

Expresiones

Ricardo Pérez López

IES Doñana, curso 2020/2021

Índice general

1. El lenguaje de programación Python	2
1.1. Historia	2
1.2. Características principales	2
2. Elementos de un programa	3
2.1. Expresiones y sentencias	3
2.2. Conceptos básicos	4
3. Valores	5
3.1. Evaluación de expresiones	5
3.2. Expresión canónica y forma normal	6
3.3. Formas normales y evaluación	6
3.4. Literales	8
3.5. Identificadores	9
4. Operaciones	9
4.1. Clasificación	9
4.2. Operadores	10
4.2.1. Aridad de operadores	10
4.2.2. Paréntesis	11
4.2.3. Prioridad de operadores	11
4.2.4. Asociatividad de operadores	12
4.2.5. Tipos de operandos	13
4.3. Funciones	13
4.3.1. Evaluación de expresiones con funciones	17
4.3.2. Composición de operaciones y funciones	18
4.4. Métodos	19
5. Otros conceptos sobre operaciones	20
5.1. Sobrecarga de operaciones	20
5.2. Equivalencia entre formas de operaciones	21
5.3. Igualdad de operaciones	23
6. Operaciones predefinidas	23

6.1. Operadores predefinidos	23
6.1.1. Operadores aritméticos	23
6.1.2. Operadores de cadenas	23
6.2. Funciones predefinidas	24
6.2.1. Funciones matemáticas	24
6.3. Métodos predefinidos	25

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**, **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 imperativa** 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.

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

```
$ python
Python 3.8.2 (default, Apr 27 2020, 15:53:34)
[GCC 9.3.0] 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`.

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 elementos que pertenecen a dos grandes grupos principales:

- **Expresiones:** son secuencias de símbolos que *representan valores* y que 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 *instrucciones* que sirven para pedirle al intérprete que *ejecute* una determinada *acción*. Las sentencias también pueden contener expresiones.

En un programa hay $\left\{ \begin{array}{l} \text{Expresiones, formadas por} \left\{ \begin{array}{l} \text{Datos} \\ \text{Operaciones} \end{array} \right. \\ \text{Sentencias, que también pueden contener expresiones} \end{array} \right.$

Las **expresiones** se *evalúan*.

Las **sentencias** se *ejecutan*.

2.2. Conceptos básicos

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, 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 **16**.

Las expresiones correctamente formadas deben satisfacer la gramática del lenguaje en el que están escritas.

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> ::= <nombre_función> ( [ <lista_argumentos> ] )
<nombre_función> ::= identificador
<llamada_método> ::= <expresión> . identificador ( [ <lista_argumentos> ] )
<lista_argumentos> ::= <expresión> { , <expresión> } *

```

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

$(3 + (4 - 7))$

Algunos ejemplos de expresiones que genera dicha gramática:

- 24

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

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.

3. Valores

3.1. 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.

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

La **evaluación de una expresión**, en esencia, es el proceso de **sustituir**, dentro de ella, unas *subexpresiones* por otras que, de alguna manera bien definida, estén *más cerca* del valor a calcular, y así hasta calcular el valor de la expresión al completo.

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 ciertos paradigmas como el *imperativo*.

Podemos decir que las expresiones:

3
 $(1 + 2)$
 $(5 - 2)$

denotan todas el mismo valor (el número abstracto **3**).

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*.

3.2. 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 2 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.

3.2.0.1. Ejemplo

De las expresiones anteriores:

3

(1 + 2)

(5 - 2)

que denotan todas el mismo valor abstracto **3**, 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 **3**).

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

Hay valores que no tienen expresión canónica:

- Las funciones (los valores de tipo *función*).
- El número π 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 $\frac{1}{0}$.

3.3. Formas normales y evaluación

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

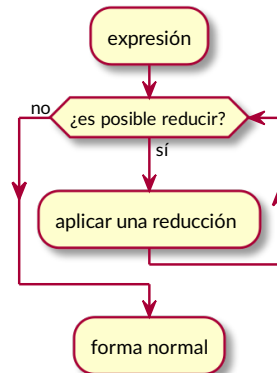
Para ello, el intérprete evalúa la expresión reduciendo sus subexpresiones según las reglas del lenguaje y las operaciones que aparecen en ellas, buscando su forma normal.

