> Funciones

Funciones DEF

SEMANA 4

INTRODUCCIÓN

Hemos alcanzado una sección intermedia del curso, donde comenzarás a adentrarte en temas más complejos de Python (así como en la mayoría de los lenguajes de programación). En esta sección, integramos los conceptos previamente aprendidos para dar paso a...

FUNCIONES - DEF

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).

El uso de funciones es fundamental, y hemos estado explorando su implementación desde el inicio del curso. No obstante, en este apartado profundizaremos en su funcionamiento interno y en cómo 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(f'Hola, {nombre}')
3
4   saludar('Info')
```

```
PROBLEMAS    SALIDA    CONSOLA DE DEPURACIÓ
Hola, Info
```

En este ejemplo, se define la función "saludar()" que recibe un parámetro "nombre". La función muestra por pantalla un saludo utilizando el valor del parámetro.

2 – Los parámetros y argumentos entrantes deben estar contenidos entre paréntesis. Para definir los parámetros, se pueden también colocar 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)
```

```
PROBLEMAS    SALIDA    CONSOLA DE DEPURAC
7
```

En este caso, la función "suma" se define con dos parámetros: "a" y "b". La función se encarga de sumar ambos argumentos y retorna el resultado.
La palabra clave "return" es utilizada para devolver el resultado de la función.

La distinción entre parámetro y argumento en programación se encuentra en su contexto y uso.

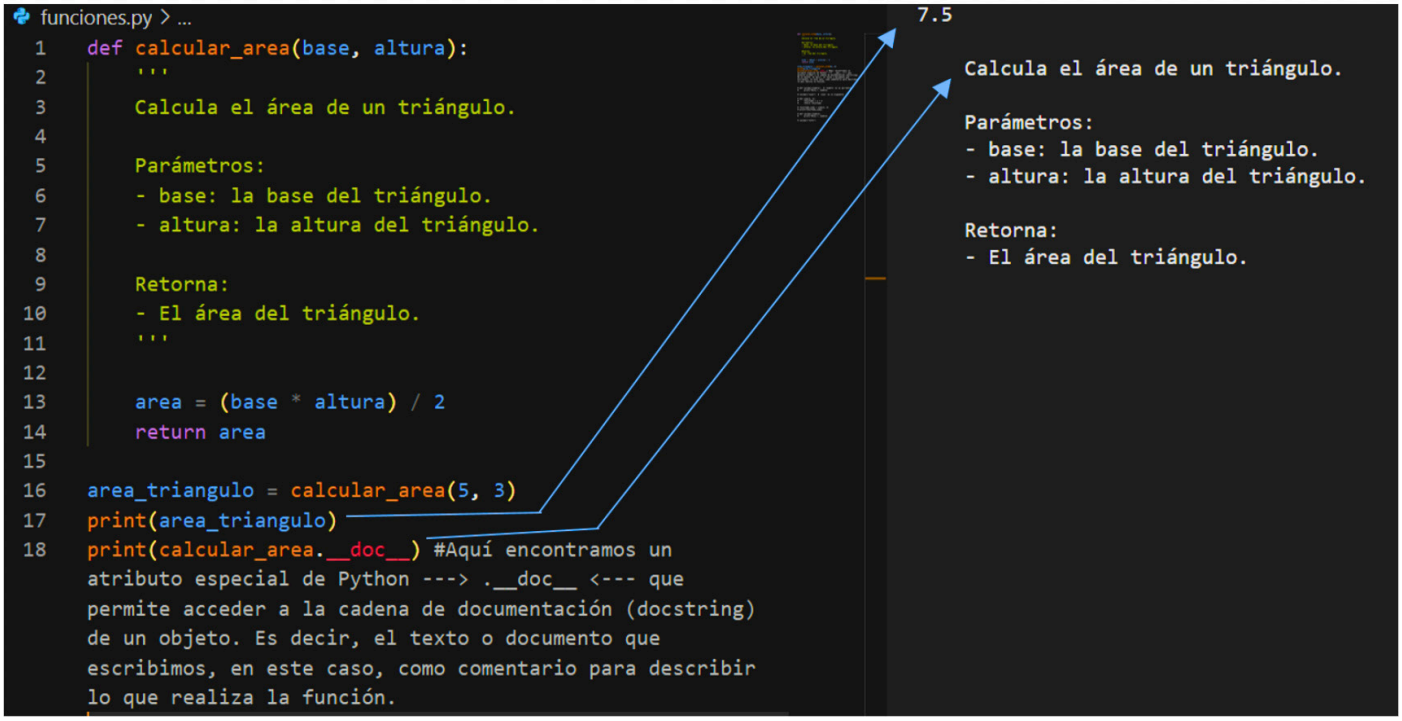
Parámetro: es una variable que se define en la declaración de una función. Su finalidad es representar un valor que se espera recibir cuando se llama a la función. Los parámetros cumplen la función de ser marcadores de posición para los valores que se pasarán a la función y se utilizan dentro del cuerpo de la función para realizar operaciones y cálculos.

