USC
UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

6. FUNCIONES

Programación I

Grado en Inteligencia Artificial

Curso 2022/2023

Contenidos

- Definición de funciones: parámetros y valores de retorno
- Variables locales y variables globales
- Mecanismo de paso de parámetros
- Recursividad
- Diseño modular

Definición de funciones

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


print(cuadrado(2))
a = 1 + cuadrado(3)
print(cuadrado(a * 3))
```

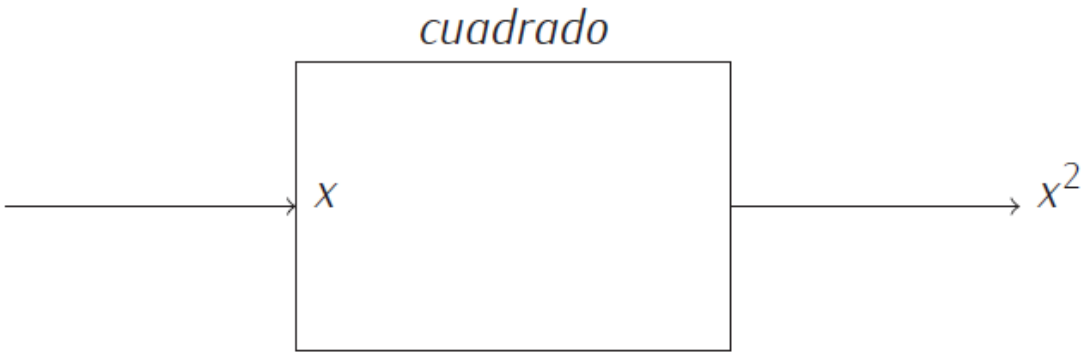
Cabecera de la función
x es el *parámetro formal*

Cuerpo de la función
x^2 es el *valor de retorno*

Llamadas a la función
2, 3 y 3·a son los *parámetros reales* o *argumentos*

Definición de funciones

- Las funciones deben recibir nombres cortos pero *significativos*
- Se pueden ver como *cajas* que transforman un dato de entrada en otro dato de salida



Ejemplos

```
from math import sin

def xsin(x):
    return x * sin(x)

r = xsin(1.571)
print(r)
```

```
def cubo(x):
    return x ** 3

y = 1
print(cubo(y))
```

Ejemplo con condicionales

```
def es_menor(edad):
    if edad < 18:
        return True
    else:
        return False

print(es_menor(23))
```

```
def es_menor(edad):
    if edad < 18:
        return True
    return False

print(es_menor(23))
```

```
def es_menor(edad):
    if edad < 18:
        res = True
    else:
        res = False
    return res

print(es_menor(23))
```

```
def es_menor(edad):
    return edad < 18

print(es_menor(23))
```

Ejemplo con bucles

```
def es_perfecto(n):
    sumatorio = 0
    for i in range(1, n):
        if n % i == 0:
            sumatorio += i
    if sumatorio == n:
        return True
    else:
        return False
```

```
def es_perfecto(n):
    sumatorio = 0
    for i in range(1, n):
        if n % i == 0:
            sumatorio += i
    return sumatorio == n
```

Ejemplo: paso de múltiples datos

```
def area_rect(alto, ancho):
    return alto * ancho

print(area_rect(3, 4))
```

```
def maximo(lista):
    if len(lista) > 0:
        c = lista[0]
        for e in lista:
            if e > c: c = e
    else: c = None
  return c

print( maximo( [1, 2, 3] ) )
```

Legibilidad del código

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

mipunto = []
for i in range(3):
    coord = float(input())
    mipunto.append(coord)

def suma_cuad(punto):
    suma = 0
    for c in punto:
        suma += cuad(c)
    return suma

s = suma_cuad(mipunto)
from math import sqrt
print('Distancia:', sqrt(s))
```

