ÍNDICE

1. Introducción

Las funciones son una herramienta fundamental en la programación de Python que nos proporciona dos ventajas principales para mejorar la eficiencia y organización del código.

Al definir una función, encapsulamos un conjunto de instrucciones con una funcionalidad específica. Esto nos permite utilizar esa funcionalidad en diferentes partes del programa sin tener que volver a escribir el mismo código repetidamente. La reutilización de funciones promueve un enfoque modular y evita la duplicación innecesaria de código.

Al agrupar bloques de código en funciones con nombres descriptivos, podemos mejorar significativamente la organización y legibilidad del código. En lugar de tener un programa monolítico y extenso, las funciones nos permiten dividir el código en tareas más manejables y comprensibles. Esto facilita el mantenimiento, la depuración y la colaboración en proyectos de programación más grandes.

1.1. Objetivos

- Entender la sintaxis para definir funciones en Python.
- Aprender a usar parámetros relacionados a las funciones y retorno de valores.
- Comprender la importancia de las variables
- Conocer algunas funciones básicas incluidas en Python.

2. Funciones

Las funciones en Python son bloques de código que encapsulan una serie de instrucciones con el propósito de realizar una tarea específica. Un experto en Python entiende que las funciones son fundamentales para escribir código limpio, modular y reutilizable. Su experiencia le permite aprovechar plenamente las ventajas que ofrecen las funciones en el desarrollo de aplicaciones complejas.

A continuación, presentamos los tres niveles de complejidad en la sintaxis para definir una función en Python, desde el formato básico hasta el avanzado:

- **Básico**: El formato básico para definir una función en Python consta de tres elementos fundamentales: la palabra clave def, el nombre de la función y, opcionalmente, los parámetros entre paréntesis. La definición de la función se cierra con dos puntos y seguido por el bloque de código indentado que representa el cuerpo de la función.

```
def nombre_funcion(parametro1, parametro2, ...):
    # Cuerpo de la función
    # Instrucciones que realizan una tarea específica
    return resultado
```

- **Intermedio**: En el formato intermedio, además de los parámetros posicionales, podemos utilizar argumentos con nombre y asignar valores predeterminados a algunos parámetros. Esto permite una mayor flexibilidad al llamar a la función, ya que no es necesario proporcionar todos los argumentos.

```
def nombre_funcion(parametro1, parametro2=valor_predeterminado,
parametro3=valor_predeterminado, ...):
    # Cuerpo de la función
    # Instrucciones que realizan una tarea específica
    return resultado
```

- **Avanzado**: En el formato avanzado, podemos utilizar argumentos indeterminados, que nos permiten manejar una cantidad variable de argumentos en la función. Esto se logra utilizando *args y **kwargs, donde

*args representa una tupla de argumentos posicionales y **kwargs representa un diccionario de argumentos con nombre.

```
def nombre_funcion(parametro1, parametro2, *args,
parametro3=valor_predeterminado, **kwargs):
    # Cuerpo de la función
    # Instrucciones que realizan una tarea específica
    return resultado
```

Ejemplo Básico de la Creación de una Función:

Supongamos que queremos crear una función simple que calcule el área de un rectángulo dado su ancho y altura. Utilizaremos el formato básico para definir la función:

```
def area_rectangulo(ancho, altura):
    area = ancho * altura
    return area
```

En este ejemplo, hemos definido la función area_rectangulo con dos parámetros ancho y altura. El cuerpo de la función realiza el cálculo del área multiplicando ambos valores, y luego devuelve el resultado mediante la instrucción return. Con esta función definida, podemos llamarla y pasarle los valores necesarios para obtener el área del rectángulo:

```
ancho_rectangulo = 5
altura_rectangulo = 10
area_del_rectangulo = area_rectangulo(ancho_rectangulo, altura_rectangulo)

print("El área del rectángulo es:", area_del_rectangulo)
```

2.1. Aspectos de las funciones

- **Modularidad y Reutilización de código:** Un experto en Python comprende la importancia de la modularidad al dividir un programa en funciones coherentes y autónomas. Esto facilita la reutilización de código, lo que ahorra tiempo y esfuerzo al evitar la duplicación de lógica en diferentes partes del programa.

