Programación funcional (I)

Ricardo Pérez López

IES Doñana, curso 2025/2026

Generado el 2025/07/10 a las 17:29:00

Índice

1. Introducción

1.1. Concepto

La **programación funcional** es un paradigma de programación declarativa basado en el uso de **definiciones**, **expresiones** y **funciones matemáticas**.

Tiene su origen teórico en el **cálculo lambda**, un sistema matemático creado en 1930 por Alonzo Church.

Los lenguajes funcionales se pueden considerar *azúcar sintáctico* (es decir, una forma equivalente pero sintácticamente más sencilla) del cálculo lambda.

En programación funcional, una función define un cálculo a realizar a partir de unos datos de entrada, con la propiedad de que el resultado de la función sólo puede depender de esos datos de entrada.

Eso significa que una función no puede tener estado interno ni su resultado puede depender del estado del programa.

Además, una función no puede producir ningún efecto observable fuera de ella (los llamados **efectos laterales**), salvo calcular y devolver su resultado.

Esto quiere decir que en programación funcional no existen los efectos laterales, o se dan de forma muy localizada en partes muy concretas e imprescindibles del programa.

Por todo lo expuesto anteriormente, se dice que las funciones en programación funcional son **funciones puras**, es decir, funciones que lo único que hacen es calcular su resultado (sin ningún otro efecto) y en las que ese resultado sólo depende de los datos de entrada.

Además, los valores nunca cambian porque no tienen estado interno que se pueda alterar con el tiempo.

Como consecuencia de todo lo anterior, en programación funcional es posible sustituir cualquier expresión por su valor, propiedad que se denomina **transparencia referencial**.

En programación funcional, las funciones también son valores, por lo que se consideran a éstas como **ciudadanas de primera clase**.

Un programa funcional está formado únicamente por definiciones de valores y por expresiones que hacen uso de los valores definidos.

Por tanto, en programación funcional, ejecutar un programa equivale a evaluar una expresión.

Para describir el proceso llevado a cabo por el programa no es necesario bajar al nivel de la máquina, sino que basta con interpretarlo como un **sistema de evaluación de expresiones**.

Esa evaluación de expresiones se lleva a cabo mediante **reescrituras** que usan las definiciones para tratar de alcanzar la forma normal de la expresión.

1.2. Transparencia referencial

En programación funcional, **el valor de una expresión depende, exclusivamente, de los valores de las subexpresiones que la forman**.

Dichas subexpresiones, además, pueden ser sustituidas libremente por otras que tengan el mismo valor.

A esta propiedad se la denomina **transparencia referencial**.

Formalmente, se puede definir así:

Transparencia referencial:

Si $p = q$, entonces $f(p) = f(q)$.

En la práctica, eso significa que la evaluación de una expresión no puede provocar **efectos laterales** y que su valor no puede depender del momento en el que se evalúe la expresión (**la expresión siempre va a valer lo mismo**).

En consecuencia, un requisito para conseguir la transparencia referencial es que las expresiones no cambien de valor dependiendo de cuándo se evalúen.

Es decir: una expresión en programación funcional siempre debe tener el mismo valor.

Por tanto, en programación funcional no se permite que la misma expresión, evaluada en dos momentos diferentes, devuelva como resultado dos valores diferentes.

Asimismo, el valor de una expresión tampoco debe depender del orden en el que se evalúen sus subexpresiones.

1.2.1. Efectos laterales

Los **efectos laterales** son aquellos que provocan un cambio de estado irremediable en el sistema, que además son observables fuera del contexto donde se producen y que puede dar lugar a que una misma expresión tenga dos valores según el momento en el que se evalúe.

Por ejemplo, las **instrucciones de E/S** (entrada y salida) provocan efectos laterales, ya que:

- Al **leer un dato** de la entrada (ya sea el teclado, un archivo del disco, una base de datos...) estamos afectando al estado del dispositivo de entrada, y además no se sabe de antemano qué valor se va a recibir, ya que éste proviene del exterior y no lo controlamos.

- Al **escribir un dato** en la salida (ya sea la pantalla, un archivo, una base de datos...) estamos realizando un cambio que afecta irremediablemente al estado del dispositivo de salida.

En posteriores temas veremos que existe un paradigma (el **paradigma imperativo**) que se basa principalmente en provocar efectos laterales.

Uno de los requisitos para alcanzar la transparencia referencial es que no existan efectos laterales.

Por tanto, en programación funcional no están permitidos los efectos laterales.

Eso significa que:

- Al evaluar una expresión no se pueden provocar efectos laterales.

  Si esto ocurriera, no podríamos sustituir una expresión por su valor.

- El valor de una expresión no puede depender de un efecto lateral ni verse afectado por la existencia de efectos laterales.

  Si esto ocurriera, la expresión podría tener valores distintos en momentos distintos.

En cualquiera de los dos casos, se rompería la transparencia referencial.

