

Índice

1. Introducción a Python	3
1.1. ¿Qué es Python?	3
1.2. Ventajas de usar Python	3
2. Instalación y configuración del entorno de desarrollo	4
2.1. Descarga e instalación de Python	4
2.2. Configuración del entorno de desarrollo	5
2.3. Tu primer programa en python	6
3. Fundamentos de la sintaxis de Python	6
3.1. Variables y asignaciones	6
3.2. Operadores Aritmeticos	11
3.3. Operadores de comparación	14
3.4. Operadores lógicos	17
3.5. Operadores de asignación	19
3.6. Operadores de pertenencia e identidad	22
3.7. Expresiones	23
3.8. Comentarios y formato de código	23
4. Variables y tipos de datos	29
4.1. Tipos numéricos	29
4.2. Tipos de texto	29
4.3. Tipos de secuencia	29
4.4. Tipos de mapeo	29
4.5. Tipos booleanos	29
5. Estructuras de control	29
5.1. Declaraciones condicionales	29
5.2. Bucles y iteraciones	32
5.3. Tipos de bucles en Python	33
6. Listas, tuplas y diccionarios	34
6.1. Listas y operaciones comunes	34
6.2. Tuplas y sus propiedades	39
6.3. Diccionarios y métodos de acceso	40
7. Funciones y modularidad	42
7.1. Definición y llamada de funciones	42
7.2. Argumentos y parámetros	43
7.3. Módulos y su importación	44
8. Manejo de excepciones y errores	45
8.1. Introducción a las excepciones	45
8.2. Uso de try, except, else	45
8.3. Tipos de excepciones y manejo específico	45

9. Entrada y salida de datos	45
9.1. Función print()	45
9.2. Función input()	46
9.3. Formateo de cadenas	47
10.Trabajo con archivos	49
10.1. Apertura, lectura y escritura de archivos	49
10.2. Operaciones comunes de archivos	50
10.3. Manejo de archivos con el bloque “with”	51
11.Introducción a la programación orientada a objetos	53
11.1. Definición de clases y objetos	53
11.2. Métodos y atributos de clase	53
12.Clases y objetos	53
12.1. Encapsulación y visibilidad	53
12.2. Métodos especiales	53
12.3. Herencia y composición	53
13.Herencia y polimorfismo	53
13.1. Herencia simple y múltiple	53
13.2. Polimorfismo y sobrecarga de métodos	53
13.3. Clases abstractas e interfaces	53
14.Módulos y paquetes	53
14.1. ¿Qué son los Módulos en Python?	53
14.2. Creación y Uso de Módulos	53
14.3. Importación de Módulos	53
14.4. Módulos Estándar de Python	53
14.5. ¿Qué son los Paquetes en Python?	53
14.6. Creación y Organización de Paquetes	53
14.7. Importación de Paquetes	53
15.Manipulación de strings y expresiones regulares	53
15.1. Operaciones comunes con cadenas de texto	53
15.2. Operaciones comunes con cadenas de texto	53
16.Trabajo con fechas y tiempos	53
16.1. Módulos datetime y time	53
16.2. Operaciones y formateo de fechas y tiempos	53
17.Introducción a Git	53
17.1. ¿Qué es Git y por qué es importante?	53
17.2. Configuración inicial de Git	53
17.3. Comandos básicos de Git	53

18. Creación de interfaces gráficas con bibliotecas como Tkinter	53
18.1. Introducción a las interfaces gráficas	53
18.2. Uso de Tkinter para GUÍAs simples	53

1. Introducción a Python

1.1. ¿Qué es Python?

Python es un lenguaje de programación, lo que este está diseñado para una fácil comprensión para los seres humanos. Fue creado por Guido van Rossum, lanzó su primera versión en 1991, Python ha ganado popularidad y se ha convertido en uno de los lenguajes de programación.

El diseño de Python se centra en la simplicidad y la elegancia lo que facilita a los desarrolladores escribir códigos claros y concisos.

Python es un lenguaje interpretado, lo que significa que el código escrito por los programadores se traduce a un lenguaje intermedio que luego es ejecutado por el intérprete de Python, esto permite un rápido desarrollo, ya que los programadores pueden ver el resultado de su código inmediatamente después de escribirlo.

Utiliza “indentación” para definir bloques de código, lo que elimina la necesidad de palabras claves adicionales, esto hace que el código sea limpio y fácil de leer. Python es un lenguaje multiparadigma, lo que significa que está orientada a programación con objetos como programación estructurada, esto permite a los desarrolladores utilizar diferentes enfoques según las necesidades de su proyecto. Además Python tiene una gran biblioteca estándar que proporciona módulos y paquetes para diversas aplicaciones, desde desarrollo web hasta procesamiento de datos y cálculos científicos

1.2. Ventajas de usar Python

1. Es un lenguaje de sintaxis amplia y legible: Al ser lenguaje de programación de alto nivel, diseñado para que los algoritmos sean expresados de forma clara y fácilmente entendibles por los seres humanos.
2. Ampliamente utilizado en múltiples campos: Su amplia variedad de usos lo ha dejado como el primero en el top 10 de los lenguajes de programación más utilizados según Tiobe, extraídos de las habilidades más desarrolladas por desarrolladores, empresas del sector y terceros.
3. Python posee una gran cantidad de bibliotecas y frameworks : La gran cantidad de usos de Python se traduce en múltiples librerías y frameworks que ayudan a llevar a cabo funcionalidades. En sí mismo ya tiene una biblioteca estándar y podemos encontrar hasta 135. 000+ más para

diversas aplicaciones. Sin embargo, entre las más populares según el sitio de AWS se encuentran Matplotlib, Pandas, Request, Numpy, Keras y OpenCV-Python. Esta variedad no se limita solo a las librerías. Así mismo la podemos encontrar en los marcos o frameworks, que facilitan el proceso de creación debido a que ahorra el proceso de escritura de un código.

4. Fácil portabilidad: Python es uno de los lenguajes de programación más portátiles y versátiles disponibles. Debido a que es un lenguaje de programación interpretado, en lugar de un lenguaje compilado, se puede ejecutar en una amplia variedad de sistemas operativos y plataformas de hardware sin necesidad de realizar ajustes o cambios significativos en el código fuente.
5. Tiene una gran comunidad de desarrolladores : Es una herramienta que constantemente evoluciona para suplir las necesidades que poco a poco van surgiendo en el campo de la tecnología, como hemos visto hasta ahora en sus usos para el machine learning.
6. Multiplataforma: Python es uno de esos lenguajes de programación que puede ser ejecutado en cualquier sistema operativo en el cual se opere. Así es: no importa si se trata de Windows, Linux, macOS, y otros, este se puede ejecutar sin problema. Y, lo mejor, es que se desarrolla el código una única vez y podrá emplearse en los demás SO.

2. Instalación y configuración del entorno de desarrollo

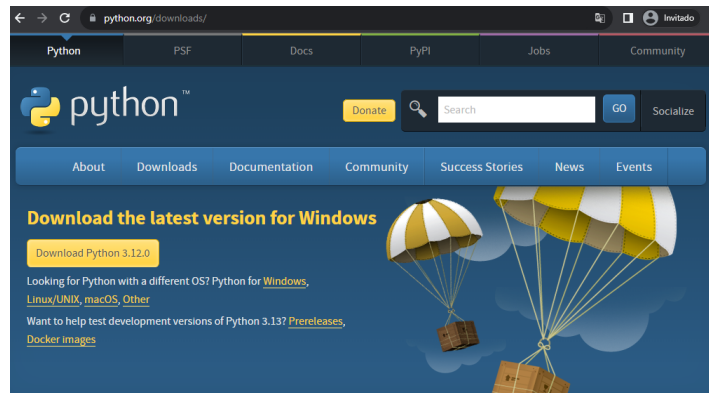
2.1. Descarga e instalación de Python

A continuación, te dejaré un video para aprender de manera más práctica y visual, tienes que seguir los pasos que se mencionan: Tutorial

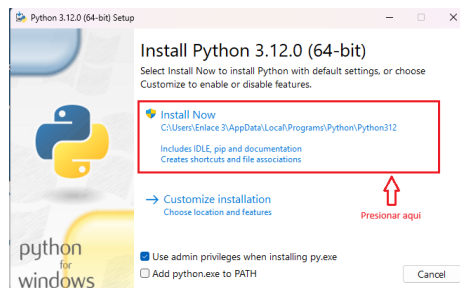


Siguiendo con algo más teórico, para comenzar a programar en Python, primero necesitas instalar Python en tu sistema. Puedes hacerlo siguiendo estos pasos:

- a. Descarga Python: Visita el sitio web oficial de Python en python.org y descarga la última versión de Python. Asegúrate de elegir la versión más reciente y compatible con tu sistema operativo (Windows, macOS, o Linux).



- b. Instalación en Windows: Ejecuta el archivo descargado y sigue las instrucciones del instalador. Asegúrate de marcar la opción “Agregar Python al PATH” durante la instalación.



2.2. Configuración del entorno de desarrollo

Puedes escribir y ejecutar código Python en un entorno de desarrollo integrado (IDE) o simplemente en un editor de texto. Algunas opciones populares para IDEs incluyen:

Elige el IDE o editor de tu preferencia y configúralo.

- **IDLE:** es un programa que te ayuda a escribir y ejecutar código en Python. Es útil para principiantes porque combina un espacio para escribir código y un lugar para ver los resultados, lo que facilita aprender y probar cosas nuevas en Python. Tutorial de Cómo instalar Python y usar la herramienta IDLE: Revisa este link para más ayuda
- **PyCharm:** Un IDE muy popular y potente para Python. Tutorial de Como instalar y configurar Pycharm: Revisa este link para más ayuda
- **Visual Studio Code (VSCode):** Un editor de código que es altamente configurable y ampliamente utilizado para Python. Tutorial de como instalar



y Configurar Visual Studio Code: Revisa este link para más ayuda



2.3. Tu primer programa en python

Una vez que hayas instalado Python y configurado tu entorno de desarrollo, puedes comenzar a escribir código Python. Aquí tienes un ejemplo de un programa Python simple: Para ejecutar el código, guárdalo en un archivo con extensión “.py” (por ejemplo, “mi_programa.py”) y ejecútalo desde la línea de comandos o desde tu IDE.

