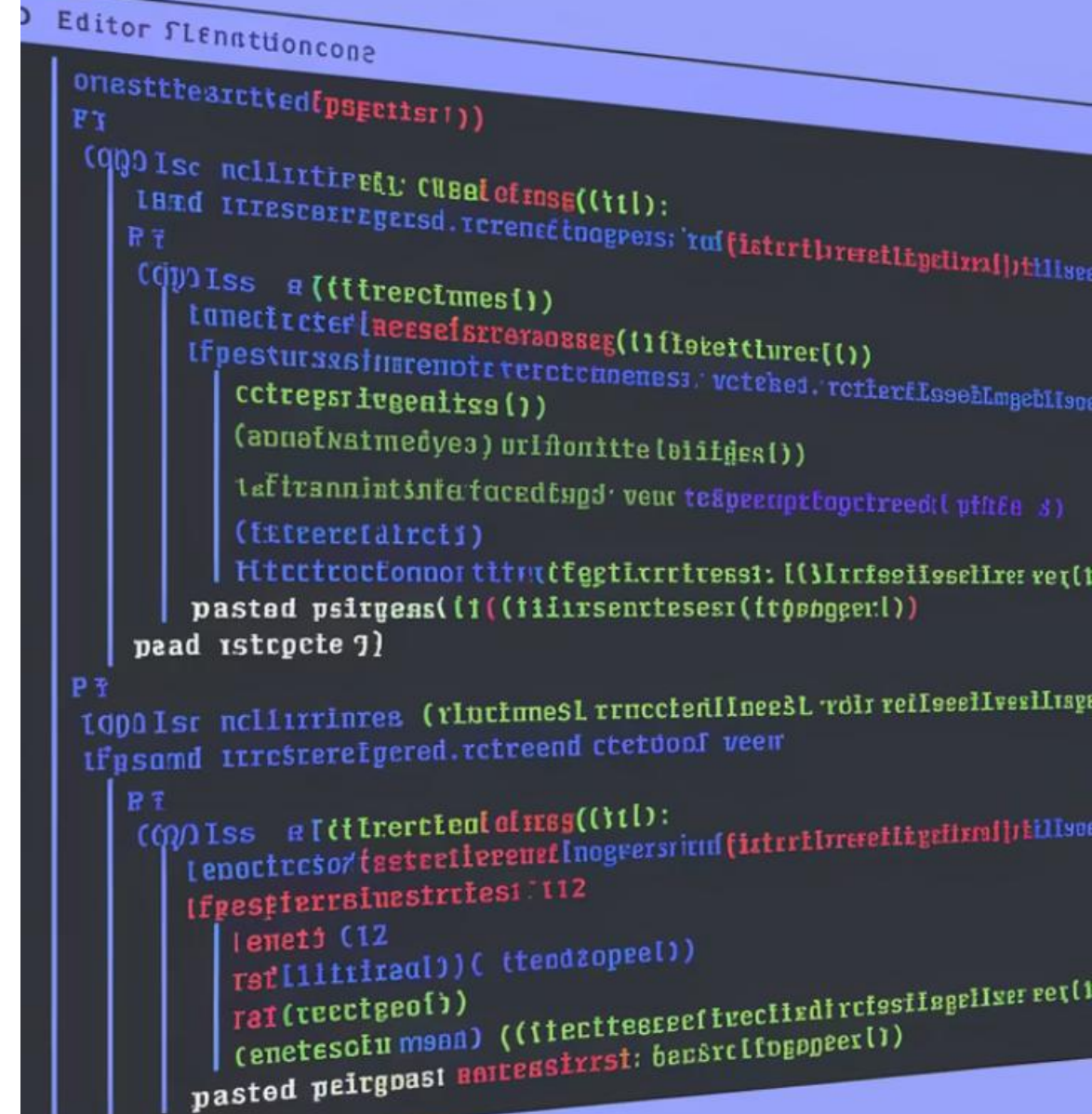


# Funciones y Módulos en Python

Las funciones y módulos son componentes fundamentales en Python que nos permiten escribir código más eficiente y organizado. Cuando nos enfrentamos a tareas repetitivas como mostrar saludos, hacer cálculos o validar datos, estas herramientas nos ayudan a evitar la duplicación de código, haciendo nuestros programas más ordenados y fáciles de mantener.

