

# Introducción a la programación

---

## Capítulo 1: Introducción a la programación

---

### Objetivos

- Entender en qué consiste la programación.
- Conocer la definición del algoritmo y su importancia dentro de la programación.
- Conocer las entidades que forman parte de un diagrama de flujo.
- Conocer la relación entre pseudocódigo y diagramas de flujo.
- Conocer la importancia de la lógica independiente del lenguaje.
- Conocer la importancia del desarrollo del pensamiento lógico.

### ¿Qué es programar?

Cuando hablamos de programar, nos imaginamos un hacker sentado frente a un computador escribiendo código en una pantalla negra con letras verdes.

### Programar es más que escribir código

La mayor parte del trabajo de un programador consiste en generar soluciones a problemas que surgen dentro de una organización. Para esto, el programador busca descomponer el problema en varias partes y posteriormente implementar estas soluciones.

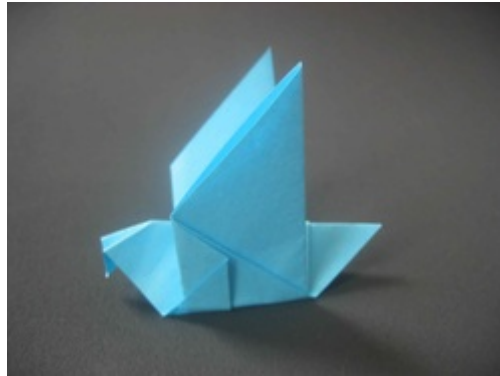
Las soluciones propuestas por un programador se conocen como algoritmos, que se expresan en código. El objetivo de un programador es desarrollar e implementar soluciones mediante los algoritmos. El código es simplemente una de las formas en que se materializa la solución.

### ¿Qué es un algoritmo?

Un algoritmo es una serie de pasos finitos para resolver un problema.

## Ejemplo de algoritmo

Por ejemplo, si deseamos ensamblar un mueble es necesario seguir todos los pasos del manual de manera secuencial. Otro ejemplo típico es una receta de cocina, o crear un origami a partir de un trozo de papel.



## Pasos para resolver un problema

Para poder indicar cómo construir la figura a otra persona, tenemos que indicar claramente todos los pasos que debe hacer de manera secuencial. El omitir un paso afecta el resultado. Para lograr el pájaro de arriba, necesitamos seguir los pasos detallados

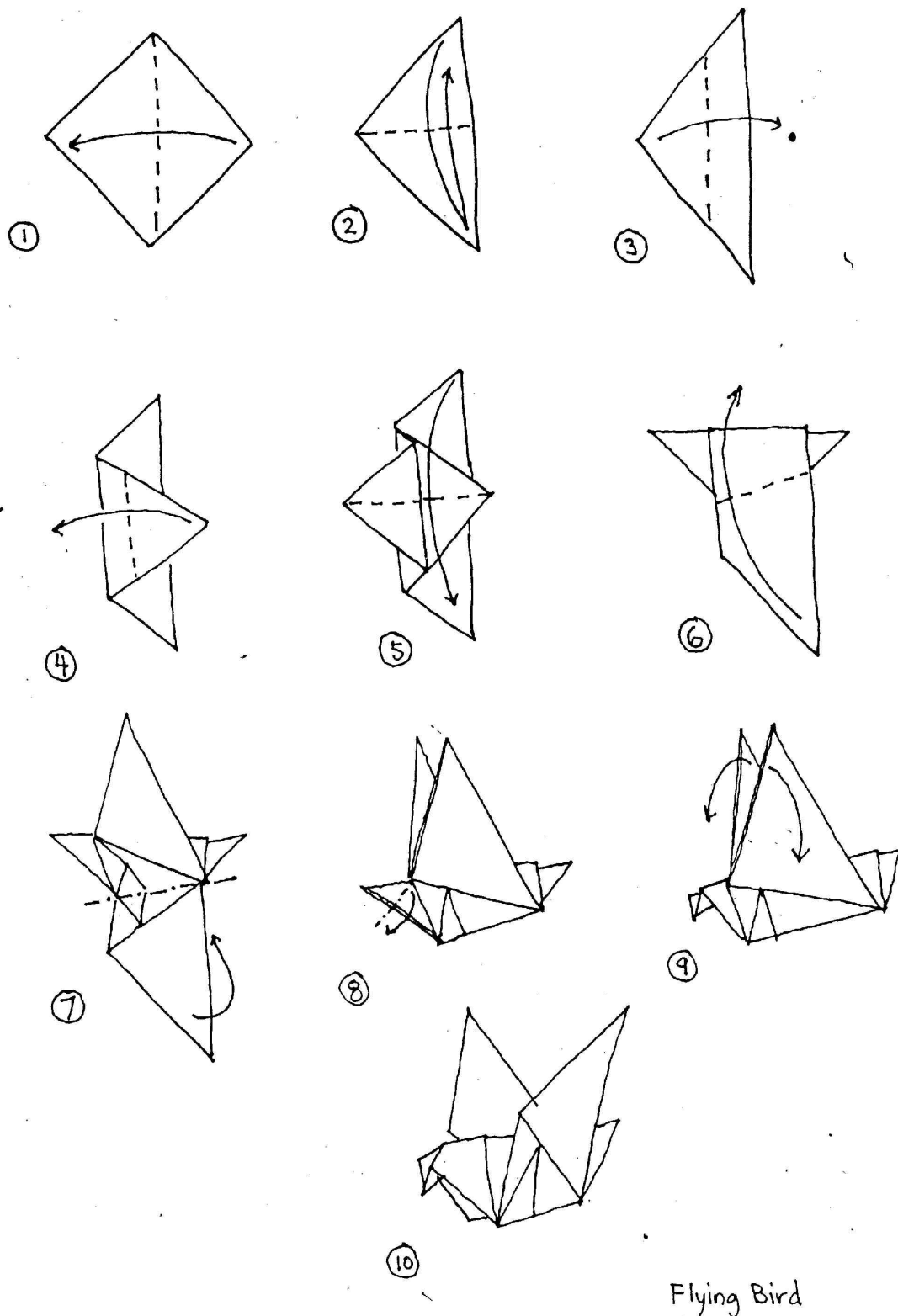


Diagrama autoría de <http://www.laurenstringer.com>.

## Escribiendo un algoritmo

Cuando generamos un algoritmo en el computador, debemos especificar paso a paso qué es lo que el computador debe realizar. Esto es similar a cuando deseamos hacer un origami, debemos seguir pasos.

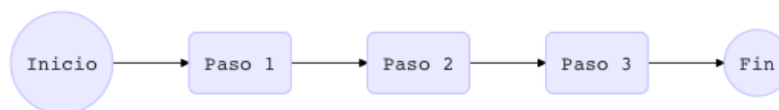
## Formas de escribir un algoritmo

Existen distintas formas de implementar un algoritmo:

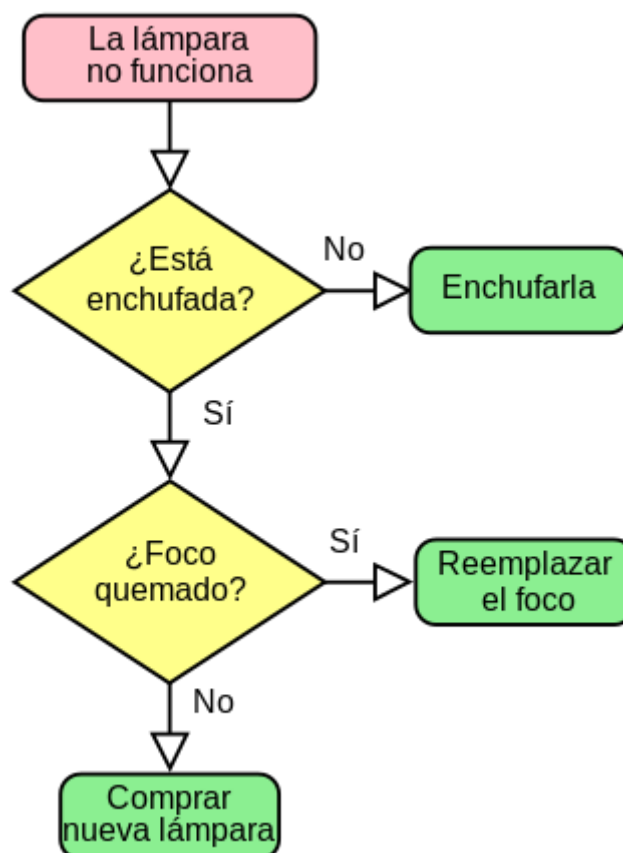
- Representarlo mediante un diagrama de flujo.
- Escribirlo en pseudocódigo.
- Escribirlo directamente en un lenguaje de programación.

## Diagrama de flujo

El diagrama de flujo es una representación gráfica de los pasos detallados en un algoritmo, éste ayuda a visualizarlo y reducir su complejidad.



Si queremos arreglar una lámpara, nuestro proceso algorítmico conlleva los siguientes pasos.

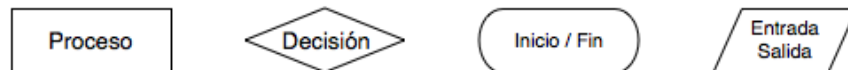


En algunos diagramas como el anterior el fin es implícito. Al llegar al último paso se supone que el algoritmo concluye. Esta distinción es necesaria para destacar que existen procesos donde no hay fines explícitamente declarados.

## Símbolos de un diagrama de flujo

Cada uno de los símbolos en un diagrama de flujo representa alguna acción a realizar en el algoritmo.

- **Inicio y fin del programa:** Por defecto todo algoritmo debe tener una serie de elementos finitos. Éstos se declaran mediante el inicio y el fin del programa.
- **Datos de entrada y salida:** Cada paso que se genera dentro del algoritmo implica tomar algún dato y devolver otro dato.
- **Procesos (la instrucción que le damos a la máquina):** Definición de los pasos a seguir en un algoritmo.
- **Decisiones:** Eventualmente el flujo lógico podrá encausar los siguientes pasos en base a la resolución de una decisión.



## Los diagramas de flujo son importantes para un programador

Existen dos habilidades importantes que tendremos que aprender:

1. Crear diagramas de flujo.
- Implementar código a partir de un diagrama de flujo.

## Pseudocódigo

Otra forma de representar un algoritmo es mediante una serie de frases que describen el flujo:

```
Algoritmo Suma
  Leer valor1
  Leer Valor2
  Mostrar valor1 + valor2
FinAlgoritmo
```

En pseudocódigo se utiliza la instrucción `leer` para especificar que el usuario tiene que ingresar un valor y `mostrar` para imprimir el valor en la pantalla.

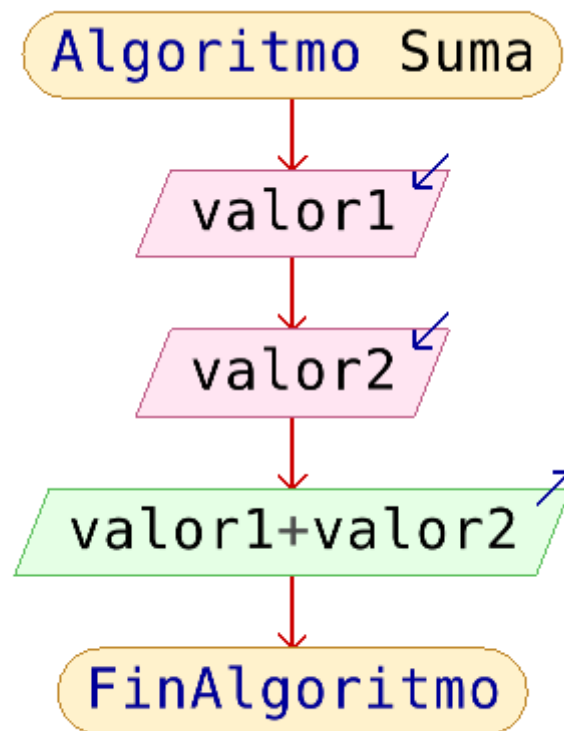
## Escribiendo pseudocódigo

Si bien la especificación del pseudocódigo no es formal, existen algunos programas que nos permiten escribir pseudocódigo y ejecutarlo, como **PSEINT**.