- **Declaración y Uso:** Un experto entiende cómo declarar funciones utilizando la palabra clave def seguida del nombre de la función, los parámetros y el bloque de código indentado. Además, sabe cómo llamar a una función para ejecutar su comportamiento y cómo manejar los valores de retorno si es necesario.
- **Parámetros y Argumentos:** Un experto en Python comprende los diferentes tipos de parámetros que pueden definirse en una función, incluidos los parámetros posicionales, los argumentos con nombre y los valores predeterminados. También es consciente de cómo manejar una cantidad variable de argumentos utilizando *args y **kwargs.
- **Variables locales y globales:** Un experto entiende la diferencia entre las variables locales, que solo son accesibles dentro de la función, y las variables globales, que tienen un alcance más amplio. Comprende cómo evitar problemas de ámbito y cómo usar variables globales de manera controlada.
- **Recursión:** Un experto en Python sabe cómo aplicar la recursión cuando una función se llama a sí misma. Comprende cuándo utilizar la recursión de manera eficiente y cómo evitar posibles problemas de desbordamiento de pila.
- **Funciones Lambda:** Un experto está familiarizado con las funciones lambda, también conocidas como funciones anónimas, que son funciones pequeñas y compactas que pueden definirse en una sola línea. Entiende cómo utilizarlas en expresiones funcionales y en situaciones donde se requieren funciones rápidas y sencillas.
- **Funciones integradas y módulos:** Un experto en Python conoce las funciones integradas que proporciona el lenguaje, como len(), map(), filter(), etc. Además, sabe cómo importar funciones y definiciones desde módulos y cómo estructurar un programa utilizando módulos para una mayor organización y claridad.
- **Buenas prácticas de nomenclatura y documentación:** Un experto en Python sigue las buenas prácticas para nombrar funciones de manera clara y descriptiva. También comprende la importancia de documentar las funciones utilizando docstrings para facilitar la comprensión y el mantenimiento del código.

2.2. Retorno de Valores de las Funciones

En Python, las funciones pueden devolver valores que son utilizados en el código que llamó a la función. El valor devuelto se especifica con la instrucción return dentro del cuerpo de la función. Al utilizar return, podemos enviar información desde la función al programa principal, lo que permite realizar tareas más complejas y utilizar los resultados obtenidos en otros cálculos o acciones.

- **Retornar valores:** Para retornar un valor desde una función, simplemente se utiliza la instrucción return seguida del valor o expresión que deseamos retornar.

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

# Llamada a la función y almacenamiento del valor retornado en
una variable
resultado_suma = sumar(3, 5)
print(resultado_suma)  # Output: 8
```

En este ejemplo, la función sumar() toma dos argumentos a y b, y devuelve la suma de ambos con return resultado. Al llamar a la función y almacenar el valor retornado en la variable resultado_suma, podemos imprimir el resultado.

- **Retornar múltiples Valores:** Para retornar múltiples valores desde una función, simplemente se separan los valores o expresiones con comas después de la instrucción return. Es común utilizar tuplas para agrupar los valores a retornar.

```
def dividir_y_obtener_resto(dividendo, divisor):
    cociente = dividendo // divisor
    resto = dividendo % divisor
    return cociente, resto

# Llamada a la función y almacenamiento de los valores retornados
en variables
resultado_cociente, resultado_resto = dividir_y_obtener_resto(17,
5)

print("Cociente:", resultado_cociente)  # Output: 3
print("Resto:", resultado_resto)        # Output: 2
```

En este ejemplo, la función dividir_y_obtener_resto() toma dos argumentos dividendo y divisor, y calcula el cociente y el resto de la división. Luego, utiliza return cociente, resto para retornar ambos valores en una tupla. Al llamar a la función y desempaquetar los valores retornados en las variables resultado_cociente y resultado_resto, podemos imprimir ambos resultados por separado.

- **Retorno de funciones como Resultados:** En Python, una función puede retornar otra función como resultado. Esto se logra utilizando las funciones como objetos de primera clase y es útil en conceptos como funciones de orden superior y cierre

```
    def crear_sumador(valor):
        def sumar(num):
            return num + valor
        return sumar

    sumar_5 = crear_sumador(5)
    resultado = sumar_5(10)  # Sumará 5 al número 10
    print(resultado)  # Output: 15
```

Las funciones pueden llamarse a sí mismas de forma recursiva, lo que significa que la función se invoca dentro de su propio cuerpo. En este caso, la función recursiva puede retornar un valor o, en algunos casos, acumular resultados a medida que la recursión se desenrolla.

```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n - 1)

resultado_factorial = factorial(5)  # 5! = 5 * 4 * 3 * 2 * 1 = 120
print(resultado_factorial)  # Output: 120
```

2.3. anotaciones

En Python, puedes consultar las anotaciones de una función utilizando la función incorporada __annotations__. Esta función devuelve un diccionario que contiene las anotaciones de tipo de los argumentos y el tipo de retorno de la función, si se han proporcionado.

