INFORMATORIO

INTRODUCCIÓN

Llegamos a una parte intermedia en el curso donde ya vas a comenzar a introducirte en una parte más avanzada de Python (y de la mayoría de los lenguajes de programación). En este apartado vamos a comenzar a unir aún más los conceptos aprendidos hasta aquí, por lo que invitamos a comenzar con…

Las funciones o def en Python, son bloques de código reutilizables que realizan una tarea específica. Son una parte fundamental de la programación, ya que nos permiten dividir nuestro código en partes más pequeñas y manejables. Además, las funciones nos ayudan a escribir un código más limpio, legible y organizado.

Son un fragmento de código con un nombre asociado que realiza una serie de tareas y devuelve un valor. A los fragmentos de código que tienen un nombre asociado y no devuelven valores se les suele llamar procedimientos.

En realidad, en Python no existen los procedimientos, ya que **cuando el programador no especifica un valor de retorno, la función devuelve el valor None** (nada).

El uso de funciones es un componente muy importante para el paradigma de la programación estructurada, y tiene varias ventajas:

- **Modularización:** permite segmentar un programa complejo en una serie de partes o módulos más simples, facilitando así la programación y el depurado.
- **Reutilización:** permite reutilizar una misma función en distintos programas (algo que mencionamos en apuntes anteriores cuando describimos las diferencias entre método y función).

Como ves, el uso de las funciones es muy importante, y desde el principio del curso las venimos viendo, sin embargo, en este apartado vamos a ver su funcionamiento interno y como crear nuestras propias funciones!!!

Sentencia def – Definiendo una función

Para crear una función en Python, utilizamos las siguientes reglas:

1. El bloque de función de palabras clave "def" en el principio, seguido por el nombre de la función y los identificadores entre paréntesis ():

```
1  def saludar(nombre):
2      print("Hola,", nombre)
3
4  saludar("Info")
5
```

```
PROBLEMAS    SALIDA    CONSOLA DE DEPURACIÓN

Hola, Info
```

En este ejemplo, se define la función *saludar* que recibe un parámetro *nombre*. La función imprime un saludo utilizando el valor del parámetro.

2. Cualquiera de los **parámetros** y **argumentos** entrantes deben ser colocados entre paréntesis en el medio. Se puede utilizar para definir los parámetros entre paréntesis:

```
1  def suma(a, b):
2      resultado = a + b
3      return resultado
4
5  resultado_suma = suma(3, 4)
6  print(resultado_suma)
7
```

```
PROBLEMAS    SALIDA    CONSOLA DE DEPURACIÓN

7
```

En este caso, la función *suma* tiene dos parámetros: *a* y *b*. La función realiza la suma de estos dos parámetros y devuelve el resultado.

La diferencia entre **parámetro** y **argumento** radica en su contexto y uso en la programación.

Parámetro: Es una variable que se define en la declaración de una función. Representa un valor que se espera recibir cuando se llama a la función. Los parámetros son como marcadores de posición para los valores que serán pasados a la función. Son utilizados dentro del cuerpo de la función para realizar operaciones y cálculos.

