Funciones, Modularización y Abstracción

Fundamentos de Programación / Algoritmos y Programación I
Semana 4 · Clase Teórica · Prof. Servetto

- Funciones: definición, parámetros, valores de retorno, alcance de variables (scope).
- Programación modular: descomposición de problemas; criterios de cohesión y acoplamiento.
- Abstracción funcional: diseño top-down, contratos de interfaz.
- [Holístico] Pila de ejecución (stack frames): introducción; compiladores e intérpretes (primeras diferencias); Python como lenguaje interpretado (bytecode, CPython).

Agenda de la clase

1. **¿Por qué funciones?**
   Motivación: repetición, legibilidad, carga cognitiva
2. **Anatomía de una función**
   def, parámetros, return, docstring, paso de argumentos
3. **Alcance de variables (scope)**
   Local vs. global, shadowing, regla LEGB
4. **Programación modular**
   Diseño top-down, cohesión, acoplamiento, refactorización
5. **Abstracción funcional**
   Contratos de interfaz, pre/postcondiciones
6. **Pila de ejecución (call stack)**
   Stack frames, stack trace, simulación
7. **Compiladores e intérpretes**
   CPython, bytecode, módulo dis

1. ¿Por qué funciones?

Tres problemas del código monolítico

Repetición

Copiar y pegar código replica también los errores.

Legibilidad

200 líneas sin estructura es como un libro sin capítulos.

Complejidad cognitiva

La memoria de trabajo humana maneja 5–9 elementos (Miller, 1956).

Las funciones resuelven los tres problemas de una sola vez: encapsulan una tarea, reciben datos y devuelven un resultado.

2. Anatomía de una función en Python

```
def calcular_area(base, altura):
    '''Calcula el área de un
    rectángulo.'''
    area = base * altura
    return area
```

- Palabra reservada def
- Nombre descriptivo (verbo)
- Parámetros formales
- Docstring (contrato)
- Valor de retorno

Parámetro (formal): nombre en la **definición**. | **Argumento** (real): valor concreto en la **invocación**.

Invocación y flujo de ejecución

Programa principal

```
b = float(input('Base: '))
h = float(input('Altura: '))

resultado = calcular_area(b, h)
    # ejecución suspendida

print('El área es:', resultado)
```

calcular_area(base, altura)

```
base ← b  |  altura ← h
area = base * altura
return area
```

① Invocación: el programa suspende su ejecución y transfiere el control a la función.

② Retorno: la función devuelve el resultado al punto exacto de la invocación.

Funciones con y sin valor de retorno

Con retorno

`return valor`

Calcula y devuelve un resultado.

Se invoca en una asignación.

```
area = calcular_area(5, 3)
```

Sin retorno

`procedimiento`

Realiza una acción (imprimir).

Devuelve None implícitamente.

```
mostrar_resultado(area)
```

⚠️ **Error frecuente: print no es return**

print() muestra texto en pantalla. return entrega un valor al invocador.

Confundirlos causa: «mi función imprime bien pero resultado vale None».

3. Alcance de variables (scope)

Tipo	Se crea	Visible en	Se destruye
Local	Dentro de una función (incluye parámetros)	Solo dentro de esa función	Al terminar la función
Global	En el nivel principal (fuera de funciones)	Todo el programa (lectura desde funciones)	Al terminar el programa

