Strings en Python (Cadenas de Texto)

En Python, una cadena de texto o string es una secuencia ordenada e inmutable de caracteres que se utiliza para representar información textual. Los strings pueden contener letras, números, símbolos, espacios o cualquier combinación de caracteres Unicode. Son uno de los tipos de datos fundamentales del lenguaje y se utilizan constantemente en programación, ya sea para mostrar mensajes, almacenar nombres, procesar datos o manipular archivos de texto.

Los strings se definen encerrando el contenido entre comillas simples (' ') o comillas dobles (" "), sin que exista diferencia funcional entre ambas. Por ejemplo: 'Hola' y "Hola" son exactamente lo mismo. También es posible definir cadenas multilínea utilizando triple comilla (''' ''' o """ """), lo cual es útil para textos largos, documentación o impresión de párrafos.

Una característica esencial de los strings en Python es que son inmutables, lo que significa que no se pueden modificar directamente una vez creados. Cualquier operación que parezca alterar una cadena en realidad genera una nueva cadena, dejando intacta la original. Esta propiedad garantiza mayor seguridad y consistencia al manipular texto, especialmente en aplicaciones grandes.

Operaciones y Manipulación de Strings

A pesar de su inmutabilidad, Python ofrece una amplia gama de operaciones y métodos incorporados para manipular strings de manera poderosa y flexible. A continuación, se describen las más utilizadas:

1. Concatenación: Permite **unir dos o más strings** usando el operador +. Es una forma sencilla de construir mensajes o frases.

```
Nombre = "Juan"

Apellido = "Pérez"

NombreCompleto = Nombre + " " + Apellido  # "Juan Pérez"
```

2. Repetición: Usando el operador *, un string puede repetirse múltiples veces.

```
Eco = "Hola " * 3  # "Hola Hola Hola "
```

3. Interpolación (f-strings): A partir de Python 3.6, se pueden insertar variables dentro de strings usando una sintaxis clara y directa con f antes de las comillas.

```
Edad = 25

Mensaje = f"Hola, me llamo {Nombre} y tengo {Edad} años."  # "Hola, me llamo Juan y tengo 25 años."
```

4. Indexación y slicing: Se puede acceder a los caracteres de un string usando índices (empezando en 0). Además, con *slicing*, se pueden extraer fragmentos.

```
Palabra = "Python"

PrimeraLetra = Palabra[0]        # "P"

UltimaLetra = Palabra[-1]        # "n"

Subcadena = Palabra[1:4]         # "yth"
```

Métodos Comunes de Strings

Python incluye métodos muy útiles para transformar y analizar strings. Algunos de los más utilizados son:

- .title() → Convierte la primera letra de cada palabra a mayúscula.
- .upper() → Convierte toda la cadena a mayúsculas.
- .lower() → Convierte toda la cadena a minúsculas.
- .capitalize() → Pone solo la primera letra de la cadena en mayúscula.
- .strip() → Elimina espacios en blanco al principio y al final.
- .replace(a, b) → Reemplaza todas las apariciones de a por b.
- .split(sep) → Divide la cadena en una lista usando un separador.
- .join(lista) → Une elementos de una lista con el string como separador.
- .find(sub) → Devuelve el índice de la primera aparición de una subcadena.
- .count(sub) → Cuenta cuántas veces aparece una subcadena.
- .startswith(sub) / .endswith(sub) → Verifica si comienza o termina con cierta subcadena.

Números en Python

Python permite trabajar con diferentes tipos de números de forma flexible y sencilla. No requiere declarar el tipo de dato de forma explícita, ya que el propio lenguaje se encarga de reconocer automáticamente si se trata de un número entero, flotante o complejo. Los números en Python pueden utilizarse tanto para operaciones matemáticas simples como para cálculos científicos o financieros más avanzados.

Existen tres tipos numéricos fundamentales en Python: los enteros (int), los flotantes (float) y los complejos (complex).

Los **enteros** son números sin parte decimal, positivos o negativos. Algunos ejemplos típicos son: 5, -10, 0. Estos se usan comúnmente para contar elementos, trabajar con índices, realizar bucles, etc.

Los **flotantes** representan números con parte decimal y permiten mayor precisión en los cálculos. Ejemplos: 3.14, -0.01, 10.0. Este tipo se usa frecuentemente en operaciones que requieren valores decimales, como cálculos financieros, medidas, promedios, etc.

Los **números complejos** incluyen una parte real y una parte imaginaria, y se representan con la letra j. Un ejemplo sería: 2 + 3j. Este tipo de número es útil en contextos matemáticos avanzados, como álgebra compleja, procesamiento de señales o simulaciones físicas. Se puede acceder a sus componentes mediante los atributos .real y .imag.

Python incluye soporte directo para operadores aritméticos básicos como suma (+), resta (-), multiplicación (*), división (/), división entera (//), módulo o resto (%) y potenciación (**). A continuación, un ejemplo simple de su uso:

```
A = 10
B = 3
Suma = A + B            # 13
Resta = A - B           # 7
Multiplicacion = A * B  # 30
Division = A / B        # 3.333...
DivisionEntera = A // B # 3
Resto = A % B           # 1
Potencia = A ** B       # 1000
```

Además de estas operaciones básicas, Python permite realizar cálculos más complejos mediante el módulo math. Este módulo incluye funciones matemáticas avanzadas como raíces cuadradas, logaritmos, trigonometría, redondeos y constantes matemáticas como pi y e. Para utilizarlo, se debe importar previamente:

```
import math
RaizCuadrada = math.sqrt(16)       # 4.0
Potencia = math.pow(2, 3)          # 8.0
Seno = math.sin(math.pi / 2)       # 1.0
Logaritmo = math.log(100, 10)      # 2.0
ValorAbsoluto = math.fabs(-5)      # 5.0
RedondeoAbajo = math.floor(3.7)    # 3
RedondeoArriba = math.ceil(3.2)    # 4
```