En esta presentación, exploraremos qué son las funciones y módulos, cómo crearlos y utilizarlos, y las mejores prácticas para implementarlos en nuestros proyectos de Python. Veremos ejemplos prácticos y aprenderemos a organizar nuestro código de manera profesional.

 por Kibernetum Capacitación S.A.



```
Editor Pythoncode
def saludar(nombre):
    """Función que devuelve un saludo personalizado"""
    return f'¡Hola, {nombre}!'

def sumar(a, b):
    """Función que devuelve la suma de dos números"""
    return a + b

def main():
    """Función principal que ejecuta el programa"""
    nombre = input('Nombre: ')
    resultado = saludar(nombre)
    print(resultado)

    a = 5
    b = 3
    resultado = sumar(a, b)
    print(f'Resultado de la suma: {resultado}')

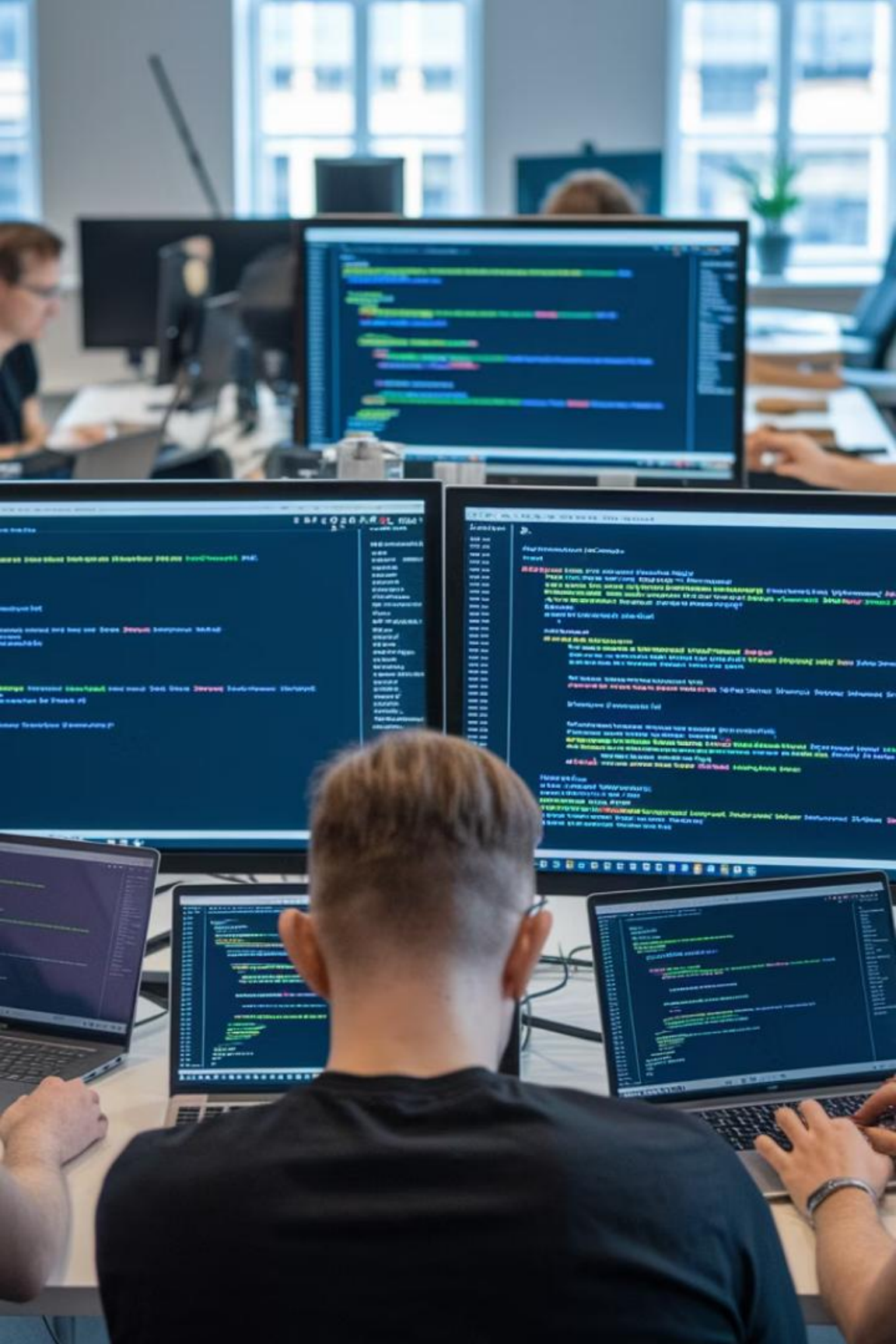
if __name__ == '__main__':
    main()
```