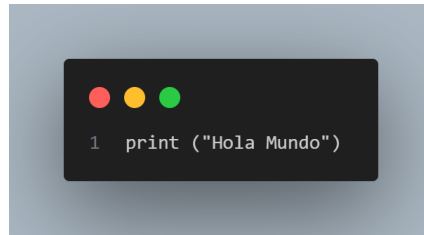
3. Fundamentos de la sintaxis de Python

Python se destaca por su legibilidad y simplicidad. Algunas pautas importantes de la sintaxis incluyen:

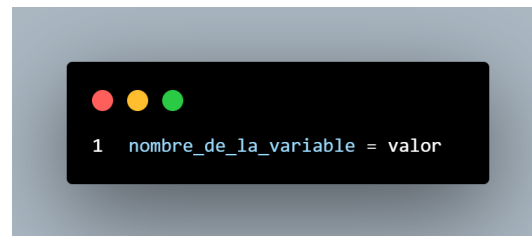
3.1. Variables y asignaciones

Las variables en Python son contenedores que se utilizan para almacenar datos. Estos datos pueden ser números, cadenas de texto, listas, objetos o cualquier otro tipo de información que necesites en tu programa. Las variables te permiten acceder, modificar y manipular datos de manera dinámica en tu código.

Declaración y Asignación: Para crear una variable en Python, simplemente elige un nombre descriptivo y utiliza el operador de asignación “=” para darle un valor. La estructura general es la siguiente:

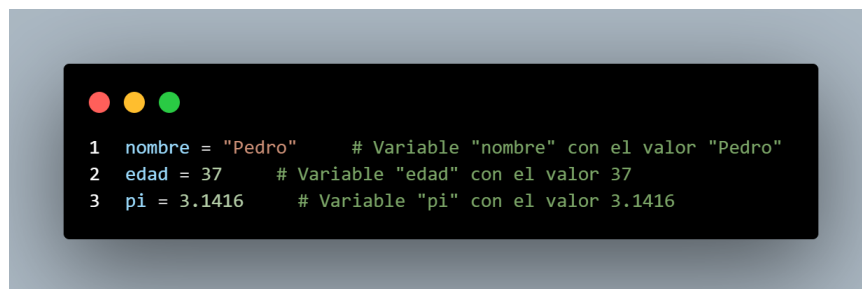
A terminal window with a dark background and three colored window control buttons (red, yellow, green) at the top left. It contains a single line of Python code: `1 print ("Hola Mundo")`.

```
1 print ("Hola Mundo")
```

A terminal window with a dark background and three colored window control buttons (red, yellow, green) at the top left. It contains a single line of Python code: `1 nombre_de_la_variable = valor`.

```
1 nombre_de_la_variable = valor
```

- “nombre_de_la_variable”: Es el nombre que eliges para la variable. Debe seguir las reglas de nomenclatura y convenciones de nombres que se mencionaron previamente.
- “valor”: Es el dato que deseas almacenar en la variable. Puede ser un número, una cadena de texto, un resultado de cálculos, una lista, un objeto, etc.

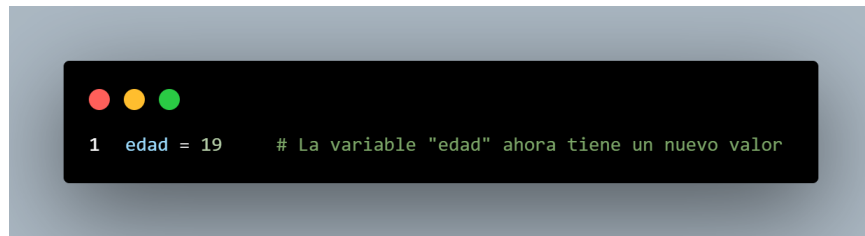
A terminal window with a dark background and three colored window control buttons (red, yellow, green) at the top left. It contains three lines of Python code, each with a comment explaining the variable and its value.

```
1 nombre = "Pedro"      # Variable "nombre" con el valor "Pedro"  
2 edad = 37             # Variable "edad" con el valor 37  
3 pi = 3.1416           # Variable "pi" con el valor 3.1416
```

Reasignación de variables: Puedes cambiar el valor de una variable en cualquier momento asignándole un nuevo valor:

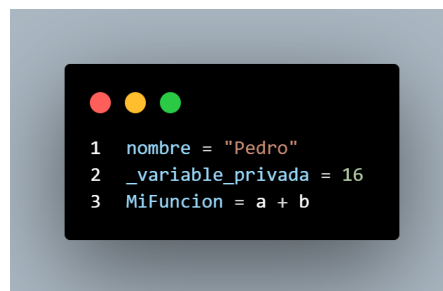
Identificadores: Los identificadores son nombres que se utilizan para representar variables, funciones, clases y otros elementos en Python. Algunas reglas claves para los identificadores en Python son:

- Deben comenzar con una letra (mayúscula o minúscula) o un guión bajo “_”.
- Pueden contener letras, números y guiones bajos.



```
1 edad = 19      # La variable "edad" ahora tiene un nuevo valor
```

- Python distingue entre mayúsculas y minúsculas, por lo que `mi_variable` y `Mi_Variable` son consideradas diferentes.
- No se pueden utilizar palabras clave de Python como identificadores. Las palabras clave son términos reservados para funciones y estructuras de control, como `if`, `for`, `while`, entre otros.



```
1 nombre = "Pedro"
2 _variable_privada = 16
3 MiFuncion = a + b
```

Uso de variables: Las variables se utilizan para almacenar valores y realizar cálculos. Puedes utilizar variables en expresiones matemáticas, en operaciones de cadena de texto, en estructuras de control (como condicionales y bucles) y para muchas otras tareas en tu programa.

Constantes: Puedes usar variables para definir constantes, que son valores que no deben modificarse a lo largo del programa. Para indicar que una variable es una constante, es común utilizar letras mayúsculas y subrayados (por ejemplo, `PI = 3.1416`).

Desestructuración de Asignación: Python permite asignar valores a Múltiples variables en una sola línea de código. Esto es especialmente útil cuando se trabaja con secuencias como listas o tuplas.

Operadores de Asignación Combinada: Python ofrece operadores de asignación combinada que permiten simplificar la asignación de variables en operaciones comunes. Por ejemplo, `+=` es una forma abreviada de incrementar el valor de una variable.

Tipos de datos: Python es un lenguaje de tipo dinámico, lo que significa que una variable no tiene un tipo de dato fijo. El tipo de dato de una variable se




```
1 # Usando variables en una expresión
2
3 a = 3
4 b = 5
5 suma = a + b    # La suma contiene el resultado de 5 + 3, es decir, 8
6
7 # Usando variables en una cadena de texto
8 nombre = "Pedro"
9 Mensaje = "Hola " + nombre    # Mensaje contiene "Hola Pedro"
```




```
1 x, y, z = 1, 2, 3
```

asigna automáticamente según el valor que contiene. Puedes verificar el tipo de dato de una variable utilizando la función “type()”:




```
1 x = 5
2 x += 3      # Esto es equivalente a x = x + 3
```



```
1 numero = 42
2 cadena = "Hola"
3 print(type(numero))      # <class 'int'> (entero)
4 print(type(cadena))      # <class 'str'> (cadena de texto)
```

Ámbito de variables: Las variables pueden tener un ámbito local o global:

- Ámbito local: Las variables declaradas dentro de una función tienen un ámbito local y solo son accesibles dentro de esa función.



```
1 def mi_funcion():
2     variable_local = 10
3     print(variable_local)
4
5 mi_funcion()      # Imprime 10
6 print(variable_local)  # Error, variable_local no está definida en este ámbito
```

- Ámbito Global: Las variables declaradas fuera de una función o declaradas como globales dentro de una función tienen un ámbito global y son accesibles desde cualquier parte del código.

```
1 variable_global = 100
2 def modificar_variable_global():
3     global variable_global
4     variable_global = 200
5
6 modificar_variable_global()
7 print(variable_global)    # Imprime 200
```

Ejemplo usando ambos:

```
1 # Variable global
2 x = 10
3
4 def mi_funcion():
5     # Variable local
6     y = 5
7     # Acceso a la variable global
8     global x
9     x = 20
10
11 mi_funcion()
12 print(x)    # Imprime 20 (el valor de la variable global ha sido modificado)
```

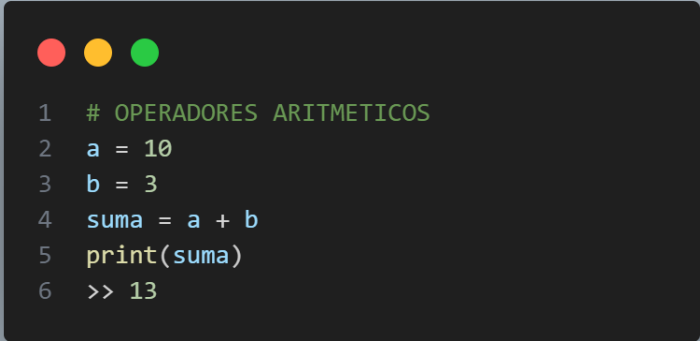
Estos son algunos aspectos adicionales relacionados con las variables en Python que pueden ser útiles para comprender en tu proceso de aprendizaje.

3.2. Operadores Aritmeticos

Los operadores aritméticos en Python se utilizan para realizar operaciones matemáticas.

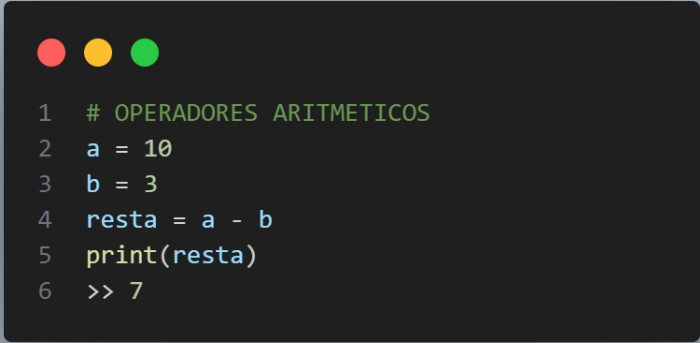
Los principales son:

- Suma (+): Se utiliza para sumar dos valores.
- Resta (-): Se utiliza para restar un valor de otro.
- Multiplicación (*): Se utiliza para multiplicar dos valores.



