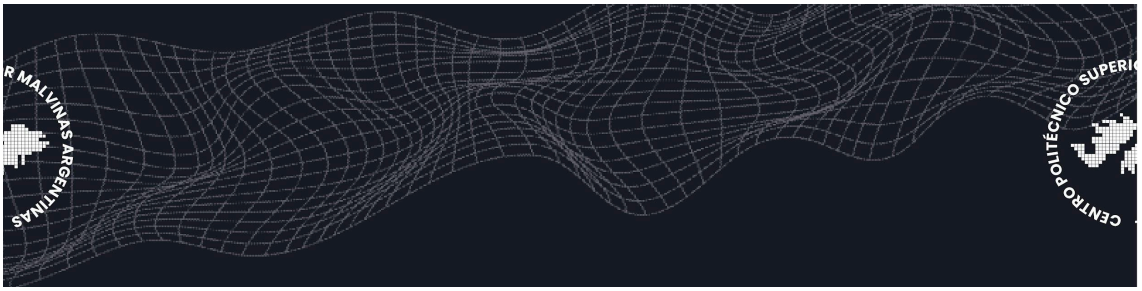


Programación 1

1° Año

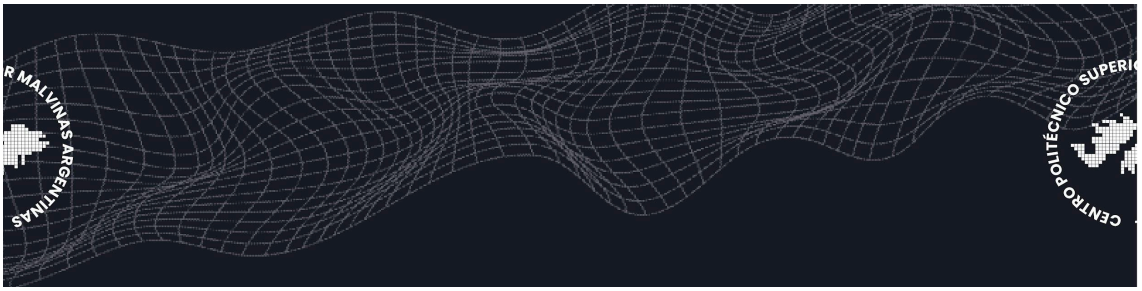
Clase N.° 6

Subprogramas (subalgoritmos) y Funciones

Contenido

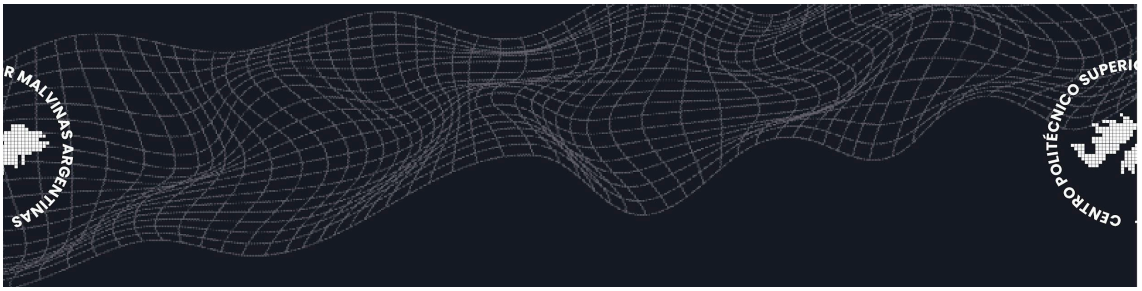
En la clase de hoy trabajaremos los siguientes temas:

- Introducción a los subalgoritmos o subprogramas.
- Funciones, Declaración de funciones, Invocación a las funciones.
- Procedimientos (subrutinas), Sustitución de argumentos/parámetros.
- Ámbito: variables locales y globales.
- Recursión (recursividad).