```
def calcular_area_circulo(radio: float) -> float:
    # Fórmula para calcular el área del círculo
    area = 3.14159 * radio ** 2
    return area

# Imprimir las anotaciones
print(calcular_area_circulo.__annotations__)
```

hemos agregado la anotación de tipo -> float al final de la definición de la función para indicar que el valor de retorno también es un número de tipo float.

El diccionario anotaciones contiene dos claves: 'radio' y 'return'. La clave 'radio' se refiere al argumento de la función calcular_area_circulo, y su valor es <class 'float'>, que indica que se espera un tipo de dato float. La clave 'return' se refiere al tipo de retorno de la función, y su valor también es <class 'float'>, indicando que la función devuelve un valor de tipo float.

3. Parámetros

Los parámetros son elementos esenciales que permiten personalizar el comportamiento de las funciones. Son valores que se pasan a las funciones cuando se llaman y que se utilizan para realizar operaciones específicas en el cuerpo de la función. Por este motivo, podemos declarar unos parámetros que la función usará para leer esa información. Hay que saber diferenciar entre los parámetros, que son los valores que definimos en una función, y los argumentos, que son los valores que introducimos a la función en el momento de la ejecución.

3.1. Parámetros Posicionales

Los parámetros posicionales son aquellos que se definen en la firma de la función y se pasan en el mismo orden en que fueron definidos al llamar a la función. El orden de los argumentos es crucial en este caso, ya que Python asigna automáticamente los valores en función de su posición. Es decir, el primer argumento que se pase al llamar a la función se asignará al primer parámetro definido, el segundo argumento al segundo parámetro, y así sucesivamente.

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

resultado = sumar(3, 5)
print(resultado)  # Output: 8
```

En este ejemplo, a y b son parámetros posicionales. Al llamar a la función sumar(3, 5), Python asigna el valor 3 al parámetro a y el valor 5 al parámetro b.

3.2. Parámetros con Nombres

Los parámetros con nombre son aquellos que se definen en la firma de la función y se pasan mediante su nombre específico al llamar a la función. Al usar argumentos con nombre, podemos proporcionar los valores en cualquier orden, siempre que indiquemos el nombre del parámetro al que pertenecen. Esto mejora la legibilidad y claridad del código, especialmente cuando se tienen funciones con múltiples parámetros.

```
def saludar(nombre, mensaje):
    print(f"{mensaje}, {nombre}.")

# Llamada a la función con argumentos con nombre
saludar(nombre="Ana", mensaje="¡Hola")  # Output: ¡Hola, Ana.
saludar(mensaje="¡Buenos días", nombre="Juan")  # Output: ¡Buenos días,
Juan.
```

3.3. Valores Predeterminados

Los valores predeterminados son valores asignados a los parámetros en la definición de una función. Si al llamar a la función no se proporciona un valor para un parámetro, se utilizará su valor predeterminado en su lugar. Esto permite que algunos argumentos sean opcionales y no sea necesario proporcionar un valor cada vez que se llama a la función.

```
def saludar(nombre, mensaje="Hola"):
    print(f"{mensaje}, {nombre}.")

# Llamada a la función sin especificar el argumento 'mensaje'
saludar("Ana")  # Output: Hola, Ana.

# Llamada a la función especificando un valor diferente para 'mensaje'
saludar("Juan", "¡Buenos días")  # Output: ¡Buenos días, Juan.
```

En este ejemplo, el parámetro mensaje tiene un valor predeterminado de "Hola". Si no se proporciona un valor para mensaje, se utilizará el valor predeterminado. Si se proporciona un valor, este reemplazará el valor predeterminado.

3.4. Argumentos Indeterminados

los argumentos indeterminados permiten manejar una cantidad variable de argumentos en una función. Esto es útil cuando no se conoce la cantidad exacta de argumentos que se van a pasar o cuando se desea proporcionar una mayor flexibilidad a la función. Existen dos tipos principales de argumentos indeterminados.

- ***args:** Este es un parámetro especial que permite recibir una cantidad variable de argumentos posicionales en forma de tupla. El nombre args es

solo una convención, pero el asterisco (*) es lo que indica que estamos trabajando con argumentos indeterminados. Al usar *args, podemos pasar cualquier número de argumentos posicionales a la función.

```
def sumar(*args):
    resultado = sum(args)
    return resultado

resultado_suma = sumar(1, 2, 3, 4, 5)
print(resultado_suma)  # Output: 15
```

En este ejemplo, hemos definido la función sumar() con el parámetro *args. Al llamar a la función con varios argumentos posicionales, Python los agrupa automáticamente en una tupla llamada args. Luego, podemos usar esta tupla para realizar operaciones, como en este caso, donde utilizamos la función sum() para obtener la suma de todos los valores pasados.

