

Programación funcional I

Ricardo Pérez López

IES Doñana, curso 2019/2020

Índice general

1. El lenguaje de programación Python	2
1.1. Historia	3
1.2. Características principales	3
2. Modelo de ejecución	4
2.1. Modelo de ejecución	4
2.2. Modelo de sustitución	4
3. Expresiones	5
3.1. Concepto	5
3.2. Evaluación de expresiones	6
3.2.1. Valores, expresión canónica y forma normal	6
3.2.2. Formas normales y evaluación	7
3.2.3. Transparencia referencial	8
3.3. Literales	8
3.4. Operaciones, operadores y operandos	9
3.4.1. Aridad de operadores	10
3.4.2. Paréntesis	10
3.4.3. Prioridad de operadores	10
3.4.4. Asociatividad de operadores	11
3.4.5. Tipos de operandos	11
3.5. Funciones y métodos	12
3.5.1. Funciones	12
3.5.2. Funciones con varios argumentos	14
3.5.3. Evaluación de expresiones con funciones	15
3.5.4. Composición de operaciones y funciones	16
3.5.5. Métodos	17
3.6. Otros conceptos sobre operaciones	18
3.6.1. Sobrecarga de operaciones	18
3.6.2. Equivalencia entre formas de operaciones	19
3.6.3. Igualdad de operaciones	20
3.7. Operaciones predefinidas	20
3.7.1. Operadores predefinidos	20

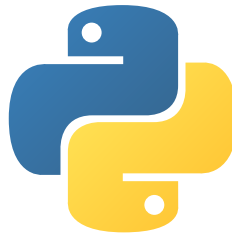
3.7.2. Funciones predefinidas	21
3.7.3. Métodos predefinidos	22
3.8. Actividades	22
4. Tipos de datos	24
4.1. Concepto	24
4.2. <code>type</code>	24
4.3. Sistema de tipos	25
4.3.1. Errores de tipos	25
4.3.2. Tipado fuerte vs. débil	25
4.4. Tipos de datos básicos	26
4.4.1. Números	26
4.4.2. Cadenas	26
4.5. Conversión de tipos	27
5. Álgebra de Boole	27
5.1. El tipo de dato <i>booleano</i>	28
5.2. Operadores relacionales	28
5.3. Operadores lógicos	28
5.3.1. Tablas de verdad	29
5.4. Axiomas	30
5.4.1. Traducción a Python	30
5.5. Teoremas fundamentales	31
5.5.1. Traducción a Python	31
5.6. El operador ternario	32
5.6.1. Actividad	33
6. Definiciones	33
6.1. Introducción	33
6.2. Identificadores y ligaduras (<i>binding</i>)	33
6.2.1. Reglas léxicas	34
6.2.2. Tipo de un identificador	34
6.3. Evaluación de expresiones con ligaduras	34
6.4. Marcos (<i>frames</i>)	35
6.5. Entorno (<i>environment</i>)	37
6.6. <i>Scripts</i>	38
6.7. Ámbitos	38
6.7.1. Ámbito de una ligadura	39
7. Documentación interna	39
7.1. Identificadores significativos	39
7.2. Comentarios	39
Bibliografía	40

1. El lenguaje de programación Python

1.1. Historia

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

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



Logo de Python

Python alcanzó la versión 1.0 en enero de 1994.

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

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

1.2. Características principales

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

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

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

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

La estructura de un programa se define por su anidamiento.