1. Presentación

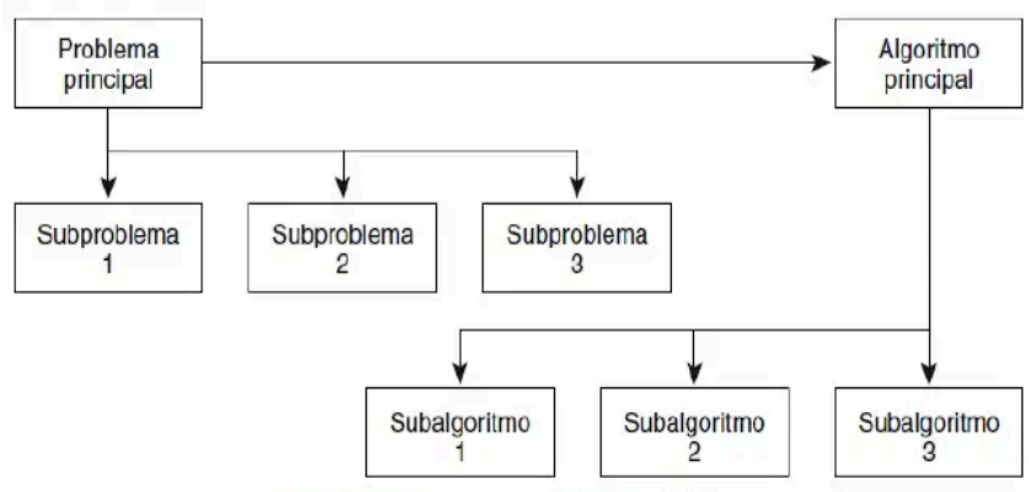
¡Bienvenidos a todos y todas a nuestra sexta clase de Programación! En esta sesión, nos centraremos en los fundamentos cruciales que marcan el inicio de nuestro viaje en este emocionante mundo. Exploraremos temas como los subalgoritmos o subprogramas, con un énfasis especial en funciones, procedimientos, argumentos y parámetros, así como en el ámbito de las variables y la recursión. Estos conceptos son fundamentales para construir programas eficientes y resolver problemas de manera efectiva.

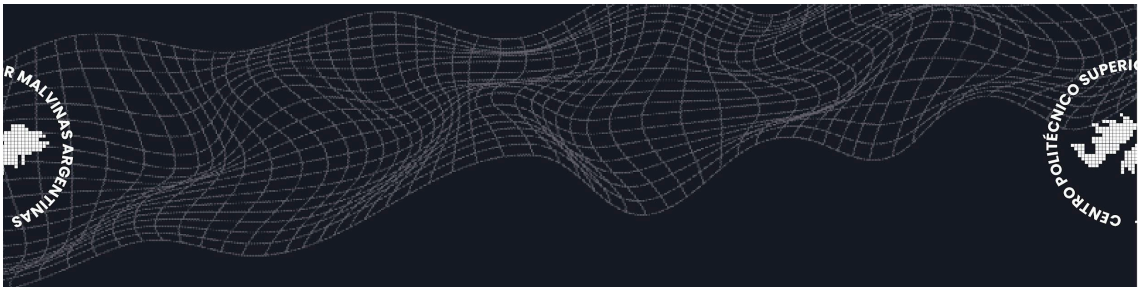


2. Desarrollo y Actividades

Introducción a los subalgoritmos o subprogramas

Los subalgoritmos o subprogramas son bloques de código que realizan una tarea específica y se pueden reutilizar en diferentes partes de un programa. Permiten dividir un programa en partes más pequeñas y manejables, lo que facilita el desarrollo, la depuración y el mantenimiento del código. Los subalgoritmos pueden ser funciones o procedimientos.





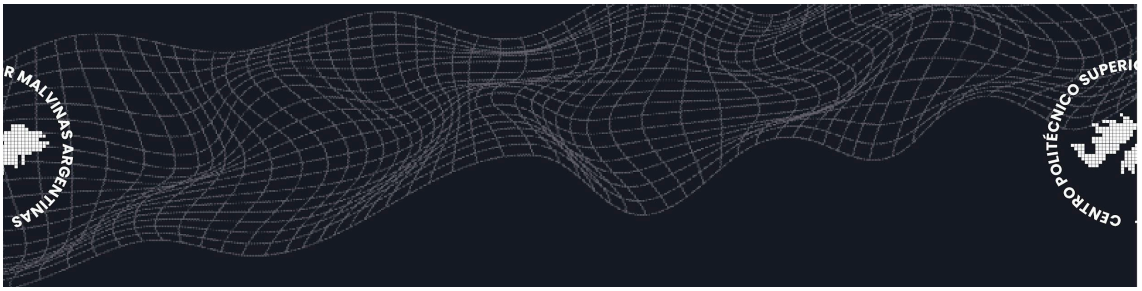
Funciones

Las funciones son subalgoritmos que reciben cero o más argumentos de entrada, realizan un conjunto de instrucciones y devuelven un valor como resultado. En Python, se definen utilizando la palabra clave "def" seguida del nombre de la función, paréntesis que pueden contener los argumentos y dos puntos. A continuación, se escribe el código de la función indentado.

