

# Programación funcional (I)

Ricardo Pérez López

IES Doñana, curso 2024/2025

Generado el 2025/07/08 a las 16:01:00

## Índice

<b>1. Introducción</b>	<b>2</b>
1.1. Concepto . . . . .	2
1.2. Transparencia referencial . . . . .	3
1.2.1. Efectos laterales . . . . .	4
1.3. Modelo de ejecución . . . . .	4
1.3.1. Modelo de sustitución . . . . .	5
<b>2. Tipos de datos</b>	<b>6</b>
2.1. Concepto . . . . .	6
2.2. Tipo de un dato . . . . .	6
2.3. <code>type</code> . . . . .	8
2.4. Sistema de tipos . . . . .	8
2.4.1. Errores de tipos . . . . .	8
2.4.2. Tipado fuerte vs. débil . . . . .	9
2.5. Tipos de datos básicos . . . . .	10
2.5.1. Números . . . . .	10
2.5.2. Cadenas . . . . .	10
2.5.3. Funciones . . . . .	11
2.6. Conversión de tipos . . . . .	12
<b>3. Álgebra de Boole</b>	<b>12</b>
3.1. El tipo de dato <i>booleano</i> . . . . .	12
3.2. Operadores relacionales . . . . .	13
3.3. Operadores lógicos . . . . .	13
3.3.1. Tablas de verdad . . . . .	14
3.4. Axiomas . . . . .	15
3.5. Teoremas fundamentales . . . . .	16
3.6. Equivalencia lógica . . . . .	18
3.7. El operador ternario . . . . .	19
<b>4. Definiciones</b>	<b>20</b>
4.1. Introducción . . . . .	20

4.2. Identificadores y ligaduras ( <i>binding</i> ) . . . . .	20
4.2.1. Ligaduras irrompibles . . . . .	21
4.2.2. Inmutabilidad . . . . .	23
4.2.3. Reglas léxicas . . . . .	23
4.2.4. Tipo de un identificador . . . . .	24
4.3. Espacios de nombres . . . . .	24
4.4. Marcos ( <i>frames</i> ) . . . . .	25
4.5. Evaluación de expresiones con identificadores . . . . .	28
4.5.1. Resolución de identificadores . . . . .	28
4.6. <i>Scripts</i> . . . . .	29
4.7. Instalación de Visual Studio Code . . . . .	31
4.7.1. Configuración básica de Visual Studio Code . . . . .	31
5. Resumen . . . . .	32
5.1. Resumen . . . . .	32
6. Documentación interna . . . . .	32
6.1. Identificadores significativos . . . . .	32
6.2. Comentarios . . . . .	33

## 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.

Definiciones 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 **cadena**s, 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 advirtiéndolo 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 tipo 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"
```

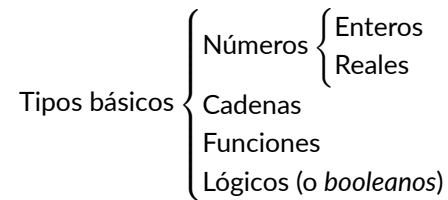
- En cambio, PHP es un lenguaje **débilmente tipado** y la expresión anterior en PHP es perfectamente válida (y vale **cinco**).

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:



### 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. 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. Álgebra de Boole

### 3.1. 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.

### 3.2. 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
```

### 3.3. 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.

### 3.3.1. 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	$A \text{ or } B$
False	False	False
False	True	True
True	False	True
True	True	True

A	B	$A \text{ and } B$
False	False	False
False	True	False
True	False	False
True	True	True

A	not A
False	True
True	False

### 3.4. 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.

#### 3.4.0.1. 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
```

### 3.5. 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:  $\begin{cases} \forall a, b \in \mathbb{B} : \neg(a \vee b) = \neg a \wedge \neg b \\ \forall a, b \in \mathbb{B} : \neg(a \wedge b) = \neg a \vee \neg b \end{cases}$

#### 3.5.0.1. 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)
```

### 3.5.0.2. 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
Producto de A y B	$A \cdot B$ $AB$
Suma de A y B	$A + B$
Complemento de A	$\bar{A}$

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:

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

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

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

### 3.6. 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:

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

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	1	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.

