceso a una REPL de python y la posibilidad de analizar las variables en memoria. Los editores únicamente sirven para escribir el código, aunque en muchos casos la línea que separa ambos conceptos es bastante borrosa: existen editores con funcionalidades avanzadas y entornos integrados muy sencillos que parecen un simple editor. Resumiendo, los entornos integrados de desarrollo, o IDE (*Integrated Development Environment*), tienen editores entre sus herramientas.

10.2.1 Entornos de desarrollo integrados

Quien escribe este documento no utiliza entornos de desarrollo avanzados (una cuestión de gusto personal), por lo que se le hace difícil recomendar alguno en particular. Sin embargo, el wiki de python recoge una larguísima lista de editores y entornos de desarrollo integrado interesantes[1].

En el entorno del análisis de datos, la distribución *Anaconda* es muy usada. Anaconda es más que un entorno de desarrollo integrado. Incluye un entorno de desarrollo llamado Spyder, una shell propia, un gestor de paquetes y dependencias propio llamado Conda, posibilidad de integración con el lenguaje de programación R y gran cantidad de paquetes instalados por defecto.

En otros entornos PyCharm es bastante común, aunque también son muy comunes los entornos de desarrollo integrados pensados para otros lenguajes que han comenzado a soportar python posteriormente como KDevelop, NetBeans y otros.

Te recomiendo que, si usas un entorno de desarrollo integrado en otros lenguajes, investigues si soporta python. De este modo no tendrás que aprender una nueva herramienta. Al

[1] `https://wiki.python.org/moin/PythonEditors`

ser un lenguaje tan común, probablemente lo soporte. Si no lo soporta prueba con un IDE que siga una filosofía similar al que uses.

10.2.2 Editores de código

Quien te escribe usa Vim, un editor de texto muy antiguo con muchas características que le hacen ser un editor muy eficiente. Existe gran variedad de editores de código que recomendar: Emacs, gEdit, Kate, Sublime, Atom... Todo dependerá de tus gustos personales.

La ventaja principal de los editores de código es que conociendo uno en profundidad es más que suficiente para cualquier lenguaje, ya que están diseñados únicamente para escribir el contenido de tus programas, dejando las peculiaridades de ejecución de cada lenguaje a parte. Esto les aporta una ligereza difícilmente alcanzable por los IDEs.

Sin embargo, esta virtud también es su mayor defecto. Al no integrar ninguna herramienta adicional, es necesario trabajar todo manualmente. En el caso de python, te fuerzan a usar una shell independiente y a interactuar con ella de forma manual. Al principio puede ser tedioso, pero aprender a gestionar los detalles manualmente es interesante ya que te permite obtener un gran conocimiento del sistema y lenguaje en el que trabajas.

10.3 Herramientas de depuración

El propio diseño de python permite que sea fácilmente depurable. La REPL facilita que se pruebe la aplicación a medida que se va desarrollando, función a función, para asegurar que su comportamiento es el correcto. Además, la capacidad introspectiva del lenguaje es una buena herramienta, en conjunción con la REPL, para comprobar el comportamiento.

Además de estas características, la instalación de python viene acompañada del programa `pdb` (*Python Debugger*), un depurador de código similar al conocido GNU-Debugger. Existen otros depuradores de código más amigables que este, pero la realidad es que no suelen ser necesarios.

10.4 Testeo de aplicaciones

Hoy en día el testeo de software es muy común, en parte gracias al desarrollo guiado por pruebas, o TDD (*Test Driven Development*).

La librería estándar de python incluye un módulo para pruebas unitarias llamado `unittest` en que la librería Nose, muy conocida y usada, se basa para facilitar el trabajo de modo similar a lo que ocurre con Requests y `urllib`.

Por supuesto, existen otras alternativas, pero estas son las principales.