# Conceptos Fundamentales de Python

¿Cuál es la diferencia entre una variable de tipo entero y una de tipo flotante en Python?

Explica cómo funcionan las estructuras condicionales (if, elif, else) en Python y proporciona un ejemplo práctico.

Describe el propósito y uso de los operadores lógicos (and, or, not) en Python.



# Ventajas de Usar Funciones y Módulos

## Reutilización de Código

Permiten utilizar el mismo código en diferentes partes del programa sin necesidad de repetirlo, ahorrando tiempo y esfuerzo.

## Organización y Claridad

Hacen que el programa sea más ordenado y fácil de entender, dividiendo problemas complejos en partes pequeñas y manejables.

## Colaboración Eficiente

Facilitan el trabajo en equipo, permitiendo que varios desarrolladores trabajen en diferentes módulos simultáneamente.

## Herramientas Reutilizables

Se convierten en herramientas personales o profesionales que puedes usar en múltiples proyectos a lo largo del tiempo.

# ¿Qué es una Función en Python?

Una función es un bloque de código que realiza una tarea específica. Es como una máquina: le das instrucciones una vez, y luego puedes llamarla cuando quieras para que haga su trabajo.



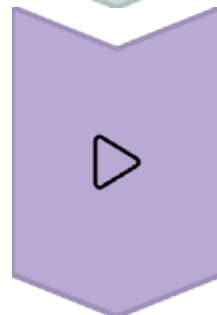
## Definición

Se define con la palabra clave "def", seguida del nombre de la función y paréntesis que pueden contener parámetros.



## Bloque de Código

El cuerpo de la función está indentado y contiene las instrucciones que se ejecutarán cuando la función sea llamada.



## Llamada

Para ejecutar la función, se escribe su nombre seguido de paréntesis, incluyendo los argumentos necesarios.

```
def nombre_de_la_funcion():  
    """ # Bloque de instrucciones  
    """
```

```
def saludar():  
    print("¡Hola desde la función!")  
  
# Aquí la usamos o "llamamos":  
saludar()
```

# Parámetros y Argumentos

## Parámetros

Son las variables que se declaran en la definición de la función, como espacios en blanco que se rellenarán posteriormente.

Aparecen entre los paréntesis al definir la función y actúan como variables locales dentro de ella.

## Argumentos

Son los valores reales que se pasan cuando llamamos a la función, rellenando los espacios definidos por los parámetros.

Pueden ser valores literales, variables, expresiones o incluso otras funciones que devuelvan un valor.

```
def saludar(nombre):  
    """# nombre es un parámetro  
    print("Hola", nombre)  
  
saludar("Facundo")  
    """# "Facundo" es el argumento
```



# Retorno de Valores con Return

```
def sumar(a, b):  
    return a + b  
  
resultado = sumar(3, 4)  
print("La suma es:", resultado)
```

💡 Usar **return** **no imprime** nada: simplemente **devuelve** el valor para que tú lo uses como quieras.

## Definir la Función con Return

La palabra clave "return" permite que una función devuelva un valor como resultado de su ejecución. Este valor puede ser almacenado en una variable o utilizado directamente.

## Procesar Datos

La función realiza cálculos o procesa información utilizando los parámetros recibidos y cualquier lógica interna definida.