### 3.7. 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.

## 4. Definiciones

### 4.1. Introducción

Introduciremos ahora en nuestro lenguaje una nueva instrucción (técnicamente es una **sentencia**) con la que vamos a poder hacer **definiciones**.

A esa sentencia la llamaremos **definición**, y expresa el hecho de que **un nombre representa un valor**.

Las definiciones tienen la siguiente sintaxis:

```
<definición> ::= identificador = <expresión>
```

Por ejemplo:

```
x = 25
```

A partir de ese momento, el identificador `x` representa el valor `25`.

Y si `x` vale `25`, la expresión `2 + x * 3` vale `77`.

### 4.2. Identificadores y ligaduras (*binding*)

Los **identificadores** son los nombres o símbolos que representan a los elementos del lenguaje.

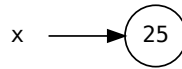
Cuando hacemos una definición, lo que hacemos es asociar un identificador con un valor.

Esa asociación se denomina **ligadura** (o *binding*).

Por esa razón, también se dice que **una definición crea una ligadura**.

También decimos que el identificador está **ligado** (*bound*).

Lo representaremos gráficamente así:

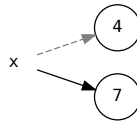


En Python (a diferencia de lo que ocurre en un **lenguaje funcional puro**) las ligaduras empiezan a existir en el momento en que se ejecuta su definición, no antes.

#### 4.2.1. Ligaduras irrompibles

En un **lenguaje funcional puro**, un identificador ya ligado no se puede ligar a otro valor. Por ejemplo, lo siguiente daría un error en un lenguaje funcional puro:

```
x = 4 # ligamos el identificador x al valor 4
x = 7 # intentamos ligar x al valor 7, pero ya está ligado al valor 4
```



En consecuencia, **las ligaduras** entre nombres y valores **no se pueden romper**, de forma que un nombre, una vez ligado a un valor, **no se puede volver a ligar a otro valor distinto** durante la ejecución del programa (efecto que se conoce como *rebinding*).

En la práctica, eso significa que el nombre representa un **dato constante**.

Que las ligaduras sean irrompibles son un requisito necesario para alcanzar la **transparencia referencial**.

Si hago:

```
x = "Hola"
```

en un lenguaje funcional puro, luego no puedo hacer:

```
x = "Hala"
```

porque eso sería hacer un *rebinding* y provocaría que, de nuevo, la expresión `x` tuviera distintos valores según el momento, lo que va en contra de la transparencia referencial.

Python no es un lenguaje funcional puro, por lo que sí se permite volver a ligar el mismo identificador a otro valor distinto.

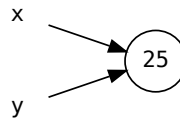
- Al hacer esto, se estaría perdiendo el valor anterior.
- Así que, por ahora, el *rebinding* está prohibido para nosotros (**no lo hagamos**).

Lo que sí se puede hacer es:

```
x = 25
y = x
```

En este caso estamos ligando a `y` el mismo valor que tiene `x`, algo perfectamente válido en un lenguaje funcional.

Lo que hace el intérprete en este caso no es crear dos valores `25` en memoria (sería un gasto inútil de memoria), sino que `x` e `y` *comparten* el único valor `25` que existe:

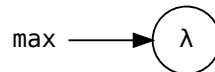


Por tanto:

```
>>> x = 25
>>> y = x
>>> y
25
```

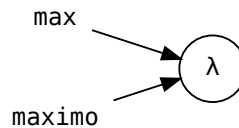
El **nombre** de una **función** es un identificador que está ligado a la función correspondiente (que en programación funcional es un valor como cualquier otro).