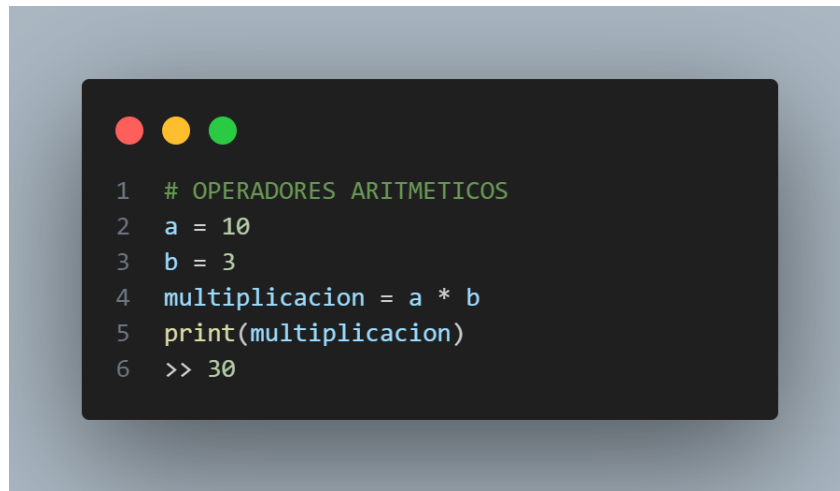
```
1  # OPERADORES ARITMETICOS
2  a = 10
3  b = 3
4  suma = a + b
5  print(suma)
6  >> 13
```

- División (/): Se utiliza para dividir un valor por otro.
- División Entera (//): Devuelve la parte entera de la división entre dos valores.



```
1  # OPERADORES ARITMETICOS
2  a = 10
3  b = 3
4  resta = a - b
5  print(resta)
6  >> 7
```

- Módulo (%): Devuelve el resto de la división entre dos valores.
- Potencia (**): Eleva un valor a una potencia.

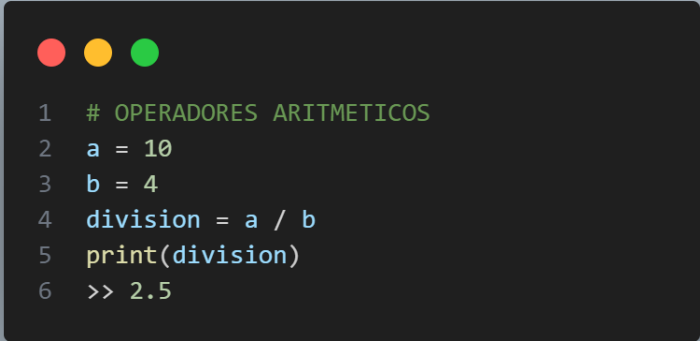


```
1  # OPERADORES ARITMETICOS
2  a = 10
3  b = 3
4  multiplicacion = a * b
5  print(multiplicacion)
6  >> 30
```

3.3. Operadores de comparación

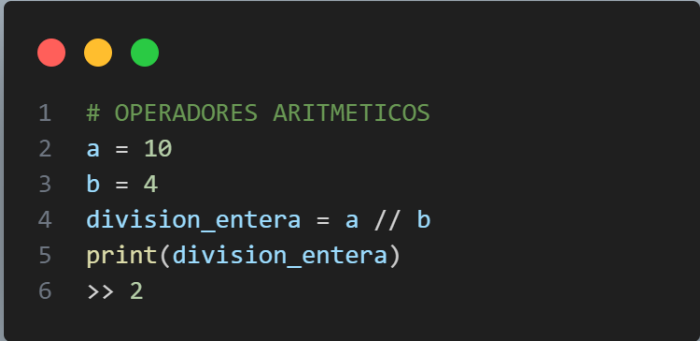
Los operadores de comparación se utilizan para comparar dos valores y devuelven un valor booleano (True o False). Los principales son:

- Igual (==): Comprueba si dos valores son iguales.



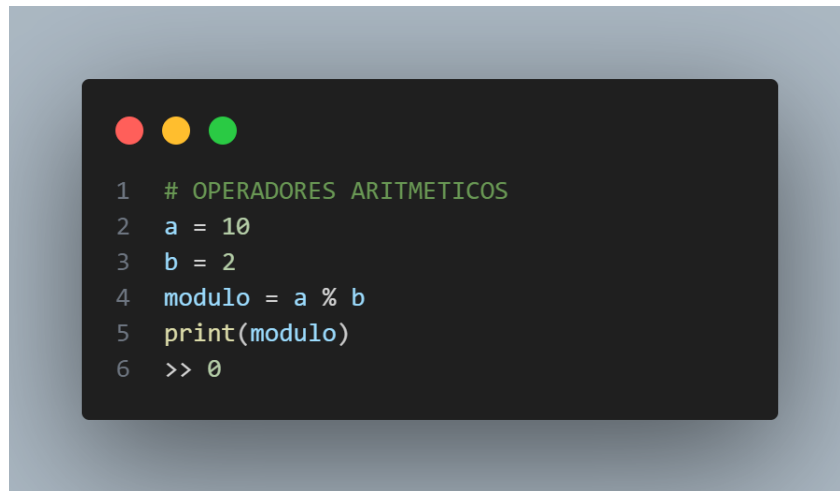
```
1  # OPERADORES ARITMETICOS
2  a = 10
3  b = 4
4  division = a / b
5  print(division)
6  >> 2.5
```

- No Igual (!=): Comprueba si dos valores no son iguales.
- Mayor que (>): Comprueba si un valor es mayor que otro. En caso de strings se compara cadenas de caracteres lexicográficamente (es decir, en función del orden alfabético).



```
1  # OPERADORES ARITMETICOS
2  a = 10
3  b = 4
4  division_entera = a // b
5  print(division_entera)
6  >> 2
```

- Menor que (<): Comprueba si un valor es menor que otro. En caso de strings se compara cadenas de caracteres lexicográficamente (es decir, en función del orden alfabético).



```
1  # OPERADORES ARITMETICOS
2  a = 10
3  b = 2
4  modulo = a % b
5  print(modulo)
6  >> 0
```

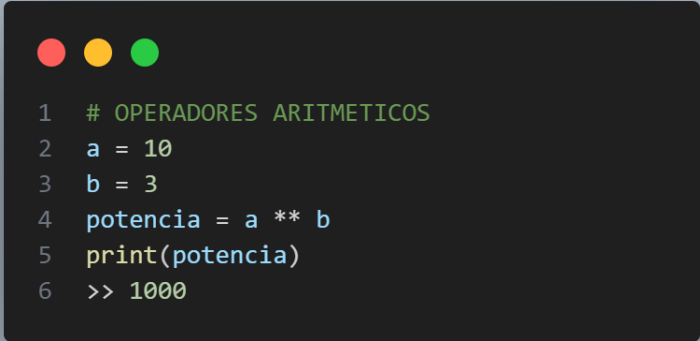
- Mayor o igual que (\geq): Comprueba si un valor es mayor o igual que otro. En caso de strings se compara cadenas de caracteres lexicográficamente (es decir, en función del orden alfabético).
- Menor o igual que (\leq): Comprueba si un valor es menor o igual que otro. En caso de strings se compara cadenas de caracteres lexicográficamente (es decir, en función del orden alfabético).

3.4. Operadores lógicos

Los operadores lógicos se utilizan para combinar expresiones lógicas y devuelven un valor booleano.

Los principales son:

- `and`: Devuelve `True` si ambas expresiones son `True`.



```
1  # OPERADORES ARITMETICOS
2  a = 10
3  b = 3
4  potencia = a ** b
5  print(potencia)
6  >> 1000
```

- or: Devuelve True si al menos una de las expresiones es True.



```
1  # OPERADORES DE COMPARACION
2  a = 10
3  b = 5
4  c = "cinco"
5  d = 5.0
6
7  # Igualdad
8  print(a==b) >> False #10 no es igual a 5
9  print(b==d) >> True #5 es igual a 5.0
10 print(b==c) >> False #5 no es igual a "cinco"
11 print(a==10) >> True #10 es igual a 10
```

- not: Devuelve el inverso de la expresión.

3.5. Operadores de asignación


Los operadores de asignación se utilizan para asignar valores a variables. Algunos de ellos incluyen:

- =: Asigna un valor a una variable.



```
1  # OPERADORES DE COMPARACION
2  a = 10
3  b = 5
4  c = "cinco"
5  d = 5.0
6
7  # Desigualdad
8  print(a!=b) >> True #10 no es igual a 5
9  print(b!=d) >> False #5 es igual a 5.0
10 print(b!=c) >> True #5 no es igual a "cinco"
11 print(a!=10) >> False #10 es igual a 10
```

- +=: Suma y asigna el resultado a la variable.
- -=: Resta y asigna el resultado a la variable.