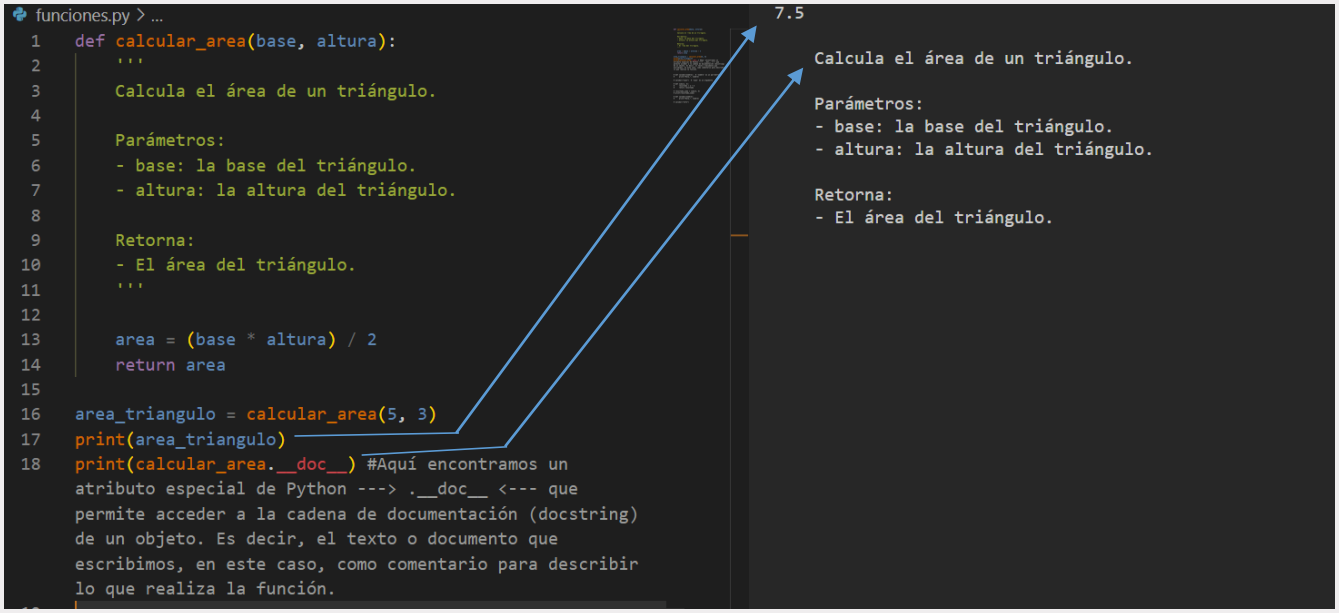
Argumento: Es el valor real que se pasa a una función cuando se llama a dicha función. Los argumentos son los valores concretos que se asignan a los parámetros de una función durante la llamada. Estos valores pueden ser constantes, variables, expresiones o incluso otras funciones. Los argumentos se utilizan para proporcionar datos específicos con los que la función trabajará.

Ejemplo:

```python
def saludar(nombre):  # 'nombre' es un parámetro
    print("Hola,", nombre)

saludar("Juan")  # 'Juan' es un argumento
```

3. En la primera línea de la sentencia de la función se puede utilizar opcionalmente un texto para describir la función:



En este ejemplo, la función *calcular_area* calcula el área de un triángulo utilizando la fórmula (base * altura) / 2.
La cadena de documentación (.__doc__)describe los parámetros y el valor de retorno de la función.

4. El contenido de la función se coloca después de dos puntos (tal cual como lo veníamos viendo en las estructuras de control de flujo). No olvides que siempre debemos indentar para que funcione todo correctamente.

```python
def contar_hasta_cinco():
    for i in range(1, 6):
        print(i)

contar_hasta_cinco()
```

```
PROBLEMAS   SALIDA   CONSOLA DE DEPURACIÓ

1
2
3
4
5
```

En este caso, la función *contar_hasta_cinco* utiliza un bucle *for* para imprimir los números del 1 al 5 en la consola.

5. Al final de una función, la palabra clave "*return*" se utiliza de manera opcional para devolver un valor al código que llamó a la función. Si se utiliza "*return*" seguido de una expresión, dicha expresión será el valor devuelto. Si no se especifica ninguna expresión después de "*return*", se devolverá automáticamente el valor especial "*None*", que indica la ausencia de un valor significativo.

```python
def obtener_promedio(lista):
    if len(lista) == 0:
        return None

    total = sum(lista)
    promedio = total / len(lista)
    return promedio

numeros = [4, 6, 8, 10]
promedio_numeros = obtener_promedio(numeros)
print(promedio_numeros)

vacio = []
promedio_vacio = obtener_promedio(vacio)
print(promedio_vacio)
```

