Capítulo 2

Una calculadora avanzada

> *—¿Sabes sumar? —le preguntó la Reina Blanca.— ¿Cuánto es uno más uno más uno más uno más uno más uno más uno más uno más uno más uno?*
> *—No lo sé —dijo Alicia—. Perdí la cuenta.*
> *—No sabe hacer una adición —le interrumpió la Reina Roja.*
>
> Lewis Carroll, *Alicia a través del espejo*.

El objetivo de este tema es que te familiarices con el *entorno interactivo* de Python, que aprendas a construir *expresiones aritméticas* almacenando los resultados en *variables* mediante *asignaciones* y que conozcas los *tipos de datos* básicos del lenguaje de programación Python.

2.1. Sesiones interactivas

Cuando programamos utilizamos un conjunto de herramientas al que denominamos *entorno de programación*. Entre estas herramientas tenemos editores de texto (que nos permiten escribir programas), compiladores o intérpretes (que traducen los programas a código de máquina), depuradores (que ayudan a detectar errores), analizadores de tiempo de ejecución (para estudiar la eficiencia de los programas), etc.

Los lenguajes interpretados suelen ofrecer una herramienta de ejecución *interactiva*. Con ella es posible dar órdenes directamente al intérprete y obtener una respuesta inmediata para cada una de ellas. Es decir, no es necesario escribir un programa completo para empezar a obtener resultados de ejecución, sino que podemos «dialogar» con el intérprete de nuestro lenguaje de programación: le pedimos que ejecute una orden y nos responde con su resultado. El entorno interactivo es de gran ayuda para experimentar con fragmentos de programa antes de incluirlos en una versión definitiva. En esta sección veremos cómo realizar sesiones de trabajo interactivo con Python.[1]

Si hemos instalado Python en nuestro ordenador podemos iniciar una sesión de trabajo escribiendo `python` en la línea de órdenes Unix.[2] El sistema nos responderá dando un mensaje informativo sobre la versión de Python que estamos utilizando (y cuándo fue compilada, con qué compilador, etc.) y, a continuación, mostrará el *prompt*:

```
$ python↲
Python 2.3 (#1, Aug 2 2003, 12:14:49)
```

[1] Abusando del lenguaje, llamaremos indistintamente Python al *entorno de programación*, al *intérprete del lenguaje* y al propio *lenguaje de programación*.

[2] En el entorno Microsoft Windows puedes arrancar el intérprete interactivo activando el icono correspondiente (que es una caricatura de una serpiente pitón) o seleccionando el programa desde el menú «Inicio».

```
[GCC 3.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

El *prompt* es la serie de caracteres «**>>>**» que aparece en la última línea. El *prompt* indica que el intérprete de Python espera que nosotros introduzcamos una orden utilizando el teclado.

> ***La orden Unix*** `python`
>
> Hemos invocado el entorno interactivo escribiendo **python** en la línea de órdenes Unix y pulsando el retorno de carro. Al hacer esto, el entorno de ejecución de órdenes Unix (al que se suele denominar *shell*) busca en el ordenador una aplicación llamada **python** y la ejecuta. Esa aplicación es un programa que lee una línea introducida por teclado, la interpreta como si fuera un fragmento de programa Python y muestra por pantalla el resultado obtenido. (En realidad hace más cosas. Ya las iremos viendo.)
>
> Por regla general, la aplicación **python** reside en **/usr/local/bin/** o en **/usr/bin/**. Son directorios donde normalmente el *shell* busca programas. Si al escribir **python** y dar al retorno de carro no arranca el intérprete, asegúrate de que está instalado el entorno Python. Si es así, puedes intentar ejecutar el entorno dando la ruta completa hasta el programa: por ejemplo **/usr/local/bin/python**.

Escribamos una expresión aritmética, por ejemplo «2**+**2», y pulsemos la tecla de retorno de carro. Cuando mostremos sesiones interactivas destacaremos el texto que teclea el usuario con texto sobre fondo gris representaremos con el símbolo « ↲ » la pulsación de la tecla de retorno de carro. Python *evalúa* la expresión (es decir, obtiene su resultado) y responde mostrando el resultado por pantalla.