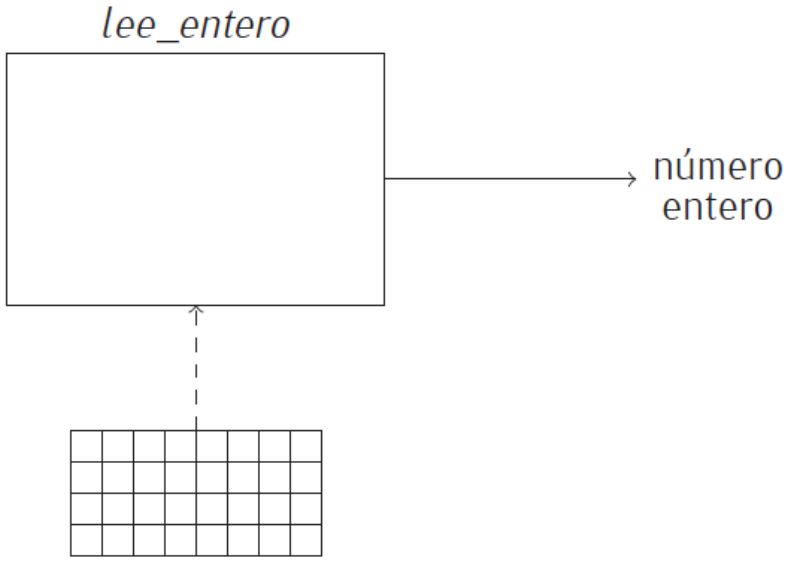
```
from math import sqrt

def cuad(x):
    return x**2

def suma_cuad(punto):
    suma = 0
    for c in punto:
        suma += cuad(c)
    return suma

mipunto = []
for i in range(3):
    coord = float(input())
    mipunto.append(coord)
s = suma_cuad(mipunto)
print('Distancia:', sqrt(s))
```

Organización recomendada:

1. Importaciones
2. Definición de funciones
3. Programa principal

Funciones sin parámetros

```
def lee_entero():
    return int(input())

a = lee_entero()
```



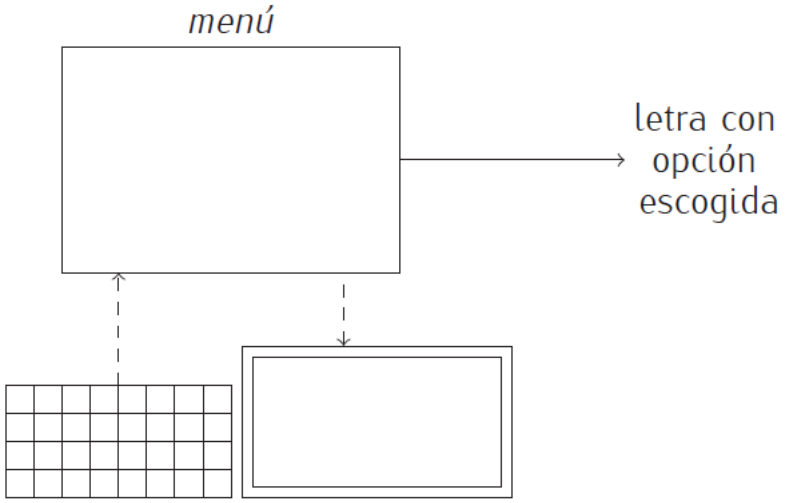
¿Qué alternativa es mejor?

```
def es_par(n):
    return n % 2 == 0
```

```
def es_par():
    n = int(input('Número? '))
    return n % 2 == 0
```

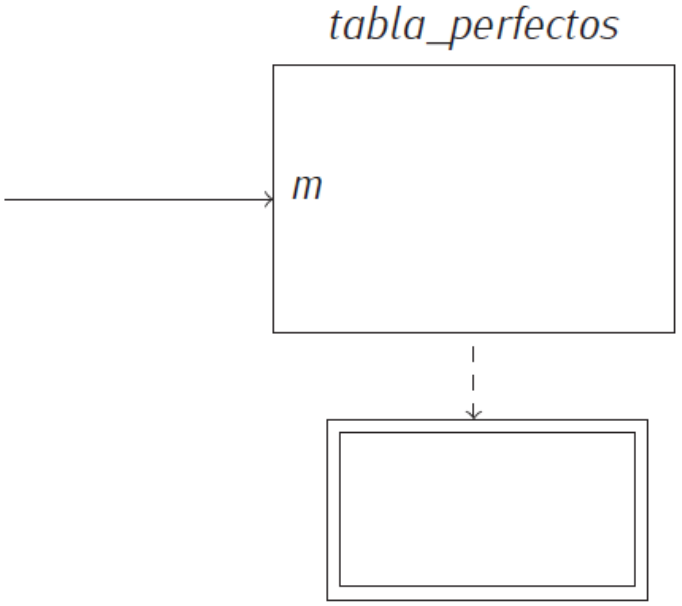
Gestión de menús

```
def menu():
    op = ''
    while not ('a' <= op <= 'c'):
        print('a) Ingresar')
        print('b) Extraer')
        print('c) Saldo')
        op = input('Su opción? ')
        if not ('a' <= op <= 'c'):
            print('Opción incorrecta')
    return op

accion = menu()
```



