Funciones y Módulos en Python

Cuando escribimos un programa, muchas veces nos enfrentamos a tareas repetitivas: mostrar saludos, hacer cálculos, validar datos, etc. Si copiamos y pegamos el mismo código varias veces, nuestro programa se vuelve largo, confuso y difícil de mantener.

Ahí es donde entran en juego las funciones y módulos.

por Kibernum Capacitación S.A.

Ventajas de usar funciones y módulos

- Permiten reutilizar código sin repetirlo
- Hacen el programa más ordenado y fácil de entender
- Ayudan a dividir un problema complejo en partes pequeñas y manejables
- Facilitan la colaboración entre varios desarrolladores
- Se convierten en herramientas personales o profesionales que puedes usar en muchos proyectos

¿Qué es una función?

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.

Sintaxis básica:

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

Ejemplo paso a paso:

```
def saludar():
    print("¡Hola desde la función!")

# Aquí la usamos o "llamamos":
saludar()
```

Parámetros y argumentos

¿Qué son?

- Parámetros: son las variables que se declaran en la función, como los espacios en blanco que rellenaremos.
- Argumentos: son los valores reales que se pasan cuando llamamos a la función.

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

saludar("Facundo")    # "Facundo" es el argumento
```

Retorno de valores (return)

A veces queremos que la función nos devuelva un resultado que podamos usar más adelante.

```
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.

Funciones preconstruidas y argumentos predeterminados

Funciones preconstruidas

Python trae muchas funciones ya listas, como:

```
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

Puedes establecer un valor por defecto:

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

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

Argumentos variables: *args y **kwargs

- *args: varios argumentos en forma de tupla (se accede por posición).
- **kwargs: varios argumentos en forma de diccionario (clave: valor).

```python
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 (funciones anónimas)

¿Qué son?

Son funciones simples y rápidas, que se definen en una sola línea. Se usan para operaciones cortas, especialmente como argumentos en otras funciones.

Sintaxis general:

```
lambda parametros: expresión
```

Esto equivale a:

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

Ejemplo básico:

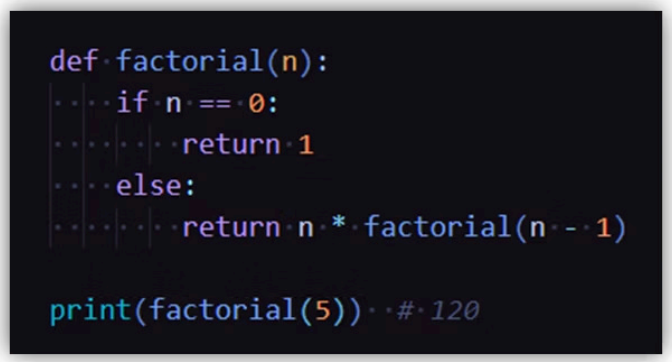
```
doble = lambda x: x * 2
print(doble(5))  # Resultado: 10
```

Otro ejemplo:

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

Funciones recursivas

Una función recursiva es aquella que se llama a sí misma. Es útil para tareas repetitivas como cálculos matemáticos complejos.



Modularización en Python: Beneficios e Inconvenientes

¿Qué es la modularización?

Es el proceso de dividir un programa grande en partes más pequeñas, llamadas módulos. Cada módulo tiene una función específica y puede ser reutilizado en otros programas.

En Python, un módulo es simplemente un archivo .py que contiene código (funciones, clases, variables) que se puede importar.

Beneficios de la modularización

Beneficio	¿Por qué es importante?
Reutilización de código	Puedes usar el mismo módulo en varios proyectos sin escribirlo de nuevo.
Mantenimiento más sencillo	Si hay un error, solo debes corregir una parte del código, no todo el programa.
Colaboración en equipo	Cada miembro puede trabajar en un módulo distinto sin pisarse con otros.
Legibilidad y orden	El código está mejor organizado, y cada módulo tiene una responsabilidad clara.
Escalabilidad	Es más fácil agregar nuevas funcionalidades sin romper lo existente.
Pruebas unitarias más fáciles	Puedes probar cada módulo por separado, facilitando la detección de errores.

Posibles inconvenientes y soluciones

Inconveniente	Solución o recomendación
Demasiados archivos	Usa una estructura de carpetas clara y documentada.
Dependencias circulares	Evita que los módulos se importen entre sí de forma cruzada.
Dificultad para principiantes	Enseñar de forma gradual con ejemplos simples.
Sobrecarga en proyectos pequeños	Modulariza solo cuando el proyecto crezca o si hay lógica que puede reutilizarse.
Gestión de nombres y rutas	Usa convenciones de nombres claras y absoluta preferencia por from ... import ...

¿Por qué usar módulos?