Se puede experimentar con pseudocódigo descargando el programa de forma gratuita en la página oficial <http://pseint.sourceforge.net/>. Algunas capturas de pantalla fueron obtenidas utilizando este programa.

## De pseudocódigo a diagrama de flujo

Transformar el pseudocódigo a un diagrama de flujo es sencillo y muy lineal, solo hay que reemplazar las instrucciones específicas por símbolos correspondientes y poner atención hacia dónde apuntan las flechas.



Todo diagrama de flujo puede transformarse a pseudocódigo y viceversa, así como a un lenguaje de programación, aunque esto no siempre será tan directo.

## Ventajas del pseudocódigo

El pseudocódigo permite pensar en términos independientes al lenguaje de programación y permite concentrarnos en describir lo que estamos tratando de hacer y los pasos necesarios en lugar de cómo lograrlo.

Para programar aprenderemos que es vital saber cómo comunicarle al computador la secuencia de pasos a seguir. Escribir pseudocódigo y diagramas de flujo ayuda a reforzar esta habilidad.

## ¿Por qué es importante que sepamos esto?

Es importante porque los algoritmos y diagramas de flujo son **independientes del lenguaje**, es decir, podemos implementar un mismo algoritmo en diferentes lenguajes de programación.

Un algoritmo corresponde a una solución lógica y pone a prueba nuestro desarrollo del pensamiento lógico.

## Enfrentándose a un problema

El desarrollo del pensamiento lógico es una habilidad **imprescindible** al momento de aprender a programar y está directamente relacionado con nuestra manera de solucionar problemas. Antes de escribir código, debemos abstraernos del código y pensar en el problema en los siguientes pasos:

1. Analizar el problema
2. Descomponer el problema en partes
3. Resolver el problema

Cuanto más desarrollemos nuestro pensamiento lógico, más rápido obtendremos soluciones a problemas cotidianos en programación y nuestros programas harán más a menudo lo que esperamos.

El pensamiento lógico y nuestra capacidad de abstracción son habilidades que desarrollaremos a lo largo del módulo.

# Capítulo 2: Introducción a Python

---

## Objetivos

- Conocer Python.
- Conocer la posición de Python dentro de los lenguajes más conocidos.
- Conocer las ventajas de Python por sobre otros lenguajes.
- Conocer el ecosistema de empresas que utilizan Python.
- Conocer las distintas versiones de Python y cuál utilizaremos en el curso.

## Python

Python es un lenguaje de programación muy flexible y potente. Fue creado por Guido Van Rossum en 1991, con su primera versión 1.0 lanzada en 1994. Dado la claridad y deliberada simpleza de su sintaxis, Python se ha transformado en uno de los lenguajes con mayores tasas de adopción y demanda tanto en la industria.

## Python permite:

- Construir de forma sencilla aplicaciones web con manejos de bases de datos.
- Hacer análisis de datos y visualización de éstos.
- Realizar web-scraping (Captura de datos de una página web).
- Crear videojuegos.
- Crear aplicaciones de escritorio.

## ¿Es Python una buena elección para comenzar a programar?

La respuesta es **sí**.

La velocidad de la tasa de adopción se explica por varios factores directamente relacionados con el diseño de Python. Dada su simpleza sintáctica y similitud al inglés, ha logrado convencer a expertos de otras áreas que no sea programación (llámese matemáticas, lingüística, biología) a programar y diseñar rutinas para agilizar su trabajo. Dado el énfasis en la comunidad de Pythonistas, la cantidad de librerías y contribuciones lo han posicionado como una excelente primera alternativa para resolver tareas.

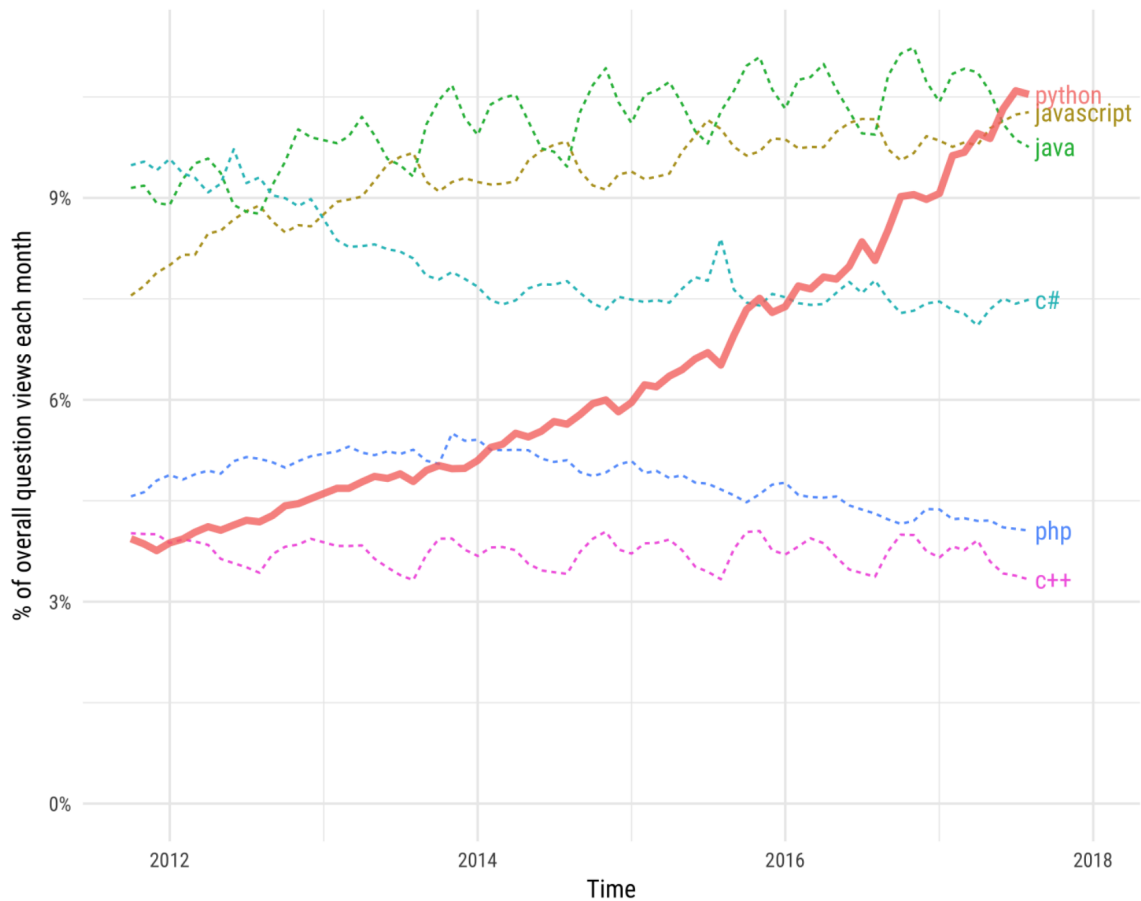
## ¿Qué tan relevante es Python en la industria?

En los últimos años Python se ha caracterizado por ser el lenguaje de programación con una mayor tasa de adopción en comparación a sus principales competidores. Esto acorde a resultados de una encuesta de StackOverflow, uno de los principales foros de programación a nivel mundial, quienes procesaron los datos de sus usuarios.



## Growth of major programming languages

Based on Stack Overflow question views in World Bank high-income countries



Existen dos grandes áreas donde Python es uno de los principales competidores:

- **Desarrollo Web:** Páginas como YouTube, Instagram y Google implementan Python en sus servicios. El lenguaje ofrece librerías como Django y Flask que permiten desarrollar servicios webs complejos dentro de un marco de trabajo sencillo.
- **Ciencia de Datos y Aprendizaje de Máquinas:** Dada la deliberada simpleza sintáctica de Python, existe un gran desarrollo de librerías de análisis y preprocesamiento de datos por parte de la academia e industria para agilizar las rutinas de análisis. Librerías como Scikit-Learn y TensorFlow dominan la implementación de modelos predictivos en distintas áreas.

## Python en Acción

¿Qué crees que hace la siguiente expresión?

```
for i in range(3):  
    print("hip hip hooray!")
```

```
hip hip hooray!  
hip hip hooray!  
hip hip hooray!
```

¿Cuál será el resultado de la siguiente expresión?

```
sum([1, 2, 3, 4])
```

```
10
```

# Capítulo 3: Instalando Python

---

## Objetivos

- Revisar el problema de versiones de Python.
- Conocer un administrador de versiones de Python.
- Instalar un administrador de versiones de Python.

## Python y sus múltiples versiones

En este módulo trabajaremos con la versión 3.7 de Python, la última gran versión de Python que se liberó al público. Una de las características más importantes de esta versión es que **no es compatible con versiones anteriores** como la 2.x. De esta manera es casi imposible correr código desarrollado en Python 2 en Python 3, sin tener que modificar su contenido.

## ¿Cómo podemos manejar distintas versiones de Python?

Una alternativa es desarrollar **ambientes virtuales**, que funcionan como un contenedor aislado donde se puede declarar una versión específica de Python y librerías específicas. Mediante éstos nos aseguramos que cualquier problema existente en el ambiente virtual no afecte a la máquina en su totalidad.

Existen varios administradores de versiones como `pyenv`, `virtualenv` y `anaconda`. Para efectos prácticos nosotros trabajaremos con `anaconda`, un administrador orientado al trabajo en ciencia de datos.

## Instalando Anaconda

La instalación de Anaconda es fácil, sólo debemos ingresar a la página <https://www.anaconda.com/download/> y seguir las instrucciones de instalador gráfico.

# Capítulo 4: Utilizando Python

---

## Objetivos

- Conocer y utilizar Python desde la consola para ejecutar código.
- Conocer las limitantes de Python para la ejecución de código.
- Ejecutar scripts de Python desde la consola.
- Conocer y utilizar iPython.
- Leer la salida de un script ejecutado.

## Formas de trabajar con Python

Existen variadas formas de utilizar Python:

1. La primera es mediante el ambiente REPL (Read, Evaluate, Print Loop), que permite escribir código directamente e ir probando los resultados de forma inmediata lo cual es muy útil para probar código, pero no permite escribir un programa completo.
2. La segunda forma y la que se utiliza tradicionalmente es escribir una serie de instrucciones en lo que se conoce como un programa en un editor de texto, y posteriormente ejecutarlo.

En este capítulo aprenderemos a trabajar de las dos formas.

## Trabajando con Python desde la consola

Para ingresar a Python desde la consola, debemos entrar al terminal. Para ello abriremos la aplicación *Command Prompt* en Windows o *Terminal* en Linux/OSX, al abrirla veremos un signo peso \$ o un signo porcentual \%. Esto se conoce como el *prompt*, un símbolo que indica que el terminal está listo para recibir instrucciones



## Ingresando a Python

Desde el terminal podemos ingresar a Python interactivo. Para ello escribiremos `python` y posteriormente daremos `Enter`.

Si lo hicimos de forma correcta, deberíamos ver que el prompt cambió, esto es porque ahora estamos dentro de otro programa `python`.

```
[isz:~] isz% python
Python 3.6.5 |Anaconda, Inc.| (default, Apr 26 2018, 08:42:37)
[GCC 4.2.1 Compatible Clang 4.0.1 (tags/RELEASE_401/final)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> █
```

## Probando Python

Dentro de este programa podemos escribir instrucciones válidas para Python. Por ejemplo, podemos escribir `2 + 2` y ver como resultado `4`.

```
[isz:~] isz% python
Python 3.6.5 |Anaconda, Inc.| (default, Apr 26 2018, 08:42:37)
[GCC 4.2.1 Compatible Clang 4.0.1 (tags/RELEASE_401/final)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 2 + 2
4
>>> █
```

## ¿Qué son los otros números que aparecen en la pantalla?

Antes de probar otras operaciones, revisemos algunos detalles que son relevantes:

- El prompt en este caso se señala con `>>>`.
- En el inicio sale `Python 3.6.5 | Anaconda Inc.`, que hace referencia a la versión de Python ocupada, y a qué distribución específica pertenece, en este caso nuestra consola está en Python 3.6.5 y fue administrada por el gestor de ambientes `Anaconda`.
- Los retornos y resultados de toda expresión válidamente ejecutada no se preceden con símbolo alguno.