Por ejemplo, `max` es un identificador ligado a la función que devuelve el máximo de dos números (que representaremos aquí como  $\lambda$ ):



Así que ese valor se puede ligar a otro identificador y, de esta forma, ambos identificadores compartirían el mismo valor (y, por tanto, representarían a la misma función). Por ejemplo:

```
>>> maximo = max
>>> maximo(3, 4)
4
```



### 4.2.2. Inmutabilidad

Para alcanzar la transparencia referencial, también es necesario que **los objetos** de datos que manipula el programa (es decir, los valores) **no tengan un estado interno** que pueda cambiar durante la ejecución del programa.

Eso se consigue haciendo que los valores sean **inmutables**.

Por ejemplo, en Python las cadenas son inmutables porque, una vez creadas, no se pueden cambiar los caracteres que la forman.

Si hago `x = "Hola"`, luego no puedo cambiar el interior de esa cadena (por ejemplo, cambiando la 'o' por una 'a'), porque entonces la expresión `x` tendría distintos valores dependiendo del momento en el que se evalúe, lo que va en contra de la transparencia referencial.

Eso significa que en programación funcional tampoco estaría permitido hacer cosas como esta:

```
import math
math.constante = 405
```

ya que entonces estaríamos **creando una nueva ligadura dentro del objeto** que representa al módulo `math`, lo que en la práctica supone que estamos **cambiando el estado interno de ese objeto** y, por tanto, estaría **dejando de ser inmutable**.

Recordemos que **los módulos son objetos** y, como tales, **son valores** como cualquier otro.

En cambio, sí sería correcto hacer algo así:

```
import math
constante = 405
```

ya que ahí no se está cambiando el estado interno de ningún valor.

### 4.2.3. Reglas léxicas

Cuando hacemos una definición debemos tener en cuenta ciertas cuestiones relativas al identificador:

- ¿Cuál es la **longitud máxima** de un identificador?
- ¿Qué **caracteres** se pueden usar?
- ¿Se distinguen **mayúsculas** de **minúsculas**?
- ¿Coincide con una palabra clave o reservada?
  - \* **Palabra clave**: palabra que forma parte de la sintaxis del lenguaje.
  - \* **Palabra reservada**: palabra que no puede emplearse como identificador.

En Python, los identificadores pueden ser combinaciones de letras minúsculas y mayúsculas (y distingue entre ellas), dígitos y subrayados (`_`), no pueden empezar por un dígito, no pueden coincidir con una palabra reservada y pueden tener cualquier longitud.

#### 4.2.4. Tipo de un identificador

Cuando un identificador está ligado a un valor, a efectos prácticos el identificador actúa como si fuera el valor.

Como cada valor tiene un tipo de dato asociado, también podemos hablar del **tipo de un identificador**.

El **tipo de un identificador** es el tipo del dato con el que está ligado.

Si un identificador no está ligado, no tiene sentido preguntarse qué tipo tiene.

### 4.3. Espacios de nombres

Ciertas estructuras o construcciones sintácticas del programa definen lo que se denomina **espacios de nombres**.

Un **espacio de nombres** (del inglés, *namespace*) es una correspondencia entre nombres y valores; es decir, es un **conjunto de ligaduras**.

Su función es, por tanto, **almacenar ligaduras y permitir varias ligaduras con el mismo nombre** en distintas partes del programa.

En un espacio de nombres, **un identificador sólo puede tener como máximo una ligadura**. En cambio, el mismo identificador puede estar ligado a diferentes valores en diferentes espacios de nombres.

Por eso decimos que una ligadura es **local** al espacio de nombres donde se almacena, o que tiene un **almacenamiento local** a ese espacio de nombres.

Los espacios de nombres pueden estar **anidados**, es decir, que puede haber espacios de nombres dentro de otros espacios de nombres, ya que también pueden estarlo las estructuras sintácticas que los definen.

Algunos espacios de nombres provienen de estructuras estáticas (o sea, que vienen definidas en tiempo de compilación) y otras provienen de estructuras dinámicas (definidas en tiempo de ejecución).