Regla LEGB: cadena de búsqueda de Python

L — **Local**
Función actual

→

E — **Enclosing**
Funciones contenedoras

→

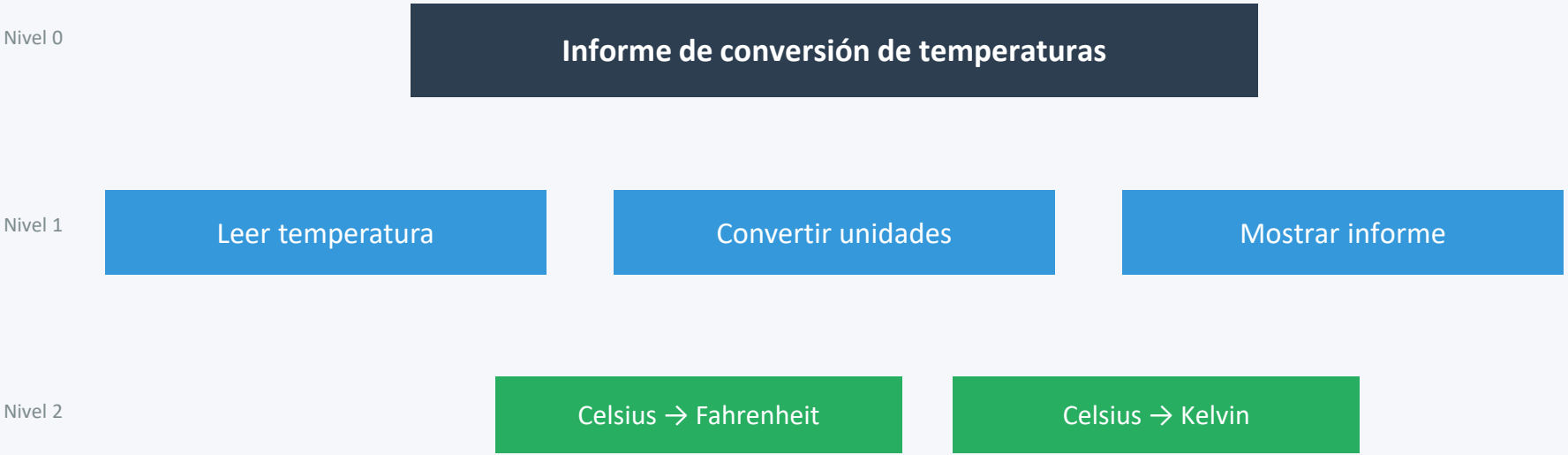
G — **Global**
Módulo principal

→

B — **Built-in**
print, len, int…

4. Programación modular: diseño top-down

Se parte del problema completo y se descompone en subproblemas más simples.



Celsius→Fahrenheit y Celsius→Kelvin son genuinamente independientes (fórmulas distintas, sin estado compartido). Comparar con máx/mín/promedio que se resuelven en un único recorrido (Semana 3).

Cohesión y acoplamiento

Alta cohesión ✓

Cada función realiza una sola tarea bien definida. Todos sus elementos contribuyen a esa tarea.

Indicador: si la docstring necesita «y» para describir qué hace, hace demasiado.

Bajo acoplamiento ✓

Las funciones se comunican exclusivamente a través de parámetros y valores de retorno.

Indicador: si modificar el interior de una función rompe otras, el acoplamiento es alto.

Objetivo: ALTA cohesión + BAJO acoplamiento

5. Abstracción funcional: el contrato

Usar una función sin conocer su implementación interna.

Precondición

¿Qué debe ser verdadero ANTES de invocar?

`base > 0, altura > 0`

Función

(caja negra)

Postcondición

¿Qué será verdadero DESPUÉS de ejecutar?

`retorno == base × altura`

La docstring documenta el contrato; el cuerpo lo implementa

> Literate programming con funciones: cada función tiene su propia sección declarativa (docstring). Un programa modular se lee en dos niveles: qué hace el programa y qué hace cada parte.