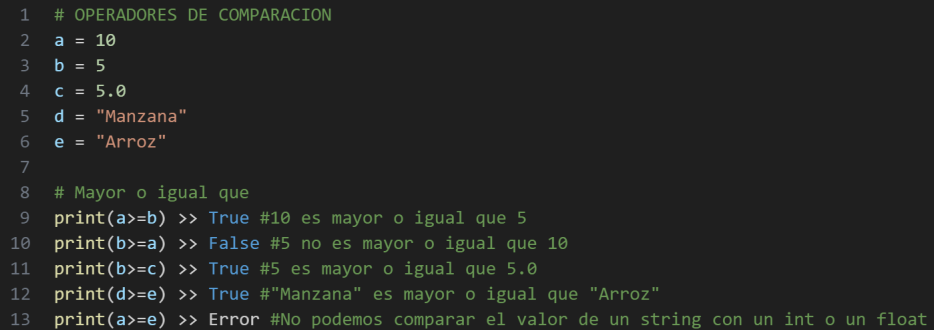
```
1  # OPERADORES DE COMPARACION
2  a = 10
3  b = 5
4  c = 5.0
5  d = "Manzana"
6  e = "Arroz"
7
8  # Mayor que
9  print(a>b) >> True #10 es mayor que 5
10 print(b>a) >> False #5 no es mayor que 10
11 print(b>c) >> False #5 no es mayor que 5.0
12 print(d>e) >> True #"Manzana" es mayor que "Arroz"
13 print(a>e) >> Error #No podemos comparar el valor de un string con un int o un float
```

- `*=`: Multiplica y asigna el resultado a la variable.
- `/=`: Divide y asigna el resultado a la variable.

```
1  # OPERADORES DE COMPARACION
2  a = 10
3  b = 5
4  c = 5.0
5  d = "Manzana"
6  e = "Arroz"
7
8  # Menor que
9  print(a<b) >> False #10 no es menor que 5
10 print(b<a) >> True #5 es menor que 10
11 print(b<c) >> False #5 no es menor que 5.0
12 print(d<e) >> False #"Manzana" no es menor que "Arroz"
13 print(a<e) >> Error #No podemos comparar el valor de un string con un int o un float
14
```

3.6. Operadores de pertenencia e identidad

- Operadores de Pertenencia: 'in' y 'not in' se utilizan para verificar si un elemento está presente en una secuencia (como una lista, tupla, etc.)
- Operadores de Identidad: 'is' y 'is not' se utilizan para verificar si dos objetos tienen la misma identidad (es decir, si son el mismo objeto en la memoria).



```

1  # OPERADORES DE COMPARACION
2  a = 10
3  b = 5
4  c = 5.0
5  d = "Manzana"
6  e = "Arroz"
7
8  # Mayor o igual que
9  print(a>=b) >> True #10 es mayor o igual que 5
10 print(b>=a) >> False #5 no es mayor o igual que 10
11 print(b>=c) >> True #5 es mayor o igual que 5.0
12 print(d>=e) >> True #"Manzana" es mayor o igual que "Arroz"
13 print(a>=e) >> Error #No podemos comparar el valor de un string con un int o un float

```

3.7. Expresiones

Una expresión es una combinación de valores, variables y operadores que se evalúa para producir un resultado. Las expresiones se utilizan en muchas partes del código, como asignaciones, condicionales, bucles y funciones. Por ejemplo:

Las expresiones pueden ser tan simples como una sola variable o tan complejas como cálculos matemáticos o lógicos.

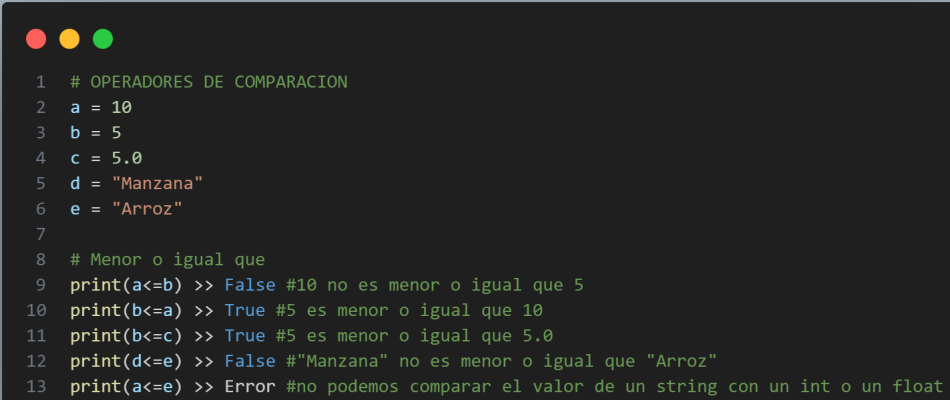
Estos son los operadores y las expresiones en Python que son fundamentales para realizar tareas de cálculo, toma de decisiones y procesamiento de datos en tus programas. Puedes combinar estos operadores y expresiones de diversas maneras para realizar tareas específicas en tu código.

3.8. Comentarios y formato de código

Los comentarios y el formato de código son aspectos cruciales de la programación en Python. Los comentarios son notas en el código que se utilizan para explicar lo que hace el código, mientras que el formato se refiere a la estructura y el estilo del código para que sea más legible y mantenible.

Comentarios: Los comentarios en Python se utilizan para agregar explicaciones o notas al código fuente. Se inician con el símbolo “#” y Python los ignora durante la ejecución. Los comentarios son útiles para documentar el código y facilitar su comprensión tanto para el programador como para otros colaboradores.

- **Comentarios de una sola línea:** Se utilizan para incluir explicaciones breves en una línea.



```


1  # OPERADORES DE COMPARACION
2  a = 10
3  b = 5
4  c = 5.0
5  d = "Manzana"
6  e = "Arroz"
7
8  # Menor o igual que
9  print(a<=b) >> False #10 no es menor o igual que 5
10 print(b<=a) >> True #5 es menor o igual que 10
11 print(b<=c) >> True #5 es menor o igual que 5.0
12 print(d<=e) >> False #"Manzana" no es menor o igual que "Arroz"
13 print(a<=e) >> Error #no podemos comparar el valor de un string con un int o un float

```

- Comentarios de varias líneas: Se utilizan para describir secciones de código más largas y pueden abarcar varias líneas.

Formato de Código: El formato del código es importante para mantener la legibilidad y la consistencia en un proyecto. Python se destaca por su énfasis en la indentación (espacios al principio de una línea) para delimitar bloques de código.

- Indentación: La sangría en Python es fundamental para definir bloques de código. A diferencia de otros lenguajes que utilizan llaves o paréntesis para delimitar bloques, en Python, la sangría se utiliza para indicar la estructura del programa. Esto fomenta la legibilidad y la claridad del código. Un bloque de código se define mediante el uso de espacios o tabulaciones con un mismo nivel de sangría.
- Interpretado: Python es un lenguaje de programación interpretado. Esto significa que no es necesario compilar el código fuente en código de máquina antes de ejecutarlo. En cambio, el intérprete de Python lee y ejecuta el código línea por línea en tiempo real. Esto facilita la depuración y la portabilidad, ya que un programa escrito en Python se puede ejecutar en cualquier sistema que tenga un intérprete de Python instalado.
- Tipado dinámico: Python es un lenguaje de tipado dinámico, lo que significa que no necesitas declarar explícitamente el tipo de una variable al crearla. El tipo de una variable se determina automáticamente en el tiempo de ejecución según el valor que contiene. Esto permite una mayor flexibilidad, pero también significa que debes tener cuidado con los tipos de datos para evitar errores inesperados.



```

1  # OPERADORES LOGICOS
2  a = True
3  b = False
4  c = True
5  d = False
6
7  print(a and b) >> False #Solo una variable tiene el valor True
8  print(a and c) >> True #Ambas variables tienen el valor True
9  print(b and d) >> False #Ninguna variable tiene el valor True

```

- Convenciones de Nombres: En Python, se siguen ciertas convenciones para nombrar variables, funciones y clases. Algunas pautas comunes incluyen:
 - Los operadores y las expresiones son componentes esenciales en Python y en la programación en general. Los operadores se utilizan para realizar operaciones en variables y valores, mientras que las expresiones son combinaciones de variables, valores y operadores que se evalúan para producir un resultado.
 - Variables y funciones: Usar letras minúsculas y guiones bajos para separar palabras (ejemplo: `mi_variable`, `mi_funcion`).
 - Clases: Usar CamelCase (iniciar cada palabra con mayúscula, sin espacios ni guiones bajos) (ejemplo: `MiClase`).
 - Constantes: Los nombres de las constantes deben estar en mayúsculas. Si es necesario separar palabras en el nombre de la constante, se pueden usar guiones bajos. Por ejemplo, `PI` o `TASA_DE_INTERES`.
 - Convenciones específicas: Para indicar que una variable o función es “privada” (es decir, no debería accederse desde fuera de su módulo), se puede anteponer un guion bajo al nombre, por ejemplo, `_variable_privada`. Para variables que actúan como constantes y no deberían modificarse, se puede utilizar un guion bajo en mayúsculas al principio del nombre, como `_CONSTANTE`.
 - Variables temporales: En bucles y expresiones cortas, es común usar nombres de variables temporales cortos como `i`, `j`, `k` para iteradores. Sin embargo, es importante que estos nombres sean descriptivos dentro del contexto.



```
1  # OPERADORES LOGICOS
2  a = True
3  b = False
4  c = True
5  d = False
6
7  print(a or b) >> True #Al menos una variable tiene el valor True
8  print(b or d) >> False #Ninguna variable tiene el valor True
```

- Espaciado: Agregar espacios alrededor de operadores y después de comas para mejorar la legibilidad.



```
1  # OPERADORES LOGICOS
2  a = True
3  b = False
4  c = True
5  d = False
6
7  print(a or b) >> True #Al menos una variable tiene el valor True
8  print(b or d) >> False #Ninguna variable tiene el valor True
```



```
1  # OPERADORES DE ASIGNACION
2
3  a = 10 # A la variable a se le asigna el valor 10
```

- Longitud de Línea: Es una buena práctica limitar la longitud de una línea de código a aproximadamente 79-80 caracteres para facilitar la lectura. Puedes usar una barra invertida para dividir una línea larga en varias líneas:
- Comentarios Descriptivos: Los comentarios deben ser informativos y explicar el propósito de una sección de código. Es especialmente útil para documentar partes complicadas o algoritmos. No es necesario comentar lo obvio.