## Realizando nuestras primeras pruebas

Ejecuta en Python las siguientes instrucciones:

- ¿Qué se obtiene al realizar la operación `5 / 2` ?
- ¿Qué se obtiene al realizar la operación `'hola' + 'mundo'` ?
- ¿Qué se obtiene al realizar la operación `sum([1, 2, 3])` ?

## Saliendo de Python

Para salir de Python, debemos escribir `quit()`. Con ell volveremos al prompt original del terminal.

## Trabajando con Python desde el editor de texto

Existen varios editores de texto gratuitos como [Atom](#), [Visual Studio Code](#) y [Sublime Text](#). Éste último es de pago, pero ofrece una versión gratuita.

Puedes escoger el editor que más te guste. Si no sabes cual escoger, te sugerimos utilizar *Visual Studio Code* ya que es el que utilizaremos para los ejemplos del módulo.

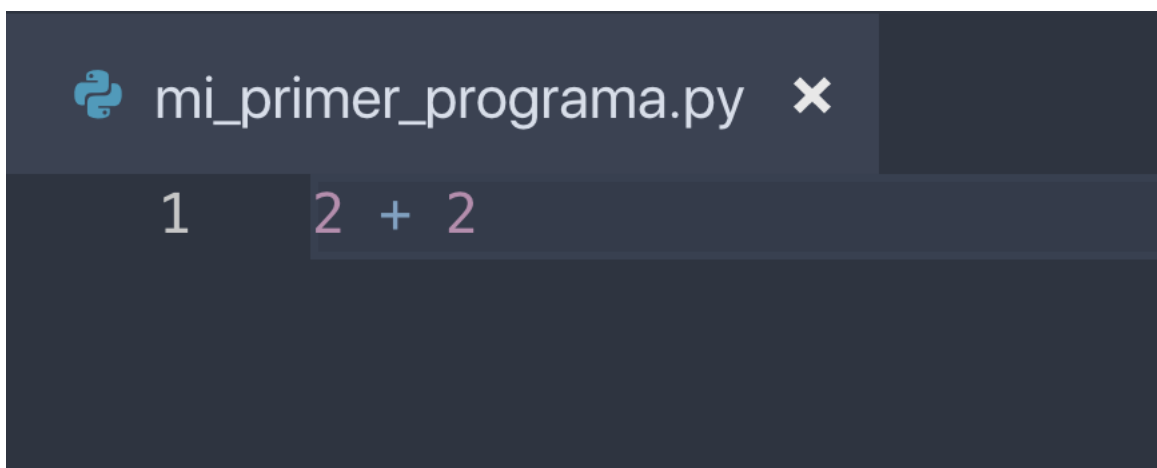
### Extensión de los archivos `.py`

Para escribir un programa desde cero en Python en un editor de texto, solo necesitamos crear un archivo con extensión `.py`.

### Nuestro primer programa en el editor

Para escribir nuestro primer programa en el editor de texto, vamos a seguir los siguientes pasos:

1. Abrir nuestro editor de texto.
2. Escribir una instrucción sencilla.
3. Guardar el archivo con extensión `.py`.



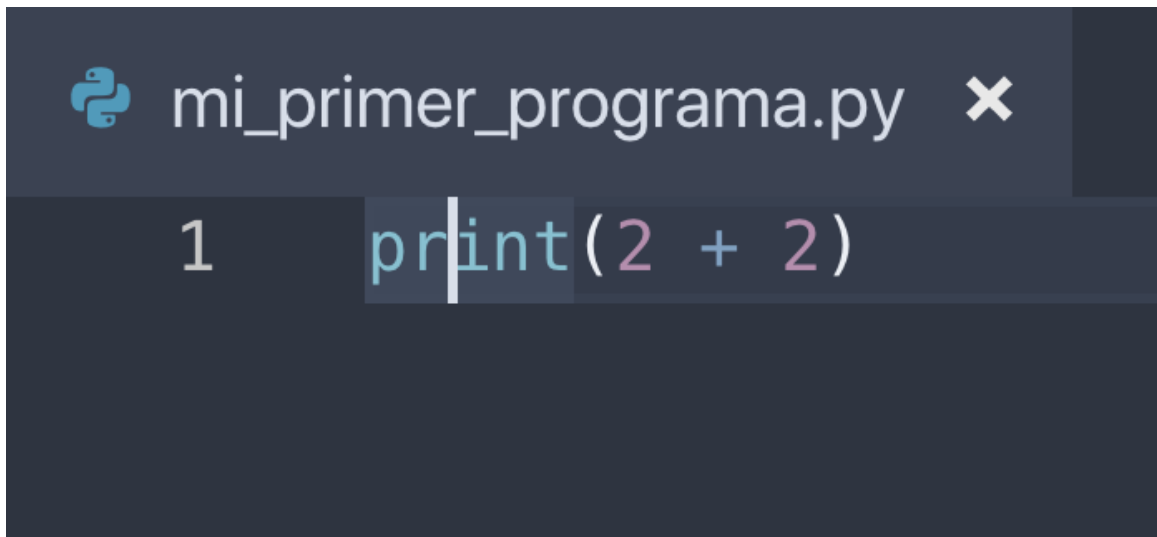
### Ejecutando el programa

A continuación debemos abrir un terminal en la misma ubicación donde se encuentra el archivo y ejecutar `python nombre_archivo.py`



### Mostrando el resultado

Lamentablemente nuestro script no devuelve nada. Este comportamiento se debe a que cuando ejecutamos un script desde el terminal, por defecto Python no devuelve el resultado de cada línea. Si deseamos mostrar el resultado, debemos indicar a Python de manera explícita que muestre el resultado en pantalla. Esto lo logramos con la instrucción `print()`

A screenshot of a code editor window. The title bar at the top shows a Python logo icon, the filename 'mi\_primer\_programa.py', and a close button 'X'. The editor area has a dark background. Line 1 of the code is highlighted, showing the command 'print(2 + 2)' in a light blue monospace font. A vertical cursor is positioned at the end of the line, after the closing parenthesis.

## Ejecutando el programa

Si ejecutamos nuevamente el script, ahora si veremos el resultado en pantalla:

```
[isz:~] isz% cd ~/Desktop/  
[isz:~/Desktop] isz% python mi_primer_programa.py  
4  
[isz:~/Desktop] isz% █
```

# Capítulo 5: Elementos básicos de Python

---

## Objetivos

- Definir comentarios, variables y constantes en Python.
- Conocer y utilizar operaciones aritméticas en Python.
- Manipular strings utilizando concatenación en Python.
- Manipular strings utilizando interpolación en Python.
- Manipular entrada y salida de datos utilizando `print` e `input` en Python.
- Diferenciar un error sintáctico de uno semántico.

## Comentarios

Los comentarios son líneas de código que son ignoradas, sirven para dar indicaciones y documentar nuestro código a otros usuarios.

```
# Esta línea es un comentario
# Los comentarios son ignorados
# Pueden ser una línea nueva
print(2 + 2) # O puede acompañar una línea de código existente
# print(2 + 3) Si comentamos al principio todo la línea será ignorada
```

```
4
```

## Comentarios en múltiples líneas

Existe otra forma de hacer comentarios multilínea que no es tan utilizada, consiste en envolver todo el comentario entre tres comillas dobles al inicio y al final.

```
"""
Comentario multilínea:
Python
lo
ignoraré
"""

print("hola")
```

```
hola
```



## Indicación importante

A lo largo del módulo utilizaremos comentarios en algunos ejemplos para representar los valores mostrados en pantalla.

Esto facilitará la lectura de ejemplos como:

```
print(2 + 2) # 4
```

## Introducción a variables

Modemos entender las variables como **contenedores que pueden almacenar valores**. Éstos valores que hay dentro pueden variar, por ello su nombre. Una variable se compone de:

- Un nombre
- Un valor

## Asignando un valor a una variable

### Integers

En Python -y muchos otros lenguajes de programación- los números enteros se denominan 'enteros' y **Integers**. De tal manera, podemos decir que: La variable `a` contiene un *Integer* y su valor es 27.

## Existen otros tipos ademas de Integer

Una variable puede almacenar otros tipos de contenido, por ejemplo, puede almacenar una palabra o una frase.

```
a = 'hola'
```

Esto se lee: La variable `a` contiene el valor `hola`.

### Strings

A una palabra o frase se le conoce como cadena de caracteres o **String**. Un *String* puede estar formado por:

```
a = 'Hola Mundo!' # Una o varias palabras  
b = 'x' # Simplemente un caracter.
```

## ¿Por qué los *Strings* se escriben entre comillas?

Python necesita algún mecanismo para diferenciar si el programador está haciendo referencia a una variable o a un *string*.

```
a = 'Esto es un string'  
b = a  
print(b)
```

```
Esto es un string
```

En la instrucción `b = a`, Python entiende que nos estamos refiriendo a la variable `a` y no a un caracter 'a'

## ¿Comillas simples o dobles?

Para trabajar con *Strings* podemos utilizar comillas simples (') o dobles ("):

```
a = 'Esto es un String'  
b = "Esto también es un String"
```

## El salto de línea

El salto de línea es un caracter especial que nos permite agregar un salto de línea dentro de un string. Este caracter es `\n`

```
a = "hola\na\ntodos"  
print(a)
```

```
hola  
a  
todos
```

El salto de línea se necesita ocuparlo dentro de comillas dobles para indicar que el backslash no significa " literalmente si no que está escapando la n para transformarlo en un salto de línea.

## Operando con variables

Hasta ahora, hemos aprendido que una variable puede contener números enteros (*Integers*) o palabras (*Strings*). Utilizando estos valores podemos realizar operaciones.

### Sumas y restas

Podemos operar con variables de tipo entero tal como si estuviéramos operando con el número mismo, por ejemplo:

```
a = 5  
b = 2  
print(a + 2)
```

7

```
print(b - a)
```

-3

### Multiplicaciones y divisiones

Para realizar operaciones de multiplicación utilizaremos `*` y para operaciones de división utilizaremos `/`

```
a = 10  
b = 2  
print(a * b)
```

20

```
print(a / b)
```

5.0

## Sobreescribiendo variables

Podemos sobreescribir el valor de una variable realizando una nueva asignación:

```
a = 'HOLA!'
print(a)
```

```
HOLA!
```

```
a = 100
print(a)
```

```
100
```

## Modificando una variable existente

También podemos modificar el valor de una variable operando sobre ella misma, este tipo de operación es muy utilizada:

```
a = 2
a = a + 1
print(a)
```

```
3
```

## Operando con variables de tipo String

¿Qué obtendremos al "sumar" dos Strings?

```
a = 'HOLA'
b = 'MUNDO'
print(a + b)
```

```
HOLA MUNDO
```

## Concatenando números

En programación, la acción de 'sumar' dos o más *Strings* se conoce como **concatenación**. Observemos el siguiente ejemplo:

```
a = '7'  
b = '3'  
print(a + b)
```

73

Al asignar los valores utilizando comillas simples, Python los interpreta como String.

## Interpolación

Otra acción muy importante y ampliamente utilizada al momento de trabajar con Strings es la **interpolación**.

La interpolación es un mecanismo que nos permite introducir una variable (o un dato) dentro un String sin necesidad de concatenarlo. Para interpolar simplemente tenemos que introducir la variable (o dato) utilizando la siguiente notación:

```
edad = 30  
texto = 'tienes {} años'.format(edad)  
print(texto)
```

tienes 30 años

## La interpolación es más fácil de usar que la concatenación

Podemos obtener los mismo resultados utilizando concatenación e interpolación, sin embargo, se prefiere la interpolación debido a que es más rápida y presenta una sintaxis más amigable para el desarrollador.

```
nombre = 'Carlos'  
apellido = 'Santana'
```

```
# Concatenación  
print("Mi nombre es " + nombre + " " + apellido)
```

Mi nombre es Carlos Santana

```
# Interpolación
print("Mi nombre es {}".format(nombre, apellido))
```

Mi nombre es Carlos Santana

## Dato interesante

A veces a mostrar le diremos imprimir, esto tiene razones históricas cuando no existían las pantallas y la única forma de mostrar el resultado era imprimirlo, por eso a veces se dice imprimir en pantalla.