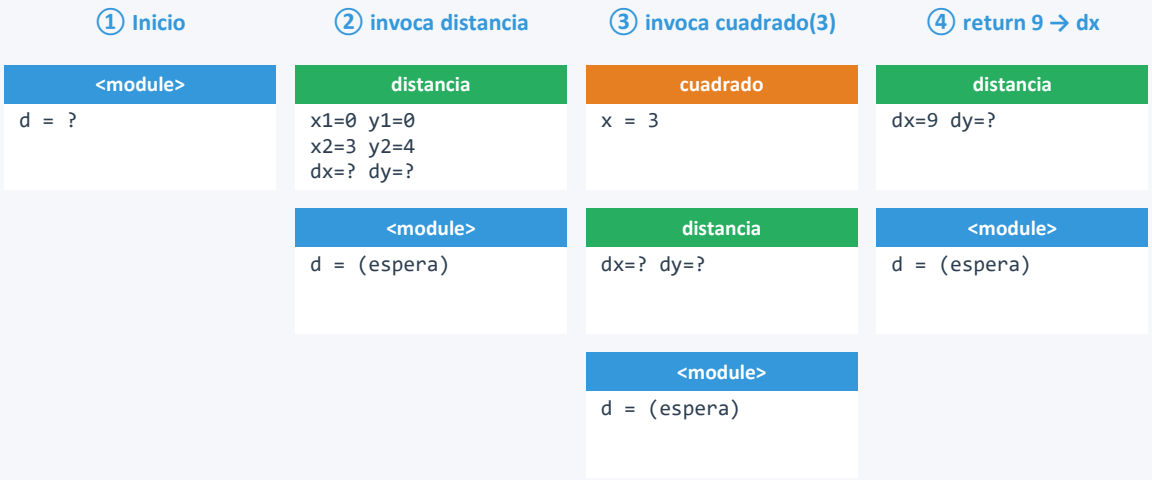
6. Pila de ejecución (call stack) — parte 1

Cada invocación crea un stack frame; cada return lo desapila (LIFO).

```
def cuadrado(x):
    return x * x


def distancia(x1,y1,x2,y2):
    dx = cuadrado(x2 - x1)
    dy = cuadrado(y2 - y1)
    return (dx+dy) ** 0.5


d = distancia(0, 0, 3, 4)
print('Distancia:', d)
```



① Solo <module> en la pila. ② Se invoca distancia: nuevo frame con parámetros. ③ Dentro de distancia se invoca cuadrado(3): tercer frame. ④ cuadrado retorna 9, su frame se desapila; dx recibe el valor.

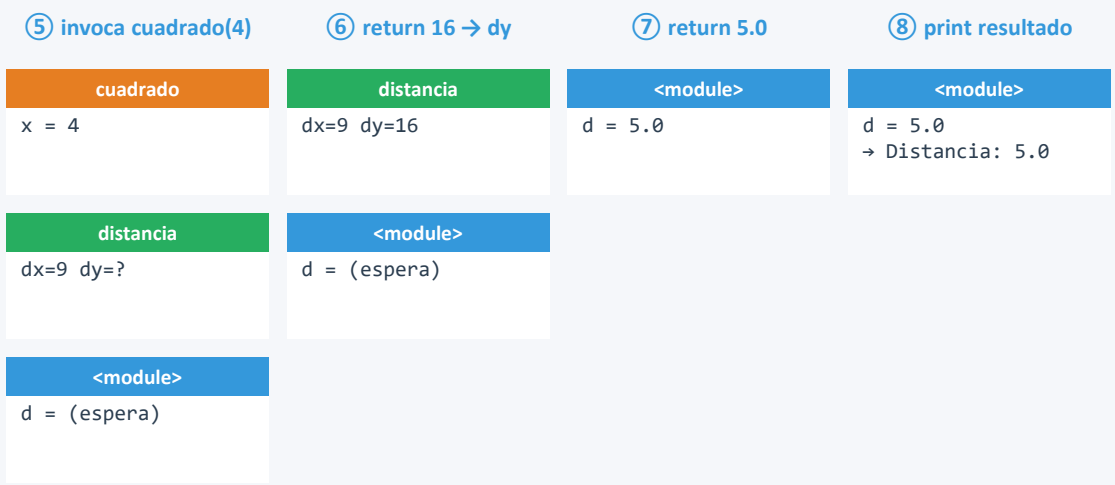
6. Pila de ejecución (call stack) — parte 2