Aquí hay un ejemplo de una función

```
def saludar(nombre):
    """
    Esta función recibe un nombre como parámetro y muestra un saludo.
    """
    print("¡Hola, " + nombre + "! Bienvenido.")

# Ejemplo de uso
saludar("Juan")
```

En este ejemplo, tenemos una función llamada saludar que recibe un parámetro nombre. La función muestra un saludo personalizado utilizando el nombre proporcionado. En este caso, la función imprime "¡Hola, Juan! Bienvenido." en la salida.



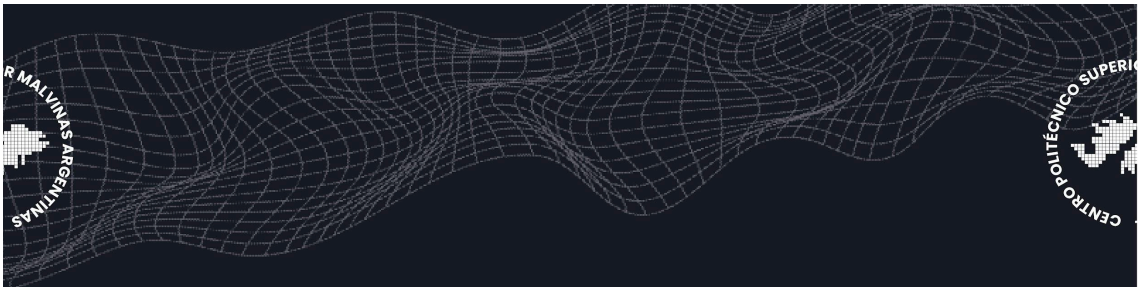
Para definir una función en Python, utilizamos la palabra clave def, seguida del nombre de la función y los parámetros entre paréntesis. En el cuerpo de la función, se especifica el código que se ejecutará cuando se llame a la función. En este caso, simplemente imprimimos un saludo utilizando el parámetro nombre.

Luego, utilizamos la función saludar pasando el argumento "Juan". Esto invoca la función y muestra el saludo correspondiente.

Recordar que la documentación entre comillas triples (""" … """) después de la definición de la función se conoce como el docstring de la función y proporciona una descripción de lo que hace la función.

Declaración de funciones

La declaración de una función implica definir su nombre, los argumentos que recibe y el tipo de valor que devuelve. En Python, no es necesario especificar el tipo de valor de retorno, ya que el lenguaje es de tipado dinámico. Sin embargo, puedes agregar comentarios en el código utilizando la sintaxis de las anotaciones de tipo para indicar el tipo esperado de los argumentos y del valor de retorno de la función.



Aquí hay un ejemplo:

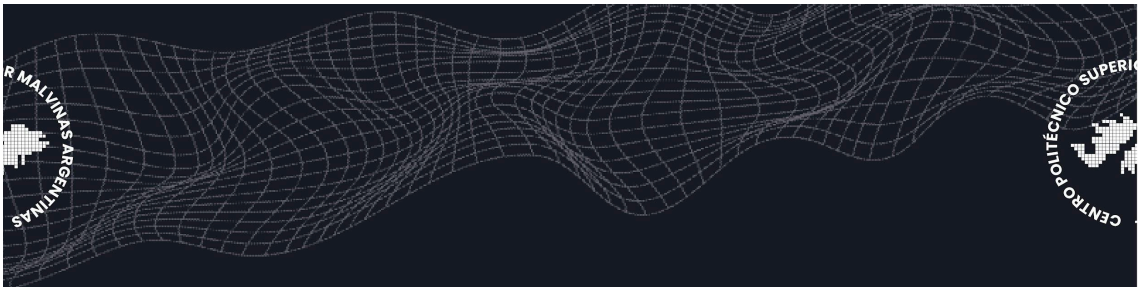
```
def saludar(nombre: str) -> str:
    return "¡Hola, " + nombre + "!"
```

- def es la palabra clave utilizada para definir una función en Python.
- saludar es el nombre de la función.
- (nombre: str) es el parámetro de la función. En este caso, se espera que se proporcione un argumento de tipo str y se asignará a la variable nombre.
- -> str indica el tipo de valor que la función devuelve. En este caso, la función devuelve un valor de tipo str.

```
return "¡Hola, " + nombre + "!"
```

return es una declaración utilizada para devolver un valor de una función. En este caso, se devuelve una cadena de texto que contiene el saludo formateado.
"¡Hola, " es una cadena de texto que representa la primera parte del saludo.
+ nombre + es la concatenación del valor de la variable nombre (el argumento proporcionado cuando se llama a la función) con la cadena de texto.
"!" es una cadena de texto que representa la última parte del saludo.