1.3. Modelo de ejecución

Cuando escribimos programas (y algoritmos) nos interesa abstraernos del funcionamiento detallado de la máquina que va a ejecutar esos programas.

Nos interesa buscar una *metáfora*, un símil de lo que significa ejecutar el programa.

De la misma forma que un arquitecto crea modelos de los edificios que se pretenden construir, los programadores podemos usar modelos que *simulan* en esencia el comportamiento de nuestros programas.

Esos modelos se denominan **modelos computacionales** o **modelos de ejecución**.

Los modelos de ejecución nos permiten **razonar** sobre los programas sin tener que ejecutarlos.

Definición:

> **Modelo de ejecución:**
>
> Es una herramienta conceptual que permite a los programadores razonar sobre el funcionamiento de un programa sin tener que ejecutarlo directamente en el ordenador.

Podemos definir diferentes modelos de ejecución dependiendo, principalmente, de:

- El paradigma de programación utilizado (ésto sobre todo).
- El lenguaje de programación con el que escribamos el programa.
- Los aspectos que queramos estudiar de nuestro programa.

1.3.1. Modelo de sustitución

En programación funcional, **un programa es una expresión** y lo que hacemos al ejecutarlo es **evaluar dicha expresión**, usando para ello las definiciones de operadores y funciones predefinidas por el lenguaje, así como las definidas por el programador en el código fuente del programa.

Recordemos que 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.

Por ello, la ejecución de un programa funcional se puede modelar como un **sistema de reescritura** al que llamaremos **modelo de sustitución**.

El modelo de sustitución es un buen modelo de ejecución para la programación funcional gracias a que se cumple la *transparencia referencial*.

La ventaja del modelo de sustitución es que no necesitamos recurrir a pensar que debajo de todo esto hay un ordenador con una determinada arquitectura *hardware*, que almacena los datos en celdas de la memoria principal, que ejecuta ciclos de instrucción en la CPU, que las instrucciones modifican los datos de la memoria…

Todo resulta mucho más fácil que eso, ya que **todo se reduce a evaluar expresiones**, reescribiendo unas subexpresiones por otras, sin importar aspectos secundarios como la tecnología, el momento en el que se evalúan, el orden en el que se evalúan, etc.

Y la evaluación de expresiones no requiere pensar que hay un ordenador que lleva a cabo el proceso de evaluación.

Esto se debe a que la programación funcional se basa en el **cálculo lambda**, que es un modelo teórico matemático.

Ya estudiamos que evaluar una expresión consiste en encontrar su forma normal.

En programación funcional:

- Los intérpretes alcanzan este objetivo a través de múltiples pasos de **reducción** de las expresiones para obtener otra equivalente más simple.
- Toda expresión posee un valor definido, y ese valor **no depende del orden ni el momento** en el que se evalúe.
- El significado de una expresión es su valor, y **no puede ocurrir ningún otro efecto**, ya sea oculto o no, en ninguna operación que se utilice para calcularlo.

2. Tipos de datos

2.1. Concepto

Los **valores** que comparten características y propiedades comunes se agrupan en **conjuntos** llamados **tipos de datos** o, simplemente, **tipos**.

Por tanto, un **tipo** (o **tipo de datos**) es un conjunto de valores:

Tipo (de datos):

Es un conjunto de **valores**.

Se dice que «*un valor pertenece a un tipo*» cuando pertenece a ese conjunto (es decir, cuando es uno de los elementos de ese conjunto).

Por ejemplo, el tipo *entero* representa el conjunto de los números enteros. Los diferentes números enteros pertenecen al tipo *entero*.

En general, las **operaciones** se definen de forma que sólo pueden actuar sobre valores de determinados tipos.

O dicho de otra forma: esas son las operaciones que tiene sentido realizar sobre esos valores.

Esto es así porque recordemos que las operaciones actúan como funciones que están definidas sobre un **dominio**, que es un subconjunto del **conjunto origen**. Ese conjunto origen sería, a grandes rasgos, el tipo de los valores sobre los que puede actuar.

Por ejemplo: sobre un valor de tipo *cadena* se puede realizar la operación *longitud* (pero no la *raíz cuadrada*), y sobre dos *enteros* se pueden realizar las operaciones de *suma* y *producto*.

En resumen, podemos decir que **un tipo**:

- Es un conjunto de **valores**.
- Que, indirectamente, define también el conjunto de **operaciones** válidas que se pueden realizar sobre dichos valores.

2.2. Tipo de un dato

Recordemos que un dato puede tomar valores.

Por extensión, el **tipo de un dato** es el conjunto de los posibles valores que puede tomar ese dato.

En cierta forma, el tipo de un dato es como una etiqueta, característica o atributo que va asociado al dato y que define una cualidad muy importante del mismo.

Se dice que «*un dato* ***es de*** *un tipo*», o que «*un dato* ***tiene*** *un tipo*» o que «*un dato* ***pertenece a*** *un tipo*» cuando ese dato tiene (o puede tener) un valor de ese tipo.