## Devolver Resultado

Al encontrar la instrucción "return", la función finaliza su ejecución y devuelve el valor especificado, que puede ser utilizado por el código que llamó a la función.

# python's in functi mprehens overview

BY MIGUEL VASQUEZ

## Funciones Preconstruidas en Python



### Matemáticas

Funciones como `abs()`, `round()`, `sum()` y `max()` que realizan operaciones matemáticas comunes sin necesidad de escribir el código desde cero.



### Manipulación de Datos Datos

Funciones como `len()`, `sorted()` y `type()` que ayudan a trabajar con diferentes tipos de datos y estructuras en Python.



### Entrada/Salida

Funciones como `print()` e `input()` que facilitan la interacción con el usuario a través de la consola o terminal.



### Conversión

Funciones como `int()`, `str()` y `float()` que permiten convertir valores entre diferentes tipos de datos según sea necesario.

```
print(len("Hola"))      # Devuelve 4
print(type(42))         # Muestra el tipo de dato: <class 'int'>
print(round(3.14159, 2)) # Redondea → 3.14
```

# Funciones con Argumentos Predeterminados

Python permite establecer valores por defecto para los parámetros de una función. Estos valores se utilizarán cuando no se proporcione un argumento específico al llamar a la función.



## Definición con Valores Predeterminados

Se asignan valores a los parámetros directamente en la definición de la función usando el operador de asignación (=).



## Argumentos Opcionales

Los parámetros con valores predeterminados se convierten en opcionales, permitiendo llamar a la función con menos argumentos.



## Mayor Flexibilidad

Permite crear funciones más versátiles que pueden adaptarse a diferentes situaciones sin necesidad de múltiples versiones.

```
def saludar(nombre="invitado"):
    print("Hola", nombre)

saludar()           # Hola invitado
saludar("Sofía")    # Hola Sofía
```



# Argumentos Variables: `*args` y `**kwargs`

## `*args`

Permite pasar un número variable de argumentos posicionales a una función. Dentro de la función, estos argumentos se tratan como una tupla.

- Se accede a los valores por posición (índice)
- Útil cuando no sabes cuántos argumentos recibirás
- Ejemplo: `def suma(*numeros)`

## `**kwargs`

Permite pasar un número variable de argumentos con nombre (clave-valor) a una función. Dentro de la función, estos argumentos se tratan como un diccionario.

- Se accede a los valores por nombre (clave)
- Útil para funciones con muchas opciones configurables
- Ejemplo: `def perfil(**datos)`

# Argumentos Variables: \*args y \*\*kwargs

```
def sumar_todo(*numeros):  
    print("Suma:", sum(numeros))  
  
sumar_todo(1, 2, 3, 4)  
  
def mostrar_info(**datos):  
    for clave, valor in datos.items():  
        print(f"{clave}: {valor}")  
  
mostrar_info(nombre="Miguel", edad=30)
```

# Funciones Lambda (Anónimas)

Las funciones lambda son funciones simples y rápidas que se definen en una sola línea. Se utilizan principalmente para operaciones cortas, especialmente como argumentos en otras funciones.



## Sintaxis Compacta

Se definen con la palabra clave "lambda", seguida de parámetros, dos puntos y la expresión a evaluar.

$f(x)$

## Funcionalidad Limitada

Solo pueden contener una expresión, sin instrucciones múltiples ni bloques complejos.



## Uso con Funciones de Orden Superior

Ideales para usar con `map()`, `filter()` y `sorted()` como funciones de callback rápidas.

