



Funciones y Modularidad

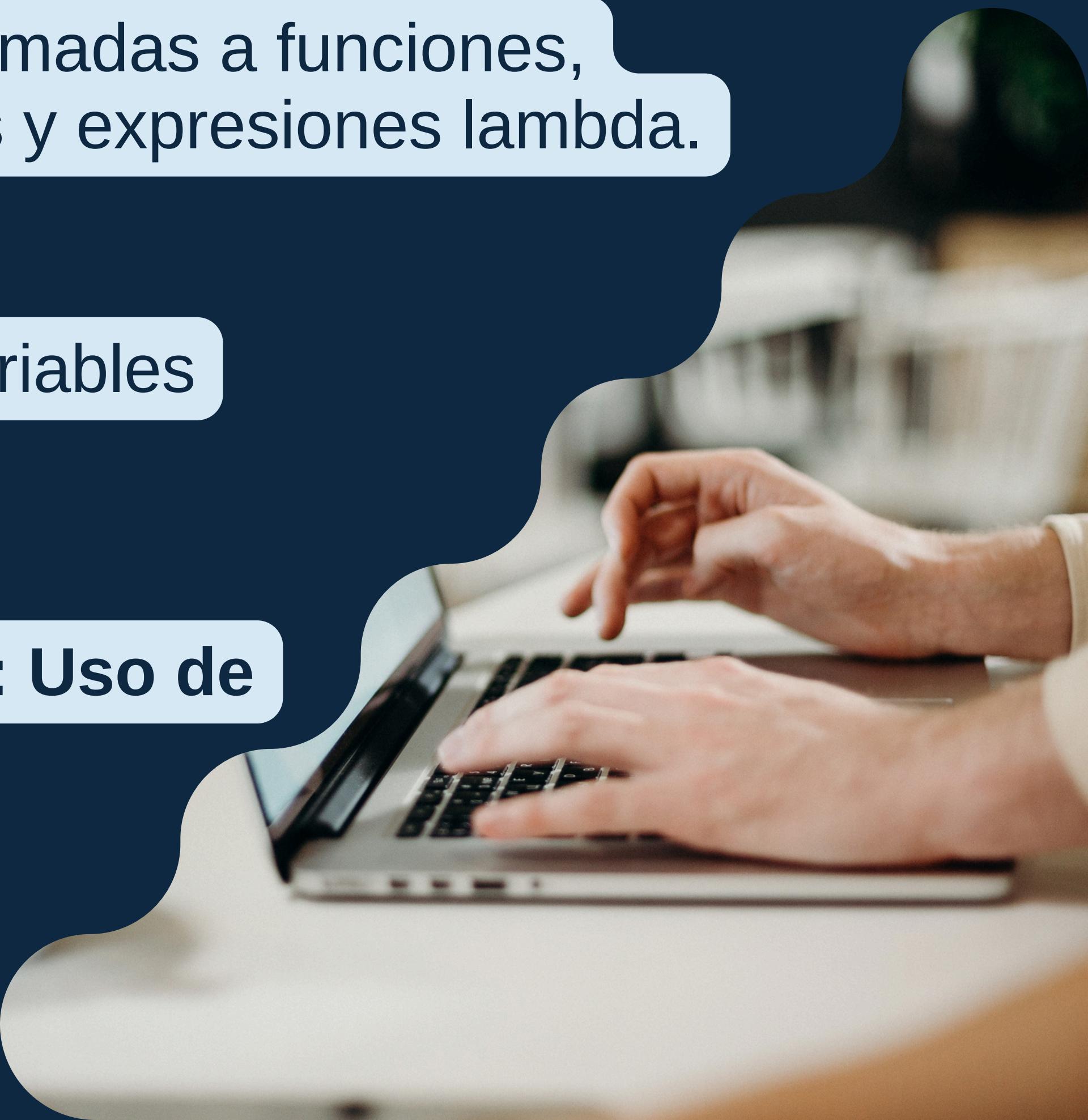
Grupo 4

Conceptos y consejos de
desarrollo

01 - Funciones: Definición, llamadas a funciones, parámetros, retorno de valores y expresiones lambda.

02 - Alcance de variables: Variables locales y globales.

03 - Importación de módulos: Uso de bibliotecas estándar



01 - Funciones: definición

Una función en Python es un bloque de código reutilizable que se define una vez y puede ser invocado múltiples veces.

Partes de una Función:

1. **def**: Esta palabra clave inicia la definición de una función.
2. **nombre_funcion**: Es el nombre de la función, que se usa para llamar a la función.
3. **(parametros)**: Son los argumentos que la función recibe. Pueden ser opcionales.
4. **"""Docstring opcional"""**: Una cadena opcional que describe brevemente lo que hace la función
6. **Bloque de código**: El cuerpo de la función, donde se define qué hace la función.
7. **return**: La instrucción que devuelve un valor opcional de la función.

python

```
def nombre_funcion(parametros):  
    """Docstring opcional"""  
    # Bloque de código  
    return valor_retorno
```

01 - Funciones: definición

¿Qué es una Llamada a una función?

Una llamada a una función es una instrucción que pide a Python que ejecute el código dentro de una función previamente definida.

¿Qué Sucede Durante una Llamada a la Función?