```
$ python↲
Python 2.3 (#1, Aug 2 2003, 12:14:49)
[GCC 3.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> 2+2↲
4
>>>
```

La última línea es, nuevamente, el *prompt*: Python acabó de ejecutar la última orden (evaluar una expresión y mostrar el resultado) y nos pide que introduzcamos una nueva orden.

Si deseamos acabar la sesión interactiva y salir del intérprete Python, debemos introducir una marca de final de fichero, que en Unix se indica pulsando la tecla de control y, sin soltarla, también la tecla **d**. (De ahora en adelante representaremos una combinación de teclas como la descrita así: **C-d**.)

2.1.1. Los operadores aritméticos

Las *operaciones* de suma y resta, por ejemplo, se denotan con los símbolos u *operadores* **+** y **-**, respectivamente, y operan sobre dos valores numéricos (los *operandos*). Probemos algunas expresiones formadas con estos dos operadores:

```
>>> 1 + 2 ↲
3
>>> 1 + 2 + 3 ↲
6
>>> 1 - 2 + 3 ↲
2
```

Final de fichero

La «marca de final de fichero» indica que un fichero ha terminado. ¡Pero nosotros no trabajamos con un fichero, sino con el teclado! En realidad, el ordenador considera al teclado como un fichero. Cuando deseamos «cerrar el teclado» para una aplicación, enviamos una marca de final de fichero desde el teclado. En Unix, la marca de final de fichero se envía pulsando `C-d`; en MS-DOS o Microsoft Windows, pulsando `C-z`.

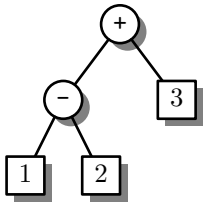
Existe otro modo de finalizar la sesión; escribe

```
>>> from sys import exit ↲
>>> exit() ↲
```

En inglés, «exit» significa «salir». Sí pero, ¿qué significa «**from** *sys* **import** *exit*» y por qué hay un par de paréntesis detrás de la palabra «*exit*» en la segunda línea? Más adelante lo averiguaremos.

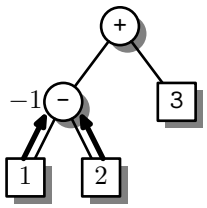
Observa que puedes introducir varias operaciones en una misma línea o expresión. El orden en que se efectúan las operaciones es (en principio) de izquierda a derecha. La expresión 1 **-** 2 **+** 3, por ejemplo, equivale matemáticamente a $((1 - 2) + 3)$; por ello decimos que la suma y la resta son operadores *asociativos por la izquierda*.

Podemos representar gráficamente el orden de aplicación de las operaciones utilizando *árboles sintácticos*. Un árbol sintáctico es una representación gráfica en la que disponemos los operadores y los operandos como nodos y en los que cada operador está conectado a sus operandos. El árbol sintáctico de la expresión «1 **-** 2 **+** 3» es éste:

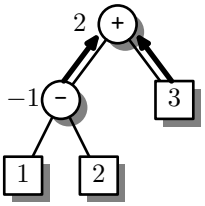


El nodo superior de un árbol recibe el nombre de *nodo raíz*. Los nodos etiquetados con operadores (representados con círculos) se denominan *nodos interiores*. Los nodos interiores tienen uno o más *nodos hijo* o *descendientes* (de los que ellos son sus respectivos *nodos padre* o *ascendientes*). Dichos nodos son nodos raíz de otros (sub)árboles sintácticos (¡la definición de árbol sintáctico es auto-referencial!). Los valores resultantes de evaluar las expresiones asociadas a dichos (sub)árboles constituyen los operandos de la operación que representa el nodo interior. Los nodos sin descendientes se denominan *nodos terminales* u *hojas* (representados con cuadrados) y corresponden a valores numéricos.

La evaluación de cada operación individual en el árbol sintáctico «fluye» de las hojas hacia la raíz (el nodo superior); es decir, en primer lugar se evalúa la subexpresión «1 **-** 2», que corresponde al subárbol más profundo. El resultado de la evaluación es −1:



A continuación se evalúa la subexpresión que suma el resultado de evaluar «1 **-** 2» al valor 3:

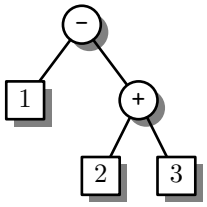


Así se obtiene el resultado final: el valor 2.

Si deseamos calcular (1−(2+3)) podemos hacerlo añadiendo paréntesis a la expresión aritmética:

```
>>> 1 - (2 + 3) ↲
-4
```

El árbol sintáctico de esta nueva expresión es

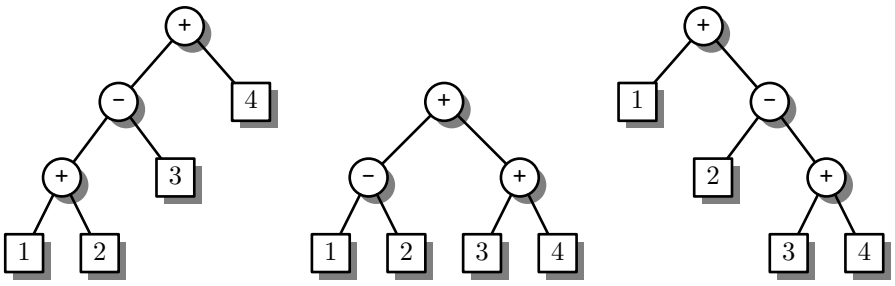


En este nuevo árbol, la primera subexpresión evaluada es la que corresponde al subárbol derecho.

Observa que en el árbol sintáctico no aparecen los paréntesis de la expresión. El árbol sintáctico ya indica el orden en que se procesan las diferentes operaciones y no necesita paréntesis. La expresión Python, sin embargo, *necesita* los paréntesis para indicar ese mismo orden de evaluación.

EJERCICIOS

▶ 12 ¿Qué expresiones Python permiten, utilizando el menor número posible de paréntesis, efectuar *en el mismo orden* los cálculos representados con estos árboles sintácticos?



▶ 13 Dibuja los árboles sintácticos correspondientes a las siguientes expresiones aritméticas:

a) 1 + 2 + 3 + 4 b) 1 − 2 − 3 − 4 c) 1 − (2 − (3 − 4) + 1)

Los operadores de suma y resta son *binarios*, es decir, operan sobre dos operandos. El mismo símbolo que se usa para la resta se usa también para un operador *unario*, es decir, un operador que actúa sobre un único operando: el de cambio de signo, que devuelve el valor de su operando cambiado de signo. He aquí algunos ejemplos:

Espacios en blanco

Parece que se puede hacer un uso bastante liberal de los espacios en blanco en una expresión.

```
>>> 10 + 20 + 30 ↲
60
>>> 10+20+30 ↲
60
>>> 10    +20   + 30 ↲
60
>>> 10+ 20+30 ↲
60
```

Es así. Has de respetar, no obstante, unas reglas sencillas:

- No puedes poner espacios en medio de un número.

  ```
  >>> 10 + 2   0 + 30 ↲
  ```

  Los espacios en blanco entre el 2 y el 0 hacen que Python no lea el número 20, sino el número 2 seguido del número 0 (lo cual es un error, pues no hay operación alguna entre ambos números).

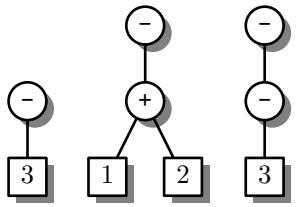
- No puedes poner espacios al principio de la expresión.

  ```
  >>>       10 + 20 + 30 ↲
  ```

  Los espacios en blanco entre el *prompt* y el 10 provocan un error. Aún es pronto para que conozcas la razón.

