

# Programación funcional I

Ricardo Pérez López

IES Doñana, curso 2019/2020

## Índice general

<b>1. El lenguaje de programación Python</b>	<b>2</b>
1.1. Historia . . . . .	2
1.2. Características principales . . . . .	2
<b>2. Modelo de ejecución</b>	<b>2</b>
2.1. Modelo de ejecución . . . . .	2
2.2. Modelo de sustitución . . . . .	3
<b>3. Expresiones</b>	<b>3</b>
3.1. Evaluación de expresiones . . . . .	3
3.1.1. Transparencia referencial . . . . .	4
3.1.2. Valores, expresión canónica y forma normal . . . . .	4
3.1.3. Formas normales y evaluación . . . . .	5
3.2. Literales . . . . .	6
3.3. Operaciones, operadores y operandos . . . . .	6
3.3.1. Aridad de operadores . . . . .	7
3.3.2. Paréntesis . . . . .	7
3.3.3. Asociatividad de operadores . . . . .	8
3.3.4. Precedencia de operadores . . . . .	8
3.4. Tipos de datos . . . . .	8
3.4.1. Concepto . . . . .	8
3.4.2. Tipos de datos básicos . . . . .	8
3.5. Algebraicas vs. algorítmicas . . . . .	9
3.6. Aritméticas . . . . .	9
3.7. Operaciones predefinidas . . . . .	9
3.7.1. Operadores predefinidos . . . . .	9
3.7.2. Funciones predefinidas . . . . .	9
3.7.3. Métodos predefinidos . . . . .	9
3.8. Constantes predefinidas . . . . .	9
<b>4. Álgebra de Boole</b>	<b>9</b>
4.1. El tipo de dato <i>booleano</i> . . . . .	9
4.2. Operadores relacionales . . . . .	9
4.3. Operadores lógicos . . . . .	10

4.4. Axiomas . . . . .	10
4.4.1. Traducción a Python . . . . .	10
4.5. Teoremas fundamentales . . . . .	11
4.5.1. Traducción a Python . . . . .	11
4.6. El operador ternario . . . . .	12
<b>5. Variables y constantes</b>	<b>12</b>
5.1. Definiciones . . . . .	12
5.2. Identificadores . . . . .	12
5.3. Ligadura ( <i>binding</i> ) . . . . .	12
5.4. Estado . . . . .	12
5.5. Tipado estático vs. dinámico . . . . .	12
5.6. Evaluación de expresiones con variables . . . . .	12
5.7. Constantes . . . . .	13
<b>6. Documentación interna</b>	<b>13</b>
6.1. Identificadores significativos . . . . .	13
6.2. Comentarios . . . . .	13
6.3. Docstrings . . . . .	13
<b>Respuestas a las preguntas</b>	<b>13</b>
<b>Bibliografía</b>	<b>13</b>

## 1. El lenguaje de programación Python

### 1.1. Historia

### 1.2. Características principales

## 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 y que el código fuente del programa.
- La **evaluación de una expresión**, en esencia, consiste en **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. 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** un valor.
- 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**, a diferencia de otros paradigmas en los que, por ejemplo, existen las *sentencias*, que no poseen ningún valor.
- Además, el orden en el que se evalúe no debe influir en el resultado.
- Podemos decir que las expresiones:  
3  
 $1 + 2$   
 $5 - 3$   
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 sistema 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*.

### 3.1.1. 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  $f(p) = f(q)$ .

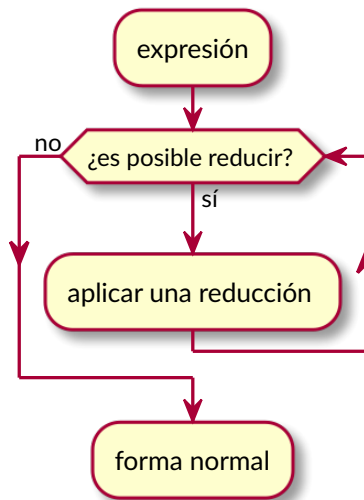
### 3.1.2. Valores, expresión canónica y forma normal

- 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
    - \*  $1 + 2$
    - \*  $5 - 3$
  - 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  $\pi$  no tiene representación decimal finita, por lo que tampoco tiene expresión canónica.
- Y hay expresiones que no tienen forma normal:
  - Si definimos  $inf = inf + 1$ , la expresión  $inf$  (que es un número) no tiene forma normal.
  - Lo mismo ocurre con  $\frac{1}{0}$ .

### 3.1.3. 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. Literales

- Un **literal** es un valor escrito directamente en el código del programa (en una expresión).
- El literal representa un valor constante.
- Ejemplos:
  - 3, -2, -1, 0, 1, 2, 3 (literales que representan números enteros)
  - 3.5, -2.7 (literales que representan números reales)
  - "hola", "pepe", "25", "" (literales de tipo cadena)
- 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**.