Por tanto, podemos tener **espacios de nombres estáticos** y **espacios de nombres dinámicos**.

En Java, las clases y los paquetes son *espacios de nombres estáticos*, que el compilador gestiona a través de su *tabla de símbolos*.

Por otra parte, durante la ejecución de un programa se pueden ir creando ciertas estructuras en memoria que representan *espacios de nombres dinámicos*.

Los ejemplos más comunes de estas estructuras son:

- Los **marcos** que se crean al ejecutar *scripts* de Python y al invocar funciones (o métodos) definidas por el programador en Python y Java.
- Los **objetos** de Python.
- Las **clases** de Python (que también son objetos).
- Los **módulos** de Python (que también son objetos).



Los marcos y los objetos son los únicos espacios de nombres que existen en Python.

Por tanto, todos los espacios de nombres en Python son dinámicos.

Un espacio de nombres muy importante en Python es el que incluye las definiciones predefinidas del lenguaje (funciones `max` o `sum`, tipos como `str` o `int`, etc.)

Ese espacio de nombres se denomina `__builtins__` y viene implementado en forma de *módulo* que se importa automáticamente en cada sesión interactiva o cada vez que se arranca un programa Python.

Pero sabemos que también podemos usar directamente las definiciones que contiene, por lo que el efecto es como si Python ejecutara las siguientes dos sentencias nada más entrar en el intérprete:

```
import __builtins__  
from __builtins__ import *
```

Esto no es exactamente así en realidad, pero por ahora haremos como si así fuera, por simplicidad.

#### 4.4. Marcos (*frames*)

Un **marco** (del inglés, *frame*) es una estructura que se crea en memoria para **representar la ejecución o activación de un script de Python o una función o método definido por el programador** en Python o Java.

Los marcos son **espacios de nombres** y, entre otras cosas, almacenan las ligaduras que se definen dentro de ese *script*, función o método.

Los marcos son conceptos *dinámicos*:

- Se crean y se destruyen a medida que la ejecución del programa pasa por ciertas partes del mismo.
- Van almacenando nuevas ligaduras conforme se van ejecutando nuevas instrucciones que crean las ligaduras.

Por ahora, el único marco que existe en nuestros programas es el llamado **marco global**, también llamado **espacio de nombres global**.

El marco global se crea en el momento en que **se empieza a ejecutar el programa** y existe durante toda la ejecución del mismo (sólo se destruye al finalizar la ejecución del programa).

Si se está trabajando en una sesión con el intérprete interactivo, el marco global **se crea justo al empezar la sesión** y existe durante toda la sesión (sólo se destruye al salir de la misma).

Por tanto, las definiciones que se ejecutan directamente en una sesión interactiva con el intérprete, crean ligaduras que se almacenan en el marco global.

Es importante aclarar que, en un programa escrito en un lenguaje funcional puro, no importa el orden en el que aparezcan las definiciones, por lo que se podría usar un nombre antes de que aparezca su definición en el código fuente del programa.

Por ejemplo, en un lenguaje funcional puro como Haskell podríamos escribir lo siguiente:

```
a = 2 + b
b = 5
```

y funcionaría perfectamente, aunque en la primera línea de código estemos usando un nombre que se define después en el código fuente, en la segunda línea.

Eso es debido a que los lenguajes funcionales puros no dependen del orden de ejecución de las instrucciones.

En cambio, Python no es un lenguaje funcional puro, así que el orden de ejecución de las instrucciones sí que importa.

Por ejemplo, si iniciamos una sesión con el intérprete interactivo y justo a continuación tecleamos lo siguiente:

```
1 >>> x
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4 NameError: name 'x' is not defined
5 >>> x = 25
6 >>> x
7 25
```

- Aquí estamos trabajando con el *marco global* (el único marco que existe hasta ahora para nosotros).
- En la línea 1, el identificador `x` aún no está ligado, por lo que su uso genera un error (el marco global no contiene hasta ahora ninguna ligadura para `x`).
- En la línea 6, en cambio, el identificador puede usarse sin error ya que ha sido ligado previamente en la línea 5 (el marco global ahora contiene una ligadura para `x` con el valor 25).