Anexo II: Licencia CC BY-SA 4.0

Creative Commons Atribución/Reconocimiento-CompartirIgual 4.0 Licencia Pública Internacional

Al ejercer los Derechos Licenciados (definidos a continuación), Usted acepta y acuerda estar obligado por los términos y condiciones de esta Licencia Internacional Pública de Atribución/Reconocimiento-CompartirIgual 4.0 de Creative Commons ("Licencia Pública"). En la medida en que esta Licencia Pública pueda ser interpretada como un contrato, a Usted se le otorgan los Derechos Licenciados en consideración a su aceptación de estos términos y condiciones, y el Licenciante le concede a Usted tales derechos en consideración a los beneficios que el Licenciante recibe por poner a disposición el Material Licenciado bajo estos términos y condiciones.

Sección 1 – Definiciones.

a. **Material Adaptado** es aquel material protegido por Derechos de Autor y Derechos Similares que se deriva o se crea en base al Material Licenciado y en el cual el Material Licenciado se traduce, altera, arregla, transforma o modifica de manera tal que dicho resultado sea de aquellos que requieran autorización de acuerdo con los Derechos de Autor y Derechos Similares que ostenta el Licenciante. A los efectos de esta Licencia Pública, cuando el Material Licenciado se trate de una obra musical, una interpretación o una grabación sonora, la sincronización temporal de este material con una imagen en movimiento siempre producirá Material Adaptado.

b. **Licencia de adaptador** es aquella licencia que Usted aplica a Sus Derechos de Autor y Derechos Similares en Sus contribuciones consideradas como Material Adaptado de acuerdo con los términos y condiciones de esta Licencia Pública.

c. **Una Licencia Compatible con BY-SA** es aquella que aparece en la lista disponible en creativecommons.org/compatiblelicenses[2], aprobada por Creative Commons, como una licencia esencialmente equivalente a esta Licencia Pública.

d. **Derechos de Autor y Derechos Similares** son todos aquellos derechos estrechamente vinculados a los derechos de autor, incluidos, de manera enunciativa y no taxativa, los derechos sobre las interpretaciones, las emisiones, las grabaciones sonoras y los Derechos "Sui Generis" sobre Bases de Datos, sin importar cómo estos derechos se encuentren enunciados o categorizados. A los efectos de esta Licencia Pública, los derechos especificados en las secciones 2(b)(1)-(2) no se consideran Derechos de Autor y Derechos Similares.

[2] `https://creativecommons.org/compatiblelicenses`

e. **Medidas Tecnológicas Efectivas** son aquellas medidas que, en ausencia de la debida autorización, no pueden ser eludidas en virtud de las leyes que cumplen las obligaciones del artículo 11 del Tratado de la OMPI sobre Derecho de Autor adoptado el 20 de diciembre de 1996, y/o acuerdos internacionales similares.

f. **Excepciones y Limitaciones** son el uso justo (fair use), el trato justo (fair dealing) y/o cualquier otra excepción o limitación a los Derechos de Autor y Derechos Similares que se apliquen al uso el Material Licenciado.

g. **Elementos de la Licencia** son los atributos que figuran en el nombre de la Licencia Pública de Creative Commons. Los Elementos de la Licencia de esta Licencia Pública son Atribución/Reconocimiento y CompartirIgual.

h. **Material Licenciado** es obra artística o literaria, base de datos o cualquier otro material al cual el Licenciante aplicó esta Licencia Pública.

i. **Derechos Licenciados** son derechos otorgados a Usted bajo los términos y condiciones de esta Licencia Pública, los cuales se limitan a todos los Derechos de Autor y Derechos Similares que apliquen al uso del Material Licenciado y que el Licenciante tiene potestad legal para licenciar.

j. **Licenciante** es el individuo(s) o la entidad(es) que concede derechos bajo esta Licencia Pública.

k. **Compartir** significa proporcionar material al público por cualquier medio o procedimiento que requiera permiso conforme a los Derechos Licenciados, tales como la reproducción, exhibición pública, presentación pública, distribución, difusión, comunicación o importación, así como también su puesta a disposición, incluyendo formas en que el público pueda acceder al material desde un lugar y momento elegido individualmente por ellos.

l. **Derechos "Sui Generis" sobre Bases de Datos** son aquellos derechos diferentes a los derechos de autor, resultantes de la Directiva 96/9/EC del Parlamento Europeo y del Consejo, de 11 de marzo de 1996 sobre la protección jurídica de las bases de datos, en sus versiones modificadas y/o posteriores, así como otros derechos esencialmente equivalentes en cualquier otra parte del mundo.

m. **Usted** es el individuo o la entidad que ejerce los Derechos Licenciados en esta Licencia Pública. La palabra **Su** tiene un significado equivalente.

