

# 01 – Introduction

Un ordenador hace dos cosas, y sólo dos: realiza cálculos y recuerda los resultados de esos cálculos. Pero hace esas dos cosas extremadamente bien.

---

## Pensamiento computacional

Todo conocimiento puede considerarse declarativo o imperativo.

- **Declarativo:** se compone de afirmaciones de hecho. Por ejemplo, «la raíz cuadrada de  $x$  es un número  $y$  tal que  $y * y = x$  y «es posible viajar en tren de París a Roma». Son afirmaciones de hecho. Por desgracia, no nos dicen nada sobre cómo hallar una raíz cuadrada o cómo viajar en tren de París a Roma.
- **Imperativo:** son conocimientos «prácticos», o recetas para deducir información. Herón de Alejandría fue el primero en documentar una forma de calcular la raíz cuadrada de un número. Su método para hallar la raíz cuadrada de un número, llámese  $x$ , puede resumirse así:

1. Empieza con una suposición,  $g$ .
2. Si  $g * g$  se aproxima lo suficiente a  $x$ , detente y di que  $g$  es la respuesta.
3. En caso contrario, crea una nueva conjetura promediando  $g$  y  $x/g$ , es decir,  $(g + x/g)/2$ .
4. Utilizando esta nueva suposición, a la que volveremos a llamar  $g$ , repite el proceso hasta que  $g * g$  se acerque lo suficiente a  $x$ .

Consideremos la búsqueda de la raíz cuadrada de 25.

1. Establezca  $g$  en algún valor arbitrario, por ejemplo, 3.
2. Decidimos que  $3 * 3 = 9$  no está suficientemente cerca de 25.
3. Establecemos  $g$  en  $(3 + 25/3)/2 = 5,67$
4. Decidimos que  $5,67 * 5,67 = 32,15$  sigue sin acercarse lo suficiente a 25.
5. Fijamos  $g$  en  $(5,67 + 25/5,67)/2 = 5,04$ .
6. Decidimos que  $5,04 * 5,04 = 25,4$  es suficientemente próximo, así que nos detenemos y declaramos que 5,04 es una aproximación adecuada a

la raíz cuadrada de 25.

### Algoritmo

Observe que la descripción del método es una *secuencia de pasos simples*, junto con un *flujo de control* que especifica cuándo ejecutar cada paso. Una descripción de este tipo se denomina **algoritmo**.

El algoritmo que utilizamos para aproximar la raíz cuadrada es un ejemplo de algoritmo "**guess-and-check**" (conjetura y comprobación). Se basa en el hecho de que es fácil comprobar si una conjetura es suficientemente buena o no.

Una definición mas formal de **algoritmo** es: una *lista finita* de instrucciones que describen un conjunto de cálculos que, cuando se ejecutan en un conjunto de entradas, pasan por una secuencia de estados bien definidos y finalmente producen una *salida*.

Las primeras máquinas de computación eran, de hecho, **ordenadores de programa fijo**, lo que significa que estaban diseñados para *resolver un problema matemático específico*, por ejemplo, calcular la trayectoria de un proyectil de artillería.

Uno de los primeros ordenadores (construido en 1941 por Atanasoff y Berry) resolvía sistemas de ecuaciones lineales, pero no podía hacer nada más. La máquina bomba de Alan Turing, desarrollada durante la Segunda Guerra Mundial, fue diseñada para descifrar los códigos Enigma alemanes.

Algunos ordenadores sencillos siguen utilizando este método. Por ejemplo, *una calculadora de cuatro funciones es un ordenador de programa fijo*. Puede realizar operaciones aritméticas básicas, pero no puede utilizarse como procesador de textos ni para ejecutar videojuegos. Para cambiar el programa de una máquina de este tipo, hay que sustituir los circuitos.

El primer ordenador verdaderamente moderno fue el **Manchester Mark 1**. Se distinguía de sus predecesores por ser un **ordenador de programa almacenado**. Un ordenador de este tipo *almacena* (y *manipula*) una secuencia de instrucciones y tiene componentes que ejecutan cualquier instrucción de esa secuencia. El corazón de un ordenador de este tipo es un **intérprete** que puede ejecutar cualquier conjunto legal de instrucciones y, por tanto, puede utilizarse para calcular cualquier cosa que pueda describirse utilizando esas instrucciones. El resultado

del cálculo puede ser incluso una nueva secuencia de instrucciones, que puede ser ejecutada por el ordenador que las generó. En otras palabras, es posible que un ordenador se programe a sí mismo.