- 1. Paso de Control:** El control del programa se transfiere a la función.
- 2. Asignación de Argumentos:** Los argumentos pasados se asignan a los parámetros de la función.
- 3. Ejecución:** Se ejecuta el código dentro de la función.
- 4. Retorno:** Si hay una declaración return, se devuelve el valor correspondiente al punto de la llamada. Si no, se devuelve None por defecto.

Ejemplo: Llamada a la Función

```
python
def informacion_personal(nombre, edad=25, *hobbies, **detalles):
    print(f"Nombre: {nombre}")
    print(f"Edad: {edad}")
    print("Hobbies:", ", ".join(hobbies))
    for key, value in detalles.items():
        print(f"{key}: {value}")

# Llamada a la función con varios tipos de argumentos
informacion_personal("Carlos", 30, "leer", "viajar", ciudad="Barcelona", profesion="Ingeniero")
```

makefile **SALIDA**

```
Nombre: Carlos
Edad: 30
Hobbies: leer, viajar
ciudad: Barcelona
profesion: Ingeniero
```

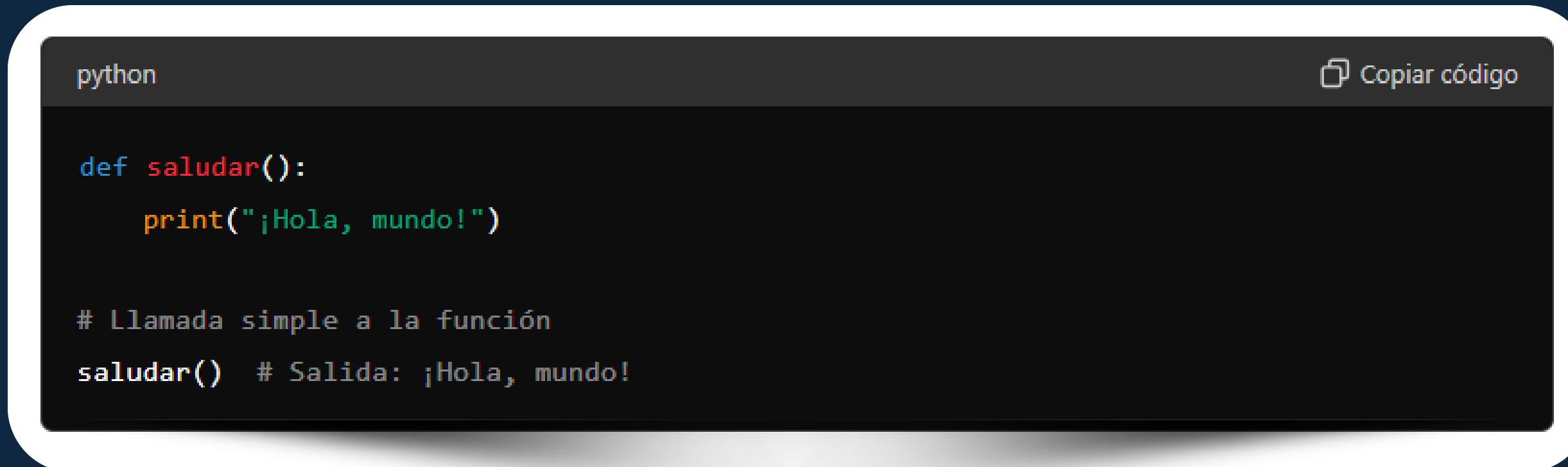
A yellow curved arrow points from the question '¿Qué es una llamada a una función?' down to the code example. Another yellow arrow points from the code example down to the resulting output.

01 - Funciones: llamada a funciones

Formas de Llamar a una Función

1. Llamada Simple

La forma más básica de llamar a una función es simplemente usar su nombre seguido de paréntesis. Si la función requiere parámetros, se pasan dentro de los paréntesis.



The screenshot shows a dark-themed code editor window with a white border. In the top left corner, it says "python". In the top right corner, there is a "Copiar código" button with a clipboard icon. The code itself is as follows:

```
python

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

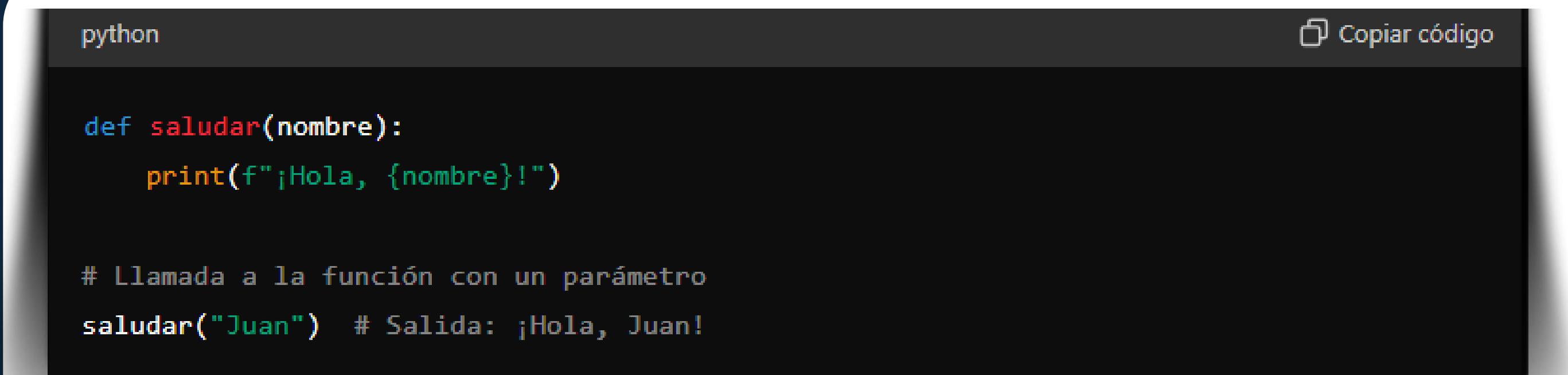
# Llamada simple a la función
saludar() # Salida: ¡Hola, mundo!
```