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL

7.0
None
```

En este ejemplo, la función *obtener_promedio* calcula el promedio de una lista de números. Si la lista está vacía, la función devuelve None. Si la lista tiene elementos, se calcula y devuelve el promedio.

En esta serie de reglas que debe seguir una función, también te dimos ejemplos de uso, por lo que ya podés ir comprendiendo como se utilizan y el potencial que tienen las funciones.

Ahora vamos a ver una serie de características de los parámetros y argumentos:

PARÁMETROS y ARGUMENTOS

Ya te dimos la definición de parámetros y argumentos, por lo que vamos a brindarte características de estos.

- **Parámetros por posición:** Los parámetros por posición se refieren a la forma en que se pasan los argumentos a una función en Python en función de su posición relativa en la lista de argumentos.
  Cuando se utilizan parámetros por posición, los argumentos se pasan a la función en el mismo orden en el que se definen los parámetros en la declaración de la función. Esto significa que el

primer argumento se asigna al primer parámetro, el segundo argumento al segundo parámetro, y así sucesivamente.

```python
def calcular_precio_total(precio_unitario, cantidad):
    precio_total = precio_unitario * cantidad
    return precio_total

precio_final = calcular_precio_total(10, 5)
print(f'El precio final es: {precio_final}')
```

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL

El precio final es: 50
```

En este ejemplo, la función *calcular_precio_total* recibe dos parámetros: *precio_unitario* y *cantidad*. Cuando llamamos a la función *calcular_precio_total(10, 5)*, el valor 10 se asigna al parámetro *precio_unitario* y el valor 5 se asigna al parámetro *cantidad*. Luego, la función realiza el cálculo del precio total multiplicando el *precio unitario* por la *cantidad* y devuelve el *resultado*.
En este caso, el precio unitario es 10 y la cantidad es 5, por lo tanto, el resultado que se muestra en la consola será 50, que es el precio total calculado.
El orden de los argumentos es muy importante. Si cambiás el orden de los argumentos al llamar a la función, el resultado puede ser diferente. Por ejemplo, *calcular_precio_total(5, 10)* daría como resultado 100, ya que el precio unitario sería 5 y la cantidad sería 10.

- **Parámetros por nombre:** Los parámetros por nombre, también conocidos como argumentos con nombre, permiten pasar los argumentos a una función en Python especificando el nombre del parámetro al que se asigna cada argumento. Esto brinda mayor flexibilidad al llamar a la función, ya que no es necesario seguir un orden específico.
  Cuando se utilizan parámetros por nombre, se proporciona el nombre del parámetro seguido de un signo igual (=) y el valor correspondiente como argumento. Esto permite asociar cada argumento con el parámetro correcto, independientemente de su posición relativa.

```python
def saludar(nombre, edad):
    print("Hola,", nombre, "tenés", edad, "años.")

saludar(nombre="Juan", edad=25)
```

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL

Hola, Juan tenés 25 años.
```

En este caso, la función saludar tiene dos parámetros: *nombre* y *edad*. Al llamar a la función saludar(nombre="Juan", edad=25), los argumentos se pasan utilizando los nombres de los parámetros. Esto significa que el valor "Juan" se asigna al parámetro nombre y el valor 25 se asigna al parámetro edad. Luego, la función imprime un saludo personalizado utilizando los valores de los parámetros.
La ventaja de los parámetros por nombre es que podés especificar los argumentos en el orden que deseás y no estás limitado por la posición de los parámetros en la declaración de la función. Esto puede hacer que el código sea más claro y legible, especialmente cuando se trabaja con funciones que tienen muchos parámetros o cuando solo se desean proporcionar valores para algunos parámetros específicos.

Es importante tener en cuenta que al utilizar parámetros por nombre, es necesario asegurarse de usar los nombres de los parámetros correctamente y que coincidan con los definidos en la función. Además, los argumentos con nombre pueden combinarse con argumentos posicionales si es necesario.

- **Llamada sin argumentos / Parámetros por defecto:** La llamada sin argumentos y parámetros por defecto se refieren a la forma de llamar a una función en Python sin proporcionar ningún argumento. Esto significa que no se pasan valores específicos para los parámetros de la función al llamarla.
  Cuando se realiza una llamada sin argumentos, la función utilizará los valores predeterminados, o por defecto, definidos en la declaración de los parámetros de la función, si los hay. Los valores predeterminados son valores asignados a los parámetros en la definición de la función, lo que permite que la función se ejecute correctamente incluso si no se proporciona un argumento específico durante la llamada.
  Estos parámetros por defecto pueden ser de cualquier tipo (str, int, float, bool).

