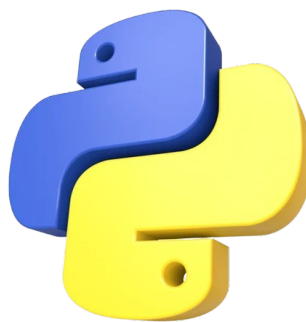


CRIPTOGRAFÍA CON PYTHON



Python básico

Esta obra está sujeta a la licencia Reconocimiento-CompartirIgual 4.0 Internacional de Creative Commons.

Para ver una copia de esta licencia, visitad
<https://creativecommons.org/licenses/by-sa/4.0/>.



Autor: Enrique Melchor Iborra Sanjaime (em.iborrasanjaime@edu.gva.es)

Contenido

1. El entorno de programación para Python.....	3
Python Shell (Interpretador Interactivo).....	3
IDLE, entorno de desarrollo de Python.....	4
Python desde bash.....	4
IPython: Un Shell Interactivo Mejorado.....	4
Jupyter Notebooks.....	5
Python en Entornos de Desarrollo Integrados (IDE).....	6
Uso de Python Interactivo desde Scripts.....	7
Uso de Python desde una plataforma web.....	7
2. El lenguaje de programación Python.....	8
2.1. Variables y Tipos de Datos.....	8
2.2. Lectura de datos por teclado.....	13
2.3. Operadores.....	14
Operadores sobre strings.....	15
2.4. Estructuras de Control.....	16
2.5. Funciones.....	18
2.6. Módulos y Paquetes en Python.....	22
2.7. Uso de __main__.....	25
3. Ejercicios.....	26
4. Python o Python3.....	28

1. El entorno de programación para Python

Python Shell (Interpretador Interactivo)

El **Python Shell** es el modo interactivo básico de Python. Cuando instalas Python, este viene con un entorno de consola interactiva que te permite ejecutar código Python línea por línea y obtener resultados inmediatos.

Cómo usar el Python Shell:

- **Abrir el Shell:**

En la terminal o línea de comandos (cmd o PowerShell en Windows, terminal en macOS/Linux), simplemente escribe:

```
$ python
```

Si estás utilizando Python 3, podrías necesitar escribir `python3` (en linux)

```
$ python3
```

- **Trabajar de manera interactiva:**

Después de ejecutar el comando anterior, verás algo como esto:

```
Python 3.x.x (default, ... )
[GCC x.x.x ...] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

En este prompt (`>>>`), puedes empezar a escribir código de Python directamente. Por ejemplo:

```
>>>2 + 3    (esto lo escribes tu, es una orden para python)
5           (esto lo escribe/interpreta python)
>>> print("Hola, mundo!")
Hola, mundo! (esto lo escribe/interpreta python)
```



El intro indica a python que ejecute la línea (orden), pero con un `\` podemos añadir mas líneas a una orden. Con esto podemos saltarnos la restricción de python de 79 caracteres por línea de ordenes.

```
>>>2 + 3 \
...*5      (este prompt (...) indica que espera mas información)
17         (esto lo escribe/interpreta python)
También podemos juntar ordenes en una línea
>>>a = 20 ; b = 14 ; c = a + b; print( c )
44
```

⚠ Cuidado con no dejar espacios en blanco al principio de una línea, a no ser que se quiera indentar (la indentación se explica mas adelante)

```
>>> 2+3
SyntaxError: unexpected indent
```

- **Salir del Shell:**

- Para salir del shell interactivo, puedes escribir `exit()` o presionar `Ctrl+D` (en macOS/Linux) o `Ctrl+Z` seguido de `Enter` (en Windows).

IDLE, entorno de desarrollo de Python

IDLE (Integrated DeveLopment Environment for Python) es un entorno gráfico de desarrollo elemental que permite editar y ejecutar programas en Python.

IDLE es también un entorno interactivo en el que se pueden ejecutar instrucciones sueltas de Python.

En Windows, IDLE se distribuye junto con el intérprete de Python, es decir, al instalar Python en Windows también se instala IDLE.

En Linux, IDLE se distribuye como una aplicación separada que se puede instalar desde los repositorios de cada distribución.

```
$ sudo apt install idle3      (debian, ubuntu o derivados)
```

Una vez termine, podremos lanzar este entorno desde el menú principal de linux. Muy parecido visualmente al Python shell, pero con GUI.

Python desde bash

Utilizando un editor del sistema operativo (nano, gedit, notepad(en windows), VSCode ...)

Editamos un fichero con extensión `.py` con el código dentro . **ejemplo.py**

Dentro de `ejemplo.py` escribimos

```
a = 5 ; b = 3 ; suma = a + b
print("La suma de", a, "y", b, "es:", suma)
```

Guardamos el fichero y ejecutamos el código del fichero desde la terminal

```
$ python ejemplo.py      ( o $ python3 ejemplo.py )      py ejemplo.py (en windows)
```

Otra forma de ejecutar un fichero python es indicando en su interior donde se encuentra el ejecutable

```
#!/usr/bin/python3
a = 5 ; b = 3 ; suma = a + b
print("La suma de", a, "y", b, "es:", suma)
```

Luego, le damos permisos al fichero de ejecución

```
$ chmod +x ejemplo.py
```

```
$ ./ejemplo.py
```

IPython: Un Shell Interactivo Mejorado

IPython es una versión mejorada del Python Shell que proporciona características adicionales como autocompletado, documentación en línea, historial de comandos y más.

Instalación de IPython:

Si no tienes **IPython** instalado, puedes instalarlo fácilmente utilizando `pip`:

```
$ pip3 install ipython      pip install ipython ( en windows )
```

****Recuerda cargar tu entorno virtual primero...**

Cómo usar IPython:

- **Abrir IPython:**

Después de la instalación, simplemente escribe `ipython` en la terminal:

```
$ ipython3
```

Verás algo como esto:

```
Python 3.x.x (default, ... )
Type "copyright", "credits" or "license" for more information.
IPython x.x.x -- An enhanced Interactive Python. Type '?' for help.
In [1]:
```

- **Características de IPython:**

Autocompletado: Puedes empezar a escribir una variable o función y presionar `Tab` para autocompletar.

Historial: Puedes usar las teclas de flecha hacia arriba y hacia abajo para navegar por los comandos previos.

Documentación rápida: Puedes acceder a la documentación de una función u objeto escribiendo el nombre seguido de `?`. Ejemplo:

```
In [1]: print?
```

- **Ejecucion de comandos del SO:** Puedes ejecutar comandos de sistema directamente desde IPython usando `!`

```
In [2]: !dir # Lista archivos en el directorio actual (equivalente a `ls` en Linux).
```

- **Magics.** IPython incluye **comandos mágicos** que comienzan con `%` (para comandos de línea) o `%%` (para comandos de celda).