01 - Funciones: llamada a funciones

Formas de Llamar a una Función

2. Llamada con Parámetros

Si una función está definida para aceptar parámetros, debemos proporcionar esos valores cuando la llamamos.



The screenshot shows a code editor window with a dark theme. In the top left corner, it says "python". In the top right corner, there is a "Copiar código" button with a copy icon. The code itself is as follows:

```
python

def saludar(nombre):
    print(f"¡Hola, {nombre}!")

# Llamada a la función con un parámetro
saludar("Juan") # Salida: ¡Hola, Juan!
```

01 - Funciones: llamada a funciones

Formas de Llamar a una Función

3. Llamada con Múltiples Parámetro

Las funciones pueden aceptar múltiples parámetros. Estos se pasan en el mismo orden en que están definidos.

```
python

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

# Llamada a la función con múltiples parámetros
resultado = sumar(5, 3)
print(resultado) # Salida: 8
```

 Copiar código

01 - Funciones: llamada a funciones

Formas de Llamar a una Función

4. Llamada con Parámetros por Defecto

Las funciones pueden tener parámetros con valores por defecto. Si no se proporciona un argumento para un parámetro con un valor por defecto, se utiliza el valor predeterminado.

```
python Copiar código
def saludar(nombre="Mundo"):
    print(f"¡Hola, {nombre}!")

# Llamada a la función sin parámetro (usa el valor por defecto)
saludar() # Salida: ¡Hola, Mundo!

# Llamada a la función con un parámetro
saludar("Ana") # Salida: ¡Hola, Ana!
```

01 - Funciones: llamada a funciones

Formas de Llamar a una Función

5. Llamada con Parámetros Nombrados

Podemos pasar argumentos a una función usando el nombre del parámetro. Esto mejora la legibilidad y permite pasar argumentos en cualquier orden.

```
python Copiar código
def saludar(nombre, mensaje):
    print(f"{mensaje}, {nombre}!")

# Llamada a la función usando parámetros nombrados
saludar(mensaje="Buenos días", nombre="Luis") # Salida: Buenos días, Luis!
```

01 - Funciones: llamada a funciones

Formas de Llamar a una Función

6. Llamada con Número Arbitrario de Argumentos

Podemos definir funciones que acepten un número variable de argumentos utilizando ***args** para argumentos posicionales y ****kwargs** para argumentos nombrados.

python Ejemplo con *args:

```
def listar_nombres(*nombres):
    for nombre in nombres:
        print(nombre)

# Llamada a la función con múltiples argumentos
listar_nombres("Juan", "Ana", "Luis")
```

python Ejemplo con **kwargs:

```
def describir_persona(**características):
    for key, value in características.items():
        print(f'{key}: {value}')

# Llamada a la función con múltiples argumentos nombrados
describir_persona(nombre="María", edad=30, ciudad="Madrid")
```

01 - Funciones: sintaxis y buenas prácticas

Buenas prácticas al definir funciones:

- **Nombres descriptivos:** Elige nombres de funciones que describan claramente lo que hace la función. Por ejemplo, `calcular_suma` es mejor que `cs`.
- **Notación snake_case:** Los nombres de funciones deben seguir la notación `snake_case`, es decir, usar minúsculas y guiones bajos para separar las palabras. Por ejemplo, `calcular_promedio`.
- **Evitar nombres demasiado largos:** Aunque los nombres descriptivos son importantes, trata de evitar nombres excesivamente largos. **Busca un equilibrio entre claridad y concisión.**
- **No empezar con mayúsculas:** Por convención, los nombres de funciones en Python no deben empezar con mayúsculas. **Las mayúsculas iniciales se reservan para los nombres de clases.**
- **Evitar nombres reservados:** No uses nombres de funciones que sean palabras reservadas en Python, como `print`, `sum`, `len`, etc.