- ****kwargs:** Este es otro parámetro especial que permite recibir una cantidad variable de argumentos con nombre en forma de diccionario. El nombre kwargs es también una convención, y el doble asterisco (**) es lo que indica que estamos trabajando con argumentos indeterminados con nombre. Al usar **kwargs, podemos pasar cualquier número de argumentos con nombre a la función.

```
def mostrar_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

mostrar_info(nombre="Ana", edad=30, ciudad="Madrid")
# Output:
# nombre: Ana
# edad: 30
# ciudad: Madrid
```

En este ejemplo, hemos definido la función mostrar_info() con el parámetro **kwargs. Al llamar a la función con varios argumentos con nombre,

Python los agrupa automáticamente en un diccionario llamado kwargs. Luego, podemos iterar sobre este diccionario para mostrar la información que contiene.

- **Uso Combinado de *args y **kwargs:** Es posible utilizar tanto *args como **kwargs en una misma función. En este caso, es importante que los argumentos posicionales sean pasados antes de los argumentos con nombre al llamar a la función.

```
def ejemplo_combinado(*args, **kwargs):
    print("Argumentos posicionales:")
    for arg in args:
        print(arg)

    print("\nArgumentos con nombre:")
    for key, value in kwargs.items():
        print(f"{key}: {value}")

ejemplo_combinado(1, 2, 3, nombre="Ana", edad=30,
ciudad="Madrid")
```

En este ejemplo, hemos definido la función ejemplo_combinado() con ambos parámetros *args y **kwargs. Al llamar a la función, hemos pasado argumentos posicionales (1, 2, 3) seguidos de argumentos con nombre (nombre="Ana", edad=30, ciudad="Madrid"). La función puede manejar ambos tipos de argumentos de manera simultánea.

En conclusión, los argumentos indeterminados (*args y **kwargs) brindan una gran flexibilidad al permitir que las funciones manejen una cantidad variable de argumentos. Estos elementos son especialmente útiles en situaciones donde la cantidad o el tipo de argumentos pueden variar, proporcionando una mayor adaptabilidad y versatilidad a nuestras funciones.

4. Aspectos adicionales relacionados con las funciones

A. Funciones Lambda:

Las funciones lambda, también conocidas como funciones anónimas, son funciones pequeñas y de una sola línea que se definen sin nombre. Se utilizan para expresiones simples y se crean utilizando la palabra clave lambda. Son útiles en situaciones donde se requiere una función temporal y no es necesario definirla formalmente.

```
cuadrado = lambda x: x ** 2
resultado = cuadrado(5)
print(resultado)  # Output: 25
```

B. Alcance de variables (Scope):

Las variables pueden tener diferentes alcances en Python. Las variables definidas dentro de una función tienen un alcance local y solo existen dentro de esa función. Las variables definidas fuera de cualquier función tienen un alcance global y pueden ser accesibles en todo el programa. Se puede utilizar la palabra clave global para modificar una variable global dentro de una función.

```
x = 10

def ejemplo_alcance():
    y = 5
    global x
    x = 20
    print(x, y)

ejemplo_alcance()  # Output: 20 5
print(x)           # Output: 20
```

C. Funciones Recursivas Terminales:

Una función recursiva es considerada "terminal" cuando no realiza ninguna operación adicional después de hacer la llamada recursiva. Estas

funciones se caracterizan por su uso eficiente de la memoria, ya que no generan una pila de llamadas recursivas en el stack.

```
def factorial(n, resultado=1):
    if n == 0 or n == 1:
        return resultado
    else:
        return factorial(n - 1, resultado * n)

resultado_factorial = factorial(5)  # 5! = 5 * 4 * 3 * 2 * 1 = 120
print(resultado_factorial)  # Output: 120
```

D. Funciones Decoradoras:

Las funciones decoradoras son funciones que envuelven a otras funciones para extender o modificar su comportamiento. Se utilizan para agregar funcionalidades adicionales a una función sin modificar su código interno.

```
def mi_decorador(funcion):
    def nueva_funcion():
        print("Realizando acciones antes de llamar a la función
original.")
        funcion()
        print("Realizando acciones después de llamar a la función
original.")
    return nueva_funcion

@mi_decorador
def saludar():
    print("¡Hola, mundo!")

saludar()
# Output:
# Realizando acciones antes de llamar a la función original.
# ¡Hola, mundo!
# Realizando acciones después de llamar a la función original.
```

E. Funciones Generadoras:

Las funciones generadoras utilizan la palabra clave yield en lugar de return para devolver valores. Cuando una función generadora se llama, no se

ejecuta completamente; en cambio, devuelve un generador que puede usarse para obtener valores bajo demanda, lo que ahorra memoria y recursos.

```
def generador_contador(maximo):
    contador = 0
    while contador <= maximo:
        yield contador
        contador += 1

contador_gen = generador_contador(5)
for numero in contador_gen:
    print(numero)  # Output: 0, 1, 2, 3, 4, 5
```