```
>>> -3 ↲
-3
>>> -(1 + 2) ↲
-3
>>> --3 ↲
3
```

He aquí los árboles sintácticos correspondientes a las tres expresiones del ejemplo:



Existe otro operador unario que se representa con el símbolo **+**: el operador *identidad*. El operador identidad no hace nada «útil»: proporciona como resultado el mismo número que se le pasa.

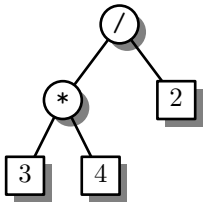
```
>>> +3 ↲
3
>>> +-3 ↲
-3
```

El operador identidad sólo sirve para, en ocasiones, poner énfasis en que un número es positivo. (El ordenador considera tan positivo el número 3 como el resultado de evaluar +3.)

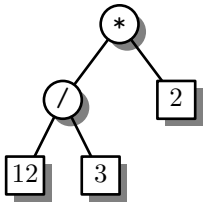
Los operadores de multiplicación y división son, respectivamente, * y /:

```
>>> 2 * 3↲
6
>>> 4 / 2↲
2
>>> 3 * 4 / 2↲
6
>>> 12 / 3 * 2↲
8
```

Observa que estos operadores también son asociativos por la izquierda: la expresión «3 * 4 / 2» equivale a $((3 \cdot 4)/2) = \frac{3 \cdot 4}{2}$, es decir, tiene el siguiente árbol sintáctico:



y la expresión `12 / 3 * 2` equivale a $((12/3) \cdot 2) = \frac{12}{3} \cdot 2$, o sea, su árbol sintáctico es:

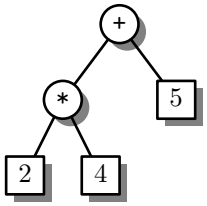


¿Qué pasa si combinamos en una misma expresión operadores de suma o resta con operadores de multiplicación o división? Fíjate en que la regla de aplicación de operadores de izquierda a derecha no siempre se observa:

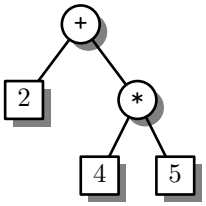
```
>>> 2 * 4 + 5↲
13
>>> 2 + 4 * 5↲
22
```

En la segunda expresión, primero se ha efectuado el producto 4 * 5 y el resultado se ha sumado al valor 2. Ocurre que los operadores de multiplicación y división son *prioritarios* frente a los de suma y resta. Decimos que la multiplicación y la división tienen *mayor nivel de precedencia* o *prioridad* que la suma y la resta.

El árbol sintáctico de 2 * 4 + 5 es:



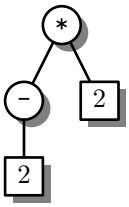
y el de 2 + 4 * 5 es:

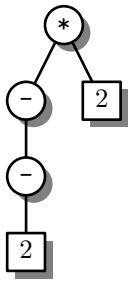


Pero, ¡atención!, el cambio de signo tiene mayor prioridad que la multiplicación y la división:

```
>>> -2 * 2 ↲
-4
>>> --2 * 2 ↲
4
```

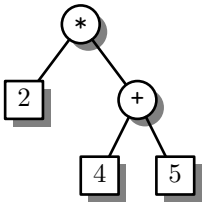
Los árboles sintácticos correspondientes a estas dos expresiones son, respectivamente:





Si los operadores siguen unas *reglas de precedencia* que determinan su orden de aplicación, ¿qué hacer cuando deseamos un orden de aplicación distinto? Usar paréntesis, como hacemos con la notación matemática convencional.

La expresión `2 * (4 + 5)`, por ejemplo, presenta este árbol sintáctico:



Comprobémoslo con el intérprete:

```
>>> 2 * (4 + 5) ↲
18
```

Existen más operadores en Python. Tenemos, por ejemplo, el operador módulo, que se denota con el símbolo de porcentaje **%** (aunque nada tiene que ver con el cálculo de porcentajes). El operador módulo devuelve el resto de la división entera entre dos operandos.

