Curso de Python

Python es un lenguaje de programación de **propósito general** que se destaca por su legibilidad y facilidad de aprendizaje. Algunas de sus características distintivas son su **tipado dinámico, orientación a objetos y su naturaleza interpretada**. Estas características hacen que Python sea una elección popular para una amplia variedad de aplicaciones.

```
#Ejemplo de tipado dinámico
nombre = "nombre"
edad = 20
```

Ventajas

- **Proposito general**: Python es versátil y se utiliza en una amplia gama de campos, desde desarrollo web hasta ciencia de datos y machine learning.
- **Alto nivel**: Al ser de alto nivel, Python permite a los programadores expresar ideas en menos líneas de código, lo que facilita la lectura y el mantenimiento.
- **Fácil de aprender**: La sintaxis simple y clara de Python facilita la entrada a la programación para principiantes, pero también es poderosa y flexible para desarrolladores más experimentados.
- **Tipado dinámico**: No es necesario declarar el tipo de variable al principio; Python infiere automáticamente el tipo de datos, lo que hace que el código sea más flexible y fácil de escribir.
- **Orientado a Objetos**: Python admite programación orientada a objetos, facilitando la organización y estructuración del código.
- **Lenguaje interpretado**: El código Python se ejecuta línea por línea, facilitando la detección de errores y permitiendo una rápida prueba y depuración.
- **Comunidad grande**: Python cuenta con una comunidad activa y solidaria que proporciona recursos, bibliotecas y soporte, lo que facilita la resolución de problemas y el aprendizaje continuo.

Desventajas

- **Rendimiento Relativo**: Comparado con lenguajes de bajo nivel como C++ o Rust, Python puede ser más lento en términos de ejecución, especialmente en aplicaciones que requieren un alto rendimiento computacional.
- **Gestión de Memoria**: Python utiliza un recolector de basura para gestionar la memoria automáticamente, lo que puede resultar en cierta pérdida de control sobre la gestión de la memoria en comparación con lenguajes de bajo nivel.
- **Menos Adecuado para Aplicaciones Móviles Nativas**: Aunque existen soluciones para el desarrollo móvil con Python, no es tan común o eficiente como otros lenguajes diseñados específicamente para este propósito.
- **Interpretación vs Compilación**: Python es un lenguaje interpretado, lo que puede afectar su rendimiento en comparación con lenguajes compilados, especialmente en aplicaciones que requieren alta velocidad de ejecución.
- **Tamaño del Ecosistema para Desarrollo de Juegos**: Aunque Python se utiliza en el desarrollo de juegos, su presencia no es tan fuerte como en otros lenguajes como C++ en este ámbito,

especialmente en juegos de alta gama.
- **Compatibilidad con Versiones**: La transición entre versiones principales de Python (por ejemplo, de Python 2 a Python 3) puede generar problemas de compatibilidad en el código existente.
- **Recursos Limitados para Desarrollo Empotrado**: En comparación con lenguajes como C o C++, Python puede no ser la elección más adecuada para el desarrollo de sistemas embebidos debido a su consumo de recursos.
- **Gestión de Hilos y Concurrencia**: Aunque Python admite hilos, su capacidad para manejar concurrencia puede no ser tan eficiente como en algunos lenguajes diseñados específicamente para este propósito.
- **Falta de Soporte para Desarrollo de Aplicaciones de Tiempo Real**: En comparación con algunos lenguajes diseñados para aplicaciones de tiempo real, Python puede no ser la opción más óptima en términos de predictibilidad y gestión de tiempos estrictos.

Utilidad

- **Desarrollo web**: Python es ampliamente utilizado para desarrollar aplicaciones web, con frameworks como Flask y Django que simplifican el proceso.
- **Aplicaciones moviles**: A través de frameworks como Kivy o BeeWare, Python se puede utilizar para el desarrollo de aplicaciones móviles.
- **Aplicaciones escritorio**: Con herramientas como Tkinter o PyQt, Python se utiliza para crear aplicaciones de escritorio con interfaces gráficas de usuario (GUI).
- **Automatización de scripts**: Python es ideal para escribir scripts que automatizan tareas repetitivas y simplifican procesos.
- **Ciencia de datos y Machine Learning**: Con bibliotecas como NumPy, pandas y scikit-learn, Python es esencial en el ámbito de la ciencia de datos y machine learning.
- **Desarrollo de software**: Python es utilizado para desarrollar una amplia variedad de software, desde pequeñas herramientas hasta sistemas complejos.
- **Automatización de pruebas de software**: Frameworks como Selenium y Pytest permiten la automatización eficiente de pruebas de software.
- **Desarrollo de juegos**: Aunque no es tan común como en otros lenguajes, Python se utiliza en el desarrollo de juegos, especialmente para prototipos y juegos 2D.
- **Hacking**: Python es una opción popular en el campo de la seguridad informática y el hacking ético debido a su sintaxis clara y a las herramientas disponibles.

1 Conceptos básicos

Tipos de datos

```
# Cadena de texto (String) - Se puede definir usando comillas simples, dobles o
triples
string_single = 'string con comillas simples'
string_double = "string con comillas dobles"
string_triple_double = """string
                        en múltiples líneas
                        con comillas dobles"""
string_triple_single = '''string
```

```
                        en múltiples líneas
                        con comillas simples'''

# Número entero (Integer)
integer = 1

# Número decimal (Float)
floating_point = 1.0

# Booleano (Boolean)
boolean_true = True
boolean_false = False
```

Definir variables

Las variables en Python son **etiquetas** que asignamos a valores, ya sean de tipo string, int, float o bool. Sirven para almacenar y referenciar información dentro de un programa. Aquí tienes un ejemplo:

```
nombre = "Jose"
edad = 28
sexo = "Masculino"
```

Ahora, al hablar sobre las dos maneras de definir variables en Python, podemos utilizar:

Camel Case: En Camel Case, la primera letra de cada palabra, excepto la primera, se escribe en mayúscula. Aunque es aceptable en Python, no es la convención más comúnmente usada.

```
nombreCompleto = "Jose Fuentes"
```

Snake Case (Recomendado): En Snake Case, las palabras se escriben en minúsculas y se separan por guiones bajos _. Este estilo es ampliamente recomendado por la PEP 8, la guía de estilo para código en Python. Es más legible y preferido en la comunidad Python.

```
nombre_completo = "Jose Fuentes"
```

La elección entre Camel Case y Snake Case no solo se trata de preferencias estilísticas, sino también de seguir las convenciones de la comunidad para facilitar la colaboración y la lectura del código. En Python, la recomendación general es utilizar Snake Case para nombrar variables.

Eliminar variable

Si queremos limpiar de memoria la variable despues de utlizarla podemos utilizar el comando **del**.

```
del variable
```

Asignar nuevo valor

```
variable = None  # Asignar un nuevo valor (puede ser None u otro valor)
```

Si queremos asignar un nuevo valor lo podemos realizar mediante un variable = None cantidad = None # Asignar un nuevo valor (puede ser None u otro valor)

Imprimir variables

```
# Definición de variables
nombre = "José"
apellido = " Fuentes"
edad = 28

# Concatenación con el operador '+'
nombre_completo = nombre + apellido

# Imprimir el nombre, apellido y edad concatenados
print("1. Concatenación con '+' (sin espacio):", nombre + apellido + " " +
str(edad))

# Concatenación con f-string
mensaje_fstring = f"Hola {nombre_completo}, tu edad es {edad}"
print("2. Concatenación con f-string:", mensaje_fstring)

# Concatenación con el operador '+' y espacio adicional
mensaje_con_espacio = nombre + " " + apellido + " " + str(edad)
print("3. Concatenación con '+' (con espacio):", mensaje_con_espacio)
```

2 Variables compuestas

Listas

Una lista es una estructura de datos que se utiliza para almacenar una colección ordenada de elementos. Estos elementos pueden ser de cualquier tipo, como números, cadenas de texto, booleanos, otras listas, entre otros. **La lista es un tipo de dato mutable**, lo que significa que se pueden modificar después de su creación.

Características clave de las listas en Python:

- **Ordenadas:** Los elementos de la lista están ordenados y se accede a ellos mediante un índice. El primer elemento tiene índice 0, el segundo índice 1, y así sucesivamente.
- **Mutables:** Puedes cambiar, agregar o eliminar elementos de una lista después de su creación.