F. Funciones Internas (Nested Functions):

Es posible definir una función dentro de otra función, lo que crea una función interna (también conocida como función anidada). La función interna tiene acceso a las variables locales de la función externa, incluso después de que la función externa haya finalizado su ejecución.

```
def funcion_externa():
    x = 10

    def funcion_interna():
        print("Función interna:", x)

    funcion_interna()

funcion_externa()
# Output: Función interna: 10
```

5. Documentación de funciones

La documentación de funciones es una práctica esencial en el desarrollo de software, ya que proporciona información detallada sobre el propósito, los parámetros y el comportamiento de una función. La documentación facilita la comprensión del código y permite a otros desarrolladores utilizar y colaborar de manera efectiva con el código que se ha escrito. En Python, se utiliza una convención específica para documentar funciones mediante el uso de cadenas de texto llamadas docstrings.

Una docstring (cadena de documentación) es una cadena de texto que se coloca dentro de la definición de una función para proporcionar una descripción clara y concisa del propósito y comportamiento de esa función. Las docstrings se utilizan para documentar funciones, módulos y clases en Python. Pueden contener información sobre los parámetros, los valores de retorno, los efectos secundarios y cualquier otra información relevante que ayude a comprender cómo utilizar la función correctamente.

- **Sintaxis de las Docstrings:** Las docstrings se definen utilizando comillas triples (''' o """) al principio y al final del bloque de texto. Las comillas triples permiten que la cadena de texto abarque varias líneas, lo que es útil para describir funciones más complejas. La docstring se coloca inmediatamente después de la definición de la función, antes de que comience el bloque de código de la función.

```
def nombre_funcion(parametro1, parametro2, ...):
    '''
    Descripción de la función.
    Parámetros:
        parametro1 (tipo): Descripción del primer parámetro.
        parametro2 (tipo): Descripción del segundo parámetro.

    Valor de retorno:
        tipo: Descripción del valor de retorno.
    '''
    # Cuerpo de la función
    # Instrucciones que realizan una tarea específica
    return resultado
```

- **Acceso a la Docstring:** Para acceder a la docstring de una función, se puede utilizar el atributo especial __doc__ de la función. Esto devuelve la cadena de texto que representa la docstring de la función.

```
print(nombre_funcion.__doc__)
```

- **Estilos de Docstrings:** Existen varios estilos y convenciones para escribir docstrings en Python. Uno de los estilos más comunes es el estilo de docstring de Google, que se utiliza ampliamente en proyectos de código abierto. Otros estilos populares incluyen el estilo de docstring de Sphinx y el estilo de docstring de NumPy.
- **Herramientas de Generación de Documentación:** Python cuenta con diversas herramientas que permiten generar documentación automáticamente a partir de las docstrings de las funciones. Entre las herramientas más utilizadas están Sphinx y Pydoc. Estas herramientas facilitan la creación de documentación detallada y formateada en formatos como HTML, PDF o incluso páginas web.

Ejemplo de Docstring:

```
def calcular_area_triangulo(base, altura):
    '''
    Calcula el área de un triángulo dado su base y altura.

    Parámetros:
        base (float): La longitud de la base del triángulo.
        altura (float): La altura del triángulo.

    Valor de retorno:
        float: El área del triángulo.

    Ejemplo:
        >>> calcular_area_triangulo(5, 3)
        7.5
    '''
    area = 0.5 * base * altura
    return area

print(calcular_area_triangulo.__doc__)
```

6. Funciones Incluidas en Python

Dentro de Python encontraremos una diversidad de módulos que incluyen funciones muy útiles para el desarrollo de nuestros programas; A continuación, veremos los módulos y funciones más importantes. Para utilizar estas funciones, primero debes importar el módulo.

A. Módulo math

El módulo math proporciona funciones matemáticas para realizar operaciones comunes, como funciones trigonométricas, logaritmos, exponenciales, redondeo, entre otros. Es un módulo muy útil para cálculos matemáticos avanzados. import math

- **math.sqrt(x):** Devuelve la raíz cuadrada de x.

```
ejemplo_sqrt = math.sqrt(25)
print("Raíz cuadrada de 25:", ejemplo_sqrt)  # Salida: 5.0
```

- **math.pow(x, y):** Devuelve x elevado a la potencia y.

```
ejemplo_pow = math.pow(2, 3)
print("2 elevado a la potencia 3:", ejemplo_pow)  # Salida: 8.0
```