01 - Funciones: parámetros y argumentos

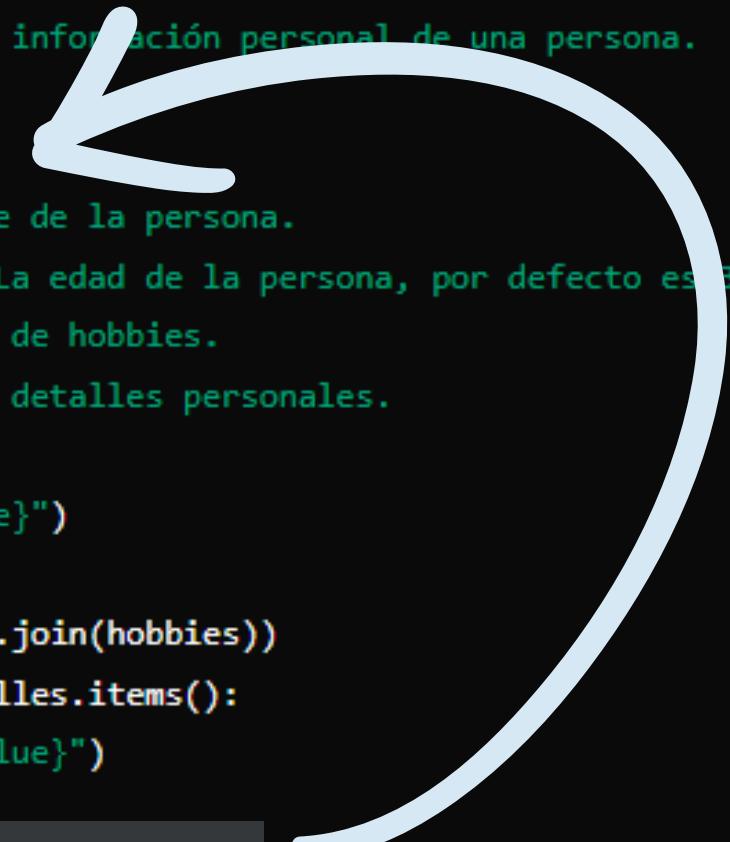
-> Parámetros

Variables en la definición de la función: Se utilizan al definir la función para especificar qué información necesita la función para realizar su tarea.

-> Argumentos

Valores pasados a la función en la llamada: Se utilizan al llamar a la función para proporcionar los valores específicos que deben ser procesados.

USO: Los parámetros se utilizan en la definición de la función para indicar qué información necesita la función, mientras que los argumentos se utilizan en la llamada a la función para proporcionar esa información.



```
python Copiar código

# Definición de la función con parámetros
def informacion_personal(nombre, edad=30, *hobbies, **detalles):
    """
    Función que imprime la información personal de una persona.

    Parámetros:
        nombre (str): El nombre de la persona.
        edad (int, opcional): La edad de la persona, por defecto es 30.
        hobbies (tuple): Lista de hobbies.
        detalles (dict): Otros detalles personales.
    """

    print(f"Nombre: {nombre}")
    print(f"Edad: {edad}")
    print("Hobbies:", ", ".join(hobbies))
    for key, value in detalles.items():
        print(f"{key}: {value}")

# Llamada a la función con argumentos
informacion_personal("Carlos", 35, "leer", "viajar", ciudad="Barcelona", profesion="Ingeniero")
```

01 - Funciones: retorno de valores

El retorno en una función de Python se refiere al valor que la función devuelve al final de su ejecución. La instrucción `return` se usa para enviar un valor de vuelta al lugar desde donde se llamó a la función.

Tipos de Retorno

Las funciones en Python pueden devolver diferentes tipos de valores:

- 1. Sin retorno (`None`):** La función no devuelve ningún valor explícitamente, por lo tanto, devuelve `None` por defecto.
- 2. Un valor único:** La función devuelve un único valor.
- 3. Múltiples valores:** La función devuelve varios valores como una tupla.
- 4. Objetos o estructuras complejas:** La función puede devolver listas, diccionarios, objetos personalizados, etc.



01 - Ejemplos con diferentes tipos de RETORNO(S)

1. Función sin Retorno (None)

La función saludar no tiene una instrucción return, por lo que devuelve None.

```
python

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

resultado = saludar()
print(resultado) # Salida: ¡Hola, mundo! seguido de None
```

2. Función con un Retorno de Valor Único

La función cuadrado devuelve el cuadrado del número pasado como argumento.

```
python

def cuadrado(numero):
    return numero ** 2

resultado = cuadrado(4)
print(resultado) # Salida: 16
```

3. Función con Múltiples Retornos

La función operaciones_basicas devuelve una tupla con cuatro valores, que representan los resultados de las operaciones aritméticas básicas.

```
python
Copiar código

def operaciones_basicas(a, b):
    suma = a + b
    resta = a - b
    multiplicacion = a * b
    division = a / b
    return suma, resta, multiplicacion, division

resultado = operaciones_basicas(10, 2)
print(resultado) # Salida: (12, 8, 20, 5.0)

# Acceso individual a los resultados
suma, resta, multiplicacion, division = operaciones_basicas(10, 2)
print(f"Suma: {suma}, Resta: {resta}, Multiplicación: {multiplicacion}, División: {division}")
```

01 - Ejemplos con diferentes tipos de RETORNO(S)

4. Función con Retorno de una Estructura Compleja

La función `crear_diccionario` devuelve un diccionario con los datos proporcionados.