- **Pueden contener diferentes tipos de datos:** Una lista puede contener elementos de diferentes tipos, como enteros, cadenas de texto, booleanos, e incluso otras listas u objetos más complejos.
- **Soportan operaciones y métodos:** Las listas en Python ofrecen una variedad de operaciones y métodos que facilitan la manipulación de los datos almacenados en ellas, como agregar elementos, eliminar elementos, obtener la longitud, invertir el orden, entre otros.

```
# Maneras de como definir una lista
lista1 = [1, "dos", 3.0, True]

#Usando la Función list():
lista2 = list([1, 2, 3, 4, 5])

#Lista Vacía:
lista_vacia = []

#Usando un for
numeros = [x for x in range(1, 6)]

#Usando el Método append():
lista = []
lista.append(1)
lista.append(2)
lista.append(3)

#Usando el Método extend():
lista = [1, 2, 3]
lista.extend([4, 5, 6])

#Usando la Función list() con Otra Secuencia (por ejemplo, una Tupla):
tupla = (1, 2, 3)
lista_desde_tupla = list(tupla)

#Usando el Método split() con una Cadena de Texto:
cadena = "a b c"
lista_desde_cadena = cadena.split()

#Usando Repetición:
lista_repetida = [0] * 5  # Crea una lista con cinco elementos 0


# Imprimiendo los datos
lista_datos = ["Jose", "Alberto", 28]

# Desempaquetado de datos
nombre, apellido, edad = lista_datos

# Impresión de datos de la lista
print(f"Lista de Datos: Nombre: {nombre} {apellido} y Edad: {edad}")

# Acceso a elementos de la lista con bucle for
print("Acceso a elementos con bucle for:")
```

```
for dato in lista_datos:
    print(dato)
```

Tuplas

Una tupla es una estructura de datos que permite almacenar un conjunto ordenado e inmutable de elementos. Las tuplas son similares a las listas, pero tienen algunas diferencias clave.

- **Lectura:** Las Tuplas a diferencia de las listas manejan mejor la memoria ya que las Listas guardan los datos en memoria para modificar la Lista original.
- **Inmutabilidad:** Una tupla no puede ser modificada después de su creación. Esto significa que no puedes agregar, eliminar o cambiar elementos en una tupla una vez que ha sido creada. En contraste, las listas son mutables.
- **Sintaxis:** Las tuplas se definen utilizando paréntesis () o, en el caso de las tuplas vacías, simplemente con paréntesis vacíos. Por ejemplo, (1, 2, 3) o ().
- **Tamaño fijo:** El tamaño de una tupla es fijo después de su creación. No puedes cambiar la longitud de una tupla añadiendo o eliminando elementos.
- **Indexación:** Al igual que las listas, las tuplas son indexadas, lo que significa que puedes acceder a sus elementos utilizando índices.
- **Heterogeneidad:** Una tupla puede contener elementos de diferentes tipos (por ejemplo, números, cadenas, booleanos, otras tuplas, etc.).

```
#Usando paréntesis:
tupla1 = ('cadena', 14, 3.14, True)

#Sin paréntesis (Empaquetado de Tupla):
tupla2 = 4, 5, 6

#Con un solo elemento (debe tener una coma al final):
tupla3 = (7,)

#Construcción de tuplas mediante la función tuple():
lista = [8, 9, 10]
tupla4 = tuple(lista)

#Usando la función zip() para combinar iterables:
nombres = ('Alice', 'Bob', 'Charlie')
edades = (25, 30, 22)
tupla5 = tuple(zip(nombres, edades))

#Desempaquetado de Tupla:
tupla6 = 11, 12, 13
a, b, c = tupla6  # Desempaquetado

#Creación de una tupla vacía:
tupla_vacia = ()
```

```
#Creación de una tupla con el constructor tuple():
tupla_construida = tuple()

#Usando comprensiones de tuplas:
tupla_comprension = tuple(x for x in range(5))


#Imprimir tuplas
mi_tupla = (1, 'dos', 3.0, True)

print("Tupla completa:", mi_tupla)

print("Primer elemento:", mi_tupla[0])
print("Segundo elemento:", mi_tupla[1])
print("Tercer elemento:", mi_tupla[2])
print("Cuarto elemento:", mi_tupla[3])

# Iterar sobre los elementos de la tupla e imprimirlos
print("Iterar e imprimir:")
for elemento in mi_tupla:
    print(elemento)
```

Conjuntos

Los conjuntos son una colección desordenada y sin duplicados de elementos. Se definen utilizando llaves {} o la función set(). Los conjuntos en Python son similares a los conjuntos en matemáticas y proporcionan operaciones comunes de conjuntos como unión, intersección y diferencia.

- **Elementos Únicos:** Los conjuntos no pueden contener elementos duplicados. Si intentas agregar un elemento que ya está presente, el conjunto no cambiará.
- **Desordenados:** Los elementos en un conjunto no tienen un orden específico. No puedes confiar en el orden en que se almacenan los elementos en un conjunto.
- **Mutables:** Los conjuntos son mutables, lo que significa que puedes agregar y eliminar elementos después de haber creado el conjunto.

```
conjunto_vacio = set()

# Definición de un conjunto con elementos
conjunto1 = {1, 2, 3, 4, 5}

# Convertir una lista a un conjunto usando set()
lista = [3, 4, 5, 6, 7]
conjunto2 = set(lista)

# Acceder a elementos de un conjunto (los conjuntos no tienen índices)
# Por lo tanto, se utiliza un bucle para acceder a los elementos
print("Acceder a elementos:")
for elemento in conjunto1:
```

```
    print(elemento)

# Verificar si un elemento está en el conjunto
elemento_a_verificar = 3
print(f"¿El elemento {elemento_a_verificar} está en conjunto1?:",
elemento_a_verificar in conjunto1)

# Operaciones con conjuntos (unión, intersección, diferencia)
conjunto3 = {4, 5, 6, 7, 8}
union = conjunto1.union(conjunto3)
interseccion = conjunto1.intersection(conjunto3)
diferencia = conjunto1.difference(conjunto3)

# Imprimir conjuntos y resultados de operaciones
print("Conjunto3:", conjunto3)
print("Unión de conjunto1 y conjunto3:", union)
print("Intersección de conjunto1 y conjunto3:", interseccion)
print("Diferencia entre conjunto1 y conjunto3:", diferencia)
```

Diccionarios

Un diccionario es una estructura de datos que permite almacenar y organizar información de manera eficiente. A diferencia de las listas o tuplas, los diccionarios no tienen un orden específico, y los datos se acceden mediante claves en lugar de índices. Cada elemento en un diccionario consiste en una clave y su correspondiente valor asociado.

- **Claves Únicas:** Cada clave en un diccionario es única, lo que significa que no puede haber duplicados. Sin embargo, los valores asociados pueden repetirse.
- **Mutable:** Los diccionarios son objetos mutables, lo que significa que puedes modificar, agregar o eliminar elementos después de haber sido creados.
- **Dinámico:** Puedes modificar la estructura del diccionario durante su vida útil, agregando nuevas claves o eliminando claves existentes.
- **Heterogéneo:** Los valores en un diccionario pueden ser de cualquier tipo de dato, y cada valor está asociado a una clave.
- **Sin Orden Definido:** A diferencia de las listas, los diccionarios no tienen un orden predefinido. No puedes asumir que los elementos estarán en un orden específico cuando los iteras.

```
diccionario1 = {
    'nombre': 'Alice',
    'edad': 25,
    'ciudad': 'Ejemplo'
}
print("Diccionario1:", diccionario1)

# Acceder a valores mediante llaves
nombre_dicc = diccionario1['nombre']
edad_dicc = diccionario1['edad']
```

```
# Creación de un diccionario desde listas de claves y valores usando zip()
claves = ['nombre', 'edad', 'ciudad']
valores = ['Bob', 30, 'Otro Ejemplo']
diccionario2 = dict(zip(claves, valores))
print("Diccionario2:", diccionario2)

# Creación de un diccionario mediante comprensión de diccionario
numeros = [1, 2, 3, 4, 5]
diccionario_cuadrados = {x: x**2 for x in numeros}
print("Diccionario de Cuadrados:", diccionario_cuadrados)

# Iterar sobre claves y valores de un diccionario
print("Iterar sobre claves y valores:")
for clave, valor in diccionario1.items():
    print(f"{clave}: {valor}")

# Creación de un diccionario con un solo elemento
diccionario_un_elemento = {'clave_unica': 'valor único'}
print("Diccionario con un solo elemento:", diccionario_un_elemento)

# Creación de un diccionario con la función dict() y una lista de tuplas clave-
valor
lista_clave_valor = [('nombre', 'David'), ('edad', 28), ('ciudad', 'Ejemplo')]
diccionario_desde_lista = dict(lista_clave_valor)
print("Diccionario desde lista de tuplas:", diccionario_desde_lista)

# Creación de un diccionario con valores predeterminados usando fromkeys()
claves = ['nombre', 'edad', 'ciudad']
diccionario_con_valores_predeterminados = dict.fromkeys(claves, 'Valor
Predeterminado')
print("Diccionario con valores predeterminados:",
diccionario_con_valores_predeterminados)
```

3 Operadores

Operadores aritmétricos

Los operadores aritméticos son utilizados para realizar operaciones matemáticas sobre números. Aquí están los operadores aritméticos básicos:

- **Suma (+):** Suma dos valores.

```
resultado_suma = 5 + 3  # resultado_suma será 8
```

- **Resta (-):** Resta el segundo valor del primero.

```
resultado_resta = 7 - 4  # resultado_resta será 3
```

- **Multiplicación** (*): Multiplica dos valores.

```
resultado_multiplicacion = 2 * 6  # resultado_multiplicacion será 12
```

- **División (/):** Divide el primer valor por el segundo. El resultado siempre es un número de punto flotante, incluso si la división es exacta.

```
resultado_division = 8 / 4  # resultado_division será 2.0
```