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

```
$ python
Python 3.8.2 (default, Apr 27 2020, 15:53:34)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

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

Para salir, se pulsa `Ctrl+D`.

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

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

2. Modelo de ejecución

2.1. 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 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.

2.2. 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.

La **evaluación de una expresión**, en esencia, es el proceso de **sustituir**, dentro de ella, unas *sub-expresiones* por otras que, de alguna manera, 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**.

La ventaja de este modelo 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, etc.

Todo resulta mucho más fácil que eso.

Todo se reduce a evaluar expresiones.

3. Expresiones

3.1. Concepto

El código fuente de un programa está formado por elementos que pertenecen a dos grandes grupos principales:

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

Las **expresiones** se *evalúan*.

Las **sentencias** se *ejecutan*.

Expresión:

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

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

- Están formados por secuencias de números y símbolos que representan operaciones aritméticas.
- Denotan un valor numérico, que es el resultado de calcular el valor de la expresión tras hacer las operaciones que aparecen en ella.

La expresión $(2 * (3 + 5))$ denota un valor, que es el número abstracto **16**.

En general, las expresiones correctamente formadas satisfacen una gramática similar a la siguiente:

```

<expresión> ::= ( <expresión> <opbin> <expresión> )
              | ( <opun> <expresión> )
              | <literal>
              | <identificador>
  
```

```

      | <identificador>([<lista_argumentos>])
<lista_argumentos> ::= <expresión>(<expresión>)*
<opbin> ::= + | - | * | / | // | ** | %
<opun> ::= + | -

```

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

```
(3 + (4 - 7))
```

3.2. Evaluación de expresiones

Ya hemos visto que la ejecución de un programa funcional consiste, en esencia, en evaluar una expresión.

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

En programación funcional, 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.

Una característica de la programación funcional es que **toda expresión posee un valor definido**, y ese valor no depende del orden en el que se evalúe.

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

Podemos decir que las expresiones:

```

3
(1 + 2)
(5 - 2)

```

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

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

Lo que hace el intérprete es buscar **la representación más simplificada o reducida** posible (en este caso, **3**).

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

3.2.1. Valores, expresión canónica y forma normal

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

- Por ejemplo: utilizan la codificación binaria en complemento a 2 para representar los números enteros.

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

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

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

Con esta restricción pueden quedar expresiones sin forma normal.

Ejemplo:

- De las expresiones anteriores:

3

(1 + 2)

(5 - 2)

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

- Igualmente, la expresión 3 es la **forma normal** de todas las expresiones anteriores (y de cualquier otra expresión con valor 3).
- Es importante no confundir el valor abstracto **3** con la expresión 3 que representa dicho valor.

Hay valores que no tienen expresión canónica:

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

Y hay expresiones que no tienen forma normal:

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

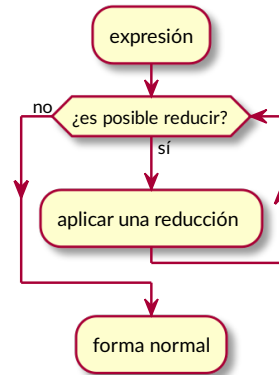
3.2.2. Formas normales y evaluación

A partir de todo lo dicho, la ejecución de un programa será el proceso de encontrar su forma normal.

Un ordenador evalúa una expresión (o ejecuta un programa) buscando su forma normal y mostrando este resultado.

Con los lenguajes funcionales los ordenadores alcanzan este objetivo a través de múltiples pasos de reducción de las expresiones para obtener otra equivalente más simple.

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



3.2.3. Transparencia referencial

En programación funcional, el valor de una expresión depende, exclusivamente, de los valores de sus sub-expresiones constituyentes.

Dichas sub-expresiones, además, pueden ser sustituidas libremente por otras que tengan el mismo valor.

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

En la práctica, eso significa que la evaluación de una expresión no puede provocar **efectos laterales**.

Formalmente, se puede definir así:

Transparencia referencial:

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

3.3. Literales

Un **literal** es un valor escrito directamente en el código del programa (en una expresión).

El literal representa un **valor constante**.

Los literales tienen que satisfacer las reglas de sintaxis del lenguaje.

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

Se deduce, pues, que **un literal debe ser la expresión canónica del valor correspondiente**.

Ejemplos de distintos tipos de literales:

Números enteros	Números reales	Cadenas
-2	3.5	"hola"
-1	-2.7	"pepe"
0		"25"

Números enteros	Números reales	Cadenas
1		" "
2		

Los números reales tienen siempre un `.` decimal.

Las cadenas van siempre entre comillas (simples `'` o dobles `"`).

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

3.4. Operaciones, operadores y operandos

En una expresión puede haber:

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

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

- Operadores
- Funciones
- Métodos



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

Ejemplos:

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

`(3 + 4)`

(aquí los operandos son los números `3` y `4`)

`(9 * 8)`

(aquí los operandos son los números `9` y `8`)

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

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

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

3.4.1. Aridad de operadores

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

- **Unarios:** operan sobre un único operando.

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