Procedimientos

```
def es_perfecto(n):
    suma = 0
    for i in range(1, n):
        if n % i == 0:
            suma += i
    return suma == n

def tabla_perfectos(m):
    for i in range(1, m+1):
        if es_perfecto(i):
            print(i, 'es perfecto')

numero = int(input('Dame un nº: '))
tabla_perfectos(numero)
```



Devolución de varios valores

```
def minmax(a, b, c):
    if a < b:
        if a < c: min = a
        else: min = c
    else:
        if b < c: min = b
        else: min = c
    if a > b:
        if a > c: max = a
        else: max = c
    else:
        if b > c: max = b
        else: max = c
    return [min, max]

a = minmax(10, 2, 5)
print('Mínimo:', a[0])
print('Máximo:', a[1])
```

Lista

```
def minmax(a, b, c):
    if a < b:
        if a < c: min = a
        else: min = c
    else:
        if b < c: min = b
        else: min = c
    if a > b:
        if a > c: max = a
        else: max = c
    else:
        if b > c: max = b
        else: max = c
    return min, max

mínimo, máximo = minmax(10, 2, 5)
print('Mínimo:', mínimo)
print('Máximo:', máximo)
```

Tupla

Variables locales y globales

- Las funciones pueden definir **variables locales** al margen del programa principal

```
from math import sqrt

def area_triangulo(a, b, c):
    s = (a + b + c) / 2
    return sqrt(s*(s-a)*(s-b)*(s-c))

print(area_triangulo(1, 3, 2.5))
print(s) #Desconocida aquí!
print(a) #Otro error!
```

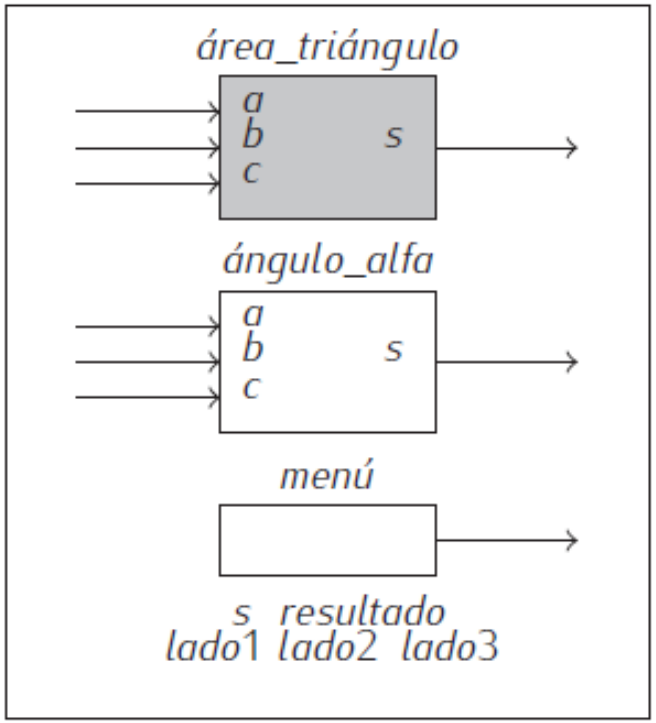
```
from math import sqrt

def area_triangulo(a, b, c):
    s = (a + b + c) / 2
    return sqrt(s*(s-a)*(s-b)*(s-c))

s = 4
print(area_triangulo(s-1, s, s+1))
print(s) #No causa problema
```