- **División Entera (//):** Divide el primer valor por el segundo y redondea hacia abajo al entero más cercano.

```
resultado_division_entera = 9 // 4  # resultado_division_entera será 2
```

- **Módulo (%):** Devuelve el resto de la división entre el primer valor y el segundo.

```
resultado_modulo = 9 % 4  # resultado_modulo será 1
```

- **Exponente ():** Eleva el primer valor a la potencia del segundo.

```
resultado_exponente = 2 ** 3  # resultado_exponente será 8
```

Operadores de comparación

Los operadores de comparación en Python permiten comparar dos valores y devuelven un resultado booleano (True o False) que indica si la comparación es verdadera o falsa. Aquí están los operadores de comparación:

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

```
resultado_igualdad = 5 == 5  # resultado_igualdad será True
```

- **Desigualdad (!=):** Comprueba si dos valores no son iguales.

```
resultado_desigualdad = 5 != 3  # resultado_desigualdad será True
```

- **Mayor que (>):** Comprueba si el primer valor es mayor que el segundo.

```
resultado_mayor_que = 7 > 4  # resultado_mayor_que será True
```

- **Menor que (<):** Comprueba si el primer valor es menor que el segundo.

```
resultado_menor_que = 3 < 6  # resultado_menor_que será True
```

-**Mayor o igual que (>=):** Comprueba si el primer valor es mayor o igual que el segundo.

```
resultado_mayor_igual_que = 5 >= 5  # resultado_mayor_igual_que será True
```

- **Menor o igual que (<=):** Comprueba si el primer valor es menor o igual que el segundo.

```
resultado_menor_igual_que = 4 <= 7  # resultado_menor_igual_que será Tru
```

Operadores lógicos

Los operadores lógicos en Python se utilizan para combinar expresiones booleanas y realizar operaciones lógicas:

- **AND (and):** Devuelve True si ambas expresiones booleanas son verdaderas, de lo contrario, devuelve False.
- **OR (or):** Devuelve True si al menos una de las expresiones booleanas es verdadera, de lo contrario, devuelve False.
- **NOT (not):** Devuelve True si la expresión booleana es falsa, y viceversa.

```
# AND
resultado1 = True & True  # True
resultado2 = False & True  # False
resultado3 = True & False  # False
resultado4 = False & False  # False

# OR
resultado5 = True | True  # True
resultado6 = False | True  # True
resultado7 = True | False  # True
resultado8 = False | False  # False

#NOT
resultado9 = not True  # False
resultado10 = not False  # True
```

Condicionales

Las estructuras condicionales en Python, principalmente representadas por las instrucciones if, elif (else if), y else, permiten que un programa tome decisiones basadas en condiciones lógicas. Estas condiciones se expresan mediante expresiones booleanas que evalúan a True o False.

- **if** La instrucción if se utiliza para ejecutar un bloque de código si una condición es verdadera. Si la condición es falsa, el bloque de código no se ejecuta.
- **elif (else if)** La instrucción elif se utiliza para evaluar múltiples condiciones después de un if. Se ejecuta solo si la condición del if es falsa y la condición elif es verdadera.
- **else** La instrucción else se utiliza para ejecutar un bloque de código cuando ninguna de las condiciones anteriores (if o elif) es verdadera.

```
ingreso_mensual = 1000001

if ingreso_mensual > 100000:
    print("Ingreso alto")
elif ingreso_mensual > 50000 and ingreso_mensual <= 100000:
    print("Ingreso medio")
else:
    print("ingreso bajo")
```

Las estructuras condicionales pueden anidarse, permitiendo condiciones más complejas y toma de decisiones más sofisticadas en un programa. Estos son elementos clave para controlar el flujo de ejecución en Python y en la programación en general.

4 Métodos de cadenas

Métodos de cadenas

Los métodos de cadenas son funciones incorporadas en Python que se aplican específicamente a objetos de tipo cadena (strings). Estos métodos permiten realizar diversas operaciones y manipulaciones en las cadenas de texto. Aquí hay algunos de los métodos de cadenas más comunes en Python:

La función **dir(variable)** devuelve la lista de atributos válidos del objeto.

```
cadena1 = "Hola soy python"
print(dir(cadena1))
```

El resultado de **print(dir(cadena1))** proporciona una lista de métodos y atributos disponibles para el objetos.

```
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
'__getnewargs__', '__getstate__', '__gt__', '__hash__', '__init__',
```

```
'__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mod__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'encode',
'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum',
'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower',
'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust',
'lower', 'lstrip', 'maketrans', 'partition', 'removeprefix', 'removesuffix',
'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split',
'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper',
'zfill']
```

Otros métodos:

- **variable.upper():** Convierte a mayusculas
- **variable.lower():** Convierte en minisculas
- **variable.capitalize():** Convierte solo la primera en mayuscula

```
cadena1 = "bienvenido"

print(cadena1.upper()) # bienvenido
print(cadena1.lower()) # BIENVENIDO
print(cadena1.capitalize()) # Bienvenido
```

- **variable.find("")** Devuelve la posición iniciar donde encuentra el valor y -1 cuando no encuentra el valor
- **variable.index("")** Devuelve la posición iniciar donde encuentra el valor y ERROR cuando no encuentra el valor

```
print("Hola a todos".find("a")) # 3
print("Hola a todos".index("a")) # 3
```

- **variable.isnumeric()** Devuelve True o False si es numérico
- **variable.isalpha()** Devuelve True o False si es alfanumérico o solo letras (Regresa False si tiene espacios)

```
print(cadena1.isnumeric()) # False
print(cadena1.isalpha()) # True
```

- **variable.count("")** Devuelve las coincidencias dentro del texto len(variable) Devuelve el tamaño de una cadena

```
print("Hola a todos".count("a")) # 2
print(len("Hola")) # 4
```

- **variable.startswith("")** Verifica si una cadena empieza con ""
- **variable.endswith("")** Verifica si una cadena termina con ""

```
print("Hola".startswith("H")) # True
print("Hola".endswith("a")) # True
```

- **variable.replace(buscar,remplazar)** Remplaza los valores que encuentra
- **variable.split()** Separa todo por espacio
- **variable.split(",")** Separa todo por coma

```
print("Todo va bien".replace("bien","mal")) # Hola todo va mal
print("hola, hola".split()) # ['hola,', 'hola']
print("hola, hola".split(", ")) # ['hola', 'hola']
```

Métodos de listas

Los métodos de las listas son funciones incorporadas que se pueden aplicar directamente a objetos de tipo lista. Estos métodos permiten realizar diversas operaciones y manipulaciones en las listas. Algunos de los métodos más comunes de las listas son:

- **lista([var,...,var])** Crea una lista
- **len(lista)** Cuenta la cantidad de elementos
- **lista.append(valor)** Agrega un elemento
- **lista.insert(ind,valor)** Agrega un elemento en un indice especificado
- **lista.extend([valor1,..., valorN])** Agrega varios elementos
- **lista.pop(indice)** Eliminando un elemento por indice
- **lista.remove("")** Eliminando un elmento por su valor
- **lista.sort()** Ordena números y boleanos de manera ascendente
- **lista.sort(reverse = True)** Ordena números y boleanos de manera descendente
- **lista.reverse()** Invierte el orden de la lista [1,2,3,4] = [4,3,2,1]
- **lista.clear()** Elimina todos los elementos

```
lista = list(["Hola", 1 ,True])
print(lista)
print(len(lista))

lista.append("bien")
print(lista)

lista.insert(3, "mal")
print(lista)

lista.extend([True, 5])
print(lista)
```

```
lista.pop(0)
print(lista)

lista.remove("mal")
print(lista)

lista = list([95,25,33,74,True, False])
lista.sort()
print(lista)
lista.sort(reverse= True)
print(lista)

lista = list([95,25,33,74,True, False])
lista.reverse()
print(lista)

lista.clear()
print(lista)
```

```
['Hola', 1, True]
3
['Hola', 1, True, 'bien']
['Hola', 1, True, 'mal', 'bien']
['Hola', 1, True, 'mal', 'bien', True, 5]
[1, True, 'mal', 'bien', True, 5]
[1, True, 'bien', True, 5]
[False, True, 25, 33, 74, 95]
[95, 74, 33, 25, True, False]
[False, True, 74, 33, 25, 95]
[]
```

Metodos de Tuplas

Las tuplas son estructuras de datos inmutables, lo que significa que no se pueden modificar una vez creadas. Aunque las tuplas no tienen tantos métodos como las listas, tienen algunos que son útiles. Aquí hay algunos métodos comunes de tuplas en Python:

- **tupla.count(valor)** Busca los elementos con el mismo valor y cuenta el número de repeticiones
- **tupla.index(valor)** Regresa la posición del elemento buscado

```
#Crear tuplas

tupla = tuple(["Jose","Alberto",28])
tupla = "Jose","Alberto",28
tupla = "Jose",
tupla = tuple(zip("Jose","Alberto"))
tupla = (1,2,3,4, "hola",1,1,1)
```

```
print(dir(tupla))
print(tupla.count(1))
print(tupla.index("hola"))
```

```
['__add__', '__class__', '__class_getitem__', '__contains__', '__delattr__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__getitem__', '__getnewargs__', '__getstate__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mul__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmul__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'count', 'index']
4
4
```