*Tanto el programa como los datos que manipula residen en la memoria.* Normalmente, un "programa contador" apunta a un lugar concreto de la memoria, y el cálculo comienza ejecutando la instrucción en ese punto. La mayoría de las veces, el intérprete simplemente pasa a la siguiente instrucción de la secuencia, pero no siempre. En algunos casos, realiza una prueba y, basándose en ella, la ejecución puede saltar a otro punto de la secuencia de instrucciones. Esto se llama **flujo de control**, y es esencial para permitirnos escribir programas que realicen tareas complejas.

A veces se utilizan diagramas para representar el flujo de control. Por convención, utilizamos cajas rectangulares para representar un paso del proceso, un rombo para representar una prueba y flechas para indicar el orden en que se hacen las cosas.

Para crear algoritmos, o secuencias de instrucciones, necesitamos un **lenguaje de programación** con el que describirlas, una forma de dar al ordenador sus órdenes de marcha.

En 1936, el matemático británico *Alan Turing* describió un hipotético dispositivo informático que se le llamo **Máquina Universal de Turing**. La máquina tenía memoria ilimitada en forma de «cinta» en la que se podían escribir ceros y unos, y un puñado de sencillas instrucciones primitivas para mover, leer y escribir en la cinta. La tesis de *Church-Turing* afirma que **si una función es computable, se puede programar una máquina de Turing para que la compute**.

El «**if**» de la tesis *Church-Turing* es importante. No todos los problemas tienen soluciones computacionales. Turing demostró, por ejemplo, que es imposible escribir un programa que tome un programa arbitrario como entrada e imprima **true** si y sólo si el programa de entrada se ejecuta para siempre. Es lo que se conoce como "**halting problem**" (*problema de la detención*).

La tesis de *Church-Turing* conduce directamente a la noción de completitud de Turing. Se dice que **un lenguaje de programación es completo de Turing si puede utilizarse para simular una máquina de Turing universal**. Todos los lenguajes de programación modernos son completos de Turing. En consecuencia, *cualquier cosa que pueda programarse en un lenguaje de programación* (por ejemplo, Python) *puede programarse en cualquier otro lenguaje de programación* (por ejemplo,

Java). Por supuesto, algunas cosas pueden ser más fáciles de programar en un lenguaje concreto, pero todos los lenguajes son fundamentalmente iguales en cuanto a potencia de cálculo.


## Primitivas básicas

Las **primitivas básicas** son **operaciones mínimas y fundamentales** con las que se pueden construir **todos los programas posibles**. Turing demostró que una máquina muy simple, con solo 6 primitivas, podía **simular cualquier cálculo computable**.

### Las 6 primitivas de una Máquina de Turing

1. **left** – mover la cabeza lectora **una posición a la izquierda**
2. **right** – mover la cabeza lectora **una posición a la derecha**
3. **print** – escribir un símbolo en la celda actual
4. **scan** – leer el símbolo en la celda actual
5. **erase** – borrar lo que hay en la celda actual
6. **no-op** – no hacer nada (esperar o continuar sin acción)

#### Nota

 *Con solo estas acciones, la máquina de Turing puede realizar cualquier cálculo que sea teóricamente computable.*

### ¿Qué pasa con los lenguajes reales?

Lenguajes como Python, Java o C tienen:

- **Más primitivas** (por comodidad, no porque puedan hacer “más cosas”)
- **Formas de combinar primitivas**: funciones, estructuras de control (if, loops), clases, etc.
- **Capacidad para definir primitivas nuevas** a partir de otras (por ejemplo, escribir tu propia función `calcular_promedio()`)

Afortunadamente, ningún programador tiene que construir programas a partir de las instrucciones primitivas de Turing. En su lugar, los lenguajes de programación modernos ofrecen un conjunto de primitivas más amplio y cómodo. Sin embargo, la idea fundamental de la programación como el proceso de ensamblar una secuencia de operaciones sigue siendo central.