Python también permite convertir entre tipos numéricos usando funciones integradas. Con int() se convierte a entero, eliminando los decimales; con float() se transforma a flotante, y con complex() se convierte a número complejo. Ejemplo:

```
Numero = 5.8
Entero = int(Numero)          # 5
Decimal = float(7)            # 7.0
Complejo = complex(4)         # (4+0j)
```

Una buena práctica al trabajar con números en Python es tener en cuenta la precisión de los flotantes, ya que algunos valores decimales no pueden representarse con exactitud en binario. Si se requiere mayor precisión (por ejemplo, en cálculos financieros), se pueden utilizar módulos como decimal o fractions. Además, para cálculos científicos o numéricos de alto rendimiento, es recomendable trabajar con bibliotecas como numpy.

Constantes en Python

En Python no existe un mecanismo interno que impida modificar el valor de una variable, por lo tanto, **no hay constantes verdaderas** como en otros lenguajes de programación (por ejemplo, `const` en JavaScript o `final` en Java). Sin embargo, existe una **convención ampliamente aceptada** en la comunidad de Python que consiste en **escribir el nombre de las variables constantes completamente en mayúsculas**, a veces con guiones bajos para separar palabras,

para indicar que su valor **no debería ser modificado una vez asignado**. Este uso es puramente semántico, es decir, depende de que el programador respete dicha convención, ya que Python no impide cambiar su valor.

```
PI = 3.14159

GRAVEDAD = 9.81

VELOCIDAD_LUZ = 299_792_458  # metros por segundo
```

Este tipo de escritura permite que el código sea más legible, claro y profesional, especialmente cuando se trabaja en equipos o proyectos grandes. Las constantes se utilizan frecuentemente para representar valores universales o configuraciones que no deben cambiar, como tasas de conversión, coeficientes matemáticos, mensajes de error fijos o rutas a archivos clave.

A pesar de ser técnicamente variables, estas constantes son tratadas como inmutables por convención, y suelen definirse al comienzo del archivo o módulo, en la parte superior del script, justo después de las importaciones. Esto ayuda a centralizar los valores importantes y hace que el código sea más fácil de mantener. Por ejemplo:

```
import math


PI = math.pi

GRAVEDAD_TIERRA = 9.81

MAX_INTENTOS = 3

MENSAJE_BIENVENIDA = "Bienvenido al sistema"
```

Comentarios en Python

Los comentarios son una parte esencial del desarrollo en Python, ya que permiten explicar el propósito, la lógica o el funcionamiento del código sin afectar su ejecución. Sirven como documentación interna para ayudar tanto al programador actual como a futuros desarrolladores (incluso uno mismo al revisar código tiempo después). Un buen uso de comentarios mejora la legibilidad, facilita el mantenimiento del software y reduce la probabilidad de errores por malinterpretaciones.

En Python, existen principalmente dos tipos de comentarios: los de una sola línea y los comentarios multilínea (también conocidos como docstrings, cuando se usan dentro de funciones, clases o módulos).

Los comentarios de una sola línea comienzan con el símbolo # y se extienden hasta el final de la línea. Son los más comunes y se utilizan para explicar instrucciones específicas, aclarar intenciones del código, señalar advertencias, marcar tareas pendientes, entre otros.

Ejemplo:

```
# Calcular el área de un círculo
```

```
Radio = 5
Area = 3.14 * (Radio ** 2)  # Fórmula del área: π * r^2
```

Los comentarios de una línea también pueden colocarse al final de una línea de código, siempre y cuando no afecten su claridad. Sin embargo, se recomienda no abusar de esta práctica si dificulta la lectura.

Los **comentarios multilínea** son útiles cuando se necesita explicar una sección de código más extensa o proporcionar una descripción general de un archivo, una función o una clase. Aunque no existe una sintaxis exclusiva para comentarios multilínea como en otros lenguajes, en Python se utiliza una cadena de texto entre comillas triples (''' o """). Si esta cadena no se asigna a ninguna variable ni se devuelve, Python simplemente la ignora en tiempo de ejecución, funcionando como un comentario.

Ejemplo:

```
"""
Este script calcula el área de diferentes figuras geométricas.
Usa funciones específicas para cada forma y pide datos al usuario.
Autor: Santino Rodriguez
Fecha: 2025-03-31
"""
```

También se utilizan comillas triples para definir **docstrings**, que son cadenas de documentación asociadas directamente a funciones, clases o módulos. A diferencia de los comentarios comunes, los docstrings **sí forman parte del objeto** al que están asociados, y se pueden acceder mediante la función help() o el atributo .__doc__.

Ejemplo de docstring en una función:

```
def CalcularAreaCirculo(Radio):
    """
    Calcula el área de un círculo dado su radio.
    Parámetro:
        Radio (float): el radio del círculo.
    Retorna:
        float: el área calculada.
    """
    return 3.14159 * (Radio ** 2)
```

Para comentarios temporales, anotaciones internas o planificación de código futuro, también es común usar etiquetas como # TODO:, # FIXME:, # IMPORTANTE: o # NOTA:. Estas etiquetas permiten localizar fácilmente puntos pendientes mediante herramientas de desarrollo o búsquedas en el editor.

Ejemplo:

```
# TODO: implementar validación de entrada del usuario

# FIXME: corregir el cálculo en casos negativos
```

Listas en Python

Las listas son uno de los tipos de estructuras de datos más utilizados en Python. Se trata de colecciones ordenadas, mutables y heterogéneas, lo que significa que pueden almacenar varios elementos en un orden específico, pueden modificarse después de ser creadas y pueden contener distintos tipos de datos (por ejemplo, enteros, cadenas, flotantes, listas anidadas, etc.).