Dependiendo del lenguaje de programación utilizado, el tipo de un dato puede venir definido:

- *Implícitamente*, como **el tipo del valor que tiene actualmente** el dato.
- *Explícitamente*, asociando el tipo al dato mediante una instrucción especial llamada **declaración**.

Como los datos tienen (o representan) valores, **las operaciones también pueden actuar sobre datos**.

En realidad, lo que hacen las operaciones es actuar sobre los valores que tienen esos datos.

Por tanto, también se puede decir que **las operaciones que se pueden realizar sobre un dato dependen del *tipo* de ese dato**.

Y, por extensión, podemos decir que **un tipo de datos**:

- Es el conjunto de **valores** que puede tomar un dato de ese tipo.

- Que, indirectamente, define también el conjunto de **operaciones** válidas que se pueden realizar sobre datos de ese tipo.

Definiciónes ampliadas:

Tipo (o tipo de datos):

Es un conjunto de **valores** que, indirectamente, define también el conjunto de **operaciones** que se pueden realizar sobre esos valores.

Tipo de un dato:

Es el tipo que tiene ese dato, es decir, una característica o atributo del dato que define el conjunto de **valores** que puede tomar ese dato y, en consecuencia, también las **operaciones** que se pueden realizar sobre ese dato.

Igualmente, por extensión podemos definir también el **tipo de una expresión** como el tipo al que pertenece el valor de la expresión:

Tipo de una expresión:

Es **el tipo del valor** resultante de **evaluar** dicha expresión.

Los tipos de un lenguaje de programación tienen un nombre (un *identificador*) que los representa.

Ejemplos en Python:

- El tipo `int` define el conjunto de los **números enteros**, sobre los que se pueden realizar, entre otras, las operaciones aritméticas.

  Se corresponde *más o menos* con el símbolo matemático $\mathbb{Z}$, que ya hemos usado antes y que representa el conjunto de los números enteros en Matemáticas.

- El tipo `float` define el conjunto de los **números reales**, sobre los que se pueden realizar también operaciones aritméticas.

  Se corresponde *más o menos* con el símbolo matemático $\mathbb{R}$, que representa el conjunto de los números reales en Matemáticas.

- El tipo `str` define el conjunto de las **cadenas**, sobre las que se pueden realizar otras operaciones (*concatenación*, *repetición*, etc.).

¿Por qué decimos «*más o menos*»?

2.3. type

La función `type` devuelve el tipo de un valor:

```
>>> type(3)
<class 'int'>
>>> type(3.0)
<class 'float'>
>>> type('hola')
<class 'str'>
```

Es muy útil para saber el tipo de una expresión compleja:

```
>>> type(3 + 4.5 ** 2)
<class 'float'>
```

2.4. Sistema de tipos

El **sistema de tipos** de un lenguaje es el conjunto de reglas que asigna un tipo a cada elemento del programa.

Exceptuando a los lenguajes **no tipados** (Ensamblador, código máquina, Forth...) todos los lenguajes tienen su propio sistema de tipos, con sus características.

El sistema de tipos de un lenguaje depende también del paradigma de programación que soporte el lenguaje. Por ejemplo, en los lenguajes **orientados a objetos**, el sistema de tipos se construye a partir de los conceptos propios de la orientación a objetos (*clases*, *interfaces*...).

2.4.1. Errores de tipos

Cuando se intenta realizar una operación sobre un dato cuyo tipo no admite esa operación, se produce un **error de tipos**.

Ese error puede ocurrir cuando:

- Los operandos de un operador no pertenecen al tipo que el operador necesita (ese operador no está definido sobre datos de ese tipo).
- Los argumentos de una función o método no son del tipo esperado.

Por ejemplo:

```
4 + "hola"
```

es incorrecto porque el operador `+` no está definido sobre un entero y una cadena (no se pueden sumar un número y una cadena).

En caso de que exista un error de tipos, lo que ocurre dependerá de si estamos usando un lenguaje interpretado o compilado:

- Si el lenguaje es **interpretado** (Python):

El error se localizará **durante la ejecución** del programa y el intérprete mostrará un mensaje de error advirtiendo del mismo en el momento justo en que la ejecución alcance la línea de código errónea, para acto seguido finalizar la ejecución del programa.

- Si el lenguaje es **compilado** (Java):

  Es muy probable que el comprobador de tipos del compilador detecte el error de tipos **durante la compilación** del programa, es decir, antes incluso de ejecutarlo. En tal caso, se abortará la compilación para impedir la generación de código objeto erróneo.

2.4.2. Tipado fuerte vs. débil

Un lenguaje de programación es **fuertemente tipado** (o de **tipado fuerte**) si no se permiten violaciones de los tipos de datos.

Es decir, un valor de un tipo concreto no se puede usar como si fuera de otro tipo distinto a menos que se haga una *conversión explícita*.

Un lenguaje es **débilmente tipado** (o de **tipado débil**) si no es de tipado fuerte.