Sea cual sea el conjunto de primitivas que tenga y los métodos que utilice para ensamblarlas, lo mejor y lo peor de la programación es lo mismo: *el ordenador hará exactamente lo que usted le diga que haga, ni más ni menos*. Esto es bueno porque significa que puedes hacer que el ordenador haga todo tipo de cosas divertidas y útiles. Es malo porque *cuando no hace lo que quieres que haga, no tienes a nadie a quien culpar excepto a ti mismo*.

Hay cientos de lenguajes de programación en el mundo. *No existe el mejor lenguaje*. Diferentes lenguajes son mejores o peores para diferentes tipos de aplicaciones. **MATLAB**, por ejemplo, es un buen lenguaje para *manipular vectores y matrices*. **C** es un buen lenguaje para *escribir programas que controlan redes de datos*. **PHP** es un buen lenguaje para *crear sitios web*. Y **Python** es un excelente lenguaje de *propósito general*.

Cada lenguaje de programación tiene un conjunto de **construcciones primitivas**, una **sintaxis**, una **semántica estática** y una **semántica**. Por analogía con un lenguaje natural, como el inglés, las *construcciones primitivas son palabras*, la *sintaxis describe qué cadenas de palabras constituyen oraciones bien formadas*, la *semántica estática define qué oraciones tienen sentido* y la *semántica define el significado de esas oraciones*. Las construcciones primitivas en Python incluyen literales (por ejemplo, el número `3.2` y la cadena `'abc'`) y operadores infijos (por ejemplo, `+` y `/`).

## Sintaxis

La **sintaxis** de un idioma *define qué cadenas de caracteres y símbolos están bien formadas*. Por ejemplo, en inglés la cadena «`Cat dog boy.`» no es una frase sintácticamente válida, porque la sintaxis inglesa no acepta frases de la forma `<noun> <noun> <noun>`. En Python, la secuencia de primitivas `3.2 + 3.2` está bien formada sintácticamente, pero la secuencia `3.2 3.2` no lo está.

## Semántica estática

La **semántica estática** *define qué cadenas sintácticamente válidas tienen un significado*. Consideremos, por ejemplo, las cadenas «`He run quickly`» y «`I runs quickly`». Cada una tiene la forma `<pronombre> <verbo regular> <adverbio>`, que es una secuencia sintácticamente aceptable. Sin embargo, ninguna de las dos es válida en inglés, debido a la regla bastante peculiar de que para un verbo regular cuando el sujeto de una

frase es primera o segunda persona, el verbo no termina con «s», pero cuando el sujeto es tercera persona del singular, sí. Son ejemplos de errores semánticos estáticos.

## Semántica

La **semántica** de una lengua *asocia un significado a cada cadena de símbolos sintácticamente correcta que no tiene errores semánticos estáticos*. En las lenguas naturales, la semántica de una frase puede ser ambigua. Por ejemplo, la frase «No puedo elogiar demasiado a este estudiante» puede ser halagadora o condenatoria. Los lenguajes de programación están diseñados para que cada programa legal tenga exactamente un significado.

---

## Idea clave: Los programas no son diferentes de otros tipos de datos

Un programa (código que se ejecuta) puede ser tratado, procesado o modificado por otro programa, de la misma forma en que se manipulan otros tipos de datos (como números, cadenas de texto, listas, etc.).

En otras palabras, un programa puede ser visto como datos.

1. **Podemos escribir programas que generen otros programas.**  
Por ejemplo, un **compilador** toma código fuente como entrada (un programa) y genera otro programa como salida (código máquina).
2. **Podemos analizar y transformar programas con programas.**  
Esto es lo que hacen los **intérpretes**, los **depuradores**, los **editores de código**, o incluso funciones como `eval()` en Python.
3. **Nos permite crear software más dinámico y flexible.**  
Por ejemplo, una inteligencia artificial que genera código automáticamente está usando esta idea.

Supongamos que tienes este código:

```
1 code = "print(2 + 3)"
2 eval(code)
```

- Aquí `code` es una **cadena de texto**, o sea, **datos**.
- Pero `eval(code)` ejecuta ese texto como si fuera un programa.

- Así, estás tratando un programa como si fuera datos. ¡Y luego lo ejecutas!

## Relación con la Máquina de Turing

La **máquina de Turing universal** también se basa en esta idea: Puede leer una "cinta" que contiene la descripción de otro programa. Es decir, trata **el código de un programa como entrada de otro programa**.