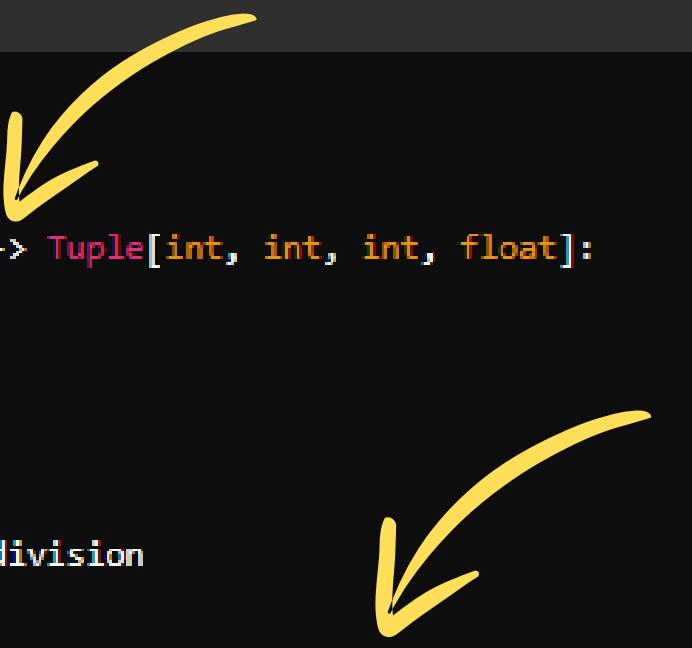
```
python

def crear_diccionario(nombre, edad, ciudad):
    return {"nombre": nombre, "edad": edad, "ciudad": ciudad}

resultado = crear_diccionario("Juan", 25, "Madrid")
print(resultado) # Salida: {'nombre': 'Juan', 'edad': 25, 'ciudad': 'Madrid'}
```

Definir Tipo de Dato Específico en el Retorno

Python es dinámico y no requiere especificar tipos de retorno. Sin embargo, se puede usar *anotaciones de tipo (type hints)* para indicar el tipo de retorno esperado, lo que mejora la legibilidad y ayuda a las herramientas de análisis estático.



```
python

from typing import Tuple, Dict

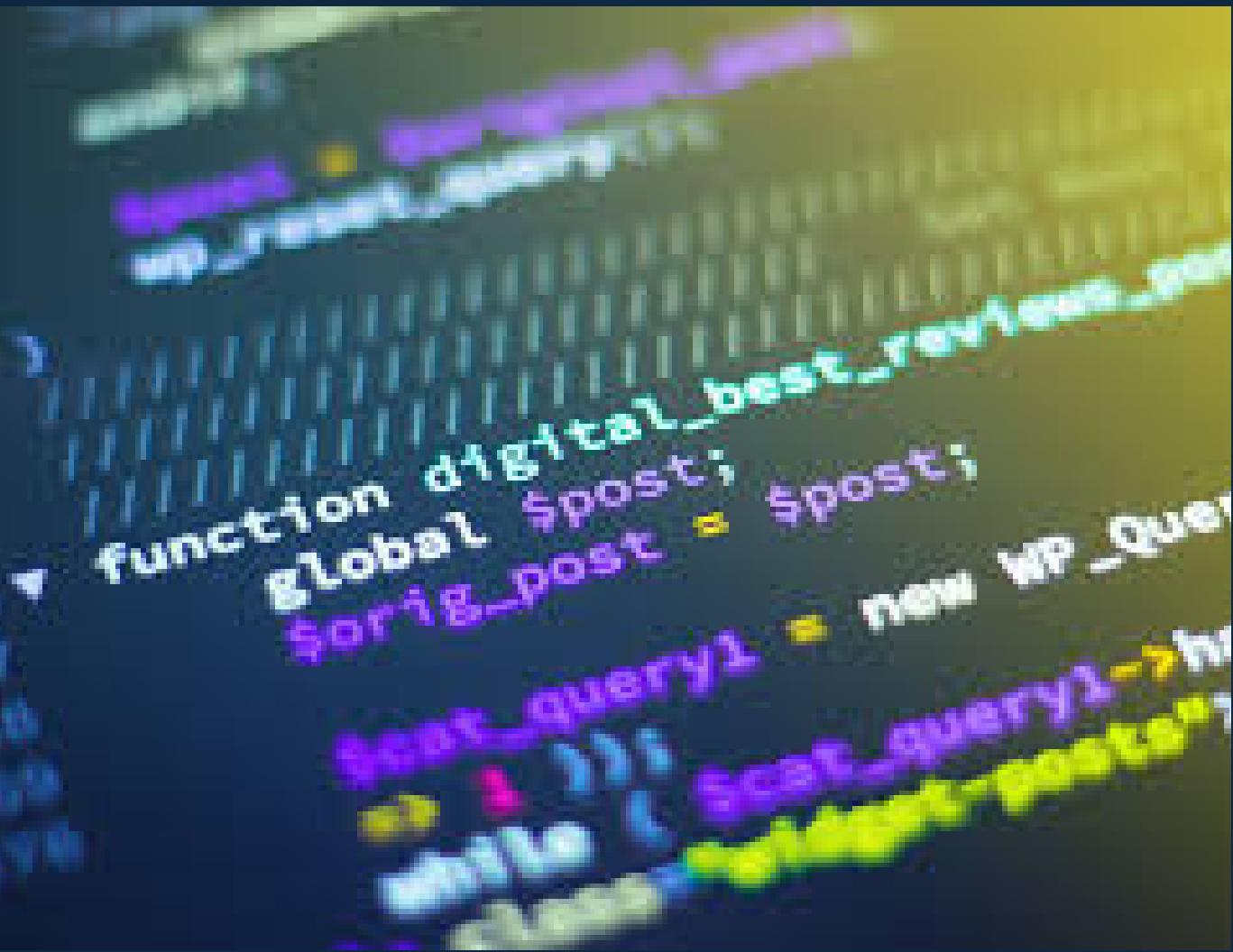
def operaciones_basicas(a: int, b: int) -> Tuple[int, int, int, float]:
    suma = a + b
    resta = a - b
    multiplicacion = a * b
    division = a / b
    return suma, resta, multiplicacion, division

def crear_diccionario(nombre: str, edad: int, ciudad: str) -> Dict[str, str]:
    return {"nombre": nombre, "edad": str(edad), "ciudad": ciudad}
```