Metodos de Conjuntos

Los métodos de conjuntos son funciones integradas que se aplican a objetos de tipo conjunto (set). Un conjunto es una colección desordenada de elementos únicos, lo que significa que no puede contener elementos duplicados. Los métodos de conjuntos proporcionan funcionalidades específicas para realizar operaciones comunes en conjuntos. Aquí hay algunos de los métodos de conjuntos más utilizados:

- **add:** Agregar un elemento al conjunto.
- **clear:** Eliminar todos los elementos del conjunto.
- **copy:** Crear una copia del conjunto.
- **difference:** Obtener la diferencia entre dos conjuntos.
- **difference_update:** Actualizar el conjunto restando elementos de otro conjunto.
- **discard:** Eliminar un elemento específico del conjunto si está presente.
- **intersection:** Obtener la intersección entre dos conjuntos.
- **intersection_update:** Actualizar el conjunto manteniendo solo los elementos comunes.
- **isdisjoint:** Verificar si dos conjuntos son disjuntos (no tienen elementos en común).
- **issubset:** Verificar si un conjunto es subconjunto de otro.
- **issuperset:** Verificar si un conjunto es superconjunto de otro.
- **pop:** Eliminar y devolver un elemento arbitrario del conjunto.
- **remove:** Eliminar un elemento específico del conjunto.
- **symmetric_difference:** Obtener la diferencia simétrica entre dos conjuntos.
- **symmetric_difference_update:** Actualizar el conjunto con su diferencia simétrica.
- **union:** Obtener la unión de dos conjuntos.
- **update:** Agregar elementos de otro conjunto al conjunto actual.

```
conj_set = {1, 2, 3, 4, "hola"}

# add: Agregar un elemento al conjunto.
conj_set.add(5)
print("Conjunto después de agregar 5:", conj_set)

# copy: Crear una copia del conjunto.
```

```
conjunto_copia = conj_set.copy()
print("Copia del conjunto:", conjunto_copia)

# difference: Obtener la diferencia entre dos conjuntos.
otro_conjunto = {3, 4, 5, 6}
diferencia = conj_set.difference(otro_conjunto)
print("Diferencia entre conjuntos:", diferencia)

# difference_update: Actualizar el conjunto restando elementos de otro conjunto.
conj_set.difference_update(otro_conjunto)
print("Conjunto después de difference_update:", conj_set)

# discard: Eliminar un elemento específico del conjunto si está presente.
conj_set.discard(3)
print("Conjunto después de descartar 3:", conj_set)

# intersection: Obtener la intersección entre dos conjuntos.
conjunto_interseccion = conj_set.intersection(otro_conjunto)
print("Intersección entre conjuntos:", conjunto_interseccion)

# intersection_update: Actualizar el conjunto manteniendo solo los elementos
comunes.
conj_set.intersection_update(otro_conjunto)
print("Conjunto después de intersection_update:", conj_set)

# isdisjoint: Verificar si dos conjuntos son disjuntos (no tienen elementos en
común).
son_disjuntos = conj_set.isdisjoint(otro_conjunto)
print("¿Los conjuntos son disjuntos?:", son_disjuntos)

# issubset: Verificar si un conjunto es subconjunto de otro.
es_subconjunto = conj_set.issubset(otro_conjunto)
print("¿El conjunto es subconjunto?:", es_subconjunto)

# issuperset: Verificar si un conjunto es superconjunto de otro.
es_superconjunto = conj_set.issuperset(otro_conjunto)
print("¿El conjunto es superconjunto?:", es_superconjunto)

conj_set = {3, 4, 5, 6, 1,2,3,4,5,6,7}
# pop: Eliminar y devolver un elemento arbitrario del conjunto.
elemento_eliminado = conj_set.pop()
print("Elemento eliminado con pop:", elemento_eliminado)

# remove: Eliminar un elemento específico del conjunto.
conj_set.remove(3)
print("Conjunto después de remover 3:", conj_set)

# symmetric_difference: Obtener la diferencia simétrica entre dos conjuntos.
diferencia_simetrica = conj_set.symmetric_difference(otro_conjunto)
print("Diferencia simétrica entre conjuntos:", diferencia_simetrica)

# symmetric_difference_update: Actualizar el conjunto con su diferencia simétrica.
conj_set.symmetric_difference_update(otro_conjunto)
print("Conjunto después de symmetric_difference_update:", conj_set)
```

```python
# union: Obtener la unión de dos conjuntos.
union_conjuntos = conj_set.union(otro_conjunto)
print("Unión de conjuntos:", union_conjuntos)

# update: Agregar elementos de otro conjunto al conjunto actual.
conj_set.update(otro_conjunto)
print("Conjunto después de update:", conj_set)

# clear: Eliminar todos los elementos del conjunto.
conj_set.clear()
print("Conjunto después de clear:", conj_set)
```

```
Conjunto después de agregar 5: {1, 2, 3, 4, 5, 'hola'}
Copia del conjunto: {1, 2, 3, 4, 5, 'hola'}
Diferencia entre conjuntos: {1, 2, 'hola'}
Conjunto después de difference_update: {1, 2, 'hola'}
Conjunto después de descartar 3: {1, 2, 'hola'}
Intersección entre conjuntos: set()
Conjunto después de intersection_update: set()
¿Los conjuntos son disjuntos?: True
¿El conjunto es subconjunto?: True
¿El conjunto es superconjunto?: False
Elemento eliminado con pop: 1
Conjunto después de remover 3: {2, 4, 5, 6, 7}
Diferencia simétrica entre conjuntos: {2, 3, 7}
Conjunto después de symmetric_difference_update: {2, 3, 7}
Unión de conjuntos: {2, 3, 4, 5, 6, 7}
Conjunto después de update: {2, 3, 4, 5, 6, 7}
Conjunto después de clear: set()
```

Métodos de Diccionarios

Los métodos de diccionarios son funciones incorporadas que se aplican a objetos de tipo diccionario. Aquí hay algunos métodos comunes de diccionarios:

- **clear:** Eliminar todos los elementos del diccionario
- **copy:** Crear una copia del diccionario
- **fromkeys:** Crear un nuevo diccionario con claves especificadas y un valor predeterminado
- **get:** Obtener el valor asociado con una clave (manejo seguro para evitar errores si la clave no existe)
- **items:** Obtener una vista de los pares clave-valor en el diccionario
- **keys:** Obtener una vista de las claves en el diccionario
- **pop:** Eliminar y obtener el valor asociado con una clave
- **popitem:** Eliminar y obtener un par clave-valor arbitrario del diccionario
- **setdefault:** Obtener el valor asociado con una clave; si la clave no existe, se agrega al diccionario con un valor predeterminado
- **update:** Actualizar el diccionario con pares clave-valor de otro diccionario
- **values:** Obtener una vista de los valores en el diccionario

```
# Diccionario inicial
diccionario = {
    'nombre': "Jose",
    'apellido': "Fuentes",
    'edad': 28
}

# Imprimir directorio del diccionario definido
print("Directorio del diccionario definido:", dir(diccionario))

# copy: Crear una copia del diccionario
diccionario_copia = diccionario.copy()
print("Copia del diccionario:", diccionario_copia)

# fromkeys: Crear un nuevo diccionario con claves especificadas y un valor
predeterminado
nuevo_diccionario = dict.fromkeys(['nombre', 'apellido', 'edad'], "Valor
Predeterminado")
print("Nuevo diccionario creado con fromkeys:", nuevo_diccionario)

# get: Obtener el valor asociado con una clave (manejo seguro para evitar errores
si la clave no existe)
valor_edad = diccionario.get('edad', "Clave no encontrada")
print("Valor de 'edad':", valor_edad)

# items: Obtener una vista de los pares clave-valor en el diccionario
vista_items = diccionario.items()
print("Vista de items:", vista_items)

# keys: Obtener una vista de las claves en el diccionario
vista_claves = diccionario.keys()
print("Vista de claves:", vista_claves)

# pop: Eliminar y obtener el valor asociado con una clave
valor_apellido = diccionario.pop('apellido')
print("Valor eliminado con pop ('apellido'):", valor_apellido)

# popitem: Eliminar y obtener un par clave-valor arbitrario del diccionario
par_arbitrario = diccionario.popitem()
print("Par arbitrario eliminado con popitem:", par_arbitrario)

# setdefault: Obtener el valor asociado con una clave; si la clave no existe, se
agrega al diccionario con un valor predeterminado
valor_nombre = diccionario.setdefault('nombre', "Valor Predeterminado")
print("Valor de 'nombre' después de setdefault:", valor_nombre)

# update: Actualizar el diccionario con pares clave-valor de otro diccionario
diccionario.update({'ubicacion': 'Ciudad X', 'ocupacion': 'Programador'})
print("Diccionario después de update:", diccionario)

# values: Obtener una vista de los valores en el diccionario
vista_valores = diccionario.values()
```

```
print("Vista de valores:", vista_valores)

# clear: Eliminar todos los elementos del diccionario
diccionario.clear()
print("Diccionario después de clear:", diccionario)
```

```
Directorio del diccionario definido: ['__class__', '__class_getitem__',
'__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__', '__getstate__',
'__gt__', '__hash__', '__init__', '__init_subclass__', '__ior__', '__iter__',
'__le__', '__len__', '__lt__', '__ne__', '__new__', '__or__', '__reduce__',
'__reduce_ex__', '__repr__', '__reversed__', '__ror__', '__setattr__',
'__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'clear', 'copy',
'fromkeys', 'get', 'items', 'keys', 'pop', 'popitem', 'setdefault', 'update',
'values']
Copia del diccionario: {'nombre': 'Jose', 'apellido': 'Fuentes', 'edad': 28}
Nuevo diccionario creado con fromkeys: {'nombre': 'Valor Predeterminado',
'apellido': 'Valor Predeterminado', 'edad': 'Valor Predeterminado'}
Valor de 'edad': 28
Vista de items: dict_items([('nombre', 'Jose'), ('apellido', 'Fuentes'), ('edad',
28)])
Vista de claves: dict_keys(['nombre', 'apellido', 'edad'])
Valor eliminado con pop ('apellido'): Fuentes
Par arbitrario eliminado con popitem: ('edad', 28)
Valor de 'nombre' después de setdefault: Jose
Diccionario después de update: {'nombre': 'Jose', 'ubicacion': 'Ciudad X',
'ocupacion': 'Programador'}
Vista de valores: dict_values(['Jose', 'Ciudad X', 'Programador'])
Diccionario después de clear: {}
```