### ✓ ¿Por qué es importante?

Porque **es la base de muchos conceptos avanzados**:

- **Compiladores e intérpretes**
- **Lenguajes funcionales** (que permiten funciones como datos)
- **Meta-programación** (programas que escriben programas)
- **Máquinas virtuales**
- **Teoremas de incompletitud y problemas no computables**

---

Aunque los **errores sintácticos son el tipo de error más común** (especialmente para quienes están aprendiendo un nuevo lenguaje de programación), **son el tipo de error menos peligroso**. Todos los lenguajes de programación serios detectan todos los errores sintácticos y no permiten que los usuarios ejecuten un programa con un solo error sintáctico. Además, en la mayoría de los casos el sistema del lenguaje da una indicación lo suficientemente clara de la localización del error como para que el programador sea capaz de solucionarlo sin pensárselo demasiado.

Identificar y resolver errores semánticos estáticos es más complejo. Algunos lenguajes de programación, como Java, realizan muchas comprobaciones semánticas estáticas antes de permitir la ejecución de un programa. Otros, como C y Python (por desgracia), hacen relativamente menos comprobaciones semánticas estáticas antes de que se ejecute un programa. Python realiza una cantidad considerable de comprobaciones semánticas mientras se ejecuta un programa.

*Si un programa no tiene errores sintácticos ni errores semánticos estáticos, tiene un significado, es decir, tiene semántica.* Por supuesto, puede que no tenga la semántica que su creador pretendía.



Cuando un programa significa algo distinto de lo que su creador cree que significa, pueden ocurrir cosas malas.

## ¿Qué puede ocurrir si el programa tiene un error y se comporta de forma no deseada?

- Puede **bloquearse**, es decir, dejar de ejecutarse y producir una indicación obvia de que lo ha hecho. En un sistema informático bien diseñado, cuando un programa se bloquea, no daña el sistema en su conjunto. Por desgracia, algunos sistemas informáticos muy populares no tienen esta agradable propiedad. Casi todo el mundo que utiliza un ordenador personal ha ejecutado alguna vez un programa que ha conseguido que sea necesario reiniciar todo el sistema.
- Puede que  **siga ejecutándose, y ejecutándose, y ejecutándose, y nunca se detenga**. Si no tienes ni idea de cuánto tiempo se supone que tarda el programa en hacer su trabajo, esta situación puede ser difícil de reconocer.
- Es posible que se **ejecute hasta el final y produzca una respuesta que puede ser correcta o no**.

Cada uno de estos resultados es malo, pero el último es sin duda el peor. Cuando un programa parece estar haciendo lo correcto pero no lo está haciendo, pueden ocurrir cosas malas: se pueden perder fortunas, los pacientes pueden recibir dosis mortales de radioterapia, los aviones pueden estrellarse.

### Ejercicio manual

Los ordenadores pueden ser muy literales. Si no les dices exactamente lo que quieres que hagan, es probable que se equivoquen. Intenta escribir un algoritmo para conducir entre dos destinos. Escríbelo como lo harías para una persona e imagina qué pasaría si esa persona fuera tan estúpida como un ordenador y ejecutara el algoritmo exactamente como está escrito. (Para una divertida ilustración de esto, echa un vistazo al vídeo <https://www.youtube.com/watch?v=FN2RM-CHkUI&t=24s>.)

### Mi aproximación al ejercicio:

- Subir a un auto, encender el auto, poner primera, apretar el embrague, encender el motor, soltar el embrague apretando apenas el



acelerador y manejar correctamente hasta el destino.  
Probablemente ocurrirían muchos errores no deseados.

---

## Introducción a Python

Aunque cada lenguaje de programación es diferente (aunque no tanto como sus diseñadores nos quieren hacer creer), pueden relacionarse a lo largo de algunas dimensiones.

### Nivel de abstracción

**Bajo nivel** vs. **alto nivel** se refiere a si programamos utilizando instrucciones y objetos de datos al nivel de la máquina (por ejemplo, mover 64 bits de datos de esta ubicación a aquella) o si programamos utilizando operaciones más abstractas (por ejemplo, desplegar un menú en la pantalla) que han sido proporcionadas por el diseñador del lenguaje.

### Lenguajes de Propósito General vs. Lenguajes Específicos de Dominio

Esto se refiere a si las operaciones primitivas del lenguaje de programación son ampliamente aplicables o están ajustadas específicamente a un dominio. Por ejemplo, **SQL está diseñado para extraer información de bases de datos relacionales**, pero no querriás usarlo para construir un sistema operativo.