- ```
In[3]: %timeit sum(range(1000)) # Mide el tiempo de ejecución
```



## Jupyter Notebooks

**Jupyter Notebooks** es una herramienta web interactiva que te permite escribir código Python, texto y ecuaciones en un solo documento, lo cual es ideal para exploración de datos, análisis y experimentación.

### Instalación de Jupyter Notebooks:

Si no tienes Jupyter instalado, puedes hacerlo utilizando `pip`:

```
$ pip3 install notebook
```

## Cómo usar Jupyter Notebooks:

### 1. Iniciar el servidor de Jupyter:

- En la terminal, escribe:

```
$ jupyter notebook
```

- Esto abrirá una página web en tu navegador, generalmente en `http://localhost:8888`, donde podrás crear, editar y ejecutar cuadernos interactivos de Python.

### 2. Ejecutar código en celdas:

- En Jupyter, el código se organiza en **celdas**. Puedes escribir código dentro de una celda y luego ejecutarlo presionando **Shift+Enter**.
- Puedes mezclar texto (en formato Markdown) y código dentro del mismo cuaderno, lo que lo hace ideal para la documentación de proyectos.

### 3. Otras características:

- **Visualización de gráficos:** Si trabajas con bibliotecas como `matplotlib` o `seaborn`, los gráficos se muestran directamente dentro del cuaderno.
- **Exportación:** Puedes exportar tus cuadernos como HTML o PDF.

---

## Python en Entornos de Desarrollo Integrados (IDE)

Muchos IDEs (Entornos de Desarrollo Integrados) también proporcionan una consola interactiva donde puedes ejecutar código Python de manera rápida, además de ofrecer características adicionales como depuración y autocompletado.

### Algunos IDEs populares:

- **PyCharm:** Ofrece un intérprete interactivo que te permite ejecutar código directamente desde el editor.
- **Visual Studio Code:** Tiene soporte para ejecutar scripts Python en la terminal integrada o en la ventana de la consola interactiva (usando el plugin de Python).
- **Thonny:** Un IDE sencillo para Python que incluye un shell interactivo fácil de usar.

### Ejemplo en Visual Studio Code:

1. Instala **Visual Studio Code** y el **plugin de Python**.
2. Abre un archivo Python y presiona **Ctrl+Shift+P** (Windows) o **Cmd+Shift+P** (Mac) y escribe "Python: Start REPL" para abrir una consola interactiva dentro de VS Code.
3. Puedes ejecutar código en la consola interactiva y ver los resultados al instante.

## Uso de Python Interactivo desde Scripts

Si prefieres escribir tu código en archivos `.py` pero ejecutar interactivamente ciertas secciones del mismo, puedes usar la función `code.InteractiveConsole` para emular un entorno interactivo desde un script de Python.

### Ejemplo de uso de `InteractiveConsole`:

```
$ import code

Crear una consola interactiva
console = code.InteractiveConsole()

Ejecutar código en la consola
console.push('x = 10')
console.push('print(x * 2)')
```

Este enfoque es útil cuando deseas crear un entorno interactivo dentro de una aplicación o script Python.

## Uso de Python desde una plataforma web

Existen plataformas web desde las que se puede practicar código python directamente escribiendo en el navegador.

Ejemplo ( sin registro )

<https://www.w3schools.com/python/>

<https://www.online-python.com/>

<https://www.programiz.com/python-programming/online-compiler/>



Ejemplos (con registro)

<https://replit.com/languages/python3>

<https://colab.research.google.com/>

## 2. El lenguaje de programación Python

Primero de todo, vamos a hacernos con una cheat sheet.

Un **cheat sheet** (hoja de referencia rápida) en el contexto de un lenguaje de programación es un documento conciso y bien organizado que resume las características, comandos, funciones y conceptos clave del lenguaje. Está diseñado como una guía práctica para programadores, ayudándoles a recordar rápidamente sintaxis, métodos y estructuras comunes sin necesidad de consultar documentación extensa. Podemos encontrar un ejemplo en el siguiente link.

<https://perso.limsi.fr/pointal/media/python:cours:mementopython3-english.pdf>

Y por supuesto, en la documentación oficial del python <https://docs.python.org/es/3.12/>

### 2.1. Variables y Tipos de Datos

Para dar nuestros primeros pasos en Python, es mas recomendable utilizar Python desde Shell o IPython, (o IDLE en Windows) , por su rápida respuesta a comandos sencillos.

En Python, no es necesario declarar el tipo de una variable. Python lo infiere automáticamente. Las variables se crean cuando se realiza su primera asignación. --Entra en Python y practica--

```
x = 10 # Número entero (integer)
nombre = "Ana" # Cadena de texto (String)
nombre2 = 'Pedro' # string con comilla simple
precio = 15.5 # Número decimal (float)
es_activo = True # Booleano (True o False)
b=None # Valor nulo, ausencia de valor
a = b = c = 100 # Asignación múltiple, un solo valor
userAge, userName = 30, 'Pedro' # Asignación múltiple, múltiples valores
a, b = 23, 67 # Asignación múltiple
a, b = b, a # Intercambio de valores (sin utilizar variable temporal)
```



Se evita el uso de variable temporal para realizar el intercambio

```
temp=a
a=b
b=temp
```

```
type(x) # Nos dice el tipo actual de la variable
del v o del(v) # Deja la variable libre, sin tipo no valor
```

```
$ python3
>>>x = 10
>>>x
10
>>>type(x)
<class 'int'>
```

Con `del(a)` se deja de usar una variable. (También con `del a` )

- ⚠ Los nombres de variables (o identificadores) son case sensitive. `username`  $\neq$  `userName`
- ⚠ Solo pueden contener letras May, letras min,números(dígitos) y `_` (y no pueden empezar con numero). A partir de Python 3, se pueden usar letras internacionales, como  $\alpha$ ,  $\beta$ , ñ, é,  $\pi$ , 変 ...
- ⚠ No pueden ser **palabras reservadas** de python

\*\*\*Prueba a equivocarte intencionadamente y familiarízate con los errores que transmite

```
Python
>>> else = 9
>>> 5var = 67
```





## Palabras reservadas de python

```
import keyword
print(keyword.kwlist)
```

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

En la parte izquierda de la asignación se indica la variable en la que se almacenara el valor. En la parte derecha de la asignación se indicará un literal que será interpretado por python. Los literales son los datos simples que Python es capaz de manejar: Numeros o Cadenas de texto.

Números: enteros, decimales y complejos, en notación decimal, octal o hexadecimal.

Cadenas : entre comillas simples o dobles

Valores lógicos ( True False ) y Valor nulo ( None )

## Comentarios en el código de python

Los comentarios en Python sirven para **añadir notas o explicaciones en el código** que no son ejecutadas por el intérprete. Son útiles para hacer el código más entendible tanto para ti como para otras personas que lo lean.

Hay dos formas de poner comentarios. En una línea o multilinea

# Esto es un comentario de una línea, puede ir también detrás de código

""" Esto es un comentario

de varias líneas : Comentario Multilinea """

''' con tres comillas simples

también se crean comentarios de varias líneas '''

Algunos usos de los comentarios son:

- Explicar el propósito del código

- Anotar recordatorios o tareas pendientes (TODOs)

- Desactivar partes del código temporalmente

- Documentar el código ( como funciona)

- Ayudar en la depuración

## Función print

La función **print()** en Python es una de las más utilizadas y sirve para mostrar mensajes, datos o cualquier salida en la consola. Es muy versátil y tiene varias opciones que permiten personalizar cómo y qué se imprime.

Por defecto, la función `print()` agrega un salto de línea (`\n`) al final de cada impresión.

