



Tema 26

Programación modular

Diseño de funciones, recursividad y uso de librerías

1.1 Introducción a la modularidad



- ◆ Divide proyectos en **módulos independientes**
 - ◆ Facilita el mantenimiento, pruebas y colaboración
 - ◆ Aumenta la claridad del código y reduce errores
-  *Anécdota:* recursión infinita por falta de caso base → crash del programa
-  Controlar el flujo es clave

1.2 Fundamentos de módulos

 Un módulo es una **unidad funcional de código** con:

- Interfaz clara (parámetros, funciones públicas)
- Lógica interna encapsulada

Principios:

-  Alta cohesión → propósito único
-  Bajo acoplamiento → independencia entre módulos

modulo_usuario.py

```
def registrar(usuario):  
    return True
```

1.3 Diseño de funciones

✓ Buenas prácticas:

- Aplicar **DRY** y **SRP**: funciones que hacen solo una cosa
- Máx. 2–3 parámetros → agrupar si son más
- Preferir **variables locales**
- Refactorizar funciones grandes → subfunciones

📁 Organización sugerida:

- `controllers/` → lógica principal
- `services/` → funciones de negocio
- `utils/` → utilidades

1.4 Recursividad

 Técnica en la que una función se **llama a sí misma**

def factorial(n):

return 1 if n == 0 else n * factorial(n - 1)

 **Componentes:**

- Caso base → corta la recursión
- Caso recursivo → reduce el problema

1.4 Recursividad (II)

Recursiva vs. Iterativa

Aspecto	Recursiva	Iterativa
Claridad	Alta en árboles/backtracking	Alta en bucles simples
Eficiencia	Menor (uso de pila)	Mayor (uso de bucles)
Aplicaciones	Fibonacci, Hanoi, árboles	Contadores, listas

 Optimización: **memoización** para evitar cálculos repetidos

1.5 Librerías y reutilización

 Tipos:

- Integradas (estándar)
- Externas (instalables con pip, npm...)

 Buenas prácticas:

- `README.md` , `docstrings` , comentarios útiles
- Versionado semántico: `mayor.menor.parche` (ej. `1.2.0`)

Empaquetado:

- Python → `setup.py` , `__init__.py` , PIP
- JS → `package.json` , NPM

 Seguridad: evitar dependencias innecesarias

1.6 Modularidad en sistemas reales

Aplicación real:

- Capas lógicas: presentación, lógica, datos
- Microservicios → cada uno con una función clara
- REST / APIs → comunicación modular
- Tests unitarios: `pytest`, `Jest`, `unittest`




 Modularidad = arquitectura mantenible y escalable

1.7 Diseño algorítmico modular

 Aplicar modularidad a la resolución de problemas:

- Dividir el algoritmo en **subfunciones**
- Separar funciones auxiliares
- Apoyarse en pseudocódigo o diagramas

Ejemplos de técnicas:

-  Divide y vencerás (MergeSort)
-  Backtracking (Sudoku)
-  Greedy (Cambio óptimo)

1.8 Pruebas y verificación de funciones

 Probar cada función por separado (testing unitario)


En Python:

```
def test_suma():  
    assert suma(2, 3) == 5
```

En JavaScript (Jest):

```
test('suma', () => {  
    expect(suma(2, 3)).toBe(5);  
});
```

 Casos típicos, límites y errores

 Modularidad = pruebas más simples y efectivas

1.9 Conclusión científica

- ◆ La **modularidad** es una habilidad clave en desarrollo profesional
- ◆ La **recursividad**, bien utilizada, es una herramienta poderosa
- ◆ Funciones bien diseñadas permiten reutilizar, probar y mantener el código
- 🚀 Pensar modularmente = pensar como desarrollador profesional