### Interprete vs Compilador

Esto se refiere a si la secuencia de instrucciones escrita por el programador, llamada **código fuente**, se ejecuta **directamente** (mediante un **intérprete**) o si primero se **convierte** (mediante un **compilador**) en una secuencia de operaciones primitivas a nivel de máquina.

#### Nota

En los primeros tiempos de los ordenadores, la gente tenía que escribir el código fuente en un lenguaje cercano al código máquina

que pudiera ser interpretado directamente por el hardware del ordenador

Ambos enfoques tienen sus ventajas. A menudo *es más fácil depurar programas escritos en lenguajes diseñados para ser interpretados*, porque el intérprete puede producir mensajes de error fáciles de relacionar con el código fuente. Los *lenguajes compilados suelen producir programas que se ejecutan más rápidamente y ocupan menos espacio*.

---

- **Anaconda** es una distribución de Python que incluye el propio lenguaje Python, un conjunto de paquetes especializados, un editor de código (Spyder) y un intérprete interactivo (IPython).
- **Spyder** (Scientific Python Development Environment) es un entorno de desarrollo integrado (IDE) orientado al uso científico de Python.
- Proporciona un espacio para **escribir, ejecutar y depurar código** de manera eficiente.

**Python** es un lenguaje de programación de **propósito general** que puede utilizarse eficazmente para construir casi cualquier tipo de programa que no necesite acceso directo al hardware del ordenador. Python no es óptimo para programas que tengan restricciones de alta fiabilidad (debido a su *débil comprobación semántica estática*) o que sean contruidos y mantenidos por muchas personas o durante un largo periodo de tiempo (de nuevo debido a la débil comprobación semántica estática).

Es un lenguaje relativamente **sencillo y fácil de aprender**. Como Python está diseñado para ser **interpretado**, puede proporcionar el tipo de retroalimentación en tiempo de ejecución que es especialmente útil para los programadores principiantes. Existe un gran número, cada vez mayor, de bibliotecas gratuitas que se conectan a Python y proporcionan funciones ampliadas de gran utilidad.

Estamos listos para introducir algunos de los elementos básicos de Python. Estos son comunes a casi todos los lenguajes de programación en concepto, aunque no en detalle.

Python es un lenguaje vivo. Desde su introducción por **Guido von Rossum** en 1990, ha experimentado muchos cambios. Durante la primera década de su vida, Python fue un lenguaje poco conocido y poco utilizado. Eso cambió con la llegada de Python 2.0 en 2000. Además de incorporar

importantes mejoras al propio lenguaje, marcó un cambio en la trayectoria evolutiva del mismo. Muchos grupos empezaron a desarrollar bibliotecas que interactuaban a la perfección con Python, y el apoyo y desarrollo continuos del ecosistema Python se convirtieron en una actividad basada en la comunidad.

**Python 3.0** se publicó a finales de 2008. Esta versión de Python eliminó muchas de las incoherencias en el diseño de Python 2. Sin embargo, Python 3 no es compatible con Python 2. Sin embargo, Python 3 no es compatible con versiones anteriores. Esto significa que la mayoría de los programas y bibliotecas escritos para versiones anteriores de Python no pueden ejecutarse utilizando implementaciones de Python 3.

En la actualidad, todas las bibliotecas importantes de Python de dominio público han sido portadas a Python 3. Hoy en día, no hay ninguna razón para utilizar Python 2.

---

## Elementos básicos de Python

Un **programa** Python, a veces llamado **script**, es una *secuencia de definiciones y comandos*. El intérprete de Python en el shell evalúa las definiciones y ejecuta los comandos.

Un **comando**, a menudo llamado sentencia, ordena al intérprete que haga algo. Por ejemplo, la sentencia `print('¡River manda!')` ordena al intérprete que llame a la función `print`, que muestra la cadena `River manda!` en la ventana asociada al intérprete de comandos.

La secuencia de comandos sería:

```
1 print('River manda!')
2 print('Pero no en el Monumental!')
3 print('River manda,', 'Pero no en el Monumental!')
```

El interprete producirá la siguiente salida:

```
1 River manda!
2 Pero no en el Monumental!
3 River manda, Pero no en el Monumental!
```

Observe que se han pasado dos valores a `print` en la tercera sentencia. La función `print` toma un número variable de argumentos

separados por comas y los imprime, separados por un carácter de espacio, en el orden en que aparecen.