01 - Expresiones Lambda

Expresiones Lambda

- Las expresiones lambda **se usan idealmente cuando necesitamos hacer algo simple** y estamos más interesados en hacer el trabajo rápidamente en lugar de nombrar formalmente la función. Las expresiones lambda también **se conocen como funciones anónimas**.
- Las expresiones lambda en Python **son una forma corta de declarar funciones pequeñas y anónimas** (no es necesario proporcionar un nombre para las funciones lambda).
- Las funciones Lambda se comportan como funciones normales declaradas con la palabra clave def. **Resultan útiles cuando se desea definir una función pequeña de forma concisa.** Pueden contener solo una expresión, por lo que no son las más adecuadas para funciones con instrucciones de flujo de control.



01 - Sintaxis de una función Lambda

Las funciones Lambda pueden tener cualquier número de argumentos, pero solo una expresión.

Código de ejemplo:

```
# Función Lambda para calcular el cuadrado de un número
square = lambda x: x ** 2
print(square(3)) # Resultado: 9

# Funcion tradicional para calcular el cuadrado de un numero
def square1(num):
    return num ** 2
print(square(5)) # Resultado: 25
```

En el ejemplo de lambda anterior, `lambda x: x ** 2` produce un objeto de función anónimo que se puede asociar con cualquier nombre. Entonces, asociamos el objeto de función con `square`. De ahora en adelante, podemos llamar al objeto `square` como cualquier función tradicional, por ejemplo, `square(10)`

01 - Ejemplo de uso función Lambda

Ejemplo para utilizar una función que calcule el área de un triángulo.

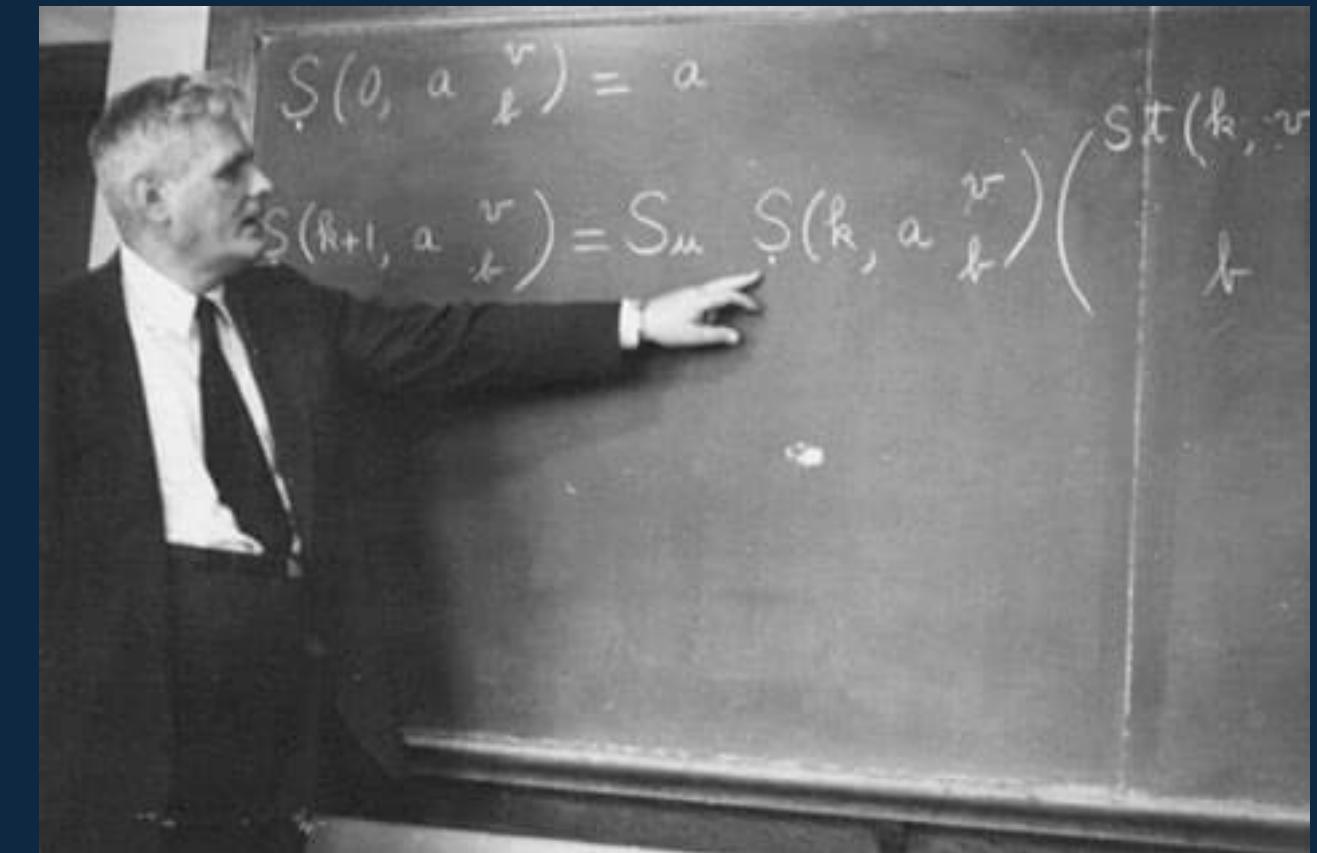
Como sabemos para calcular el área de un triángulo, se debe calcular la base por su altura y dividir entre 2.