En resumen, la función saludar toma un nombre como argumento, lo concatena con una cadena de texto que representa un saludo y devuelve el resultado como una cadena de texto.
Es importante destacar que, en este caso, el tipo de los parámetros y



el tipo de retorno están anotados utilizando la sintaxis de anotaciones de tipo de Python. Esto no afecta directamente el comportamiento de la función, pero puede ser útil para documentar y comprender mejor los tipos esperados y retornados por una función.

Declaración de devolución

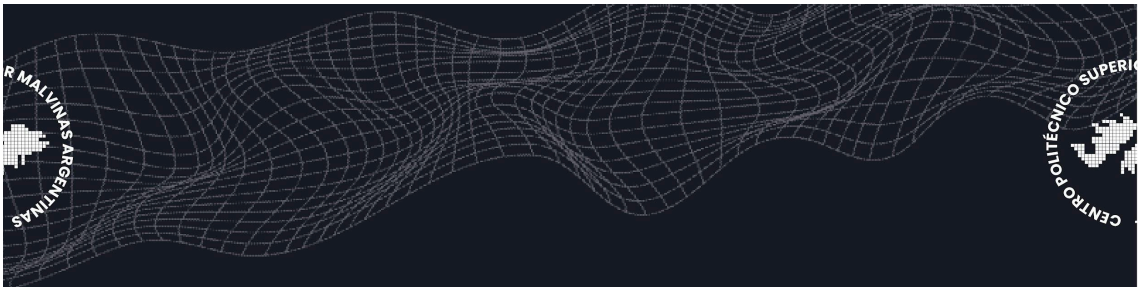
una función puede usar la return declaración para devolver un valor como resultado de la función. Este valor puede asignarse a una variable o usarse en otras expresiones. Por ejemplo:

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

result = add_numbers(3, 5)
print(result)  # Output: 8
```

Argumentos de palabras clave

al llamar a una función, puede especificar argumentos por sus nombres de parámetro. Esto le permite pasar argumentos en cualquier orden y hace que el código sea más legible. Por ejemplo:

```

def describe_person(name, age):
    print("Name:", name)
    print("Age:", age)

describe_person(age=25, name="John")
# Output:
# Name: John
# Age: 25
```



Número variable de argumentos

las funciones de Python pueden recibir un número variable de argumentos utilizando la sintaxis *argso **kwargs. *argspermite pasar múltiples argumentos posicionales, mientras que **kwargspermite pasar múltiples argumentos de palabras clave. Aquí hay un ejemplo:

```
def calculate_total(*args):
    total = sum(args)
    return total

result = calculate_total(2, 4, 6, 8)
print(result)  # Output: 20
```

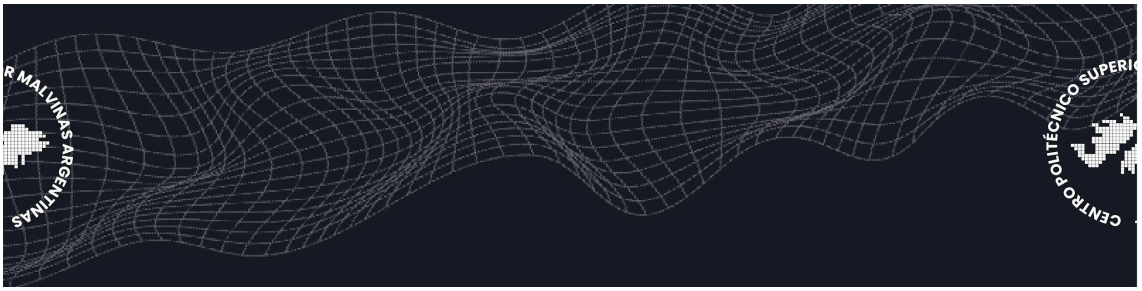
Estos son solo algunos aspectos adicionales de las funciones en Python. Las funciones son un concepto clave en la programación, ya que ayudan a organizar el código, promueven la reutilización y facilitan la comprensión y el mantenimiento de sus programas.

Invocación a las funciones

La invocación o llamada a una función se realiza escribiendo el nombre de la función seguido de paréntesis que pueden contener los argumentos necesarios. Aquí hay un ejemplo de invocación a la función "saludar" del ejemplo anterior:

```
nombre = "Juan"
mensaje = saludar(nombre)
print(mensaje)  # Imprimirá "¡Hola, Juan!"
```

En este caso, se pasa el valor de la variable "nombre" como argumento a la función "saludar" y se guarda el valor de retorno en la variable "mensaje".



Procedimientos (subrutinas)

Los procedimientos, también conocidos como subrutinas, son subalgoritmos similares a las funciones, pero no devuelven un valor. En Python, se definen de la misma manera que las funciones, pero no se utiliza la instrucción "return". Aquí hay un ejemplo de un procedimiento que imprime un mensaje:

```
def imprimir_mensaje():
    print("Este es un mensaje")
```

Para invocar a un procedimiento, simplemente se escribe su nombre seguido de paréntesis, sin necesidad de asignar el valor de retorno a una variable. Por ejemplo:

```
imprimir_mensaje()
```

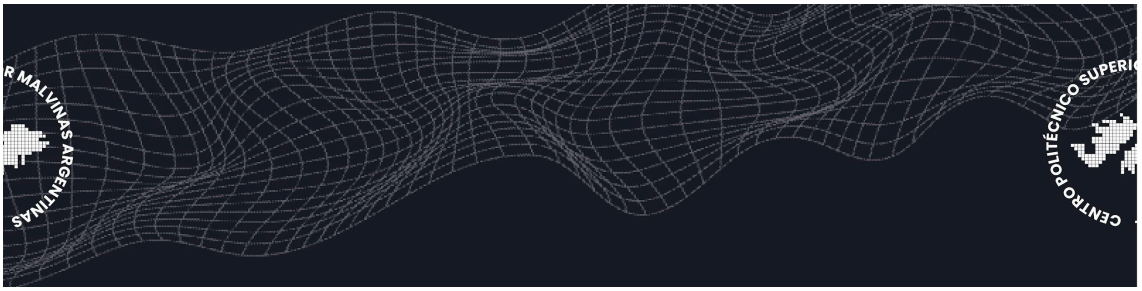
El procedimiento "imprimir_mensaje" se ejecutará y mostrará el mensaje en la consola.

Sustitución de argumentos/parámetros

La sustitución de argumentos o parámetros en Python se refiere al proceso de proporcionar valores concretos a los parámetros de una función al llamarla. Estos valores se utilizan para realizar cálculos o ejecutar acciones dentro de la función.

Cuando se define una función, se pueden especificar uno o más parámetros que actúan como espacios reservados para los valores que se utilizarán cuando se llame a la función. Estos parámetros pueden ser variables que almacenan los valores pasados a la función. Aquí hay un ejemplo sencillo para ilustrar la sustitución de argumentos en Python:

```
def saludar(nombre):
    print("¡Hola, " + nombre + "!")
```




```
# Llamada a la función con un argumento concreto
saludar("Juan")
```

En este caso, se define una función llamada saludar que tiene un parámetro llamado nombre. Al llamar a la función saludar("Juan"), se está sustituyendo el argumento "Juan" en lugar del parámetro nombre dentro de la función. Como resultado, la función imprimirá "¡Hola, Juan!" en la consola.

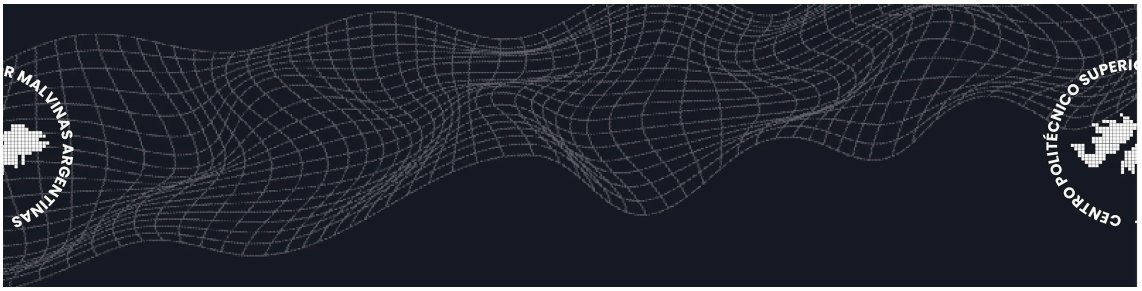
Es importante tener en cuenta que los argumentos pueden ser de diferentes tipos, como cadenas de texto, números, listas, diccionarios, entre otros. Además, es posible pasar múltiples argumentos separados por comas al llamar a una función, siempre y cuando se correspondan con los parámetros definidos en la función.

Por ejemplo:

```
def sumar(a, b):
    resultado = a + b
    print("El resultado de la suma es:", resultado)