---

## Objetos, expresiones y tipos numéricos

Los **objetos** son el núcleo de las cosas que manipulan los programas Python. Cada objeto tiene un **tipo** que *define lo que los programas pueden hacer con ese objeto*. Los tipos *pueden ser escalares o no escalares*. Los **objetos escalares son indivisibles**. Piensa en ellos como los átomos del lenguaje. Los **objetos no escalares**, como las cadenas, **tienen una estructura interna**.

Muchos tipos de objetos pueden denotarse mediante **literales** en el texto de un programa. Por ejemplo, el texto `2` es un literal que representa un número y el texto `'abc'` es un literal que representa una cadena.

Python tiene cuatro tipos de objetos escalares:

- **int** se utiliza para representar **números enteros**. Los literales de tipo **int** se escriben de la forma en que se suelen denominar los números enteros (por ejemplo, `-3` o `5` o `10002`).
- **float** se utiliza para representar **números reales**. Los literales de tipo **float** siempre incluyen un punto decimal (por ejemplo, `3.0` o `3.17` o `-28.72`). (También es posible escribir literales de tipo **float** utilizando notación científica. Por ejemplo, el literal `1.6E3` significa  $1,6 * 10^3$ , es decir, es lo mismo que `1600.0`). Quizá se pregunte por qué este tipo no se llama real. En el ordenador, los valores de tipo **float** se almacenan como números de coma flotante. Esta representación, que utilizan todos los lenguajes de programación modernos, tiene muchas ventajas. Sin embargo, en algunas situaciones hace que la aritmética de coma flotante se comporte de forma ligeramente diferente a la aritmética sobre números reales.
- **bool** se utiliza para representar los valores **booleanos** `True` y `False`.
- **None** es un tipo con un **único valor**.

Los **objetos** y los **operadores** pueden combinarse para formar **expresiones**, cada una de las cuales se evalúa a un objeto de algún tipo. Éste se denomina **valor de la expresión**. Por ejemplo, la expresión `3 + 2` denota el objeto `5` de tipo **int**, y la expresión `3.0 + 2.0` denota el objeto `5.0` de tipo **float**.

El operador `==` se utiliza para comprobar si dos expresiones se evalúan con el mismo valor, y el operador `!=` se utiliza para comprobar si dos expresiones se evalúan con valores diferentes. Un solo `=` significa algo muy distinto.

La función incorporada de Python `type()` se puede utilizar para averiguar el tipo de un objeto:

```
1 type(3)
2 # int
3 type(3.0)
4 # float
```

Los operadores aritméticos tienen la precedencia habitual. Por ejemplo, `*` se vincula más estrechamente que `+`, por lo que la expresión `x + y * 2` se evalúa multiplicando primero `y` por `2` y luego sumando el resultado a `x`. El orden de evaluación puede cambiarse utilizando **paréntesis** para agrupar subexpresiones, por ejemplo, `(x + y) * 2` suma primero `x` e `y`, y luego multiplica el resultado por `2`.

- `i + j` es la suma de `i` y `j`. Si `i` y `j` son ambos de tipo `int`, el resultado es un `int`. Si alguno es un `float`, el resultado es un `float`.
- `i - j` es `i` menos `j`. Si `i` y `j` son ambos de tipo `int`, el resultado es un `int`. Si alguno es un `float`, el resultado es un `float`.
- `i * j` es el producto de `i` y `j`. Si `i` y `j` son ambos de tipo `int`, el resultado es un `int`. Si alguno es un `float`, el resultado es un `float`.
- `i // j` es la división entera. Por ejemplo, el valor de `6 // 2` es el `int 3` y el valor de `6 // 4` es el `int 1`. El valor es `1` porque la división entera devuelve el cociente e ignora el residuo. Si `j == 0`, ocurre un error.
- `i / j` es `i` dividido por `j`. El operador `/` realiza una división de punto flotante. Por ejemplo, el valor de `6 / 4` es `1.5`. Si `j == 0`, ocurre un error.
- `i % j` es el residuo cuando el `int i` se divide entre el `int j`. Se pronuncia típicamente “`i mod j`”, que es la abreviación de “`i módulo j`”.
- `{python}i j` es `i` elevado a la potencia `j`. Si `i` y `j` son ambos de tipo `int`, el resultado es un `int`. Si alguno es un `float`, el resultado es un `float`.