- Para dividir el código en partes
- Para no repetir código
- Para reutilizar funciones en distintos proyectos

Trabajando con módulos

Importar un módulo existente

```
import math

print(math.sqrt(25))  # 5.0
print(math.pi)        # 3.141592...
```

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 y paquetes

Librería estándar

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

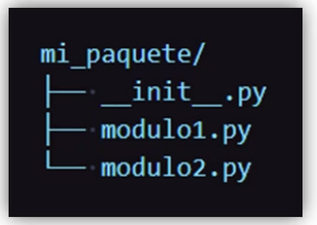
Módulo	¿Para qué sirve?
math	Operaciones matemáticas
random	Números aleatorios
datetime	Fechas y tiempos
os	Acceso a archivos y carpetas del SO

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.

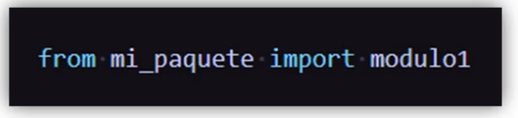
Ejemplo:




```
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:



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 π.
- 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 en Python

Docstrings (PEP 257)

Un docstring es una cadena de texto que documenta un módulo, clase o función. Se coloca justo debajo de la definición usando comillas triples """ """.

¿Para qué sirve?

- Explica qué hace el bloque de código.
- Se puede acceder con help() en Python.
- Mejora el mantenimiento y colaboración del código.

```python
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))
```

Recomendación: Siempre escribe un docstring claro y conciso. Para funciones simples, puede ser de una sola línea.

Convenciones de Nombrado (PEP 8)

Python recomienda estilos de nombre específicos según el tipo de elemento:

Tipo	Ejemplo	Estilo
Variables y funciones	calcular_total	snake_case
Clases	FacturaCliente	PascalCase
Constantes	PI = 3.14	UPPER_SNAKE
Módulos	mis_utilidades.py	snake_case
Paquetes	procesador_texto/	snake_case

Evita:

- Nombres genéricos como data, value, info.
- Usar mayúsculas para variables o funciones.
- Nombres muy largos o sin sentido.

Uso de Comentarios (también parte de PEP 8)

Los comentarios son útiles, pero deben usarse con criterio.

Tipos de comentarios:

- Comentario en línea (al final de una línea)

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

- Comentario de bloque (antes de un bloque de código)

```python
# Calcula el promedio de una lista de números
total = sum(lista)
promedio = total / len(lista)
```

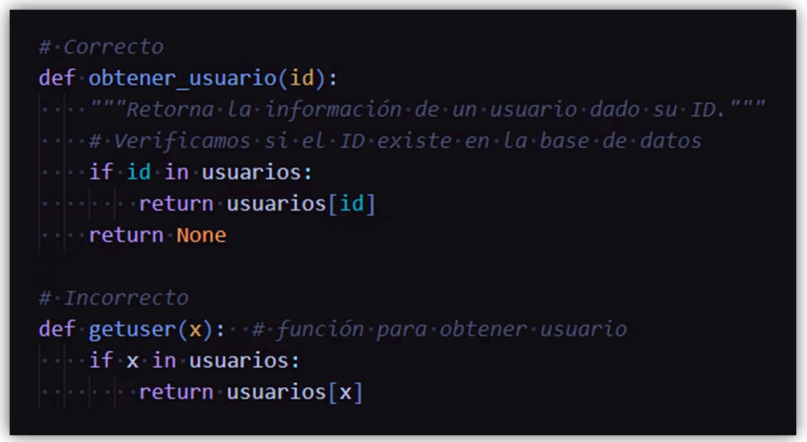
- Comentarios TODO o FIX (para tareas pendientes)

```python
# TODO: implementar validación de datos
# FIX: revisar error de división por cero
```

Buenas prácticas:

- Sé claro y específico.
- No repitas lo que el código ya dice por sí solo.
- Mantén los comentarios actualizados si el código cambia.

Mini resumen visual:



```python
# Correcto
def obtener_usuario(id):
    """Retorna la información de un usuario dado su ID."""
    #Verificamos si el ID existe en la base de datos
    if id in usuarios:
        return usuarios[id]
    return None

# Incorrecto
def getuser(x):  #función para obtener usuario
    if x in usuarios:
        return usuarios[x]
```

Sección de Actividad Práctica Guiada

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 a paso detallado con código

Paso 1: Crear el módulo saludos.py

```python
def saludar(nombre):
    return f"¡Hola, {nombre}! Bienvenido a Python."

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

Paso 2: Crear el archivo principal main.py

```python
import saludos

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

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

Paso 3: Ejecutar el programa

```
python main.py
```

Reflexión:

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

Actividad propuesta para el estudiante

Ahora que has visto un ejemplo guiado, realiza un ejercicio similar, pero con un **nuevo caso de uso**.

Actividad: 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 para la entrega:

- Crear el módulo operaciones.py con las funciones correspondientes.
- Crear el archivo main.py que importe y use las funciones del módulo.
- Insertar **pantallazos del código en el editor y de la ejecución en consola.**
- Asegurarse de incluir comentarios y docstrings en cada función.
- Usar input() para obtener los números del usuario y print() para mostrar los resultados.