Ámbito de las variables

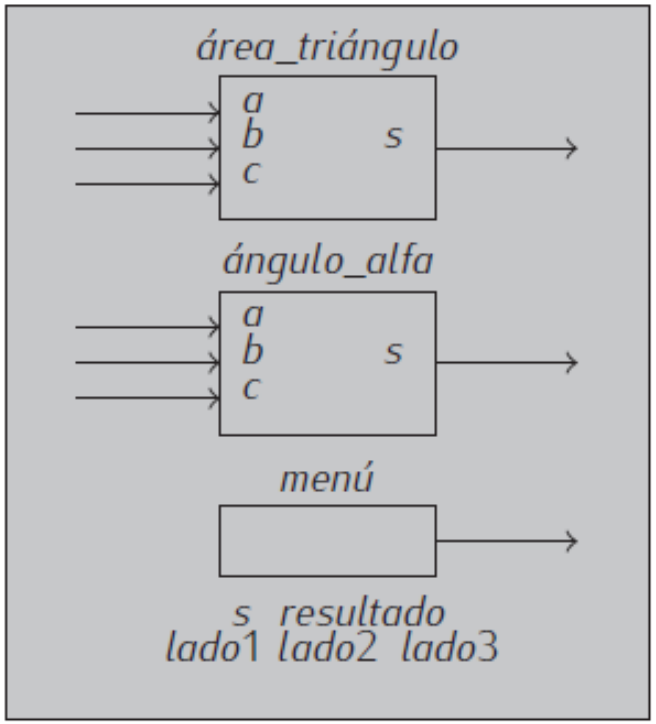
- Cada función define un **ámbito local** propio
- Las **variables locales** son sólo visibles en el cuerpo de la función que las define
- Las líneas que no son parte del cuerpo de una función tienen **ámbito global**
- Las **variables globales** son accesibles desde cualquier punto del código
- Si el identificador de una variable local coincide con el de una variable global, ésta **queda oculta**

Ejemplo: 289_area_y_angulo.py

- Ámbito local



- Ámbito global



Ejemplo

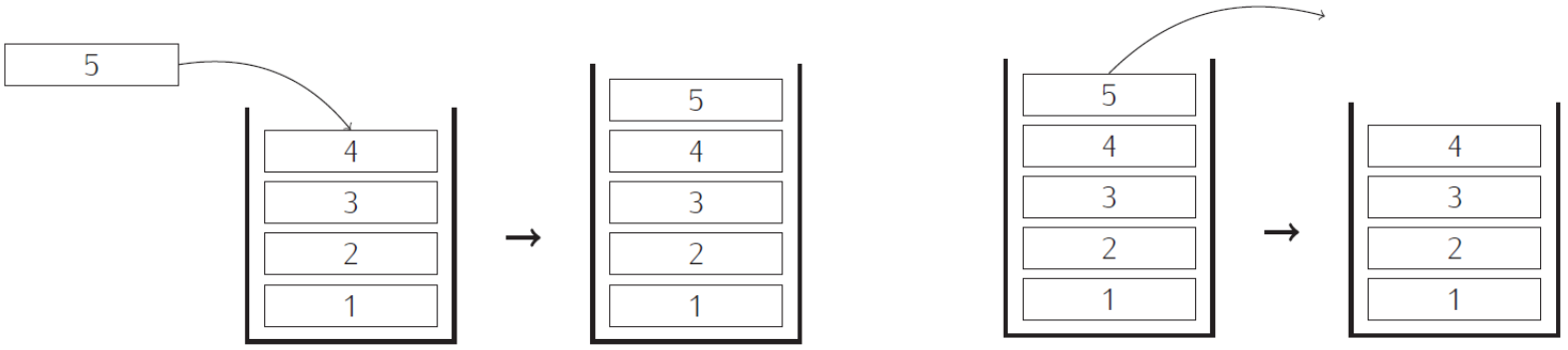
```
def unicos(lista):
    res = []
    for e in lista:
        if e not in res:
            res.append(e)
    return res

lista = unicos([1, 2, 1, 3, 2])
print(lista)
```