```
>>> 27 % 5 ↲
2
>>> 25 % 5 ↲
0
```

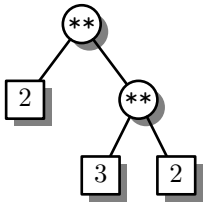
El operador **%** también es asociativo por la izquierda y su prioridad es la misma que la de la multiplicación o la división.

El último operador que vamos a estudiar es la exponenciación, que se denota con dos asteriscos juntos, no separados por ningún espacio en blanco: ******.

Lo que en notación matemática convencional expresamos como 2^3 se expresa en Python con `2 ** 3`.

```
>>> 2 ** 3 ↲
8
```

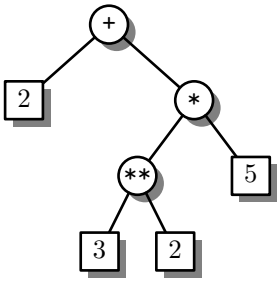
Pero, ¡ojo!, la exponenciación es *asociativa por la derecha*. La expresión `2 ** 3 ** 2` equivale a $2^{(3^2)} = 2^9 = 512$, y no a $(2^3)^2 = 8^2 = 64$, o sea, su árbol sintáctico es:



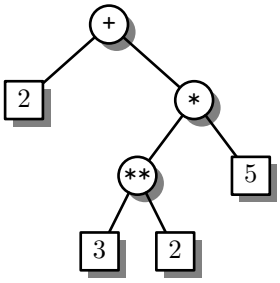
Por otra parte, la exponenciación tiene mayor precedencia que cualquiera de los otros operadores presentados.

He aquí varias expresiones evaluadas con Python y sus correspondientes árboles sintácticos. Estúdialos con atención:

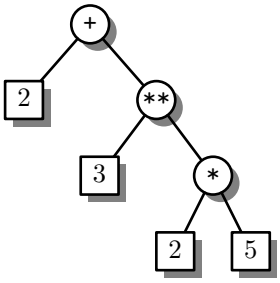
```
>>> 2 + 3 ** 2 * 5 ↲
47
```



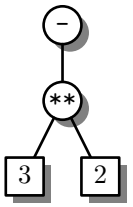

```
>>> 2 + ((3 ** 2) * 5) ↲
47
```



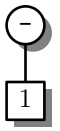

```
>>> 2 + 3 ** (2 * 5) ↲
59051
```




```
>>> -3 ** 2 ↲
-9
```



```
>>> -1 ↲
-1
```



La tabla 2.1 resume las características de los operadores Python: su aridad (número de operandos), asociatividad y precedencia.

Operación	Operador	Aridad	Asociatividad	Precedencia
Exponenciación	**	Binario	Por la derecha	1
Identidad	+	Unario	—	2
Cambio de signo	-	Unario	—	2
Multiplicación	*	Binario	Por la izquierda	3
División	/	Binario	Por la izquierda	3
Módulo (o resto)	%	Binario	Por la izquierda	3
Suma	+	Binario	Por la izquierda	4
Resta	-	Binario	Por la izquierda	4

Tabla 2.1: Operadores para expresiones aritméticas. El nivel de precedencia 1 es el de mayor prioridad y el 4 el de menor.

EJERCICIOS

▶ 14 ¿Qué resultados se obtendrán al evaluar las siguientes expresiones Python? Dibuja el árbol sintáctico de cada una de ellas, calcula a mano el valor resultante de cada expresión y comprueba, con la ayuda del ordenador, si tu resultado es correcto.

a) `2 + 3 + 1 + 2`
b) `2 + 3 * 1 + 2`
c) `(2 + 3) * 1 + 2`
d) `(2 + 3) * (1 + 2)`
e) `+---6`
f) `-+-+6`

▶ 15 Traduce las siguientes expresiones matemáticas a Python y evalúalas. Trata de utilizar el menor número de paréntesis posible.

a) $2 + (3 \cdot (6/2))$
b) $\dfrac{4+6}{2+3}$
c) $(4/2)^5$
d) $(4/2)^{5+1}$
e) $(-3)^2$
f) $-(3^2)$

(Nota: El resultado de evaluar cada expresión es: a) 11; b) 2; c) 32; d) 64; e) 9; f) −9.)

2.1.2. Errores de tecleo y excepciones

Cuando introducimos una expresión y damos la orden de evaluarla, es posible que nos equivoquemos. Si hemos formado incorrectamente una expresión, Python nos lo indicará con un *mensaje de error*. El mensaje de error proporciona información acerca del tipo de error cometido y del lugar en el que éste ha sido detectado. Aquí tienes una expresión errónea y el mensaje de error correspondiente:

```
>>> 1 + 2) ↲
  File "<stdin>", line 1
    1 + 2)
```

```
        ^
SyntaxError: invalid syntax
```

En este ejemplo hemos cerrado un paréntesis cuando no había otro abierto previamente, lo cual es incorrecto. Python nos indica que ha detectado un *error de sintaxis* (*SyntaxError*) y «apunta» con una flecha (el carácter `^`) al lugar en el que se encuentra. (El texto «**`File "<stdin>", line 1`**» indica que el error se ha producido al leer de teclado, esto es, de la *entrada estándar* —**`stdin`** es una abreviatura del inglés «standard input», que se traduce por «entrada estándar»—.)

En Python los errores se denominan *excepciones*. Cuando Python es incapaz de analizar una expresión, produce una excepción. Cuando el intérprete interactivo detecta la excepción, nos muestra por pantalla un mensaje de error.

Veamos algunos otros errores y los mensajes que produce Python.

```
>>> 1 + * 3↲
  File "<stdin>", line 1
    1 + * 3
        ^
SyntaxError: invalid syntax
>>> 2 + 3 %↲
  File "<stdin>", line 1
    2 + 3 %
          ^
SyntaxError: invalid syntax
>>> 1 / 0↲
Traceback (innermost last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo
```

En el ejemplo, el último error es de naturaleza distinta a los anteriores (no hay un carácter `^` apuntando a lugar alguno): se trata de un *error de división por cero* (*ZeroDivisionError*), cuando los otros eran *errores sintácticos* (*SyntaxError*). La cantidad que resulta de dividir por cero no está definida y Python es incapaz de calcular un valor como resultado de la expresión `1 / 0`. No es un error sintáctico porque la expresión está sintácticamente bien formada: el operador de división tiene dos operandos, como toca.

Edición avanzada en el entorno interactivo

Cuando estemos escribiendo una expresión puede que cometamos errores y los detectemos antes de solicitar su evaluación. Aún estaremos a tiempo de corregirlos. La tecla de borrado, por ejemplo, elimina el carácter que se encuentra a la izquierda del cursor. Puedes desplazar el cursor a cualquier punto de la línea que estás editando utilizando las teclas de desplazamiento del cursor a izquierda y a derecha. El texto que teclees se insertará siempre justo a la izquierda del cursor.

Hasta el momento hemos tenido que teclear desde cero cada expresión evaluada, aun cuando muchas se parecían bastante entre sí. Podemos teclear menos si aprendemos a utilizar algunas funciones de edición avanzadas.

Lo primero que hemos de saber es que el intérprete interactivo de Python memoriza cada una de las expresiones evaluadas en una sesión interactiva por si deseamos recuperarlas más tarde. La lista de expresiones que hemos evaluado constituye la *historia* de la sesión interactiva. Puedes «navegar» por la historia utilizando la teclas de desplazamiento de cursor hacia arriba y hacia abajo. Cada vez que pulses la tecla de desplazamiento hacia arriba recuperarás una expresión más antigua. La tecla de desplazamiento hacia abajo permite recuperar expresiones más recientes. La expresión recuperada aparecerá ante el *prompt* y podrás modificarla a tu antojo.