5 Inputs

En programación, input es una función que se utiliza para recibir datos ingresados por el usuario desde la consola o la interfaz de línea de comandos. En Python, la función input toma una cadena (opcional) como argumento, la imprime como un mensaje al usuario, y luego espera a que el usuario ingrese datos.

Aquí hay un ejemplo simple:

```
nombre_usuario = input("Ingrese su nombre: ")
print(f"Hola, {nombre_usuario}!")
```

En este ejemplo, la función input muestra el mensaje "Ingrese su nombre: " y espera a que el usuario escriba algo y presione Enter. Es importante tener en cuenta que la función input siempre devuelve una cadena (tipo str en Python), incluso si el usuario ingresa un número.

Si se espera un número, es necesario convertir la cadena a un tipo numérico, como int o float. Por ejemplo:

```
edad_str = input("Ingrese su edad: ")
edad_int = int(edad_str)
print(f"El doble de su edad es {2 * edad_int}.")
```

En este caso, la entrada del usuario se convierte a un entero antes de realizar operaciones matemáticas. Es fundamental manejar la conversión de tipos de manera adecuada para evitar errores en tiempo de ejecución.

Validar input (numerico)

```
    try:
        input_usuario = input("Ingrese un número entero: ")
        numero_entero = int(input_usuario)

        # Si la conversión fue exitosa, salir del bucle
        break
    except ValueError:
        # Si ocurre un error al intentar convertir a entero,
        # informar al usuario y volver a solicitar la entrada
        print("Error: Por favor, ingrese un número entero válido.")

print(f"Ha ingresado el número entero: {numero_entero}")
```

6 Bucles

Un bucle o ciclo en programación es una estructura de control que permite ejecutar un conjunto de instrucciones repetidamente mientras se cumpla una condición específica. Estos bucles son fundamentales para la automatización y la repetición de tareas en un programa. Hay varios tipos de bucles, pero los dos más comunes son el bucle for y el bucle while.

Bucle for

Un ciclo for es una estructura de control de flujo en programación que se utiliza para iterar sobre una secuencia de elementos. En Python, el ciclo for es muy versátil y se puede utilizar para recorrer diversos tipos de objetos iterables, como listas, tuplas, cadenas, diccionarios y otros.

La sintaxis básica de un bucle for en Python es la siguiente:

for simple

Iterando sobre una lista

```
frutas = ["manzana", "banana", "cereza"]

for fruta in frutas:
    print(fruta)
```

```
manzana
banana
cereza
```

for multiple

Recorriendo dos o más bucles FOR con mismo número de elementos, si alguna lista tiene más elementos se itera el de menor cantidad de elementos

```python
numeros = [1,2,3]
frutas = ["manzana", "banana", "cereza"]

for fruta,numero in zip(frutas,numeros):
    print(f"La {fruta} tiene el número {numero}")
```

```
La manzana tiene el número 1"
La banana tiene el número 2"
La cereza tiene el número 3"
```

for else

```python
for numero in [1,2,3,4]:
    print(f"numero: {numero}")
else:
    print("Terminó")
```

```
1
2
3
4
Terminó
```

for continue

```python
frutas = ["manzana", "banana", "cereza", "uva", "pera"]
for fruta in frutas:
    if fruta == "uva":
        # Si la fruta es "uva", saltamos a la siguiente iteración sin ejecutar el
código debajo
```

```
        continue
    print(fruta)
```

```
manzana
banana
cereza
pera
```

for break

```
frutas = ["manzana", "banana", "cereza", "uva", "pera"]
for fruta in frutas:
    if fruta == "uva":
        # Si la fruta es "uva", salimos del bucle
        break
    print(fruta)
```

```
manzana
banana
cereza
```

inicializando o modificando variables

```
numeros = [ x for x in range(1,6)]
print(f"lista: {numeros}")

numeros = [x*2 for x in numeros]
print(f"lista modificada: {numeros}")
```

```
lista: [1, 2, 3, 4, 5]
lista modificada: [2, 4, 6, 8, 10]
```

Iterando elementos

Ejemplo 1: Iterando sobre un rango de números

```
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

Ejemplo 2: Iterando sobre una cadena de texto

```
mensaje = "Hola"

for letra in mensaje:
    print(letra)
```

```
H
o
l
a
```

Ejemplo 3: Iterando sobre un Lista

```
frutas = ["manzana", "banana", "cereza"]

for fruta in frutas:
    print(fruta)
```

```
manzana
banana
cereza
```

Si queremos acceder a los indices de una lista podemos utilizar range(list), sin embargo, lo correcto es utilizar enumerate ya que nos regresa una tupla(ind,valor).

INCORRECTO

```
numeros = ["cero","uno", "dos"]

for num in range(len(numeros)):
    print(f"numero: {num}")
```

```
0
1
2
```

CORRECTO

```
numeros = ["cero","uno", "dos"]

for ind, num in enumerate(numeros):
    print(f"ind: {ind} y valor: {num}")
```

```
ind: 0 y valor: cero
ind: 1 y valor: uno
ind: 2 y valor: dos
```

Ejemplo 4: Iterando sobre un Tupla

```
colores = ("rojo", "verde", "azul")

for color in colores:
    print(color)
```

```
rojo
verde
azul
```

Ejemplo 5: Iterando sobre un conjunto

```
numeros = {1, 2, 3, 4, 5}

for numero in numeros:
    print(numero)
```

```
1
2
3
4
5
```

Ejemplo 7: Iterando sobre un diccionario

```
diccionario = {"a": 1, "b": 2, "c": 3}

for clave, valor in diccionario.items():
    print(f"Clave: {clave}, Valor: {valor}")
```

```
Clave: a, Valor: 1
Clave: b, Valor: 2
Clave: c, Valor: 3
```

Bucle while

Un bucle while es otra estructura de control de flujo que se utiliza para repetir un bloque de código mientras una condición sea verdadera. Aquí hay ejemplos de cómo se usa un bucle while en Python:

```
contador = 0

while contador < 5:
    print(contador)
    contador += 1
```

```
0
1
2
3
4
```

do-while simulado

```
while True:
    # Código que siempre se ejecutará al menos una vez

    respuesta = input("¿Desea realizar otra iteración? (s/n): ")

    if respuesta.lower() != 's':
        break
```

while continue y break

```
numero = 0

while numero < 10:
    numero += 1

    if numero % 2 == 0:
        # Si el número es par, continuamos con la siguiente iteración
        continue

    print(numero)

    if numero == 7:
        # Si el número es 7, salimos del bucle
        break
```

```
1
3
5
7
```

while con inputs

```
respuesta = ""

while respuesta.lower() != "salir":
    respuesta = input("Ingrese 'salir' para finalizar: ")
    print(f"Usted ingresó: {respuesta}")
```

En este ejemplo, el bucle while seguirá pidiendo al usuario que ingrese algo hasta que escriba "salir". La función lower() se utiliza para hacer que la comparación sea insensible a mayúsculas y minúsculas.

while para manejo de errores

```
while True:
    try:
        numero = int(input("Ingrese un número entero: "))
        print(f"El doble del número ingresado es: {2 * numero}")
        break
    except ValueError:
        print("Error: Ingrese un número entero válido.")
```

En este ejemplo, el bucle while continuará solicitando al usuario que ingrese un número entero hasta que se proporcione una entrada válida. Se utiliza un bloque try-except para manejar errores si el usuario ingresa algo que no se puede convertir a un número entero.

7 Funciones

Una función en programación es un bloque de código que realiza una tarea específica y puede ser reutilizado en diferentes partes de un programa. Las funciones ayudan a organizar el código, hacerlo más legible, modular y facilitar el mantenimiento. En Python y muchos otros lenguajes de programación, las funciones se definen con la palabra clave def.

Funcion integradas (build-in)

En Python, las funciones integradas (built-in functions) son funciones predefinidas que están disponibles en el intérprete de Python sin necesidad de importar módulos adicionales. Estas funciones son parte del núcleo del lenguaje y proporcionan funcionalidades esenciales que son comúnmente utilizadas en programación.

Aquí hay algunos ejemplos de funciones built-in en Python:

Funciones para operaciones numéricas:

- **max(iterable):** Retorna el valor máximo en el iterable.
- **min(iterable):** Retorna el valor mínimo en el iterable.