- **math.exp(x):** Devuelve la función exponencial e^x.

```
ejemplo_exp = math.exp(2)
print("e elevado a la potencia 2:", ejemplo_exp)  # Salida:
7.3890560989306495
```

- **math.log(x, base):** Devuelve el logaritmo de x en la base especificada (si no se proporciona la base, se usa la base natural, e).

```
ejemplo_log = math.log(10, 2)
print("Logaritmo base 2 de 10:", ejemplo_log)  # Salida:
3.3219280948873626
```

- **math.log10(x):** Devuelve el logaritmo base 10 de x.

```
ejemplo_log10 = math.log10(100)
print("Logaritmo base 10 de 100:", ejemplo_log10)  # Salida: 2.0
```

- **math.sin(x), math.cos(x), math.tan(x):** Funciones trigonométricas básicas (seno, coseno y tangente) en radianes.

```
angulo_radianes = math.radians(45)
ejemplo_sin = math.sin(angulo_radianes)
ejemplo_cos = math.cos(angulo_radianes)
ejemplo_tan = math.tan(angulo_radianes)
print("Seno de 45 grados:", ejemplo_sin)  # Salida:
0.7071067811865475
print("Coseno de 45 grados:", ejemplo_cos)  # Salida:
0.7071067811865476
print("Tangente de 45 grados:", ejemplo_tan)  # Salida:
0.9999999999999999
```

- **math.asin(x), math.acos(x), math.atan(x):** Funciones trigonométricas inversas (arcoseno, arcocoseno y arcotangente) en radianes.

```
ejemplo_asin = math.asin(0.5)
ejemplo_acos = math.acos(0.5)
ejemplo_atan = math.atan(1)
print("Arcoseno de 0.5:", math.degrees(ejemplo_asin))  # Salida:
30.000000000000004 (en grados)
print("Arcocoseno de 0.5:", math.degrees(ejemplo_acos))  #
Salida: 59.99999999999999 (en grados)
print("Arcotangente de 1:", math.degrees(ejemplo_atan))  #
Salida: 45.0 (en grados)
```

- **math.radians(x):** Convierte el ángulo x de grados a radianes.

```
ejemplo_radians = math.radians(180)
print("180 grados en radianes:", ejemplo_radians)  # Salida:
3.141592653589793
```

- **math.degrees(x):** Convierte el ángulo x de radianes a grados.

```
ejemplo_degrees = math.degrees(math.pi)
print("π en grados:", ejemplo_degrees)  # Salida: 180.0
```

- **math.floor(x):** Devuelve el entero más grande menor o igual que x.

```
ejemplo_floor = math.floor(3.8)
print("Parte entera de 3.8:", ejemplo_floor)  # Salida: 3
```

- **math.ceil(x):** Devuelve el entero más pequeño mayor o igual que x.

```
ejemplo_ceil = math.ceil(3.2)
print("Parte entera de 3.2:", ejemplo_ceil)   # Salida: 4
```

- **math.trunc(x):** Devuelve la parte entera de x (truncando los decimales).

```
ejemplo_trunc = math.trunc(4.9)
print("Parte entera de 4.9:", ejemplo_trunc)  # Salida: 4
```

- **math.fabs(x):** Devuelve el valor absoluto de x (su magnitud sin signo).

```
ejemplo_fabs = math.fabs(-10)
print("Valor absoluto de -10:", ejemplo_fabs)  # Salida: 10.0
```

- **math.factorial(x):** Devuelve el factorial de x.

```
ejemplo_factorial = math.factorial(5)
print("Factorial de 5:", ejemplo_factorial)  # Salida: 120
```

- **math.gcd(x, y):** Devuelve el máximo común divisor de x e y.

```
ejemplo_gcd = math.gcd(24, 36)
print("Máximo común divisor de 24 y 36:", ejemplo_gcd)  # Salida:
12
```

- **math.isqrt(n):** Devuelve la raíz cuadrada entera de n.

```
ejemplo_isqrt = math.isqrt(25)
print("Raíz cuadrada entera de 25:", ejemplo_isqrt)  # Salida: 5
```

- **math.isfinite(x):** Devuelve True si x es finito (ni infinito ni NaN).

```
ejemplo_isfinite = math.isfinite(10)
print("¿Es 10 un número finito?", ejemplo_isfinite)  # Salida:
True
```

- **math.isinf(x):** Devuelve True si x es infinito.

```
ejemplo_isinf = math.isinf(math.log(0))
print("¿Es -infinito el logaritmo de 0?", ejemplo_isinf)  #
Salida: True
```

- **math.isnan(x):** Devuelve True si x es NaN (Not a Number).

```
ejemplo_isfinite = math.isfinite(10)
print("¿Es NaN la raíz cuadrada de -1?", ejemplo_isnan)  #
Salida: True
```