Mecanismo de paso de parámetros

- Cuando una función **invoca** a otra, su ejecución se *suspende* a la espera de que la segunda concluya su trabajo
- Si coincide el nombre de una **variable local** con el de una **global**, ésta queda oculta durante la ejecución de la función
- Se necesita *mecanismo* para llevar cuenta de las funciones en suspenso y restaurar los valores asociados a un identificador

Pila de llamadas a función

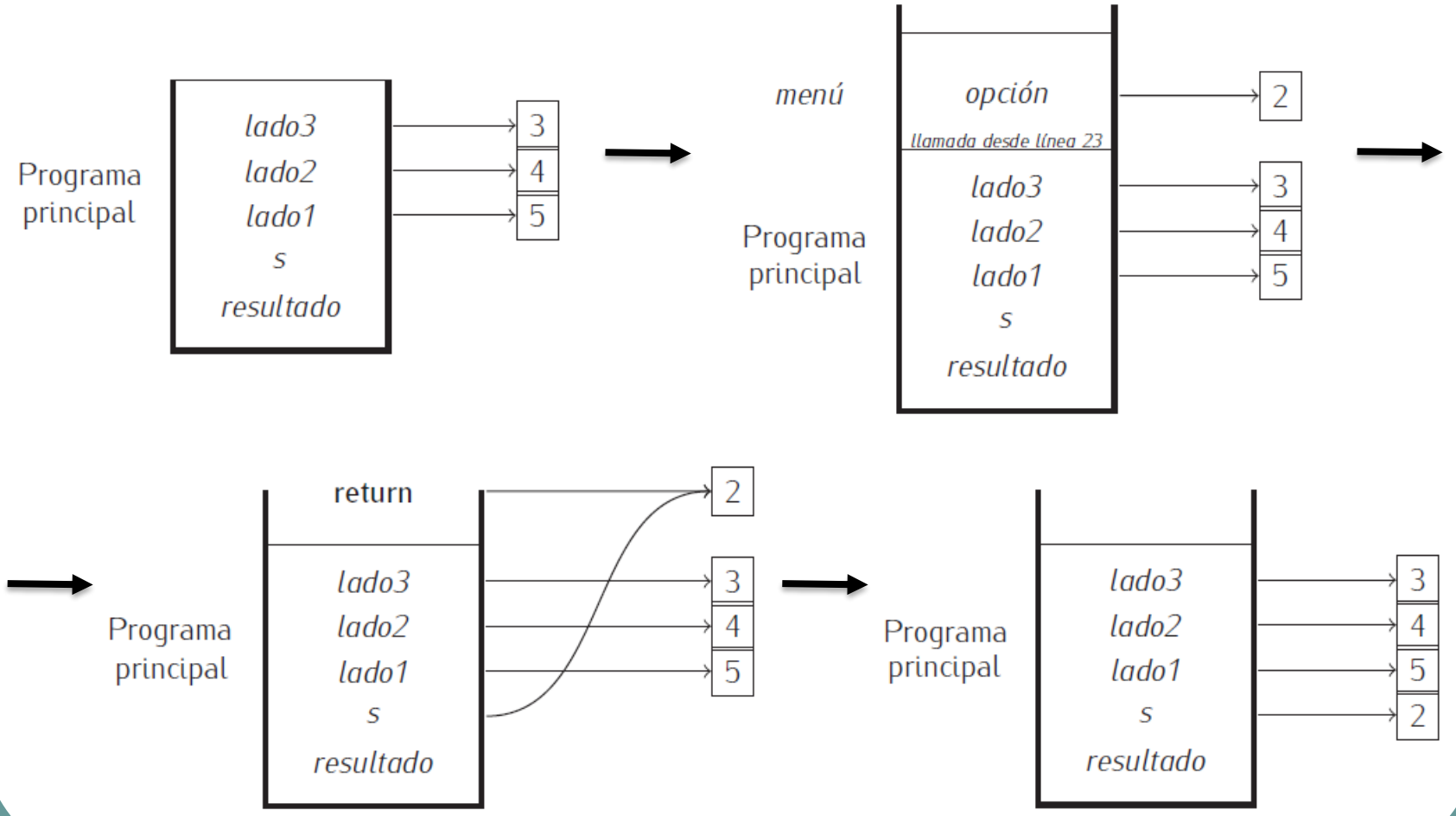
- Es una **estructura especial** de memoria que permite recordar la información asociada a cada llamada a función
- En una *pila* sólo se puede apilar en la *cima* y retirar el último elemento apilado