El formato de código y los comentarios son esenciales para que tu código sea comprensible, colaborativo y mantenible. Adherirse a las convenciones de estilo y escribir comentarios informativos ayuda a otros programadores (y a ti mismo) a entender y mantener el código de manera más eficaz.



```
1  # OPERADORES DE ASIGNACION
2
3  a = 10
4  a += 2 #Se suman 2 a la variable y se reasigna su valor
5  print(a) >> 12
```



```
1  # OPERADORES DE ASIGNACION
2
3  a = 10
4  a -= 2 #Se restan 2 a la variable y se reasigna su valor
5  print(a) >> 8
```



```
1  # OPERADORES DE ASIGNACION
2
3  a = 10
4  a *= 2 #La variable se multiplica por 2 y se reasigna su valor
5  print(a) >> 20
```



```
1  # OPERADORES DE ASIGNACION
2
3  a = 10
4  a /= 2 #La variable se divide en dos y se reasigna su valor
5  print(a) >> 5
```

4. Variables y tipos de datos

4.1. Tipos numéricos

4.2. Tipos de texto

4.3. Tipos de secuencia

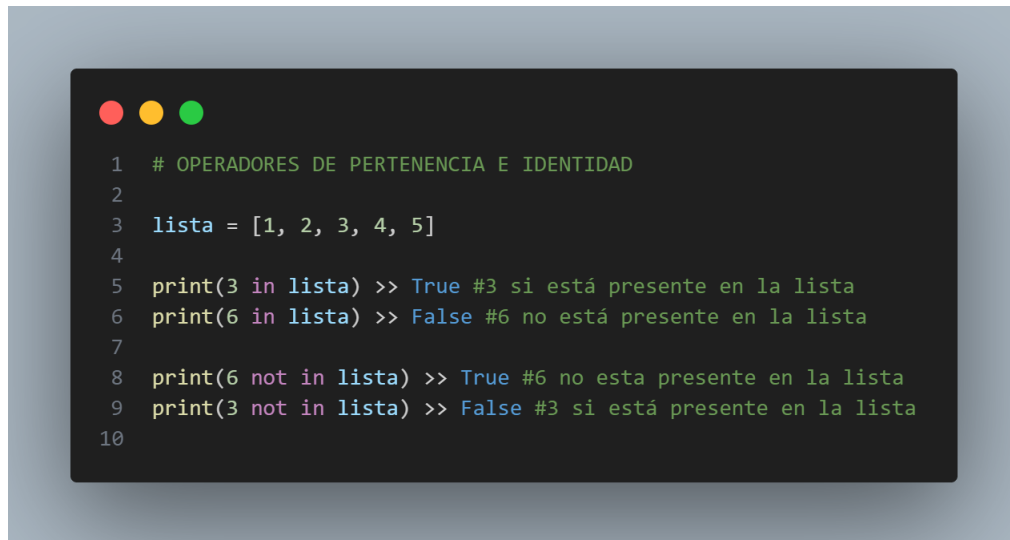
4.4. Tipos de mapeo

4.5. Tipos booleanos

5. Estructuras de control

5.1. Declaraciones condicionales

Las sentencias condicionales en Python permiten controlar el flujo del programa en función del valor de una expresión booleana. Una expresión booleana

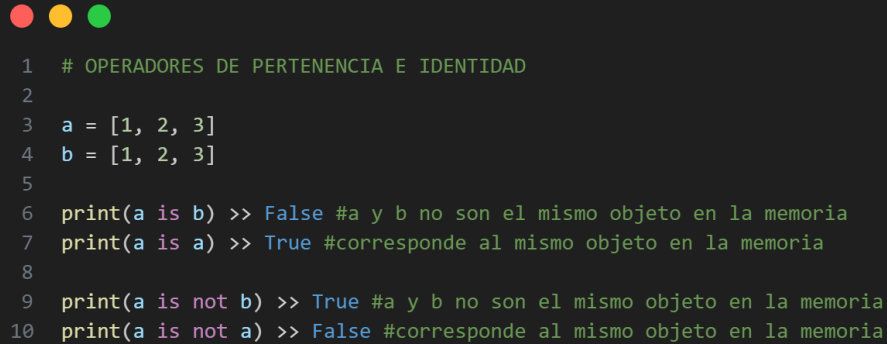
A terminal window with a dark background and light green text. It shows a Python script with 10 lines of code. The code defines a list 'lista' and uses 'in' and 'not in' operators to check for the presence of 3 and 6 in the list. Comments in Spanish explain the results. The terminal has three colored window control buttons (red, yellow, green) in the top left corner.

```
1  # OPERADORES DE PERTENENCIA E IDENTIDAD
2
3  lista = [1, 2, 3, 4, 5]
4
5  print(3 in lista) >> True #3 si está presente en la lista
6  print(6 in lista) >> False #6 no está presente en la lista
7
8  print(6 not in lista) >> True #6 no esta presente en la lista
9  print(3 not in lista) >> False #3 si está presente en la lista
10
```

es una expresión que puede evaluarse como verdadera o falsa.

Sentencia if: se considera la más sencilla de las tres y toma decisiones basadas en sí la condición es verdadera o falsa. Si la condición es verdadera, se ejecuta el bloque de código sangrado. Si la condición es falsa, se omite la ejecución del bloque.

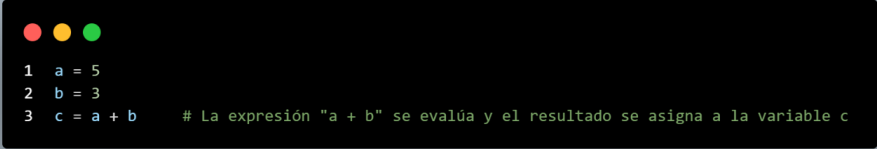
La sintaxis básica de la condición if es la siguiente:



```

1  # OPERADORES DE PERTENENCIA E IDENTIDAD
2
3  a = [1, 2, 3]
4  b = [1, 2, 3]
5
6  print(a is b) >> False #a y b no son el mismo objeto en la memoria
7  print(a is a) >> True #corresponde al mismo objeto en la memoria
8
9  print(a is not b) >> True #a y b no son el mismo objeto en la memoria
10 print(a is not a) >> False #corresponde al mismo objeto en la memoria

```



```

1  a = 5
2  b = 3
3  c = a + b      # La expresión "a + b" se evalúa y el resultado se asigna a la variable c

```

Por ejemplo, el siguiente código imprime “El número es mayor que 10” si el número es mayor que 10:

En este ejemplo, la condición es que el número sea mayor que 10. El bloque de código contiene una sola instrucción, que es imprimir el mensaje “El número es mayor que 10”.

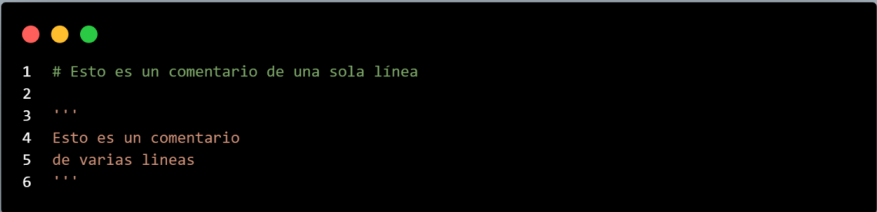
Sentencia else (opcional): Puedes agregar un bloque de código que se ejecutará si la primera condición es falsa.

La sintaxis básica de la condición else es la siguiente:

Por ejemplo, el siguiente código imprime “El número es menor que 10” si el número es menor que 10:

En este ejemplo, la condición es que el número sea mayor que 10. El bloque de código principal se ejecuta si la condición es verdadera. Si la condición es falsa, se ejecuta el bloque de código alternativo.

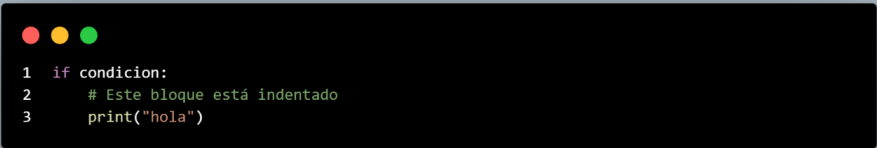
Sentencia elif (opcional): Para verificar múltiples condiciones, puedes utilizar elif después de la declaración if. Ten en cuenta que solo se ejecutará el bloque de código correspondiente a la primera condición verdadera.



```

1 # Esto es un comentario de una sola línea
2
3 '''
4 Esto es un comentario
5 de varias líneas
6 '''

```



```

1 if condicion:
2     # Este bloque está indentado
3     print("hola")

```

La sintaxis básica de la sentencia `elif` es la siguiente:

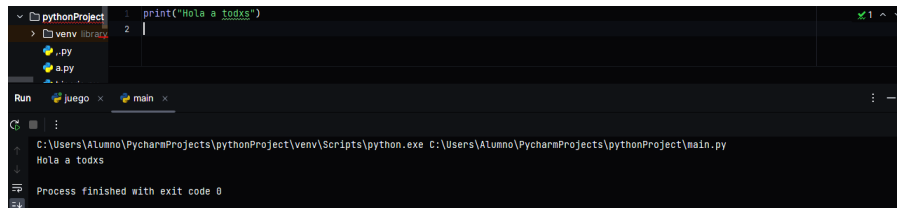
Por ejemplo, el siguiente código imprime “El número es mayor que 10” si el número es mayor que 10, “El número es igual a 10” si el número es igual a 10, y “El número es menor que 10” si el número es menor que 10:

En este ejemplo, la condición principal es que el número sea mayor que 10. Si la condición principal es verdadera, se ejecuta el bloque de código principal. Si la condición principal es falsa, se evalúa la primera condición `elif`. Si la primera condición `elif` es verdadera, se ejecuta el bloque de código alternativo. Si la primera condición `elif` es falsa, se evalúa la segunda condición `elif`, y así sucesivamente. Si ninguna de las condiciones `elif` es verdadera, se ejecuta el bloque de código `else`.

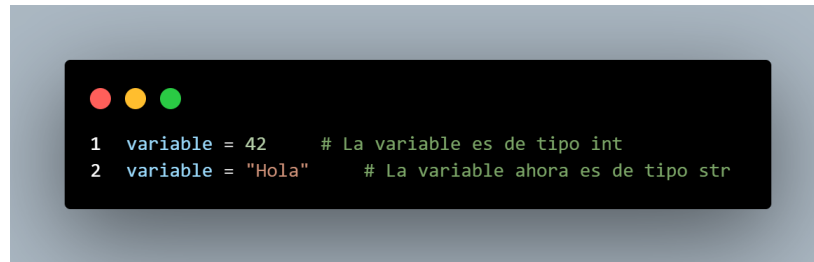
5.2. Bucles y iteraciones

En el contexto de la programación, los términos `loops` e `iteraciones` a menudo se utilizan indistintamente, pero hay una distinción sutil:

- **Iteración:** La iteración se refiere al proceso de repetir una acción o un bloque de código para cada elemento de una secuencia o hasta que se cumpla una condición. La iteración no está limitada a los bucles, ya que puede ocurrir en otros contextos, como al recorrer elementos en una lista o realizar operaciones en un conjunto de datos.
- **Loop (bucle):** Un loop es una estructura de control que permite repetir un bloque de código múltiples veces. Puede ser un ciclo `for` o “`while`”, y generalmente se utiliza cuando se conoce la cantidad de repeticiones necesarias. Un loop puede realizar iteraciones.