Sección 2 – Ámbito de Aplicación.

a. **Otorgamiento de la licencia**.

1. Sujeto a los términos y condiciones de esta Licencia Pública, el Licenciante le otorga a Usted una licencia de carácter global, gratuita, no transferible a terceros, no exclusiva e irrevocable para ejercer los Derechos Licenciados sobre el Material Licenciado para:
   A. reproducir y Compartir el Material Licenciado, en su totalidad o en parte; y
   B. producir, reproducir y Compartir Material Adaptado.

2. Excepciones y Limitaciones. Para evitar cualquier duda, donde se apliquen Excepciones y Limitaciones al uso del Material Licenciado, esta Licencia Pública no será aplicable, y Usted no tendrá necesidad de cumplir con sus términos y condiciones.
3. Vigencia. La vigencia de esta Licencia Pública está especificada en la sección 6(a).
4. Medios y formatos; modificaciones técnicas permitidas. El Licenciante le autoriza a Usted a ejercer los Derechos Licenciados en todos los medios y formatos, actualmente conocidos o por crearse en el futuro, y a realizar las modificaciones técnicas necesarias para ello. El Licenciante renuncia y/o se compromete a no hacer valer cualquier derecho o potestad para prohibirle a Usted realizar las modificaciones técnicas necesarias para ejercer los Derechos Licenciados, incluyendo las modificaciones técnicas necesarias para eludir las Medidas Tecnológicas Efectivas. A los efectos de esta Licencia Pública, la mera realización de modificaciones autorizadas por esta sección 2(a)(4) nunca produce Material Adaptado.
5. Receptores posteriores.
   A. Oferta del Licenciante – Material Licenciado. Cada receptor de Material Licenciado recibe automáticamente una oferta del Licenciante para ejercer los Derechos Licenciados bajo los términos y condiciones de esta Licencia Pública.
   B. Oferta adicional por parte del Licenciante – Material Adaptado. Cada receptor del Material Adaptado por Usted recibe automáticamente una oferta del Licenciante para ejercer los Derechos Licenciados en el Material Adaptado bajo las condiciones de la Licencia del Adaptador que Usted aplique.
   C. Sin restricciones a receptores posteriores. Usted no puede ofrecer o imponer ningún término ni condición diferente o adicional, ni puede aplicar ninguna Medida Tecnológica Efectiva al Material Licenciado si haciéndolo restringe el ejercicio de los Derechos Licenciados a cualquier receptor del Material Licenciado.
6. Sin endoso. Nada de lo contenido en esta Licencia Pública constituye o puede interpretarse como un permiso para afirmar o implicar que Usted, o que Su uso del Material Licenciado, está conectado, patrocinado, respaldado o reconocido con estatus oficial por el Licenciante u otros designados para recibir la Atribución/Reconocimiento según lo dispuesto en la sección 3(a)(1)(A)(i).

b. **Otros derechos**.

1. Los derechos morales, tales como el derecho a la integridad, no están comprendidos bajo esta Licencia Pública ni tampoco los derechos de publicidad y privacidad ni otros derechos personales similares. Sin embargo, en la medida de lo posible, el Licenciante renuncia y/o se compromete a no hacer valer ninguno de estos derechos que ostenta como Licenciante, limitándose a lo necesario para que Usted pueda ejercer los Derechos Licenciados, pero no de otra manera.
2. Los derechos de patentes y marcas no son objeto de esta Licencia Pública.
3. En la medida de lo posible, el Licenciante renuncia al derecho de cobrarle regalías a Usted por el ejercicio de los Derechos Licenciados, ya sea directamente o a través de una entidad de gestión colectiva bajo cualquier esquema de licenciamiento voluntario, renunciable o no renunciable. En todos los demás casos, el Licenciante se reserva expresamente cualquier derecho de cobrar esas regalías.