```
max_valor = max(5, 10, 3, 8)
min_valor = min(5, 10, 3, 8)
print(max_valor)  # Output: 10
print(min_valor)  # Output: 3
```

- **round(numero, ndigits=None):** Redondea un número al número de dígitos especificado (o 0 si no se especifica).

```
numero_redondeado = round(3.14159, 2)
print(numero_redondeado)  # Output: 3.14
```

Funciones para trabajar con booleanos:

- **bool(valor):** Convierte un valor a un booleano.

```
# Retorna False -> 0, vacio, False, None
# Retorna True -> True, cadena, float, int != 0
print(f"Verificando 0: {bool(0)}")          # -> False
print(f"Verificando vacio: {bool()}")       # -> False
print(f"Verificando False: {bool(False)}")  # -> False
print(f"Verificando None: {bool(None)}")    # -> False

print(f"Verificando True: {bool(True)}")    # -> True
print(f"Verificando cadena: {bool("ad")}")  # -> True
print(f"Verificando 21.1: {bool(21.1)}")    # -> True
print(f"Verificando -1: {bool(-1)}")        # -> True
```

```
resultado_bool = bool(0)
print(resultado_bool)  # Output: False
```

- **all(iterable):** Retorna True si todos los elementos en el iterable son verdaderos.

```
lista_booleanos = [True, True, False, True]
resultado_all = all(lista_booleanos)
print(resultado_all)  # Output: False
```

- **any(iterable):** Retorna True si al menos un elemento en el iterable es verdadero.

```
lista_booleanos = [True, False, False, False]
resultado_any = any(lista_booleanos)
print(resultado_any)  # Output: True
```

Crear funciones

Crear funciones en programación proporciona varios beneficios y facilita el desarrollo de software. Algunas razones clave para crear y utilizar funciones son:

- **Reutilización de código:** Las funciones permiten encapsular un conjunto de instrucciones en una unidad independiente. Esto facilita la reutilización del código en diferentes partes del programa o incluso en diferentes programas. En lugar de repetir el mismo bloque de código, puedes llamar a la función cuando sea necesario.
- **Modularidad:** La modularidad es un principio de diseño que aboga por dividir un programa en partes más pequeñas y manejables (módulos o funciones). Cada función realiza una tarea específica, lo que facilita la comprensión del código y el mantenimiento del mismo.
- **Legibilidad del código:** Al dividir el código en funciones más pequeñas y enfocadas, el código general del programa se vuelve más legible. Cada función puede tener un propósito claro y un nombre descriptivo, facilitando la comprensión del flujo del programa.
- **Abstracción:** Las funciones permiten abstraer detalles de implementación. Cuando llamas a una función, no necesitas conocer los detalles internos de cómo se realiza la tarea; solo necesitas saber qué hace la función y qué tipo de resultados esperar.
- **Facilita la depuración:** Al tener funciones independientes y específicas, es más fácil aislar problemas y depurar el código. Puedes probar y corregir funciones individuales sin afectar otras partes del programa.
- **Facilita la colaboración:** En proyectos más grandes o en equipos de desarrollo, las funciones proporcionan una forma eficiente de dividir el trabajo. Cada miembro del equipo puede trabajar en funciones específicas sin interferir demasiado con el código de los demás.

- **Encapsulamiento:** Las funciones también permiten encapsular datos al aceptar parámetros y devolver resultados. Esto ayuda a crear un entorno controlado para la manipulación de datos, evitando efectos secundarios no deseados en otras partes del programa.
- **Mejora el mantenimiento del código:** Cuando se realizan cambios en los requisitos o se corrigen errores, las funciones permiten realizar modificaciones en un lugar específico, evitando la necesidad de cambiar múltiples instancias del mismo código.

Crear funciones es una práctica esencial en la programación, ya que mejora la legibilidad, promueve la reutilización del código, facilita la depuración y el mantenimiento.

Creando una función

```
def saludar():
    return "Hola a todos"
```

Llamando a la función

```
print(saludar())
```

Funciones con parámetros

```
def tabla(numero, operacion):
    operacion = operacion.lower()
    if operacion == "multiplicacion":
        for num in range(11):
            print(f"{numero} * {num} = {numero * num}")
    elif operacion == "suma":
        for num in range(11):
            print(f"{numero} + {num} = {numero + num}")
    elif operacion == "resta":
        for num in range(11):
            print(f"{numero} - {num} = {numero - num}")
    else:
        print(f"No encontrado")
    return
```

```
tabla(2, "multiplicacion")
tabla(2, "suma")
tabla(2, "resta")
```

Regresando múltiples parámetros

```
def numero_maximo(numero1, numero2):
    maximo = max(numero1, numero2)
    return (numero1, numero2, maximo)
```

```
num1, num2, max = numero_maximo(5,4)
print(f"Entre el número {num1} y {num2} el máximo es {max}")
```

Funciones con multiples parámetros

```
def suma(*numeros):
    return sum(numeros)

print(suma(1,2,3))
print(suma(1,2,3,4,5,6,7,8))
```

Si queremos agregar más parámetros, debemos utilizar *numeros al final.

```
def suma(nombre, *numeros):
    return f"{nombre} la suma de tus número es {sum(numeros)}"

print(suma("Mario",1,2,3))
```

```
# Si agregamos una lista tambien funciona
# Ésta es la forma óptima de sumar valores
def suma(numeros):
    print(f"No los suma...")
    print(*numeros)
    print(f"Si los suma {sum([*numeros])}")

suma([1,2,3,4,5,6,7,8])
```

Reordenando orden de parámetros

```
# Podemos cambiar el orden de los argumentos siempre y cuando agreguemos el nombre
de la variable a cada entrada
def saludo_personalizado(nombre, apellido, adjetivo):
    return f"Hola {nombre} {apellido} eres un {adjetivo}"

print(saludo_personalizado("Pedro","Pérez","capo"))
print(saludo_personalizado(apellido="Pérez", nombre="Pedro",adjetivo="capo"))
```

Predefiniendo parámetros

```
def saludo_personalizado(nombre, apellido, adjetivo ="genial"):
    return f"¡Hola {nombre} {apellido}! Eres realmente {adjetivo}."

print(saludo_personalizado("Pedro","Pérez"))
print(saludo_personalizado(apellido="Pérez", nombre="Pedro",adjetivo="increíble"))
```

Funciones lambda

Las funciones lambda en Python son **funciones anónimas**, es decir, funciones sin nombre. Se crean utilizando la palabra clave lambda y son útiles en situaciones donde se requiere una función simple y breve. Aquí hay algunas características clave y usos comunes de las funciones lambda:

```
# Función normal
def multiplicar_por_dos(numero):
    return numero*2

print(multiplicar_por_dos(5)) # 10

# Misma función pero con lambda
#nombre de funcion = lambda parm: expresion
multiplicar_por_dos = lambda  x:  x * 2

print(multiplicar_por_dos(5)) # 10
```

Otro ejemplo práctico es si queremos obtener los números impares con la función **filter()**.

```
def es_par(numero):
    if(numero % 2 == 0):
        return True

numeros = [1,2,3,4,5,6,7,8,9]

#Usando filter con una función común
numeros_pares = filter(es_par,numeros)

#Regresa solo los pares
print(list(numeros_pares))
```

```
# Con lambda lo hacemos en una sola linea
numeros_pares = filter(lambda num: num % 2 == 0,numeros)
print(list(numeros_pares))
```

8 Modulos

Un módulo es un archivo que contiene código (funciones, variables, y/o clases) que puede ser reutilizado en otros programas. Los módulos permiten organizar y estructurar el código de manera más eficiente, al dividirlo en unidades lógicas y reutilizables. Algunos de los benefícios son:

- **Reutilización de Código:** Los módulos permiten reutilizar funciones y clases en varios programas. Puedes escribir una vez y utilizarlo en múltiples lugares, lo que facilita el mantenimiento y evita la repetición de código.
- **Organización del Código:** Los módulos ayudan a organizar el código al agrupar funciones y clases relacionadas en archivos separados. Esto mejora la legibilidad y mantenimiento del código.
- **Separación de Responsabilidades:** Al dividir el código en módulos, puedes asignar diferentes responsabilidades a cada módulo. Cada módulo puede encargarse de una tarea específica, lo que facilita la gestión y comprensión del código.
- **Espacio de Nombres:** Los módulos proporcionan un espacio de nombres separado para sus variables y funciones. Esto evita colisiones de nombres entre diferentes partes de tu código y mejora la encapsulación.
- **Facilita la Colaboración:** En proyectos grandes, los módulos facilitan la colaboración entre diferentes desarrolladores o equipos. Cada módulo puede ser desarrollado y mantenido de manera independiente.
- **Facilita las Actualizaciones:** Si necesitas actualizar o mejorar una parte específica de tu código, solo necesitas modificar el módulo correspondiente sin afectar otras partes del programa.
- **Mejora la Escalabilidad:** A medida que tu proyecto crece, los módulos te permiten escalar de manera más efectiva. Puedes agregar nuevos módulos o modificar existentes sin afectar el resto del código.
- **Facilita las Pruebas Unitarias:** La modularidad facilita la creación y ejecución de pruebas unitarias. Puedes probar cada módulo de manera independiente, asegurando que cada parte del código funcione correctamente.

Para utilizar un módulo en un programa Python, puedes importarlo mediante la palabra clave import.

Creando módulo