Se definen mediante corchetes [] y los elementos se separan por comas:

```
Colores = ["Rojo", "Verde", "Azul"]

Numeros = [1, 2, 3, 4, 5]

Mixta = ["Texto", 3.14, True, None]
```

Las listas son mutables, lo que significa que se pueden modificar: agregar, eliminar, cambiar o reordenar elementos sin necesidad de crear una nueva lista.

Algunos de los métodos más comunes y útiles que permiten manipular listas son los siguientes:

- append(Elemento): agrega un elemento al final de la lista.
- insert(Posicion, Elemento): inserta un elemento en una posición específica.
- remove(Elemento): elimina la primera aparición del elemento indicado.
- pop([Indice]): elimina y devuelve el elemento en la posición dada (por defecto, el último).
- sort(): ordena la lista en orden ascendente (o con clave personalizada).
- reverse(): invierte el orden de los elementos.
- index(Elemento): devuelve el índice de la primera aparición del elemento.
- count(Elemento): cuenta cuántas veces aparece un elemento.

Ejemplo práctico:

```
Colores = ["Rojo", "Verde", "Azul"]

Colores.append("Amarillo")              # ["Rojo", "Verde",
"Azul", "Amarillo"]

Colores.insert(1, "Negro")              # ["Rojo", "Negro",
"Verde", "Azul", "Amarillo"]

Colores.remove("Verde")                 # ["Rojo", "Negro",
"Azul", "Amarillo"]

Ultimo = Colores.pop()                  # Elimina "Amarillo"
```

```
Colores.sort()                          # Orden alfabético:
["Azul", "Negro", "Rojo"]

Colores.reverse()                       # Inversión: ["Rojo",
"Negro", "Azul"]

Indice = Colores.index("Azul")          # 2

Cantidad = Colores.count("Rojo")        # 1
```

Es importante saber que las listas son indexadas, es decir, se puede acceder a cada elemento usando su índice, comenzando desde 0. También permiten el uso de índices negativos para acceder desde el final hacia el principio.

```
PrimerColor = Colores[0]      # "Rojo"

UltimoColor = Colores[-1]     # "Azul"
```

Además, se pueden usar slices (rebanadas) para acceder a subconjuntos de la lista:

```
SubLista = Colores[1:3]       # ["Negro", "Azul"]
```

Las listas pueden recorrerse utilizando bucles for, lo cual es muy útil para procesar sus elementos:

```
for Color in Colores:

    print(Color.title())
```

También se pueden crear listas a partir de expresiones usando comprensiones de listas, una forma compacta y eficiente de construir listas nuevas:

```
Cuadrados = [X ** 2 for X in range(1, 6)]  # [1, 4, 9, 16, 25]
```

Python permite incluso listas anidadas, es decir, listas dentro de otras listas, lo cual es útil para representar estructuras más complejas como matrices o tablas:

```
Matriz = [[1, 2], [3, 4], [5, 6]]

Elemento = Matriz[1][0]  # Accede al valor 3
```

Tuplas en Python

Las tuplas son estructuras de datos muy similares a las listas, con la diferencia principal de que son inmutables, es decir, una vez creadas, no se pueden modificar. Esto significa que no se pueden agregar, eliminar ni cambiar elementos de una tupla. Se definen utilizando paréntesis () en lugar de corchetes [].

Se utilizan en situaciones donde se necesita garantizar que los datos no se alteren, lo cual aporta seguridad, claridad semántica y mejor rendimiento en comparación con las listas.

Ejemplo básico:

```
Dimensiones = (800, 600)

print(Dimensiones[0])  # Salida: 800
```

```
print(Dimensiones[1])  # Salida: 600
```

Las tuplas pueden contener cualquier tipo de dato, incluso combinaciones heterogéneas (enteros, cadenas, listas, otras tuplas, etc.):

```
TuplaMixta = ("Texto", 42, 3.14, True)
```

Aunque son inmutables, se puede acceder a sus elementos mediante índices, exactamente igual que en las listas. También soportan slicing (rebanado), búsqueda con in, conteo y otras operaciones que no alteren su contenido:

```
Colores = ("Rojo", "Verde", "Azul")

print("Verde" in Colores)           # True

print(Colores.count("Rojo"))        # 1

print(Colores.index("Azul"))        # 2

print(Colores[:2])                  # ("Rojo", "Verde")
```

Si una tupla contiene objetos mutables (como listas), estos objetos sí pueden modificarse internamente, aunque la estructura general de la tupla no cambie:

```
Tupla = ([1, 2], "Inmutable")

Tupla[0].append(3)

print(Tupla)  # Salida: ([1, 2, 3], "Inmutable")
```

Las tuplas también se pueden definir sin paréntesis explícitos, separando los elementos con comas. Python interpretará automáticamente la expresión como una tupla:

```
Coordenadas = 10, 20

print(type(Coordenadas))  # <class 'tuple'>
```

Para definir una tupla de un solo elemento, es obligatorio colocar una coma final para que Python la reconozca como tal:

```
TuplaUnidad = (5,)

print(type(TuplaUnidad))  # <class 'tuple'>
```

Las tuplas son especialmente útiles en los siguientes casos:

1. **Datos que no deben cambiar**, como dimensiones de pantalla, coordenadas geográficas, fechas, configuraciones fijas, etc.
2. **Desempeño optimizado**: al ser inmutables, ocupan menos memoria y su acceso es más rápido que el de las listas.
3. **Seguridad en el código**: al no poder modificarse, se evita la alteración accidental de datos.
4. **Uso como claves en diccionarios**: las listas no pueden ser utilizadas como claves porque son mutables, pero las tuplas sí, ya que son hashables.