```
>>> a = 3332 + 23223 * 23
```

```
>>> print(a)
```

```
>>> print(a + 23334)
```

```
>>> print("Hola, Mundo!")
```



La función `print` cobra mas sentido al ser utilizada en un entorno 'no interactivo' como en programas `.py` ejecutados directamente desde la línea de comandos del sistema operativo.

Por defecto, la función `print()` agrega un salto de línea (`\n`) al final de cada impresión. Puedes cambiar este comportamiento usando el argumento `end`.

```
Imprimir en la misma línea
print("Hola", end=" ")
print("Mundo", end="!")
```

\*Sin salto de línea (vacío en `end`)

```
for i in range(5):
 print(i, end="") # No agrega salto de línea
```



\*Insertar un salto de línea con `\n`

```
print("Hola\nMundo")
```

Otras consideraciones sobre `print`

Sintaxis de `print()`

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

\*objects:

- 4. Puede ser uno o varios objetos que deseas imprimir.(separados por comas (,))
- 5. Los valores serán convertidos a cadenas automáticamente (usando la función `str()`).

`sep` (opcional):

- Define el separador entre los objetos.
- Por defecto, es un espacio (' ').

`end` (opcional):

- Especifica lo que se imprimirá al final de la salida.
- Por defecto, es un salto de línea ('\n').

`file` (opcional):

- Indica el flujo donde se enviará la salida.
- Por defecto, es la consola estándar (`sys.stdout`).
- También puedes redirigir la salida a un archivo.

```
with open("salida.txt", "w") as archivo:
 print("Hola Mundo", file=archivo)
```

`flush` (opcional):

- Si se establece como `True`, vacía (`flush`) el búfer de salida inmediatamente.
- Es útil en programas que requieren que los datos se impriman en tiempo real.

Ejemplos

```
txt=['a', 'b', 'c'] #esto es una lista, se ve mas adelante
print(txt) # ['a' 'b' 'c']
print(*txt) # a b c
print(*txt, sep=' -->') # a --> b --> c
```



```
import json
print(json.dumps(mi_diccionario, indent=4))
```

## Literales numéricos

Podemos representar números enteros en representación decimal, ej. 345

binaria (anteponiendo un 0b), ej. 0b101001001

octal (anteponiendo un 0o) ej. 0o546

o hexadecimal (anteponiendo un 0x). ej. 0xf9af9d.

Podemos representar números reales (con decimales) separando la parte entera de la parte decimal con un punto "." ej. 123.50 0.56 .56 12. 12.00

o con notación científica : ej. 1.2e3 5e-2

También podemos representar números complejos (con parte real y parte imaginaria) acabando el número con una "j" o una "J", indistintamente. ej. 1+2j 8+15J

En Python todos los elementos son objetos, incluso los literales. Así que cuando vamos a utilizar un literal, Python crea un objeto al que asocia un identificador único, un tipo y un valor. El identificador de un elemento/objeto se puede obtener utilizando la función predefinida `id(nom_obj)`

## Literales Strings o Cadenas

Los literales de tipo string van delimitados con comillas simples o dobles ( o triples)

En Python las comillas dobles y las comillas simples son completamente equivalentes, pero en otros lenguajes de programación no lo son.

```
>>> print("Esto es una cadena") #OK
>>> print('Esto es una cadena') #OK
```



Si utilizamos las comillas simples para delimitar, la cadena debe empezar y acabar con este mismo delimitador (no con el otro)

```
>>> print("Esto es una cadena') #dará error
```

y dentro no se puede utilizar este delimitador como carácter de la cadena.

```
>>> print("Esto "es una cadena") #dará error
```

Las comillas triples sirven para "romper" el límite de 79 caracteres por línea

```
>>> print("""Esto es una cadena
... que ocupa
... varias líneas""")
```

También se puede romper con \

```
>>> print("Esto es una cadena \
... que ocupa \
... varias líneas")
```



\*\*\* Aunque en estos casos, el resultado es una cadena seguida, sin saltos de línea

Si se quiere poner unas comillas del mismo tipo que el delimitador dentro de la cadena, se debe utilizar un carácter especial.

Caracteres especiales dentro de strings \ " \' \t (tabulador) \n (salto de línea)  
\\ la barra \v (tabulador vertical)

Python tiene varios mecanismos para insertar valores de variables dentro de cadenas.

**Formateo de strings con % (antiguo)** Parecido al lenguaje C

```
brand="HP"
exchangeRate=1.334
message = 'The price of this %s laptop is %d USD and the exchange rate is %4.2f
USD to 1 EUR' %(brand, 1299, exchangeRate)
```

**Formateo de strings con .format()**

```
message = 'The price of this {0:s} laptop is {1:d} USD and the exchange rate is
{2:4.2f} USD to 1 EUR'.format('Apple', 1299, 1.235235245)
0 es el primer argumento de .format(), 1 el segundo,...
```

**Uso de cadenas f-strings (a partir de python 3.6)**

```
print(f"Mi nombre es {nombre} y tengo {edad} años.")
print(f"La suma de {a} y {b} es {a + b}.") # permiten cálculos
print(f" {edad=} ") # Salida edad=25 sirve para depurar código
print(f"El valor de pi con 3 decimales es {pi:.3f}.") # Salida: 3.142
print(f"Porcentaje: {0.25:.2%}") # Salida: 25.00%
```



```
print(f"|{nombre:<10}|") # Alineado a la izquierda
print(f"|{nombre:>10}|") # Alineado a la derecha
print(f"|{nombre:_>10}|") # Alineado a la derecha con relleno de "_"
print(f"|{nombre:^10}|") # Centrado
print(f"|{nombre:*^10}|") # Centrado con relleno de "*"
print(f"Decimal: {n}, Binario: {n:b}, Hexadecimal: {n:x}, Octal: {n:o}")
print(f"Con separador de miles: {monto:,.2f}")
print(f"{numero:+d}") # Signo explícito: -42
print(f"{42:06d}") # Relleno con ceros: 000042
```



En Python, una letra o un prefijo que aparece antes de una cadena, como r, f, b, o u, indica que la cadena tiene un propósito o comportamiento especial

**r o R (raw string):**

- Indica una cadena "cruda" en la que los caracteres de escape como \n o \t no se interpretan como nuevas líneas o tabulaciones, sino como texto literal.
- Útil para trabajar con expresiones regulares o rutas de archivos.

```
raw_string = r"C:\Users\John\Documents"
print(raw_string) # Salida: C:\Users\John\Documents
```

**f o F (formatted string):**

2. Introduce una **f-string**, que permite incrustar expresiones o variables directamente en la cadena usando llaves {}

**b o B (byte string):**

- Indica una cadena de bytes en lugar de una cadena de texto. Esto es útil para trabajar con datos binarios.
- Las cadenas de bytes están codificadas en ASCII y no admiten caracteres Unicode directamente.