Función tradicional:

```
#Función tradicional para calcular el área de un triángulo.  
def area_triangulo (base,altura):  
    return (base*altura)/2  
  
print (area_triangulo(3,5))
```

Función lambda:

```
#Función lambda para calcular el área de un triángulo.  
area_triangulo = lambda base,altura:(base*altura)/2  
  
print (area_triangulo(3,5))
```



El resultado para este ejemplo, en ambos casos seria:

7.5

01 - Sintaxis de una función Lambda

Es posible utilizar condicionales (if, elif, else), booleanos (True, False) y expresiones como (and, or, not) dentro de una expresión Lambda.

```
lambda_func = lambda x: True if x**2 >= 10 else False  
lambda_func(3) # Retorna False  
lambda_func(4) # Retorna True
```

Sin embargo, hay que tener en cuenta que en una función lambda no podremos utilizar por ejemplo bucles como for o while ya que llevan un flujo de control, para este cometido debemos crear una función normal.

Todo lo que hagamos con una función lambda se puede hacer con una función “normal” pero no al revés (funciones complejas con muchas líneas de código no se pueden simplificar con funciones lambda)

02 - Alcance de variables: variables locales

```
def my_function():
    local_var = "Soy una variable local"
    print(local_var)

my_function() # Mostrará: Soy una variable local
```

- Definidas y accesibles solo dentro de la función donde son utilizadas
- Cada función puede tener sus propias variables locales
- Su alcance está limitado al bloque de código en el que se declara. No se puede acceder a ella fuera de esa función.
- Útiles para cálculos temporarios o variables que no necesitan persistir en el tiempo

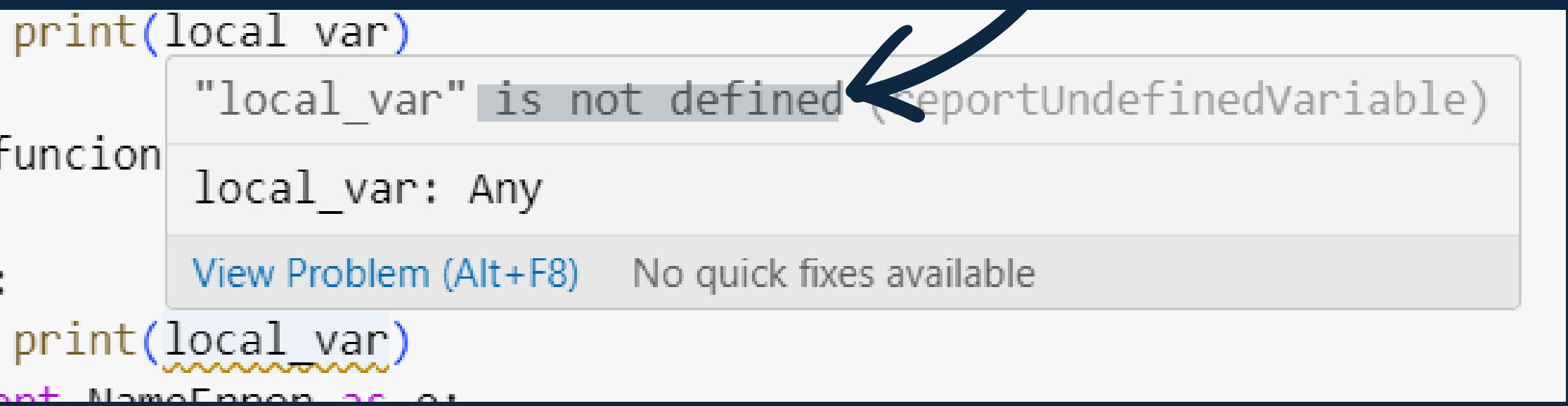
Alcance de una variable local

```
def mi_funcion():
    local_var = "Soy una variable local"
    print(local_var)

mi_funcion() # Mostrará: Soy una variable local

try:
    print(local_var)
except NameError as e:
    print(e) # Mostrará el mensaje de error ya que no podemos acceder a local_var fuera de mi_funcion
```

Soy una variable local
name 'local_var' is not defined



02 - Alcance de variables: variables globales

```
# Variable global
var_global = 'Soy una variable global'

def mi_funcion():
    print(var_global)

mi_funcion()      # Mostrará en pantalla: Soy una variable global
print(var_global) # Esto también mostrará en pantalla: Soy una variable global
```

- Scope global (accesible dentro del módulo)
- Definidas fuera de una función
- Útil para variables que deben ser utilizadas de forma global
- Utilizar palabra reservada “global” para modificar

Modificación de Variables Globales dentro de Funciones

python

```
a = 5 # a es una variable global

def mi_funcion():
    global a
    a = 10 # Modifica la variable global a
    print(a)

mi_funcion() # Imprime: 10
print(a)      # También imprime: 10
```

- Para modificar una variable global dentro de una función, se debe usar la palabra clave global.