Ejemplo de tupla como clave en un diccionario:

```
Puntos = {
    (0, 0): "Origen",
    (1, 2): "Punto A",
    (3, 4): "Punto B"
}
print(Puntos[(1, 2)])  # Salida: "Punto A"
```

También se usan mucho para retornar múltiples valores desde una función, lo cual facilita la agrupación de resultados:

```
def ObtenerCoordenadas():
    return (100, 200)

X, Y = ObtenerCoordenadas()
print(X)  # 100
print(Y)  # 200
```

Condicionales en Python

Las estructuras condicionales permiten que un programa tome decisiones y ejecute distintas secciones de código según se cumplan o no ciertas condiciones lógicas. Son fundamentales para controlar el flujo del programa y hacer que reaccione de forma dinámica a diferentes situaciones.

En Python, la estructura básica de una condición utiliza la palabra clave if, seguida opcionalmente por elif (abreviación de “else if”) y else. Cada condición evalúa una expresión booleana, que retorna True o False.

Ejemplo básico:

```
Temperatura = 18

if Temperatura > 30:
    print("Hace Calor")
elif Temperatura < 10:
    print("Hace Frío")
else:
    print("Está Templado")
```

En este ejemplo, se evalúa la variable Temperatura y se imprime un mensaje diferente según su valor. El bloque que cumple la primera condición verdadera es el único que se ejecuta. Si ninguna condición es verdadera, se ejecuta el bloque else por defecto.

Python utiliza identación (sangría) para definir el contenido de cada bloque. Es muy importante respetarla, ya que no se utilizan llaves {} como en otros lenguajes.

Los **operadores de comparación** más comunes que se utilizan dentro de las condiciones son:

- ==: igual a
- !=: distinto de
- >: mayor que
- <: menor que
- >=: mayor o igual que
- <=: menor o igual que

También se pueden combinar condiciones usando **operadores lógicos**:

- and: verdadero si ambas condiciones son verdaderas
- or: verdadero si al menos una condición es verdadera
- not: invierte el valor de verdad (no)

Ejemplos de uso de operadores lógicos:

```
Edad = 25
Nacionalidad = "Argentina"

if Edad >= 18 and Nacionalidad == "Argentina":
    print("Puede votar en Argentina")

if Edad < 18 or Nacionalidad != "Argentina":
    print("No cumple los requisitos para votar")

if not Edad < 18:
    print("Es mayor de edad")
```

También se pueden realizar comparaciones entre cadenas de texto, listas, y otros tipos, siempre que el tipo de datos sea compatible:

```
Nombre = "Jorge"
if Nombre == "Jorge":
    print("Nombre reconocido")
```

Otra forma útil de usar condicionales es mediante expresiones condicionales en una sola línea (también llamadas ternarias):

```
Resultado = "Aprobado" if Nota >= 60 else "Desaprobado"
```

Por último, dentro de los bloques condicionales se puede incluir cualquier tipo de instrucción: cálculos, llamadas a funciones, estructuras repetitivas, etc., lo cual permite construir lógica compleja de forma clara y modular.

Diccionarios en Python

Los diccionarios son estructuras de datos en Python que permiten almacenar y acceder a valores mediante claves. A diferencia de las listas o tuplas, que utilizan índices numéricos, los diccionarios utilizan pares clave-valor, donde cada clave está asociada a un valor determinado. Son mutables, lo que significa que pueden modificarse después de ser creados, y desde Python 3.7 conservan el orden de inserción de los elementos.

Se definen utilizando llaves {} con los pares separados por dos puntos ::

```
Persona = {"Nombre": "Ana", "Edad": 25}
```

Cada clave debe ser única e inmutable (por ejemplo, cadenas, enteros o tuplas), mientras que los valores pueden ser de cualquier tipo, incluso listas u otros diccionarios.

Se puede acceder a los valores usando sus claves:

```
print(Persona["Nombre"])    # Salida: "Ana"
```

También se pueden modificar valores existentes o agregar nuevos pares:

```
Persona["Edad"] = 26                # Modifica la edad
Persona["Ciudad"] = "Córdoba"       # Agrega una nueva clave
```

Para evitar errores cuando no se está seguro si una clave existe, se puede usar el método .get(), que devuelve None si la clave no existe (o un valor por defecto):

```
Email = Persona.get("Correo", "No especificado")
```

Algunos de los métodos más útiles para trabajar con diccionarios son:

- .get(Clave, ValorPorDefecto): obtiene el valor asociado a una clave o devuelve un valor por defecto si no existe.
- .keys(): devuelve una vista con todas las claves.
- .values(): devuelve una vista con todos los valores.
- .items(): devuelve pares clave-valor como tuplas.
- .pop(Clave): elimina una clave específica y devuelve su valor.

Ejemplo práctico:

```
Estudiante = {
    "Nombre": "Lucía",
    "Edad": 22,
    "Materias": ["Matemática", "Historia"]
}

NombreEstudiante = Estudiante.get("Nombre")          # "Lucía"
MateriasInscritas = Estudiante["Materias"]           # ["Matemática", "Historia"]
Claves = list(Estudiante.keys())                     # ["Nombre", "Edad", "Materias"]
Valores = list(Estudiante.values())                  # ["Lucía", 22, ["Matemática", "Historia"]]
Pares = list(Estudiante.items())                     # [("Nombre", "Lucía"), ("Edad", 22), ("Materias", [...])]
Estudiante.pop("Edad")                               # Elimina "Edad"
```

Los diccionarios también permiten anidación, es decir, tener diccionarios dentro de otros diccionarios, lo que es útil para representar estructuras más complejas:

```
Usuarios = {
    "Usuario1": {"Nombre": "Carlos", "Edad": 30},
    "Usuario2": {"Nombre": "Marta", "Edad": 28}
}

print(Usuarios["Usuario1"]["Nombre"])  # "Carlos"
```

Además, se pueden recorrer usando bucles for, accediendo a claves, valores o ambos:

```
for Clave, Valor in Estudiante.items():
    print(f"{Clave.title()}: {Valor}")
```

Entrada de Datos con input() en Python

La función **input()** permite que un programa interactúe con el usuario solicitando datos por consola. Es una herramienta fundamental para hacer que el programa sea dinámico y responda a la información proporcionada externamente, en lugar de funcionar siempre con valores fijos.