- **math.pi:** Constante que representa el valor de π (pi).

```
print("El valor de π es:", math.pi)  # Salida: 3.141592653589793
```

- **math.e:** Constante que representa el número e (base del logaritmo natural).

```
print("El valor de e es:", math.e)  # Salida: 2.718281828459045
```

B. Modulo os

El módulo os proporciona funciones para interactuar con el sistema operativo, permitiendo acceder a funcionalidades del sistema de archivos,

manipulación de directorios, ejecución de comandos del sistema y más. import os

- **os.getcwd():** Devuelve el directorio de trabajo actual como una cadena de texto.

```
current_dir = os.getcwd()
print("Directorio de trabajo actual:", current_dir)
# Devuelve el directorio de trabajo actual.
```

- **os.chdir(path):** Cambia el directorio de trabajo actual al especificado en path.

```
# Cambia el directorio de trabajo actual.
print("Directorio de trabajo actual:", os.getcwd())

new_dir = "/path/to/new/directory"
os.chdir(new_dir)

print("Directorio de trabajo actual después del cambio:",
os.getcwd())
```

- **os.listdir(path='.'):** Devuelve una lista con los nombres de los archivos y directorios en el directorio dado por path. Si no se proporciona path, lista los elementos del directorio actual.

```
# Lista los archivos y directorios en el directorio dado.
path = "/path/to/directory"
files_and_directories = os.listdir(path)
print("Archivos y directorios en el directorio:",
files_and_directories)
```

- **os.mkdir(path):** Crea un directorio con el nombre especificado en path.

```
# Crea un nuevo directorio.

new_directory = "/path/to/new/directory"
os.mkdir(new_directory)
print("Directorio creado:", new_directory)
```

- **os.makedirs(path):** Crea directorios recursivamente para el path especificado.

```
# Crea directorios recursivamente.

new_directory = "/path/to/new/directory/with/subdirectories"
os.makedirs(new_directory)
print("Directorios creados:", new_directory)
```

- **os.remove(path):** Elimina un archivo con el nombre especificado en path.

```
# Elimina un archivo.

file_to_delete = "/path/to/file.txt"
os.remove(file_to_delete)
print("Archivo eliminado:", file_to_delete)
```

- **os.rmdir(path):** Elimina un directorio con el nombre especificado en path. El directorio debe estar vacío para ser eliminado

```
# Elimina un directorio vacío.

directory_to_delete = "/path/to/empty/directory"
os.rmdir(directory_to_delete)
print("Directorio eliminado:", directory_to_delete)
```

- **os.path.join(path, *paths):** Concatena varios componentes de ruta para formar una única ruta. Es útil para trabajar con rutas de archivos y directorios de manera portátil.

```
# Concatena componentes de ruta.

base_path = "/path/to/directory"
file_name = "example.txt"

full_path = os.path.join(base_path, file_name)
print("Ruta completa:", full_path)
```

- **os.path.exists(path):** Devuelve True si el path dado existe en el sistema de archivos.

```
# Verifica si una ruta existe.

path_to_check = "/path/to/file_or_directory"

if os.path.exists(path_to_check):
    print("La ruta existe:", path_to_check)
else:
    print("La ruta no existe:", path_to_check)
```

- **os.path.isdir(path):** Devuelve True si el path dado es un directorio.

```
# Verifica si una ruta es un directorio.

path_to_check = "/path/to/directory"

if os.path.isdir(path_to_check):
    print("Es un directorio:", path_to_check)
else:
    print("No es un directorio:", path_to_check)
```

C. Modulo random

El módulo random permite generar números pseudoaleatorios y realizar selecciones aleatorias de listas o secuencias. import random

- **random():** Esta función devuelve un número decimal de punto flotante aleatorio entre 0 y 1 (incluido 0 pero excluido 1).

```
# Generar un número decimal aleatorio entre 0 y 1
random_number = random.random()
print(random_number)
```

- **randrange(start, stop[, step]):** Esta función devuelve un número entero aleatorio dentro del rango especificado. Se puede proporcionar un paso opcional para especificar el incremento.

```
# Generar un número entero aleatorio en el rango de 10 a 50
(excluyendo 50)
random_integer = random.randrange(10, 50)
print(random_integer)
```

- **randint(a, b):** Esta función devuelve un número entero aleatorio entre a y b (ambos inclusive).

```
# Generar un número entero aleatorio entre 1 y 100 (ambos
inclusive)
random_integer = random.randint(1, 100)
print(random_integer)
```