## Entrada de datos

Con frecuencia nuestro script necesitará interactuar con el usuario, ya sea para seleccionar una opción de un menú o para simplemente ingresar un valor sobre el cuál nuestro script va a operar:

Podemos capturar datos introducidos por un usuario ocupando la instrucción `input`, y de la siguiente forma:

## Ingresando datos con input

Antes de mostrar el valor, la consola quedará bloqueada hasta que ingresemos una secuencia de caracteres y presionemos la tecla enter.

```
Python 3.6.5 |Anaconda, Inc.| (default, Apr 26 2018, 08:42:37)
[GCC 4.2.1 Compatible Clang 4.0.1 (tags/RELEASE_401/final)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> nombre = input()
Carlos Santana
>>> nombre
'Carlos Santana'
>>> █
```

## Creando un programa con lo aprendido

Con esta simple entrada de datos ya podemos crear programas sencillos. Por ejemplo, un programa que solicite tu nombre y luego te salude:

```
nombre = input()
print('Hola {}'.format(nombre))
```

## Concatenando números por error

Otro detalle interesante es que `input` siempre entrega un *String*, por lo tanto, si aplicamos operaciones de suma (+) a números ingresados por teclado, estos serán concatenados.

```
num1 = input() #4
num2 = input() #6
num1 + num2
```

```
'46'
```

## Transformando los datos

Este comportamiento se puede modificar aplicando **transformaciones a los tipos de datos**.

```
num1 = int(input()) #4
num2 = int(input()) #6
num1 + num2
```

```
10
```

La transformación de tipos de datos la estudiaremos en profundidad en un próximo capítulo.

## Errores sintácticos vs errores semánticos

Los errores podemos clasificarlos en dos tipos: sintácticos y semánticos.

- Los errores sintácticos son muy similares a los errores gramaticales, donde escribimos algo que contraviene las reglas de como debe ser escrito.

Por ejemplo: asignar una variable es de izquierda a derecha, `a = 5` mientras lo contrario sería un error sintáctico `5 = a`

- Los errores semánticos son muy distintos, el programa funcionará de forma normal pero el resultado será distinto al esperado. Los errores semánticos son errores de implementación o de diseño.

## Resumen capítulo

En este capítulo hablamos de varios elementos claves para construir un programa en Python, estos son:

- Variables: Contenedores de datos
- Operadores: Nos permiten sumar, restar y hasta concatenar datos
- Entrada y salida de datos: Con `input` y `print`
- Tipos de datos: Concepto importante
- Interpolación: Forma de utilizar una variable dentro de un string
- Comentarios: Forma de anotar sin que modifique el comportamiento del código
- Salto de línea: `"\n"` Permite ingresar un salto de línea en un string



# Capítulo 6: Introducción a objetos

---

## Objetivos:

- Conocer la documentación oficial de Python.
- Conocer la sintaxis de métodos y argumentos.
- Interpretar documentación oficial de Python.

## Objetos y métodos

Hasta el momento hemos trabajado con tipos de dato **Integer** y **String**. En Python estos tipos de dato reciben el nombre de **objetos** y las operaciones que realizamos sobre ellos reciben el nombre de **métodos**.

## Los métodos nos permiten operar con los objetos bajo ciertas restricciones

Por ejemplo, el método `+` nos permite sumar dos enteros, o concatenar dos strings, pero no nos permite sumar un string con un entero.

Ya hemos ocupado varios métodos.

- `input`
- `.format`
- `int`
- operaciones como `+`

A través de la documentación podemos conocer otros métodos.

## Revisando la documentación

Para conocer qué operaciones son válidas sobre un objeto debemos leer su [documentación oficial](#).

Consultar la documentación oficial es un hábito que **debemos** adoptar al momento de aprender un nuevo lenguaje de programación.

## ¿Cómo se lee la documentación?

Utilizaremos la documentación de [Python](#). En esta página podemos ver la documentación todos los objetos incluidos en Python.

Partamos revisando uno de los objetos que más hemos utilizado: Los [integers](#).

## Leyendo la documentación

La manera más fácil de encontrar la documentación asociada a un elemento en específico es utilizando el buscador integrado a la página de documentación que se encuentra en la esquina superior derecha.

### Numeric Types — `int`, `float`, `complex`

There are three distinct numeric types: *integers*, *floating point numbers*, and *complex numbers*. In addition, Booleans are a subtype of integers. Integers have unlimited precision. Floating point numbers are usually implemented using `double` in C; information about the precision and internal representation of floating point numbers for the machine on which your program is running is available in `sys.float_info`. Complex numbers have a real and imaginary part, which are each a floating point number. To extract these parts from a complex number `z`, use `z.real` and `z.imag`. (The standard library includes additional numeric types, `fractions` that hold rationals, and `decimal` that hold floating-point numbers with user-definable precision.)

Numbers are created by numeric literals or as the result of built-in functions and operators. Unadorned integer literals (including hex, octal and binary numbers) yield integers. Numeric literals containing a decimal point or an exponent sign yield floating point numbers. Appending `'j'` or `'J'` to a numeric literal yields an imaginary number (a complex number with a zero real part) which you can add to an integer or float to get a complex number with real and imaginary parts.

Python fully supports mixed arithmetic: when a binary arithmetic operator has operands of different numeric types, the operand with the “narrower” type is widened to that of the other, where integer is narrower than floating point, which is narrower than complex. Comparisons between numbers of mixed type use the same rule. [2] The constructors `int()`, `float()`, and `complex()` can be used to produce numbers of a specific type.

All numeric types (except complex) support the following operations, sorted by ascending priority (all numeric operations have a higher priority than comparison operations):

Operation	Result	Notes	Full documentation
<code>x + y</code>	sum of <code>x</code> and <code>y</code>		
<code>x - y</code>	difference of <code>x</code> and <code>y</code>		
<code>x * y</code>	product of <code>x</code> and <code>y</code>		
<code>x / y</code>	quotient of <code>x</code> and <code>y</code>		
<code>x // y</code>	floored quotient of <code>x</code> and <code>y</code>	(1)	
<code>x % y</code>	remainder of <code>x / y</code>	(2)	
<code>-x</code>	<code>x</code> negated		
<code>+x</code>	<code>x</code> unchanged		
<code>abs(x)</code>	absolute value or magnitude of <code>x</code>		<a href="#">abs()</a>
<code>int(x)</code>	<code>x</code> converted to integer	(3)(6)	<a href="#">int()</a>
<code>float(x)</code>	<code>x</code> converted to floating point	(4)(6)	<a href="#">float()</a>
<code>complex(re, im)</code>	a complex number with real part <code>re</code> , imaginary part <code>im</code> ; <code>im</code> defaults to zero	(6)	<a href="#">complex()</a>

Al preguntar por `integer`, Python nos redireccionará a la sección de tipos de datos numéricos. En esta página se encuentran los detalles de cuál es la definición de un dato de tipo numérico.

Si hacemos click en `int`, la documentación hará referencia a la clase que permite generar un número entero.

```
class int([x])
```

```
class int(x, base=10)
```

Return an integer object constructed from a number or string *x*, or return 0 if no arguments are given. If *x* defines `__int__()`, `int(x)` returns `x.__int__()`. If *x* defines `__trunc__()`, it returns `x.__trunc__()`. For floating point numbers, this truncates towards zero.

If *x* is not a number or if *base* is given, then *x* must be a string, `bytes`, or `bytearray` instance representing an [integer literal](#) in radix *base*. Optionally, the literal can be preceded by `+` or `-` (with no space in between) and surrounded by whitespace. A base-*n* literal consists of the digits 0 to *n*-1, with `a` to `z` (or `A` to `Z`) having values 10 to 35. The default *base* is 10. The allowed values are 0 and 2-36. Base-2, -8, and -16 literals can be optionally prefixed with `0b/0B`, `0o/0O`, or `0x/0X`, as with integer literals in code. Base 0 means to interpret exactly as a code literal, so that the actual base is 2, 8, 10, or 16, and so that `int('010', 0)` is not legal, while `int('010')` is, as well as `int('010', 8)`.

The integer type is described in [Numeric Types — int, float, complex](#).

*Changed in version 3.4:* If *base* is not an instance of `int` and the *base* object has a `base.__index__` method, that method is called to obtain an integer for the base. Previous versions used `base.__int__` instead of `base.__index__`.

*Changed in version 3.6:* Grouping digits with underscores as in code literals is allowed.

*Changed in version 3.7:* *x* is now a positional-only parameter.

## Built-in methods

Python presenta una serie de operaciones que son nativas (o *Built-in methods*): se comportan de manera diferente dependiendo del tipo de dato que se ingresa. Supongamos que deseamos contar cuántas letras existen en la cadena `'Lorem ipsum dolor sit amet'`.

```
cadena = 'Lorem ipsum dolor sit amet'
len(cadena)
```

26

Un aspecto importante a considerar es el hecho que cada objeto creado vendrá con una serie de métodos que podremos ocupar. Si deseamos ocupar alguno de estos, debemos implementar la sintaxis: `objeto.metodo`. Si deseamos transformar todas las letras de la cadena a mayúsculas, podemos ocupar el método `upper`

```
cadena.upper()
```

'LOREM IPSUM DOLOR SIT AMET'

## Métodos con opciones

También existen métodos que requieren recibir información para operar, por ejemplo el método `count` del objeto String recibe un substring (un conjunto de caracteres mas corto) para poder contar cuantas veces está contenido ese substring en el String principal.

Imaginemos que queremos saber cuantas letras 'i' tiene la frase `Lorem ipsum dolor sit amet` :

```
cadena.count('i')
```

2

## El retorno de un método

Lo que devuelve un método se conoce como el **retorno del método** y la información que recibe se conoce como **parámetros**.

De tal manera podemos decir que el método `count` del objeto String:

- Recibe como **parámetro**, al menos, un substring.
- **Retorna** un entero correspondiente a la cantidad de veces que está contenido el substring en el objeto String.

```
'paralelepipedo'.count('p')
```

3

## ¿Y los paréntesis?

Cuando utilizamos un método, todo argumento **debe pasarse entre paréntesis**.

```
'paralelepipedo'.count('p')
```

3

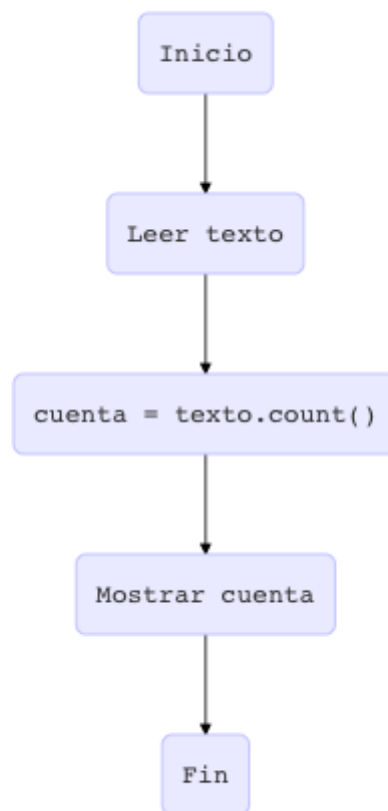
## Existen diversas formas de escribir lo mismo

Algunos métodos como el más `+` pueden ser utilizados sin el punto. Por ejemplo `2.+(2)` es lo mismo `2 +(2)` y esto es lo mismo que `2 + 2`

## Ejercitando lo aprendido

Lo aprendido lo podemos utilizar para crear un pequeño programa donde el usuario introduce un valor y mostramos la cantidad de letras:

## Algoritmo



## Código

```
content = input()
letters = len(content)
print("El contenido {} tiene {} letras".format(content, letters))
```

El contenido paralelepipedo tiene 14 letras

## Resumen del capítulo

En este capítulo aprendimos sobre objetos, métodos y como consultar la documentación.

Sobre objetos aprendimos:

- Los tipos de datos son realmente objetos (Esto es en Python y solo aplica a algunos lenguajes).
- Los objetos tienen métodos que nos permiten realizar todo tipo de acciones.

Sobre métodos aprendimos:

- Ocupar un método, llamar, o invocar un método son sinónimos.
- Solo podemos llamar métodos que ya han sido definidos (podemos definirlos nosotros o buscarlos en la documentación).
- Los métodos pueden recibir valores y esto lo hace mas flexibles.

Al consultar la documentación sobre un método debemos fijarnos en:

- Nombre del método.
- Parámetros que recibe: opcionales y obligatorios.
- Información que devuelve.

