# Expresiones

Ricardo Pérez López

IES Doñana, curso 2020/2021

# 1. El lenguaje de programación Python

# 1.1. Historia

# 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**.
  - 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.

Logo de Python

# 1.2. Características principales

# 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. Elementos de un programa

# 2.1. Expresiones y sentencias

# Expresiones y sentencias

- 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.

En un programa hay
- **Expresiones**, formadas por
  - **Datos**
  - **Operaciones**
- **Sentencias**, que también pueden contener expresiones

Las **expresiones** se *evalúan*.

Las **sentencias** se *ejecutan*.

# 2.2. Conceptos básicos

# Conceptos básicos

**Definición:**

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. Valores

# 3.1. Evaluación de expresiones

# Evaluación de expresiones

- ***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.
- 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.
- 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 ciertos 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.
- Cuando introducimos una expresión en el intérprete, lo que hace éste es buscar **la representación más simplificada o reducida** posible.
  - En el ejemplo anterior, sería la expresión `3`.
- Por eso a menudo usamos, indistintamente, los términos *reducir*, *simplificar* y *evaluar*.

# 3.2. Expresión canónica y forma normal

# 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)
    ```

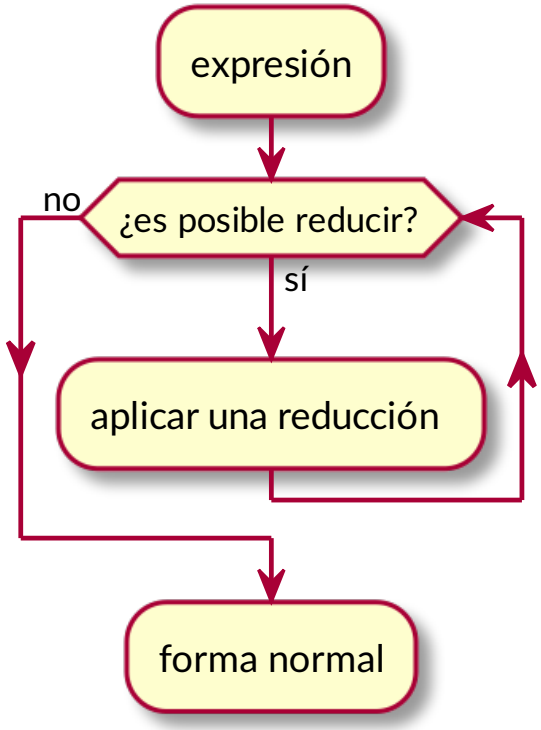
    que denotan todas el mismo valor abstracto **3**, seleccionamos una (la expresión `3`) como la **expresión canónica** de ese valor.
  - Igualmente, la expresión `3` es la **forma normal** de todas las expresiones anteriores (y de cualquier otra expresión con valor **3**).
  - Es importante no confundir 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 $\pi$ 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 $\frac{1}{0}$.

# 3.3. Formas normales y evaluación

# Formas normales y evaluación

- Según lo visto hasta ahora, la evaluación de una expresión es el proceso de encontrar su forma normal.
- El intérprete evalúa una expresión buscando su forma normal y mostrando este resultado.
- El sistema de evaluación dentro del intérprete está hecho de forma tal que cuando ya no es posible reducir la expresión es porque se ha llegado a la forma normal.



# 3.4. Literales

# 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"` |
| `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.

# 4. Operaciones

# 4.1. Clasificación

# Clasificación

- 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

# 4.2. Operadores

# 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"`)

# 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.

# 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.

# 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.

# 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.

# 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.

# 4.3. Funciones

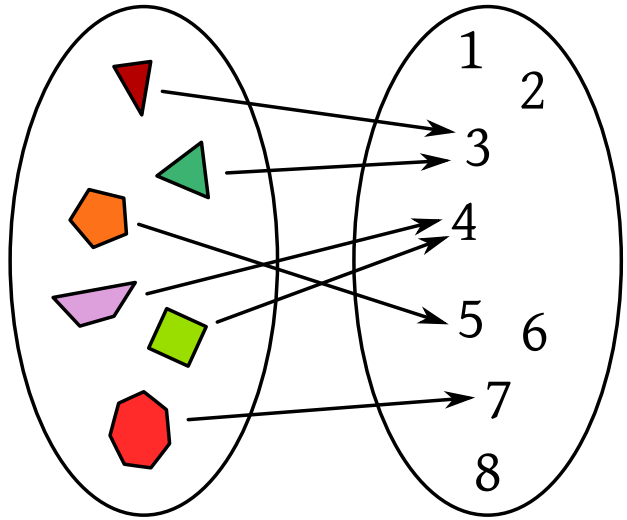
# Funciones

- 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:

$$f : A \to B$$

$$x \to f(x)$$

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




Función que asocia a cada polígono con su número de lados

- La expresión $f : A \to B$ 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:

$$longitud : \mathbb{C} \rightarrow \mathbb{N}$$

$$x \rightarrow longitud(x)$$

- 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:

$$longitud(hola) = 4$$

- 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*».

# 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:

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

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

  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:

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

- 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:

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

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

- Otro ejemplo de función con varios argumentos:

$$pow : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$$
$$(x, y) \to pow(x, y)$$

- 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, $pow(x, y) = x^y$.
- Por ejemplo, al aplicar la función *pow* sobre los valores 3 y 2, obtenemos:

$$pow(3, 2) = 9$$

- 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:

$$pow(2, 3) = 8$$

- 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.

# 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.
$$\begin{cases} \underline{abs(-12)} + max(13, 28) \\ = 12 + \underline{max(13, 28)} \\ = \underline{12 + 28} \\ = 40 \end{cases}$$

2.
$$\begin{cases} abs(-12) + \underline{max(13, 28)} \\ = \underline{abs(-12)} + 28 \\ = \underline{12 + 28} \\ = 40 \end{cases}$$

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
```

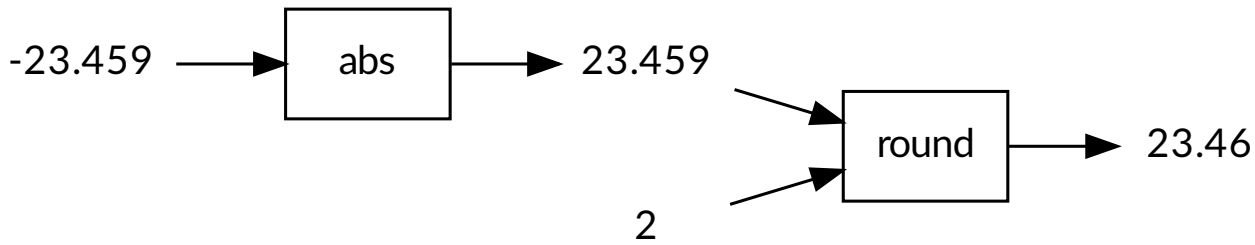
# Composición de operaciones y funciones

- 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**.



# 4.4. Métodos

# 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:

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

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

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

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

  y tener una función y hacer:

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

- 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')
```

# 5. Otros conceptos sobre operaciones

5.1 Sobrecarga de operaciones

5.2 Equivalencia entre formas de operaciones

5.3 Igualdad de operaciones

# 5.1. Sobrecarga de operaciones

# 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**.

# 5.2. Equivalencia entre formas de operaciones

# 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).

# 5.3. Igualdad de operaciones

# 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.

# 6. Operaciones predefinidas

# 6.1. Operadores predefinidos

# 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`<br>`8 % 3` | `1`<br>`2` | Resto de la división |
| `**` | Exponente | `3 ** 4` | `81` | Devuelve $3^4$ |
| `//` | División entera hacia abajo | `4 // 3`<br>`-4 // 3` | `1`<br>`-2` | <br>?? |

# Operadores de cadenas

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

# 6.2. Funciones predefinidas

## 6.2.1 Funciones matemáticas

# Funciones predefinidas

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

# 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)
└┬─ └┬┘
 │   └──── función
 └─ módulo
```

- El punto `.` es un operador que nos permite acceder al interior de estructuras que tienen contenido propio, como los módulos.

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

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

# 6.3. Métodos predefinidos

# 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

# 7. Ejercicios

# 7.1. Ejercicios

# Ejercicios

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)`

# 8. Bibliografía

# 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.