The screenshot shows the PyCharm IDE interface. The top pane displays a Python script with two lines of code: `1 print("Hola a todxs")` and `2`. The bottom pane shows the 'Run' output, indicating that the program executed successfully and printed 'Hola a todxs'. The status bar at the bottom indicates 'Process finished with exit code 0'.



The screenshot shows a code editor with two lines of Python code. The first line is `1 variable = 42` followed by a comment `# La variable es de tipo int`. The second line is `2 variable = "Hola"` followed by a comment `# La variable ahora es de tipo str`. The code is displayed in a dark-themed editor with syntax highlighting.

5.3. Tipos de bucles en Python

En Python, existen dos tipos de bucles:

- Bucle for: El bucle for se utiliza para iterar sobre una colección de datos. La sintaxis es la siguiente:

Por ejemplo, el siguiente código imprime los números del 1 al 10:

```
1 variable = 42
2 lista = [1, 2, 3]
3 resultado = x + y
```

```
1 texto_largo = "Esta es una línea de texto muy larga que se divide en varias líneas \n
para mejorar la legibilidad."
```

- Bucle while: El bucle while se utiliza para iterar mientras una condición sea verdadera. La sintaxis es la siguiente:

Por ejemplo, el siguiente código imprime los números del 1 al 10:

6. Listas, tuplas y diccionarios

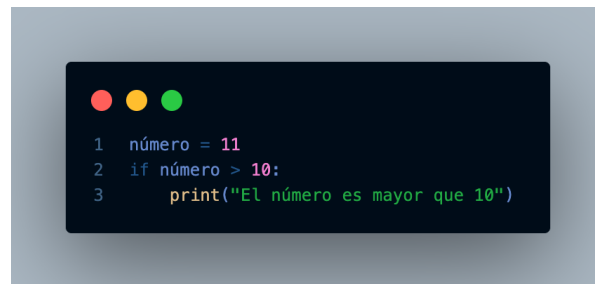
6.1. Listas y operaciones comunes

Las listas en python son un conjunto ordenado y editable de elementos, son dinámicas esto quiere decir que puede contener diferente tipos de datos, y aparte cabe recalcar que puede tener elementos duplicados y no generará error. Estas se pueden ocupar tanto como para recopilar datos como para organizar y alguna de sus funciones son las siguientes:

- Creación de listas: Se puede crear una listas abriendo corchetes “[]” y separando los datos por comas “,”




```
1 if condición:
2     # Bloque de código
```




```
1 número = 11
2 if número > 10:
3     print("El número es mayor que 10")
```

- Acceso a elementos: se puede acceder a un elemento en específico de la lista poniendo el nombre de este y dentro de corchetes el número en el lugar que se encuentra (se empieza a contar desde el 0, y para seleccionar el último será -1).
- Modificar elementos: Seleccionando el puesto en el que se encuentra el dato e igualarlo al valor que queramos.




```
1  if condición:
2      # Bloque de código
3  else:
4      # Bloque de código alternativo
```




```
1  número = 5
2
3  if número > 10:
4      print("El número es mayor que 10")
5  else:
6      print("El número es menor que 10")
```

- Añadir elementos: Utilizando el método “.append(elemento)” se pueden agregar datos.
- Insertar elementos: Si bien cumple la misma función que se describió anteriormente se ocupa el método “.insert(posición, elemento)” con esta también se puede detallar la posición en que queremos que esté.
- Eliminar elementos: Para eliminar elementos hay dos métodos el “.pop(lugar del elemento)” que elimina el dato que haya en ese puesto y el “.remove(elemento)” que elimina el elemento que se mencionó en los paréntesis.



```
1  if condición:
2      # Bloque de código
3  elif condición:
4      # Bloque de código alternativo
```



```
1  número = 10
2
3  if número > 10:
4      print("El número es mayor que 10")
5  elif número == 10:
6      print("El número es igual a 10")
7  else:
8      print("El número es menor que 10")
```

- Largo de una lista: Para saber cuántos elementos hay en la lista ocupamos el método “.len(mi_lista)”.
- Rebanado(slicing): Puedes acceder a una porción de una lista utilizando el operador de rebanado “[:]”.

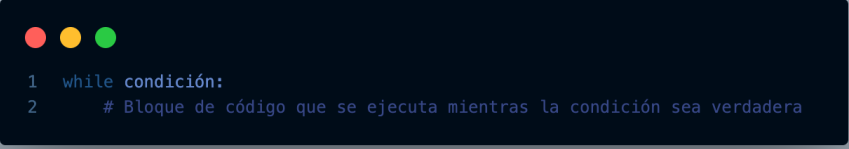


```
1 for elemento in colección:  
2     # Bloque de código que se ejecuta para cada elemento de la colección
```

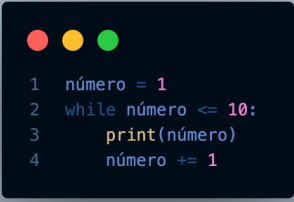


```
1 for número in range(1, 11):  
2     print(número)
```

- Concatenación de listas: Se pueden unir dos o más listas utilizando el operador de concatenación “+”.
- Ordenar una lista: Para ordenar una lista de manera ascendente o descendente se pueden utilizar los métodos “.sort()” o “.sorted()” respectivamente.



```
1 while condición:
2     # Bloque de código que se ejecuta mientras la condición sea verdadera
```



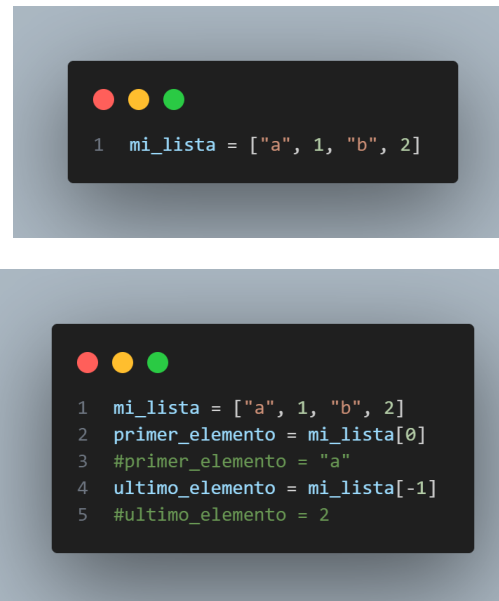
```
1 número = 1
2 while número <= 10:
3     print(número)
4     número += 1
```

- Buscar elementos: Se puede buscar un elemento de la lista utilizando el operador “in” o el método “.index(elemento)”.

6.2. Tuplas y sus propiedades

Si bien las tuplas son un conjunto de datos similares a las listas estas son inmutables esto quiere decir que una vez creada no se puede volver a modificar, algunas de sus propiedades son:

- Creación de tuplas: Primero que nada para crear una tupla solo se necesita de paréntesis “()” y separar los datos por comas “,”, aunque también se pueden crear solo separando los elementos con comas.
- Acceso a elementos: Al igual que en las listas para acceder a un elemento en específico solo se necesita de corchetes “[índice]”.
- Datos inmutables: Como dije anteriormente los datos no se pueden cambiar por lo que si seleccionamos un elemento y lo tratamos de igualar a otro nos tirará un error.
- Largo de una tupla: Para saber el largo de la tupla tenemos que utilizar la función “len(tupla)”.



- Rebanado (slicing): Esta propiedad sirve para seleccionar una porción de la tupla creando una nueva con estos datos y para esto necesitamos utilizar el operador de rebanado “[:]”.
- Concatenación de tuplas: Del mismo modo que en las listas para concatenar las tuplas se necesita del operador de concatenación “+”.
- Asignación de múltiple: Este quiere decir que en una sola línea de código se puede asignar a distintas variables los elementos de la tupla.

6.3. Diccionarios y métodos de acceso

Los diccionarios en vez de las tuplas y listas contienen pares de datos que se identifican como clave-valor cada clave dentro del diccionario tiene un único valor, esto sirve para organizar todo de mejor manera y acceder a los valores de una forma mas rápida como por ejemplo:

- Creación de diccionarios: Primero que nada para crear un diccionario necesitamos utilizar llaves “{}” y dentro separando clave-valor por “:” y cada par por “,”.
- Acceso al valor: Esta vez al igual que en las anteriores se ocupan corchetes “[clave]” pero en vez de poner un índice esta vez se pone la clave para obtener su respectivo valor.
- Modificación del valor: Para modificar un valor solo necesitamos seleccionar su clave e igualarlo al nuevo valor que queremos que obtenga.


A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. It contains four lines of Python code:

```
1 mi_lista = ["a", 1, "b", 2]
2 mi_lista[3] = 35
3 #mi_lista = ["a", 1, "b", 35]
4
```


A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. It contains three lines of Python code:

```
1 mi_lista = ["a", 1, "b", 2]
2 mi_lista.append("c")
3 #mi_lista = ["a", 1, "b", 2, "c"]
```

- Agregar pares: Si queremos agregar nuevos pares de clave-valor solo necesitamos seleccionar una nueva clave del diccionario e igualarlo a un nuevo valor.
- Eliminación de pares: Cuando queramos eliminar un par que este en el diccionario solo debemos utilizar la declaración “del” y especificar la clave.
- Verificación de existencia de clave: Para verificar si una clave en específico existe dentro de un diccionario podemos utilizar el operador “in”.
- Obtener clave-valor: Para obtener o todas las claves o todos los valores de un diccionario podemos utilizar los métodos “.keys()” o “.values()” respectivamente, pero si queremos obtener el par de elementos utilizaremos el método “.items()”.



```
1 mi_lista = ["a", 1, "b", 2]
2 mi_lista.insert(3, "x")
3 #mi_lista = ["a", 1, "b", "x", 2]
```



```
1 mi_lista = ["a", 1, "b", 2]
2 mi_lista.pop(2)
3 #mi_lista = ["a", 1, 2]
```

7. Funciones y modularidad

Las funciones son bloques de código reutilizables que realizan una tarea específica. Permiten dividir un programa en partes más pequeñas y manejables, lo que facilita la comprensión y el mantenimiento del código. La modularidad se refiere a la práctica de dividir un programa en módulos o funciones independientes que pueden ser desarrolladas y probadas de forma separada. Esto promueve el código limpio y organizado, facilitando la colaboración en equipos de desarrollo.