SPOILER: Más adelante profundizaremos en métodos, parámetros y retornos cuando creando nuestros propios métodos!

# Capítulo 7: Tipos de objeto

---

## Objetivos:

- Identificar un entero en Python.
- Identificar un flotante en Python.
- Identificar un string en Python.
- Identificar un boolean en Python.
- Diferenciar el comportamiento de método `+` en Integer vs String.
- Transformar Strings a Integers.

En Python existen distintos tipos de datos. Ya sabemos que estos tipos de datos son objetos, por ende, lo correcto es llamarlos **tipos de objeto**.

Cada uno de estos objetos se comporta de forma distinta.

Dentro de los tipos de datos más utilizados podemos destacar:

- `Integer` : Corresponde a un número entero.
- `String` : Corresponde a un caracter o una cadena de caracteres.
- `Float` : Corresponde a un número decimal.
- `Time` : Corresponde a una fecha y hora.
- `Boolean` : Corresponde a `True` o `False`. Son el resultado de una evaluación.
- `None` : corresponde a la ausencia de un valor.

Profundizaremos más en estos tipos de objeto a medida que avancemos.

## ¿Cómo saber de qué tipo es un objeto?

Podemos saber el tipo de dato utilizando el método `type(x)`

Por ejemplo si dentro de la consola de Python escribimos `type(2)` obtendremos como resultado `int`, o si probamos con `type('hola')` obtendremos como resultado `str`.

```
suma = 5 + 2
```

```
type(suma)
```

```
int
```

```
otra_suma = 2.3 + 0.1
```

```
type(otra_suma)
```

```
float
```

## ¿Por qué es importante el tipo de objeto?

Es importante porque existen distintas reglas para operar entre estos distintos tipos de objeto. Estas reglas las conoceremos consultando la documentación oficial.

Por ejemplo: al sumar dos números obtenemos el resultado de la suma, pero al sumar dos palabras obtenemos la concatenación de estas.

En algunas situaciones, cuando faltemos a estas reglas, las operaciones no serán válidas.

## Concatenando strings

Observemos el siguiente ejemplo: el método `+` del objeto String recibe como parámetro otro String a concatenar.

```
"Hola " + "Mundo"
```

```
'Hola Mundo'
```

## Concatenando un string con otro tipo de dato

¿Qué sucede si intentamos concatenar un Integer a un String?

```
"HOLA" + 2
```

```
TypeError
```

```
Traceback (most recent call last)
```

```
<ipython-input-34-8946f4d722d7> in <module>()
```

```
----> 1 "HOLA" + 2
```

```
TypeError: must be str, not int
```

¿Y si queremos sumar dos números ingresados por teclado?



```
numero_uno = input() #10
numero_dos = input() #20
print(numero_uno + numero_dos)
```

1020

Para solucionar este problema y, dependiendo de nuestro objetivo, podemos aplicar transformaciones a los tipos de objeto.

## Trasformando tipos de objeto

Existen distintos métodos que nos permiten transformar un objeto de un tipo a otro. Dentro de estos métodos podemos destacar:

- El método `int(x)` nos permite convertir un objeto en Integer.
- El método `str(x)` nos permite convertir un objeto en String.

```
int(2018)
```

2018

```
str(2018)
```

'2018'

¡Ahora sí podemos sumar dos números ingresados por teclado!

```
numero_uno = int(input()) #10
numero_dos = int(input()) #20
print(numero_uno + numero_dos)
```

30

## Interpolación vs transformación vs concatenación

Observemos el siguiente comportamiento:

```
nombre = 'Carlos Santana'  
edad = 71
```

```
# Concatenación  
print("Hola! Soy " + nombre + " y tengo " + edad + "años!")
```

```
-----  
TypeError                                Traceback (most recent call last)  
  
<ipython-input-40-fa33bec775b5> in <module>()  
      3  
      4 # Concatenación  
----> 5 print("Hola! Soy " + nombre + " y tengo " + edad + "años!")
```

```
TypeError: must be str, not int
```

```
# Concatenación + Transformación  
print("Hola! Soy " + nombre + " y tengo " + str(edad) + "años!")
```

```
Hola! Soy Carlos Santana y tengo 71años!
```

```
#Interpolación  
print("Hola! Soy {} y tengo {} años!".format(nombre, edad))
```

```
Hola! Soy Carlos Santana y tengo 71 años!
```

Al utilizar interpolación, no necesitaremos aplicar transformación al objeto `edad`. El método `to_s` será aplicado de forma automática al objeto que escribamos entre las llaves.

## Capítulo 8: Operaciones aritméticas

En este capítulo aprenderemos a hacer aplicaciones del tipo de calculadora, donde los usuarios ingresan valores y le entregamos resultados.

### Objetivos:

- Construir aplicaciones para calcular una función matemática.

Para lograr esto tenemos que:

- Conocer operadores aritméticos.
- Conocer la precedencia de los operadores aritméticos.
- Utilizar parentesis para priorizar operaciones.
- Operar sobre números enteros y flotantes.

### Introducción a operaciones y operadores

En Python existen herramientas que nos permiten, entre otras cosas: sumar, restar, asignar valores, comparar, etc. Todo esto -y mucho más- es posible gracias a los operadores.

### Operadores aritméticos

Los operadores aritméticos nos permiten realizar operaciones matemáticas sobre números.

Operador	Nombre	Ejemplo	Resultado
+	Suma	2 + 3	5
-	Resta	2 - 3	-1
*	Multiplicación	3 * 4	12
/	División	12 / 4	3
**	Potencia	2 ** 4	16

### Operaciones con variables

El proceso es exactamente igual si guardamos los valores en variables:

```
a = 2
b = 3
print(a + b)
```

## Creando un calculadora

Esto nos permite que el usuario ingrese los valores, transformarlos a números y luego operar.

```
a = int(input())
b = int(input())
print("a + b es: {}".format(a + b))
print("a * b es: {}".format(a * b))
```

```
a + b es: 12
a * b es: 32
```

## Precedencia de operadores

Un concepto muy importante que debemos conocer es el de precedencia, dicho de otro modo, **saber en qué orden se realiza un grupo de operaciones.**

## Ejemplo de precedencia

Por ejemplo, en la siguiente instrucción ¿cuál es el resultado?

```
10 - 5 * 2
```

$$\begin{array}{r} 10 - 5 * 2 \\ \hline 10 - 10 \\ 0 \end{array}$$

```
10 - 5 * 2 # 0
```

## Orden de las operaciones

Veamos una tabla simplificada de precedencia. Esta tabla está ordenada de mayor a menor prioridad, esto quiere decir que la operación de exponenciación tiene mayor precedencia que la suma

Operador	Nombre
**	Exponenciación (potencia)
*, %	Multiplicación, división y módulo
+, -	Suma y resta

Cuando dos operaciones tienen el mismo nivel de prioridad entonces se resuelven de izquierda a derecha.

## Operaciones y paréntesis

Al igual en matemáticas, los paréntesis cambian el orden en que preceden las operaciones dando prioridad a las operaciones que estén dentro de los paréntesis.

$$(10 - 5) * 2$$

$$(10 - 5) * 2$$



$$10 * 2$$

$$20$$

¡Los paréntesis importan!

## Operaciones con números enteros y decimales

Si dividimos números enteros nos encontraremos con una sorpresa.

```
5 / 3 # => 1
```

Esto es muy común en todos los lenguajes de programación, para obtener la respuesta que esperamos necesitamos ocupar otro tipo de dato, el float.

## Float

En el capítulo anterior mencionamos que existía el tipo de dato asociado a los números **decimales** se llamaba float.

```
type(3.1)
```

## Enteros y floats

La división entre entero y float, o float y entero da como resultado un float.

```
5.0 / 3.0 # 1.6666666666666667  
5 / 3.0 # 1.6666666666666667
```

La división entre flotas también es un float.

Los floats son muy importantes dentro de la programación y tienen propiedades curiosas, una de las mas importantes es que solo son una representación aproximada de los números.

```
(10 / 3.0)
3.3333333333333335
```

También podemos transformar a float ocupando el método `to_f`, esto será especialmente útil cuando estemos trabajando con variables que contengan enteros.

```
a = int(input()) # 1
b = int(input()) # 2
print(a / float(b)) # 0.5
```

O, pudimos haber transformado en un inicio la variable a float.

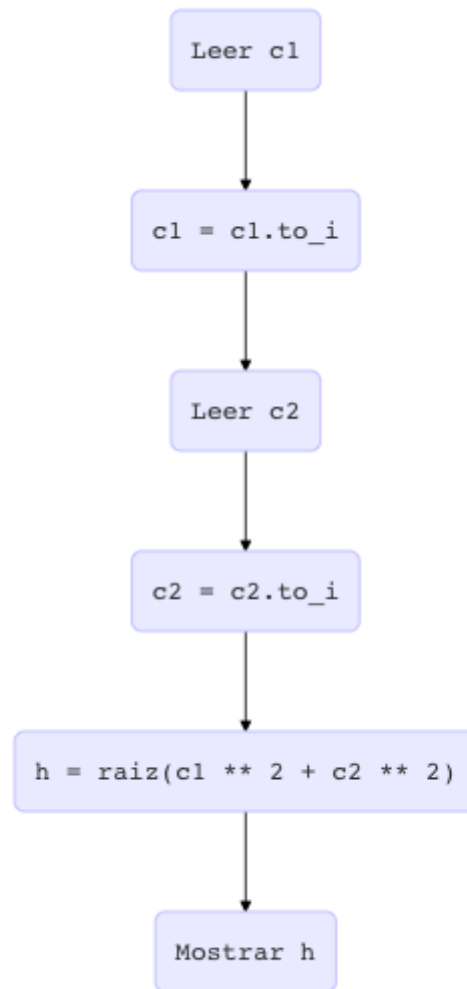
## Ejercicio de Pitágoras

La fórmula de Pitágoras nos permite calcular el largo de la hipotenusa de un triángulo rectángulo a partir de los largos de los catetos. Crearemos un programa donde el usuario introduzca los valores de ambos catetos y entreguemos como resultado el largo de la hipotenusa.

Para implementar el código solo necesitamos conocer la fórmula

$$h = \sqrt{c1^2 + c2^2}$$

## Algoritmo



Al leer un algoritmo tenemos que identificar que partes no sabemos implementar, o pueden ser complejas. En el caso del algoritmo anterior lo único que no sabemos es calcular la raíz.

Si buscamos en google `square root y python` encontraremos rápido la respuesta.

## Código

```
import math
c1 = int(input()) # 3
c2 = int(input()) # 4
h = math.sqrt(c1 ** 2 + c2 ** 2) # 5.0
```

`math` es un módulo, en cierto modo son similares a los objetos en el sentido de que tienen varios métodos. La documentación de un módulo se lee de igual forma que el de un objeto.

## Fahrenheit



Fahrenheit es una escala de temperatura, podemos transformar una temperatura de la escala Fahrenheit a Celsius ocupando la siguiente ecuación.

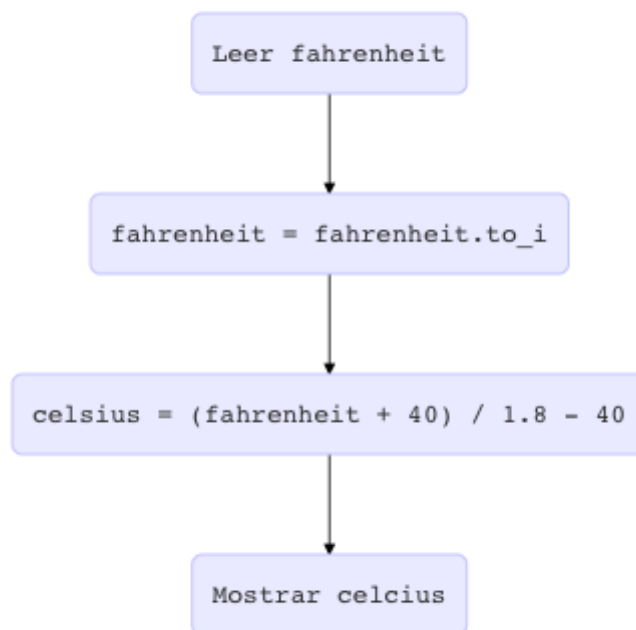
$$((f + 40) \div 1.8) - 40 = c$$

Antes de desarrollar el algoritmo revisar los siguientes puntos:

- ¿Cuántos valores tiene que ingresar el usuario, 1 o 2?
- ¿Existe algún punto donde tengamos que tener cuidado con la precedencia de operadores?