En los lenguajes de tipado débil se pueden hacer operaciones entre datos cuyo tipos no son los que espera la operación, gracias al mecanismo de *conversión implícita*.

Existen dos mecanismos de conversión de tipos:

- **Conversión implícita o *coerción***: cuando el intérprete convierte un valor de un tipo a otro sin que el programador lo haya solicitado expresamente.
- **Conversión explícita o *casting***: cuando el programador solicita expresamente la conversión de un valor de un tipo a otro usando alguna construcción u operación del lenguaje.

Los lenguajes de tipado fuerte no realizan conversiones implícitas de tipos salvo excepciones muy concretas (por ejemplo, conversiones entre enteros y reales en expresiones aritméticas).

Los lenguajes de tipado débil se caracterizan, precisamente, por realizar conversiones implícitas cuando, en una expresión, el tipo de un valor no se corresponde con el tipo necesario.

Ejemplo:

- Python es un lenguaje **fuertemente tipado**, por lo que no podemos hacer lo siguiente (da un error de tipos):

  ```
  2 + "3"
  ```

- En cambio, PHP es un lenguaje **débilmente tipado** y la expresión anterior en PHP es perfectamente válida (y vale **cinco**).

  El motivo es que el sistema de tipos de PHP convierte *implícitamente* la cadena `"3"` en el entero `3` cuando se usa en una operación de suma (`+`).

Es importante entender que **la conversión de tipos no modifica el dato original**, sino que devuelve un nuevo dato a partir del dato original pero con el tipo cambiado.

2.5. Tipos de datos básicos

Los tipos de datos básicos que empezaremos a estudiar en Python son:

- Tipos básicos
  - Números
    - Enteros
    - Reales
  - Cadenas
  - Funciones
  - Lógicos (o *booleanos*)

2.5.1. Números

Hay dos tipos numéricos básicos en Python: los enteros y los reales.

- Los **enteros** se representan con el tipo `int`.

  Sólo contienen parte entera, y sus literales se escriben con dígitos sin punto decimal (ej: `13`).

- Los **reales** se representan con el tipo `float`.

  Contienen parte entera y parte fraccionaria, y sus literales se escriben con dígitos y con punto decimal separando ambas partes (ej: `4.87`). Los números en notación exponencial (`2e3`) también son reales ($2e3 = 2.0 \times 10^3$).

Las **operaciones** que se pueden realizar con los números son los que cabría esperar (aritméticas, trigonométricas, matemáticas en general).

Los enteros y los reales generalmente se pueden combinar en una misma expresión aritmética y suele resultar en un valor real, ya que se considera que los reales *contienen* a los enteros.

- Ejemplo: `4 + 3.5` devuelve `7.5`.

Por ello, y aunque el lenguaje sea de tipado fuerte, se permite la conversión implícita entre datos de tipo `int` y `float` dentro de una misma expresión para realizar las operaciones correspondientes.

En el ejemplo anterior, el valor entero `4` se convierte implícitamente en el real `4.0` debido a que el otro operando de la suma es un valor real (`3.5`). Finalmente, se obtiene un valor real (`7.5`).

2.5.2. Cadenas

Las **cadenas** son secuencias de cero o más caracteres codificados en Unicode.

En Python se representan con el tipo `str`.

- No existe el tipo *carácter* en Python. Un carácter en Python es simplemente una cadena que contiene un solo carácter.

Un literal de tipo cadena se escribe encerrando sus caracteres entre comillas simples (`'`) o dobles (`"`).

- No hay ninguna diferencia entre usar unas comillas u otras, pero si una cadena comienza con comillas simples, debe acabar también con comillas simples (y viceversa).

Ejemplos:

`"hola"`

`'Manolo'`

`"27"`

También se pueden escribir literales de tipo cadena encerrándolos entre triples comillas (`'''` o `"""`).

Estos literales se usan para escribir cadenas formadas por varias líneas. La sintaxis de las triples comillas respeta los saltos de línea. Por ejemplo:

```
>>> """Bienvenido
... a
... Python"""
'Bienvenido\na\nPython'  # el carácter \n representa un salto de línea
```

No es lo mismo `27` que `"27"`.

- `27` es un número entero (un literal de tipo `int`).
- `"27"` es una cadena (un literal de tipo `str`).

Una **cadena vacía** es aquella que no contiene ningún carácter. Se representa con los literales `''`, `""`, `''''''` o `""""""`.

Si necesitamos meter el carácter de la comilla simple (`'`) o doble (`"`) en un literal de tipo cadena, tenemos dos opciones:

- Delimitar la cadena con el otro tipo de comillas. Por ejemplo:
  - `'Pepe dijo: "Yo no voy.", así que no fuimos.'`
  - `"Bienvenido, Señor O'Halloran."`
- «*Escapar*» la comilla, poniéndole delante una barra inclinada hacia la izquierda (`\`):
  - `"Pepe dijo: \"Yo no voy.\", así que no fuimos."`
  - `'Bienvenido, Señor O\'Halloran.'`

2.5.3. Tuplas

Las **tuplas** (datos de tipo `tuple`) son una generalización de las cadenas.