Argumento: es el valor real que se pasa a una función cuando se la llama. 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 como datos específicos con los que la función trabajará.

Ej:

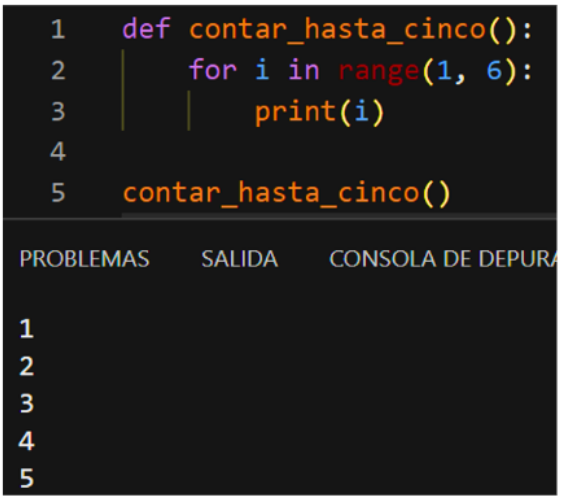
```
1   def saludar(nombre): #"nombre" es un parámetro
2       print(f'Hola, {nombre}')
3
4   saludar('Juan') #"Juan" es un argumento
```

3 - En la primera línea de la función, se puede optar por incluir un texto descriptivo para contextualizar su finalidad.



*El ejemplo muestra la función "calcular_area" que permite calcular el área de un triángulo utilizando la fórmula (base * altura) / 2.*
La cadena de documentación (.__doc__) describe los diferentes parámetros de la función y el valor que se espera como resultado.

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.



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 el ejemplo, la función "obtener_promedio" se encarga de calcular el promedio de una lista de números. En caso de que la lista esté vacía, la función devolverá "None".
Sin embargo, si existen elementos en la lista, la función calculará y devolverá el promedio correspondiente.

En esta serie de reglas para una función, se han proporcionado ejemplos de uso, permitiéndote entender mejor su aplicación y potencial.

PARÁMETROS y ARGUMENTOS

Hemos proporcionado la definición de los términos "parámetros" y "argumentos". Ahora, procederemos a detallar sus características.

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 Ej.: "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 la asignación de argumentos a una función en Python mediante la especificación del nombre del parámetro correspondiente. Esta técnica provee una mayor flexibilidad en la llamada a la función, ya que no se requiere seguir un orden específico.

Al utilizar parámetros por nombre, se provee el nombre del parámetro seguido de un signo igual (=) y el valor correspondiente como argumento, lo que permite asociar cada argumento con el parámetro correcto independientemente de su posición relativa.

```
1  def saludar(nombre, edad):
2      print(f'Hola {nombre}, tenés {edad} años.')
3
4  saludar(nombre = 'Juan', edad = 25)
```

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  PUERTOS  TERMINA
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 a una función en Python sin proporcionar argumentos se considera una llamada "sin argumentos" o "por defecto". En otras palabras, se ejecuta la función sin pasar valores específicos a sus parámetros. Si la función tiene valores predeterminados para sus parámetros, estos valores se utilizarán durante la llamada.
Los valores predeterminados son valores asignados a los parámetros en la definición de la función. Estos valores permiten que la función se ejecute correctamente incluso si no se proporciona un argumento específico durante la llamada. Los parámetros por defecto

pueden ser de cualquier tipo (str, int, float, bool).

```
1  def saludar(nombre = 'Usuario'):
2      print(f'Hola {nombre}.')
3
4  saludar() #llamada sin argumentos
5  saludar('Juan') #llamada con argumento
```

```
PROBLEMAS   SALIDA   CONSOLA DE DEPURACIÓN   PUERTOS
Hola Usuario.
Hola Juan.
```

Para este caso, la función "saludar" posee un parámetro denominado "nombre" que cuenta con un valor predeterminado establecido en "Usuario". Si ejecutamos la función "saludar()" sin proveer ningún argumento, se utilizará el valor predeterminado "Usuario", y se imprimirá "Hola Usuario" en la consola.

No obstante, si proporcionamos un argumento, este se pasará a la función al momento de la llamada, tal como sucede en el segundo ejemplo, donde el nombre especificado como argumento se mostrará en el mensaje mostrado por la función.

La llamada sin argumentos puede resultar útil cuando se desea que la función tenga un comportamiento preestablecido y no requiera de valores específicos en ese momento. Los valores predeterminados permiten que la función sea más versátil y se ejecute de manera correcta incluso si no se proporcionan argumentos durante la llamada.