### 3.3. Operaciones, operadores y operandos

- En una expresión puede haber:
  - **Datos**
  - **Operaciones** a realizar sobre esos datos
- A su vez, las operaciones se pueden representar en forma de:
  - Operadores
  - Funciones

- Métodos
- Empezaremos hablando de los operadores.
- Un **operador** es un símbolo o palabra clave que representa la realización de una *operación* sobre unos datos llamados **operandos**.
- Ejemplos:
  - Los operadores aritméticos: `+`, `-`, `*`, `/` (entre otros):

`3 + 4`

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

`9 * 8`

(aquí los operandos son los números `9` y `8`)
  - El operador `in` para comprobar si un carácter pertenece a una cadena:

`"c" in "barco"`

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

### 3.3.1. 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.

### 3.3.2. Paréntesis

- Los **paréntesis** sirven para agrupar elementos dentro de una expresión.
- 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

### 3.3.3. Asociatividad de operadores

- En ausencia de paréntesis, cuando un operando está afectado a derecha e izquierda por el **mismo operador**, se aplican las reglas de la **asociatividad**:

$$8 / 4 / 2$$

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

$$(8 / 4) / 2$$

Si hiciéramos

$$8 / (4 / 2)$$

el resultado sería distinto.

### 3.3.4. Precedencia 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.

## 3.4. Tipos de datos

### 3.4.1. Concepto

#### 3.4.1.1. Tipo de un valor

#### 3.4.1.2. Tipo de una expresión

### 3.4.2. Tipos de datos básicos

#### 3.4.2.1. Números



#### 3.4.2.1.1. Operadores aritméticos

#### 3.4.2.2. Cadenas

### 3.5. Algebraicas vs. algorítmicas

### 3.6. Aritméticas

### 3.7. Operaciones predefinidas

#### 3.7.1. Operadores predefinidos

#### 3.7.2. Funciones predefinidas

#### 3.7.3. Métodos predefinidos

### 3.8. Constantes predefinidas

## 4. Álgebra de Boole

### 4.1. El tipo de dato *booleano*

- Un dato **lógico** o *booleano* es aquel que puede tomar uno de dos posibles valores, que se denotan normalmente como **verdadero** y **falso**.
- Esos dos valores tratan de representar los dos valores de verdad de la **lógica** y el **álgebra booleana**.
- Su nombre proviene de **George Boole**, matemático que definió por primera vez un sistema algebraico para la lógica a mediados del S. XIX.
- En Python, el tipo de dato lógico se representa como `bool` y sus posibles valores son `False` y `True` (con la inicial en mayúscula).
- Esos dos valores son *formas especiales* para los enteros 0 y 1, respectivamente.

### 4.2. Operadores relacionales

- Los **operadores relacionales** son operadores que toman dos operandos (que usualmente deben ser del mismo tipo) y devuelven un valor *booleano*.
- Los más conocidos son los **operadores de comparación**, que sirven para comprobar si un dato es menor, mayor o igual que otro, según un orden preestablecido.
- Los operadores de comparación que existen en Python son:

< > <= >= == !=

### 4.3. Operadores lógicos

- Los **operadores lógicos** son operadores que toman uno o dos operandos *booleanos* y devuelven un valor *booleano*.
- Representan las operaciones básicas del álgebra de Boole llamadas **suma**, **producto** y **complemento**.
- En **lógica proposicional** (un tipo de lógica matemática que tiene estructura de álgebra de Boole), se llaman:
  - **Disyunción** ( $\vee$ ),
  - **Conjunción** ( $\wedge$ ) y
  - **Negación** ( $\neg$ ).
- En Python se representan como **or**, **and** y **not**, respectivamente

### 4.4. 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.

#### 4.4.1. 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
```

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

### 4.5.1. 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)
```

## 4.6. 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:

```
<expresión_condicional> ::= <valor_si_verdadero> if <condición> else <valor_si_falso>
```

- donde:
  - *<condición>* debe ser una expresión lógica
  - *<valor\_si\_verdadero>* y *<valor\_si\_falso>* pueden ser expresiones de cualquier tipo

## 5. Variables y constantes

### 5.1. Definiciones

### 5.2. Identificadores

### 5.3. Ligadura (*binding*)

### 5.4. Estado

### 5.5. Tipado estático vs. dinámico

### 5.6. Evaluación de expresiones con variables

## 5.7. Constantes

# 6. Documentación interna

## 6.1. Identificadores significativos

## 6.2. Comentarios

## 6.3. Docstrings

## Respuestas a las preguntas

### 6.3.0.1. Respuestas a las preguntas

## 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.