# Llamada a la función con dos argumentos
sumar(3, 5)
```

En este caso, la función sumar recibe dos argumentos: a y b. Al llamar a la función sumar(3, 5), se sustituyen los argumentos 3 y 5 en lugar de los parámetros a y b, respectivamente. La función imprimirá "El resultado de la suma es: 8" en la consola.

La sustitución de argumentos o parámetros es fundamental para que las funciones puedan recibir datos específicos y realizar operaciones basadas en ellos. Permite que las funciones sean más flexibles y reutilizables, ya que se pueden llamar con diferentes valores en diferentes contextos.



Ámbito: variables locales y globales

El ámbito en Python se refiere al alcance o contexto en el que una variable es válida y puede ser accedida. En Python, existen dos tipos principales de ámbito: variables locales y variables globales.

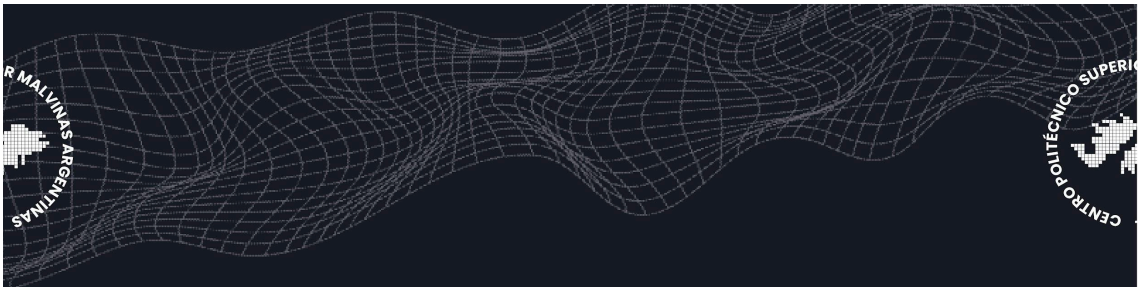
Variables locales

Las variables locales son aquellas que se definen dentro de una función y solo pueden ser accedidas desde dentro de esa función. Estas variables son visibles y utilizables solo dentro del bloque de código en el que se definen. Una vez que la función termina su ejecución, las variables locales se eliminan de la memoria.

Aquí tienes un ejemplo de una variable local

```
def mi_funcion():
    x = 10  # Variable local
    print(x)

mi_funcion()  # Salida: 10
print(x)  # Error: NameError: name 'x' is not defined
```

En este caso, la variable x es una variable local que se define dentro de la función mi_funcion(). Solo se puede acceder a ella dentro de la función y no fuera de ella. Al tratar de acceder a x fuera de la función, se generará un error.



Variables globales

Las variables globales son aquellas que se definen fuera de cualquier función o bloque de código y están disponibles en todo el programa. Estas variables pueden ser accedidas y utilizadas desde cualquier parte del código, ya sea dentro o fuera de las funciones.

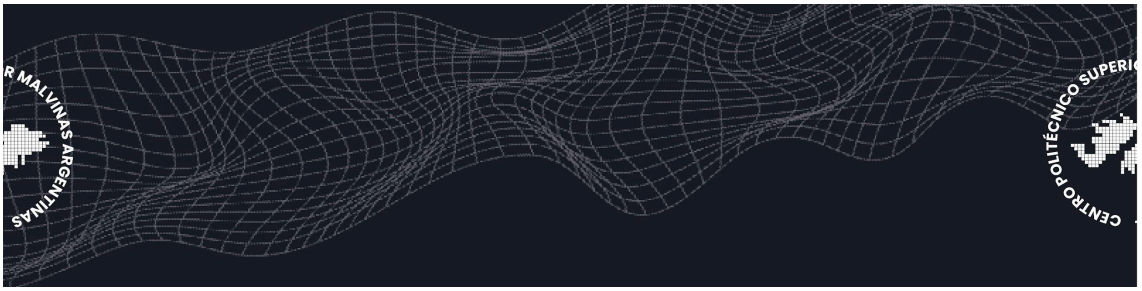
Aquí tienes un ejemplo de una variable global:

```
x = 10  # Variable global

def mi_funcion():
    print(x)

mi_funcion()  # Salida: 10
print(x)  # Salida: 10
```

En este caso, la variable x es una variable global que se define fuera de cualquier función. Puede ser accedida tanto dentro de la función mi_funcion() como fuera de ella, ya que su ámbito se extiende a todo el programa.

Es importante tener en cuenta que, si se define una variable local con el mismo nombre que una variable global, la variable local tendrá precedencia dentro del ámbito de la función. Esto se conoce como "enmascaramiento" de la variable global.