Es importante destacar que no todas las funciones contienen valores predeterminados para todos los parámetros. Algunas funciones pueden requerir argumentos obligatorios y, en caso de que se intente llamarlas sin proporcionar los argumentos requeridos, se lanzará un error. Esto dependerá de cómo se haya definido la función y de los requisitos específicos que 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(f'Hola {nombre}.')
3
4  saludar() #llamada sin argumentos
```

```
PROBLEMAS   SALIDA   CONSOLA DE DEPURACIÓN   PUERTOS   TERMINAL
Traceback (most recent call last):
  File "c:\Users\Pc\Desktop\Info 2023 B\Practica\def.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 el ejemplo, podemos ver como la cantidad de argumentos que pasamos es indefinida, 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 cualquiera, como venimos haciendo, 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. Se puede 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
```

Aquí, 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.

Para estos casos, el uso del ciclo for no es totalmente necesario, 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])
```

```python
def por_nombre(**kwargs):
    print(kwargs)

por_nombre(nombre="Alumno", edad=20, ciudad="Argentina")
```

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL
{'nombre': 'Alumno', 'edad': 20, 'ciudad': 'Argentina'}
```

La flexibilidad es la clave de los argumentos indeterminados en una función, ya que permiten manejar una cantidad indefinida de argumentos. Existen dos tipos de argumentos indeterminados - posicional y de palabras clave - que pueden ser utilizados de manera independiente o simultánea para satisfacer las necesidades específicas de la función. A modo de ejemplo, te presentamos una función que utiliza ambos tipos de argumentos, identificados tanto por su posición como por su 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 escenario, la función "por_posicion_y_nombre" emplea argumentos indeterminados por posición (*args) y por nombre (**kwargs). Al efectuar la llamada a la función con los argumentos "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 realiza una iteración sobre los argumentos por posición (args) y se presenta cada uno de ellos. Posteriormente, se itera sobre los argumentos por nombre (kwargs) y se muestra tanto la clave como el valor correspondiente.

Y como extra, te presentamos un ejemplo más para que puedas ver con mayor claridad el potencial que ofrece esto:

```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
```

Los argumentos indeterminados en Python brindan flexibilidad y versatilidad a las funciones, permitiendo que reciban un número variable de argumentos durante su llamada. Estos argumentos pueden agruparse como argumentos indeterminados por posición (*args) o argumentos indeterminados por nombre (**kwargs).
Los argumentos indeterminados por posición ayudan a manejar una cantidad variable de argumentos posicionales que se agrupan en una tupla dentro de la función. Esto permite pasar múltiples valores sin tener que 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 se agrupan 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. Combinar 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.

Es importante tener en cuenta cuál es la mejor opción según el contexto y los requisitos específicos de la función al utilizar argumentos indeterminados. Esto permite escribir código más legible, mantenible y flexible, adaptándose a diferentes situaciones sin necesidad de redefinir la función para cada caso.
Para finalizar, resumimos el concepto: los argumentos indeterminados son una herramienta poderosa en Python para crear funciones más versátiles y adaptables. Permiten que las funciones reciban 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.

MODULOS

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.

Importando un Módulo en Python

En Python, para utilizar un módulo en un programa, es necesario seguir unos sencillos pasos. En primer lugar, hay que importarlo mediante la instrucción "**import**", seguida del nombre del módulo. Luego, se puede acceder a las definiciones contenidas en el módulo utilizando el nombre del módulo, seguido de un punto y el nombre de la definición correspondiente.
Por ejemplo, si tienes un módulo llamado "mimodulo.py" que incluye una función llamada "saludar()", se puede importar y utilizar dicha función de la siguiente forma:

```
1   import mimodulo
2
3   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:

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

PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL

```
5.0
1.0
2.0
```

El módulo matemático "**math**" ofrece constantes y funciones matemáticas. Con él, se pueden realizar operaciones avanzadas, incluyendo cálculos trigonométricos, exponenciales, logarítmicos, entre otros.

```
1  import datetime
2
3  fecha_actual = datetime.date.today()  # Obtener la fecha actual
4  print(fecha_actual)
5
6  tiempo_actual = datetime.datetime.now()  # Obtener la fecha y hora actual
7  print(tiempo_actual)
```

El módulo "**datetime**" (el cual lo usarás en el proyecto final) proporciona clases y funciones para trabajar con fechas y tiempos.

```
1  import os
2
3  print(os.getcwd())  # Obtener el directorio de trabajo actual
4  os.mkdir('nuevo_directorio')  # Crear un nuevo directorio
```

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

```
1  import csv
2
3  with open('datos.csv', 'r') as archivo:
4      lector_csv = csv.reader(archivo)
5      for fila in lector_csv:
6          print(fila)
```

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

Existen muchos módulos en Python para diferentes propósitos, tales como la manipulación de archivos, el acceso a bases de datos, la generación de gráficos, el procesamiento de imágenes, entre otros.
Aquí te presentamos algunos ej. de módulos en Python, pero hay muchos más disponibles. También es posible crear módulos personalizados (como el primer ejemplo) para organizar y reutilizar el código en diferentes proyectos.

Es importante investigar más allá de las clases y de los apuntes proporcionados para tener un conocimiento más completo sobre las opciones disponibles en Python.

¡No dudes en consultar!