# Funciones Lambda (Anónimas)

Sintaxis general:

```
lambda parametros: expresión
```

Ejemplo básico:

```
doble = lambda x: x * 2  
print(doble(5)) # Resultado: 10
```

Otro ejemplo:

```
suma = lambda a, b: a + b  
print(suma(3, 4)) # Resultado: 7
```

Esto equivale a:

```
def doble(x):  
    return x * 2
```

# Funciones Recursivas

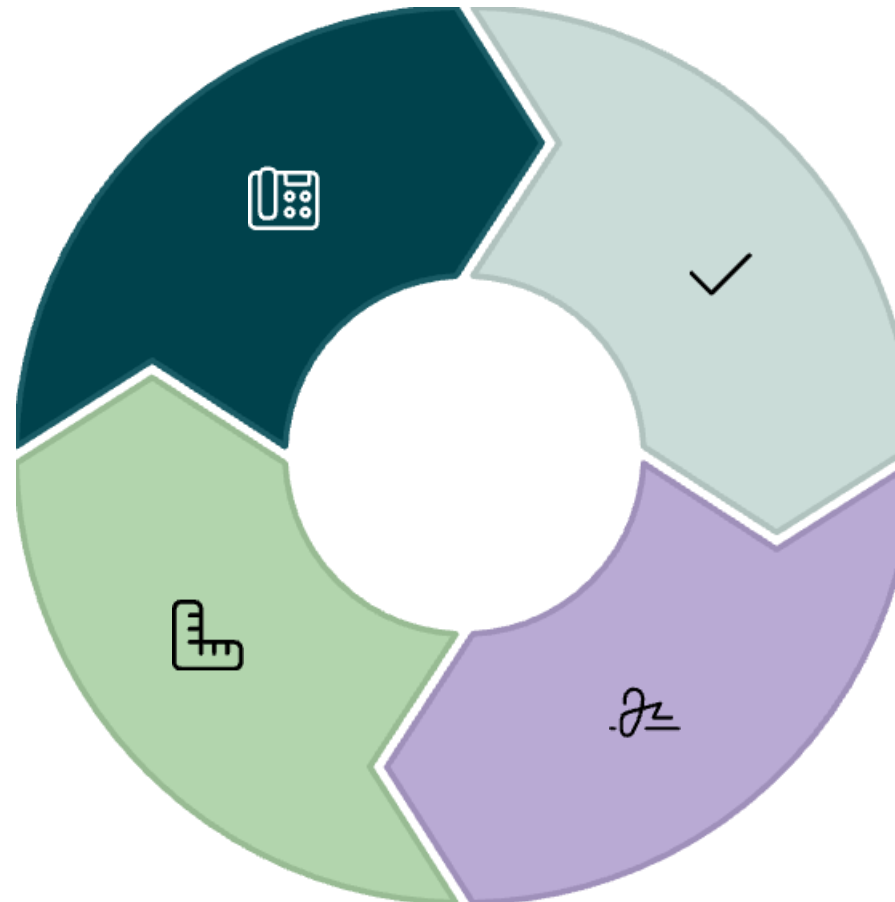
Una función recursiva es aquella que se llama a sí misma durante su ejecución. Este enfoque es útil para resolver problemas que pueden descomponerse en casos más simples del mismo problema.

## Llamada Inicial

La función se invoca con los parámetros iniciales del problema a resolver.

## Combinación de Resultados

Se combinan los resultados de las llamadas recursivas para obtener la solución final.



## Caso Base

Se verifica si se ha llegado al caso más simple que puede resolverse directamente.