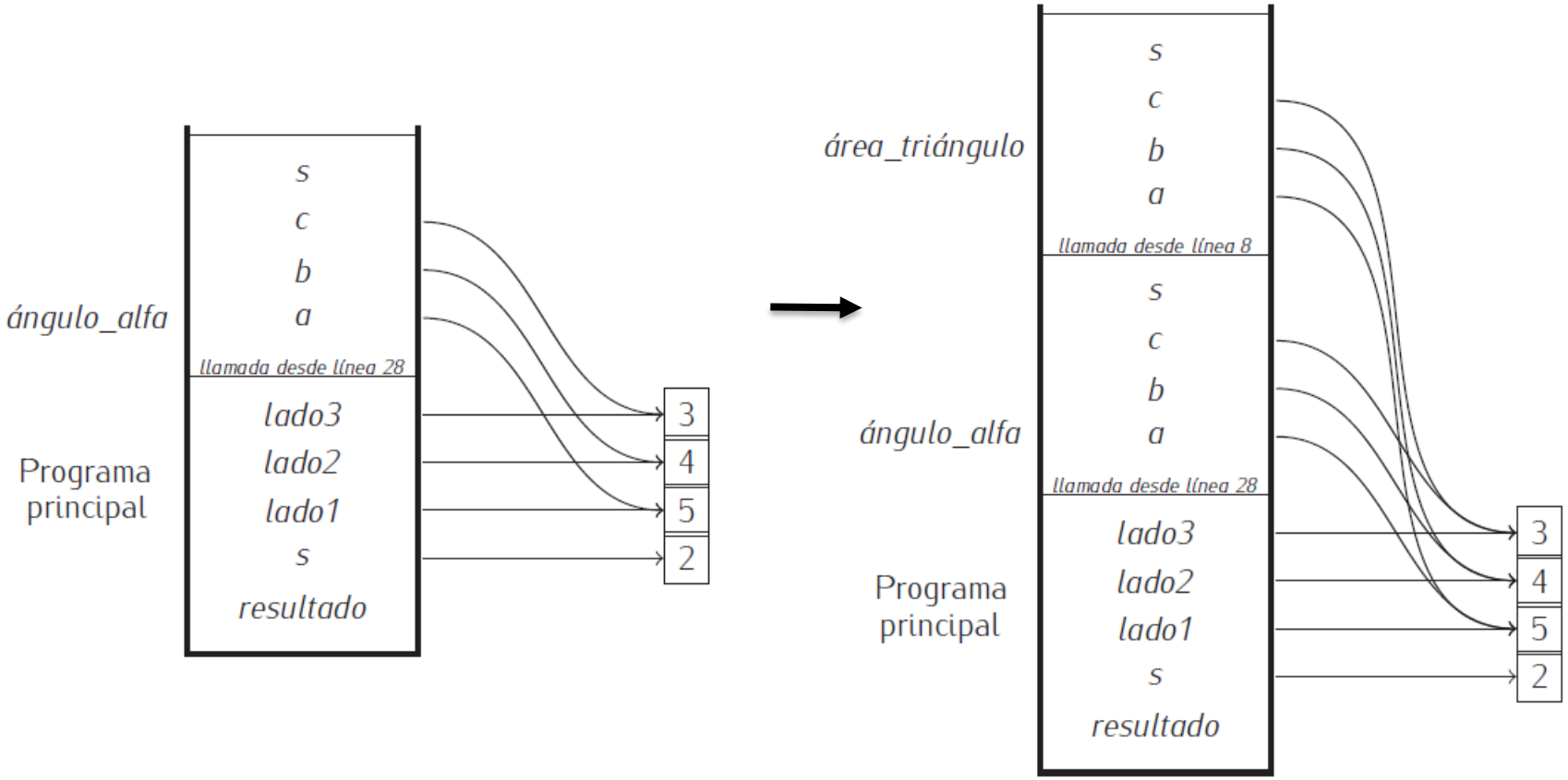
Tramas de activación

- Cada llamada a función apila un nuevo componente llamado *trama de activación*
- Una trama es una zona de memoria con espacio para referencias asociadas a parámetros y variables locales y *otra información* a recordar
- Al iniciar la ejecución de un programa se reserva una trama *especial* para las variables globales