```
byte_string = b"Hola"
print(byte_string) # Salida: b'Hola'
```

u o U (Unicode string):

4. Usado en versiones antiguas de Python (antes de la 3.0) para indicar cadenas Unicode. En Python 3, todas las cadenas son Unicode por defecto, por lo que este prefijo es redundante.

Combinaciones de prefijos:

- Puedes combinar prefijos, como `r f` para una cadena cruda y formateada.

```
path = "Users"
raw_formatted_string = rf"C:\{path}\John"
print(raw_formatted_string) # Salida: C:\Users\John
```

## 2.2. Lectura de datos por teclado

En cualquier lenguaje de programación es esencial la entrada de datos a un programa para que este pueda reaccionar de una manera u otra, tomado estos como base para cálculos o como parámetros de actuación, realizando una acción u otra dependiendo de estos parámetros.

Usamos `input` para requerir al usuario que ejecuta un programa Python la introducción de datos al mismo para que este los utilice en su ejecución. Veamos código..

```
numero1 = input("Dime el primer numero: ") # Aquí, numero1 será string
#Cuando el programa llega a este punto, se queda esperando a que el usuario introduzca un valor con el teclado
#Cuando el usuario teclea una secuencia de teclas y pulsa intro, esta secuencia se asigna a la variable numero1
numero1 = int(numero1) # si no se convierte, siempre sera string
print(numero1, "+", numero2, "=", numero1+numero2)
print ("{numero1} + {numero2} = ", numero1+numero2)
```

```
n1 = input("Dime el primer numero: ")
n1 = input("Dime el segundo numero: ")
print (" la suma es " , n1 + n2)
!!! que ???
print (" la suma es " , int(n1) + int(n2))
```



También se puede requerir una entrada de valores separados por comas o espacios.

```
entrada = input("Ingresa números separados por comas: ")
numeros = entrada.split(',')
Convertir los valores en enteros
numeros = [int(num.strip()) for num in numeros]
print("Lista de números:", numeros)
```



### Lectura de datos desde la llamada al programa Python

Otra manera de dar entrada de datos a un programa es utilizar la misma llamada al mismo junto con dichos datos en la misma línea. En Python, puedes pasar parámetros a un programa desde la línea de comandos utilizando el módulo `sys` o `argparse`. Estas herramientas permiten que tu programa reciba argumentos externos, lo cual es útil para personalizar la ejecución sin modificar el código

```
*** Pasar parámetros con sys. Ejemplo básico con sys.argv:
archivo: programa.py
import sys
```

```
Mostrar argumentos
```



```
print("Argumentos recibidos:", sys.argv)

Acceder a los argumentos individuales
if len(sys.argv) > 1:
 print("Primer argumento:", sys.argv[1])
else:
 print("No se proporcionaron argumentos.")
```

Ejecución en la terminal:

```
$ python3 programa.py hola mundo
```

### \*\*\* Pasar parámetros con argparse

El módulo **argparse** ofrece una forma más robusta y flexible para gestionar argumentos. Te permite definir opciones y descripciones, verificar tipos y manejar errores automáticamente.

```
archivo: programa.py
import argparse

Crear un objeto ArgumentParser
parser = argparse.ArgumentParser(description="Un programa de ejemplo para
recibir parámetros.")

Definir los argumentos que se aceptan
parser.add_argument('nombre', help="Tu nombre") # Argumento posicional
parser.add_argument('--edad', type=int, help="Tu edad (opcional)") # Argumento
opcional

Parsear los argumentos
args = parser.parse_args()

Usar los argumentos
print(f"Hola, {args.nombre}!")
if args.edad:
 print(f"Tienes {args.edad} años.")
```

Ejecución en la terminal:

```
$ python3 programa.py Juan --edad 25
```

## 2.3. Operadores

- Aritméticos:

|                           |                                  |
|---------------------------|----------------------------------|
| suma = 10 + 5             | # 15                             |
| resta = 10 - 5            | # 5                              |
| multiplicación = 10 * 5   | # 50                             |
| división = 10 / 5         | # 2.0 NO ENTERO, decimal o float |
| division_entera = 10 // 3 | # 3 (entero)                     |
| modulo = 10 % 3           | # 1 (resto)                      |
| exponente = 2 ** 3        | # 8 (2 elevado a 3) entero       |
| exponente 2.2**3          | # 10.64800 float                 |
| exponente 64 **0.5        | # 8.0 float raiz cuadrada        |



- Asignación

|      |            |    |    |       |                                   |    |     |
|------|------------|----|----|-------|-----------------------------------|----|-----|
| =    | +=         | -= | *= | /=    | //=                               | %= | **= |
| x+=1 | equivale a |    |    | x=x+1 | En python no existe el x++ ni x-- |    |     |

- Comparación:

```
5 == 5 # True (igual) !! el == no es lo mismo que el = !!
5 != 3 # True (diferente)
5 < 10 # True (menor que)
5 > 10 # False (mayor que)
5 <= 10 # True (menor o igual que)
5 >= 10 # False (mayor o igual que)
```

- Lógicos:

```
True and False # False
True or False # True
not True # False
```

- Identidad (hacen referencia al mismo objeto de memoria)

```
a is b
a is not b
```

- Pertenencia o membresía

```
'a' in 'hola'
'a' not in 'hola'
```

- A nivel de bits (Bitwise)

```
5 & 3 # AND bit a bit
5 | 3 # OR bit a bit
5 ~ 3 # NOT bit a bit
5 ^ 3 # XOR bit a bit
5 << 3 # desplazamiento a la izquierda bit a bit
5 >> 3 # desplazamiento a la derecha bit a bit
```

\*\*\* En números decimales, debido a la representación interna de los mismos, podemos obtener resultados 'poco precisos'. Veamos un ejemplo.

```
>>>a=0.1
>>>b=0.2
>>>a+b
0.30000000000000004
```



No pasa en todas las operaciones pero hay que tenerlo presente a la hora de controlar esta casuística

Se puede comparar con

```
import math
math.isclose(0.1+0.2,0.3)
True
```

## Operadores sobre strings

**Definir strings:** Usa comillas simples ( ' ), dobles ( " ) o triples ( ' ' ' o " " " ).

### Concatenación:

```
saludo = "Hola" + " " + "mundo"
print(saludo) # Hola mundo
```

### Repetición:

```
print("Hola" * 3) # HolaHolaHola
```

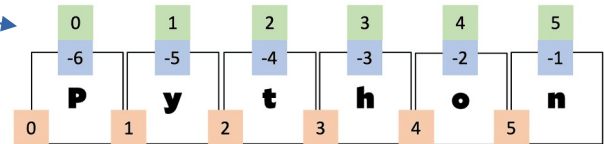


**Comparación:**

```
print("Hola" == "hola") # False (distinción de mayúsculas/minúsculas)
print("a" < "b") # True (orden alfabético) tambien > !=
```

**Acceso por índice:**