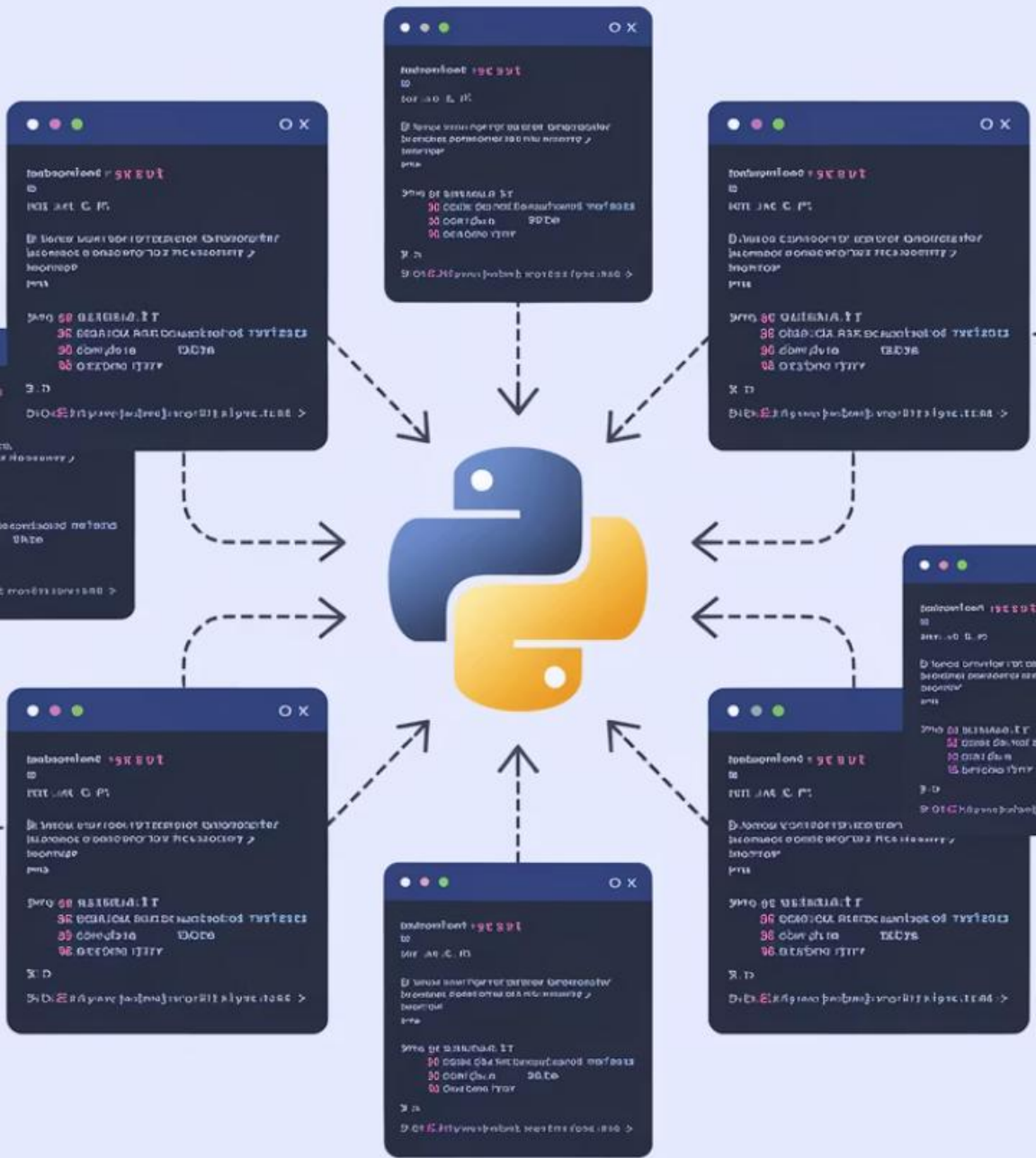
## Llamada Recursiva

Si no es el caso base, la función se llama a sí misma con un problema más pequeño.



# Funciones Recursivas

```
def factorial(n):  
    if n == 0:  
        return 1  
    else:  
        return n * factorial(n - 1)  
  
print(factorial(5)) # 120
```



# ¿Qué es la Modularización en Python?

La modularización es el proceso de dividir un programa grande en partes más pequeñas y manejables llamadas módulos. En Python, un módulo es simplemente un archivo .py que contiene código (funciones, clases, variables) que puede ser importado y reutilizado en otros programas.

1

## Archivo

Cada módulo es un archivo .py independiente con su propio código y funcionalidad.

∞

## Reutilización

Los módulos pueden importarse y utilizarse en múltiples programas diferentes.

5+

## Organización

Permiten estructurar proyectos grandes en componentes lógicos y manejables.