Los operadores primitivos del tipo `bool` son `and`, `or` y `not`:

- `a and b` es Verdadero si tanto a como b son Verdaderos, y Falso en caso contrario.
  - `a or b` es Verdadero si al menos uno de los dos es Verdadero, y Falso en caso contrario.
  - `not a` es Verdadero si a es Falso, y Falso si a es Verdadero.
- 

## Variables y asignaciones

Las variables permiten asociar nombres a los objetos. Considere el código:

```
1 pi = 3
2 radius = 11
3 area = pi * (radius**2)
4 radius = 14
```

El código vincula primero los nombres `pi` y `radius` a diferentes objetos de tipo `int`. A continuación, vincula el nombre `area` a un tercer objeto de tipo `int`.

Si el programa entonces ejecuta `radius = 14`, el nombre `radius` es rebotado a un objeto diferente de tipo `int`. Observe que esta asignación no tiene ningún efecto sobre el valor al que está ligada `area`. Sigue estando ligado al objeto denotado por la expresión `3 * (11**2)`.

**Una variable es sólo un nombre, nada más.** Recuérdalo, es importante. Una sentencia de asignación asocia el nombre a la izquierda del símbolo `=` con el objeto denotado por la expresión a la derecha del símbolo `=`. Recuerde esto también: *un objeto puede tener uno, más de uno o ningún nombre asociado a él.*

Quizá no deberíamos haber dicho «*una variable es sólo un nombre*». A pesar de lo que dijo Juliet, los nombres importan. Los lenguajes de programación nos permiten describir cálculos para que los ordenadores puedan ejecutarlos. Esto no significa que sólo los ordenadores lean programas.

Los programadores experimentados confirmarán que pasan mucho tiempo leyendo programas para intentar entender por qué se comportan como lo hacen. Por tanto, *es de vital importancia escribir programas que sean fáciles de leer*. La elección adecuada de los nombres de las variables desempeña un papel importante en la legibilidad.

Considere los dos fragmentos de código:

```
1 a = 3.14159
2 b = 11.2
3 c = a*(b**2)
```

```
1 pi = 3.14159
2 diameter = 11.2
3 area = pi*(diameter**2)
```

En lo que respecta a Python, los fragmentos de código no son diferentes. **Cuando se ejecuten, harán lo mismo.** Para un lector humano, sin embargo, son bastante diferentes. Cuando leemos el primer fragmento, no hay ninguna razón a priori para sospechar que algo va mal. Sin embargo, un rápido vistazo al segundo código debería hacernos sospechar que algo va mal. O bien la variable debería haberse llamado `radius` en lugar de `diameter`, o bien el `diameter` debería haberse dividido por `2.0` en el cálculo del área.

Los nombres de variables pueden contener letras mayúsculas y minúsculas, dígitos (aunque no pueden empezar por un dígito) y el carácter especial `_` (guion bajo). Los nombres de variables en Python distinguen entre mayúsculas y minúsculas, por ejemplo, `Romeo` y `romeo` son nombres diferentes. Por último, hay algunas **palabras reservadas** (a veces llamadas palabras clave) en Python que tienen significados incorporados y no pueden usarse como nombres de variables. Las diferentes versiones de Python tienen listas ligeramente diferentes de palabras reservadas. Las palabras reservadas en Python 3.8 son:

```
1 and break elif for in not True as class else from is or try assert
  continue except global lambda pass while async def False if nonlocal raise
  with await del finally import None return yield
```

Otra buena forma de mejorar la legibilidad del código es añadir comentarios. El texto que sigue al símbolo `#` no es interpretado por Python. Por ejemplo, podríamos escribir

```
1 side = 1 #length of sides of a unit square
2 radius = 1 #radius of a unit circle
3 #subtract area of unit circle from area of unit square
4 area_circle = pi*radius**2
5 area_square = side*side
6 difference = area_square - area_circle
```

Python permite la asignación múltiple. La sentencia `x, y = 2, 3` asocia `x` a `2` e `y` a `3`. Todas las expresiones de la parte derecha de la asignación **se evalúan antes de modificar cualquier asociación.**



Esto es conveniente ya que le permite utilizar la asignación múltiple para intercambiar los enlaces de dos variables.