Continuación: segunda invocación a cuadrado y retorno final.

```python
def cuadrado(x):
    return x * x

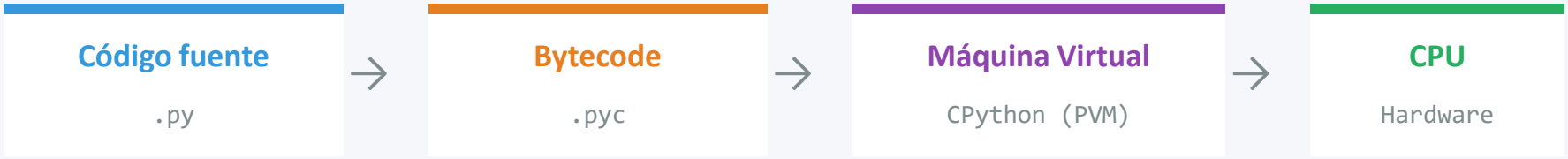

def distancia(x1,y1,x2,y2):
    dx = cuadrado(x2 - x1)
    dy = cuadrado(y2 - y1)
    return (dx+dy) ** 0.5


d = distancia(0, 0, 3, 4)
print('Distancia:', d)
```



Stack trace: cuando ocurre un error, Python muestra la cadena de invocaciones (contenido de la pila). Se lee **de abajo hacia arriba**: la última línea dice *qué* error ocurrió; las anteriores dicen *dónde* (en qué función y en qué línea).

7. Compiladores e intérpretes



	Compilación	Interpretación
Proceso	Todo el código → ejecutable (una vez)	Línea a línea en tiempo de ejecución
Errores	Antes de ejecutar	Durante la ejecución
Velocidad	Más rápida (código nativo)	Más lenta (traducción en vivo)
Ejemplos	*C, C++, Rust, Go*	*Python, JavaScript, Ruby*

Python (CPython) es un caso híbrido: compila a bytecode intermedio + la PVM lo interpreta instrucción a instrucción.

Modelo de programa modular

SECCIÓN DECLARATIVA

Docstring general: problema, casos de prueba, recursos

DEFINICIÓN DE FUNCIONES

Cada función con su docstring (contrato), parámetros y return

Ordenadas de menor a mayor nivel de abstracción

SECCIÓN ALGORÍTMICA (Programa Principal)

1. PRÓLOGO

Presentación

2. RESOLUCIÓN

Invocación de funciones

3. EPÍLOGO

Resultados

¿Cuándo crear una función?

Reutilización | **Legibilidad** | **Complejidad** | **Testabilidad** | **Modificabilidad**

Conexiones holísticas



Arquitectura de Computadoras

Las funciones se implementan con instrucciones CALL/RET. El Stack Pointer (SP) gestiona la pila en hardware. El bytecode de CPython opera como una ALU virtual.



Sistema Operativo

Cada proceso tiene su propia pila (1–8 MB). Una recursión infinita causa stack overflow: desbordamiento que el SO detecta y termina el proceso.



Compiladores e Intérpretes

CPython compila a bytecode (.pyc) y la PVM lo interpreta. El módulo dis permite inspeccionar el bytecode: LOAD_FAST, BINARY_ADD, RETURN_VALUE.

Próximas semanas: Búsqueda lineal y binaria (S5) · Referencias y mutabilidad (S7) · Recursividad (S8)

Ideas clave de la clase

- Una función encapsula una tarea: recibe parámetros, devuelve un resultado.
- Las variables locales son invisibles desde fuera (scope).
- Diseño top-down: descomponer antes de codificar.
- Alta cohesión + bajo acoplamiento = buena modularización.
- La docstring es el contrato de la función.
- La pila de ejecución gestiona las invocaciones (LIFO).