# Beneficios e Inconvenientes de la Modularización

## Beneficios

- Reutilización de código sin repetirlo
- Mantenimiento más sencillo y localizado
- Facilita la colaboración en equipo
- Mejora la legibilidad y organización
- Permite mayor escalabilidad del proyecto
- Facilita las pruebas unitarias

## Inconvenientes y Soluciones

- Demasiados archivos → Usar estructura de carpetas clara
- Dependencias circulares → Evitar importaciones cruzadas
- Dificultad para principiantes → Enseñar con ejemplos simples
- Sobrecarga en proyectos pequeños → Modularizar solo cuando sea necesario
- Gestión de nombres y rutas → Usar convenciones claras

# Creación y Uso de Módulos



## Crear un Archivo de Módulo

Crea un archivo Python (por ejemplo, utilidades.py) con las funciones, clases o variables que desees reutilizar.



## Importar el Módulo

En tu programa principal, usa la instrucción "import" para acceder al contenido del módulo (import utilidades).



## Usar las Funciones

Accede a las funciones del módulo usando la notación de punto (utilidades.funcion()) o importándolas directamente (from utilidades import funcion).



## Organizar en Paquetes

Para proyectos más grandes, agrupa módulos relacionados en paquetes (carpetas con un archivo `__init__.py`).

# Creación y Uso de Módulos

## Crear un módulo propio

1. Crea un archivo llamado `utilidades.py` con esto:

```
def saludar(nombre):  
    print("Hola", nombre)  
    . . .
```

2. En tu programa principal:

```
import utilidades  
utilidades.saludar("Facundo")
```



# Librería Estándar

Python incluye cientos de módulos listos para usar. Algunos comunes:

Módulo	¿Para qué sirve?
math	Operaciones matemáticas
random	Números aleatorios
datetime	Fechas y tiempos
os	Acceso a archivos y carpetas del SO

Estos módulos forman parte de la *librería estándar de Python*, lo que significa que no necesitas instalarlos, ya vienen incluidos con el lenguaje.

# Paquetes

Un **paquete** es una carpeta que contiene varios módulos y un archivo `__init__.py`. Permite organizar grandes proyectos.  
Ejemplo:

```
mi_paquete/  
├── __init__.py  
├── modulo1.py  
└── modulo2.py
```

```
from mi_paquete import modulo1
```

¿Y para qué sirve `__init__.py` en este contexto?

En el ejemplo mostrado, el archivo `__init__.py` convierte la carpeta `mi_paquete` en un *paquete válido de Python*. Esto permite que puedas importar módulos como `modulo1` usando la sintaxis:

```
from mi_paquete import modulo1
```

Aunque el archivo puede estar vacío, también se puede usar para inicializar variables, configurar el entorno del paquete o exponer funciones directamente al importar el paquete.

Así, los paquetes ayudan a **organizar el código en proyectos grandes**, separando funcionalidades en distintos archivos reutilizables.

# Módulo Math

El módulo math proporciona funciones matemáticas estándar definidas por el lenguaje C. Sirve para realizar operaciones matemáticas más avanzadas que las básicas como suma o resta.

## Algunas funciones comunes del módulo math:

- `math.sqrt(x)` → raíz cuadrada de x.
- `math.pow(x, y)` → x elevado a la potencia y.
- `math.floor(x)` → redondea hacia abajo.
- `math.ceil(x)` → redondea hacia arriba.
- `math.pi` → constante  $\pi$ .
- `math.e` → constante e (número de Euler).
- `math.sin(x)`, `math.cos(x)`, `math.tan(x)` → funciones trigonométricas.
- `math.log(x[, base])` → logaritmo de x, base natural por defecto.

## ¿Cómo usar Math?

Primero hay que importarlo:

```
import math

print(math.sqrt(16)) # Resultado: 4.0
print(math.pi)      # Resultado: 3.141592653589793
```

Es útil cuando necesitas precisión y operaciones científicas o técnicas.

# Documentación y Buenas Prácticas

```
def sumar(a, b):  
    """  
    ... Suma dos números y devuelve el resultado.  
  
    ... Parámetros:  
    ... a (int o float): Primer número.  
    ... b (int o float): Segundo número.  
  
    ... Retorna:  
    ... int o float: La suma de a y b.  
    ... """  
    return a + b  
  
print(sumar(2, 3))
```

**Comentario en línea (al final de una línea):**

```
x = 10 # Número de elementos iniciales
```



## Docstrings

Usa comillas triples ("""") para documentar módulos, clases y funciones. Explica qué hace el código, parámetros y valores de retorno.