Por ejemplo, el código:

```
1 x, y = 2, 3
2 x, y = y, x
3 print('x =', x)
4 print('y =', y)
```

Imprimirá:

```
1 x = 3
2 y = 2
```

---

## Ejercicio manual de la lectura

Asume que ya tienes definidas 3 variables: `a`, `b` y `c`. Crea una variable llamada `total` que sume `a` y `b` y luego multiplique el resultado por `c`. Incluye una última línea en tu código para imprimir el valor: `print(total)`

## Mi aproximación

```
1 total = (a + b) * c
2 print(total)
```

Asumiendo que ya tengo definidas las 3 variables, puedo crear una variable llamada `total` donde realizare el calculo exigido y luego lo mostrare en la consola mediante la función `print()`.

---

## Términos

- **Conocimiento declarativo:** Tipo de conocimiento que describe **qué** se quiere lograr sin especificar **cómo** hacerlo. Se enfoca en los hechos, reglas o relaciones. Ej.: Consultas en SQL, lógica proposicional.
- **Conocimiento imperativo:** Describe **cómo** lograr algo paso a paso. Es procedimental, indica instrucciones explícitas para alcanzar un objetivo. Ej.: Programación en C, Python.

- **Algoritmo:** Conjunto finito y ordenado de pasos o instrucciones que resuelven un problema o realizan una tarea. Debe ser preciso, finito y efectivo.
- **Computación:** Proceso de realizar cálculos o resolver problemas mediante algoritmos, ya sea de forma manual, mecánica o automática (computadoras).
- **Computadoras de programas fijos:** Computadoras cuyo comportamiento está definido por **circuitos físicos**. No se pueden reprogramar fácilmente. Ej.: Calculadoras básicas, dispositivos electrónicos específicos.
- **Computadoras de programas almacenados:** Computadoras que pueden **almacenar instrucciones (programas)** en su memoria, permitiendo cambiar el comportamiento sin modificar el hardware. Ej.: Todas las computadoras modernas (modelo de von Neumann).
- **Interprete:** Programa que **ejecuta código fuente línea por línea**, sin necesidad de compilarlo previamente. Ej.: Intérprete de Python.
- **Contador de programa:** Registro de la CPU que contiene la dirección de memoria de la **siguiente instrucción a ejecutar**.
- **Flujo de control:** Orden en que se ejecutan las instrucciones en un programa. Puede ser secuencial, condicional (if), repetitivo (while/for), o por funciones/llamadas.
- **Diagrama de flujo:** Representación gráfica del flujo de control de un algoritmo o programa, usando símbolos estandarizados (óvalos, flechas, rombos, etc.).
- **Lenguaje de programación:** Lenguaje formal diseñado para **describir algoritmos y estructuras de datos**, que puede ser entendido y ejecutado por una computadora.
- **Maquina de Turing universal:** Modelo teórico de computación que puede **simular cualquier otra máquina de Turing**, es decir, puede ejecutar cualquier algoritmo describable.
- **Tesis Church-Turing:** Hipótesis que establece que **todo problema que puede resolverse mediante un procedimiento efectivo (algoritmo)** puede resolverse con una máquina de Turing.
- **Problema de detención:** Problema que consiste en determinar, dado un programa y una entrada, **si el programa terminará o se ejecutará para siempre**. Se ha demostrado que **no es resoluble** por ningún algoritmo general.
- **Completitud de Turing:** Propiedad de un lenguaje de programación o sistema computacional que indica que puede **simular cualquier máquina de Turing**, y por tanto, puede realizar cualquier cálculo computable.

- **Literales**: Valores constantes **escritos directamente** en el código fuente. Ej.: `5`, `3.14`, `'hola'`, `True`.
- **Operadores infijos**: Operadores que se colocan **entre dos operandos**. Ej.: `a + b`, `x * y`.
- **Sintaxis**: Conjunto de reglas que define **cómo deben escribirse correctamente las expresiones** en un lenguaje de programación.
- **Semántica estática**: Propiedades del programa que se pueden verificar **en tiempo de compilación** sin ejecutarlo. Ej.: Tipado, declaraciones previas, alcances.
- **Semántica**: Significado de los elementos del lenguaje. Describe **qué efecto produce la ejecución de una instrucción o programa**.