Una tupla es una **secuencia de elementos** que no tienen por qué ser caracteres, sino que cada uno de ellos pueden ser **de cualquier tipo** (números, cadenas, booleanos, ..., incluso otras tuplas).

Los literales de tipo tupla se representan enumerando sus elementos separados por comas y encerrados entre paréntesis.

Por ejemplo:

```
tupla = (27, 'hola', True, 73.4, ('a', 'b', 'c'), 99)
```

Si sólo tiene un elemento, hay que poner una coma detrás:

```
tupla = (35,)
```

Al igual que ocurre con las cadenas, tenemos las operaciones `t[0]`, `t[1:]` y `+` (**concatenación**).

Con la concatenación se pueden crear nuevas tuplas a partir de otras tuplas.

Por ejemplo:

```
>>> (1, 2, 3) + (4, 5, 6)
(1, 2, 3, 4, 5, 6)
```

Eso significa que, si `t` es una tupla no vacía, se cumple que `t == (t[0],) + t[1:]`.

Esta propiedad es similar (aunque no exactamente igual) a la que se cumple en las cadenas no vacías.

2.5.4. Funciones

En programación funcional, **las funciones también son datos**:

```
>>> type(max)
<class 'builtin_function_or_method'>
```

La **única operación** que se puede realizar sobre una función es **llamarla**, que sintácticamente se representa poniendo paréntesis `( )` justo a continuación de la función.

Dentro de los paréntesis se ponen los *argumentos* que se aplican a la función en esa llamada (si es que los necesita), separados por comas.

Por tanto, `max` es la función en sí (un **valor** de tipo *función*) , y `max(3, 4)` es una llamada a la función `max` con los argumentos `3` y `4`.

```
>>> max                  # la función max
<built-in function max>
>>> max(3, 4)            # la llamada a max con los argumentos 3 y 4
4
```

Recordemos que **las funciones no tienen expresión canónica**, por lo que el intérprete no intentará nunca visualizar un valor de tipo función.

2.6. Conversión de tipos

Hemos visto que en Python las conversiones de tipos deben ser **explícitas**, es decir, que debemos indicar en todo momento qué dato queremos convertir a qué tipo.

Para ello existen funciones cuyo nombre coincide con el tipo al que queremos convertir el dato: `str`, `int` y `float`, entre otras.

```
>>> 4 + '24'
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> 4 + int('24')
28
```

Convertir un dato a cadena suele funcionar siempre, pero convertir una cadena a otro tipo de dato puede fallar dependiendo del contenido de la cadena:

```
>>> int('hola')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'hola'
```

Recordando lo que dijimos anteriormente, la conversión de tipos no modifica el dato original, sino que devuelve un nuevo dato a partir del dato original pero con el tipo cambiado.

Las funciones de conversión de tipos hacen precisamente eso: devuelven un nuevo dato con un determinado tipo a partir del dato original que reciben como argumento.

Por tanto, la expresión `int('24')` devuelve el entero `24` pero no cambia en modo alguno la cadena `'24'` que ha recibido como argumento.

3. Operaciones predefinidas

3.1. Operadores predefinidos

3.1.1. 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 `float`
`%`	Módulo	`4 % 3` `8 % 3`	`1` `2`	Resto de la división
`**`	Exponente	`3 ** 4`	`81`	Devuelve 3^4
`//`	División entera hacia abajo	`4 // 3` `-4 // 3`	`1` `-2`	 ??

3.1.2. Operadores de cadenas

Operador	Descripción	Ejemplo	Resultado
`+`	Concatenación	`'ab' + 'cd'` `'ab' 'cd'`	`'abcd'`
`*`	Repetición	`'ab' * 3` `3 * 'ab'`	`'ababab'` `'ababab'`
`[0]`	Primer carácter	`'hola'[0]`	`'h'`
`[1:]`	Resto de cadena	`'hola'[1:]`	`'ola'`

3.2. Funciones predefinidas

Función	Descripción	Ejemplo	Resultado
`abs(`*n*`)`	Valor absoluto	`abs(-23)`	`23`
`len(`*cad*`)`	Longitud de la cadena	`len('hola')`	`4`
`max(`n_1`(, `n_2`)`$^+$`)`	Valor máximo	`max(2, 5, 3)`	`5`
`min(`n_1`(, `n_2`)`$^+$`)`	Valor mínimo	`min(2, 5, 3)`	`2`
`round(`*n*`[, `*p*`])`	Redondeo	`round(23.493)` `round(23.493, 1)`	`23` `23.5`
`type(`*v*`)`	Tipo del valor	`type(23.5)`	`<class 'float'>`

3.2.1. 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**.