```
(-5)
```

- **Binarios:** operan sobre dos operandos.

Ejemplo: la mayoría de operadores aritméticos.

- **Ternarios:** operan sobre tres operandos.

Veremos un ejemplo más adelante.

3.4.2. Paréntesis

Los **paréntesis** sirven para agrupar elementos dentro de una expresión y romper la ambigüedad sobre el orden en el que se han de realizar las operaciones.

Se usan, sobre todo, para hacer que varios elementos actúen como uno solo en el contexto de una operación.

- Por ejemplo:

$((3 + 4) * 5)$ vale 35

$(3 + (4 * 5))$ vale 23

Para reducir la cantidad de paréntesis en una expresión, se puede:

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

3.4.3. Prioridad de operadores

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

```
8 + 4 * 2
```

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

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

Si hiciéramos

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

el resultado sería distinto.

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

3.4.4. Asociatividad de operadores

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

```
8 / 4 / 2
```

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

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

Si hiciéramos

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

el resultado sería distinto.

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

3.4.5. Tipos de operandos

Es importante respetar el tipo de los operandos que espera recibir un operador. Si los intentamos aplicar sobre operandos de tipos incorrectos, obtendremos resultados inesperados (o, directamente, un error).

Por ejemplo, los operadores aritméticos esperan operandos de tipo *numérico*. Así, si intentamos dividir dos cadenas usando el operador `/`:

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

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

3.5. Funciones y métodos

3.5.1. Funciones

Las funciones son otra forma de representar operaciones.

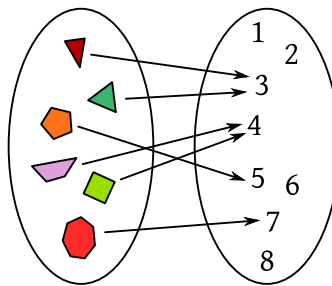
Matemáticamente, una **función** es una regla que **asocia** a cada elemento de un conjunto (el conjunto *origen* o *dominio*) un único elemento de un segundo conjunto (el conjunto *imagen* o *codominio*).

Se representa así:

$$f : A \rightarrow B$$

$$x \rightarrow f(x)$$

donde A es el conjunto *origen* y B el conjunto *imagen*.



Función que asocia a cada polígono con su número de lados

La expresión $f : A \rightarrow B$ es la **declaración** o **signatura** de la función.

La **aplicación de la función** f sobre el elemento x se representa por $f(x)$ y corresponde al valor que la función asocia al elemento x en el conjunto imagen.

En la aplicación $f(x)$, al valor x se le llama **argumento** de la función.

Por ejemplo:

La función **valor absoluto**, que asocia a cada número entero ese mismo número sin el signo (un número natural). Su signatura es:

$$abs : \mathbb{Z} \rightarrow \mathbb{N}$$

$$x \rightarrow abs(x)$$

Cuando aplicamos la función abs al valor -35 obtenemos:

$$abs(-35) = 35$$

El valor 35 es el **resultado** de aplicar la función abs al argumento -35 .

Otra forma de expresarlo es decir que la función *abs* **recibe** un argumento de tipo entero y **devuelve** un resultado de tipo natural.

La función *abs* se puede usar directamente en Python ya que una función **predefinida** en el lenguaje. Por ejemplo:

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

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

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

Otro ejemplo:

$$\begin{aligned} \textit{longitud} : \mathbb{C} &\rightarrow \mathbb{N} \\ x &\rightarrow \textit{longitud}(x) \end{aligned}$$

La función *longitud* asocia a cada cadena su longitud (la *longitud* de una cadena es el número de caracteres que contiene).

También puede decirse que devuelve la longitud de la cadena que recibe como argumento.

Así:

$$\textit{longitud}(\textit{hola}) = 4$$

En Python, la función *longitud* se llama `len`:

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

En Programación, a la aplicación de una función también se le denomina **invocación** o **llamada** a la función.

Por ejemplo, cuando hacemos `abs(-35)` podemos decir que:

- Estamos *aplicando* la función *abs* al argumento `-35`.