Luego dibuja el diagrama de flujo y finalmente escribe el código.

## Algoritmo



```
fahrenheit = int(input())
celsius = (fahrenheit + 40) / 1.8 - 40
print("la temperatura es de {} celsius".format(celsius))
```

la temperatura es de 0.0 celsius

## Ejercicios propuestos

- El usuario ingresa 3 valores y se debe calcular el promedio de estos.
- El usuario ingresa un radio de una circunferencia y se debe calcular el área  
 $area = \pi * r^2$

Pregunta:

¿Qué tipo de dato da como resultado  $5.0 * 3$ ?

- Entero
- Float
- String

**Float**

## Resumen del capítulo

- Precedencia: El orden en que deben ser resuelta las operaciones.
- División entre enteros: El resultado es entero `6 / 4 #=> 0`
- División entre flotante y entero: El resultado es flotante `6 / 4.0 #=> 1.5`
- Paréntesis(): Al igual que en matemáticas nos ayudan a darle prioridad a una operación.

## Capítulo 9: Manejo básico de flujo

### Objetivos:

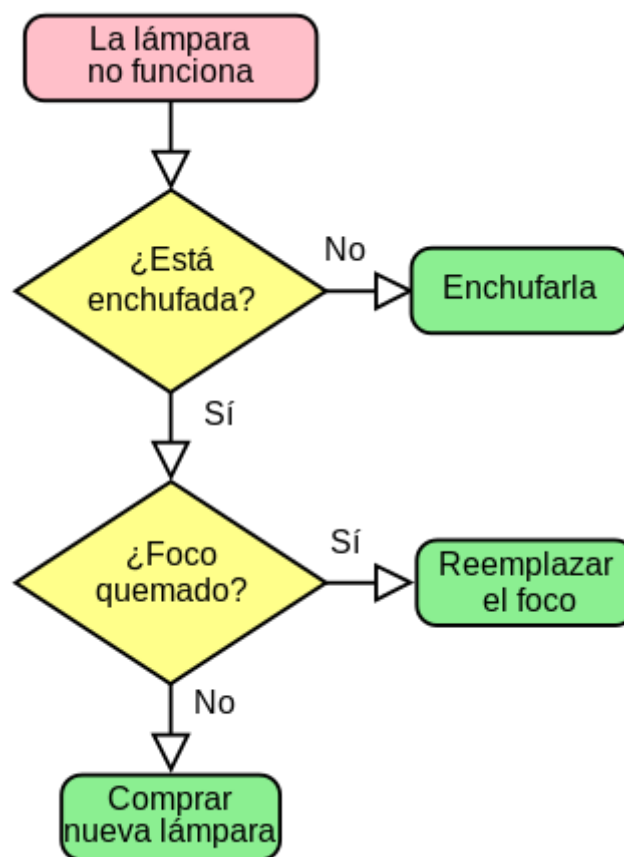
En este capítulo aprenderemos a resolver algoritmos donde hayan una o dos opciones a resolver.

- Crear algoritmos que tomen decisiones en base al valor de una variable

Para esto tenemos que:

- Conocer operadores de comparación.
- Diferenciar asignación de comparación.

Al inicio de la unidad hablamos mostramos el siguiente diagrama de flujo.



El rombo corresponde a una **evaluación condicional**. En este punto se toma una decisión y el programa sigue en función de esa decisión. En este capítulo aprenderemos a escribir programas que implementen estas decisiones.

### La instrucción IF

Python -y muchos otros lenguajes de programación- tiene instrucciones para implementar condiciones. Una de la más utilizada es la instrucción `if`.

```
if condition:  
    # código que se ejecutará SÓLO si se cumple la condición
```

Los anterior se lee como: "**Si** se cumple la condición, **entonces** ejecuta el código".

## Ejemplo de if

Analicemos el siguiente ejemplo: En muchos países de Latinoamérica, la mayoría de edad se alcanza a la edad de 18 años. Crearemos un programa que pregunte la edad al usuario. Si la edad es mayor o igual a 18 entonces le diremos que es mayor de edad.

```
edad = int(input("¿Qué edad tienes?"))  
  
if edad >= 18:  
    print("Eres mayor de edad")
```

Eres mayor de edad

## Probando el programa

Si ejecutamos el programa e introducimos un valor mayor o igual a 18 veremos el mensaje "Eres mayor de edad" en caso contrario no veremos ningún mensaje.

```
[isz:~/Desktop] isz% python ex1.py  
¿Qué edad tienes?  
24  
Eres mayor de edad  
[isz:~/Desktop] isz% python ex1.py  
¿Qué edad tienes?  
6
```

En este ejercicio comparamos utilizando el operador `>=`, pero existen varios operadores que nos permiten comparar, los estudiaremos a continuación.

## Operadores de comparación

Los operadores de comparación son aquellos que comparan dos valores y obtienen como resultado `True` (verdadero) o `False` (falso). A este tipo de objeto, resultado de una comparación, se le conoce como **booleano**.

## Operadores de comparación en integers

Operador	Nombre	Ejemplo	Resultado
==	Igual a	2 == 2	true
!=	Distinto a	2 != 2	false
>	Mayor a	3 > 4	false
>=	Mayor o igual a	3 >= 3	true
<	Menor a	4 < 3	false
<=	Menor o igual a	3 <= 4	true

Realicemos una prueba en Python donde el usuario ingrese los valores y veamos si el primero es mayor que el segundo.

```
a = int(input()) # 3
b = int(input()) # 5
print(a > b)
```

False

## Operadores de comparación en strings

Aunque en la tabla solo hayamos mostrado números, podemos comparar dos objetos utilizando un operador de comparación:

```
'texto1' == 'texto2' # False
```

## ¿Puede un texto ser mayor que otro?

Otros operadores también pueden ser aplicados a los strings, por ejemplo el `>`

```
'a' > 'b' # False
```

En este caso la comparación es por orden alfabético, las letras que aparecen primero en el alfabeto son menores que las que aparecen después. Para entenderlo de forma sencilla cuando comparamos dos palabras es menor la que aparecería antes en un diccionario.



En el ejemplo vemos que la palabra recycle es menor que la palabra red porque la letra **c** es menor que **d**.

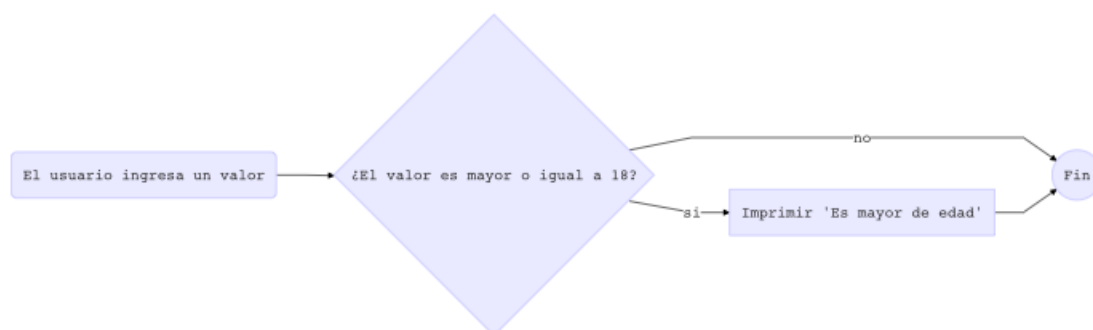
Ahora que ya conocemos los operadores de comparación volvamos a nuestro código anterior y analicemos el diagrama de flujo.

```
edad = int(input("¿Qué edad tienes?"))
```

```
if edad >= 18:
```

```
    print("Eres mayor de edad")
```

Eres mayor de edad



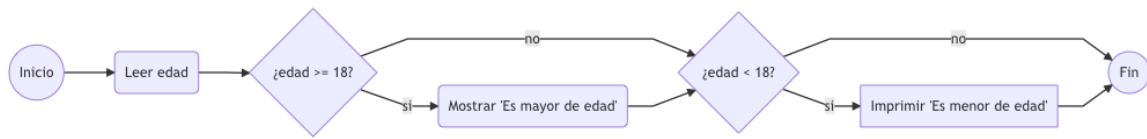
## Importante

- Todo lo que está dentro del `if`, sucederá solo si se cumple la condición.
- El código dentro del `if` **debe** estar indentado por 4 espacios para poder reconocer de forma sencilla y visualmente dentro del código donde empieza, termina y que código se ejecuta dentro de la condición.

## En caso contrario

¿Cómo podemos modificar nuestro programa para que muestre un mensaje cuando el usuario sea menor edad y otro mensaje cuando el usuario sea mayor de edad?

Una muy **buena práctica** es la de realizar un diagrama de flujo antes de comenzar a programar. Esto ayuda a abstraerse del código y pensar en los pasos críticos.



Transcribamos nuestro diagrama de flujo a código Python:

```
edad = int(input("¿Qué edad tienes?"))
```

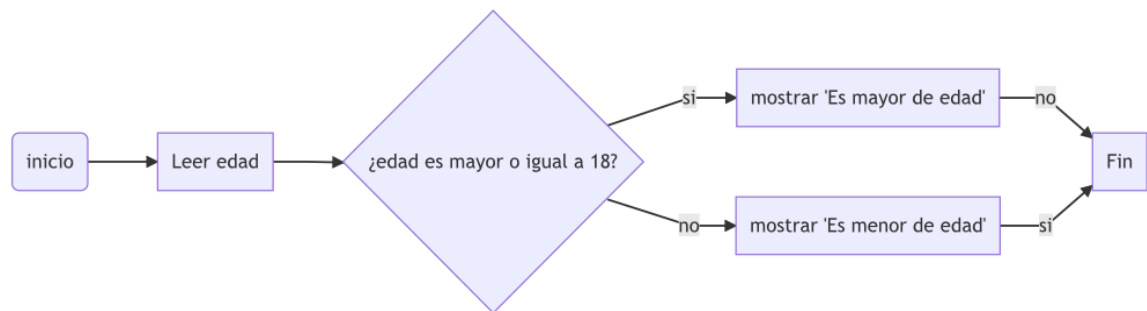
```
if edad >= 18:
    print("Eres mayor de edad")
```

```
if edad <= 18:
    print("Eres menor de edad")
```

Eres mayor de edad

Una pregunta interesante que nuestro diagrama refleja de manera implícita es: ¿Puede el usuario ser mayor y menor de edad a la vez?

La respuesta es evidentemente **no**. Este tipo de situaciones se puede modelar mejor de la siguiente forma.



Para implementarlo tenemos que introducir una nueva instrucción: `else`.

## La instrucción ELSE

La instrucción `else` se utiliza junto con `if` para seguir el flujo de código **en cualquier caso donde no se cumpla la condición**.

```
if condition:
    # código que se ejecutará SÓLO SI se cumple la condición
else:
    # código que se ejecutará si NO se cumple la condición
```

Los anterior se lee como: "**Si** se cumple la condición, **entonces** ejecuta el bloque de código, **sino** ejecuta el otro bloque de código".

```
edad = int(input("¿Qué edad tienes?"))

if edad >= 18:
    print("Eres mayor de edad")
else:
    print("Eres menor de edad")
```

Eres mayor de edad

## Ejercicio práctico

Crear un programa donde el usuario deba ingresar dos números. El programa debe imprimir el mayor de ambos números.

```
valor1 = int(input("Ingresa valor 1"))
valor2 = int(input("Ingresa valor 2"))

if valor1 >= valor2:
    print("valor1 {} es mayor".format(valor1))
else:
    print("valor2 {} es mayor".format(valor2))
```

valor1 45 es mayor

## Ejercicios propuestos:

- El usuario ingresa debe ingresar un password en la plataforma, si el password tiene menos de 6 letras se debe mostrar un aviso.
- El usuario debe ingresar un password, si el password es 12345 se debe informar que el password es incorrecto.

