Programación funcional I

Ricardo Pérez López

IES Doñana, curso 2019/2020

Índice general

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 *subexpresiones* 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

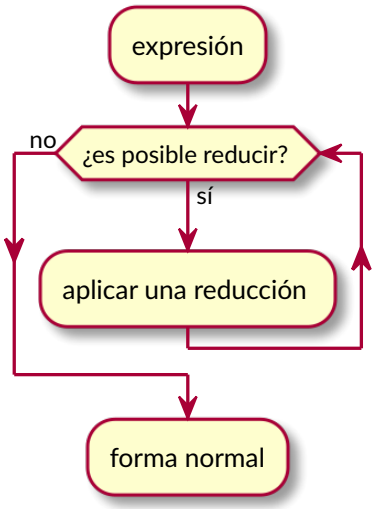
- Los ordenadores no manipulan valores, sino que sólo pueden manejar representaciones concretas de los mismos.
  - Por ejemplo: utilizan la codificación binaria en complemento a 2 para representar los números enteros.
- Pidamos que la **representación del valor** resultado de una evaluación sea **única**.
- 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 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 $\frac{1}{0}$.

3.1.3. 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.

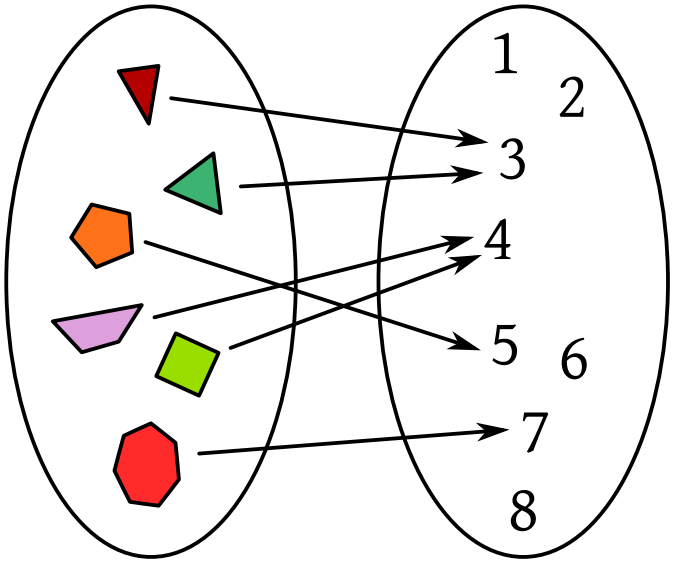
3.4. Funciones y métodos

3.4.1. Funciones

- 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:

$$f : A \to B$$
$$x \to f(x)$$

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



Función que asocia a cada polígono con su número de lados

- La **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).

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

$$x \to abs(x)$$

  Cuando aplicamos la función *abs* al valor -35 obtenemos:

$$abs(-35) = 35$$

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

3.4.1.1. 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 el máximo de los dos.

$$max : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$$

$$(x, y) \to max(x, y)$$

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

$$max(13, -25) = 13$$

3.4.2. Métodos

- Los **métodos** son, para la programación orientada a objetos, el equivalente a las **funciones** para la programación funcional.
- Los métodos son como funciones que actúan *sobre* un valor.
- La *aplicación* de un método se denomina **invocación** o **llamada** al método, y se escribe:

$$v.m()$$

  que representa la **invocación** del método *m* sobre el valor *v*.
- Los métodos también pueden tener argumentos como cualquier función:

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

- En la práctica, no hay mucha diferencia entre hacer:

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

  y hacer

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

- Pero conceptualmente, hay una gran diferencia entre un estilo y otro:
  - El primero es más **orientado a objetos** (el *objeto v* «recibe» un mensaje solicitando la ejecución del método *m*).
  - En cambio, el segundo es más **funcional** (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')
  ```

- Una operación podría tener *forma* de operador, de función o de método.
- De hecho, en Python hay operaciones que tienen **las tres formas**.
- Por ejemplo, la suma de dos números 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 números (y pasando el otro número como *argumento* del método):

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

3.5. Tipos de datos

3.5.1. 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.
  - El tipo `str` define el conjunto de las **cadenas**, sobre las que se pueden realizar otras operaciones (la *concatenación*, la *indexación*, etc.).

3.5.1.1. 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*…).

3.5.1.2. 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 (`+`).

3.5.1.3. 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.

3.5.2. Tipos de datos básicos

3.5.2.1. 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.
- 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`.

3.5.2.2. 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`).

3.6. Operaciones predefinidas

3.6.1. Operadores predefinidos

3.6.1.1. Operadores aritméticos

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

3.6.1.2. Operadores de cadenas

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

3.6.2. Funciones predefinidas

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

3.6.3. Métodos predefinidos

- ...

3.7. 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
- El valor de la expresión completa será *<valor_si_verdadero>* 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. 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.