- Estamos *llamando* a la función *abs* (con el argumento `-35`).
- Estamos *invocando* a la función *abs* (con el argumento `-35`).

De hecho, en Programación es mucho más común decir «*se llama a la función*» que decir «*se aplica la función*».

También se suele decir que «*se ejecuta la función*».

3.5.2. Funciones con varios argumentos

El concepto de función se puede generalizar para obtener **funciones con más de un argumento**.

Por ejemplo, podemos definir una función *max* que asocie, a cada par de números enteros, el máximo de los dos:

$$\begin{aligned} \text{max} : \mathbb{Z} \times \mathbb{Z} &\rightarrow \mathbb{Z} \\ (x, y) &\rightarrow \text{max}(x, y) \end{aligned}$$

También podemos decir que *max* es una función que recibe dos argumentos enteros y devuelve un entero.

Si aplicamos la función *max* a los argumentos 13 y -25, el resultado sería 13:

$$\text{max}(13, -25) = 13$$

Al igual que ocurre con los operadores, la **aridad de una función** es el número de argumentos que necesita la función.

El símbolo \times representa el **producto cartesiano** de dos conjuntos, y devuelve todas las parejas que se pueden formar emparejando elementos del primer conjunto con elementos del segundo conjunto.

Por ejemplo, si tenemos los conjuntos $A = \{1, 2, 3\}$ y $B = \{a, b\}$, el producto cartesiano $A \times B$ resultaría:

$$A \times B = \{(1, a), (1, b), (2, a), (2, b), (3, a), (3, b)\}$$

Este concepto es fundamental en **bases de datos**.

Otro ejemplo de función con varios argumentos:

$$\begin{aligned} \text{pow} : \mathbb{R} \times \mathbb{R} &\rightarrow \mathbb{R} \\ (x, y) &\rightarrow \text{pow}(x, y) \end{aligned}$$

La función *pow* recibe dos números (el primero es la *base* y el segundo es el *exponente*) y devuelve el resultado de elevar la base al exponente. Es decir, $\text{pow}(x, y) = x^y$.

Por ejemplo, al aplicar la función *pow* sobre los valores 3 y 2, obtenemos:

$$\text{pow}(3, 2) = 9$$

Es importante respetar el *orden* de los argumentos. El primero siempre es la base y el segundo siempre es el exponente. Si los pasamos al revés, tendríamos un resultado diferente:

$$\text{pow}(2, 3) = 8$$

Curiosamente, la operación de elevar un número a la potencia del otro existe en Python de dos formas diferentes:

- Como operador (`**`):

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

- Como función (`pow`):

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

En ambos casos, la operación es exactamente la misma.

3.5.3. Evaluación de expresiones con funciones

En una expresión podemos colocar aplicaciones de función en cualquier lugar donde sea sintácticamente correcto situar un valor.

La evaluación de una expresión que contiene aplicaciones de funciones se realiza sustituyendo (*reduciendo*) cada aplicación por su valor correspondiente, es decir, por el valor que dicha función asocia a sus argumentos.

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

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

Aquí se aplica la función *abs* al argumento -12 y la función *max* a los argumentos 13 y 28, y finalmente se suman los dos valores obtenidos.

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

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

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

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

¿Cuál se evalúa primero?

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

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

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

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

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

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

Sin embargo, Python no es un lenguaje funcional puro, y llegado el caso será importante tener en cuenta el orden de evaluación que aplica.

En concreto, y mientras no se diga lo contrario, **Python siempre evalúa las expresiones de izquierda a derecha**.

Importante:

En **Python**, salvo excepciones, los operandos y los argumentos de las funciones se evalúan **de izquierda a derecha**.

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

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

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