- **choice(seq):** Esta función devuelve un elemento aleatorio de la secuencia seq, que puede ser una lista, tupla o cadena.

```
# Elegir un elemento aleatorio de una lista
fruits = ['apple', 'banana', 'orange', 'grape', 'watermelon']
random_fruit = random.choice(fruits)
print(random_fruit)
```

- **shuffle(seq):** Esta función mezcla los elementos de la secuencia seq de forma aleatoria. Modifica la secuencia original in-place.

```
# Mezclar una lista de números
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
random.shuffle(numbers)
print(numbers)
```

- **sample(population, k):** Esta función devuelve una muestra aleatoria de longitud k de la población proporcionada sin reemplazo. Es decir, los elementos en la muestra no se repiten.

```
# Obtener una muestra aleatoria de 3 elementos sin reemplazo
population = ['red', 'green', 'blue', 'yellow', 'orange',
'purple']
random_sample = random.sample(population, 3)
```

```
print(random_sample)
```

- **random.seed():** Función para establecer una semilla para el generador de números aleatorios.

```
# Inicializar el generador con una semilla específica (42 en este
caso)
random.seed(42)

# Generar números aleatorios con la semilla establecida
for _ in range(5):
    print(random.random())
```

D. Modulo sys

El módulo sys proporciona funcionalidades y variables específicas del intérprete de Python. Permite interactuar con el intérprete y acceder a sus propiedades. import sys

- **sys.argv:** Lista que contiene los argumentos de línea de comandos pasados al script Python. El primer elemento de la lista (sys.argv[0]) es el nombre del script en sí.

```
def main():
    print("Argumentos de línea de comandos:", sys.argv)

if __name__ == "__main__":
    main()
```

- **sys.executable:** Ruta al intérprete de Python utilizado para ejecutar el script actual.

```
print("Ruta del intérprete de Python:", sys.executable)
```

- **sys.exit([arg]):** Finaliza la ejecución del programa de forma inmediata. Se puede proporcionar un valor opcional como argumento para devolver un código de salida al sistema operativo.

```
def main():
    print("Ejecución antes de sys.exit()")
    sys.exit(1)
    print("Esta línea no se imprimirá debido a sys.exit()")

if __name__ == "__main__":
    main()
```

- **sys.getdefaultencoding():** Devuelve la codificación predeterminada utilizada por el intérprete.

```
print("Codificación predeterminada:", sys.getdefaultencoding())
```

- **sys.getsizeof(object[, default]):** Devuelve el tamaño en bytes del objeto especificado. Puede ser útil para hacer un seguimiento del uso de memoria.

```
data = [1, 2, 3, 4, 5]
print("Tamaño de la lista 'data' en bytes:", sys.getsizeof(data))
```

- **sys.modules:** Un diccionario que contiene todos los módulos cargados actualmente. Las claves son los nombres de los módulos y los valores son los objetos de módulo.

```
print("Módulos cargados actualmente:", sys.modules.keys())
```

- **sys.path:** Lista que contiene las rutas de búsqueda para los módulos importados. Es una lista de cadenas.

```
print("Rutas de búsqueda para módulos importados:", sys.path)
```

- **sys.platform:** Cadena que identifica la plataforma en la que se está ejecutando Python (por ejemplo, "win32", "linux2", "darwin", etc.).

```
print("Plataforma en la que se está ejecutando Python:",
sys.platform)
```

```

```

- **sys.stdin, sys.stdout, sys.stderr:** Flujos de entrada estándar, salida estándar y error estándar, respectivamente.

```
sys.stdout.write("Este es un mensaje en la salida estándar.\n")
sys.stderr.write("Este es un mensaje en el error estándar.\n")
```

Para el caso de sys.stdin ejeuctamos el siguiente codigo en un script .py; esto es debido a que el comando lee lo que el usuario ingresa en terminal; similar a input()

```
import sys

def main():
    print("Ingrese un número:")
    input_number = sys.stdin.readline().strip()

    try:
        number = int(input_number)
        result = number * 2
        print("El doble del número ingresado es:", result)
    except ValueError:
        print("Error: Por favor, ingrese un número válido.")

if __name__ == "__main__":
    main()
```

- **sys.version:** Cadena que muestra la versión completa del intérprete de Python.

```
print("Versión completa del intérprete de Python:", sys.version)
```

- **sys.version_info:** Tupla que contiene la información de la versión de Python, como sys.version_info.major, sys.version_info.minor, etc.

```
print("Versión de Python:", sys.version_info)
print("Versión mayor:", sys.version_info.major)
print("Versión menor:", sys.version_info.minor)
```

```
```

- **sys.warnoptions:** Lista de advertencias que se controlan mediante la opción -W en la línea de comandos.

```
print("Opciones de advertencias:", sys.warnoptions)
```