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.

Importante (orden de evaluación de las expresiones):

Al analizar una expresión buscando subexpresiones que reducir, las subexpresiones siempre se evaluarán **de izquierda a derecha**.



3.3.0.1. 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
  
```

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 8)
= (2 * 8)      # se evalúa (2 * 8) (que devuelve 16)
= 16
  
```

= 17

3.4. Literales

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

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

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.).

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	Números reales	Cadenas
-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.
- Las cadenas van siempre entre comillas (simples `'` o dobles `"`).

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 **3.5**, 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 **3.5**.

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 **3.5**. 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 distintas de escribir un literal que denote el valor **3.5**, 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 **2** es el literal **2**.

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

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 **2** 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.

3.5. Identificadores

Los **identificadores** son nombres que representan valores u operaciones.

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

Los identificadores deben cumplir unas reglas sintácticas que dependen del lenguaje de programación, pero que 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

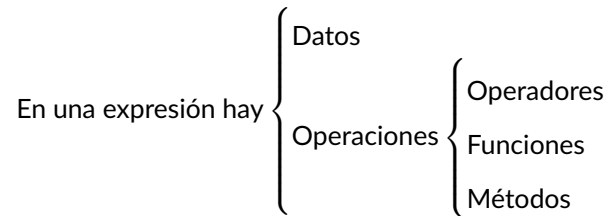
4.1. Clasificación

En una expresión puede haber:

- **Datos**
- **Operaciones** a realizar sobre esos datos

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

- Operadores
- Funciones
- Métodos



4.2. Operadores

Un **operador** es un símbolo o palabra clave que representa la realización de una *operación* sobre unos datos llamados **operandos**.

Ejemplos:

- Los operadores aritméticos: `+`, `-`, `*`, `/` (entre otros):

```
( 3 + 4 )
```

(aquí los operandos son los números 3 y 4)

```
( 9 * 8 )
```

(aquí los operandos son los números 9 y 8)

- El operador `in` para comprobar si un carácter pertenece a una cadena:

```
("c" in "barco")
```

(aquí los operandos son las cadenas "c" y "barco")

4.2.1. Aridad de operadores

Los operadores se clasifican en función de la cantidad de operandos sobre los que operan en:

- Unarios:** operan sobre un único operando.

Ejemplo: el operador `-` que cambia el signo de su operando:

```
( -( 5 + 3 ) )
```

- Binarios:** operan sobre dos operandos.

Ejemplo: la mayoría de operadores aritméticos.

- Ternarios:** operan sobre tres operandos.

Veremos un ejemplo más adelante.

4.2.2. 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, se puede cambiar nuestra gramática:

- Quitando los paréntesis más externos que rodean a toda la expresión.
- Acudiendo a un esquema de **prioridades** y **asociatividades** de operadores.

Así ya no exigiremos que las expresiones estén totalmente parentizadas.

4.2.3. Prioridad de operadores

En ausencia de paréntesis, cuando un operando está afectado a derecha e izquierda por **distinto operador**, se aplican las reglas de la **prioridad**:

$8 + 4 * 2$

El 4 está afectado a derecha e izquierda por distintos operadores (+ y *), por lo que se aplican las reglas de la prioridad. El * tiene *más prioridad* que el +, así que agrupa primero el *. Equivale a hacer:

$8 + (4 * 2)$

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>.

4.2.4. Asociatividad de operadores

En ausencia de paréntesis, cuando un operando está afectado a derecha e izquierda por el **mismo operador** (o distintos operadores con la **misma prioridad**), 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 agrupa primero 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.

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 operador más prioritario es el `*` y, por tanto, habría que hacer primero el producto antes que la suma.

El problema es que no podemos hacer el producto hasta haber calculado primero la suma de 3 y 5.