```
abs(-12) + max(13, 28)  # se evalúa primero abs(-12)
= 12 + max(13, 28)      # ahora se evalúa max(13, 28)
= 12 + 28               # se evalúa el operador +
= 40
```

3.5.4. Composición de operaciones y funciones

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

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

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

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

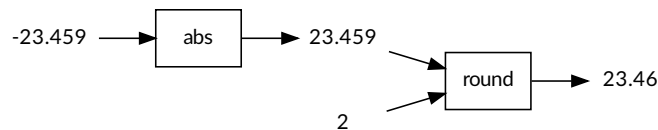
La manera más sencilla de realizar varias operaciones sobre los mismos datos es componer las operaciones, es decir, hacer que el resultado de una operación sea la entrada de otra operación.

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

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

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

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



3.5.5. Métodos

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

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

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

$$v.m()$$

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

Esta llamada se puede leer de cualquiera de estas formas:

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

Los métodos también pueden tener argumentos, como cualquier función:

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

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

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

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

y tener una función y hacer:

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

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

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

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

Por ejemplo:

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

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

devuelve 2.

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

devuelve 4.

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

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

3.6. Otros conceptos sobre operaciones

3.6.1. Sobrecarga de operaciones

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

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

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

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

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

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

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

3.6.2. Equivalencia entre formas de operaciones

Una operación podría 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 *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
```

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

- El operador **in**:

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

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

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

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

De hecho, en Python hay operaciones que tienen **las tres formas**. Por ejemplo, la suma de dos números enteros se puede expresar:

- Mediante el operador **+**:

```
4 + 3
```

- Mediante la función **int.__add__**:

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

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

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

La forma **más general** y destacada de representar una operación es la **función**, ya que cualquier operador o método se puede expresar en forma de función (lo contrario no siempre es cierto).

Es decir: los operadores y los métodos son formas sintácticas especiales para expresar operaciones que se podrían expresar igualmente mediante funciones.

Por eso, cuando hablemos de operaciones, y mientras no se diga lo contrario, supondremos que están representadas como funciones.

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

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

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

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

3.6.3. Igualdad de operaciones

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

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

Principio de extensionalidad:

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

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

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

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

3.7. Operaciones predefinidas

3.7.1. Operadores predefinidos

3.7.1.1. Operadores aritméticos

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

3.7.1.2. Operadores de cadenas

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

3.7.2. Funciones predefinidas

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

3.7.2.1. Funciones matemáticas

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

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

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

Para *importar* una función de un módulo se puede usar la orden `from`. Por ejemplo, para importar la función `gcd` (que calcula el máximo común divisor de dos números) del módulo `math` se haría:

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

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

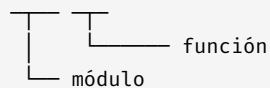
También se puede importar directamente el módulo en sí:

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

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

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

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



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

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

3.7.3. Métodos predefinidos

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

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

3.8. Actividades

1. Representar, según el modelo de sustitución, la evaluación las siguientes expresiones, aplicando paso a paso la reducción que corresponda. Indicar también el tipo del valor resultante:

- a. $3 + 6 * 14$
- b. $8 + 7 * 3.0 + 4 * 6$

- c. `-4 * 7 + 2 ** 3 / 4 - 5`
 d. `4 / 2 * 3 / 6 + 6 / 2 / 1 / 5 ** 2 / 4 * 2`

2. Convertir en expresiones aritméticas algorítmicas las siguientes expresiones algebraicas:

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

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

- a. `(8 + (7 * 3) + 4 * 6)`
 b. `-(2 ** 3)`
 c. `(33 + (3 * 4)) / 5`
 d. `2 ** (2 * 3)`
 e. `(3.0) + (2 * (18 - 4 ** 2))`
 f. `(16 * 6) - (3) * 2`

4. Usar la función `math.sqrt` para escribir dos expresiones en Python que calculen las dos soluciones a la ecuación de segundo grado $ax^2 + bx + c = 0$.