```
# El módulo es un archivo.py (m_saludo.py)
def saludar(name):
    return f"Hola {name} cómo te va?"

def saludar_formal(name):
    return f"Hola {name} cómo se encuentra?"
```

Importando módulo

```
import m_saludo
saludo = m_saludo.saludar("Jose")
```

```
import m_saludo as saludar
saludo = saludar.saludar("Jose")
print(saludo)
```

Importando funciones de modulos

```
#from m_saludo import saludar                # Llamando una función
#from m_saludo import *                      # Llamando todas las funciones
from m_saludo import saludar, saludar_formal # Llamando dos funciones

saludo = saludar("Jose")
saludo_formal = saludar_formal("Jose")
print(saludo)
print(saludo_formal)
```

Enrutamiento de módulos

El enrutamiento de módulos es la manera en como se estructuran y utilizan los módulos en un proyecto.

Módulo en carpeta

```
# Si el modulo está en una carpeta
#      nombre_carpeta    .nom_modulo. función
import modulos_en_carpeta.m_en_carpeta
print(modulos_en_carpeta.m_en_carpeta.imprimir())
```

```
# Agregando un nombre a nuestro módulo
import modulos_en_carpeta.m_en_carpeta as saludar
print(saludar.imprimir())
```

Submódulos

```
# Si el modulo está dentro de varias carpetas las podemos llamar de la sigueinte
manera
import modulos_en_carpeta1.modulos_en_carpeta2.m_en_carpeta
print(modulos_en_carpeta1.modulos_en_carpeta2.m_en_carpeta.imprimir())
```

```
# Agregando un nombre a nuestro módulo
import modulos_en_carpeta1.modulos_en_carpeta2.m_en_carpeta as saludar
print(saludar.imprimir())
```

Agregando ruta al sistema (sys)

Al agregar la ruta al sistema podemos acceder a los datos de manera sencilla. Esto funciona por ejecución no se agrega permanentemente.

```
import sys

# sys.builtin_module_names es parecido a dir() y regresa todos los nombres de los
modulos creados
# Ésto sirve para ver los módulo creados por python no se repitan con los nuestros
# Si esto pasa nuestro módulo tendra conflictos y nunca será llamado
print(sys.builtin_module_names)

# Ver rutas de python
print(sys.path)

# Agregamos una ruta
sys.path.append("c:\\Users\\Jose\\Desktop\\Curso
Python\\8_Modulos\\modulos_en_carpeta")
print(sys.path)

import m_saludar
print(m_saludar.saludar("Jose"))
```

Paquetes

En Python, un paquete es una forma de organizar y estructurar módulos relacionados en un directorio. Un paquete es simplemente un directorio que contiene un archivo especial llamado **init**.py y posiblemente subdirectorios y archivos adicionales. La presencia del archivo **init**.py indica que el directorio debe tratarse como un paquete de Python.

Aquí hay algunas características clave de los paquetes en Python:

- **Organización Jerárquica:** Los paquetes permiten organizar módulos en una estructura jerárquica. Puedes tener subpaquetes (directorios dentro de un paquete) para organizar aún más tu código.

```
mi_paquete/
├── __init__.py
├── modulo_a.py
├── modulo_b.py
└── subpaquete/
    ├── __init__.py
    └── modulo_c.py
```

- **Archivo init.py:** El archivo **init**.py puede estar vacío o contener código de inicialización para el paquete. La presencia de este archivo es necesaria para que Python trate el directorio como un paquete.
- **Importación Jerárquica:** Puedes importar módulos de un paquete utilizando la notación de puntos. Por ejemplo:

```
from mi_paquete import modulo_a
from mi_paquete.subpaquete import modulo_c
```

- **Facilita la Organización:** Los paquetes son útiles para organizar proyectos más grandes y complejos. Proporcionan una manera estructurada de organizar y acceder a los módulos.
- **Evita Colisiones de Nombres:** Al tener subpaquetes y módulos organizados en una estructura de directorios, se evitan colisiones de nombres entre módulos de diferentes partes de tu código.

9 Archivos

En Python, los archivos son utilizados para realizar operaciones de entrada y salida (I/O) de datos. Los archivos permiten a los programas leer información de fuentes externas, como archivos en el sistema de archivos del computador, y escribir datos para almacenarlos de forma persistente. Un archivo es un es un contenedor de información con un formato (txt, csv, tsv, png, jpg, mp4).

Archivos txt

Leer archivo

```
# Encoding UTF-8 (codificación universal) se utiliza para no tener problemas con
carácteres especiales
archivo_sin_leer = open("texto.txt", encoding="UTF-8")

# Leemos el archivo completo
archivo_leido = archivo_sin_leer.read()
print(archivo_leido)

# Leemos el archivo por lineas
archivo_por_lineas = archivo_sin_leer.readlines()
print(archivo_por_lineas)

# Leemos la primera linea del archivo   Si agregamos .readline(20) solo lee los
primeros 20 caracteres
archivo_primera_linea = archivo_sin_leer.readline()
print(archivo_primera_linea)
```

Cuando abrimos un archivo es importante cerrarlos despues de utilizarlos, para ello utilizamos la función .close()

```
#Cerramos el archivo
archivo_sin_leer.close()
```

Si no queremos hacerlo manualmente utilizamos la clausula with para cerrarlos automáticamente

```
with open("texto.txt", encoding="UTF-8") as archivo_sin_leer:
    archivo_primera_linea = archivo_sin_leer.readline()
    print(archivo_primera_linea)
```

Escribir archivo

```
#w write a append
with open("texto2.txt", 'w', encoding="UTF-8") as archivo:
    texto = """Escribiendo en el archivo\nRevisando si se escribió bien"""

    #re-escribiendo el archivo
    archivo.write(texto)

    archivo.writelines(["\nAgregando más lineas","\nMuchas más"])
```

Archivos csv

Abrir archivo

```
import csv

with open("9_Archivos\\archivo_csv.csv") as archivo:
    #Retorna un iterable
    reader = csv.reader(archivo)
    for row in reader:
        print(row)
```

Dataframe

Un DataFrame es una estructura de datos bidimensional en pandas, una biblioteca de análisis de datos en Python. Es similar a una hoja de cálculo o una tabla de base de datos, y se utiliza comúnmente para representar y manipular datos tabulares.

Algunas características clave de un DataFrame en pandas incluyen:

- **Bidimensional:** Un DataFrame es bidimensional, lo que significa que tiene filas y columnas. Puedes pensar en él como una tabla en la que cada fila representa un conjunto de datos y cada columna representa una variable.
- **Índice:** Los DataFrames tienen un índice que identifica de manera única cada fila. Puedes utilizar el índice para acceder a filas específicas y realizar operaciones de manipulación de datos.
- **Columnas con Nombres:** Cada columna en un DataFrame tiene un nombre que la identifica. Puedes acceder a las columnas por su nombre y realizar operaciones en columnas específicas.
- **Datos Heterogéneos:** Los DataFrames pueden contener datos de diferentes tipos en diferentes columnas. Puedes tener columnas con números enteros, números de punto flotante, cadenas, etc.
- **Funciones Integradas:** Pandas proporciona muchas funciones integradas para realizar operaciones comunes en los DataFrames, como filtrar datos, calcular estadísticas, realizar agrupaciones y más.