## Convenciones de Nombrado

Sigue PEP 8: snake\_case para variables y funciones, PascalCase para clases, y UPPER\_SNAKE para constantes.



## Comentarios Efectivos

Usa comentarios para explicar el "por qué" del código, no el "qué". Mantén los comentarios actualizados y relevantes.



## Estructura Coherente

Organiza tu código de manera lógica y consistente. Agrupa funcionalidades relacionadas y separa las que no lo están.

# Actividad Práctica: Creación de Módulos

## Objetivo:

Aplicar los conceptos de funciones y módulos creando un módulo saludos.py con funciones personalizadas, y utilizando dicho módulo desde un archivo principal main.py.

## Paso 1: Crea el modulo saludos.py

```
def saludar(nombre):  
    return f"¡Hola, {nombre}! Bienvenido a Python."  
  
def despedirse(nombre):  
    return f"Hasta luego, {nombre}. ¡Buen trabajo!"  
.....
```



# Actividad Práctica: Creación de Módulos

## Objetivo:

Aplicar los conceptos de funciones y módulos creando un módulo saludos.py con funciones personalizadas, y utilizando dicho módulo desde un archivo principal main.py.

## Paso 2: Crea el archivo principal main.py

```
import saludos

nombre = input("¿Cuál es tu nombre? ")

print(saludos.saludar(nombre))
print("Haciendo algo interesante en Python...")
print(saludos.despedirse(nombre))
```

# Actividad Práctica: Creación de Módulos

## Objetivo:

Aplicar los conceptos de funciones y módulos creando un módulo saludos.py con funciones personalizadas, y utilizando dicho módulo desde un archivo principal main.py.

## Paso 3: Ejecuta el programa



```
python main.py
```

## Reflexión:

1. ¿Qué ventaja tuviste al separar el código en dos archivos?
2. ¿Cómo podrías extender el módulo saludos.py con más funciones útiles?

# Desafío: Crear un Módulo de Operaciones Matemáticas Básicas

## Objetivo:

Crear un módulo operaciones.py que contenga funciones para sumar, restar, multiplicar y dividir dos números. Luego, usar ese módulo desde un archivo main.py que pida al usuario ingresar dos números y mostrar los resultados de cada operación.

## Requisitos:

- Crear el módulo operaciones.py con las funciones correspondientes.
- Crear el archivo main.py que importe y use las funciones del módulo.
- Asegurarse de incluir comentarios y docstrings en cada función.
- Usar input() para obtener los números del usuario y print() para mostrar los resultados.
- Tiempo de desarrollo: 20 minutos
- Modalidad: individual

# Material Complementario

Para profundizar en el tema de **Funciones y Módulos en Python**, te recomiendo el siguiente video explicativo:

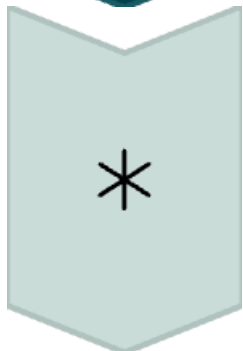


[https://www.youtube.com/watch?v=hWbD\\_6xhYe0](https://www.youtube.com/watch?v=hWbD_6xhYe0)

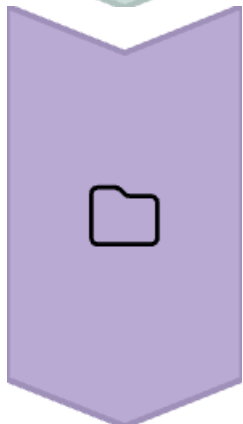
# Preguntas de Reflexión Final



¿Cómo pueden las funciones mejorar la modularidad y reutilización del código en tus proyectos de Python?



Explica la diferencia entre `*args` y `**kwargs` en la definición de funciones. ¿En qué situaciones sería útil cada uno?



Después de aprender sobre módulos y paquetes, ¿cómo organizarías un proyecto grande en Python para mantener un código limpio y mantenible?