Por eso el intérprete calcula primero la suma y finalmente hace el producto, pero su intención era hacer primero el producto, según las reglas de la prioridad.

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
```

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 5 (que devuelve 5)
= 5 * (4 + 5)        # se evalúa (4 + 5) (que devuelve 9)
= 5 * 9              # se evalúa 5 * 9 (que devuelve 45)
45
```

4.2.5. Tipos de operandos

Es importante respetar el tipo de los operandos que espera recibir un operador. Si los intentamos aplicar 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 `/`:

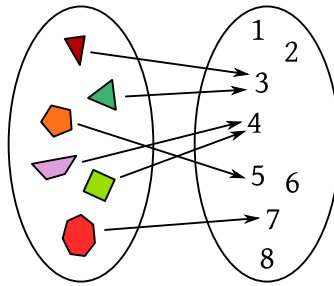
```
>>> "hola" / "pepe"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for /: 'str' and 'str'
>>>
```

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

4.3. Funciones

Las funciones son una forma de representar operaciones.

Matemáticamente, una **función** es una regla que **asocia** a cada elemento de un conjunto (el conjunto *origen* o *dominio*) **un único elemento** de un segundo conjunto (el conjunto *imagen*, *rango* o *codominio*).



Función que asocia a cada polígono con su número de lados

En Programación, el concepto de *función* es similar al de una función matemática, aunque con su propia terminología y funcionamiento.

En Programación, las funciones son operaciones que actúan sobre unos datos de entrada llamados **argumentos** y que **devuelven un resultado** que depende de la operación a realizar y de los datos recibidos como argumentos.

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

Las funciones se definen mediante su **signatura**, la cual informa de:

- El nombre de la función.
- El número, tipo y posición de sus parámetros.
- El tipo del resultado que devuelve.

Por ejemplo, la función `abs` está predefinida en Python y tiene la siguiente signatura:

`abs(x: Number) -> Number`

Esa signatura nos dice que:

- La función se llama `abs`.
- Recibe un único parámetro llamado `x` que puede tomar cualquier valor numérico.
- Devuelve un resultado numérico.

La **aplicación de una función a unos argumentos** es una expresión mediante la cual solicitamos la realización de la operación correspondiente pasándole en forma de argumentos los datos sobre los que deseamos que actúe la operación.

A la aplicación de una función también se la llama **llamada** o **invocación** de 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.

Debe haber tantos argumentos como parámetros, ni más ni menos.

Sintácticamente, la llamada a una función tiene esta forma:

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

Por ejemplo, si queremos calcular el valor absoluto del número -35 , podemos llamar a la función `abs` pasándole el `-35` como su argumento:

```
abs(-35)
```

El argumento `-35` se pasará a la función a través de su parámetro `x`. Por tanto, se puede decir que, en esta llamada, el parámetro `x` toma el valor `-35`.

El resultado de la llamada a la función será el valor que devuelve. En este caso, el valor `35`.

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

```
>>> abs(-35)  
35
```

Al igual que pasa con los operadores, es importante respetar la signature de una función. Si la aplicamos a un argumento de un tipo incorrecto (por ejemplo, una cadena en lugar de un número), obtendremos un error:

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

Otro ejemplo es la función `len`, que devuelve la longitud de una cadena, es decir, el número de caracteres que contiene. Su signature podría ser:

```
len(obj: str)->int
```

Un ejemplo de llamada a la función `len`:

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

Siempre hay que cumplir la signature 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)
```

Esto es común a cualquier tipo de operación, tenga la forma que tenga. Por ejemplo, con los operadores también hay que cumplirlo.

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

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

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 tanto, 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 (*exponente*).

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

4.3.1. Evaluación de expresiones con funciones

Una llamada a función es una expresión válida, por lo que podemos colocar llamadas 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.

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 el resultado de 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:

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

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

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, y llegado el momento será importante tener en cuenta el orden de evaluación que sigue al evaluar las subexpresiones que forman una expresión.

Por eso, no debemos olvidar 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 -12 (devuelve -12)
abs(-12) + max(13, 28) # se evalúa abs(-12) (devuelve 12)
= 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 `abs(-12) + max(13, 28)`, los resultados de llamar a las funciones `abs` y `max` son los operandos del operador `+`.

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

El orden en el que se evalúan este tipo de expresiones (aquellas que contienen composición de operaciones) es algo que se estudiará más en profundidad en el siguiente tema.

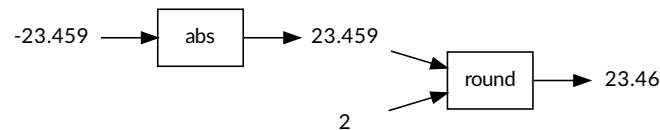
La manera más sencilla de realizar varias operaciones sobre los mismos datos es componer las operaciones, es decir, hacer que el resultado de una operación sea la entrada de otra operación.

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**.

Los métodos son como funciones, pero en este caso se dice que actúan *sobre* un valor, que es el **objeto** sobre el que recae la acción del método.

La *aplicación* de un método se denomina **invocación** o **llamada** al método, y se escribe:

$$v.m()$$

que representa la **llamada** al método *m* **sobre el objeto** *v*.

La gramática de las llamadas a métodos es la siguiente:

```

<llamada_método> ::= <expresión>.identificador([<lista_argumentos>])
<lista_argumentos> ::= <expresión>[, <expresión>]*
  
```

Esta llamada se puede leer de cualquiera de estas formas:

- Se **llama** (o **invoca**) al método *m* sobre el objeto *v*.
- Se **ejecuta** el método *m* sobre el objeto *v*.
- Se **envía el mensaje** *m* al objeto *v*.

Los métodos también pueden tener argumentos, como cualquier función:

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

y en tal caso, los argumentos se pasarían al método correspondiente.

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

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

y tener una función y hacer:

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

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

- El primero es más **orientado a objetos**: decimos que el *objeto* v «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 v es uno más.

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

Por ejemplo:

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

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

devuelve 2.

```
'hola caracola'.count('a')
```

devuelve 4.

Si `count` fuese una función en lugar de un método, recibiría dos parámetros: la cadena y la subcadena. En tal caso, se usaría así:

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

5. Otros conceptos sobre operaciones

5.1. Sobrecarga de operaciones

Un **mismo operador** (o nombre de función o 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**.

5.2. 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`:

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

- Con el método `__len__`:

```
>>> "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
```

Otro ejemplo es la operación *contiene*, que consiste en comprobar si una cadena contiene a otra (una *subcadena*). Esa operación también tiene tres formas:

- El operador `in`:

```
>>> "o" in "hola"
True
```

- La función `str.__contains__`:

```
>>> str.__contains__("hola", "o")
True
```

- El método `__contains__` ejecutado sobre la cadena (y pasando la subcadena como argumento):

```
>>> "hola".__contains__("o")
True
```

Observar que, en este caso, el objeto que recibe el mensaje (es decir, el objeto al que se le pide que ejecute el método) es la cadena `"hola"`. Es como si le preguntáramos a la cadena `"hola"` si contiene la subcadena `"o"`.

Y la suma de dos números enteros se puede expresar:

- Mediante el operador `+`:

```
4 + 3
```

- Mediante la función `int.__add__`:

```
int.__add__(4, 3)
```

- Mediante el método `__add__` ejecutado sobre uno de los enteros (y pasando el otro número como *argumento* del método):