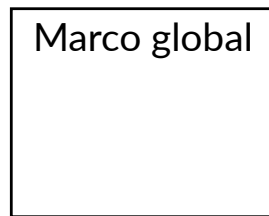
Los marcos son espacios de nombres dinámicos, que se van creando y destruyendo durante la ejecución del programa.

Igualmente, las ligaduras que contiene también se van creando y destruyendo a medida que se van ejecutando las instrucciones que forman el programa.

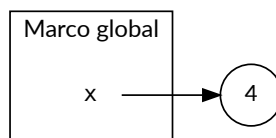
Si tenemos la siguiente sesión interactiva:

```
1 >>> 2 + 3
2 5
3 >>> x = 4
4 >>> y = 3
5 >>> z = y
```

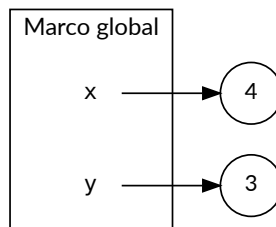
Según hasta donde hayamos ejecutado, el marco global contendrá lo siguiente:



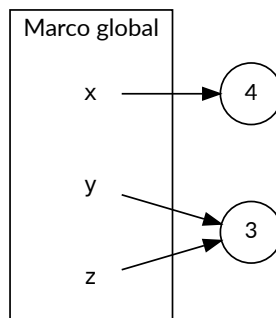
Marco global en las líneas 1-2



Marco global en la línea 3



Marco global en la línea 4



Marco global en la línea 5

Hemos visto que una ligadura es una asociación entre un identificador y un valor.

También hemos visto que los espacios de nombres almacenan ligaduras, y que un marco es un espacio de nombres.

Por tanto, **los marcos almacenan ligaduras, pero NO almacenan los valores** a los que están asociados los identificadores de esas ligaduras.

Por eso hemos dibujado a los valores fuera de los marcos en los diagramas anteriores.

Los valores se almacenan en una zona de la memoria del intérprete conocida como el **montículo**.

En cambio, los marcos se almacenan en otra zona de la memoria conocida como la **pila de control**, la cual estudiaremos mejor más adelante.

## 4.5. Evaluación de expresiones con identificadores

Podemos usar un identificador ligado dentro de una expresión, siempre que la expresión resulte ser válida según las reglas del lenguaje.

El identificador representa a su valor ligado y se evalúa a dicho valor en cualquier expresión donde aparezca ese identificador:

```
>>> x = 25
>>> 2 + x * 3
77
```

Intentar usar en una expresión un identificador no ligado provoca un error de tipo **NameError** (*nombre no definido*):

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

### 4.5.1. Resolución de identificadores

Durante la evaluación de una expresión, para cada uno de los diferentes identificadores que aparecen en ella, habrá que comprobar si ese identificador está ligado y a qué valor.

- Si no está ligado, es un error de *nombre no definido*.
- En caso contrario, tendrá que determinar a qué valor está ligado para poder sustituir, en la expresión, cada aparición del identificador por su valor.

Para ello, habrá que buscar una ligadura con ese identificador en uno o varios espacios de nombres.

El proceso de localizar (si es que existe) la ligadura adecuada que liga a un identificador con su valor, se denomina **resolución del identificador**.

En general, al proceso de determinar con qué valores están ligados los identificadores de un programa se le denomina **resolución de identificadores** o **resolución de nombres**.

Los espacios de nombres donde se buscan las ligaduras para ese identificador dependerán del *contexto* en el que se está intentando resolver dicho identificador.

Por ejemplo, en el siguiente código:

```
>>> x = 4
>>> x
4
```

se busca una ligadura para `x` en el marco global.