Su sintaxis básica es:

```
Variable = input("Mensaje para el usuario: ")
```

El texto entre comillas se muestra en pantalla como mensaje o pregunta, y el valor ingresado por el usuario se almacena como una cadena de texto (str). Es importante recordar que, sin importar lo que el usuario escriba, input() siempre devuelve una cadena, por lo tanto, si se necesitan otros tipos de datos, es necesario hacer una conversión explícita.

Ejemplo de entrada numérica:

```
Edad = int(input("¿Cuántos Años Tenés? "))

print(f"Tenés {Edad} Años")
```

En este caso, la cadena ingresada se convierte en un entero utilizando int(). Si se esperara un número con decimales, se podría usar float() en lugar de int():

```
Altura = float(input("¿Cuál Es Tu Altura En Metros? "))
```

También es posible pedir varias entradas consecutivas y almacenarlas en distintas variables:

```
Nombre = input("¿Cuál Es Tu Nombre? ")

Ciudad = input("¿De Qué Ciudad Sos? ")

print(f"{Nombre.title()} Vive En {Ciudad.title()}")
```

Se puede combinar input() con condicionales para tomar decisiones en base a lo que el usuario escribe:

```
Respuesta = input("¿Te Gusta Python? (sí/no): ").lower()


if Respuesta == "sí":
    print("¡Genial! Python Es Muy Potente")
elif Respuesta == "no":
    print("Qué Lástima, Tal Vez Cambies De Opinión")
else:
    print("Respuesta No Reconocida")
```

También es común validar los datos ingresados para asegurar que tengan el formato correcto. Por ejemplo, evitar que el usuario ingrese letras cuando se espera un número. Para esto, se puede usar try y except para manejar errores de conversión:

```
try:
    Numero = int(input("Ingresá Un Número Entero: "))
    print(f"Ingresaste El Número {Numero}")
except ValueError:
    print("Eso No Es Un Número Válido")
```

Otra técnica útil es verificar si lo ingresado cumple con ciertas condiciones antes de aceptarlo:

```
Edad = input("¿Qué Edad Tenés? ")


if Edad.isdigit():
    Edad = int(Edad)
    print(f"Edad Aceptada: {Edad}")
else:
    print("Por Favor Ingresá Solo Números")
```

Como input() devuelve cadenas, se pueden aplicar métodos de string como .strip(), .lower(), .upper() o .title() directamente a la entrada:

```
Nombre = input("Ingresá Tu Nombre: ").strip().title()
```

Bucles en Python

En Python, los bucles o estructuras de repetición permiten ejecutar un bloque de código varias veces, ya sea recorriendo elementos de una secuencia o repitiendo mientras se cumpla una condición lógica. Esto es esencial para automatizar tareas repetitivas y procesar grandes cantidades de datos de forma eficiente.

Python ofrece dos tipos principales de bucles:

1. Bucle for

El bucle for se utiliza para recorrer secuencias, como listas, cadenas, tuplas, diccionarios o rangos numéricos. La sintaxis es clara y directa:

```
for Variable in Secuencia:
    # Código a ejecutar en cada iteración
```

Ejemplo con range():

```
for Intento in range(3):
    print(f"Intento {Intento + 1}")
```

Esto imprimirá tres líneas numeradas del 1 al 3. La función range(3) genera los valores 0, 1, 2, y en cada vuelta la variable Intento toma uno de esos valores.

Ejemplo recorriendo una lista:

```
Colores = ["Rojo", "Verde", "Azul"]


for Color in Colores:
    print(Color.title())
```

También se puede usar enumerate() para obtener el índice junto al valor:

```
for Indice, Color in enumerate(Colores):
    print(f"{Indice}: {Color.title()}")
```

2. Bucle while

El bucle while se utiliza para repetir un bloque de código mientras se cumpla una condición. La condición se evalúa antes de cada iteración, y el bucle se detiene cuando deja de cumplirse.

```
Intentos = 0
while Intentos < 5:
    Intentos += 1
    print(f"Intento Número {Intentos}")
```

Este bucle se ejecutará mientras la variable Intentos sea menor que 5. Es importante que haya alguna instrucción que modifique la condición, para evitar un bucle infinito.

Control de flujo con break y continue

- break: termina el bucle inmediatamente, sin importar si la condición aún se cumple.
- continue: salta la iteración actual y pasa a la siguiente.

Ejemplo con break:

```
for Numero in range(1, 10):
    if Numero == 5:
        break
    print(Numero)
```

Salida: 1 2 3 4 (el ciclo se detiene al llegar a 5)

Ejemplo con continue:

```
for Numero in range(1, 6):
    if Numero == 3:
        continue
    print(Numero)
```

Salida: 1 2 4 5 (se salta el 3)

Casos de uso comunes:

- Procesar todos los elementos de una lista
- Repetir hasta que el usuario escriba un dato válido
- Ejecutar un bloque un número determinado de veces
- Realizar operaciones acumulativas o de búsqueda
- Simular ciclos de prueba o intentos

Bucle while con validación de entrada:

```
Contraseña = ""
while Contraseña != "Python123":
    Contraseña = input("Ingresá La Contraseña Correcta: ")

print("Acceso Concedido")
```

Bucle for con acumulación:

```
Numeros = [4, 7, 1, 3]
Suma = 0

for Numero in Numeros:
    Suma += Numero

print(f"La Suma Total Es: {Suma}")
```