```
texto = "Python"
print(texto[0]) # P
print(texto[-1]) # n (último carácter)
```

**Acceso por slicing (o rebanado):**

```
print(texto[0:4]) # Pyth (índices 0 a 3)
print(texto[::2]) # Pto (caracteres del 0 al 5 de 2 en 2)
print(texto[1:4:2]) # yh (caracteres 1 al 3 de 2 en 2)
print(texto[:3]) # Hol (del 0 al 2)
```

**Iteración sobre strings**

Puedes recorrer una cadena carácter por carácter usando un bucle for:

```
texto = "Python"
for letra in texto:
 print(letra)
```

**Inmutabilidad de strings**

```
>>> texto[0]='J'
Traceback (most recent call last):
 File "<pyshell#2>", line 1, in <module>
 texto[0]='J'
TypeError: 'str' object does not support item assignment
```

## 2.4. Estructuras de Control

**Condicionales (if, elif, else)**

Permiten ejecutar bloques de código según condiciones.

el operador **:** **marca el inicio de un bloque de código estructurado e indentado**, y es esencial para que Python interprete correctamente la jerarquía del programa (el indentado se indica con 4 espacios (min) o un tabulador).

Esto es fundamental en Python porque **la indentación define la estructura del programa**, a diferencia de otros lenguajes que usan llaves {}. Se utilizará en ( if, for, while, try, def, class )

```
edad = 20
if edad >= 18:
 print("Eres mayor de edad") # esta orden esta indentada
 print("=====") # esta orden esta indentada, las dos ordenes
 # forman un bloque
else:
 print("Eres menor de edad") # esta orden esta indentada
```

Si la estructura solo tiene **una línea**, no hace falta indentar (con indentacion también funciona y es lo habitual)

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





Este es un buen momento para presentar la guía de estilos de python. Hay un resumen traducido en <https://recursospython.com/pep8es.pdf>. Python Enhancement Proposals -PEP- Es un documento oficial que establece las mejores prácticas para que el código sea más **legible, consistente y mantenible**.

### Principales reglas de PEP 8:

**Indentación:** Usar 4 espacios por nivel de indentación (no tabulaciones).

**Líneas de código:** Máximo 79 caracteres por línea.

**Espacios en blanco:**

- Uno antes y después de los operadores (`a = b + c`).
- No espacios dentro de paréntesis (`func(a, b)`).

**Nombres de variables y funciones:**

- **Variables y funciones:** `snake_case` (ejemplo: `mi_variable`).
- **Clases:** `CamelCase` (ejemplo: `MiClase`).

**Importaciones:**

- Cada importación en una línea (`import os`).
- Primero las librerías estándar, luego las de terceros, y al final las del proyecto.

**Comentarios:** Deben ser claros y concisos, preferiblemente en inglés.

### Condicional ternario

```
mensaje = "Mayor de edad" if edad >= 18 else "Menor de edad"
```

Asigna una de las dos cadenas a la variable mensaje

### Bucles (for, while)

- **for:** Itera sobre una secuencia (como una lista).

```
for i in range(5): # Desde 0 hasta 4
 print(i)

for i in range(2,5): # Desde 2 hasta 4
for i in range(2,10,2): # Desde 2 hasta 8 con paso 2

for i in frutas: # frutas es una lista
break # sale del bucle
continue # vuelve al principio del bucle
else # se ejecuta si el bucle no termina con un break
```



- **while:** Ejecuta mientras una condición sea verdadera.

```
contador = 0
while contador < 5:
 print(f"Contador: {contador}")
 contador += 1 # Aumenta el contador
else:
 print("el bucle ha terminado con contador >=5 ")
```



## Manejador de errores

Usamos `try` y `except` para capturar y manejar errores.

```
try:
 resultado = 10 / 0
except ZeroDivisionError:
 print("Error: No se puede dividir entre cero.")
else:
 print("La operación ha sido correcta")
finally:
 print("Esta parte se ejecuta siempre")
```



## Estructura de control de flujo: `pass`

El `pass` se utiliza como un marcador de lugar. Se puede usar cuando se requiere una sintaxis válida, pero no se quiere que el código haga nada en esa parte.

```
if condición:
 pass # No hace nada, pero se puede dejar como espacio para agregar código más tarde
```

## 2.5. Funciones

En Python, al igual que en otros lenguajes de programación, se pueden utilizar funciones (ya incorporadas en el lenguaje o Built-In) para realizar cálculos matemáticos, conversiones de tipos, obtención de información del sistema, etc.

### Conversión de tipos

`int()` `float()` `str()` `bool()` `bin()` `chr()` `ord()`

### Funciones útiles con variables

`type(x)` `id(x)` `isinstance(x,int)` `globals()` `locals()` `callable(nomfunc)`

### Funciones matemáticas

`abs(x)` `round(x)` `max(numeros)` `min(numeros)` `sum(numeros)` `divmod(a,b)`

y algunas mas.... <https://docs.python.org/es/3.13/library/functions.html>

### Funciones para cadenas ( Built-In String Functions)

`'pedro'.upper()` `v.upper()` ~~`v2=upper(v)`~~ pero... `len('sdfs')` ~~`v=len()`~~ `str(335)`  
`lower`, `capitalize`, `title`, `strip`, `replace`, `split`, `join`, `find`, `index`, `startswith`,  
`endswith`, `isalpha`, `isdigit`, `isalnum`, `isspace`, `format`, `count`, `zfill`, `center`,  
`ljust`, `rjust`,

### Funcion `eval()`

Devuelve la interpretación de una cadena  
`eval('x+1')`


### Funciones definidas por el usuario

Además de las funciones incorporadas en el lenguaje, Python permite la definición de funciones nuevas, o también llamadas funciones definidas por el usuario.

En Python, las funciones definidas por el usuario son bloques de código reutilizables que realizan una tarea específica. Estas funciones se crean utilizando la palabra clave `def`, seguida del nombre de la función y paréntesis que pueden incluir parámetros. El cuerpo de la función está indentado y puede incluir declaraciones, cálculos y una declaración `return` para devolver un valor.

Estructura básica de una función:

```
def nombre_de_la_funcion(parametro1, parametro2, ...):
 # Cuerpo de la función
 # Realiza alguna operación
 return resultado # Opcional
```



Partes de una función:

- Nombre de la función: Identificador único para llamar a la función.
- Parámetros: Valores que se pasan a la función para que los use (opcional).
- : Los dos puntos indican que empieza el bloque de código **indentado**
- Cuerpo de la función: Bloque de código que realiza la tarea. (**indentado**)
- Valor de retorno: Resultado que la función devuelve (opcional). Si no se especifica `return`, la función devuelve `None`.

**Indentación:** La indentación se refiere a la técnica de agregar un/os espacio/s inicial (indentado) al principio de las líneas de código, que ayuda a delimitar visualmente los bloques y estructuras de control.

Ejemplos:

```
def saludar(nombre):
 print(f"Hola, {nombre}!")
```

```
saludar("Carlos")
```

```
def sumar(a, b):
 return a + b
```

```
def saludar(nombre="Amigo"):
 print(f"¡Hola, {nombre}!")
```

```
def dividir(a: float, b: float) -> float: # type hint
```

```
def sumar_todos(*numeros): # varios argumentos pasados con comas
```

```
def mostrar_info(**info): # varios argumentos pasados con clave-valor
```

```
multiplicar = lambda x, y: x * y # funciones lambda (siguiente tema)
@decoradores (siguiente tema)
```



## Ámbito o Scope

En Python, el ámbito de una variable (también conocido como `scope`) se refiere a la región del código donde una variable es accesible. Python tiene reglas específicas para determinar dónde una variable puede ser usada o modificada. Los ámbitos más comunes son:

### Ámbito global:

Una variable definida fuera de todas las funciones o bloques de código tiene un ámbito global.

Puede ser accedida desde cualquier parte del código, incluyendo dentro de funciones.

Si necesitas modificar una variable global dentro de una función, debes usar la palabra clave global.

Ejemplo:

```
x = 10 # Variable global
def mostrar_x():
 print(x) # Acceso a la variable global
mostrar_x() # Salida: 10
```

Si intentas modificar una variable global dentro de una función sin usar global, Python creará una nueva variable local en su lugar:

```
x = 10
def modificar_x():
 x = 20 # Esto crea una nueva variable local, no modifica la global
 print(x) # Salida: 20
```

```
modificar_x()
print(x) # Salida: 10 (la variable global no cambió)
```



```
x = 10
def modificar_x():
 x = x + 1 # Esto dará un error cuando se llame a la función
 print(x)
```

```
modificar_x():
```

**UnboundLocalError: cannot access local variable 'x' where it is not associated with a value**

Para modificar la variable global (desde dentro de la función)

```
x = 10
def modificar_x():
 global x
 x = 20 # Modifica la variable global
 print(x) # Salida: 20
modificar_x()
print(x) # Salida: 20 (la variable global cambió)
```

### Ámbito local:

Una variable definida dentro de una función tiene un ámbito local.

Solo puede ser accedida dentro de esa función.

Si intentas acceder a una variable local fuera de su función, obtendrás un error.

Ejemplo:

```
def funcion():
 y = 5 # Variable local
 print(y) # Salida: 5
funcion()
print(y) # Error: NameError, 'y' no está definida en el ámbito global
```

### Ámbito no local (nonlocal):

Este ámbito se aplica a variables definidas en una función externa (pero no global) y que son accedidas desde una función interna (anidada).

Para modificar una variable de una función externa dentro de una función interna, se usa la palabra clave `nonlocal`.

Ejemplo:

```
def funcion_externa():
 z = 15 # Variable no local
 def funcion_interna():
 nonlocal z
 z = 25 # Modifica la variable de la función externa
 print(z) # Salida: 25
 funcion_interna()
 print(z) # Salida: 25 (la variable no local cambió)
funcion_externa()
```

### Ámbito de bucle o bloque:

En Python, las variables definidas dentro de un bucle (como `for` o `while`) o un bloque (como `if`) tienen un ámbito local al bloque o función en el que están definidas.

Sin embargo, a diferencia de otros lenguajes, en Python las variables definidas en un bucle o bloque no están limitadas a ese bloque y pueden ser accedidas fuera de él.

Ejemplo:

```
for i in range(3):
 print(i) # Salida: 0, 1, 2
print(i) # Salida: 2 (la variable 'i' sigue accesible fuera del bucle)
```

Esto puede ser confuso si vienes de otros lenguajes donde las variables de bucle tienen un ámbito limitado al bloque.

### Buenas prácticas al definir funciones:

**Nombres descriptivos:** El nombre de la función debe indicar claramente su propósito.

**Comentarios:** Usa docstrings (""" ... """) para documentar la función.

```
def sumar(a, b):
 """
 Suma dos números y devuelve el resultado.
 :param a: Primer número
 :param b: Segundo número
 :return: La suma de a y b
 """
 return a + b
```

**Evitar efectos secundarios:** Una función debería hacer solo una cosa y no modificar variables globales innecesariamente.

**Mantén las funciones cortas:** Si una función es demasiado larga, considera dividirla en funciones más pequeñas.

### Uso del guion bajo (`_`)

En Python, el guion bajo (`_`) tiene varios usos dependiendo del contexto en el que se encuentre. Aquí algunos de los más comunes:

#### 1. Ignorar valores en asignaciones

Se usa como una variable "desechable" cuando no necesitas usar un valor.

```
_, y, z = (1, 2, 3)
print(y, z) # 2 3
```

## 2. Iteraciones cuando no se necesita el valor

En los bucles, se usa `for _ in range(n)` cuando no se necesita la variable del bucle.

```
for _ in range(3):
 print("Hola") # Se imprime "Hola" tres veces
```

## 3. Último resultado en la consola interactiva

En la terminal de Python, `_` almacena el último resultado calculado.

```
>>> 5 + 3
8
>>> _ * 2
16
```



## 4. Prefijo para variables privadas

Se usa `_variable` para indicar que una variable es "privada" (convención, no restricción real).

```
class MiClase:
 def __init__(self):
 self._secreta = 42 # Indica que es privada (pero aún accesible)
```

## 5. Doble guion bajo para evitar colisiones de nombres (name mangling)

Cuando una variable empieza con `__`, Python cambia su nombre internamente para evitar colisiones.

```
class Ejemplo:
 def __init__(self):
 self.__privada = 99

e = Ejemplo()
print(dir(e)) # Se convierte en _Ejemplo__privada
```

## 6. Usar `_` en nombres de funciones y variables para evitar conflictos

Se usa `_` en nombres para evitar conflictos con palabras clave de Python.

```
def _class_(nombre):
 return f"Clase: {nombre}"
```

## 2.6. Módulos y Paquetes en Python

Un **módulo** es simplemente un archivo de Python con extensión `.py` que contiene definiciones de funciones, clases y variables. Puedes reutilizar el código de un módulo importándolo en otros archivos o programas.

Crear un archivo Python (`mi_modulo.py`):

```
mi_modulo.py
def saludar(nombre):
 return f"Hola, {nombre}!"
PI = 3.14159
```

**Importar y utilizar el módulo en otro archivo:**

```
programa.py
import mi_modulo
print(mi_modulo.saludar("Juan")) # Output: Hola, Juan!
print(mi_modulo.PI) # Output: 3.14159
```

Cuando hemos importado un módulo y queremos utilizar algún elemento del mismo, debemos utilizar su *namespace* seguido de un punto y el nombre del elemento que queremos utilizar. El namespace no es más que el nombre que hemos indicado después del import.

**Formas de importar módulos:**

- **Importar todo el módulo:**

```
import mi_modulo # El namespace es => mi_modulo
print(mi_modulo.saludar("Ana")) # De esta forma se tiene que usar el namespace
```

- **Importar elementos específicos del módulo:** (sin namespace)