Recordar que las soluciones son:

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

5. Evaluar las siguientes expresiones:

- a. `9 - 5 - 3`
 b. `2 // 3 + 3 / 5`
 c. `9 // 2 / 5`
 d. `7 % 5 % 3`
 e. `7 % (5 % 3)`
 f. `(7 % 5) % 3`
 g. `(7 % 5 % 3)`
 h. `((12 + 3) // 2) / (8 - (5 + 1))`
 i. `12 / 2 * 3`
 j. `math.sqrt(math.cos(4))`
 k. `math.cos(math.sqrt(4))`
 l. `math.trunc(815.66) + round(815.66)`

6. Escribir las siguientes expresiones algorítmicas como expresiones algebraicas:

- a. `b ** 2 - 4 * a * c`
 b. `3 * x ** 4 - 5 * x ** 3 + x * 12 - 17`
 c. `(b + d) / (c + 4)`
 d. `(x ** 2 + y ** 2) ** (1 / 2)`

4. Tipos de datos

4.1. Concepto

Los datos que comparten características y propiedades se agrupan en **conjuntos**.

Asimismo, sobre cada conjunto de valores se definen una serie de **operaciones**, que son aquellas que tiene sentido realizar con esos valores.

Un **tipo de datos** define un conjunto de **valores** y el conjunto de **operaciones** válidas que se pueden realizar sobre dichos valores.

Definición:

Tipo de un dato:

Es una característica del dato que indica el conjunto de *valores* al que pertenece y las *operaciones* que se pueden realizar sobre él.

El **tipo de una expresión** es el tipo del valor resultante de evaluar dicha expresión.

Ejemplos:

- El tipo `int` en Python define el conjunto de los **números enteros**, sobre los que se pueden realizar las operaciones aritméticas (suma, producto, etc.) entre otras.
 - * Se corresponde *más o menos* con el símbolo \mathbb{Z} que ya hemos usado antes.
- El tipo `str` define el conjunto de las **cadenas**, sobre las que se pueden realizar otras operaciones (la *concatenación*, la *repetición*, etc.).
 - * Se corresponde *más o menos* con el símbolo \mathbb{C} que ya hemos usado antes.

(¿Por qué «más o menos»?)

4.2. `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'>
```


4.3. 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...*).

4.3.1. 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.

4.3.2. 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*.

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 5).

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 (+).

4.4. Tipos de datos básicos

4.4.1. 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`.

4.4.2. 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 respetan los saltos de línea.
- Ejemplo:

```
"""Bienvenido  
a  
Python"""
```

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 el literal `' '` o `''`.

4.5. 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'
```

5. Álgebra de Boole

5.1. 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` (con la inicial en mayúscula).

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

5.2. 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
```

5.3. 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.

5.3.1. 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

5.4. Axiomas

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

Luego $(\mathfrak{B}, \neg, \vee, \wedge)$ es un álgebra de Boole.

5.4.1. 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
```

5.5. Teoremas fundamentales

6. Ley de idempotencia: $\begin{cases} \forall a \in \mathfrak{B} : a \vee a = a \\ \forall a \in \mathfrak{B} : a \wedge a = a \end{cases}$

7. Ley de absorción: $\begin{cases} \forall a \in \mathfrak{B} : a \vee V = V \\ \forall a \in \mathfrak{B} : a \wedge F = F \end{cases}$

8. Ley de identidad: $\begin{cases} \forall a \in \mathfrak{B} : a \vee F = a \\ \forall a \in \mathfrak{B} : a \wedge V = a \end{cases}$