Funciones en Python

Las funciones en Python permiten agrupar instrucciones que se pueden reutilizar varias veces sin escribir el mismo código. Se definen con la palabra clave def seguida del nombre de la función, paréntesis y dos puntos. El cuerpo de la función debe estar indentado. Una vez definida, se puede llamar escribiendo su nombre seguido de paréntesis. Por ejemplo:

```
def Saludar():
    print("Hola, Bienvenido Al Programa")

Saludar()
```

El código anterior define una función llamada Saludar que imprime un mensaje al ejecutarse. La llamada a la función ocurre en la última línea y ejecuta el contenido de la función. Las funciones ayudan a mantener el código organizado y claro, especialmente cuando se debe realizar una misma acción en diferentes partes del programa. También pueden usarse para dividir un programa en pasos lógicos. Es posible escribir varias funciones en el mismo programa, y cada una puede realizar una tarea distinta. Por ejemplo:

```
def MostrarMenu():
    print("1. Ver Datos")
    print("2. Salir")
```

```
def MensajeSalida():
    print("Gracias Por Usar El Programa")
```

Cada función puede ser llamada cuando se la necesite, en cualquier parte del código mientras esté definida antes de su uso. También se puede documentar una función usando comillas triples para explicar su propósito, lo que se conoce como docstring:

```
def MostrarMensaje():
    """
    Esta Función Muestra Un Mensaje De Bienvenida
    """
    print("Bienvenido Al Sistema")
```

Aunque esta documentación no afecta la ejecución, sirve para que otros programadores entiendan mejor el código. Además, la función help() puede mostrar el contenido del docstring si se quiere consultar la descripción de la función desde la consola.

Parámetros y Argumentos Python

Las funciones en Python pueden recibir datos externos llamados parámetros, que se definen entre los paréntesis al momento de crear la función. Cuando se llama a la función, se pasan los valores concretos llamados argumentos. Estos permiten que la función trabaje con distintos datos sin necesidad de modificar su contenido. Por ejemplo:

```
def Saludar(Nombre):
    print(f"Hola, {Nombre.title()}")


Saludar("lucía")
```

En este caso la función recibe un parámetro llamado Nombre y al llamarla se le pasa el argumento "lucía". Es posible definir más de un parámetro separándolos con comas. Python permite asignar valores por defecto a los parámetros, de modo que si no se pasa ningún argumento al llamarla, se utilice ese valor. Por ejemplo:

```
def Saludar(Nombre="Usuario"):
    print(f"Hola, {Nombre.title()}")


Saludar()
Saludar("andrés")
```

El primer llamado imprimirá "Hola, Usuario" y el segundo "Hola, Andrés". También se pueden usar argumentos variables cuando no se sabe cuántos valores se van a pasar. Para eso se usa *args si se trata de una lista de argumentos posicionales, y **kwargs si se trata de un conjunto de argumentos con nombre. Por ejemplo:

```
def MostrarNombres(*Nombres):
    for Nombre in Nombres:
        print(Nombre.title())


MostrarNombres("ana", "juan", "sofia")
```

Y con **kwargs:

```
def MostrarDatos(**Datos):
    for Clave, Valor in Datos.items():
        print(f"{Clave.title()}: {Valor}")


MostrarDatos(Nombre="Carlos", Edad=28, Ciudad="Rosario")
```

Los parámetros y argumentos permiten que las funciones sean flexibles y reutilizables, adaptándose a diferentes situaciones sin necesidad de escribir código repetido.

Return en Python

El return en Python se utiliza dentro de una función para devolver un resultado al lugar donde fue llamada. Al ejecutarse, la función termina su ejecución inmediatamente y entrega el valor indicado después de return. Esto permite que el resultado sea guardado en una variable, mostrado en pantalla, o usado como parte de otra operación. Por ejemplo:

```
def Multiplicar(X, Y):
    return X * Y

Resultado = Multiplicar(4, 5)
print(Resultado)
```

En este caso la función Multiplicar devuelve el producto de los dos números y el valor se almacena en la variable Resultado. El uso de return permite separar el procesamiento de datos de su uso, lo cual hace el código más limpio y flexible. También es posible devolver cadenas, booleanos, listas u otros tipos de datos. Si no se especifica return, la función devuelve None por defecto. Además, return permite construir funciones que se pueden combinar entre sí, como por ejemplo:

```
def Doble(Numero):
    return Numero * 2

def Triple(Numero):
    return Numero * 3

Valor = Doble(3) + Triple(2)
print(Valor)
```

Una función también puede devolver el resultado de una condición o un cálculo más complejo. Incluso es posible devolver múltiples valores separados por comas, que serán agrupados automáticamente en una tupla:

```
def Operaciones(A, B):
    return A + B, A - B, A * B

Suma, Resta, Producto = Operaciones(10, 5)
print(Suma)
print(Resta)
print(Producto)
```

El uso adecuado de return permite reutilizar los resultados de una función en otras partes del programa, facilitando el trabajo con datos dinámicos y mejorando la claridad del código.

Módulos en Python

Un módulo en Python es un archivo con extensión .py que contiene funciones, variables o clases que pueden ser utilizadas en otros programas. Sirve para organizar el código y reutilizarlo sin tener que escribirlo nuevamente. Para usar un módulo se debe importar con la palabra clave import seguida del nombre del módulo. Por ejemplo:

```
import Math

print(Math.Pi)
```

En este caso se importa el módulo Math que contiene funciones matemáticas y se accede al valor de Pi. También se pueden importar partes específicas de un módulo utilizando la sintaxis from seguido del nombre del módulo, import y luego el elemento deseado. Por ejemplo:

```
from Math import Sqrt

print(Sqrt(25))
```

Esto importa únicamente la función Sqrt del módulo Math y permite usarla directamente sin escribir el nombre del módulo. Es posible renombrar un módulo al importarlo usando as para abreviar el nombre y hacerlo más cómodo de usar:

```
import Math as M

print(M.Factorial(5))
```

Existen módulos integrados en Python como random, datetime, os, y también se pueden crear módulos propios simplemente escribiendo funciones o clases en un archivo .py y luego importándolo desde otro archivo. Usar módulos mejora la estructura del código, evita repeticiones y permite dividir un programa grande en partes más manejables.