```
x = 10  # Variable global
```



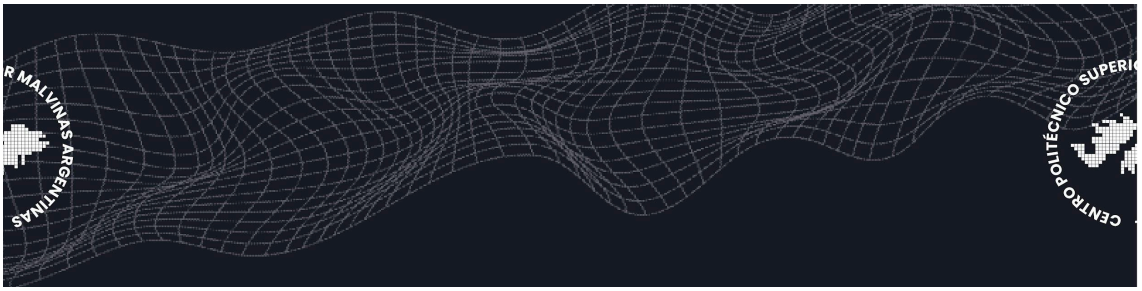

```python
def mi_funcion():
    x = 5  # Variable local que enmascara a la variable global
    print(x)

mi_funcion()  # Salida: 5
print(x)  # Salida: 10
```

En este ejemplo, la variable local x dentro de la función mi_funcion() enmascara a la variable global x. Por lo tanto, dentro de la función, se imprimirá el valor de la variable local (5), mientras que fuera de la función, se imprimirá el valor de la variable global (10).

Es importante entender y tener en cuenta el ámbito de las variables en Python para evitar errores y asegurarse de acceder a las variables adecuadas en el contexto correcto.

Recursión (recursividad)

La recursión, también conocida como recursividad, es un concepto en programación que se refiere a la capacidad de una función de llamarse a sí misma directa o indirectamente. En Python, se puede utilizar la recursión para resolver problemas dividiéndolos en subproblemas más pequeños y resolviendo cada subproblema de manera recursiva.



La recursión se basa en dos conceptos fundamentales:

1. Caso base:

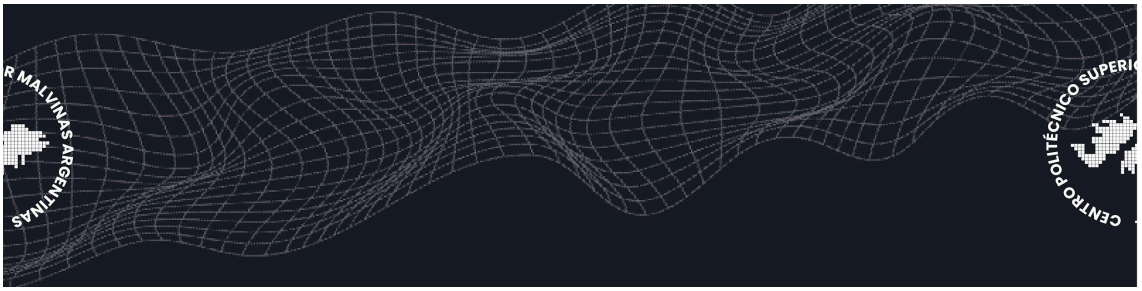
   Es el punto de terminación de la recursión. Define una condición que se evalúa dentro de la función recursiva y determina cuándo se debe detener la llamada recursiva. El caso base evita que la función se llame infinitamente y asegura que la recursión se detenga en algún momento.

2. Caso recursivo:

   Es la parte donde la función se llama a sí misma para resolver un problema más pequeño o similar al original. En cada llamada recursiva, el problema se simplifica o se reduce en tamaño hasta que se alcance el caso base.

Aquí tienes un ejemplo sencillo para ilustrar la recursión en Python:

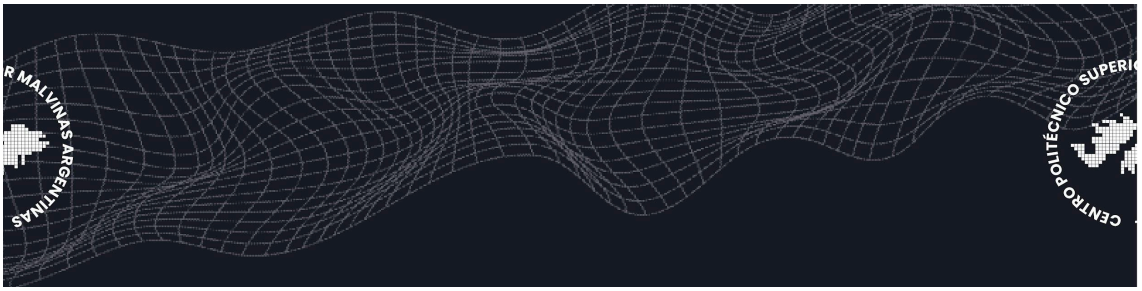
```python
def contar_hasta_cero(n):
    if n <= 0:  # Caso base
        print("¡Boom!")
    else:  # Caso recursivo
        print(n)
        contar_hasta_cero(n - 1)  # Llamada recursiva