```
1  def saludar(nombre="Usuario"):
2      print("Hola,", nombre)
3
4  saludar()  # Llamada sin argumentos
5  saludar("Juan")  # Llamada con argumento
```

PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL

```
Hola, Usuario
Hola, Juan
```

En este caso, la función *saludar* tiene un parámetro llamado *nombre* con un valor predeterminado de "Usuario". Cuando llamamos a la función *saludar()* sin proporcionar ningún argumento, se utilizará el valor predeterminado, o parámetro por defecto, "Usuario". La función imprimirá "Hola, Usuario" en la consola.
Caso contrario, si especificamos un argumento, entonces al llamar a la función se pasará el valor del argumento. Como en el segundo caso donde se pasa un nombre y la función muestra el mensaje con el valor que pasamos como argumento.
La llamada sin argumentos puede ser útil cuando deseás que la función tenga un comportamiento predefinido y no requiere ningún valor específico en ese momento. Los valores predeterminados, o por defecto, permiten que la función sea más flexible y se ejecute correctamente incluso si no se proporcionan argumentos durante la llamada.
Es importante mencionar que no todas las funciones tienen valores predeterminados para todos los parámetros. Algunas funciones pueden requerir argumentos obligatorios y lanzarán un error si se intenta llamarlas sin proporcionar los argumentos adecuados. Esto depende de cómo se haya definido la función y qué requisitos tenga en cuanto a los argumentos necesarios.

- **Llamada sin argumentos – caso de error:** Como mencionábamos en el ejemplo anterior, si la función no tiene un argumento para el parámetro de forma predeterminada, esta va a lanzar un error si al llamar a la función no se incluye un argumento.

```
1  def saludar(nombre): # parámetro sin argumento de forma predeterminada
2      print("Hola,", nombre)
3
4  saludar()  # Llamada sin argumentos
```

```
PROBLEMAS   SALIDA   CONSOLA DE DEPURACIÓN   TERMINAL

Traceback (most recent call last):
  File "c:\Users\Pc\Desktop\Info2023\funciones.py", line 4, in <module>
    saludar()  # Llamada sin argumentos
    ^^^^^^^^^
TypeError: saludar() missing 1 required positional argument: 'nombre'
```

En el error nos dice que se requiere un argumento posicional para el parámetro "nombre", por lo que debemos cuidar esos detalles al momento de llamar a la función.

- **Argumentos indeterminados:** También conocidos como argumentos variables, son una característica en Python que permite definir funciones que pueden recibir un número variable de argumentos durante su llamada. Esto brinda flexibilidad al trabajar con funciones que pueden necesitar manejar diferentes cantidades de argumentos.
  Python admite dos tipos de argumentos indeterminados: argumentos indeterminados posicionales y argumentos indeterminados de palabras clave o nombre.

  - **Argumentos indeterminados posicionales:** Los argumentos indeterminados posicionales permiten que una función reciba un número variable de argumentos posicionales, los cuales se agrupan en forma de tupla dentro de la función. Para definir este tipo de argumentos, se utiliza el asterisco (*) antes del nombre del parámetro.