## Asignación V.S. Comparación

No debemos confundir asignación del valor de una variable con el de comparación.



```
# Asignación
nombre = 'Carlos Santana'
print(nombre) # 'Carlos Santana'

# Comparación
nombre == 'Carlos Santana'
```

## Resumen de lo aprendido

- En este capítulo aprendimos a utilizar las instrucciones `if` y `else` las cuales nos permiten manejar el flujo de un programa.
- Existe operadores de comparación que nos permiten comparar si una expresión es verdad o mentira.
- Podemos utilizar los operadores de comparación dentro de los `if`.
- Debemos tener cuidado de no confundir la asignación de la comparación. o sea: `=` es distinto que `==`.
- La instrucción `end` es importante, por cada `if` tiene que haber un `end`.
- Todo lo que está dentro del `if`, sucederá solo si se cumple la condición.

# Capítulo 10: Profundizando en flujo

## Objetivo:

En este capítulo aprenderemos a resolver problemas donde las posibles alternativas a una decisión sean mas de dos.

- Analizar situaciones más allá de opción y caso contrario.
- Utilizar la instrucción 'elif' en Python

Abordemos el problema desde un ejemplo.

En el capítulo anterior creamos un algoritmo donde de dos números debíamos determinar cual es el mayor. Utilizaremos el mismo ejemplo agregando que debamos mostrar cuando ambos números son iguales.

Antes:

- caso1:  $A \geq B \Rightarrow$  Decimos que A es mayor
- caso2:  $A < B \Rightarrow$  Decimos que A es menor

Ahora:

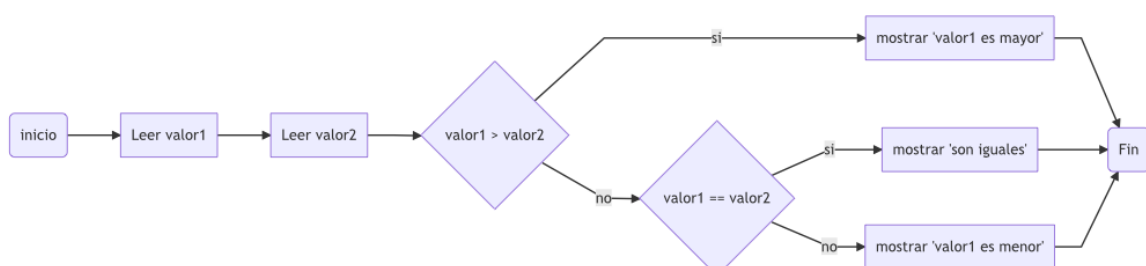
- caso1:  $A > B \Rightarrow$  Decimos que A es mayor
- caso2:  $A == B \Rightarrow$  Decimos que A y B son iguales
- caso3:  $A < B \Rightarrow$  Decimos que A es menor que B

## El mayor de dos números: Escenario donde se ingresan dos números iguales

¿Cómo hacemos para mostrar que ambos son iguales? Ante cualquier problema de este tipo lo mejor es ir paso a paso y analizar el problema.

- Si el primero es mayor -> Mostramos mensaje.
- Si NO es menor -> Tenemos dos opciones:
  - El primer valor es menor
  - Ambos son iguales

Con esta descomposición del problema tenemos toda la información necesaria para construir nuestro diagrama:



Con el diagrama de flujo realizado, la transcripción a código Python se vuelve sencilla:

```
valor1 = int(input("Ingrese valor 1"))
valor2 = int(input("Ingrese valor 2"))

if valor1 > valor2:
    print("valor1 {} es mayor".format(valor1))
else:
    if valor1 == valor2:
        print("Ambos valores son iguales")
    else:
        print("valor2 {} es mayor".format(valor2))
```

valor1 45 es mayor

Un código siempre debe ser probado y debemos probar todos los casos, probemos con los valores:

- 2, 1
- 2, 2
- 2, 3

Existe otra solución sin tener que agregar un if dentro del otro.

## La instrucción ELIF

La instrucción `elif` nos permite capturar el flujo y volver a realizar una evaluación condicional cuando no se cumplió una evaluación previa. Con esta instrucción podemos reescribir el algoritmo del ejemplo anterior:

```
valor1 = int(input("Ingrese valor 1"))
valor2 = int(input("Ingrese valor 2"))

if valor1 > valor2:
    print("valor1 {} es mayor".format(valor1))
elif valor1 == valor2:
    print("Ambos valores son iguales")
else:
    print("valor2 {} es mayor".format(valor2))
```

valor1 87 es mayor

La instrucción `elif` nos permite continuar realizando tantas evaluaciones condicionales como necesitemos. Finalmente podemos utilizar la instrucción `else` para cuando no se cumpla ninguna condición anterior.

Cabe destacar que dos instrucciones `if` **no es lo mismo** que utilizar `elif`.

## Clasificación según rangos

Es normal estar en situaciones donde nos pidan clasificar alguna elemento según un rango.

Por ejemplo supongamos que tenemos una palabra y queremos clasificarla en corta, mediana y larga:

- 4 letras o menos será corta.
- 5 a 10 será mediana.
- Más de 10 será larga.

```
palabra = input("Ingrese una palabra")
largo = len(palabra)

if largo <= 4:
    print("Pequeña")
elif largo < 10:
    print("Mediana")
else:
    print("Larga")
```

Pequeña

Ejercicio: Modifica el código anterior para poder distinguir palabras muy largas, cuando tengan 15 o mas caracteres.

```
# solución
palabra = input("Ingrese una palabra")
largo = len(palabra)

if largo <= 4:
    print("Pequeña")
elif largo < 10:
    print("Mediana")
elif largo < 14:
    print("Larga")
else:
    print("Muy larga")
```

Muy larga

# Capítulo 11: Condiciones de borde

---

## Objetivos:

- Conocer la importancia de condiciones de borde.
- Analizar condiciones de borde.
- Evaluar comportamiento de operadores  $>$ ,  $>=$ ,  $<$  y  $<=$  en Python.

## Motivación

En este capítulo aprenderemos a analizar código y buscar errores semánticos, o sea cuando el código está bien escrito pero no hace exactamente lo que queremos que haga. Este tipo de errores es muy típico y es el que mas tiempo consume.

La falta de un símbolo, un `;` a algo similar son errores sintácticos, estos errores son bastante fáciles de detectar, cuando un programa no funciona como debería puede llegar a ser bastante mas complejo, porque no hay ningún mensaje de error asociado, ni una línea de código en específico que falle.

En este capítulo aprenderemos un tipo de análisis que nos ayuda a descubrir ese tipo de problemas, se llama análisis de condiciones de borde.

Cuando evaluamos los casos específicos alrededor de una condición estamos hablando de **condiciones de borde**. Estos son, precisamente, los puntos donde debemos tener más cuidado.

## Analizando una condición de borde

Revisemos algunos problemas que hemos visto hasta ahora y analicemos sus bordes.

En el siguiente ejemplo donde se pregunta si una persona es mayor de edad.

```
edad = int(input("¿Qué edad tienes?"))

if edad >= 18:
    print("Eres mayor de edad")
else:
    print("Eres menor de edad")
```

Eres mayor de edad

En este ejemplo el punto que define una rama de la otra es el 18. A este punto se le suele llamar punto crítico, mientras que los bordes son los números que "bordean" el 18, o sea analizar los bordes sería probar con los valores **17, 18 y 19**.

En el siguiente caso intenta analizar los bordes:

```
palabra = input("Ingrese una palabra")
largo = len(palabra)

if largo <= 4:
    print("Pequeña")
elif largo < 10:
    print("Mediana")
else:
    print("Larga")
```

Larga

En el ejemplo los puntos críticos, son 4 y 10, nuestros puntos de analizar deberían ser 3,4,5 y 9,10,11

Veamos un caso un poco distinto, cuando el punto crítico no es un número exacto.

```
valor1 = int(input("Ingrese valor 1"))
valor2 = int(input("Ingrese valor 2"))

if valor1 > valor2:
    print("valor1 {} es mayor".format(valor1))
elif valor1 == valor2:
    print("Ambos valores son iguales")
else:
    print("valor2 {} es mayor".format(valor2))
```

valor2 90 es mayor

En este ejemplo el caso críticos es cuando ambos valores son iguales, y los bordes son cuando el primer número es mayor que el segundo y cuando el primer número es menor que el segundo.

En este caso podemos escoger un par de números para hacer la prueba, digamos 2 y 2, y luego los bordes serían 3 y 2, y 1 y 2.

## Capítulo 12: Operadores lógicos

### Objetivos:

- Hacer uso de los operadores lógicos para evaluar y simplificar expresiones.
- Invertir una condición.

¿Cómo evaluarías la siguiente expresión?

```
a = 24
a > 20 y a < 30
```

Para probarla aprenderemos que la expresión y se puede escribir como `and`

```
a = 24
a > 20 and a < 30 # Ytrue
```

¿Cómo evaluarías la siguiente expresión?

```
a = 32
a > 20 and a < 30
```

```
a = 32
a > 20 and a < 30 # false
```

Es falsa porque solo cumple uno de los criterios, no ambos.

### Motivación

Los operadores lógicos nos ayudan a simplificar los flujos y a evaluar condiciones más complejas, en este capítulo aprenderemos a utilizarlos.

### Operadores lógicos

Operator	Nombre	Ejemplo	Resultado
&	y (and)	False && True	Devuelve True <b>si ambos operandos son true</b> , en este ejemplo se devuelve False.
&#124;	o (or)	False &#124;&#124; True	Devuelve True <b>si al menos uno de los operando es true</b> , en este ejemplo devuelve True.

Operator	Nombre	Ejemplo	Resultado
!	no (not)	!False	Devuelve lo opuesto al resultado de la evaluación, en este ejemplo devuelve True.

Observemos los siguiente ejemplos:

```
nombre = 'Carlos'
apellido = 'Santana'

nombre == 'Carlos' and apellido == 'Santana'
# True

nombre == 'Carlos' and apellido == 'Vives'
# False

nombre == 'Carlos' or apellido == 'Vives'
# True
```

## Identities

Hay varias formas de expresar una afirmación en español, de la misma forma sucede en la lógica y en la programación. Por lo mismo hablamos de identidades.

Veamos ejemplos de esto:

## Igual es lo mismo que no distinto

Negar algo dos veces es afirmarlo (esto no siempre es así en español, pero en programación siempre es así).

Por lo mismo estas dos afirmaciones son equivalentes.

```
a = 18
print(a == 18) # True
print(not (a != 18)) # True
```

Son identidades porque para cualquier valor de a ambas expresiones siempre serán iguales, prueba cambiando el valor asignado.

```
a = 17
print(a == 18) # False
print(not(a != 18)) # False
```

Veamos otro ejemplo:



## Mayor y no menor igual

Un caso similar es la comparación  $a > 18$ . Decir que  $a$  no es mayor a 18, es decir que menor o igual a 18, (debemos incluir el 18 al negar)

```
a = 18
print(a > 18) # False
print(!(a <= 18)) # False
```

```
a = 19
print(a > 18) # True
print(!(a <= 18)) # True
```

## Resumen del capítulo:

- Operadores lógicos: Son importante porque nos ayudan a determinar si una expresión es cierta o falsa (la base de la programación).
- Los operadores lógicos nos pueden ayudar a simplificar expresiones.
- Trataremos de escribir siempre las condiciones en positivo.
- $(a > b)$  no es lo mismo que  $(a <= b)$ .
- $(a == b)$  no es lo mismo que  $(a != b)$ .