9. Ley de involución: $\begin{cases} \forall a \in \mathfrak{B} : \neg\neg a = a \\ \neg V = F \\ \neg F = V \end{cases}$

10. Leyes de De Morgan: $\begin{cases} \forall a, b \in \mathfrak{B} : \neg(a \vee b) = \neg a \wedge \neg b \\ \forall a, b \in \mathfrak{B} : \neg(a \wedge b) = \neg a \vee \neg b \end{cases}$

5.5.1. Traducción a Python

6. Ley de idempotencia:

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

7. Ley de absorción:

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

8. Ley de identidad:

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

9. Ley de involución:

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

10. Leyes de De Morgan:

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

5.6. 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.

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.

5.6.1. Actividad

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

6. Definiciones

6.1. Introducción

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

A esa sentencia (en este momento) 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.

6.2. Identificadores y ligaduras (*binding*)

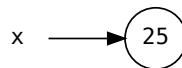
Los **identificadores** son los nombres o símbolos que representan a los elementos del lenguaje.

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

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

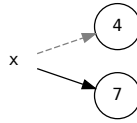
Por esa razón, también se dice que una definición es una ligadura.

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



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

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



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

- Eso hace que se pierda el valor anterior.
- Por ahora, **no lo hagamos**.

6.2.1. 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.

6.2.2. 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 de dato tiene.

6.3. Evaluación de expresiones con ligaduras

Podemos usar un identificador ligado dentro de una expresión (siempre que la expresión sea una expresión válida según las reglas del lenguaje, claro está):

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

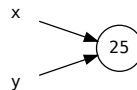
Intentar usar en una expresión un identificador no ligado provoca un error (*nombre no definido*):

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

Podemos hacer:

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

- En este caso estamos ligando a *y* el mismo valor que tiene *x*.
- *x* e *y* comparten valor.



6.4. Marcos (*frames*)

Un **marco** (del inglés *frame*) es un **conjunto de ligaduras**.

Las ligaduras se almacenan en marcos.

En un marco, un identificador sólo puede tener como máximo una ligadura. En cambio, **el mismo identificador puede estar ligado a diferentes valores en diferentes marcos**.

Los marcos son conceptos **dinámicos**:

- Se crean en memoria cuando la ejecución del programa entra en ciertas partes del mismo y se destruyen cuando sale.
- Van incorporando nuevas ligaduras a medida que se van ejecutando nuevas instrucciones.

El **marco global** es un marco que siempre existe en cualquier punto del programa y contiene las ligaduras definidas fuera de cualquier construcción o estructura del mismo.

- Por ahora es el único marco que existe para nosotros.