```
(4).__add__(3)
```

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 *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).

5.3. 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	1	Resto de la división
		8 % 3	2	
**	Exponente	3 ** 4	81	Devuelve 3 ⁴
//	División entera hacia abajo	4 // 3	1	
		-4 // 3	-2	??

6.1.2. Operadores de cadenas

Operador	Descripción	Ejemplo	Resultado
+	Concatenación	'ab' + 'cd' 'ab'	'abcd'
		'cd'	
*	Repetición	'ab' * 3 3 * 'ab'	'ababab'
			'ababab'

Operador	Descripción	Ejemplo	Resultado
<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>	23
<code>len(cad)</code>	Longitud de la cadena	<code>len('hola')</code>	4
<code>max(n₁(, n₂)*)</code>	Valor máximo	<code>max(2, 5, 3)</code>	5
<code>min(n₁(, n₂)*)</code>	Valor mínimo	<code>min(2, 5, 3)</code>	2
<code>round(n[, p])</code>	Redondeo	<code>round(23.493)</code> <code>round(23.493, 1)</code>	23 23.5
<code>type(v)</code>	Tipo del valor	<code>type(23.5)</code>	<code><class 'float'></code>

6.2.1. Funciones matemáticas

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` (que calcula el máximo común divisor de dos números) 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 como cualquier otra.

También se puede importar directamente el módulo en sí:

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

Al importar el módulo, lo que se importan no son sus funciones, sino el nombre y la definición del propio módulo, el cual contiene dentro la definición de sus funciones.

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:

```
math.gcd(16, 6)
```

El punto . es un operador que nos permite acceder al interior de estructuras que tienen definiciones propias, como los módulos.

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

```
<nombre_función> ::= [identificador.]identificador
```

La lista completa de funciones que incluye el módulo `math` se puede consultar en su documentación:

<https://docs.python.org/3/library/math.html>

6.3. Métodos predefinidos

Igualmente, en la documentación podemos encontrar una lista de métodos interesantes que operan con datos de tipo cadena:

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

Ejercicios

Ejercicios

- Representar la evaluación las siguientes expresiones, aplicando paso a paso la reducción que corresponda. Indicar también el tipo del valor resultante:

- $3 + 6 * 14$
- $8 + 7 * 3.0 + 4 * 6$
- $-4 * 7 + 2 ** 3 / 4 - 5$
- $4 / 2 * 3 / 6 + 6 / 2 / 1 / 5 ** 2 / 4 * 2$

- Convertir en expresiones aritméticas algorítmicas las siguientes expresiones algebraicas:

- $5 \cdot (x + y)$
- $a^2 + b^2$
- $\frac{x+y}{u+\frac{w}{a}}$
- $\frac{x}{y} \cdot (z + w)$

- Determinar, según las reglas de prioridad y asociatividad del lenguaje Python, qué paréntesis sobran en las siguientes expresiones. Reescribirlas sin los paréntesis sobrantes. Calcular su valor y deducir su tipo:

- a. `(8 + (7 * 3) + 4 * 6)`
 - b. `-(2 ** 3)`
 - c. `(33 + (3 * 4)) / 5`
 - d. `2 ** (2 * 3)`
 - e. `(3.0) + (2 * (18 - 4 ** 2))`
 - f. `(16 * 6) - (3) * 2`
5. Usar la función `math.sqrt` para escribir dos expresiones en Python que calculen las dos soluciones a la ecuación de segundo grado $ax^2 + bx + c = 0$.

Recordar que las soluciones son:

$$x_1 = -b + \frac{\sqrt{b^2 - 4ac}}{2a}, \quad x_2 = -b - \frac{\sqrt{b^2 - 4ac}}{2a}$$

6. Evaluar las siguientes expresiones:
- a. `9 - 5 - 3`
 - b. `2 // 3 + 3 / 5`
 - c. `9 // 2 / 5`
 - d. `7 % 5 % 3`
 - e. `7 % (5 % 3)`
 - f. `(7 % 5) % 3`
 - g. `(7 % 5 % 3)`
 - h. `((12 + 3) // 2) / (8 - (5 + 1))`
 - i. `12 / 2 * 3`
 - j. `math.sqrt(math.cos(4))`
 - k. `math.cos(math.sqrt(4))`
 - l. `math.trunc(815.66) + round(815.66)`
7. Escribir las siguientes expresiones algorítmicas como expresiones algebraicas:
- a. `b ** 2 - 4 * a * c`
 - b. `3 * x ** 4 - 5 * x ** 3 + x * 12 - 17`
 - c. `(b + d) / (c + 4)`
 - d. `(x ** 2 + y ** 2) ** (1 / 2)`

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