Sección 3 – Condiciones de la Licencia.

Su ejercicio de los Derechos Licenciados está expresamente sujeto a las condiciones siguientes.

a. **Atribución/Reconocimiento**.

1. Si Usted comparte el Material Licenciado (incluyendo en forma modificada), Usted debe:

   A. Conservar lo siguiente si es facilitado por el Licenciante con el Material Licenciado:

   i. identificación del creador o los creadores del Material Licenciado y de cualquier otra persona designada para recibir Atribución/Reconocimiento, de cualquier manera razonable solicitada por el Licenciante (incluyendo por seudónimo si este ha sido designado);
   ii. un aviso sobre derecho de autor;
   iii. un aviso que se refiera a esta Licencia Pública;
   iv. un aviso que se refiera a la limitación de garantías;
   v. un URI o un hipervínculo al Material Licenciado en la medida razonablemente posible;

   B. Indicar si Usted modificó el Material Licenciado y conservar una indicación de las modificaciones anteriores; e C. Indicar que el Material Licenciado está bajo esta Licencia Pública, e incluir el texto, el URI o el hipervínculo a esta Licencia Pública.

2. Usted puede satisfacer las condiciones de la sección 3(a)(1) de cualquier forma razonable según el medio, las maneras y el contexto en los cuales Usted Comparta el Material Licenciado. Por ejemplo, puede ser razonable satisfacer las condiciones facilitando un URI o un hipervínculo a un recurso que incluya la información requerida.

3. Bajo requerimiento del Licenciante, Usted debe eliminar cualquier información requerida por la sección 3(a)(1)(A) en la medida razonablemente posible.

b. **CompartirIgual**.

   Además de las condiciones de la sección 3(a), si Usted Comparte Material Adaptado producido por Usted, también aplican las condiciones siguientes.

1. La Licencia del Adaptador que Usted aplique debe ser una licencia de Creative Commons con los mismos Elementos de la Licencia, ya sea de esta versión o una posterior, o una Licencia Compatible con la BY-SA.
2. Usted debe incluir el texto, el URI o el hipervínculo a la Licencia del Adaptador que aplique. Usted puede satisfacer esta condición de cualquier forma razonable según el medio, las maneras y el contexto en los cuales Usted Comparta el Material Adaptado.
3. Usted no puede ofrecer o imponer ningún término o condición adicional o diferente, o aplicar ninguna Medida Tecnológica Efectiva al Material Adaptado que

restrinja el ejercicio de los derechos concedidos en virtud de la Licencia de Adaptador que Usted aplique.

Sección 4 – Derechos "Sui Generis" sobre Bases de Datos.

Cuando los Derechos Licenciados incluyan Derechos "Sui Generis" sobre Bases de Datos que apliquen a Su uso del Material Licenciado:

a. para evitar cualquier duda, la sección 2(a)(1) le concede a Usted el derecho a extraer, reutilizar, reproducir y Compartir todo o una parte sustancial de los contenidos de la base de datos;
b. si Usted incluye la totalidad o una parte sustancial del contenido de una base de datos en otra sobre la cual Usted ostenta Derecho "Sui Generis" sobre Bases de Datos, entonces ella (pero no sus contenidos individuales) se entenderá como Material Adaptado para efectos de la sección 3(b); y
c. Usted debe cumplir con las condiciones de la sección 3(a) si Usted Comparte la totalidad o una parte sustancial de los contenidos de la base de datos.

Para evitar dudas, esta sección 4 complementa y no sustituye Sus obligaciones bajo esta Licencia Pública cuando los Derechos Licenciados incluyen otros Derechos de Autor y Derechos Similares.