En cambio, si se intenta acceder a un atributo de un objeto:

```
>>> import math
>>> math.pi
3.141592653589793
```

se busca una ligadura para `pi` en el espacio de nombres asociado al módulo `math`.

La resolución de identificadores es un proceso que usa mecanismos distintos dependiendo de si el lenguaje es interpretado o compilado:

- Si es un **lenguaje interpretado** (como Python): el intérprete usa un concepto llamado **entorno** en tiempo de ejecución para localizar la ligadura.

En tal caso, hablamos de **resolución de nombres dinámica**.

- Si es un **lenguaje compilado** (como Java): el compilador determina, en tiempo de compilación, si una ligadura es accesible haciendo uso del concepto de **ámbito** y, en caso afirmativo, deduce en qué espacio de nombres está la ligadura.

En tal caso, hablamos de **resolución de nombres estática**.

En general, la resolución de identificadores puede ser una tarea complicada ya que puede involucrar muchos conceptos como espacios de nombres, ámbitos, entornos, reglas de visibilidad, sombreado, sobrecargas... muchos de los cuales aún no hemos estudiado.

## 4.6. Scripts

Cuando tenemos muchas definiciones o muy largas, resulta tedioso tener que introducirlas una y otra vez cada vez que abrimos una nueva sesión con el intérprete interactivo.

Lo más cómodo es teclearlas todas una sola vez dentro un archivo que luego cargaremos desde dentro del intérprete.

Ese archivo se llama **script** y, por ahora, contendrá una lista de las definiciones que nos interese usar en nuestras sesiones interactivas con el intérprete.

Al cargar el **script**, se ejecutarán sus instrucciones una tras otra casi de la misma forma que si las estuviéramos tecleando nosotros directamente en nuestra sesión con el intérprete, en el mismo orden.

Llegado el momento, los **scripts** contendrán el código fuente de nuestros programas y los ejecutaremos desde el intérprete por lotes.

Los nombres de archivo de los **scripts** en Python llevan extensión `.py`.

Para cargar un **script** en nuestra sesión tenemos dos opciones:

1. Usar la orden **from** dentro de la sesión actual.

Por ejemplo, para cargar un *script* llamado `definiciones.py`, usaremos:

```
>>> from definiciones import *
```

Observar que en el **from** se pone el nombre del script pero **sin la extensión .py**.

2. Iniciar una nueva sesión con el intérprete interactivo indicándole que cargue el *script* mediante la opción **-i** en la línea de órdenes del sistema operativo:

```
$ python -i definiciones.py
>>>
```

En este caso **sí** se pone el nombre completo del script, **con la extensión .py**.

A partir de ese momento, en el intérprete interactivo podremos usar las definiciones que se hayan cargado desde el *script*.

Por ejemplo, si el *script* `definiciones.py` tiene el siguiente contenido:

```
x = 25
j = 14
```

Al cargar el *script* (usando cualquiera de las dos opciones que hemos visto anteriormente) se ejecutarán sus instrucciones (las dos definiciones) y, en consecuencia, se crearán en el marco global las ligaduras `x → 25` y `j → 14`:

```
>>> x                                     # antes de cargar el script, da error
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
>>> from definiciones import *           # aquí cargamos el script, y ahora
>>> x                                     # la x sí está ligada a un valor
25
```

Una limitación importante que hay que tener en cuenta es que **el script sólo puede usar definiciones que se hayan creado en el mismo script** (exceptuando las definiciones predefinidas del lenguaje).

Por ejemplo, si tenemos el siguiente *script* llamado `prueba.py`:

```
j = w + 1
```

donde se intenta ligar a `j` el valor ligado a `w` más uno, lo siguiente no funcionará:

```
>>> w = 3
>>> from prueba import *
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/ricardo/python/prueba.py", line 1, in <module>
    j = w + 1
NameError: name 'w' is not defined
```

Aunque `w` esté ligado a un valor al cargar el *script*, éste no podrá acceder a esa ligadura y da error de «*nombre no definido*».