```
from mi_modulo import saludar
print(saludar("Carlos")) # De esta forma NO se tiene que usar el namespace
```

- **Importar TODOS los elementos del módulo:** (sin namespace)

```
from mi_modulo import *
print(saludar("Carlos")) # De esta forma NO se tiene que usar el namespace
```

- **Renombrar un módulo:**

```
import mi_modulo as mm
print(mm.PI) # De esta forma se tiene que usar el namespace
```

Un **paquete** es una colección de módulos organizados en un directorio. Sirve para estructurar proyectos grandes en jerarquías de submódulos.

**Cómo crear un paquete:**

1. Crea una carpeta para el paquete (por ejemplo, `mi_paquete`).
2. Dentro de la carpeta, añade un archivo especial llamado `__init__.py`. Este archivo puede estar vacío o contener código de inicialización para el paquete.

Estructura básica de un paquete:

```
mi_paquete/
 __init__.py
 modulo1.py
 modulo2.py
```

3. Ejemplo de módulos dentro del paquete:

- **modulo1.py:**

```
def sumar(a, b):
```

```
return a + b
```

- **modulo2.py:**

```
def restar(a, b):
 return a - b
```

#### 4. Usa el paquete en un programa:

```
programa.py
from mi_paquete.modulo1 import sumar
from mi_paquete.modulo2 import restar

print(sumar(5, 3)) # Output: 8
print(restar(5, 3)) # Output: 2
```

### Paquetes anidados:

Los paquetes pueden contener subpaquetes. Por ejemplo:

```
mi_paquete/
 __init__.py
 utilidades/
 __init__.py
 herramientas.py
```

Puedes importar submódulos así:

```
from mi_paquete.utilidades.herramientas import alguna_funcion
```

### Módulos y paquetes estándar de Python

Python incluye muchos **módulos estándar** listos para usar. Algunos ejemplos comunes son:

- **math:** Funciones matemáticas.

```
import math
print(math.sqrt(16)) # Output: 4.0
```



- **os:** Operaciones del sistema operativo.

```
import os
print(os.getcwd()) # Obtiene el directorio actual.
```

- **random:** Generación de números aleatorios.

```
import random
print(random.randint(1, 10)) # Número aleatorio entre 1 y 10.
```

### Instalación de paquetes externos



Para usar paquetes desarrollados por terceros, puedes instalarlos desde el repositorio **PyPI** usando **pip**:

```
pip3 install nombre_del_paquete
```

Ejemplo: instalar `requests` para realizar solicitudes HTTP:

```
import requests
response = requests.get("https://example.com")
print(response.status_code)
```

## 2.7. Uso de `__main__`

En Python, la construcción `if __name__ == "__main__":` es una forma estándar de asegurarse de que un bloque de código solo se ejecute cuando el archivo se ejecuta directamente, y no cuando se importa como un módulo en otro archivo.

**Desglose de su función:**

1. `__name__`: Es una variable especial en Python que se define automáticamente.
  - Si el archivo se ejecuta directamente, `__name__` toma el valor de `"__main__"`.
  - Si el archivo se importa como módulo, `__name__` toma el nombre del archivo (sin la extensión `.py`).
2. `if __name__ == "__main__":`
  - Este condicional evalúa si el script está siendo ejecutado directamente (no importado).
  - Si la condición es verdadera, el código dentro del bloque se ejecutará.

**¿Por qué es útil?**

Esto es especialmente útil para:

- Separar el código que debe ejecutarse solo cuando el archivo es el "principal" del programa (como pruebas, lógica principal o ejecución directa).
- Permitir que el archivo sea importado como módulo sin ejecutar automáticamente su código principal.

**Ejemplo práctico:**

Archivo: `mi_script.py`

```
def funcion_util():
 print("Esta función puede ser usada en otros módulos.")

if __name__ == "__main__":
 print("Este script se está ejecutando directamente.")
 funcion_util()
```

Si ejecutas `mi_script.py` directamente:

```
$ python mi_script.py
Este script se está ejecutando directamente.
```

Esta función puede ser usada en otros módulos.

Si importas `mi_script` desde otro archivo:

```
import mi_script
mi_script.funcion_util()
```

Esta función puede ser usada en otros módulos.

El bloque dentro de `if __name__ == "__main__":` no se ejecutará porque el archivo no se está ejecutando directamente.

---

## 3. Ejercicios

### Ejercicios para practicar:

1. **Saludo:** Crea un programa que pida el nombre del usuario y le dé la bienvenida: (Utiliza una f-string)
2. **Sumar dos números:** Crea una función que reciba dos números y devuelva su suma.
3. **Dividir dos números:** Crea una función que reciba dos números y devuelva su división entera. (suponemos que los dos son distintos de cero).
4. **Par o impar:** Crea un programa que verifique si un número es par o impar.
5. **Temperatura.** Crea un programa que te pregunte la temperatura y te indique si hace falta ponerte chaqueta o no.
6. **Contar palabras:** Crea una función que reciba una cadena de texto y devuelva cuántas palabras tiene.
7. **Tabla de multiplicar:** Imprime una tabla con las multiplicaciones del número 5 (del 1 al 10).
8. **Tabla de multiplicar genérica:** Imprime una tabla con las multiplicaciones de un numero dado (del 1 al 10).
9. **Menu.** Crea un programa (`calcu.py`) que muestre por pantalla un menú de opciones /operaciones de una calculadora.

---

### Ejercicios Básicos

1. **Contar vocales:** Crea un programa que reciba una cadena de texto y cuente cuántas vocales (a, e, i, o, u) contiene.
2. **Cuenta regresiva:** Crea un programa que reciba un número entero y haga una cuenta regresiva desde ese número hasta 1, imprimiendo cada número en la consola.
3. **Inversión de una cadena:** Crea una función que reciba una cadena de texto y devuelva la misma cadena pero invertida.

4. **Fibonacci:** Crea una función que reciba un número  $n$  y devuelva los primeros  $n$  números de la secuencia de Fibonacci.
5. **Suma de números en una lista:** Crea una función que reciba una lista de números y devuelva la suma de todos ellos.
6. **Número mayor en una lista:** Crea un programa que reciba una lista de números y devuelva el mayor número de la lista.

### Ejercicios Intermedios

6. **Palíndromo:** Crea un programa que determine si una palabra o frase es un palíndromo (se lee igual de izquierda a derecha que de derecha a izquierda). Por ejemplo, "ana" es un palíndromo, pero "hola" no lo es.
7. **Factorial de un número:** Crea una función que calcule el factorial de un número  $n$ . (El factorial de  $n$  es el producto de todos los números enteros desde 1 hasta  $n$ , es decir,  $n! = n * (n-1) * (n-2) * \dots * 1$ ).
8. **Adivina el número:** Crea un juego en el que el ordenador elija un número aleatorio entre 1 y 100, y el usuario tenga que adivinarlo. El programa debe decirle al usuario si su adivinanza es demasiado alta o baja hasta que adivine correctamente.
9. **Numero primo.** Crea una función que reciba un numero (entero) y nos devuelva si dicho número es primo o no.

### Ejercicios Avanzados

10. **Sumar números en un rango:** Crea una función que sume todos los números en un rango de números dado. La función debe recibir dos parámetros: el inicio y el final del rango, y devolver la suma de todos los números entre esos dos valores (incluyendo los límites).
11. **Anagramas:** Crea una función que reciba dos cadenas de texto y determine si son anagramas entre sí (es decir, si tienen las mismas letras en el mismo número de veces, pero en un orden diferente).
12. **Calculadora de operaciones:** Crea una calculadora que permita realizar operaciones básicas: suma, resta, multiplicación y división. La función debe pedir al usuario la operación a realizar y los números involucrados, y luego imprimir el resultado.
13. **Encuentra los números primos:** Crea una función que reciba un número  $n$  y devuelva una lista con todos los números primos menores o iguales a  $n$ .
14. **Conversión de unidades:** Crea una función que convierta entre diferentes unidades de medida. Por ejemplo, de kilómetros a metros, de libras a kilogramos, etc.