Clases en Python

En Python, una clase es una plantilla que define la estructura y el comportamiento de los objetos que se crearán a partir de ella. Dentro de una clase se pueden definir atributos, que son las variables propias del objeto, y métodos, que son funciones que describen sus acciones. La clase se define usando la palabra clave class seguida del nombre de la clase con la primera letra en mayúscula. Dentro de la clase, se define un método especial llamado **init** que se ejecuta automáticamente al crear un objeto y sirve para inicializar los atributos. Por ejemplo:

```
class Animal:

    def __init__(self, Especie):

        self.Especie = Especie


    def Hablar(self):

        print(f"Soy Un {self.Especie.title()}")
```

En este ejemplo se define una clase llamada Animal con un atributo llamado Especie y un método llamado Hablar. La palabra self se refiere al propio objeto y se usa para acceder a sus atributos y métodos. Para crear un objeto se llama a la clase como si fuera una función:

```
Mascota = Animal("perro")

Mascota.Hablar()
```

Esto crea un objeto llamado Mascota de tipo Animal con el valor "perro" en su atributo Especie y ejecuta el método Hablar mostrando el mensaje correspondiente. Las clases permiten reutilizar código mediante herencia, donde una clase hija hereda atributos y métodos de una clase padre. También permiten aplicar encapsulación, que consiste en ocultar detalles internos y exponer solo lo necesario a través de métodos públicos. Además, permiten el polimorfismo, donde diferentes clases pueden compartir métodos con el mismo nombre pero comportamientos distintos, lo que hace que el código sea más flexible y extensible. Las clases son una de las bases principales de la programación orientada a objetos y permiten organizar programas complejos de forma ordenada y modular.

Librerías en Python

Python cuenta con una amplia biblioteca estándar que incluye módulos ya integrados para realizar tareas comunes sin necesidad de instalar nada adicional. Estas librerías cubren áreas como matemáticas, generación de números aleatorios, manejo de fechas, manipulación de archivos, expresiones regulares, entre otras. Por ejemplo, el módulo random permite generar valores aleatorios y se puede usar de la siguiente forma:

```
import Random

print(Random.Randint(1, 100))
```

En este caso se importa la librería random y se utiliza la función randint para generar un número entero entre 1 y 100. Además de la biblioteca estándar, Python tiene un ecosistema muy amplio

de librerías externas creadas por la comunidad que permiten extender las funcionalidades del lenguaje. Estas librerías no vienen preinstaladas y se pueden descargar usando la herramienta pip desde la terminal. Por ejemplo, NumPy permite trabajar con arreglos y cálculos numéricos avanzados, Pandas se utiliza para análisis y manipulación de datos, Matplotlib para gráficos, Flask para desarrollo web, y muchas otras según el área de trabajo. Para instalar una librería se escribe:

```
pip install numpy
```

Una vez instalada, se importa en el programa como cualquier otro módulo:

```
import Numpy as Np

Arreglo = Np.Array([1, 2, 3])

print(Arreglo)
```

El uso de librerías externas permite ahorrar tiempo, evitar reinventar soluciones ya disponibles y aprovechar herramientas optimizadas y bien mantenidas por la comunidad. Python es muy valorado por su gran cantidad de librerías disponibles, lo que lo hace apto para tareas como inteligencia artificial, ciencia de datos, automatización, desarrollo web, gráficos, redes, pruebas, interfaces gráficas, entre muchas otras.

Programación Orientada a Objetos (POO) en Python

La Programación Orientada a Objetos (POO) es un paradigma de programación que organiza el código en torno a "objetos". Estos objetos combinan datos y funcionalidades y son instancias de clases. Las clases actúan como plantillas que definen las características (atributos) y comportamientos (métodos) de los objetos.

Una clase es una definición general de un tipo de objeto, mientras que un objeto es una instancia concreta de esa clase. Por ejemplo, una clase "Coche" puede definir atributos como marca, modelo y color, y métodos como arrancar y frenar. Luego, cada "Coche" creado es un objeto con sus propios valores para esos atributos.

Las principales ventajas de la Programación Orientada a Objetos son:

1. **Reutilización de código mediante herencia**: La herencia permite crear nuevas clases basadas en clases existentes, heredando sus atributos y métodos. Esto facilita la reutilización del código, evitando duplicaciones y promoviendo la extensión de funcionalidades.

2. **Mayor organización**: Al dividir el código en clases y objetos, el programa se organiza mejor, lo que facilita su mantenimiento y escalabilidad. Cada clase puede tener su propio conjunto de atributos y métodos que se enfocan en una tarea específica, lo que mejora la modularidad del código.

3. **Simulación de entidades del mundo real**: La POO permite modelar entidades del mundo real de manera más natural. Cada objeto puede tener propiedades y comportamientos que reflejan características del mundo físico o conceptual que se desea simular, lo que facilita la comprensión y diseño del sistema.

4. **Encapsulación**: Los datos de un objeto se encapsulan dentro de la clase, es decir, no pueden ser modificados directamente desde fuera de la clase. Esto protege los datos y asegura que solo se modifiquen a través de métodos controlados.
5. **Polimorfismo**: El polimorfismo permite que un mismo nombre de método se utilice para diferentes tipos de objetos, pero con un comportamiento específico para cada tipo. Esto hace que los programas sean más flexibles y fáciles de extender.