3.2.1.1. 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 `operator`
`+`	Suma	`add`
`-`	Resta	`sub`
`-`	Cambio de signo	`neg`
`*`	Multiplicación	`mul`
`/`	División	`truediv`
`%`	Módulo	`mod`
`**`	Exponente	`pow`
`//`	División entera hacia abajo	`floordiv`

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
```

3.3. 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

4. Álgebra de Boole

4.1. El tipo de dato *booleano*

Un dato **lógico** o *booleano* es aquel que puede tomar uno de dos posibles valores, que se denotan normalmente como **verdadero** y **falso**.

Esos dos valores tratan de representar los dos valores de verdad de la **lógica** y el **álgebra *booleana***.

Su nombre proviene de **George Boole**, matemático que definió por primera vez un sistema algebraico para la lógica a mediados del S. XIX.

En Python, el tipo de dato lógico se representa como `bool` y sus posibles valores son `False` y `True`.

Esos dos valores son *formas especiales* para los enteros `0` y `1`, respectivamente.

4.2. Operadores relacionales

Los **operadores relacionales** son operadores que toman dos operandos (que usualmente deben ser del mismo tipo) y devuelven un valor *booleano*.

Los más conocidos son los **operadores de comparación**, que sirven para comprobar si un dato es menor, mayor o igual que otro, según un orden preestablecido.

Los operadores de comparación que existen en Python son:

`<` `>` `<=` `>=` `==` `!=`

Por ejemplo:

```
>>> 4 == 3
False
>>> 5 == 5
True
>>> 3 < 9
True
```

```
>>> 9 != 7
True
>>> False == True
False
>>> 8 <= 8
True
```

4.3. Operadores lógicos

Las **operaciones lógicas** se representan mediante **operadores lógicos**, que son aquellos que toman uno o dos operandos *booleanos* y devuelven un valor *booleano*.

Las operaciones básicas del álgebra de Boole se llaman **suma**, **producto** y **complemento**.

En **lógica proposicional** (un tipo de lógica matemática que tiene estructura de álgebra de Boole), se llaman:

Operación	Operador
Disyunción	$\vee$
Conjunción	$\wedge$
Negación	$\neg$

En Python se representan como `or`, `and` y `not`, respectivamente.

4.3.1. Tablas de verdad

Una **tabla de verdad** es una tabla que muestra el valor lógico de una expresión compuesta, para cada uno de los valores lógicos que puedan tomar sus componentes.

Se usan para definir el significado de las operaciones lógicas y también para verificar que se cumplen determinadas propiedades.

Las tablas de verdad de los operadores lógicos son:

A	B	$A \vee B$
F	F	F
F	V	V
V	F	V
V	V	V

A	B	$A \wedge B$
F	F	F
F	V	F
V	F	F
V	V	V

A	$\neg A$
F	V
V	F

Que traducido a Python sería:

A	B	A **or** B
False	False	False
False	True	True
True	False	True
True	True	True

A	B	A **and** B
False	False	False
False	True	False
True	False	False
True	True	True

A	**not** A
False	True
True	False

4.4. Axiomas

1. Ley asociativa: $\begin{cases} \forall a, b, c \in \mathbb{B} : (a \vee b) \vee c = a \vee (b \vee c) \\ \forall a, b, c \in \mathbb{B} : (a \wedge b) \wedge c = a \wedge (b \wedge c) \end{cases}$
2. Ley conmutativa: $\begin{cases} \forall a, b \in \mathbb{B} : a \vee b = b \vee a \\ \forall a, b \in \mathbb{B} : a \wedge b = b \wedge a \end{cases}$
3. Ley distributiva: $\begin{cases} \forall a, b, c \in \mathbb{B} : a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c) \\ \forall a, b, c \in \mathbb{B} : a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c) \end{cases}$
4. Elemento neutro: $\begin{cases} \forall a \in \mathbb{B} : a \vee F = a \\ \forall a \in \mathbb{B} : a \wedge V = a \end{cases}$
5. Elemento complementario: $\begin{cases} \forall a \in \mathbb{B}; \exists \neg a \in \mathbb{B} : a \vee \neg a = V \\ \forall a \in \mathbb{B}; \exists \neg a \in \mathbb{B} : a \wedge \neg a = F \end{cases}$

Si $(\mathbb{B}, \neg, \vee, \wedge)$ cumple lo anterior, entonces es un álgebra de Boole.

4.4.0.1. Traducción a Python

1. Ley asociativa:

```
(a or b) or c == a or (b or c)
(a and b) and c == a and (b and c)
```

2. Ley conmutativa:

```
a or b == b or a
a and b == b and a
```

3. Ley distributiva:

```
a or (b and c) == (a or b) and (a or c)
a and (b or c) == (a and b) or (a and c)
```

4. Elemento neutro:

```
a or False == a
a and True == a
```

5. Elemento complementario:

```
a or (not a) == True
a and (not a) == False
```

4.5. Teoremas fundamentales

6. Ley de idempotencia: $\begin{cases} \forall a \in \mathbb{B} : a \vee a = a \\ \forall a \in \mathbb{B} : a \wedge a = a \end{cases}$
7. Ley del elemento absorbente: $\begin{cases} \forall a \in \mathbb{B} : a \vee V = V \\ \forall a \in \mathbb{B} : a \wedge F = F \end{cases}$
8. Ley de identidad: $\begin{cases} \forall a \in \mathbb{B} : a \vee F = a \\ \forall a \in \mathbb{B} : a \wedge V = a \end{cases}$
9. Ley de absorción: $\begin{cases} \forall a \in \mathbb{B} : a \vee (a \wedge b) = a \\ \forall a \in \mathbb{B} : a \wedge (a \vee b) = a \end{cases}$
10. Ley de involución: $\forall a \in \mathbb{B} : \neg\neg a = a$
11. Ley del complemento: $\begin{cases} \neg V = F \\ \neg F = V \end{cases}$
12. Leyes de De Morgan: $\begin{cases} \forall a, b \in \mathbb{B} : \neg(a \vee b) = \neg a \wedge \neg b \\ \forall a, b \in \mathbb{B} : \neg(a \wedge b) = \neg a \vee \neg b \end{cases}$

4.5.0.1. Traducción a Python

6. Ley de idempotencia:

```
a or a == a
a and a == a
```

7. Ley del elemento absorbente:

```
a or True == True
a and False == False
```

8. Ley de identidad:

```
a or False == a
a and True == a
```

9. Ley de absorción:

```
a or (a and b) == a
a and (a or b) == a
```

10. Ley de involución:

```
not (not a) == a
```

11. Ley del complemento:

```
not True == False
not False == True
```

12. Leyes de De Morgan:

```
not (a or b) == (not a) and (not b)
not (a and b) == (not a) or (not b)
```

4.5.0.2. Lógica binaria

Otra forma de representar los operadores y los valores del álgebra de Boole es mediante la notación de la **lógica binaria**.

Según la notación de la lógica binaria, los diferentes valores y operaciones del álgebra de Boole se representan de la siguiente forma:

Valor u operación	Notación
Valor verdadero	1
Valor falso	0

Valor u operación	Notación
Producto de A y B	$A \cdot B$ AB
Suma de A y B	$A + B$
Complemento de A	$\overline{A}$

Por ejemplo, las leyes de DeMorgan, que en Python se escriben así:

```
not (a or b) == (not a) and (not b)
not (a and b) == (not a) or (not b)
```

se escribirían así según la notación de la lógica binaria:

$$\overline{A + B} = \overline{A} \cdot \overline{B}$$
$$\overline{AB} = \overline{A} + \overline{B}$$

Esta notación se emplea principalmente en el diseño de circuitos digitales.

4.6. Equivalencia lógica

Decimos que dos expresiones lógicas P y Q son **equivalentes** (y se representa como $P \equiv Q$, aunque nosotros lo representaremos simplemente como $P = Q$) si tienen las mismas tablas de verdad, es decir, si se cumple que P vale V cuando Q también, y viceversa.

Para demostrar que se cumple una equivalencia en el álgebra de Boole, se pueden usar dos técnicas:

- Demostrar que la propiedad es un *teorema* que se puede deducir de los axiomas y de otros teoremas ya demostrados.
- Usar las *tablas de verdad* para comprobar si los valores de verdad de P y Q coinciden exactamente.

Por ejemplo, supongamos que queremos demostrar la siguiente propiedad:

$$A(A + B) = A$$

Podemos demostrarlo siguiendo la siguiente secuencia de razonamientos:

$$
\begin{aligned}
& A(A+B) \\
= & \qquad \{\text{Ley distributiva}\} \\
& AA+AB \\
= & \qquad \{\text{Ley de idempotencia}\} \\
& A+AB \\
= & \qquad \{\text{Ley de absorción}\} \\
& A
\end{aligned}
$$

También podemos obtener sus tablas de verdad y comprobar que son idénticas:

A	B	$A+B$	$A(A+B)$
0	0	0	0
0	1	1	0
1	0	1	1
1	0	1	1

Como podemos observar, las columnas de A y de $A(A+B)$ son idénticas, lo que significa que toman siempre los mismos valores de verdad y, por tanto, ambas expresiones son equivalentes.

4.7. El operador ternario

Las expresiones lógicas (o *booleanas*) se pueden usar para comprobar si se cumple una determinada **condición**.

Las condiciones en un lenguaje de programación se representan mediante expresiones lógicas cuyo valor (*verdadero* o *falso*) indica si la condición se cumple o no se cumple.

Con el **operador ternario** podemos hacer que el resultado de una expresión varíe entre dos posibles opciones dependiendo de si se cumple o no una condición.

El operador ternario se llama así porque es el único operador en Python que actúa sobre tres operandos.

El uso del operador ternario permite crear lo que se denomina una **expresión condicional**.

Su sintaxis es:

⟨*expr_condicional*⟩ ::= ⟨*valor_si_cierto*⟩ **`if`** ⟨*condición*⟩ **`else`** ⟨*valor_si_falso*⟩

donde:

- ⟨*condición*⟩ debe ser una expresión lógica

- ⟨*valor_si_cierto*⟩ y ⟨*valor_si_falso*⟩ pueden ser expresiones de cualquier tipo

El valor de la expresión completa será ⟨*valor_si_cierto*⟩ si la ⟨*condición*⟩ es cierta; en caso contrario, su valor será ⟨*valor_si_falso*⟩.

Ejemplo:

```
25 if 3 > 2 else 17
```

evalúa a `25`.

El operador ternario, así como los operadores lógicos **`and`** y **`or`**, se evalúan siguiendo una estrategia según la cual **no siempre se evalúan todos sus operandos**.

La expresión condicional:

```
⟨valor_si_cierto⟩ if ⟨condición⟩ else ⟨valor_si_falso⟩
```

se evalúa de la siguiente forma:

- Primero siempre se evalúa la ⟨*condición*⟩.
- Si es `True`, evalúa ⟨*valor_si_cierto*⟩.
- Si es `False`, evalúa ⟨*valor_si_falso*⟩.

Por tanto, en la expresión condicional nunca se evalúan todos sus operandos, sino sólo los estrictamente necesarios.

Además, no se evalúan de izquierda a derecha, como es lo normal.

La evaluación de los operadores **`and`** y **`or`** sigue un proceso similar:

- Primero se evalúa el operando izquierdo.
- El operando derecho sólo se evalúa si el izquierdo no proporciona la información suficiente para determinar el resultado de la operación.

Esto es así porque:

- `True` **`or`** *x*

  siempre es igual a `True`, valga lo que valga *x*.

- `False` **`and`** *x*

  siempre es igual a `False`, valga lo que valga *x*.

En ambos casos no es necesario evaluar *x*.

Ejercicio

1. ¿Cuál es la asociatividad del operador ternario? Demostrarlo.

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.

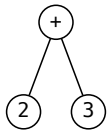
Si una expresión está correctamente escrita según la sintaxis del lenguaje, sólo tendrá un único árbol sintáctico equivalente.

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

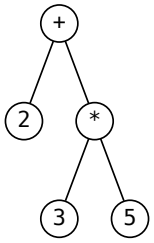
Por ejemplo, tenemos las siguientes expresiones y sus árboles sintácticos equivalentes:

```
2
```

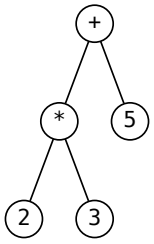

```
2 + 3
```




```
2 + 3 * 5
```

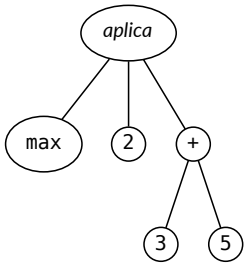


```
2 * 3 + 5
```



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 uilizado.

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.

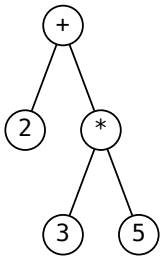
La idea es que, antes de evaluar un nodo, debemos evaluar primero todos sus nodos hijos, en orden, de izquierda a derecha.

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:

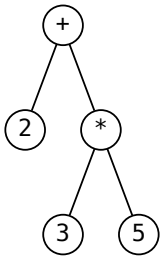


El orden en el que vamos evaluando los nodos sería el siguiente:

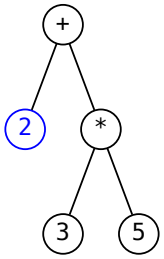
2, 3, 5, *, +

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

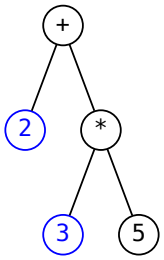
Paso 1: Se empieza visitando la raíz + pero, como tiene hijos, antes de evaluarlo se pasa a visitar su primer hijo (2).



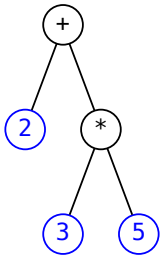
Paso 2: Como estamos en el nodo 2 y éste no tiene hijos, se puede evaluar directamente, ya que es un nodo hoja y, por tanto, representa un valor. La evaluación del nodo no cambia el nodo ni lo sustituye por ningún otro.



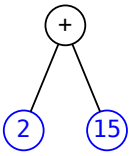
Paso 3: Volvemos al padre del nodo 2, que es el nodo raíz +, el cual todavía no lo podemos evaluar porque aún le queda otro nodo hijo por evaluar (el nodo *), así que bajamos hasta él. Éste, a su vez, tampoco se puede evaluar porque tiene hijos que hay que evaluar antes, el primero de los cuales es el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

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

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

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

$\langle expr \rangle_1$ `+` $\langle expr \rangle_2$

el intérprete llamará a una de las dos operaciones, dependiendo de los tipos de $\langle expr \rangle_1$ y $\langle expr \rangle_2$.

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 signatura, 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.

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.