

# Expresiones

Ricardo Pérez López

IES Doñana, curso 2024/2025

Generado el 2024/09/18 a las 10:55:00

## Índice

<b>1. El lenguaje de programación Python</b>	<b>2</b>
1.1. Historia . . . . .	2
1.2. Características principales . . . . .	3
1.3. Instalación . . . . .	3
1.4. Funcionamiento del intérprete . . . . .	3
1.4.1. Entrar y salir del intérprete interactivo . . . . .	4
<b>2. Elementos de un programa</b>	<b>4</b>
2.1. Expresiones y sentencias . . . . .	4
2.2. Sintaxis y semántica de las expresiones . . . . .	5
<b>3. Valores</b>	<b>7</b>
3.1. Información, datos, tipos y valores . . . . .	7
3.2. Evaluación de expresiones . . . . .	10
3.3. Expresión canónica y forma normal . . . . .	10
3.4. Formas normales y evaluación . . . . .	11
3.5. Literales . . . . .	14
3.6. Identificadores . . . . .	15
<b>4. Operaciones</b>	<b>15</b>
4.1. Clasificación . . . . .	15
4.2. Operadores . . . . .	18
4.2.1. Aridad de operadores . . . . .	18
4.2.2. Notación de los operadores . . . . .	18
4.2.3. Paréntesis . . . . .	19
4.2.4. Prioridad de operadores . . . . .	19
4.2.5. Asociatividad de operadores . . . . .	20
4.2.6. Paréntesis y orden de evaluación . . . . .	22
4.2.7. Tipos de operandos . . . . .	23
4.3. Funciones . . . . .	24
4.3.1. Evaluación de expresiones con funciones . . . . .	28
4.3.2. Composición de operaciones y funciones . . . . .	30

4.4. Métodos . . . . .	30
<b>5. Otros conceptos sobre operaciones</b>	<b>33</b>
5.1. Árboles sintácticos y evaluación . . . . .	33
5.2. Tipos polimórficos y operaciones polimórficas . . . . .	37
5.3. Sobrecarga de operaciones . . . . .	38
5.4. Equivalencia entre formas de operaciones . . . . .	38
5.5. Igualdad de operaciones . . . . .	40
<b>6. Operaciones predefinidas</b>	<b>40</b>
6.1. Operadores predefinidos . . . . .	40
6.1.1. Operadores aritméticos . . . . .	40
6.1.2. Operadores de cadenas . . . . .	40
6.2. Funciones predefinidas . . . . .	41
6.2.1. Funciones matemáticas y módulos . . . . .	41
6.3. Métodos predefinidos . . . . .	44

## 1. El lenguaje de programación Python

### 1.1. Historia

Python fue creado a finales de los ochenta por **Guido van Rossum** en el Centro para las Matemáticas y la Informática (CWI, Centrum Wiskunde & Informatica), en los Países Bajos, como un sucesor del lenguaje de programación ABC.

El nombre del lenguaje proviene de la afición de su creador por los humoristas británicos **Monty Python**.



Logo de Python

Python alcanzó la versión 1.0 en enero de 1994.

Python 2.0 se publicó en octubre de 2000 con muchas grandes mejoras. Actualmente, Python 2 está obsoleto.

Python 3.0 se publicó en septiembre de 2008 y es una gran revisión del lenguaje que no es totalmente retrocompatible con Python 2.

## 1.2. Características principales

Python es un lenguaje **interpretado**, de **tipado dinámico** y **multiplataforma**, cuya filosofía hace hincapié en una sintaxis que favorezca un **código legible**.

Es un lenguaje de programación **multiparadigma**. Esto significa que más que forzar a los programadores a adoptar un estilo particular de programación, permite varios estilos: **programación orientada a objetos**, **programación estructurada** y **programación funcional**.

Tiene una **gran biblioteca estándar**, usada para una diversidad de tareas. Esto viene de la filosofía «pilas incluidas» (*batteries included*) en referencia a los módulos de Python.

Es administrado por la **Python Software Foundation** y posee una licencia de **código abierto**.

La estructura de un programa se define por su anidamiento.

## 1.3. Instalación

### Instalación en Debian:

1. Python 3 ya viene instalado en Debian 11 y posteriores, pero es conveniente ejecutar los siguientes comandos desde un terminal del sistema operativo:

```
$ sudo apt update
$ sudo apt install python-is-python3 python3-pip
$ mkdir -p ~/.local/bin
```

Al acabar, cerrar la sesión con el sistema operativo y abrir una nueva antes de continuar.

2. Asimismo, es conveniente asegurarse de que no hay ciertos paquetes de Python instalados provenientes del repositorio de Debian (se instalarán directamente desde Visual Studio Code):

```
$ sudo apt purge --autoremove pylint jupyter mypy python3-autopep8
```

### Instalación en Windows:

1. Visitar <https://www.python.org/downloads/windows> y descargar desde ahí la última versión estable.
2. Durante la instalación, marcar la casilla que activa la opción de añadir comandos al *PATH*.

## 1.4. Funcionamiento del intérprete

El intérprete de Python tiene dos **modos de funcionamiento**:

1. **Modo interactivo (REPL)**: se abre una **sesión interactiva** con el intérprete donde éste nos solicita que introduzcamos una instrucción (sentencia o expresión) y, a continuación, la ejecuta inmediatamente, para luego volver a solicitar una nueva instrucción, repitiendo este ciclo indefinidamente en un **diálogo continuo entre el programador y el intérprete**.

Es el modo más apropiado para aprender el lenguaje o hacer pruebas rápidas, y el que empezaremos a usar nosotros.

2. **Modo por lotes:** el intérprete ejecuta un programa previamente escrito por el programador y almacenado en archivos fuente (llamados *scripts*) que contienen instrucciones que el intérprete va ejecutando una tras otra.

Es el modo que utilizan los usuarios para ejecutar los programas previamente confeccionados por los programadores.

### 1.4.1. Entrar y salir del intérprete interactivo

Para entrar en el intérprete interactivo, se usa el comando `python` desde la línea de órdenes del sistema operativo:

```
$ python
Python 3.10.7 (main, Sep  8 2022, 14:34:29) [GCC 12.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

El mensaje que obtengamos puede que no sea exactamente igual, pero es importante comprobar que estamos usando Python 3 y no 2.

Para salir, se pulsa `Ctrl+D` en Ubuntu o `Ctrl+Z` en Windows.

El `>>>` es el *prompt* del intérprete de Python, desde el que se ejecutan las expresiones y sentencias que tecleemos:

```
>>> 4 + 3
7
>>>
```

## 2. Elementos de un programa

### 2.1. Expresiones y sentencias

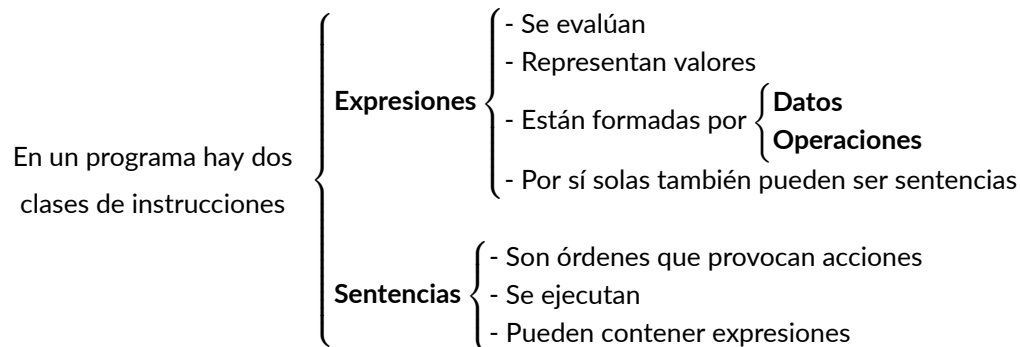
El código fuente de un programa está formado por **instrucciones**.

Las instrucciones se clasifican en:

- **Expresiones:** son secuencias de símbolos las cuales *representan valores* y están formados por *datos* y (posiblemente) *operaciones* a realizar sobre esos datos. El valor al que representa la expresión se obtiene *evaluando* dicha expresión.
- **Sentencias:** son *órdenes* que sirven para pedirle al intérprete que *ejecute* una determinada *acción*.

Las sentencias pueden contener expresiones.

En muchos lenguajes de programación, una expresión por sí sola también es una sentencia válida, ya que expresan la orden de calcular el valor de la expresión.



Las **expresiones** se *evalúan* y denotan un *valor*.

Las **sentencias** se *ejecutan* y provocan una *acción*.

## 2.2. Sintaxis y semántica de las expresiones

### Definición:

Una **expresión** es una frase (secuencia de símbolos) sintáctica y semánticamente correcta según las reglas del lenguaje que estamos utilizando, cuya finalidad es la de *representar* o **denotar** un determinado objeto abstracto, al que denominamos el **valor** de la expresión.

El ejemplo clásico es el de las *expresiones aritméticas*:

- Están formadas por secuencias de números junto con símbolos que representan operaciones aritméticas a realizar con esos números.
- Denotan un valor numérico, que es el resultado de calcular el valor de la expresión tras hacer las operaciones que aparecen en ella.

Por ejemplo, la expresión  $(2 * (3 + 5))$  denota un valor, que es el número abstracto **dieciséis**.

La **sintaxis** de las expresiones correctamente formadas deben satisfacer la gramática del lenguaje en el que están escritas.

Si una expresión es sintácticamente correcta, su **semántica** (es decir, su *significado*) es **el valor al que representa**.

En un lenguaje de programación existen muchos tipos de expresiones, dependiendo del tipo de los datos y de las operaciones involucradas en dicha expresión.

Empezaremos trabajando con las expresiones aritméticas más sencillas para ir incorporando cada vez más elementos nuevos que nos permitan crear expresiones más complejas.

Para ello, nos basaremos en la siguiente gramática, la cual es una simplificación modificada de la gramática real que deben satisfacer las expresiones en Python:

```

<expresión> ::= <operación> | <literal> | <nombre>
<operación> ::= (<expresión> <operador_binario> <expresión>)
               | (<operador_unario> <expresión>)
               | <llamada_función> | <llamada_método>
<nombre> ::= identificador
<literal> ::= entero | real | cadena | ...
<operador_binario> ::= + | - | * | / | // | ** | % | ...
<operador_unario> ::= + | - | ...
<llamada_función> ::= <función>(<lista_argumentos>)]
<función> ::= identificador
<llamada_método> ::= <objeto>.<método>(<lista_argumentos>)]
<objeto> ::= <expresión>
<método> ::= identificador
<lista_argumentos> ::= <expresión>{, <expresión>)*

```

Los símbolos terminales **identificador**, **entero**, **real** y **cadena** representan cada uno de ellos a una multitud de secuencias de caracteres que siguen un determinado patrón.

A las secuencias de caracteres que se ajustan al patrón léxico de un símbolo terminal determinado se les denomina **lexemas**.

- Un **identificador** es una secuencia de caracteres que puede estar formada por letras y dígitos, siempre y cuando empiece por una letra. Los identificadores representan *nombres*. Por ej.: `x`, `n1` o `dni`.
- Un **entero** es una secuencia de dígitos sin punto decimal que representa a un número entero. Por ej.: `25` o `140`.
- Un **real** es una secuencia de dígitos y otros caracteres especiales (el `.` o la `e`) que representa a un número real. Por ej.: `12.4` o `4e3`.
- Una **cadena** es una secuencia de caracteres encerrada entre comillas simples (`'`) o dobles (`"`). Por ej.: `"Hola"` o `'Juan Martínez'`.

Normalmente, los espacios en blanco se usan para separar y distinguir los diferentes símbolos terminales. Así se sabe que `12` representa al entero `12` y no a los dos enteros `1` y `2`.

Esta gramática reconoce **expresiones totalmente parentizadas**, en las que cada operación a realizar con operadores va agrupada entre paréntesis, aunque no sea estrictamente necesario, como por ejemplo:

( 3 + ( 4 - 7 ) )

Otros ejemplos de expresiones que satisfacen dicha gramática:

- `24`
- `( 4 + 5 )`
- `-( 8 * 3.5 )`
- `( 9 * ( x - 2 ) )`
- `z`
- `( abs(-3) + ( max(8, 5) / 2 ) )`

Sabemos que todas esas expresiones son sintácticamente correctas según nuestra gramática porque podemos construir derivaciones desde el símbolo inicial *⟨expresión⟩* hasta cada expresión.

### Ejercicio

1. Obtener las derivaciones correspondientes de cada una de las expresiones.

Muchas veces, de ahora en adelante representaremos las expresiones combinando **distintos colores y estilos tipográficos** con la única finalidad de facilitar la lectura y ayudar a reconocer los diferentes elementos sintácticos que las componen.

A esta técnica se la denomina **resaltado de sintaxis**.

Por ejemplo, las expresiones anteriores quedarían así:

- $(3 + (4 - 7))$
- $24$
- $(4 + 5)$
- $-(8 * 3.5)$
- $(9 * (x - 2))$
- $z$
- $(\text{abs}(-3) + (\text{max}(8, 5) / 2))$

## 3. Valores

### 3.1. Información, datos, tipos y valores

La Informática es la ciencia que estudia los sistemas de procesamiento automático de la información, también llamados *sistemas informáticos*.

Esos sistemas procesan la información de forma automática siguiendo las instrucciones de un programa.

Las instrucciones que forman el programa son las que dictan qué operaciones hay que realizar con la información.

Esa información se codifica, almacena y manipula en forma de **datos**.

Los datos, por tanto, son información codificada y almacenada en un formato y en un soporte adecuados para ser manipulados por un sistema informático.

Esos datos pueden ser:

- **Datos de entrada o salida**, que representan información de interés para el usuario del programa y que, o bien se reciben del usuario o bien se envían al usuario, respectivamente.
- **Datos internos** que usa el programa para su correcto funcionamiento.

El objeto abstracto que representa un dato en un momento dado (es decir, la información que actualmente «contiene» el dato) se denomina el **valor** del dato. Se dice que un dato *posee* (o *tiene* o *contiene*) un **valor**, o que *vale* ese valor.

- Si el valor de un dato nunca cambia, decimos que ese dato es **constante**.
- En cambio, si el valor de un dato puede cambiar durante el funcionamiento del programa, decimos que es un dato **variable**.

Los valores se agrupan en **tipos**, que son conjuntos de valores que comparten características comunes.

Entre esas características comunes destacan, principalmente, las **operaciones** que se pueden realizar con esos valores.

En consecuencia, el tipo al que pertenece un valor determina también el conjunto de operaciones que se pueden realizar sobre ese valor.

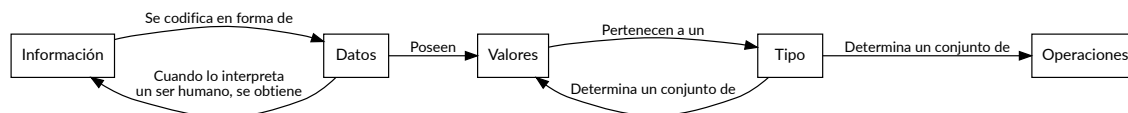
Por tanto, **cada valor pertenece a un tipo**.

Igualmente, también decimos que **los datos tienen un tipo**, que es el tipo de los valores que puede tener el dato. Por eso, a los tipos también se les llama **tipos de datos**.

El **tipo de un dato** determina el conjunto de **valores** que puede tomar un dato de ese tipo y, por tanto, también las **operaciones** que se pueden realizar con ese dato.

Los datos se pueden manipular dentro de un programa haciendo que formen parte de **expresiones** y evaluando dichas expresiones.

El **valor de una expresión** se obtiene a través del valor de los datos que contiene, y podemos manipular esos datos a través de las **operaciones** que actúan sobre ellos dentro de la expresión.



Relación entre información, datos, valores, tipos y operaciones

### Ejemplo

Supongamos que queremos representar en un sistema informático la edad de una persona.

Esa información la codificamos en un **dato** al que llamaremos *edad*.

Ese dato *edad* será un dato **variable**, ya que la edad de una persona cambia con el tiempo y, por tanto, el valor que contenga el dato podrá cambiar.

Por otra parte, la edad de una persona se puede representar mediante un número entero no negativo (mayor o igual que cero), así que el dato *edad* debería contener un valor que sea un número entero sin signo.

Por tanto, el **tipo** del dato *edad* es el conjunto de los números enteros sin signo y, en consecuencia, el **valor** del dato *edad* podría ser cualquiera de los números que están dentro de ese conjunto.



Si, por ejemplo, una determinada persona tiene una edad de 12 años, decimos que el **valor del dato** *edad* para esa persona es 12, o que **su edad vale 12** (actualmente).

Dentro de un año, el valor de ese dato pasará a ser 13.

Por eso decimos que la edad es un dato *variable*.

En cambio, su fecha de nacimiento es un valor *constante*, ya que nunca cambia.

Recordemos que el tipo de un dato también determina las **operaciones** que podemos realizar con él, ya que (en principio) cada operación sólo actúa sobre datos de un determinado tipo.

Como el dato *edad* es un número entero, podemos realizar operaciones aritméticas sobre él. Por ejemplo, podemos restar la edad al año actual para averiguar el año de nacimiento de esa persona.

En Python, los **tipos de datos básicos** son los siguientes (entre paréntesis va el nombre que tienen cada uno de esos tipos en Python):

- **Números enteros (`int`)**: los números (positivos o negativos) que sólo tienen parte entera, como el **cuatro** o el **menos tres**.
- **Números reales (`float`)**: los números (positivos o negativos) que tienen parte entera y parte fraccionaria, como el **siete con cuatro** o el **menos ocho con diecisiete**.
- **Cadenas de caracteres (`str`)**: secuencias de caracteres (letras, dígitos, símbolos, etc.), como nombres de personas, direcciones, o cualquier texto en general.
- **Lógicos (`bool`)**: sólo contiene dos valores que representan dos posibilidades contrarias, como **verdadero** o **falso**, **sí** o **no**, **encendido** o **apagado**, etc.

Por tanto, cuando un dato es un número entero, en Python decimos que pertenece al tipo `int`; cuando es una cadena, decimos que pertenece al tipo `str`; etcétera.

Para indicar que un dato tiene (o pertenece a) un determinado tipo, se pueden usar distintas notaciones dependiendo del lenguaje empleado.

Por ejemplo, si queremos indicar que el dato *edad* es un número entero (es decir, que es de tipo entero), se puede representar así:

- En Matemáticas:

$edad \in \mathbb{Z}$

- En Teoría de tipos:

$edad : \mathbb{Z}$

- En Python:

`edad: int`

- En Java:

`int edad`

### 3.2. Evaluación de expresiones

**Evaluar una expresión** consiste en determinar el **valor de la expresión**. Es decir, una expresión *representa* o *denota* el valor que se obtiene al evaluarla.

El **tipo de la expresión** es el tipo del valor de esa expresión.

Una **subexpresión** es una expresión contenida dentro de otra.

La **evaluación de una expresión**, en esencia, es el proceso de **sustituir** (o **reescribir**), dentro de ella, unas *subexpresiones* por otras que, según indiquen ciertas reglas, estén *más cerca* del valor final a calcular, y así hasta calcular el **valor** de la expresión al completo.

Al mismo tiempo, también se va calculando el **tipo** de sus *subexpresiones* (el tipo del valor de cada *subexpresión*) y, finalmente, se obtiene el tipo de la propia expresión, que será el tipo del valor de esa expresión.

Además de las expresiones existen las *sentencias*, que no poseen ningún valor y que, por tanto, no se evalúan sino que se *ejecutan*. Las sentencias son básicas en los paradigmas *imperativos*.

Podemos decir que las expresiones:

3

( 1 + 2 )

( 5 - 2 )

denotan todas el mismo **valor** (el número abstracto **tres**, cuyo **tipo** es el conjunto de los *números enteros*).

Es decir: todas esas expresiones son representaciones diferentes del mismo ente abstracto.

Cuando introducimos una expresión en el intérprete, lo que hace éste es buscar la **representación más simplificada o reducida** posible.

En el ejemplo anterior, sería la expresión 3.

Por eso a menudo usamos, indistintamente, los términos *reducir*, *simplificar* y *evaluar*.

### 3.3. Expresión canónica y forma normal

Los ordenadores no manipulan valores, sino que sólo pueden manejar representaciones concretas de los mismos.

Por ejemplo: utilizan la codificación binaria en complemento a dos para representar los números enteros.

Pedimos que la **representación del valor** resultado de una evaluación sea **única**.

De esta forma, seleccionaremos de cada conjunto de expresiones que denoten el mismo valor, a lo sumo una que llamaremos **expresión canónica de ese valor**.

Además, llamaremos a la expresión canónica que representa el valor de una expresión la **forma normal de esa expresión**.

Con esta restricción pueden quedar expresiones sin forma normal.

### Recuerda

Se dice:

**Expresión canónica de un valor.**

**Forma normal de una expresión.**

### Ejemplo

De las expresiones anteriores:

3

(1 + 2)

(5 - 2)

que denotan todas el mismo valor abstracto **tres**, seleccionamos una (la expresión 3) como la **expresión canónica** de ese valor.

Igualmente, la expresión 3 es la **forma normal** de todas las expresiones anteriores (y de cualquier otra expresión con valor **tres**).

Es importante no confundir el valor abstracto **tres** con la expresión 3 que representa dicho valor.

Hay **valores que no tienen expresión canónica**:

- Las funciones (los valores de tipo *función*).
- El número  $\pi$  no tiene representación decimal finita, por lo que tampoco tiene expresión canónica.

Y hay **expresiones que no tienen forma normal**:

- Si definimos  $inf = inf + 1$ , la expresión *inf* (que es un número) no tiene forma normal.
- Lo mismo ocurre con  $\frac{1}{0}$ .

## 3.4. Formas normales y evaluación

Según lo visto hasta ahora, la evaluación de una expresión es el proceso de encontrar su forma normal.

Para ello, el intérprete evalúa la expresión reduciendo sus subexpresiones según las reglas del lenguaje y las operaciones que aparecen en ellas, buscando su forma normal.

El sistema de evaluación dentro del intérprete está hecho de tal forma que cuando ya no es posible reducir más la expresión es porque se ha llegado a la forma normal.

Recordemos que no todos los valores tienen forma normal.

El orden en el que se van reduciendo las subexpresiones no debería influir en el resultado de evaluar una expresión, así que debería dar igual elegir una u otra subexpresión.

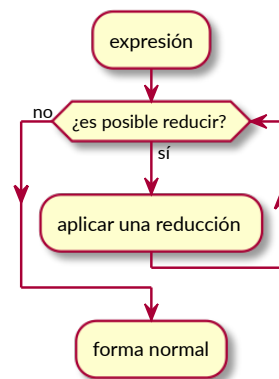
De todas formas, los lenguajes de programación suelen imponer un orden concreto a la hora de evaluar las expresiones.

Tanto en Python como en Java (los dos lenguajes que veremos), el orden de evaluación es de izquierda a derecha (salvo excepciones):

#### Orden de evaluación de las expresiones:

Al evaluar una expresión, las subexpresiones que la forman siempre se evaluarán **de izquierda a derecha**.

El orden de evaluación de las subexpresiones es un asunto más complejo de lo que parece, y lo estudiaremos en profundidad en posteriores apartados.



Evaluación de una expresión

### Ejemplos

Evaluar la expresión  $(2 + 3)$ :

- La expresión está formada por un operador  $+$  que actúa sobre las dos subexpresiones  $2$  y  $3$ . Por tanto, habrá que evaluar primero esas dos subexpresiones, siempre de izquierda a derecha:

```

(2 + 3)      # se evalúa primero 2 (que devuelve 2)
= (2 + 3)    # luego se evalúa 3 (que devuelve 3)
= (2 + 3)    # ahora se evalúa (2 + 3) (que devuelve 5)
= 5
  
```

El  $+$  por sí solo no se evalúa, ya que no es una expresión válida.

Por tanto, la expresión  $5$  es la *forma normal* de la expresión  $(2 + 3)$ , y ambas representan al valor **cinco**, que es un valor de **tipo entero**.

Asimismo, decimos que  $5$  es la *expresión canónica* del valor **cinco**, de **tipo entero** (es la expresión que mejor representa a ese valor).

Evaluar la expresión  $(2 + (3 * 5))$ :

- La expresión está formada por un operador  $+$  que actúa sobre las dos subexpresiones  $2$  y  $(3 * 5)$ .

- La segunda subexpresión, a su vez, está formada por un operador  $*$  que actúa sobre las dos subexpresiones 3 y 5.
- Todas las subexpresiones se evalúan siempre de izquierda a derecha, a medida que se van reduciendo:

```

(2 + (3 * 5))      # se evalúa primero 2 (que devuelve 2)
= (2 + (3 * 5))    # se evalúa 3 (que devuelve 3)
= (2 + (3 * 5))    # se evalúa 5 (que devuelve 5)
= (2 + (3 * 5))    # se evalúa (3 * 5) (que devuelve 15)
= (2 + 15)         # se evalúa (2 + 15) (que devuelve 17)
= 17

```

Por tanto, la expresión 17 es la *forma normal* de la expresión  $(2 + (3 * 5))$ . Ambas representan al valor **diecisiete**, que es un valor de **tipo entero**, pero 17 es la *expresión canónica* de ese valor.

Evaluar la expresión  $((2 + 5) * 3)$ :

- La expresión está formada por un operador  $*$  que actúa sobre las dos subexpresiones  $(2 + 5)$  y 3.
- La primera subexpresión, a su vez, está formada por un operador  $+$  que actúa sobre las dos subexpresiones 2 y 5.
- Todas las subexpresiones se evalúan siempre de izquierda a derecha, a medida que se van reduciendo:

```

((2 + 5) * 3)      # se evalúa primero 2 (que devuelve 2)
= ((2 + 5) * 3)    # se evalúa 5 (que devuelve 5)
= ((2 + 5) * 3)    # se evalúa (2 + 5) (que devuelve 7)
= (7 * 3)          # se evalúa 3 (que devuelve 3)
= (7 * 3)          # se evalúa (7 * 3) (que devuelve 21)
= 21

```

Por tanto, la expresión 21 es la *forma normal* de la expresión  $((2 + 5) * 3)$ . Ambas representan al valor **veintiuno**, que es un valor de **tipo entero**, pero 21 es la *expresión canónica* de ese valor.

De ahora en adelante, para simplificar las explicaciones, **a menudo usaremos la expresión canónica de un valor como si fuera el valor mismo**, cuando ya sabemos que, en realidad, no son lo mismo.

Por ejemplo: diremos «el valor 21» cuando, en realidad, 21 no es el propio valor **veintiuno**, sino una expresión que *representa* al valor **veintiuno**.

Esto lo haremos en contextos donde no haya confusión, y siempre entendiendo que cuando decimos «el valor 21» nos referimos al valor que representa la expresión 21.

### 3.5. Literales

Los literales constituyen **las expresiones más sencillas** del lenguaje.

Un literal es una expresión simple que denota **un valor concreto, constante y fijo**, escrito directamente en la expresión y ya totalmente reducido (o casi) a su expresión canónica.

Los literales tienen que satisfacer las **reglas léxicas** del lenguaje, que son las que determinan qué forma pueden tener los componentes léxicos del programa (como números, cadenas, identificadores, etc.).

Esos componentes léxicos son los *símbolos terminales* de la gramática del lenguaje.

Gracias a esas reglas, el intérprete puede identificar qué literales son, qué **valor** representan y de qué **tipo** son.

Ejemplos de distintos tipos de literales:

Números enteros (tipo <code>int</code> )	Números reales (tipo <code>float</code> )	Cadenas (tipo <code>str</code> )
-2	3.5	"hola"
-1	-2.7	'pepe'
0		"25"
1		" "
2		

Algunas reglas léxicas son:

- Si el número tiene un `.` decimal, es que es un número real.
- Si algo va entre comillas (simples `'` o dobles `"`) es que es una cadena.

En apartados posteriores estudiaremos los tipos de datos con más profundidad.

Con frecuencia, un literal resulta ser la *expresión canónica* del valor al que denotan y la forma normal de todas las posibles expresiones que denotan ese valor.

Por consiguiente, suelen estar ya totalmente simplificados.

Por ejemplo, el `3.5` es un literal que denota el valor numérico **tres y medio**, es su expresión canónica y es la forma normal de cualquier expresión que denote dicho valor.

Por tanto, el literal `3.5` es la forma más reducida de representar el valor **tres y medio**.

Es decir: si le pedimos al intérprete que calcule el resultado de `7 / 2`, nos devolverá la expresión `3.5`.

Sin embargo, el `3.5` no es el único literal que denota el valor numérico **tres y medio**. Por ejemplo, los literales `3.50`, `3.500` o `03.50` también denotan ese mismo valor, pero la forma normal de todos ellos es `3.5`.

O sea: hay varias maneras de escribir un literal que denote el valor **tres y medio**, pero sólo el literal **3.5** es la forma normal de todas ellas.

Igualmente, la forma normal de todas las posibles expresiones que denotan el valor numérico **dos** es el literal **2**.

El literal **2** es la forma más reducida de representar el valor **dos**.

Pero no es el único literal que denota dicho valor.

El literal **02** no es correcto según las reglas léxicas del lenguaje, pero sí que podemos usar la expresión **0b10**, que es un literal que representa el valor **dos** escrito en binario.

Igualmente, las reglas léxicas del lenguaje permiten usar el carácter `_` dentro de un número, por lo que el valor numérico **cuatro millones** se puede representar con el literal **4\_000\_000**, si bien su forma normal sigue siendo simplemente **4000000**.

Finalmente, las cadenas se pueden escribir con comillas simples (') o dobles ("), pero la forma normal de una cadena siempre usa las simples (salvo que la propia cadena incluya una comilla simple como un carácter más, en cuyo caso se usarán las comillas dobles en su forma normal).

### 3.6. Identificadores

Los **identificadores** son nombres que representan elementos dentro de un programa.

Por ejemplo, el nombre de una función es un identificador porque representa a la función.

Los identificadores **deben cumplir unas reglas léxicas** que dependen del lenguaje de programación, pero generalmente se resumen en que:

- Pueden estar formados por combinaciones de **letras**, **dígitos** y algunos **caracteres especiales** como `_` (por ejemplo, `salida_principal23`).
- **No pueden empezar con un dígito**, ya que eso los confundiría con un número (por ejemplo, `9abc`).
- La mayoría de los lenguajes **distinguen las mayúsculas de las minúsculas**, por lo que `cantidad`, `Cantidad` y `CANTIDAD` son normalmente identificadores distintos (así ocurre en Python y Java).

## 4. Operaciones

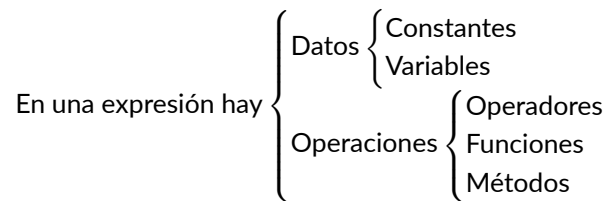
### 4.1. Clasificación

Ya hemos visto que en una expresión puede haber:

- **Datos** (constantes o variables).
- **Operaciones** a realizar sobre esos datos.

A su vez, las operaciones pueden aparecer en forma de:

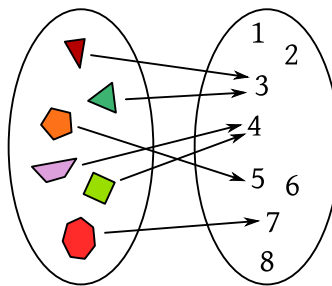
- Operadores
- Funciones
- Métodos



Matemáticamente, una **operación** se corresponde con el concepto de *función*.

En Matemáticas, una **función** es una regla que **asocia** a cada elemento de un conjunto (el **dominio**) **exactamente un elemento** (ni más ni menos) de un segundo conjunto (el **codominio** o **rango**).

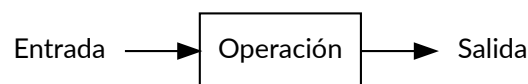
A su vez, las funciones están definidas sobre un **conjunto origen** y un **conjunto imagen**.



Función matemática que asocia a cada polígono con su número de lados

En Programación, el concepto de *operación* es también similar al de *función matemática*, pero con su propia terminología y funcionamiento.

Desde el punto de vista de la Programación, las operaciones son dispositivos que **transforman datos de entrada en datos de salida**:



Esos datos de entrada se denominan **argumentos** u **operandos**, según el tipo de operación.

Asimismo, los datos de salida representan el **resultado** o **valor de retorno** de la operación.

El valor de retorno se calcula como resultado de procesar los datos de entrada y, por tanto, depende de la operación a realizar y de los datos recibidos por la entrada.

Visto así, las operaciones son **subprogramas**, es decir, pequeños programas dentro de otros programas.

Los operadores, las funciones y los métodos no son más que **diferentes formas sintácticas** de expresar una operación.

En la práctica, apenas hay diferencias sustanciales entre las tres formas.

Por ejemplo, la operación «cuadrado» actúa asociando cada número real con otro número real: el cuadrado del número (el número multiplicado por sí mismo).



Así, esa operación asocia el número 4 con el 16, y el 7 con el 49.

Aquí, tanto el dominio como el codominio de la operación serían  $\mathbb{R}$ , el conjunto de los números reales.

Por tanto, podemos definirla diciendo que es una función que asocia  $\mathbb{R}$  (su *conjunto origen*) con  $\mathbb{R}$  (su *conjunto imagen*), lo que se puede representar así:

$$\text{cuadrado} : \mathbb{R} \longrightarrow \mathbb{R}$$

Otro ejemplo sería la operación «inverso», que asocia a cada número real otro número real: el inverso del número (1 dividido entre el número).

Así, esa operación asocia el número 4 con el 0.25 ( $\frac{1}{4}$ ) y el 5 con 0.2 ( $\frac{1}{5}$ ).

El conjunto origen y el conjunto imagen siguen siendo  $\mathbb{R}$ .

Pero, en este caso, el dominio no coincide con el conjunto origen, ya que no existe el inverso del número 0 ( $\frac{1}{0}$ ) («inverso» es una operación *parcial*). Por tanto, el conjunto origen podrá ser  $\mathbb{R}$  pero su dominio deberá excluir al cero:

$$\text{dom}(\text{inverso}) = \mathbb{R} \setminus \{0\}$$

(En cambio, el conjunto imagen sí coincide con el codominio.)

En consecuencia, podemos definir la función de la siguiente manera:

$$\text{inverso} : \mathbb{R} \longrightarrow \mathbb{R}$$

sabiendo que el dominio de la función no contiene a todo el conjunto origen (el cero no está).

Otro ejemplo sería la operación «suma», que asocia a cada pareja de números reales otro número real: la suma de ambos.

Esa operación asocia, por ejemplo, los números 2 y 3 con el número 5.

En este caso, el dominio de la operación «suma» sería el producto cartesiano del conjunto de los números reales consigo mismo (es decir, el conjunto  $\mathbb{R} \times \mathbb{R}$  formado por parejas de números reales).

Por tanto, podemos definir la operación «suma» diciendo que es una operación que asocia  $\mathbb{R} \times \mathbb{R}$  (su conjunto origen) con  $\mathbb{R}$  (su conjunto imagen), lo que se puede representar así:

$$\text{suma} : \mathbb{R} \times \mathbb{R} \longrightarrow \mathbb{R}$$

En Programación, las operaciones no se definen sobre conjuntos, sino sobre **tipos**.

Por ejemplo, la operación «suma» podría definirse sobre sobre el tipo `float` de Python, que es el más parecido al conjunto  $\mathbb{R}$  de los números reales.

Eso se podría representar así:

$$\text{suma} : \text{float} \times \text{float} \longrightarrow \text{float}$$

Pero, en la práctica, usaremos una notación más apropiada para representar los tipos sobre los que se definen las operaciones en nuestros programas y nuestros lenguajes de programación.

Esa notación, llamada *signatura*, la estudiaremos en breve.

## 4.2. Operadores

Un **operador** es un símbolo o palabra clave que representa una *operación* a realizar sobre unos *datos*.

Los datos sobre los que actúa un operador se denominan **operandos**.

Por ejemplo, los operadores aritméticos  $+$ ,  $-$ ,  $*$  y  $/$ , cuando actúan sobre operandos numéricos, representan las operaciones de suma, resta, multiplicación y división, respectivamente.

- En la expresión  $(3 + 4)$ :
  - \* Los operandos son 3 y 4.
  - \* El operador es  $+$  y representa la operación «suma de números».
- En la expresión  $(9 * 8)$ :
  - \* Los operandos son 9 y 8.
  - \* El operador es  $*$  y representa la operación «multiplicación de números».

Por tanto, la operación «suma» de la que hablamos anteriormente, se puede representar así usando el operador  $+$ :

$$+_{} : \mathbb{R} \times \mathbb{R} \longrightarrow \mathbb{R}$$

Eso significa que el operador  $+$  acepta dos operandos (dos números reales) y devuelve un número real.

### 4.2.1. Aridad de operadores

La **aridad** de un operador es el número de operandos sobre los que opera.

Los operadores se clasifican según su aridad en:

- **Unarios**: operan sobre un único operando.  
Por ejemplo, el operador  $-$  de cambio de signo:  $-(5 + 3)$ .
- **Binarios**: operan sobre dos operandos.  
Por ejemplo, la mayoría de los operadores aritméticos.
- **Ternarios**: operan sobre tres operandos.  
Veremos un ejemplo más adelante.

### 4.2.2. Notación de los operadores

La **notación** de un operador indica en qué posición se escribe dicho operador con respecto a sus operandos en una expresión.

Los operadores se pueden escribir:

- Con **notación prefija**: cuando el operador se escribe *antes* que los operandos.  
Es el que se usa habitualmente para los operadores **unarios**, como el operador  $-$  de cambio de signo.

- Con **notación infija**: cuando el operador se escribe *entre* los operandos.  
Es el que se usa habitualmente para los operadores **binarios**.
- Con **notación postfija**: cuando el operador se escribe *después* que los operandos.  
Se usa poco en la práctica, salvo casos especiales.

#### 4.2.3. Paréntesis

Los **paréntesis** sirven para agrupar elementos dentro de una expresión y romper la posible ambigüedad que pueda haber respecto a qué operador actúa sobre qué operandos.

Se usan, sobre todo, para hacer que varios elementos de una expresión actúen como uno solo (una subexpresión) al realizar una operación.

- Por ejemplo:  
 $((3 + 4) * 5)$  vale 35  
 $(3 + (4 * 5))$  vale 23

Una expresión está **correctamente parentizada** si tiene los paréntesis bien colocados según dicta la gramática del lenguaje.

Una expresión está **totalmente parentizada** si agrupa con paréntesis a **todas** las operaciones con sus operandos.

Hasta ahora, según nuestra gramática, las expresiones correctamente parentizadas son precisamente las que están totalmente parentizadas.

Por ejemplo:

- $2 + 3 * (5$  no está correctamente parentizada.
- $(4 + (2 * 5))$  está correcta y totalmente parentizada.
- $2 + 5$  no está totalmente parentizada y, por tanto, no está correctamente parentizada según nuestra gramática.

Para reducir la cantidad de paréntesis en una expresión, podemos cambiar nuestra gramática acudiendo a un esquema de **prioridades** y **asociatividades** de operadores.

Haciendo eso, ya no hará falta exigir que las expresiones estén totalmente parentizadas.

#### 4.2.4. Prioridad de operadores

En ausencia de paréntesis, cuando un operando está afectado a izquierda y derecha por **operadores de distinta prioridad**, se aplican las reglas de la **prioridad**:

$8 + 4 * 2$

El 4 está afectado a izquierda y derecha por los operadores + y \*, que tienen distinta prioridad, por lo que se aplican las reglas de la prioridad. El \* tiene *más prioridad* que el +, así que se agrupa con el \*. Equivale a:

```
8 + (4 * 2)
```

Si hiciéramos:

```
(8 + 4) * 2
```

el resultado sería distinto.

Ver prioridad de los operadores en Python en <https://docs.python.org/3/reference/expressions.html#operator-precedence>.

#### 4.2.5. Asociatividad de operadores

En ausencia de paréntesis, cuando un operando está afectado a izquierda y derecha por **operadores de la misma prioridad** (o por el **mismo operador**), se aplican las reglas de la **asociatividad**:

```
8 / 4 / 2
```

El 4 está afectado a derecha e izquierda por el mismo operador /, por lo que se aplican las reglas de la asociatividad. El / es *asociativo por la izquierda*, así que se agrupa con el operador que está a la izquierda. Equivale a hacer:

```
(8 / 4) / 2
```

Si hiciéramos:

```
8 / (4 / 2)
```

el resultado sería distinto.

En Python, todos los operadores son **asociativos por la izquierda** excepto el \*\*, que es asociativo por la derecha.

Por ejemplo, en este caso tenemos a un operando (el 3) afectado a derecha e izquierda por el mismo operador (el \*\*), por lo que, de nuevo, tenemos que aplicar las reglas de la asociatividad:

```
2 ** 3 ** 2
```

Pero como el operador \*\* es asociativo por la derecha, se agrupa con el operador de la derecha, resultando equivalente a hacer:

```
2 ** (3 ** 2)
```

En este ejemplo, el mismo operando (el 4) está afectado a derecha e izquierda por distintos operadores (el \* y el /), pero estos operadores tienen la misma prioridad, así que, de nuevo, tenemos que aplicar las reglas de la asociatividad:

```
8 * 4 / 2
```

Los operadores (`*` y `/`) son asociativos por la izquierda, por lo que, nuevamente, el `4` se agrupa con el operador que está a la izquierda, lo que equivale a hacer:

```
(8 * 4) / 2
```

En todos los casos, es necesario que ambos operadores tengan la misma asociatividad.

Podríamos plantearnos ahora qué ocurriría si los dos operadores tuvieran la misma prioridad pero distinta asociatividad.

Por ejemplo, supongamos que tenemos dos extraños operadores `&` y `@`.

Ahora mismo no nos interesa saber qué hacen; sólo que ambos tienen la misma prioridad pero distinta asociatividad.

Para este ejemplo, supongamos que `&` es asociativo por la izquierda y `@` por la derecha).

En ese caso, sería imposible interpretar correctamente la siguiente expresión, ya que un operador dice que hay que agrupar por la izquierda pero el otro dice lo contrario:

```
4 & 3 @ 5
```

Y lo mismo ocurriría con esta expresión:

```
4 @ 3 & 5
```

Si se diera un caso así, sería porque el lenguaje no está bien diseñado, ya que la gramática del lenguaje no aclararía con qué operador hay que agrupar en casos así.

De hecho, se nos estaría obligando a usar paréntesis para dejar claro cómo se agrupan los elementos que forman la expresión.

Por eso, lo normal es que los operadores que tienen la misma prioridad tengan también la misma asociatividad.

También puede ocurrir que existan operadores que no son asociativos ni por la derecha ni por la izquierda.

Tales operadores se dice que son **no asociativos**.

En Python no hay ningún operador no asociativo, pero, por ejemplo, en el lenguaje PHP existen varios operadores que sí lo son, entre ellos `==`.

Por tanto, la siguiente expresión en PHP sería incorrecta y daría lugar a un error sintáctico:

```
2 == 3 == 4
```

Tendríamos que usar paréntesis para dejar claro cómo se deben agrupar los elementos que forman la expresión, lo que daría lugar a dos posibles expresiones distintas:

```
(2 == 3) == 4
```

```
2 == (3 == 4)
```

Incorporando las reglas de la prioridad y la asociatividad, y eliminando la necesidad de que las expresiones estén totalmente parentizadas, nuestra gramática quedaría ahora sí:

```

<expresión> ::= <operación> | <literal> | <nombre> | (<expresión>)
<operación> ::= <expresión> <operador_binario> <expresión>
                | <operador_unario> <expresión>
                | <llamada_función> | <llamada_método>
<nombre> ::= identificador
<literal> ::= entero | real | cadena | ...
<operador_binario> ::= + | - | * | / | // | ** | % | ...
<operador_unario> ::= + | - | ...
<llamada_función> ::= <función>([<lista_argumentos>])
<función> ::= identificador
<llamada_método> ::= <objeto>.<método>([<lista_argumentos>])
<objeto> ::= <expresión>
<método> ::= identificador
<lista_argumentos> ::= <expresión>{, <expresión>}*

```

Ahora, cualquier expresión puede llevar paréntesis si es necesario, pero si no son necesarios se pueden omitir.

#### 4.2.6. Paréntesis y orden de evaluación

Es importante entender que los paréntesis sirven para agrupar elementos, pero por sí mismos no son suficientes para imponer un determinado **orden de evaluación**.

Por ejemplo, en la expresión  $4 * (3 + 5)$ , el intérprete intenta hacer el producto de 4 por  $(3 + 5)$ , pero no puede hacerlo hasta haber calculado primero la suma de 3 y 5.

Por eso el intérprete calcula primero la suma y finalmente hace el producto, porque para hacer el producto primero necesita haber reducido sus operandos a dos números que se puedan multiplicar.

El efecto final es que parece que los paréntesis han obligado a hacer primero la suma, como si los paréntesis fuesen una especie de operador cuya finalidad es la de aumentar la prioridad de lo que hay dentro.

En concreto, la evaluación de esa expresión sería:

```

4 * (3 + 5)      # se evalúa 4 (que devuelve 4)
= 4 * (3 + 5)    # se evalúa 3 (que devuelve 3)
= 4 * (3 + 5)    # se evalúa 5 (que devuelve 5)
= 4 * (3 + 5)    # se evalúa (3 + 5) (que devuelve 8)
= 4 * 8          # se evalúa 4 * 8 (que devuelve 32)
= 32

```

Sin paréntesis, la expresión  $4 * 3 + 5$  se evaluaría así:

```

4 * 3 + 5      # se evalúa 4 (que devuelve 4)
= 4 * 3 + 5    # se evalúa 3 (que devuelve 3)
= 4 * 3 + 5    # se evalúa 4 * 3 (que devuelve 12)
= 12 + 5       # se evalúa 5 (que devuelve 5)
= 12 + 5       # se evalúa 12 + 5 (que devuelve 17)
= 17

```

Aquí se puede hacer el producto antes que la suma porque los dos operandos del `*` están ya totalmente reducidos.

Pero, ¿qué ocurre con expresión `(2 + 3) * (4 + 5)`?

En un principio, ocurre algo parecido a lo de antes: para poder hacer el producto, primero hay que calcular las dos sumas, ya que los operandos del `*` son los valores que resultan de hacer esas sumas.

La cuestión es: ¿qué suma se hace primero? O dicho de otra forma: ¿en qué orden se evalúan los operandos del operador `*`?

Matemáticamente no hay ninguna diferencia entre calcular primero `2 + 3` y luego `4 + 5` o hacerlo al revés.

Pero ya sabemos que Python impone un orden de evaluación de izquierda a derecha al reducir las subexpresiones.

Por tanto, primero se evaluaría `(2 + 3)`, y después `(4 + 5)`.

El orden de evaluación no viene determinado por los paréntesis, sino por las reglas del lenguaje y el funcionamiento interno del intérprete.

En concreto, la evaluación de esa expresión sería:

```

(2 + 3) * (4 + 5)  # se evalúa 2 (que devuelve 2)
= (2 + 3) * (4 + 5) # se evalúa 3 (que devuelve 3)
= (2 + 3) * (4 + 5) # se evalúa (2 + 3) (que devuelve 5)
= 5 * (4 + 5)      # se evalúa 4 (que devuelve 4)
= 5 * (4 + 5)      # se evalúa el último 5 (que devuelve 5)
= 5 * (4 + 5)      # se evalúa (4 + 5) (que devuelve 9)
= 5 * 9            # se evalúa 5 * 9 (que devuelve 45)
45

```

#### 4.2.7. Tipos de operandos

Los operadores esperan que cada uno de sus operandos pertenezcan a un determinado tipo. Si intentamos aplicar un operador sobre operandos de tipos incorrectos, obtendremos resultados inesperados (o, directamente, un error).

Por ejemplo, los operadores aritméticos esperan operandos de tipo *numérico*. Así, si intentamos dividir dos cadenas usando el operador `/`, nos dará un error de tipos (`TypeError`):

```

>>> "hola" / "pepe"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for /: 'str' and 'str'
>>>

```

Durante la evaluación de una expresión, se deberá ir calculando tanto el valor como el tipo de cada subexpresión, y se debería ir comprobando si los tipos de los operandos son los adecuados para cada operación.

El concepto de **tipo de dato** es uno de los más importantes en Programación y lo estudiaremos en profundidad más adelante.

### 4.3. Funciones

Como cualquier otro tipo de operación, las funciones en Programación reciben unos datos de entrada y devuelven un valor de salida.

En Programación, los datos de entrada de una función se denominan **argumentos**.

Por tanto, los argumentos para las funciones son como los operandos para los operadores.

Las funciones reciben los argumentos a través de sus **parámetros**.

Para usar una función hay que conocer su **signatura**, la cual informa de:

- El **nombre** de la función, que es un *identificador* que hace referencia a la función.
- El **número, tipo, nombre y posición** de sus **parámetros**.
- El **tipo de retorno**, o **tipo** del *resultado* que devuelve.

Por ejemplo, la función `abs`, que está predefinida en Python, podría tener la siguiente signatura:

```
abs(x: int) -> int
```

Esa signatura nos dice que:

- La función se llama `abs`.
- Tiene un único parámetro llamado `x` que puede tomar cualquier valor de tipo `int` (un número entero).  
Por tanto, los argumentos que puede recibir la función (sus datos de entrada) deben ser valores numéricos de tipo entero, ya que los parámetros de una función representan a los argumentos dentro de la función.
- Su tipo de retorno es `int`, por lo que devuelve como resultado un número entero.

**La signatura de una función es el tipo de la función**, si consideramos que las funciones también son valores que pertenecen a un tipo, como cualquier otro valor.

Lo mismo se aplica a cualquier operación, no sólo a funciones.

El tipo de una función (su signatura) se puede representar de varias formas según el lenguaje utilizado. Por ejemplo:

- En Matemáticas y en Teoría de tipos:

$$\text{abs} : \mathbb{Z} \longrightarrow \mathbb{Z}$$
$$\text{pow} : \mathbb{R} \times \mathbb{R} \longrightarrow \mathbb{R}$$

- En Python:

```
abs(x: int) -> int
```



```
pow(base: float, exp: float) -> float
```

- En Java:

```
int abs(int x)
```

```
double pow(double base, double exp)
```

De aquí se puede deducir que:

- El **producto cartesiano de los tipos de los parámetros** de una función en Programación se corresponde con el **conjunto origen** de una función matemática.
- El **tipo de retorno** de una función en Programación se corresponde con el **conjunto imagen** de una función matemática.

Para usar la función, lo que hacemos es **llamar** o **invocar** a la función.

«*Llamar a una función*» consiste en **aplicar dicha función a unos argumentos**, que son los datos sobre los que queremos que actúe la función.

La **aplicación de una función a unos argumentos** es una expresión mediante la cual solicitamos que se realice una operación (que tiene forma de función) pasándole a ésta (a través de los argumentos) los datos sobre los que queremos que actúe la operación.

A la aplicación de una función a unos argumentos también se la denomina **invocación** de la función o **llamada** a la función.

En la llamada a la función, **los argumentos sustituyen a los parámetros** según el orden en el que aparecen en la llamada, haciendo corresponder el primer argumento con el primer parámetro, el segundo con el segundo y así sucesivamente.

Dicho de otra forma: **los parámetros toman los valores de los argumentos correspondientes**.

También se dice que **los argumentos se pasan a los parámetros**.

Por eso debe haber tantos argumentos como parámetros, ni más ni menos.

Sintácticamente, la **llamada a una función** tiene esta forma:

```
<llamada_función> ::= <función>([<lista_argumentos>])
<función> ::= identificador
<lista_argumentos> ::= <expresión>{, <expresión>}*
```

Por ejemplo, si queremos calcular el valor absoluto del número  $-3$ , podemos llamar a la función `abs` pasándole  $-3$  como su argumento:

```
abs(-3)
```

En esta llamada, las siguientes tres afirmaciones son equivalentes (son tres formas distintas de decir lo mismo):

- «El argumento  $-3$  se pasa a la función `abs` a través del parámetro  $x$ ».
- «El argumento  $-3$  se pasa al parámetro  $x$ ».
- «El parámetro  $x$  toma el valor  $-3$ ».

El resultado de la llamada a la función será el valor que devuelve (en este caso, el valor `3`). Por tanto, la expresión `abs(-3)` vale `3`.

Como la función `abs` está **predefinida** en Python, se puede usar directamente. Por ejemplo:

```
>>> abs(-3)
3
```

Al igual que pasa con los operadores, es importante respetar la **signatura** de una función. Eso significa, entre otras cosas, que **sus argumentos deben pertenecer al tipo correcto** (el tipo indicado por su parámetro correspondiente).

Por ejemplo, si aplicamos la función `abs` a un argumento de un tipo incorrecto (digamos, una cadena en lugar de un número), obtendremos un error `TypeError`:

```
>>> abs("hola")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: bad operand type for abs(): 'str'
>>>
```

Igualmente, el **número de argumentos** que se pasan a la función durante la llamada a la misma debe ser el correcto.

Es decir: el número de argumentos en la llamada a la función **debe coincidir con el número de parámetros** de la función.

Si pasamos a la función más o menos argumentos de los indicados en su lista de parámetros, obtendremos un error `TypeError`:

```
>>> abs()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: abs() takes exactly one argument (0 given)
>>> abs(4, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: abs() takes exactly one argument (2 given)
```

Siempre hay que respetar el número y tipo de los datos sobre los que actúa una operación, tenga ésta la forma que tenga. Por ejemplo, con los operadores también hay que hacerlo.

Otro ejemplo es la función `len`, que devuelve la longitud de una cadena (el número de caracteres que contiene). Su signatura sería:

`len(cadena: str) -> int`

Un ejemplo de llamada a la función `len`:

```
>>> len("hola")
4
```

Sabemos que hay que cumplir la signatura de la función. Por tanto, debemos pasarle un único argumento de tipo cadena. Si le pasamos más argumentos, o bien le pasamos un argumento de otro tipo,

dará error:

```
>>> len(23)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'int' has no len()
>>> len("hola", "pepe")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: len() takes exactly one argument (2 given)
```

Otro ejemplo es la función `pow`, que realiza la operación de elevar un número a la potencia de otro. Su signatura (simplificada) podría ser:

`pow(base: int, exp: int) -> int`

Curiosamente, la misma operación existe en Python de dos formas diferentes:

- Como operador (`**`):

```
>>> 2 ** 3
8
```

- Como función (`pow`):

```
>>> pow(2, 3)
8
```

- En ambos casos, la operación es exactamente la misma.

Al llamar a la función `pow` hay que tener en cuenta que tiene dos parámetros, por lo que hay que pasarle dos argumentos.

Además, hay que recordar que importa el orden al pasar los argumentos en la llamada a la función.

El primer argumento se pasaría al primer parámetro (*base*) y el segundo se pasaría al segundo (*exp*).

Por tanto, el primer argumento debe ser la base y el segundo debe ser el exponente, y no al revés.

No es lo mismo hacer `pow(2, 3)` que hacer `pow(3, 2)`:

```
>>> pow(2, 3) # aquí, el parámetro «base» toma el valor 2 y «exp» el valor 3
8
>>> pow(3, 2) # aquí, el parámetro «base» toma el valor 3 y «exp» el valor 2
9
```

Como último ejemplo, la función `max` devuelve el máximo de dos valores recibidos como argumentos:

`max(arg1, arg2)`

Aquí es más complicado definir su signatura, ya que `max` admite argumentos de varios tipos (se puede calcular el máximo de dos números, de dos cadenas... de casi cualquier par de cosas que sean *comparables* entre sí).

Por ejemplo:

```
>>> max(13, 28)
28
>>> max("hola", "pepe")
'pepe'
>>> max(2, "hola")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '>' not supported between instances of 'str' and 'int'
```

Es interesante tener en cuenta que el nombre de una función (como `abs`, `len` o `max`) es un **identificador** que **representa a la propia función**.

Es decir, que `max` (sin paréntesis detrás) es una **expresión válida cuyo valor es la propia función «máximo de dos números»**.

Si le pedimos al intérprete que nos muestre el valor de la expresión `max`, nos dice:

```
>>> max
<built-in function max>
```

(Recordemos que las funciones no tienen forma normal.)

En resumen: la expresión `max` devuelve la propia función «máximo de dos números» y la expresión `max(4, 8)` devuelve el valor `8`.

**Conclusión: las funciones también son valores.**

#### 4.3.1. Evaluación de expresiones con funciones

Una llamada a función es una **expresión válida**, por lo que podemos colocar una llamada a función en cualquier lugar donde sea sintácticamente correcto situar un valor.

La evaluación de una expresión que contiene llamadas a funciones se realiza **sustituyendo (reduciendo) cada llamada a función por su valor correspondiente**, es decir, por el valor que dicha función devuelve dependiendo de sus argumentos (su **resultado**).

Por ejemplo, en la siguiente expresión se combinan varias funciones y operadores:

```
abs(-12) + max(13, 28)
```

Aquí se llama a la función `abs` con el argumento `-12` y a la función `max` con los argumentos `13` y `28`, y finalmente se suman los dos valores obtenidos.

¿Cómo se calcula el valor de toda la expresión anterior?

En la expresión `abs(-12) + max(13, 28)` tenemos que calcular la suma de dos valores, pero esos valores aún no los conocemos porque son los resultados que se obtienen al llamar a dos funciones.

Por tanto, lo primero que tenemos que hacer es evaluar las dos subexpresiones principales que contiene dicha expresión:

- `abs(-12)`
- `max(13, 28)`

¿Cuál se evalúa primero?

En Matemáticas no importa el orden de evaluación de las subexpresiones, ya que el resultado debe ser siempre el mismo, así que da igual evaluar primero uno u otro.

Por tanto, la evaluación paso a paso de la expresión matemática anterior, podría ser de cualquiera de estas dos formas:

$$1. \left\{ \begin{array}{l} \underline{abs(-12)} + \underline{max(13, 28)} \\ = 12 + \underline{max(13, 28)} \\ = 12 + 28 \\ = 40 \end{array} \right.$$

$$2. \left\{ \begin{array}{l} \underline{abs(-12)} + \underline{max(13, 28)} \\ = \underline{abs(-12)} + 28 \\ = 12 + 28 \\ = 40 \end{array} \right.$$

En cada paso, la subexpresión subrayada es la que se va a evaluar (reducir) en el paso siguiente.

En programación funcional ocurre lo mismo que en Matemáticas, gracias a que se cumple la *transparencia referencial*.

Sin embargo, Python no es un lenguaje funcional puro, por lo que resulta importante tener en cuenta el orden de evaluación que sigue al evaluar las subexpresiones que forman una expresión.

Concretamente, ya sabemos que **Python siempre evalúa las expresiones de izquierda a derecha**.

En Python, la expresión anterior se escribe exactamente igual, ya que Python conoce las funciones *abs* y *max* (son **funciones predefinidas** en el lenguaje):

```
>>> abs(-12) + max(13, 28)
40
```

Sabiendo que Python evalúa de izquierda a derecha, la evaluación de la expresión anterior en Python sería:

```
abs(-12) + max(13, 28) # se evalúa abs (devuelve una función)
= abs(-12) + max(13, 28) # se evalúa -12 (devuelve -12)
= abs(-12) + max(13, 28) # se evalúa abs(-12) (devuelve 12)
= 12 + max(13, 28) # se evalúa max (devuelve una función)
= 12 + max(13, 28) # se evalúa 13 (devuelve 13)
= 12 + max(13, 28) # se evalúa 28 (devuelve 28)
= 12 + max(13, 28) # se evalúa max(13, 28) (devuelve 28)
= 12 + 28 # se evalúa 12 + 28 (devuelve 40)
= 40
```

### 4.3.2. Composición de operaciones y funciones

Como acabamos de ver, el resultado de una operación puede ser un dato sobre el que aplicar otra operación dentro de la misma expresión:

- En  $4 * (3 + 5)$ , el resultado de  $(3 + 5)$  se usa como operando para el operador  $*$ .
- En  $\text{abs}(-12) + \text{max}(13, 28)$ , los resultados de llamar a las funciones  $\text{abs}$  y  $\text{max}$  son los operandos del operador  $+$ .

A esto se le denomina **composición de operaciones**.

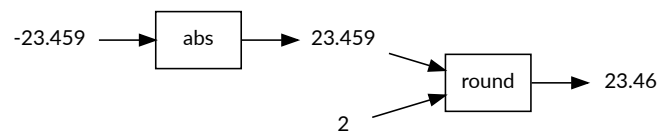
La manera más sencilla de realizar varias operaciones sobre los mismos datos es componer las operaciones, es decir, hacer que el resultado de una operación sea la entrada de otra operación.

Se va creando así una **secuencia de operaciones** donde la salida de una es la entrada de la siguiente.

Cuando el resultado de una función se usa como argumento de otra función le llamamos **composición de funciones**:

```
round(abs(-23.459), 2) # devuelve 23.46
```

En programación funcional, la composición de funciones es una técnica que ayuda a **descomponer un problema en partes** que se van resolviendo por pasos como en una **cadena de montaje**.



## 4.4. Métodos

Los **métodos** son, para la **programación orientada a objetos**, el equivalente a las **funciones** para la **programación funcional**.

En programación orientada a objetos, los objetos son **datos activos o inteligentes** a los que podemos **enviar mensajes** y que son capaces de **reaccionar y responder** ante la recepción de esos mensajes.

Si le mandamos un mensaje a un objeto, **le estamos pidiendo que ejecute un método**. Al hacerlo, el objeto está **respondiendo** a la recepción de ese mensaje.

**Los métodos son como funciones** pero, al llamar a un método, en la lista de argumentos siempre hay uno especialmente destacado, que es **el objeto que recibe la orden de ejecutar el método** y que, por tanto, es el objeto sobre el que recae directamente la acción del método.

Por eso decimos que el objeto es quien ejecuta el método, o que el método se invoca (o se llama) **sobre ese objeto**.

La gramática de las llamadas a métodos es la siguiente:

```

<llamada_método> ::= <objeto>.<método>([<lista_argumentos>])
<objeto> ::= <expresión>
<método> ::= identificador
  
```

```
⟨lista_argumentos⟩ ::= ⟨expresión⟩(, ⟨expresión⟩)*
```

Según esta gramática, las llamadas a métodos tienen esta forma:

$$o.m(a_1, a_2, \dots, a_n)$$

donde:

- $o$  es el **objeto** que recibe el mensaje (dicho de otra forma: el objeto sobre el que se ejecuta el método).
- $m$  es el **nombre** del método.
- $a_1, a_2, \dots, a_n$  son los **argumentos** del método (si los hay).

Esta llamada se puede leer de cualquiera de estas formas:

- «Se **llama** (o **invoca**) al método  $m$  sobre el objeto  $o$  con los argumentos  $a_1, a_2, \dots, a_n$ ».
- «Se **ejecuta** el método  $m$  sobre el objeto  $o$  con los argumentos  $a_1, a_2, \dots, a_n$ ».
- «Se **envía el mensaje**  $m$  al objeto  $o$  con los argumentos  $a_1, a_2, \dots, a_n$ ».

En la práctica, no habría mucha diferencia entre tener un método y hacer:

$$o.m(a_1, a_2, \dots, a_n)$$

y tener una función y hacer:

$$m(o, a_1, a_2, \dots, a_n)$$

Pero conceptualmente, hay una gran diferencia entre un estilo y otro:

- El primero es más **orientado a objetos**: decimos que el *objeto*  $o$  «recibe» un mensaje solicitando la ejecución del método  $m$ .
- En cambio, el segundo es más **funcional**: decimos que la *función*  $m$  se aplica a sus argumentos, de los cuales  $o$  es uno más.

Python es un lenguaje *multiparadigma* que soporta ambos estilos y, por tanto, dispone de funciones y de métodos. Hasta que no estudiemos la orientación a objetos, supondremos que un método es como otra forma sintáctica de escribir una función.

Por ejemplo:

Las cadenas responden al método `count`, que devuelve el número de veces que aparece una subcadena dentro de la cadena:

```
>>> 'hola caracola'.count('ol')
2
>>> 'hola caracola'.count('a')
4
```

Si `count` fuese una función en Python en lugar de un método (cosa que no ocurre), recibiría dos parámetros: la cadena y la subcadena. En tal caso, se usaría así:

```
count('hola caracola', 'ol')
```

(Esto no funciona en Python.)

Hemos visto que, para acceder a un método de un objeto, se utiliza el **operador punto** (`.`).

Es un operador *binario*, por lo que requiere dos operandos.

Su **operando izquierdo** debe ser un **objeto**:

- Técnicamente, un **objeto** es un valor estructurado que contiene elementos a los que se puede acceder mediante su nombre.
- A los elementos que contiene un objeto se les denomina **atributos** de ese objeto.
- Por tanto, **los métodos son atributos**, aunque un objeto puede tener otros atributos que no son métodos y que veremos en posteriores temas.

Su **operando derecho** debe ser el nombre de un *atributo* contenido en dicho objeto.

Dos consecuencias interesantes de lo anterior:

- El nombre de un método (como `count`) es el **identificador** de un atributo que pertenece al objeto sobre el que se ejecuta el método.
- Si `<método>` es el nombre de un método válido para el objeto `<objeto>`, entonces la expresión `<objeto>.<método>` (sin paréntesis detrás) **nos devuelve el propio método en sí**.

Por tanto, si le pedimos al intérprete que nos muestre el valor de la expresión `'hola'.count`, nos dirá algo así:

```
>>> 'hola'.count
<built-in method count of str object at 0x7f41e9cb6870>
```

(Los métodos son casos especiales de funciones y, por tanto, no tienen forma normal.)

En resumen: la expresión `'hola'.count` devuelve el propio método, y la expresión `'hola'.count('o')` devuelve el valor `1`.

**Conclusión: los métodos (como las funciones) también son valores.**

Como el método pertenece al objeto, sólo existe en el contexto de ese objeto.

Eso quiere decir que la expresión `'hola'.count` devuelve el método `count` definido sobre los objetos de tipo `str`.

En cambio, `count`, por sí solo, no es nada, ya que sería como intentar evaluar una *función* llamada `count`, que no existe en Python.

Por tanto, la evaluación de la expresión `'hola'.count('o')` se haría de la siguiente forma:



```

'hola'.count('o')      # se evalúa 'hola' (devuelve 'hola')
= 'hola'.count('o')    # se evalúa 'hola'.count (devuelve un método)
= 'hola'.count('o')    # se evalúa 'o' (devuelve 'o')
= 'hola'.count('o')    # se evalúa 'hola'.count('o') (devuelve 1)
1

```

Observar que en el segundo paso se evalúa `'hola'.count` y no `count`, ya que `count` es un atributo del objeto `'hola'` y no existe fuera de las cadenas.

## 5. Otros conceptos sobre operaciones

### 5.1. Árboles sintácticos y evaluación

Durante la fase de análisis sintáctico, el compilador o el intérprete traducen el programa fuente en una representación intermedia llamada **árbol sintáctico**.

Resulta conveniente comprender qué forma tiene ese árbol sintáctico para entender adecuadamente cómo se evalúan las expresiones y, más concretamente, en qué orden se van evaluando las subexpresiones.

En un árbol sintáctico, las hojas representan valores, mientras que los nodos intermedios representan operaciones.

Si una expresión está correctamente escrita según la sintaxis del lenguaje, sólo tendrá un único árbol sintáctico equivalente.

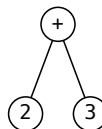
En caso contrario, es que la expresión no es sintácticamente correcta, o bien que la gramática del lenguaje no está bien diseñada.

Por ejemplo, tenemos las siguientes expresiones y sus árboles sintácticos equivalentes:

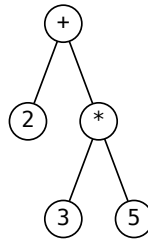
2



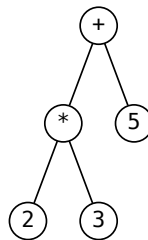
2 + 3



2 + 3 \* 5

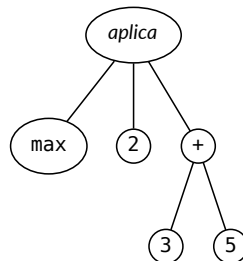


2 \* 3 + 5



Si la expresión contiene llamadas a funciones, se haría:

max(2, 3 + 5)



El nodo *aplica* representa la llamada a la función representada por su primer hijo, pasándole los argumentos representados por el resto de sus hijos.

Por tanto, tendrá tantos hijos como parámetros tenga la función, más uno (la propia función).

Para evaluar una expresión representada por su árbol sintáctico, se va recorriendo éste siguiendo un orden que dependerá del lenguaje de programación utilizado.

En Python se sigue un esquema de recorrido llamado **primero en profundidad**, donde se van visitando los nodos del árbol de izquierda a derecha y de arriba abajo, buscando siempre el nodo que está más al fondo.

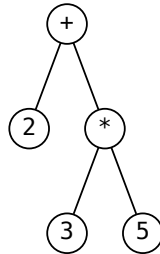
La idea es que, antes de evaluar un nodo, debemos evaluar primero todos sus nodos hijos, en orden, de izquierda a derecha.

De esta forma, para evaluar (reducir) un nodo, debemos reducir primero todos sus nodos hijo antes de reducir el propio nodo.

Si el nodo no tiene hijos, entonces se podrá evaluar directamente.

La evaluación consiste en ir sustituyendo unos subárboles por otros más reducidos hasta acabar teniendo un árbol que represente la forma normal de la expresión a evaluar.

Por ejemplo, en la expresión  $2 + 3 * 5$ , representada por este árbol:

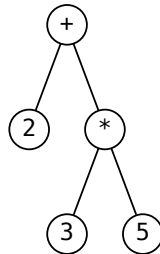


El orden en el que vamos evaluando los nodos sería el siguiente:

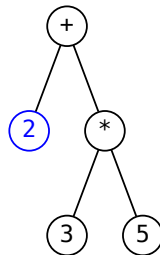
2, 3, 5, \*, +

La evaluación se realizaría de la siguiente forma, donde en azul destacamos los nodos que ya están evaluados:

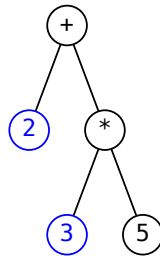
**Paso 1:** Se empieza visitando la raíz + pero, como tiene hijos, antes de evaluarlo se pasa a visitar su primer hijo (2).



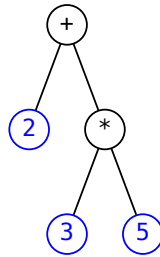
**Paso 2:** Como estamos en el nodo 2 y éste no tiene hijos, se puede evaluar directamente, ya que es un nodo hoja y, por tanto, representa un valor. La evaluación del nodo no cambia el nodo ni lo sustituye por ningún otro.



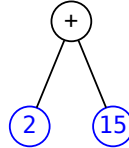
**Paso 3:** Volvemos al padre del nodo 2, que es el nodo raíz +, el cual todavía no lo podemos evaluar porque aún le queda otro nodo hijo por evaluar (el nodo \*), así que bajamos hasta él. Éste, a su vez, tampoco se puede evaluar porque tiene hijos que hay que evaluar antes, el primero de los cuales es el nodo 3, así que evaluamos 3, que se evalúa directamente ya que es un nodo hoja.



**Paso 4:** Volvemos al padre del nodo 3, que es el nodo \*, el cual todavía no lo podemos evaluar porque aún le queda otro nodo hijo por evaluar (el nodo 5), así que bajamos hasta éste, el cual se evalúa directamente ya que es un nodo hoja.



**Paso 5:** Volvemos al padre del nodo 5, que es el nodo \*, el cual ya se puede evaluar porque ya se han evaluado todos sus hijos, así que se realiza la operación 3 \* 5, dando como resultado 15, por lo que el subárbol que cuelga del nodo \* se reduce y se sustituye por un único nodo hoja 15.



**Paso 6:** Volvemos al padre del que ahora es el nodo 15, que es el nodo +, el cual ya se puede evaluar porque ya se han evaluado todos sus hijos, así que se realiza la operación 2 + 15, dando como resultado 17, por lo que el subárbol que cuelga del nodo + se reduce y se sustituye por un único nodo hoja 17.



Como ya se ha reducido el nodo raíz, la evaluación de la expresión ha terminado, dando como resultado un árbol que representa a la forma normal de la expresión inicial.

### 5.1.0.1. Importante

Recordar que este orden concreto de evaluación (*primero en profundidad*, donde se evalúan primero todos los nodos hijos antes de evaluar al nodo padre) es uno más de entre varios órdenes de evaluación que existen.

El orden de evaluación concreto que se use dependerá del lenguaje de programación utilizado.

Incluso dentro de un mismo lenguaje, podemos encontrarnos con algunas operaciones concretas que no siguen este orden de evaluación, aunque el resto de las operaciones sí lo hagan.

Por ejemplo, los operadores lógicos o el operador ternario en Python se evalúan siguiendo un orden diferente al indicado aquí, como ya veremos más adelante.

## 5.2. Tipos polimórficos y operaciones polimórficas

Hasta ahora, hemos visto que la función `abs` de Python tiene la siguiente signatura:

```
abs(x: int) -> int
```

Pero sabemos que también puede actuar sobre números reales, por lo que también podría tener la siguiente signatura:

```
abs(x: float) -> float
```

En realidad, podríamos definir la función `abs` de Python con la siguiente signatura:

```
abs(x: Number) -> Number
```

donde `Number` es un tipo que representa a todos los tipos numéricos en Python (como `int` o `float`).

Eso quiere decir que el parámetro `x` de la función `abs` admite un valor de cualquier tipo numérico, ya sea un entero o un real.

Por tanto, `Number` es un tipo que representa a varios tipos a la vez.

Cuando eso ocurre, decimos que **ese tipo es polimórfico**.

Por eso podemos afirmar que `Number` es un tipo polimórfico en Python.

De la misma forma (aunque se utiliza menos), podemos decir que un **valor polimórfico** es un valor que pertenece a un tipo polimórfico.

Asimismo, una **operación polimórfica** es aquella en cuya signatura aparece algún tipo polimórfico.

Por ejemplo, la función `abs` definida con un parámetro de tipo `Number` sería polimórfica, ya que ese parámetro tendría un tipo polimórfico.

### 5.3. Sobrecarga de operaciones

Un **mismo operador, nombre de función o nombre de método** puede representar **varias operaciones diferentes**, dependiendo del tipo de los operandos o argumentos sobre los que actúa.

Un ejemplo sencillo en Python es el operador `+`:

- Cuando actúa sobre números, representa la operación de suma:

```
>>> 2 + 3
5
```

- Cuando actúa sobre cadenas, representa la *concatenación* de cadenas:

```
>>> "hola" + "pepe"
'holapepe'
```

Cuando esto ocurre, decimos que el operador (o la función, o el método) está **sobrecargado**.

Es decir, es como si el operador `+` representara dos operaciones distintas con dos signatures distintas:

```
+(a: Number, b: Number) -> Number
```

```
+(a: str, b: str) -> str
```

de forma que, al usar el operador en una expresión del tipo:

```
<expr>1 + <expr>2
```

el intérprete llamará a una de las dos operaciones, dependiendo de los tipos de `<expr>1` y `<expr>2`.

La sobrecarga no es polimorfismo, pero induce un cierto tipo de polimorfismo que se denomina **polimorfismo *ad-hoc***.

Esto es así porque tener varias operaciones diferentes con el mismo nombre pero con distinta signature, equivale a tener una sola operación polimórfica donde algunos operandos pueden tomar un valor de varios tipos.

Por ejemplo, los tipos de `a` y `b` representarían a la vez a `Number` y `str`.

### 5.4. Equivalencia entre formas de operaciones

Una operación puede tener *forma* de **operador**, de **función** o de **método**.

También podemos encontrarnos operaciones con más de una forma.

Por ejemplo, ya vimos anteriormente la operación «*longitud*», que consiste en determinar el número de caracteres que tiene una cadena. Esta operación se puede hacer:

- Con la función `len`, pasando la cadena como argumento:

```
>>> len("hola")
4
```

- Con el método `__len__` ejecutado sobre la cadena:

```
>>> "hola".__len__()  
4
```

De hecho, en Python hay operaciones que tienen **las tres formas**. Por ejemplo, ya vimos anteriormente la operación *potencia*, que consiste en elevar un número a la potencia de otro ( $x^y$ ). Esta operación se puede hacer:

- Con el operador **`**`**:

```
>>> 2 ** 4  
16
```

- Con la función **`pow`**:

```
>>> pow(2, 4)  
16
```

- Con el método **`__pow__`**:

```
>>> (2).__pow__(4)  
16
```

La forma **más general** de representar una operación es la **función**, ya que **cualquier operación se puede expresar en forma de función** (cosa que no ocurre con los operadores y los métodos).

Los operadores y los métodos son **formas sintácticas especiales** para representar operaciones que se podrían representar igualmente mediante funciones.

Por eso, al hablar de operaciones, y mientras no se diga lo contrario, podremos suponer que están representadas como funciones.

Eso implica que los conceptos de *conjunto origen*, *conjunto imagen*, *dominio*, *rango*, *aridad*, *argumento*, *resultado*, *composición* y *asociación* (o *correspondencia*), que estudiamos cuando hablamos de las funciones, también existen en los operadores y los métodos.

Es decir: todos esos son conceptos propios de cualquier operación, da igual la forma que tenga esta.

Muchos lenguajes de programación no permiten definir nuevos operadores, pero sí permiten definir nuevas funciones (o métodos, dependiendo del paradigma utilizado).

En algunos lenguajes, los operadores son casos particulares de funciones (o métodos) y se pueden definir como tales. Por tanto, en estos lenguajes se pueden crear nuevos operadores definiendo nuevas funciones (o métodos).

## 5.5. Igualdad de operaciones

Dos operaciones son **iguales** si devuelven resultados iguales para argumentos iguales.

Este principio recibe el nombre de **principio de extensionalidad**.

**Principio de extensionalidad:**

$f = g$  si y sólo si  $f(x) = g(x)$  para todo  $x$ .

Por ejemplo: una función que calcule el doble de su argumento multiplicándolo por 2, sería exactamente igual a otra función que calcule el doble de su argumento sumándolo consigo mismo.

En ambos casos, las dos funciones devolverán siempre los mismos resultados ante los mismos argumentos.

Cuando dos operaciones son iguales, podemos usar una u otra indistintamente.

## 6. Operaciones predefinidas

### 6.1. Operadores predefinidos

#### 6.1.1. Operadores aritméticos

Operador	Descripción	Ejemplo	Resultado	Comentarios
+	Suma	3 + 4	7	
-	Resta	3 - 4	-1	
*	Producto	3 * 4	12	
/	División	3 / 4	0.75	Devuelve un <code>float</code>
%	Módulo	4 % 3 8 % 3	1 2	Resto de la división
**	Exponente	3 ** 4	81	Devuelve 3 <sup>4</sup>
//	División entera hacia abajo	4 // 3 -4 // 3	1 -2	??

#### 6.1.2. Operadores de cadenas

Operador	Descripción	Ejemplo	Resultado
+	Concatenación	'ab' + 'cd' 'ab' 'cd'	'abcd'



Operador	Descripción	Ejemplo	Resultado
<code>*</code>	Repetición	<code>'ab' * 3</code> <code>3 * 'ab'</code>	<code>'ababab'</code> <code>'ababab'</code>
<code>[0]</code>	Primer carácter	<code>'hola'[0]</code>	<code>'h'</code>
<code>[1:]</code>	Resto de cadena	<code>'hola'[1:]</code>	<code>'ola'</code>

## 6.2. Funciones predefinidas

Función	Descripción	Ejemplo	Resultado
<code>abs(n)</code>	Valor absoluto	<code>abs(-23)</code>	<code>23</code>
<code>len(cad)</code>	Longitud de la cadena	<code>len('hola')</code>	<code>4</code>
<code>max(n<sub>1</sub>(, n<sub>2</sub>)<sup>+</sup>)</code>	Valor máximo	<code>max(2, 5, 3)</code>	<code>5</code>
<code>min(n<sub>1</sub>(, n<sub>2</sub>)<sup>+</sup>)</code>	Valor mínimo	<code>min(2, 5, 3)</code>	<code>2</code>
<code>round(n[, p])</code>	Redondeo	<code>round(23.493)</code> <code>round(23.493, 1)</code>	<code>23</code> <code>23.5</code>
<code>type(v)</code>	Tipo del valor	<code>type(23.5)</code>	<code>&lt;class 'float'&gt;</code>

### 6.2.1. Funciones matemáticas y módulos

Python incluye una gran cantidad de funciones matemáticas agrupadas dentro del módulo `math`.

Los **módulos** en Python son conjuntos de funciones (y más cosas) que se pueden **importar** dentro de nuestra sesión o programa.

Son la base de la **programación modular**, que ya estudiaremos.

Para *importar* una función de un módulo se puede usar la orden **from**. Por ejemplo, para importar la función `gcd` del módulo `math` se haría:

```
>>> from math import gcd # importamos la función gcd que está en el módulo math
>>> gcd(16, 6)           # la función se usa como cualquier otra
2
```

Una vez importada, la función ya se puede usar directamente como cualquier otra.

También se puede **importar directamente el módulo en sí** usando la orden **import**.

```
>>> import math          # importamos el módulo math
```

Al importar el módulo, lo que se importan no son sus funciones, sino el propio módulo, el cual es un **objeto** (de tipo `module`) al que se accede a través de su nombre y cuyos **atributos** son (entre otras cosas) las funciones que están definidas dentro del módulo.

Por eso, para poder llamar a una función del módulo usando esta técnica, debemos indicar el nombre del módulo, seguido de un punto (.) y el nombre de la función:

```
>>> import math          # importamos el módulo math
>>> math.gcd(16, 6)      # la función gcd sigue estando dentro del módulo
2
```

```
math.gcd(16, 6)
└──┬──┬── función
   │   │
   └───┴── módulo
```

Eso significa que podríamos ampliar nuestra gramática para permitir que el nombre de una función en una llamada pudiera contener la parte del módulo:

```
<llamada_función> ::= <función>([<lista_argumentos>])
<función> ::= [<módulo>].<identificador>
<módulo> ::= <identificador>
```

Pero técnicamente no es necesario, ya que las funciones contenidas en un módulo se invocan como si fueran **métodos que se ejecutan sobre el objeto módulo**, por lo que la sintaxis es la misma que para los métodos y está ya recogida en nuestra gramática:

```
<llamada_método> ::= <objeto>.<método>([<lista_argumentos>])
<objeto> ::= <expresión>
<método> ::= <identificador>
```

Esto nos dice que hay una relación muy estrecha entre funciones y métodos (de hecho, los métodos son funciones que se invocan de una forma especial).

De hecho, cuando el objeto es un módulo, no hablamos de métodos sino de funciones (los módulos no contienen métodos).

No es lo mismo `math`, que `math.gcd`, que `math.gcd(16, 6)`:

- `math` es un *módulo* (un objeto de tipo `module`).
- `math.gcd` es una *función* (no es un método porque `math` es un módulo).
- `math.gcd(16, 6)` es una *llamada a función*.

```
>>> import math
>>> math
<module 'math' (built-in)>
>>> math.gcd
<built-in function gcd>
>>> math.gcd(16, 6)
```

2

La lista completa de funciones que incluye el módulo `math` se puede consultar en su documentación: <https://docs.python.org/3/library/math.html>

El lenguaje Python es, principalmente, un lenguaje **orientado a objetos**.

De hecho, **todos los datos en Python son objetos** que tienen sus propios atributos (métodos, entre otros) a los que se le puede acceder usando el operador punto (`.`).

Por ello, en Python los términos «dato», «valor» y «objeto» son sinónimos en la práctica.

Los números, las cadenas, los módulos, las funciones... todos son objetos.

Incluso los métodos son objetos, ya que, en realidad, son funciones contenidas dentro de otros objetos, y las funciones son objetos.

Hasta los tipos (como `int` o `str`) son objetos que tienen sus propios atributos.

Entraremos a estudiar más en detalle estas características cuando veamos la **programación orientada a objetos**.

#### 6.2.1.1. El módulo `operator`

El módulo `operator` contiene, en forma de funciones, las operaciones básicas que hasta ahora hemos utilizado en forma de operadores:

Operador	Operación	Función en el módulo <code>operator</code>
<code>+</code>	Suma	<code>add</code>
<code>-</code>	Resta	<code>sub</code>
<code>-</code>	Cambio de signo	<code>neg</code>
<code>*</code>	Multiplicación	<code>mul</code>
<code>/</code>	División	<code>truediv</code>
<code>%</code>	Módulo	<code>mod</code>
<code>**</code>	Exponente	<code>pow</code>
<code>//</code>	División entera hacia abajo	<code>floordiv</code>

Gracias al módulo `operator`, podemos reescribir con funciones las expresiones que utilizan operadores.

Por ejemplo, la expresión:

```
>>> 3 * (4 + 5) - 10
17
```

se puede reescribir como:

```
>>> from operator import add, mul, sub
>>> sub(mul(3, add(4, 5)), 10)
17
```

Pasar los operadores de una expresión a funciones es un ejercicio muy interesante que ayuda a entender en qué orden se evalúan las subexpresiones y por qué.

En Python, en una llamada a función, los argumentos se evalúan siempre antes que la propia llamada (y de izquierda a derecha).

La expresión `3 * (4 + 5) - 10` se evalúa así:

```
3 * (4 + 5) - 10      # se evalúa 3 (devuelve 3)
= 3 * (4 + 5) - 10    # se evalúa 4 (devuelve 4)
= 3 * (4 + 5) - 10    # se evalúa 5 (devuelve 5)
= 3 * (4 + 5) - 10    # se evalúa (4 + 5) (devuelve 9)
= 3 * 9 - 10          # se evalúa 3 * 9 (devuelve 27)
= 27 - 10             # se evalúa 27 - 10 (devuelve 17)
= 17
```

Y la expresión `sub(mul(3, add(4, 5)), 10)` se evalúa así:

```
sub(mul(3, add(4, 5)), 10) # se evalúa sub (devuelve la función resta)
= sub(mul(3, add(4, 5)), 10) # se evalúa mul (devuelve la función multiplicación)
= sub(mul(3, add(4, 5)), 10) # se evalúa 3 (devuelve 3)
= sub(mul(3, add(4, 5)), 10) # se evalúa add (devuelve la función suma)
= sub(mul(3, add(4, 5)), 10) # se evalúa 4 (devuelve 4)
= sub(mul(3, add(4, 5)), 10) # se evalúa 5 (devuelve 5)
= sub(mul(3, add(4, 5)), 10) # se evalúa add(4, 5) (devuelve 9)
= sub(mul(3, 9), 10)         # se evalúa mul(3, 9) (devuelve 27)
= sub(27, 10)               # se evalúa 10 (devuelve 10)
= sub(27, 10)               # se evalúa sub(27, 10) (devuelve 17)
= 17
```

### 6.3. Métodos predefinidos

Igualmente, en la documentación podemos encontrar una lista de métodos interesantes que operan sobre cadenas:

<https://docs.python.org/3/library/stdtypes.html#string-methods>

## Bibliografía

- Abelson, Harold, Gerald Jay Sussman, and Julie Sussman. 1996. *Structure and Interpretation of Computer Programs*. 2nd ed. Cambridge, Mass. : New York: MIT Press ; McGraw-Hill.
- Blanco, Javier, Silvina Smith, and Damián Barsotti. 2009. *Cálculo de Programas*. Córdoba, Argentina: Universidad Nacional de Córdoba.
- Python Software Foundation. n.d. "Sitio Web de Documentación de Python." <https://docs.python.org/3>.

Van-Roy, Peter, and Seif Haridi. 2004. *Concepts, Techniques, and Models of Computer Programming*. Cambridge, Mass: MIT Press.