Cuando estudiemos la programación modular entenderemos por qué.

## 4.7. Instalación de Visual Studio Code

### Instalación en Ubuntu:

**NO** instalar Visual Studio Code desde el *Centro de software de Ubuntu*.

En cambio, visitar <https://code.visualstudio.com> para descargar e instalar el paquete adecuado a la versión de Ubuntu que se está usando, que suele ser la que ya sugiere la propia página web (normalmente es el paquete `.deb` de 64 bits).

### Instalación en Windows:

Visitar <https://code.visualstudio.com>, descargar e instalar la versión adecuada a la versión de Windows que se está usando.

#### 4.7.1. Configuración básica de Visual Studio Code

1. Abrir la aplicación.
2. Acceder al apartado de *Extensiones* (`Ctrl+Mayús+X`) e instalar las siguientes extensiones:

Python

Python Indent

Pylance

3. Configurar las siguientes opciones:

Python: *Language Server*: Pylance

Python > *Linting: Enabled*: Activado

Python > *Linting: Pylint Enabled*: Activado

Python > *Linting: Pylint Args*:

`--disable=invalid-name,redefined-outer-name,missing-docstring,unnecessary-lambda-assignment`

4. Se recomiendan, además, configurar las siguientes opciones:

Editor: *Render Final Newline*: Desactivado

Editor: *Smooth Scrolling*: Activado

Files: *Insert Final Newline*: Activado

Files: *Trim Final Newlines*: Activado

Files: *Trim Trailing Whitespace*: Activado

## 5. Resumen

### 5.1. Resumen

Resumiendo, los conceptos fundamentales sobre los que se asienta la programación funcional son:

- Casi todas las instrucciones son expresiones, no sentencias.
- Definiciones, las únicas sentencias de un programa funcional.
- Transparencia referencial.
- Ausencia de efectos laterales.
- Funciones puras.
- El valor de una expresión no depende de nada ajeno a la misma, sólo de las subexpresiones que la forman.
- Ligaduras irrompibles.
- Inmutabilidad.
- Las funciones también son valores.
- Ejecutar un programa es evaluar una expresión.
- No importa el orden en el que se ejecuten las instrucciones.
- No importa el orden en el que se evalúen las subexpresiones de una expresión.

Y otros dos conceptos fundamentales que aún no hemos estudiado pero que veremos en breve:

- Abstracciones lambda.
- Funciones de orden superior.

## 6. Documentación interna

### 6.1. Identificadores significativos

Se recomienda usar identificadores descriptivos.

Es mejor usar:

```
ancho = 640
alto = 400
superficie = ancho * alto
```

que

```
x = 640
y = 400
z = x * y
```

aunque ambos programas sean equivalentes en cuanto al efecto que producen y el resultado que generan.

Si el identificador representa varias palabras, se puede usar el carácter de guión bajo (`_`) para separarlas y formar un único identificador:



```
altura_triangulo = 34.2
```

## 6.2. Comentarios

Los comentarios en Python empiezan con el carácter `#` y se extienden hasta el final de la línea.

Los comentarios pueden aparecer al comienzo de la línea o a continuación de un espacio en blanco o una porción de código.

Los comentarios no pueden ir dentro de un literal de tipo cadena.

Un carácter `#` dentro de un literal cadena es sólo un carácter más.

```
# este es el primer comentario
spam = 1 # y este es el segundo comentario
        # ... y este es el tercero
texto = "# Esto no es un comentario porque va entre comillas."
```

Cuando un comentario ocupa varias líneas, se puede usar el «truco» de poner una cadena con triples comillas:

```
x = 1
"""
Esta es una cadena
que ocupa varias líneas
y que actúa como comentario.
"""
y = 2
```

Python evaluará la cadena pero, al no usarse dentro de ninguna expresión ni ligarse a ningún identificador, simplemente la ignorará (como un comentario).

## 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.