7.1. Definición y llamada de funciones

Definición de Funciones:

En la mayoría de los lenguajes de programación, las funciones se definen con la palabra clave “def” (en Python), “function” (en JavaScript), “fun” (en Kotlin), o “void” (en C++), seguido del nombre de la función y una lista de parámetros entre paréntesis. Por ejemplo, en Python:



Llamada de Funciones:

Para utilizar una función, se realiza una llamada a la función, pasando los valores necesarios como argumentos. Los argumentos son los valores reales que se pasan a la función durante la llamada. Por ejemplo:

En este caso, “Juan” es el argumento que se pasa a la función “saludar”.

7.2. Argumentos y parámetros

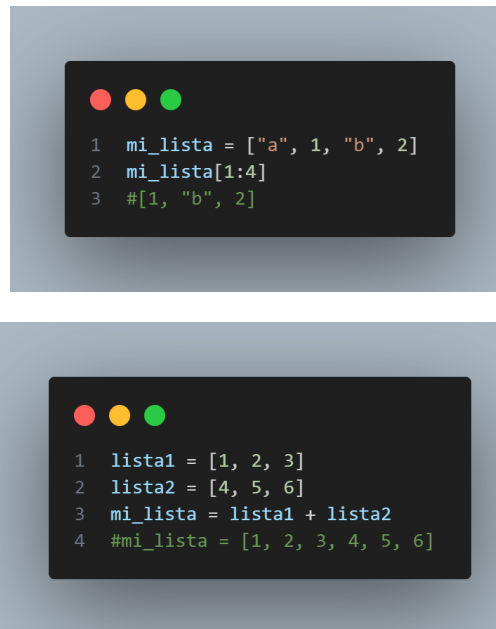
Parámetros de Funciones:

Los parámetros son variables que se utilizan en la definición de la función para aceptar valores. En el ejemplo anterior, “nombre” es un parámetro de la función “saludar”.

Argumentos de Funciones:

Los argumentos son los valores reales que se pasan a la función durante su llamada. En el ejemplo de la llamada a la función ‘saludar("Juan")’, “Juan” es el argumento que se pasa al parámetro “nombre” de la función “saludar”

Las funciones pueden tener múltiples parámetros y se pueden pasar argumentos de diferentes tipos (números, cadenas, listas, etc.). Por ejemplo:



En este caso, la función “sumar” tiene dos parámetros “a” y “b”, y se llama con los argumentos 3 y 5, respectivamente. El resultado de la suma se almacena en la variable “resultado_suma”.

Las funciones y la modularidad son conceptos fundamentales en programación, ya que permiten escribir código más eficiente, fácil de entender y mantener. La práctica constante con estos conceptos te ayudará a mejorar tus habilidades de programación.

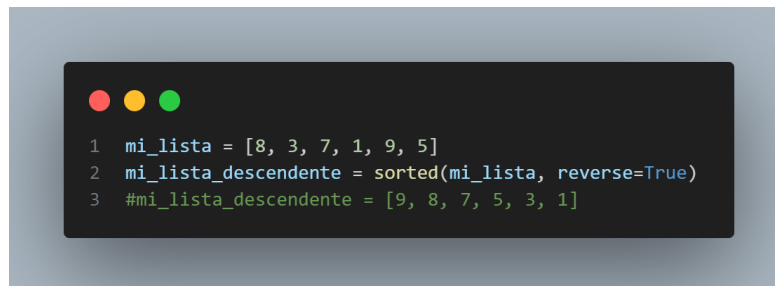
7.3. Módulos y su importación

Los módulos son archivos que contienen definiciones y declaraciones de funciones, clases y variables en Python. Ayudan a organizar el código en archivos separados para hacerlo más legible y reutilizable. Para utilizar las funciones y variables definidas en un módulo en otro archivo, se necesita importar el módulo en el archivo donde se desea utilizar.

- Creación de un Módulo: Supongamos que tenemos un archivo llamado `mi_modulo.py` con la siguiente definición de función:

A terminal window with a dark background and three colored window control buttons (red, yellow, green) at the top left. It contains three lines of Python code:

```
1 mi_lista = [8, 3, 7, 1, 9, 5]
2 mi_lista.sort()
3 #mi_lista = [1, 3, 5, 7, 8, 9]
```

A terminal window with a dark background and three colored window control buttons (red, yellow, green) at the top left. It contains three lines of Python code:

```
1 mi_lista = [8, 3, 7, 1, 9, 5]
2 mi_lista_descendente = sorted(mi_lista, reverse=True)
3 #mi_lista_descendente = [9, 8, 7, 5, 3, 1]
```

- Importación de un Módulo: En otro archivo Python, podemos importar y utilizar la función saludar del módulo `mi_modulo.py` de la siguiente manera:

También se puede importar una función específica de un módulo para evitar usar el nombre del módulo cada vez que se llama a la función:

La importación de módulos es esencial para organizar proyectos grandes y complejos en Python, ya que permite dividir el código en archivos manejables y fácilmente comprensibles. Además, facilita la reutilización del código, ya que las funciones y variables definidas en un módulo pueden ser utilizadas en varios archivos del proyecto.

8. Manejo de excepciones y errores

8.1. Introducción a las excepciones

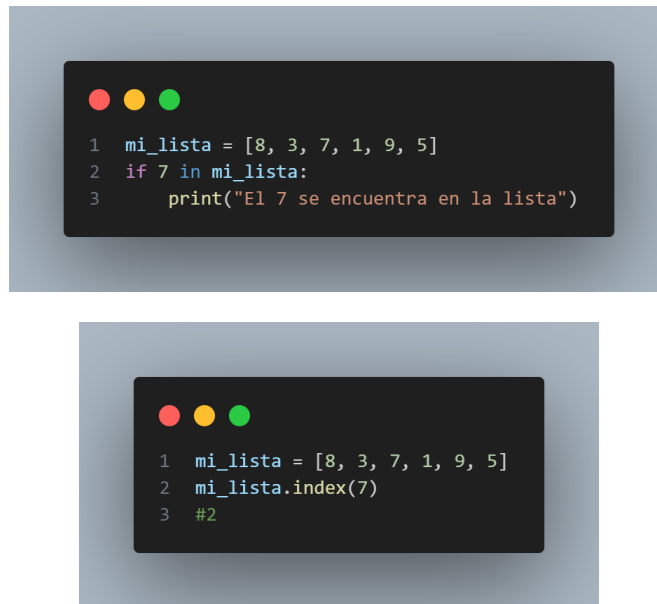
8.2. Uso de `try`, `except`, `else`

8.3. Tipos de excepciones y manejo específico

9. Entrada y salida de datos

9.1. Función `print()`

La función `print()` en Python es una herramienta fundamental para mostrar información en la consola o en la salida estándar del programa. Su propósito



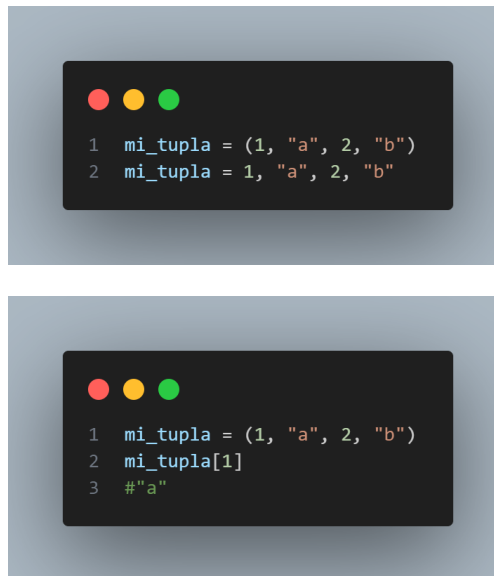
principal es facilitar a lo programadores imprimir mensajes, variables, listas y otros datos que permiten observar el comportamiento del código.

- Salida de texto: `print()` Se utiliza para mostrar cadenas de textos en la consola. Esto se útil para comunicar mensajes, indicadores o información relevante durante la ejecución del programa.
- Imprimir variables: Imprimir variables es una práctica común en programación, ya que te permite observar el contenido de las variables en diferentes momentos de la ejecución de tu programa. Esto es esencial para el seguimiento del flujo de datos en tu código, a continuación se muestran algunos ejemplos.
 - Ejemplo 1 : Para imprimir el valor de una variable, simplemente incluye a la variable como argumento de la función.
 - Ejemplo 2 : Concatenar variables de texto. Puedes combinar el valor de una variable con texto utilizando el operador de concatenación(+). esto es útil para crear mensajes informativos

9.2. Función `input()`

Ahora cómo enviar un dato mediante el teclado para que el programa lo tome en cuenta mediante el método `input()`

- Ejemplo 1



Esta función permite obtener el texto escrito por el usuario, al cual se le asignará un espacio a la memoria con la variable que el programador crea conveniente. Al llegar a la línea del comando, la consola esperara respuesta. Cuando el usuario escriba algo y presione la tecla Enter, el código seguirá ejecutándose.

Lo que estamos indicando al programa es que la variable «nombre» va a tomar el valor que el usuario ingrese cuando se le muestre el mensaje «Hola ¿Cuál es tu nombre?», para posteriormente, responder con otro mensaje y el valor que se ingresó. Debemos tener en cuenta que al usar `input()`, los datos ingresados siempre serán guardados como tipo `string`. Si necesitáramos ingresar números para utilizarlos en alguna operación matemática, debemos convertirlos a un tipo de dato adecuado (por ejemplo `int` o `float`, dependiendo si requerimos decimales). Podemos hacerlo de las siguientes maneras

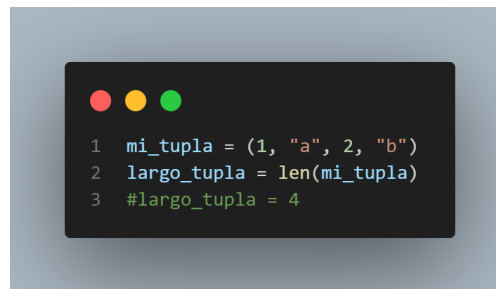
9.3. Formateo de cadenas

Python ofrece varias formas de formatear cadenas, y dos de los más comunes son mediante f-strings (cadenas formateadas) y métodos de formatos.