contar_hasta_cero(5)
```



En este caso, la función contar_hasta_cero() cuenta desde un número dado n hasta cero. El caso base se verifica cuando es menor o igual a cero, en cuyo caso se imprime "¡Boom!". En el caso recursivo, se imprime el valor actual de n y luego se llama a la función contar_hasta_cero() con n - 1 para continuar el proceso de contar hacia cero.

```
5
4
3
2
1
¡Boom!
```

Es importante tener en cuenta que la recursión debe utilizarse con precaución, ya que puede consumir mucha memoria y tiempo de ejecución si no se implementa correctamente. Un error común en la recursión es olvidar definir un caso base o no asegurarse de que el caso recursivo converja hacia el caso base. Esto puede provocar un bucle infinito y agotar los recursos del sistema.



1. Suma de los elementos de una lista

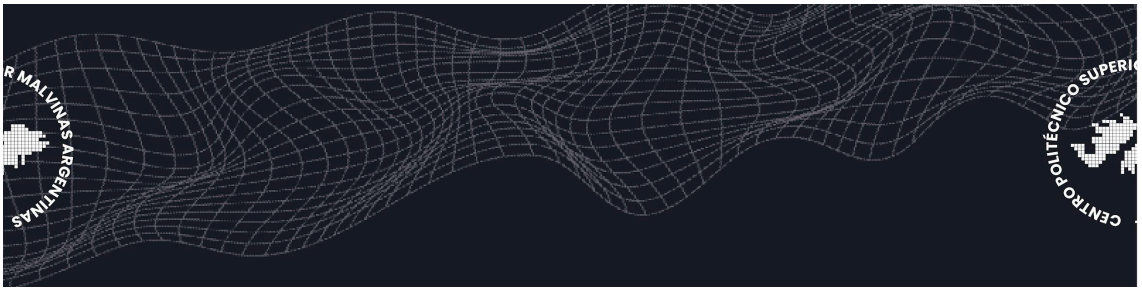
Escribir una función llamada "sumar_lista" que reciba como parámetro una lista de números enteros y devuelva la suma de todos los elementos de la lista.

2. Cálculo del factorial de un número utilizando recursión.

Escribe una función llamada "factorial" que tome un número entero como parámetro y devuelva su factorial. El factorial de un número se calcula multiplicando todos los números enteros desde 1 hasta el número dado.

3. Ordenar cadena de caracteres.

Escribe una función llamada ordenar_cadena que tome una cadena de texto como parámetro y devuelva una nueva cadena con las letras ordenadas alfabéticamente, manteniendo la distinción entre mayúsculas y minúsculas. Por ejemplo, si se pasa la cadena "PythonEsDivertido", la función debería devolver "DEIhnoprstuvy".



3. Conclusión

En resumen y como cierre, los temas mencionados se centran en la programación estructurada y la modularidad del código.

Estos conceptos son fundamentales en programación y promueven la modularidad, la reutilización del código y una mejor estructura en el desarrollo de programas. Comprender y aplicar estos temas permite escribir programas más legibles, mantenibles y eficientes.

Bibliografía recomendada

- **"Python Crash Course" de Eric Matthes:** Este libro es ideal para principiantes y brinda una introducción completa a Python, incluyendo funciones y procedimientos.

- **"Learning Python" de Mark Lutz:** Este libro abarca todos los aspectos de Python y ofrece una cobertura detallada de funciones, procedimientos y otros temas avanzados.

- **"Python Cookbook" de David Beazley y Brian K. Jones:** Este libro es una excelente referencia para programadores de Python más experimentados.

- **"Fluent Python" de Luciano Ramalho:** Este libro explora las características más avanzadas de Python y proporciona una comprensión profunda del lenguaje.