```
1  def por_posicion(*args):
2      for arg in args:
3          print(arg)
4
5  por_posicion('Hola', 'Mundo', [1, 2, 3])
```

```
PROBLEMAS   SALIDA   CONSOLA DE DEPURACIÓN   TERMINAL

Hola
Mundo
[1, 2, 3]
```

En este ejemplo podemos ver como la cantidad de argumentos que pasamos es variable, no se limita al momento de definir la función gracias al asterisco antes del nombre del parámetro. Este nombre de parámetro "args" es una convención usada, no necesariamente debe ir ese nombre como parámetro, puede ser cualquier, como venimos usando, pero solo por convención lo usamos de esa

manera. Luego dentro del ciclo *for* usamos "arg" como nombre de la variable de iteración, también por convención. Podés usar cualquier nombre tanto para el parámetro, como para la variable de iteración.

- **Argumentos indeterminados de palabras clave o nombre:** Los argumentos indeterminados de palabras clave permiten que una función reciba un número variable de argumentos con nombres específicos, los cuales se agrupan en forma de **diccionario** dentro de la función.
  Para definir este tipo de argumentos, se utiliza el doble asterisco (**) antes del nombre del parámetro.

```
1   def por_nombre(**kwargs):
2       for clave, valor in kwargs.items():
3           print(f"{clave} : {valor}")
4
5   por_nombre(nombre="Alumno", edad=20, ciudad="Argentina")
```

```
PROBLEMAS   SALIDA   CONSOLA DE DEPURACIÓN   TERMINAL

nombre : Alumno
edad : 20
ciudad : Argentina
```

En este caso, la función *por_nombre* puede recibir cualquier cantidad de argumentos con nombres clave - valor. Los argumentos se agrupan en un diccionario llamado "kwargs" (de igual forma que *args*, el nombre *kwargs* es una convención. Podés usar cualquier nombre), y luego se itera sobre el diccionario para mostrar cada clave y valor.

También aclaramos que no es totalmente necesario el uso del ciclo *for,* ya que se puede hacer sin él. En los ejemplos anteriores los usamos para que visiblemente sea más ordenado, pero te dejamos los ejemplos sin el uso de este ciclo dentro de la función.

```
1   def por_posicion(*args):
2       print(args)
3
4   por_posicion('Hola', 'Mundo', [1, 2, 3])
```

```
PROBLEMAS   SALIDA   CONSOLA DE DEPURACIÓN   TERMINAL

('Hola', 'Mundo', [1, 2, 3])
```

```
1   def por_nombre(**kwargs):
2       print(kwargs)
3
4   por_nombre(nombre="Alumno", edad=20, ciudad="Argentina")
```

```
PROBLEMAS   SALIDA   CONSOLA DE DEPURACIÓN   TERMINAL

{'nombre': 'Alumno', 'edad': 20, 'ciudad': 'Argentina'}
```

Los argumentos indeterminados proporcionan una forma conveniente de manejar una cantidad variable de argumentos en una función, lo que permite que sea más flexible y adaptable a diferentes situaciones. Puedes utilizar argumentos indeterminados posicionales, argumentos indeterminados de palabras clave o incluso ambos en una misma función, según tus necesidades. Así que cómo último ejemplo te dejamos por posición y nombre.

- **Argumentos por posición y nombre:**

```python
def por_posicion_y_nombre(*args, **kwargs):
    for arg in args:
        print(f'Argumento posicional: {arg}')

    for clave, valor in kwargs.items():
        print(f'Argumento por nombre: {clave}, Valor: {valor}')

por_posicion_y_nombre('Alumno', 25, ciudad='Argentina', profesion='Ingeniero')
```

```
PROBLEMAS   SALIDA   CONSOLA DE DEPURACIÓN   TERMINAL

Argumento posicional: Alumno
Argumento posicional: 25
Argumento por nombre: ciudad, Valor: Argentina
Argumento por nombre: profesion, Valor: Ingeniero
```

En este caso, la función *por_posicion_y_nombre* utiliza argumentos indeterminados tanto por posición (*args) como por nombre (**kwargs).

Al llamar a la función *por_posicion_y_nombre* ("Alumno", 25, ciudad="Argentina", profesion="Ingeniero"), el primer argumento "Alumno" se asigna al primer parámetro por posición (*args), el segundo argumento 25 se asigna al segundo parámetro por posición (*args), y los argumentos restantes "ciudad=Argentina" y "profesion=Ingeniero" se asignan a los parámetros por nombre (**kwargs).

Dentro de la función, se itera sobre los argumentos por posición (args) y se imprime cada uno de ellos. Luego, se itera sobre los argumentos por nombre (kwargs) y se muestra tanto la clave como el valor correspondientes.

Y como plus te dejamos otro ejemplo más para que puedas seguir viendo el potencial:

```python
def super_calculadora(*args, **kwargs):
    resultado = 0
    for arg in args:
        resultado += arg
    print(f'La suma de los números da como resultado: {resultado}')

    for clave, valor in kwargs.items():
        print(f'Muchas gracias por usar esta calculadora: {clave}, {valor}')

super_calculadora(30, 25, 10, 58, 1900, Alumno = 'Info')
```

```
PROBLEMAS   SALIDA   CONSOLA DE DEPURACIÓN   TERMINAL

La suma de los números da como resultado: 2023
Muchas gracias por usar esta calculadora: Alumno, Info
```

En conclusión, los argumentos indeterminados en Python proporcionan flexibilidad y versatilidad al permitir que una función reciba un número variable de argumentos durante su llamada. Estos argumentos pueden ser agrupados como argumentos indeterminados por posición (*args) o argumentos indeterminados por nombre (**kwargs).

Los **argumentos indeterminados por posición** permiten manejar una cantidad variable de argumentos posicionales, que son agrupados en una tupla dentro de la función. Esto permite pasar múltiples valores sin la necesidad de definir una cantidad específica de parámetros.

Por otro lado, los **argumentos indeterminados por nombre** permiten manejar una cantidad variable de argumentos con nombres clave-valor, que son agrupados en un diccionario dentro de la función. Esto brinda la posibilidad de especificar argumentos de manera más explícita y facilita el paso de múltiples configuraciones.

La combinación de argumentos indeterminados *por posición y por nombre* en una misma función permite una mayor flexibilidad y adaptabilidad en el diseño de funciones, ya que pueden manejar diferentes escenarios y tipos de argumentos de manera dinámica.

Al utilizar argumentos indeterminados, es importante tener en cuenta cuál es la mejor opción según el contexto y los requisitos específicos de la función. Esto permite escribir código más legible, mantenible y flexible, adaptándose a diferentes situaciones sin necesidad de redefinir la función para cada caso.

En resumen, los argumentos indeterminados son una herramienta poderosa en Python que permite crear funciones más versátiles y adaptables, capaces de recibir una cantidad variable de argumentos, ya sea por posición o por nombre, brindando mayor flexibilidad en el desarrollo de aplicaciones y facilitando la reutilización del código.

Y ahora seguimos con *módulos* →

INFORMATORIO

MÓDULOS

En Python, los módulos son archivos que contienen definiciones de variables, funciones y clases que pueden ser utilizadas en otros programas. Los módulos permiten organizar y reutilizar código de manera efectiva, lo que facilita el desarrollo de aplicaciones más grandes y complejas.

Un módulo puede contener cualquier cantidad de declaraciones y definiciones de código, como variables, funciones, clases y otras estructuras de datos. Estas definiciones son accesibles desde otros programas mediante la **importación** del módulo correspondiente.

Los módulos en Python tienen varias ventajas:

- **Modularidad**: Los módulos ayudan a organizar y estructurar el código en unidades lógicas y coherentes. Esto facilita la comprensión, el mantenimiento y la reutilización del código.
- **Reutilización de código**: Los módulos permiten que el código sea compartido y reutilizado en diferentes programas. Esto evita tener que escribir el mismo código una y otra vez, lo que ahorra tiempo y reduce errores.
- **Encapsulación**: Los módulos proporcionan un nivel de encapsulación, lo que significa que las definiciones dentro de un módulo están encapsuladas y no interfieren con el código en otros módulos. Esto ayuda a evitar conflictos de nombres y promueve una mejor organización del código.
- **Espacio de nombres**: Los módulos proporcionan un espacio de nombres separado para las definiciones que contienen. Esto evita colisiones de nombres entre diferentes partes de un programa y ayuda a mantener el código ordenado y legible.

Para utilizar un módulo en un programa Python, primero debés importarlo. Esto se hace utilizando la instrucción `import`, seguida del nombre del módulo. Una vez importado, podés acceder a las definiciones contenidas en el módulo utilizando el nombre del módulo seguido de un punto y el nombre de la definición.

Por ejemplo, si tenés un módulo llamado "mimodulo.py" que contiene una función llamada "saludar()", podés importar y usar esa función de la siguiente manera:

INFORMATORIO