```
import pandas as pd

df_archivo  = pd.read_csv("9_Archivos\\archivo_csv.csv", names=
['nombre','apellido','edad'], skiprows=[0])
print(df_archivo.head())

# reordenar columnas
print(df_archivo['edad'].head())

# reordenar por edad
print(df_archivo.sort_values(by='edad',ascending= True))

#concatenar df
print(pd.concat([df_archivo,df_archivo]))

print("###################################")
#Primeras 3
print(df_archivo.head(3))
#Ultimas 3
print(df_archivo.tail(3))
print("###################################")

filas, columnas = df_archivo.shape

#Analisis estadistico
print(df_archivo.describe())
```

Accediendo a elementos

```
#Accediendo a un elemento específico con loc
#Accediendo a la edad de la fila 2 con loc
#                          fila, columna
elemento_df = df_archivo.loc[2,'edad']
```

```
print(elemento_df)

#Accediendo a la edad de la fila 2 con iloc
#                indice de fila, columna
elemento_df = df_archivo.iloc[2,2]
print(elemento_df)

# Accediendo a todos los valores de una columna con loc
columna_df = df_archivo.loc[:,'nombre']
print(columna_df)

# Accediendo a todos los valores de una columna con iloc
columna_df = df_archivo.iloc[:,1]
print(columna_df)


# Accediendo a todos los valores de una columna con loc
fila_df = df_archivo.loc[2,:]
print(fila_df)

# Accediendo a todos los valores de una columna con iloc
fila_df = df_archivo.iloc[2,:]
print(fila_df)

mayor_que_20 = df_archivo.loc[df_archivo['edad']>20,:]
print(mayor_que_20)

#Slicing
cadena = "abcdefghijklmnopqrstuvwxyz"

print(cadena[0:10])
print(cadena[3:10])
```

Accediendo a Archivos grandes

Para archivos muy grandes es mejor utilizar pandas y acceder a los datos por paquetes

```
import pandas as pd
def read_csv_in_chuncks(file_name):
    for i, chunk in enumerate(pd.read_csv(file_name,chunksize=1000)):
        print("chunk #{}".format(i))
        print(chunk)

read_csv_in_chuncks("big_file.csv")
```

10 Graficas

Para realizar gráficos en Python, puedes utilizar las bibliotecas pandas, matplotlib y seaborn. A continuación, te proporcionaré ejemplos básicos de cómo utilizar estas bibliotecas para crear gráficos a partir de un

DataFrame de pandas.

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

df = pd.read_csv("datos_graficos.csv",names=["fecha","pasajeros"])


#           Tiene que tener el mismo nombre en el df
sns.lineplot(x="fecha", y="pasajeros", data=df)


# Encontrar el índice del máximo valor en la columna 'pds'
indice_maximo = df['pasajeros'].idxmax()

# Obtener las coordenadas del punto máximo
x_maximo = df.loc[indice_maximo, 'fecha']
y_maximo = df.loc[indice_maximo, 'pasajeros']

# Resaltar el punto máximo en el gráfico
plt.scatter(x_maximo, y_maximo, color='red', label='Máximo')

# Mostrar la leyenda
plt.legend()

plt.show()
```

Excepciones

Una excepción es un evento que ocurre durante la ejecución de un programa y que interrumpe el flujo normal de instrucciones. Las excepciones son situaciones inesperadas o errores que pueden ocurrir durante la ejecución del código y que pueden necesitar ser manejados de manera especial.

En Python, las excepciones se representan mediante objetos llamados "excepciones". Cada tipo de error tiene su propia clase de excepción. Algunos ejemplos comunes de excepciones en Python incluyen:

```
def suma():
    while True:
        num1 = input("Numero 1: ")
        num2 = input("Numero 2: ")
        try:
            resultado = int(num1) + int(num2)
        except:
            print("ambos valores deben ser númericos")
            # Podemos enviar como mensaje el error que lanza python
            print(f"ERROR: {e}")
            # Obteber el nombre de la excepción y agregarla al try (tomar en
cuenta que si except general está en el try los demás no se ejecutan)
```

```
            print(f"Nombre del error: {type(e).__name__}")
        else:
            # Si todo sale bien, terminamos el bucle
            break
        finally:
            # Esto se ejecuta cada iteración casi no se utiliza
            print("programa terminado")
    return resultado

print(suma())
```

Identificando excepciones

```
#Podemos guardar tambien las excepciones
def suma():
    while True:
        num1 = input("Numero 1: ")
        num2 = input("Numero 2: ")
        try:
            resultado = int(num1) + int(num2)
        except ZeroDivisionError as e:
            print(f"No dividas por cero")
        except ValueError as ex:
            print(f"Agrega solo números")
        else:
            break
        finally:
            # Esto se ejecuta cada iteración casi no se utiliza
            print("programa terminado")
    return resultado

print(suma())
```

Identificando nombre de error

```
import traceback

try:
    x = 10
    y = "abc"
    resultado = x + y  # Esto generará un TypeError
except Exception as e:
    # Imprime la información detallada sobre la excepción
    traceback.print_exc()
```

Excepciones propias

```
class MiExcepcion(Exception):
    def __init__(self, err):
        print(f"Cometiste el siguiente error: {err}")


# raise sirve para lanzar excepciones por lo tanto se ejecuta el try y except
try:
    raise MiExcepcion("Prueba de error de excepcion")
except:
    print("No te vuelvas a equivocar mano")
```

12 Expresiones regulares

Las expresiones regulares, comúnmente conocidas como "regex" o "regexp", son secuencias de caracteres que forman un patrón de búsqueda. Estos patrones se utilizan para buscar, analizar y manipular texto basándose en ciertas reglas definidas. Las expresiones regulares son una herramienta poderosa y flexible utilizada en la mayoría de los lenguajes de programación y en diversas aplicaciones para realizar operaciones relacionadas con cadenas de texto.

Características clave de las expresiones regulares:

- **Patrones de Búsqueda:** Las expresiones regulares permiten definir patrones de búsqueda que describen conjuntos específicos de cadenas de texto. Estos patrones pueden incluir caracteres literales, metacaracteres y constructos especiales.
- **Metacaracteres:** Los metacaracteres son caracteres especiales con significados particulares dentro de una expresión regular. Por ejemplo, el punto (.) coincide con cualquier carácter, el asterisco (*) coincide con cero o más repeticiones del carácter anterior, y el signo de interrogación (?) indica que el carácter anterior es opcional.
- **Coincidencia y Extracción:** Las expresiones regulares se utilizan para buscar coincidencias o para extraer partes específicas de un texto. Puedes buscar si una cadena cumple con un patrón específico y extraer información relevante de esa cadena.
- **Validación de Formato:** Las expresiones regulares son útiles para validar si una cadena de texto cumple con un formato específico. Puedes verificar si un número de teléfono, una dirección de correo electrónico o cualquier otro formato sigue la estructura esperada.
- **Reemplazo de Texto:** Las expresiones regulares se utilizan para buscar y reemplazar patrones específicos en un texto. Puedes realizar cambios en un texto basándote en reglas definidas por patrones.
- **Agrupación y Captura:** Puedes agrupar partes de un patrón utilizando paréntesis, lo que permite capturar y acceder a subcadenas específicas. Esto es útil para extraer información específica de una cadena.
- **\d =** Busca digitos numéricos del 0 - 9

- **\D =** Busca TODO menos los digitos numéricos
- **\w =** Busca caracteres alfanuméricos [a-z A-Z 0-9 _]
- **\W =** Busca TODO menos caracteres alfanuméricos [a-z A-Z 0-9 _]
- **\s =** Busca los espacios en blanco > espacios, tabs, saltos de linea
- **\S =** Busca TODO menos los espacios en blanco > espacios, tabs, saltos de linea
- **\n =** Busca saltos en linea
- **. =** Busca todo menos saltos de linea
- **\ =** Cancela caracteres especiales . para buscar punto _ para buscar _
- **^ =** Busca el comienzo de una linea
- **$ =** Busca el final de una linea
- **{n} =** repite n cantidad de veces el valor de la izquierda
- **{n,m} =** Busca como mínimo n y máximo m
- **| =** Busca una cosa o la otra (si ambos cumples devuelve ambos)
- *** =** Es un cuatificador que indica que el elemento precedente puede parecer cero o más veces

```
import re

texto = """Hola maestro, esta es la cadena 111, como estas mi capitan
esta es la segunda linea de texto (2. ) (probando abbb ababab)
Y esta es la tercera y definitiva capitanazo Hola
"""

# Crear un objeto de compilación de expresiones regulares con ignorecase
regex = re.compile("Hola", flags=re.IGNORECASE)

# Buscar la primera palabra dentro de la variable y retorna el inicio y final
resultado = regex.search(texto)
inicio, fin = resultado.start(), resultado.end()

print(f"El inicio es: {inicio} y el final es: {fin}")

# Buscar un número, punto y espacio en blanco
resultados = re.findall(r"\d\.\s", texto)

# Buscar el principio de línea
resultados = re.findall(r"^Hola", texto)

# Buscar el principio de línea (multilínea)
resultados = re.findall(r"^esta", texto, flags=re.M)
```

```
# Buscar el final de línea
resultados = re.findall(r"Hola$", texto)
resultados = re.findall(r"capitan$", texto, flags=re.M)

# Buscar tres números juntos
resultados = re.findall(r"\d{3}", texto)
resultados = re.findall(r"1{3}", texto)

# Buscar una letra 'a' seguida de tres 'b'
resultados = re.findall(r"ab{3}", texto)

# Buscar tres 'a' y tres 'b' usando grupos
resultados = re.findall(r"(ab){3}", "abababab ababab abc ab abab")
# Regresa ['ab', 'ab'] porque solo abababab y ababab cumplen

# Buscar tres 'a' o tres 'b' usando conjuntos
resultados = re.findall(r"[ab]{3}", "abaaaabbabbbb")
# Regresa ['aba', 'aaa', 'bba', 'bbb'] porque separa abaaaabbabbbb en grupos [aba
aaa bba bbb] b

print(resultados)
```