```
def convertir_unidad(valor, unidad_origen, unidad_destino):
 # Implementa la conversión aquí
```

15. **Simulación de un sistema de calificación:** Crea un programa que permita ingresar las calificaciones de varios estudiantes y luego calcule el promedio de todas las calificaciones, además de mostrar las calificaciones por encima o debajo del promedio.

## 4. Python o Python3

### Python 2 vs Python 3

- **Python 2.x:** Fue la versión dominante de Python durante muchos años. La última versión estable de Python 2 fue la **2.7**, lanzada en 2010. A partir de 2020, **Python 2** llegó al final de su vida útil (EOL, por sus siglas en inglés) y ya no recibe actualizaciones ni soporte oficial, lo que significa que no se añaden mejoras de seguridad ni corrección de errores.
- **Python 3.x:** Es la versión actual y en la que los desarrolladores deben centrarse. Se lanzó en 2008 como una versión incompatible con Python 2, lo que significaba que el código de Python 2 no funcionaba automáticamente en Python 3. Esta versión está en constante desarrollo y es la recomendada para nuevos proyectos.

### Diferencias clave entre Python 2 y Python 3

Aquí están algunas de las diferencias más notables entre **Python 2** y **Python 3**:

#### Impresión de datos (print)

- **Python 2:**

La función `print` no es una función, sino una declaración, por lo que no requiere paréntesis.

```
print "Hola, Mundo!" # Sin paréntesis
```

- **Python 3:**

En Python 3, `print` es una función, por lo que se requiere usar paréntesis.

```
print("Hola, Mundo!") # Con paréntesis
```

#### Division de números

- **Python 2:**

La división de enteros en Python 2 devuelve un número entero (truncado hacia abajo).

```
5 / 2 # Devuelve 2, no 2.5
```

- **Python 3:**

En Python 3, la división de enteros devuelve un número flotante.

```
5 / 2 # Devuelve 2.5
```

Si quieres realizar una división entera en Python 3, debes usar `//`:

```
5 // 2 # Devuelve 2
```

## Iteradores y rangos

- **Python 2:**

Las funciones como `range()` y `map()` devuelven listas en lugar de iteradores.

```
range(5) # Devuelve una lista [0, 1, 2, 3, 4]
```

- **Python 3:**

`range()` y otras funciones similares devuelven un objeto iterable (no una lista). Si deseas convertirlo en una lista, puedes envolverlo en `list()`.

```
range(5) # Devuelve un objeto range
list(range(5)) # Devuelve una lista [0, 1, 2, 3, 4]
```

## Cadenas de texto (Strings)

- **Python 2:**

Las cadenas de texto se tratan como ASCII de forma predeterminada. Para manejar caracteres Unicode, debes usar un prefijo `u`:

```
s = u"Hola" # Unicode
```

- **Python 3:**

Las cadenas de texto son Unicode por defecto. No necesitas el prefijo `u` para trabajar con texto Unicode.

```
s = "Hola" # Ya es Unicode por defecto
```

## Excepciones

- **Python 2:**

Para capturar excepciones, se usa la sintaxis:

```
try:
 # Código que puede generar error
except Exception, e:
 # Manejo de excepción
```

- **Python 3:**

En Python 3, la sintaxis de las excepciones cambió para usar la palabra clave `as`:

```
try:
 # Código que puede generar error
except Exception as e:
 # Manejo de excepción
```

## Entrada del usuario

- **Python 2:**

La función para recibir entrada del usuario es `raw_input()`. La función `input()` evalúa el contenido introducido por el usuario.

```
nombre = raw_input("¿Cómo te llamas? ") # Devuelve una cadena
edad = input("¿Cuántos años tienes? ") # Devuelve el valor como un número
```

- **Python 3:**

La función `input()` siempre devuelve una cadena (como en Python 2 `raw_input()`).

```
nombre = input("¿Cómo te llamas? ") # Devuelve una cadena
edad = int(input("¿Cuántos años tienes? ")) # Convierte la entrada a un número
```

## Manejo de archivos

- **Python 2:**

El manejo de archivos en modo binario requiere el uso explícito de la cadena `b`.

```
f = open("archivo.txt", "rb") # Modo binario
```

- **Python 3:**

Python 3 maneja las cadenas de manera más eficiente con el soporte nativo para codificación de texto y archivos binarios.

```
f = open("archivo.txt", "rb") # También válido en Python 3
```

## Por qué Python 3 es importante

- **Python 3** introduce mejoras significativas en el lenguaje que hacen que el código sea más fácil de mantener y más consistente.
- **Python 3** es la versión recomendada para nuevos proyectos, ya que **Python 2** ya no recibe actualizaciones, lo que significa que las aplicaciones y librerías que usan Python 2 pueden volverse vulnerables con el tiempo.

## Compatibilidad entre Python 2 y Python 3

Debido a las diferencias incompatibles entre Python 2 y Python 3, el código escrito para Python 2 no funcionará automáticamente en Python 3. Existen herramientas como `2to3` y `futurize` que ayudan a migrar el código de Python 2 a Python 3, pero aún así puede ser necesario realizar ajustes manuales.

## ¿Por qué aún usar Python 2?

Aunque Python 2 ha sido descontinuado oficialmente, algunas aplicaciones y sistemas heredados aún utilizan Python 2. Sin embargo, el objetivo es migrar esas aplicaciones a Python 3, ya que Python 2 ya no recibe parches de seguridad ni nuevas características.

## ¿Qué versión deberías usar?

- **Para nuevos proyectos:** Siempre debes usar **Python 3**.
- **Para proyectos existentes en Python 2:** Si es posible, migra a Python 3 para garantizar que tu código siga siendo compatible con futuras versiones y reciba soporte de la comunidad.

La principal diferencia entre **Python** y **Python3** es que **Python3** es una versión más moderna y mejorada, que tiene una sintaxis más limpia y es más eficiente en términos de manejo de cadenas, entrada de datos y operaciones matemáticas. Además, **Python 3** es la versión recomendada para futuros desarrollos, ya que **Python 2** ya no tiene soporte oficial desde 2020.