```python
import mimodulo

mimodulo.saludar()

```

Entonces podemos decir que los módulos en Python son archivos que contienen definiciones de código reutilizables. Permiten organizar el código de manera modular, facilitan la reutilización del código y proporcionan un espacio de nombres separado para evitar colisiones de nombres. Los módulos son una parte fundamental de la estructura y la filosofía de Python como lenguaje de programación.

Algunos ejemplos de los módulos más populares en Python:

- **Módulo math**:
  El módulo **math** proporciona funciones y constantes matemáticas. Permite realizar operaciones matemáticas avanzadas, como cálculos trigonométricos, exponenciales, logarítmicos y más.

```python
import math

print(math.sqrt(25))  # Raíz cuadrada de 25
print(math.sin(math.pi/2))  # Seno de pi/2 (90 grados)
print(math.log10(100))  # Logaritmo base 10 de 100
```

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL

5.0
1.0
2.0
```

- **Módulo random**:
  El módulo **random** proporciona funciones para generar números aleatorios y realizar operaciones relacionadas con la aleatoriedad.

```python
import random

print(random.randint(1, 10))  # Generar un número entero aleatorio entre 1 y 10
print(random.choice(['rojo', 'verde', 'azul']))  # Seleccionar un elemento aleatorio de una lista
```

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL

2
verde
PS C:\Users\Pc\Desktop\Info2023> & "C:/Program Files/Python311/python.exe" c:/Users/Pc/Desktop/Info2023/modu
4
rojo
PS C:\Users\Pc\Desktop\Info2023> & "C:/Program Files/Python311/python.exe" c:/Users/Pc/Desktop/Info2023/modu
6
verde
```

- **Módulo datetime**:
  El módulo **datetime** proporciona clases y funciones para trabajar con fechas y tiempos.

```python
import datetime

fecha_actual = datetime.date.today()  # Obtener la fecha actual
print(fecha_actual)

tiempo_actual = datetime.datetime.now()  # Obtener la fecha y hora actual
print(tiempo_actual)
```

- **Módulo os**:
  El módulo os proporciona funciones relacionadas con la interacción con el sistema operativo, como acceder a rutas de archivos, crear y eliminar directorios, etc.

```python
import os

print(os.getcwd())  # Obtener el directorio de trabajo actual
os.mkdir('nuevo_directorio')  # Crear un nuevo directorio
```

- **Módulo csv**:
  El módulo csv proporciona funciones para leer y escribir archivos CSV (Comma Separated Values), que son comunes en el intercambio de datos estructurados.

```python
import csv

with open('datos.csv', 'r') as archivo:
    lector_csv = csv.reader(archivo)
    for fila in lector_csv:
        print(fila)
```

Estos son solo algunos ejemplos de módulos en Python. Hay una amplia variedad de módulos disponibles para diferentes propósitos, como manipulación de archivos, acceso a bases de datos, generación de gráficos, procesamiento de imágenes, entre otros. Además, también podés crear tus propios módulos para organizar y reutilizar tu propio código en diferentes proyectos.

No nos vamos a enfocar en mostrar todos estos módulos, sin embargo, te los mostramos a modo de ejemplo para que los puedas investigar si tenés curiosidad por saber más de ellos.

La recomendación de siempre es investigar más allá de los apuntes o las clases que te ofrecemos.