Manejo de Archivos en Python

El manejo de archivos en Python permite trabajar con datos persistentes almacenados en disco, como archivos de texto. Existen dos formas principales de hacerlo: usando la función integrada open() o utilizando la clase Path del módulo pathlib, que ofrece una interfaz más moderna y orientada a objetos. Para abrir un archivo y leer su contenido, se puede usar el modo 'r' (read). Por ejemplo:

```
Archivo = open("datos.txt", "r")

Contenido = Archivo.read()

print(Contenido)

Archivo.close()
```

Es importante cerrar el archivo después de usarlo para liberar recursos del sistema. Una forma más segura es usar with, que se encarga de cerrar el archivo automáticamente:

```
with open("datos.txt", "r") as Archivo:

    Contenido = Archivo.read()

    print(Contenido)
```

Para escribir en un archivo se usa el modo 'w', que sobreescribe el contenido si el archivo ya existe:

```
with open("resultado.txt", "w") as Archivo:

    Archivo.write("Este Es El Nuevo Contenido\n")
```

Si se desea agregar contenido sin borrar lo anterior, se puede usar el modo 'a' (append):

```
with open("resultado.txt", "a") as Archivo:

    Archivo.write("Línea Adicional\n")
```

Otra forma moderna de trabajar con archivos es mediante pathlib. Por ejemplo, leer un archivo completo con read_text():

```
from pathlib import Path

Ruta = Path("datos.txt")

Contenido = Ruta.read_text()

print(Contenido)
```

Y para escribir texto:

```
Ruta.write_text("Este Texto Será Guardado En El Archivo")
```

Se puede dividir el contenido en líneas con splitlines():

```
Lineas = Contenido.splitlines()

for Linea in Lineas:

    print(Linea.title())
```

También es fundamental manejar posibles errores con bloques try/except, por ejemplo, cuando el archivo no existe:

```
try:

    with open("inexistente.txt", "r") as Archivo:

        Datos = Archivo.read()

except FileNotFoundError:

    print("El Archivo No Existe")
```

El manejo de archivos es esencial en muchos programas, ya sea para guardar información del usuario, procesar grandes cantidades de datos, crear registros de actividad o trabajar con configuraciones. Python ofrece herramientas potentes y simples para estas tareas, facilitando la lectura, escritura y procesamiento de archivos de texto y otros formatos.

Uso de Pytest

La biblioteca pytest en Python permite automatizar pruebas para verificar que funciones y módulos se comporten como se espera. No viene incluida por defecto, por lo que primero se debe instalar mediante pip:

```
pip install pytest
```

Esto brinda acceso a una herramienta muy potente para escribir pruebas unitarias de manera simple, y escalar a proyectos más complejos a medida que el desarrollo crece.

Supongamos que queremos probar la siguiente función, que devuelve un nombre completo formateado:

```
def ObtenerNombreFormateado(Nombre, Apellido):

    """Generar un nombre completo con formato prolijo."""

    NombreCompleto = f"{Nombre} {Apellido}"

    return NombreCompleto.title()
```

Podemos usar pytest para automatizar una prueba que verifique que esta función devuelve el resultado correcto. Para ello, escribimos un archivo de prueba que comience con test_, por ejemplo: test_nombre.py.

```
from nombre_funcion import ObtenerNombreFormateado

def test_nombre_apellido():
    """¿Funciona con nombres simples como 'pepe sota'?"""
    Resultado = ObtenerNombreFormateado('pepe', 'sota')
    assert Resultado == 'Pepe Sota'
```

Es importante que la función de prueba comience con test_, ya que pytest detecta automáticamente todos los archivos y funciones que comienzan con esa palabra para ejecutarlas.

Luego, para correr las pruebas, desde la terminal navegamos a la carpeta del archivo de prueba y ejecutamos:

```
pytest
```

La salida esperada se verá así:

```
========================= test session starts ==========================
collected 1 item

test_nombre.py .                                                  [100%]

========================== 1 passed in 0.01s ===========================
```

Esto indica que la función pasó la prueba correctamente. Si más adelante modificamos la función y algo falla, pytest nos alertará de inmediato.

Supongamos ahora que queremos que la función también acepte un segundo nombre. Podemos modificarla así:

```
def ObtenerNombreFormateado(Nombre, Apellido, SegundoNombre=''):
    """Generar un nombre completo con formato prolijo."""
    if SegundoNombre:
        NombreCompleto = f"{Nombre} {SegundoNombre} {Apellido}"
    else:
        NombreCompleto = f"{Nombre} {Apellido}"
    return NombreCompleto.title()
```

Ahora podemos agregar una segunda prueba para cubrir este nuevo caso:

```
def test_nombre_con_segundo_nombre():
    """¿Funciona con nombres como 'Wolfgang Amadeus Mozart'?"""
    Resultado = ObtenerNombreFormateado('wolfgang', 'mozart',
'amadeus')
    assert Resultado == 'Wolfgang Amadeus Mozart'
```

Al correr pytest de nuevo, deberíamos ver que ambas pruebas pasan:

```
========================= test session starts ==========================

collected 2 items


test_nombre.py ..                                                  [100%]


========================== 2 passed in 0.01s ===========================
```

Esto nos da la seguridad de que la función funciona tanto con como sin segundo nombre. Además, si en el futuro se introduce un error, las pruebas fallarán y sabremos exactamente qué tipo de entrada dejó de funcionar.

Las pruebas unitarias permiten validar una parte específica del comportamiento de una función. Un conjunto de estas pruebas se conoce como caso de prueba. Cuando abarca todos los posibles escenarios de uso, se considera que tiene cobertura total. Aunque lograrla puede ser complejo, es recomendable al menos cubrir los aspectos más críticos.