Sección 5 – Exención de Garantías y Limitación de Responsabilidad.

a. **Salvo que el Licenciante se haya comprometido mediante un acuerdo por separado, en la medida de lo posible el Licenciante ofrece el Material Licenciado tal como es y tal como está disponible y no se hace responsable ni ofrece garantías de ningún tipo respecto al Material Licenciado, ya sea de manera expresa, implícita, legal u otra. Esto incluye, de manera no taxativa, las garantías de título, comerciabilidad, idoneidad para un propósito en particular, no infracción, ausencia de vicios ocultos u otros defectos, la exactitud, la presencia o la ausencia de errores, sean o no conocidos o detectables. Cuando no se permita, totalmente o en parte, la declaración de ausencia de garantías, a Usted puede no aplicársele esta exclusión.**
b. **En la medida de lo posible, en ningún caso el Licenciante será responsable ante Usted por ninguna teoría legal (incluyendo, de manera no taxativa, la negligencia) o de otra manera por cualquier pérdida, coste, gasto o daño directo, especial, indirecto, incidental, consecuente, punitivo, ejemplar u otro que surja de esta Licencia Pública o del uso del Material Licenciado, incluso cuando el Licenciante haya sido advertido de la posibilidad de tales pérdidas, costes, gastos o daños. Cuando no se permita la limitación de responsabilidad, ya sea totalmente o en parte, a Usted puede no aplicársele esta limitación.**
c. La renuncia de garantías y la limitación de responsabilidad descritas anteriormente deberán ser interpretadas, en la medida de lo posible, como lo más próximo a una exención y renuncia absoluta a todo tipo de responsabilidad.

Sección 6 – Vigencia y Terminación.

a. Esta Licencia Pública tiene una vigencia de aplicación igual al plazo de protección de los Derechos de Autor y Derechos Similares licenciados aquí. Sin embargo, si Usted incumple las condiciones de esta Licencia Pública, los derechos que se le conceden mediante esta Licencia Pública terminan automáticamente.

b. En aquellos casos en que Su derecho a utilizar el Material Licenciado se haya terminado conforme a la sección 6(a), este será restablecido:

    1. automáticamente a partir de la fecha en que la violación sea subsanada, siempre y cuando esta se subsane dentro de los 30 días siguientes a partir de Su descubrimiento de la violación; o
    2. tras el restablecimiento expreso por parte del Licenciante.

    Para evitar dudas, esta sección 6(b) no afecta ningún derecho que pueda tener el Licenciante a buscar resarcimiento por Sus violaciones de esta Licencia Pública.

c. Para evitar dudas, el Licenciante también puede ofrecer el Material Licenciado bajo términos o condiciones diferentes, o dejar de distribuir el Material Licenciado en cualquier momento; sin embargo, hacer esto no pondrá fin a esta Licencia Pública.

d. Las secciones 1, 5, 6, 7, y 8 permanecerán vigentes a la terminación de esta Licencia Pública.

Sección 7 – Otros Términos y Condiciones.

a. El Licenciante no estará obligado por ningún término o condición adicional o diferente que Usted le comunique a menos que se acuerde expresamente.
b. Cualquier arreglo, convenio o acuerdo en relación con el Material Licenciado que no se indique en este documento se considera separado e independiente de los términos y condiciones de esta Licencia Pública.

Sección 8 – Interpretación.

a. Para evitar dudas, esta Licencia Pública no es ni deberá interpretarse como una reducción, limitación, restricción, o una imposición de condiciones al uso de Material Licenciado que legalmente pueda realizarse sin permiso del titular, más allá de lo contemplado en esta Licencia Pública.
b. En la medida de lo posible, si alguna disposición de esta Licencia Pública se considera inaplicable, esta será automáticamente modificada en la medida mínima necesaria para hacerla aplicable. Si la disposición no puede ser reformada, deberá ser eliminada de esta Licencia Pública sin afectar la exigibilidad de los términos y condiciones restantes.
c. No se podrá renunciar a ningún término o condición de esta Licencia Pública, ni se consentirá ningún incumplimiento, a menos que se acuerde expresamente con el Licenciante.
d. Nada en esta Licencia Pública constituye ni puede ser interpretado como una limitación o una renuncia a los privilegios e inmunidades que aplican al Licenciante o a Usted, incluyendo aquellos surgidos a partir de procesos legales de cualquier jurisdicción o autoridad.

ElenQ
TECHNOLOGY