CON USO DE “GLOBAL”

```
variable_global = 'David'

def modificar_var(var):
    global variable_global
    variable_global = 'Jonnathan'
    print(variable_global)

modificar_var(variable_global)
print(variable_global)
```

Jonnathan
Jonnathan

SIN USO DE “GLOBAL”

```
variable_global = 'David'

def modificar_var(var):
    variable_global = 'Jonnathan'
    print(variable_global)

modificar_var(variable_global)
print(variable_global)
```

Jonnathan
David

03 - Interacción entre Variables Locales y Globales

```
python

z = 30 # z es una variable global

def mi_funcion():
    z = 10 # z es una variable local dentro de esta función
    print(z)

mi_funcion() # Imprime: 10
print(z)      # Imprime: 30
```

- Si una variable global y una variable local tienen el mismo nombre, dentro de la función la variable local ocultará a la variable global.

Acceso de variables en diferentes directorios y ficheros

```
# modulo1.py
var_global = "Una variable global en el módulo 1"

# modulo2.py
import modulo1

def modificar_var_global():
    modulo1.var_global = 'He sido modificada en el módulo 2'

def acceso_var_global():
    print(modulo1.var_global)

acceso_var_global() # Mostrará: Una variable global en el módulo 1

modificar_var_global() # Modificará la variable global

acceso_var_global() # Mostrará: He sido modificada en el módulo 2
```



REL Path: import/directorio1/modulo1.py

03 - Importación de módulos: Uso de bibliotecas estándar

Definición de módulo y biblioteca

Módulo: es un archivo que contiene definiciones y declaraciones de código Python creándose con la extensión .py

1

2

3

4

```
variable_global = "David"
```

import modulo1
print(modulo1.variable_global)

def modificar_variable(var):
 ... modulo1.variable_global = "Jonnathan"
 ... print(modulo1.variable_global)
...
modificar_variable(modulo1.variable_global)

→ python
David
Jonnathan

03 - Importación de módulos: Uso de bibliotecas estándar

Definición de módulo y biblioteca

Biblioteca: es una colección de varios módulos

Biblioteca estándar: conjunto de módulos preinstalados que vienen con Python.

	Nombre	Edad	Ciudad
0	Ana	34	Madrid
1	Luis	28	Barcelona
2	Carlos	45	Valencia

```
import pandas

# Crear un DataFrame simple con Pandas
datos = {
    'Nombre': ['Ana', 'Luis', 'Carlos'],
    'Edad': [34, 28, 45],
    'Ciudad': ['Madrid', 'Barcelona', 'Valencia']
}

df = pandas.DataFrame(datos)

# Mostrar el DataFrame
print(df)
```

03 - Importación de módulos: Uso de bibliotecas estándar

Clasificación de bibliotecas

1. **Librerías Estándar:** Son las que vienen incluidas con Python y proporcionan funcionalidades básicas.
2. **Librerías de Terceros:** Estas son desarrolladas por la comunidad y se pueden instalar usando herramientas como **pip**. Ejemplos: **requests** para peticiones HTTP, **Beautiful Soup (bs4)** para web scraping, y **pandas** para análisis de datos.
3. **Librerías de Cálculo Numérico:** Como **numpy**.
4. **Librerías de Visualización de Datos:** Tales como **matplotlib** y **seaborn**
5. **Librerías de Machine Learning y Deep Learning:** Como **scikit-learn** para machine learning y **tensorflow** o **keras** para deep learning.

03 - Importación de módulos: Uso de bibliotecas estándar

Clasificación de bibliotecas

6. **Librerías de Procesamiento de Lenguaje Natural:** Como **nltk** y **spaCy**, que se utilizan para trabajar con lenguaje humano.
7. **Librerías de Inteligencia Artificial Explicable:** Que buscan proporcionar resultados óptimos en IA, utilizando diversas metodologías tecnológicas.
8. **Librerías de Web Scraping:** Que permiten extraer información de páginas web, como **scrapy**.
9. **Librerías de Generación de Números Aleatorios:** Para tareas que requieren generación de números aleatorios.
10. **Librerías de Procesamiento de Texto:** Como **re** para expresiones regulares y **textwrap** para el formateo de texto.

03 - Importación de módulos: Uso de bibliotecas estándar

Instalación de bibliotecas

Paso 1: Verificar si pip está instalado, que es el gestor de paquetes oficial para Python.

```
→ ~ pip --version  
pip 23.3.2 from /Lib
```

Paso 2: Instalar la biblioteca:

```
pip install nombre_de_la_biblioteca
```

Paso 3: Verificar la instalación.

```
import pandas  
print(pandas.__version__)
```

```
● → python  
2.2.2
```

03 - Importación de módulos: Uso de bibliotecas estándar.

Buenas prácticas

1. **Entornos Virtuales:** Permite aislar dependencias y evitar conflictos entre proyectos.
Utiliza venv o virtualenv.
2. **Utiliza PIP**
3. **Importa solo lo necesario:** importa solo las funciones o clases que necesitas.
4. **Documentación y Comunidad:** leer la documentación oficial y elige bibliotecas con buena comunidad de soporte.
5. **Actualiza regularmente:** Mantén tus bibliotecas actualizadas **pip install --upgrade**
6. **Versiones de Python:** Busca bibliotecas compatibles con la versión de Python que usas.

¡Gracias!

Equipo:

Erika Álvares

Ángel Martínez

Javier Gregoris Cano

Ingo Heredia

Alejandra Piñango