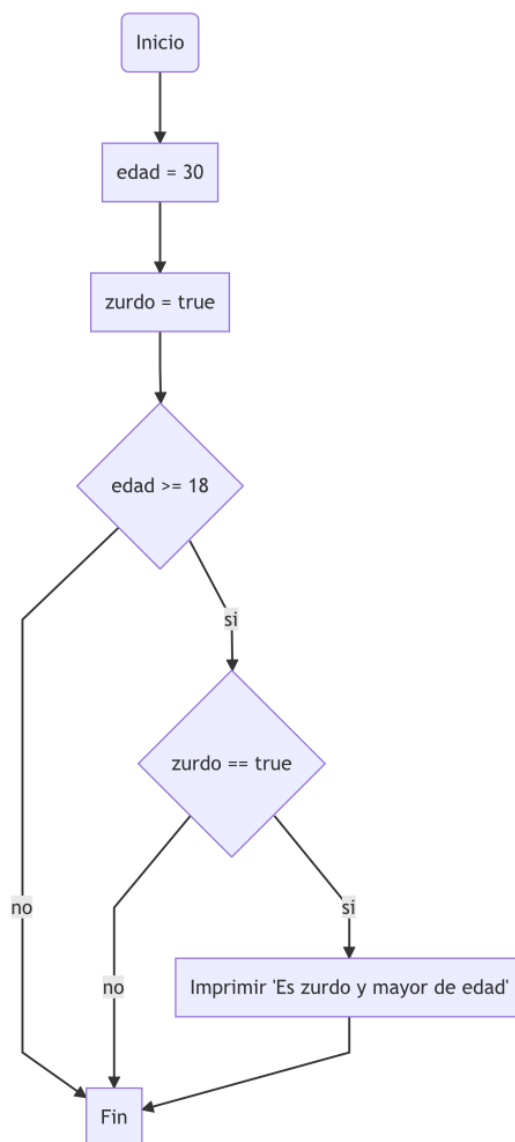
## Capítulo 13: Simplificando IFs anidados

### Objetivos:

- Simplificar problemas con condiciones anidadas en Python.
- Entender complicaciones en ifs anidados en Python.

### Identificando ifs anidados

Analicemos el siguiente ejemplo:



```
edad = 30
zurdo = True

if edad >= 30:
    if zurdo is True:
        print("Es zurdo y mayor de edad")
```

Es zurdo y mayor de edad

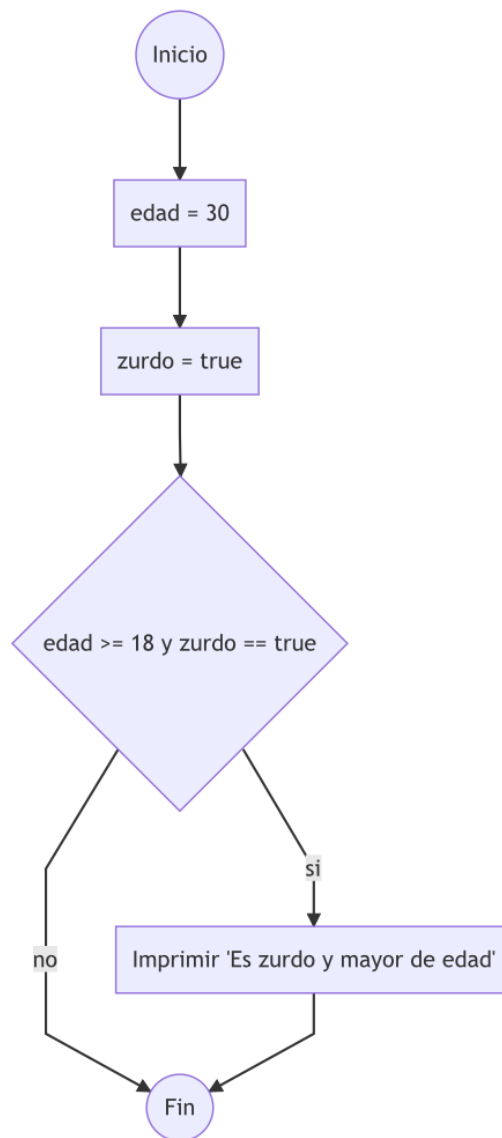
En el ejemplo anterior vemos un if dentro de otro if, a esto se le llama tener if anidados.

El código escrito muestra el texto sólo si se cumplen las dos condiciones.

## Operadores lógicos

Podemos simplificar el código para evaluar ambas condiciones en una misma instrucción `if`. Para una correcta utilización de instrucciones condicionales múltiples, debemos recordar la tabla de operadores lógicos:

¿Qué operador podemos utilizar si necesitamos que **ambas condiciones sean verdaderas**?



```
edad = 30
zurdo = True
```

```
if edad >= 18 and zurdo is True:
    print("Es mayor de edad y zurdo")
```

Es mayor de edad y zurdo

## Ejercicio de integración

Se busca crear un programa que solicite al usuario ingresar tres números. El programa debe determinar el mayor de ellos. Se asume que los números ingresados serán distintos.

¿Cómo lo podemos resolver?

Si bien es posible utilizar ifs anidados, vamos a utilizar lo aprendido para resolverlo de forma inteligente.

Opción 1: El **número 1** es mayor o igual que el **número 2** y mayor o igual que el **número 3**

Opción 2: El número

```
a = int(input('Ingresa primer número:\n'))
b = int(input('Ingresa segundo número:\n'))
c = int(input('Ingresa tercer número:\n'))

if a >= b and a >= c:
    print("a es el mayor")
elif b >= c:
    print("b es el mayor")
else:
    print("c es el mayor")
```

c es el mayor

# Capítulo 14: Simplificando el flujo

---

## Objetivos

- Simplificar el código de un script en Python utilizando buenas prácticas y la regla de condición en positivo.
- Refactorizar flujo condicional en Python.

## Refactor

Cuando comenzamos a operar con condicionales es común que, en algunos casos, caigamos en redundancias innecesarias. Analicemos el siguiente ejemplo:

```
mayor_de_edad = True
zurdo = False

if mayor_de_edad is True:
    if zurdo is True:
        print("Mayor de edad y zurdo!")
    else:
        print("Mayor de edad pero no zurdo!")
else:
    if zurdo is True:
        print("Menor de edad y zurdo")
    else:
        print("Menor de edad pero no zurdo!")
```

Reemplacemos los `if` anidados por condiciones múltiples:

```
if mayor_de_edad is True and zurdo is True:
    print("Mayor de edad y zurdo")
elif mayor_de_edad is True and zurdo is False:
    print("Mayor de edad pero no zurdo")
elif mayor_de_edad is False and zurdo is True:
    print("Menor de edad y zurdo")
else:
    print("Menor de edad y no zurdo")
```

Ahora podemos refactorizar las comparaciones en los condicionales.

Mencionamos con anterioridad que podemos tratar una variable como si fuera el objeto mismo.

Nuestras variables `mayor_de_edad` y `zurdo` son booleanas. Cuando evaluamos variables booleanas en un condicional no es necesaria la comparación.

## ¿Cómo?

Recordemos que la instrucción `if` espera que el resultado se evalúe como `true` o `false`. Por lo tanto:

```
mayor_de_edad = True

if mayor_de_edad is True:
    print("Mayor de edad")
```

Mayor de edad

Es lo mismo que:

```
mayor_de_edad = True

if mayor_de_edad:
    print("Mayor de edad")
```

Mayor de edad

La comparación `mayor_de_edad == True` es redundante ya que la variable por si sola se puede evaluar como `True` o `False`. Apliquemos esto en el ejemplo:

```
mayor_de_edad = True
zurdo = False

if mayor_de_edad and zurdo:
    print("Mayor de edad y zurdo")
elif mayor_de_edad and zurdo is False:
    print("Mayor de edad pero no zurdo")
elif mayor_de_edad is False and zurdo:
    print("Menor de edad y zurdo")
else:
    print("Menor de edad y no zurdo")
```

Mayor de edad pero no zurdo

## Capítulo 15: Análisis Léxico

---

La forma en que un computador lee nuestro código es similar a la manera en que aprendimos a analizar la estructura de una oración. Python genera una interpretación mediante un programa encargado de traducir cada una de nuestras instrucciones a un lenguaje que nuestro computador entiende.

Entender cómo Python interpreta nuestras expresiones al leer un programa nos facilita entender el por qué es necesario respetar estas normas sintácticas.

### ¿Qué sucede cuando escribimos `python programa.py`?

Al ejecutar esta instrucción, Python lee y procesa cada una de las líneas de código, permitiendo el cálculo de operaciones y subsecuentes resultados. Resulta que suceden muchos procesos para llegar a éste resultado.

No necesitamos saber estos procesos de memoria para programar en Python, pero comprender el flujo nos ayudará a entender de una mejor manera cómo funciona Python, sanitizar nuestro código y anticiparnos a posibles bugs.

Podemos segmentar el proceso de Python entres grandes pasos: *Tokenización*, *Lexing* y *Parsing*

### Etapa 1: Tokenización

Al leer código, Python busca separar todas las palabras que componen las instrucciones en unidades llamadas **tokens**. Si tenemos la instrucción `x = 10`, Python separará la expresieon en tres partes:

- `x`
- `=`
- `10`

### Etapa 2: Lexing

Python tiene una serie de definiciones y reglas para cada uno de los tokens. Mediante el proceso de **lexing**, Python imputa significado a cada componente para que posteriormente sepan cómo interactuar.

En este caso, la asignación `x = 10` funciona de la siguiente manera:

- Como `x` no está entre comillas y precede al operador de asignación `=`, se considera como una variable.
- Sabiendo que `x` se comporta como una variable, el operador de asignación busca en el lado derecho de la expresión qué imputar.
- En el lado derecho de la expresión se encuentra `10` que se asigna a la variable.

## Etapa 3: Parsing

La última etapa de la expresión `x = 10` consiste en generar un árbol de *sintaxis abstracta*, que busca genera una representación del código y ordenar la forma de resolucieon de las expresiones.

## Reglas Básicas

En Python existe una serie de reglas reglas básicas que se establecen a partir de los tokens. Éstos pueden tener algunas de las siguiente categorías léxicas:

- Identificadores.
- Operadores.
- Delimitadores.
- Literales.
- Comentarios.

## Identificadores

Cuando hablamos de identificadores estamos hablando de **nombres de variables y métodos**. Para que un identificar sea válido debe cumplir con ciertas características:

- Debe empezar con algún carácter alfabético (de la **a** a la **z**) o un guión bajo `_`. Puede incluir números siempre y cuando no sean el primer carácter.
- Los identificadores son sensibles a la mayúsculas.
- No se pueden empezar con números.

Si intentamos crear la variable `1dia`, Python avisará que la sintaxis no es válida

```
1dia = '24 horas'
```

```
File "<ipython-input-14-4e40c8852bb3>", line 1
```

```
1dia = '24 horas'
```

```
^
```

```
SyntaxError: invalid syntax
```

Si definimos una variable con la primera letra en mayúscula, y posteriormente la invocamos ignorándola, Python avisará que el nombre no se ha definido, porque no se encuentra en memoria.

```
Animal = "gato"
```

```
animal
```



-----

NameError

Traceback (most recent call last)

```
<ipython-input-15-ae420d7ae5e6> in <module>()
    1 Animal = "gato"
----> 2 animal
```

NameError: name 'animal' is not defined

## Operadores

Los operadores calculan un resultado en base a los valores de sus operandos. La mayoría de éstos componen de uno o dos caracteres ordinarios, pero en Python también existen algunos operandos lógicos que se escriben como palabras. Los operadores disponibles son:

`+`, `-`, `*`, `/`, `//`, `%`, `-`, `**`, `=`, `!=`, `<`, `>`, `<=`, `>=`, `&`, `|`, `~`, `^`, `<<`, `>>`, `and`, `in`, `is`, `not`, `or`.

## Literales

Los literales son los valores que se asignan a las variables para operar. Para almacenar y realizar manipulaciones con la información, se necesita definir el tipo de información que se almacenará y será posteriormente interpretada por Python. Ejemplos de literales incluyen a los tipos de datos `int`, `float`, `str`, `bool`.

## Delimitadores

Los delimitadores son símbolos que desempeñan tres funciones en Python: agrupar, separar objetos con puntuaciones, realizar asignación o concatenación de información a objetos. Ejemplos de delimitadores de agrupación y separación son `[]`, `{}`, `()`, `:`. Ejemplos de concatenadores incluyen `=`, `+=`, `--`.

## Comentarios

Los comentarios son tokens que serán ignorados, nos ayudarán a comentar el código y son de vital importancia para implementar documentación sobre el código.

```
# Esta línea es un comentario
# Los comentarios son ignorados

print(2 + 2) # También pueden existir contiguos a una línea de código válida

# print(2+ 3) (Esta línea también es ignorada)
```

4

## Palabras reservadas

Corresponden a ciertas palabras que poseen un significado predefinido en el lenguaje y por tanto **no se pueden utilizar para nombrar variables**. Podemos solicitar la lista de palabras reservadas en Python con:

```
import keyword
keyword.kwlist
```

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

```
'with',  
'yield']
```