- Utilizan la sintaxis de cadenas con una "f" o "F"
- Permiten incrustar expresiones o variables mediante llaves "{}"
- Pueden realizar cambios y formateos dentro de la llave.
- Son una forma más legibles y eficiente de formatear cadenas en python.

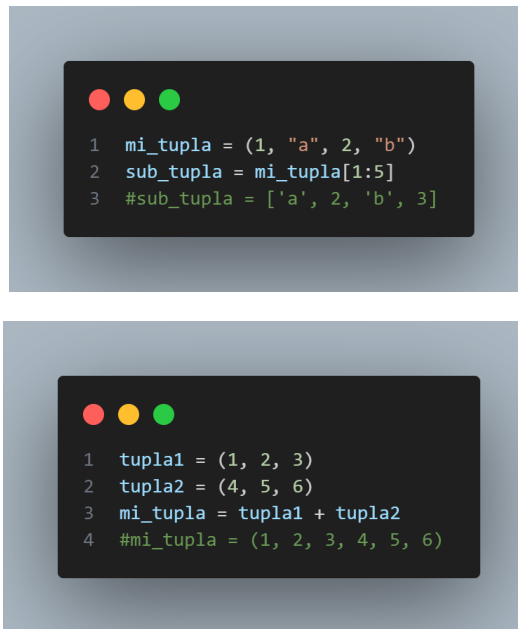


```
1 mi_tupla = (1, "a", 2, "b")
2 mi_tupla[1] = "x"
3 #Genera un error
```



```
1 mi_tupla = (1, "a", 2, "b")
2 largo_tupla = len(mi_tupla)
3 #largo_tupla = 4
```

En python, una cadena de texto normal se escribe entre comillas (""), para crear f-strings solo tienes que agregar la letra f o F mayúscula antes de la comilla.



¿Cómo imprimir variables usando f-strings?

¿Cómo evaluar expresiones en python con f-string?

Como las f-Strings son evaluadas al momento de ejecución (cuando se está ejecutando el código Python), bien podrías usar una f-string para evaluar expresiones válidas.

En el siguiente ejemplo, `num1` y `num2` son nuestras variables, para obtener el producto de estas variables únicamente basta con escribir la siguiente expresión `num1 * num2` dentro de llaves

En cualquier f-string, variable nombre y expresión se sustituirán por los valores que representan o por el valor resultante de una operación en el momento de ejecución (cuando se está ejecutando el código Python)

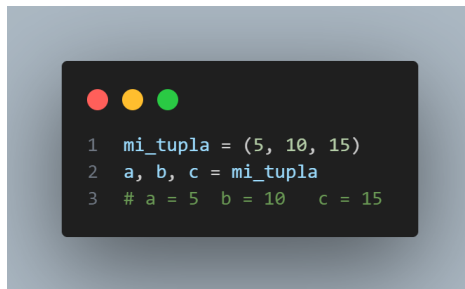
10. Trabajo con archivos

10.1. Apertura, lectura y escritura de archivos

Para trabajar con archivos en Python, se utiliza la función `open()`. Esta función toma dos argumentos: el nombre del archivo y el modo de apertura.

El modo de apertura determina cómo se puede acceder al archivo. Los modos de apertura más comunes son:

- `r`: Lectura. El archivo se abre en modo de lectura.



```
1 mi_tupla = (5, 10, 15)
2 a, b, c = mi_tupla
3 # a = 5 b = 10 c = 15
```



```
1 mi_diccionario = {"Nombre": "Maria Jose", "Edad": 17, "Signo": "Capricornio"}
```

- w: Escritura. El archivo se abre en modo de escritura. Si el archivo no existe, se crea uno nuevo. Si el archivo existe, se sobrescribe su contenido.
- a: Adición. El archivo se abre en modo de adición. Los datos se escriben al final del archivo.

Por ejemplo, para abrir un archivo de texto en modo de lectura, se puede usar el siguiente código:

Una vez que un archivo está abierto, se puede leer su contenido usando la función `read()`. Esta función devuelve una cadena con todo el contenido del archivo.

Por ejemplo, para leer el contenido de un archivo de texto, se puede usar el siguiente código:

También se puede leer el contenido de un archivo de texto línea por línea usando un bucle `for`:

Para escribir datos en un archivo, se usa la función `write()`. Esta función toma una cadena como argumento y la escribe al final del archivo.

Por ejemplo, para escribir el siguiente texto en un archivo de texto:
Se puede usar el siguiente código:

10.2. Operaciones comunes de archivos

Además de la apertura, lectura y escritura, Python ofrece otras operaciones comunes de archivos.

- `seek()`: Permite mover el puntero de lectura o escritura a una posición determinada en el archivo.



```
1 mi_diccionario = {"Nombre": "Maria Jose", "Edad": 17, "Signo": "Capricornio"}
2 mi_nombre = mi_diccionario["Nombre"]
3 #mi_nombre = "Maria Jose"
```



```
1 mi_diccionario = {"Nombre": "Maria Jose", "Edad": 17, "Signo": "Capricornio"}
2 mi_diccionario["Signo"] = "Cancer"
3 #mi_diccionario = {"Nombre": "Maria Jose", "Edad": 17, "Signo": "Cancer"}
```

- `tell()`: Devuelve la posición actual del puntero de lectura o escritura en el archivo.
- `flush()`: Escribe cualquier dato que aún no se haya escrito en el archivo al disco.
- `close()`: Cierra el archivo y libera los recursos asociados con él.

10.3. Manejo de archivos con el bloque “with”

El bloque `with` es una forma segura y conveniente de abrir y cerrar archivos. Cuando se usa el bloque `with`, el archivo se abre automáticamente al principio del bloque y se cierra automáticamente al final del bloque.

Por ejemplo, el siguiente código abre un archivo de texto en modo de lectura y luego imprime su contenido:

Por ejemplo, el siguiente código abre un archivo de texto en modo de lectura y luego imprime su contenido:

El uso del bloque `with` evita errores comunes, como olvidar cerrar un archivo abierto.



```
1 mi_diccionario = {"Nombre": "Maria Jose", "Edad": 17, "Signo": "Capricornio"}
2 mi_diccionario["Ciudad"] = "Las Garzas"
3 #mi_diccionario = {"Nombre": "Maria Jose", "Edad": 17, "Signo": "Capricornio", "Ciudad": "Las Garzas"}
```



```
1 mi_diccionario = {"Nombre": "Maria Jose", "Edad": 17, "Signo": "Capricornio"}
2 del(mi_diccionario["Edad"])
3 #mi_diccionario = {"Nombre": "Maria Jose", "Signo": "Capricornio"}
```



```
1 mi_diccionario = {"Nombre":"Maria Jose", "Edad":17, "Signo":"Capricornio"}
2 if "Nombre" in mi_diccionario:
3     print("El par clave-valor si se encuentra en la lista")
```



```
1 mi_diccionario = {"Nombre":"Maria Jose", "Edad":17, "Signo":"Capricornio"}
2 claves = mi_diccionario.keys()
3 #clves = (['Nombre', 'Edad', 'Signo'])
4 valores = mi_diccionario.values()
5 #valores = (['Maria Jose', 17, 'Capricornio'])
6 pares = mi_diccionario.items()
7 #pares = (('Nombre', 'Maria Jose'), ('Edad', 17), ('Signo', 'Capricornio'))
```

11. Introducción a la programación orientada a objetos

11.1. Definición de clases y objetos

11.2. Métodos y atributos de clase

12. Clases y objetos

12.1. Encapsulación y visibilidad

12.2. Métodos especiales

12.3. Herencia y composición

13. Herencia y polimorfismo

13.1. Herencia simple y múltiple

13.2. Polimorfismo y sobrecarga de métodos

13.3. Clases abstractas e interfaces

14. Módulos y paquetes

14.1. ¿Qué son los Módulos en Python?

14.2. Creación y Uso de Módulos

14.3. Importación de Módulos

14.4. Módulos Estándar de Python

14.5. ¿Qué son los Paquetes en Python?



```
1 def saludar(nombre):  
2     print("Hola, " + nombre + "!")
```



```
1 saludar("Juan")
```




```
1 def saludar(nombre):
```




```
1 def sumar(a, b):  
2     resultado = a + b  
3     return resultado  
4  
5 resultado_suma = sumar(3, 5)
```




```
1 # mi_modulo.py  
2  
3 def saludar(nombre):  
4     print("Hola, " + nombre + "!")
```




```
1 # archivo_principal.py
2 import mi_modulo
3
4 mi_modulo.saludar("Juan") # Output: Hola, Juan!
```




```
1 # archivo_principal.py
2 from mi_modulo import saludar
3
4 saludar("Juan") # Output: Hola, Juan!
```




```
1 print("holaaa")
2 >> holaa
```




```
1 numero = 42
2 print(numero)
3 >> 42
```




```
1 nombre = "pedro "  
2 print("hola mi nombre es " + nombre)  
3 >> hola mi nombre es pedro
```



```
1 nombre = input("hola ¿cual es tu nombre?----> ") #Benjamin  
2 print("hola",nombre,"como estas ")  
3 >> hola Benjamin, como estas
```



```
1 edad = int(input("¿Que edad tienes?----> "))  
2 print("Tu edad es",edad,"años ")  
3 >> Tu edad es 17 años
```



```
1 nombre = "pedro"  
2 edad = 42  
3 print(f"holaa, me llamo {nombre} y mi edad es {edad}")  
4 >> holaa, me llamo pedro y mi edad es 42
```




```
1 num1 = 83
2 num2 = 9
3 print(f"El producto de {num1} y {num2} es {num1 * num2}.")
4 >> El producto de 83 y 9 es 747
```