En un momento dado, un marco contendrá las ligaduras que se hayan definido hasta ese momento en el *ámbito* asociado a ese marco:

```
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 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*).

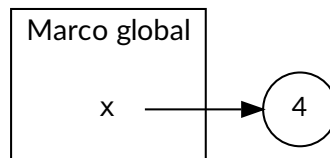
- En la línea 6, en cambio, el identificador puede usarse sin error ya que ha sido ligado previamente en la línea 5 (el marco global ahora contiene una ligadura para `x` con el valor 25).

Los marcos se van creando y destruyendo durante la ejecución del programa, y su contenido (las ligaduras) también va cambiando con el tiempo, a medida que se van ejecutando sus instrucciones.

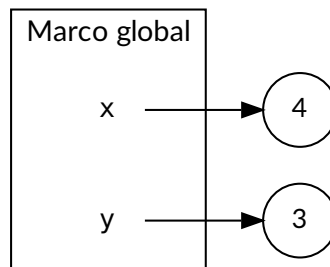
Si tenemos:

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

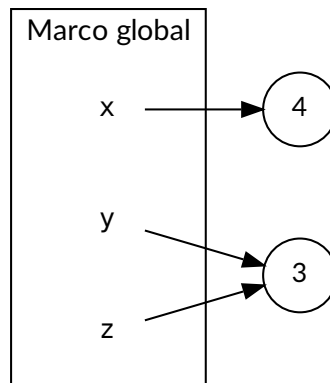
Según la línea hasta donde hayamos ejecutado, el marco global contendrá lo siguiente:



Marco global en la línea 1



Marco global en la línea 2



Marco global en la línea 3

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

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**.

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

6.5. Entorno (*environment*)

El entorno es una extensión del concepto de *marco*.

Un **entorno** (del inglés, *environment*) es una **secuencia o cadena de marcos** que contienen todas las ligaduras válidas en un momento concreto de la ejecución del programa.

Es decir, el entorno nos dice **qué identificadores son accesibles en un momento dado, y con qué valores están ligados**.

El entorno, por tanto, es un concepto **dinámico** que depende del momento en el que se calcule, es decir, de por dónde va la ejecución del programa o, lo que es lo mismo, de qué instrucciones se han ejecutado hasta ahora.

Ya hemos visto que, durante la ejecución del programa, se van creando y destruyendo marcos a medida que la ejecución va entrando y saliendo de ciertas partes del programa.

Según se van creando en memoria, esos marcos van enlazándose unos con otros creando la secuencia de marcos que forman el entorno.

Por tanto, en un momento dado, el entorno contendrá más o menos marcos dependiendo de por dónde haya pasado la ejecución del programa hasta ese momento.

El entorno **siempre contendrá**, al menos, un marco: el *marco global*.

El marco global siempre será el último de la cadena de marcos que forman el entorno.

Como por ahora sólo tenemos ese marco, nuestro entorno sólo contendrá un único marco. Por tanto, el entorno coincidirá con el marco global.

La cosa cambiará en cuanto empecemos a crear funciones.

6.6. Scripts

Cuando tenemos varias definiciones o muy largas resulta tedioso tener que introducirlas una y otra vez en el intérprete interactivo.

Lo más cómodo es teclearlas juntas 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.

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

Para cargar un *script* en nuestra sesión usamos la orden `from`. Por ejemplo, para cargar un *script* llamado `definiciones.py`, usaremos:

```
from definiciones import *
```

6.7. Ámbitos

Existen ciertas construcciones sintácticas que, cuando se ejecutan, provocan la creación de nuevos marcos.

Cuando eso ocurre, decimos que la construcción sintáctica define un **ámbito**, y que el ámbito viene definido por la porción de texto que ocupa esa construcción sintáctica dentro del programa.

Durante la ejecución del programa, se creará un nuevo marco cuando se entre en el ámbito (es decir, cuando se entre en su construcción sintáctica correspondiente) y se destruirá cuando se salga del ámbito.

Los ámbitos **se anidan recursivamente**, o sea, que están contenidos unos dentro de otros.

El **ámbito actual** es el ámbito más interno en el que se encuentra la porción de código que se está ejecutando actualmente.

El concepto de *ámbito* es un concepto nada trivial y, a medida que vayamos incorporando nuevos elementos al lenguaje, tendremos que ir adaptándolo para tener en cuenta más condicionantes.

Por ahora sólo tenemos un ámbito que abarca todo el *script* que se está ejecutando (o la sesión actual si estamos en el intérprete interactivo).

A ese ámbito se le llama **ámbito global** y es el que crea el **marco global**.

Es decir: el intérprete crea el marco global cuando empieza a ejecutar el *script* (o cuando inicia una nueva sesión con el intérprete interactivo) y lo asocia al ámbito global.

6.7.1. Ámbito de una ligadura

El **ámbito de una ligadura** es el ámbito en el que dicha ligadura tiene validez.

Hasta ahora, todas las ligaduras las hemos definido en el ámbito global, por lo que se almacenan en el marco global.

Por eso también decimos que esas ligaduras tienen ámbito global, o que pertenecen al ámbito global, o que están definidas en el ámbito global, o que son **globales**.

7. Documentación interna

7.1. 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
```

7.2. 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.