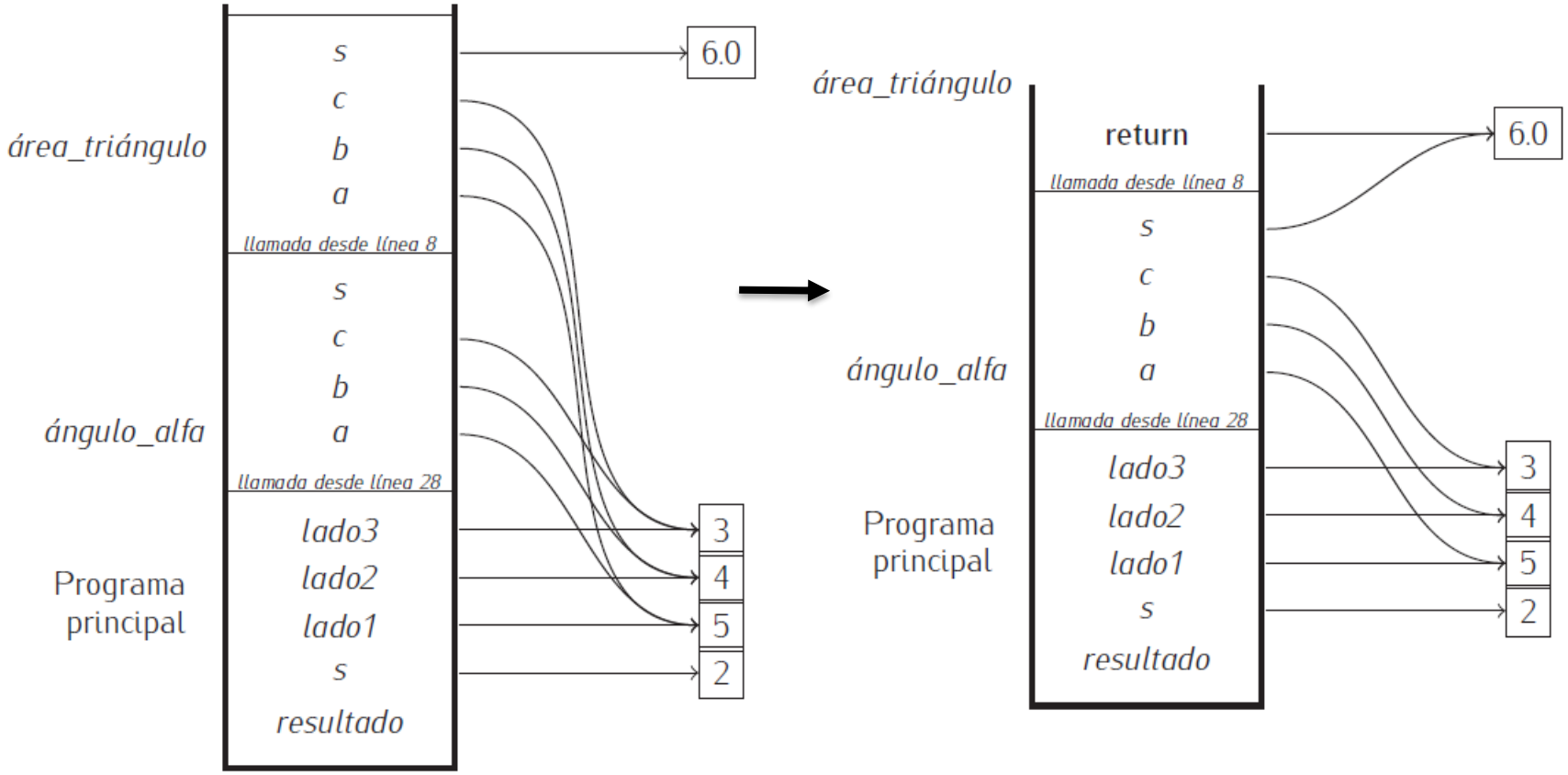
Ejemplo: 289_area_y_angulo.py



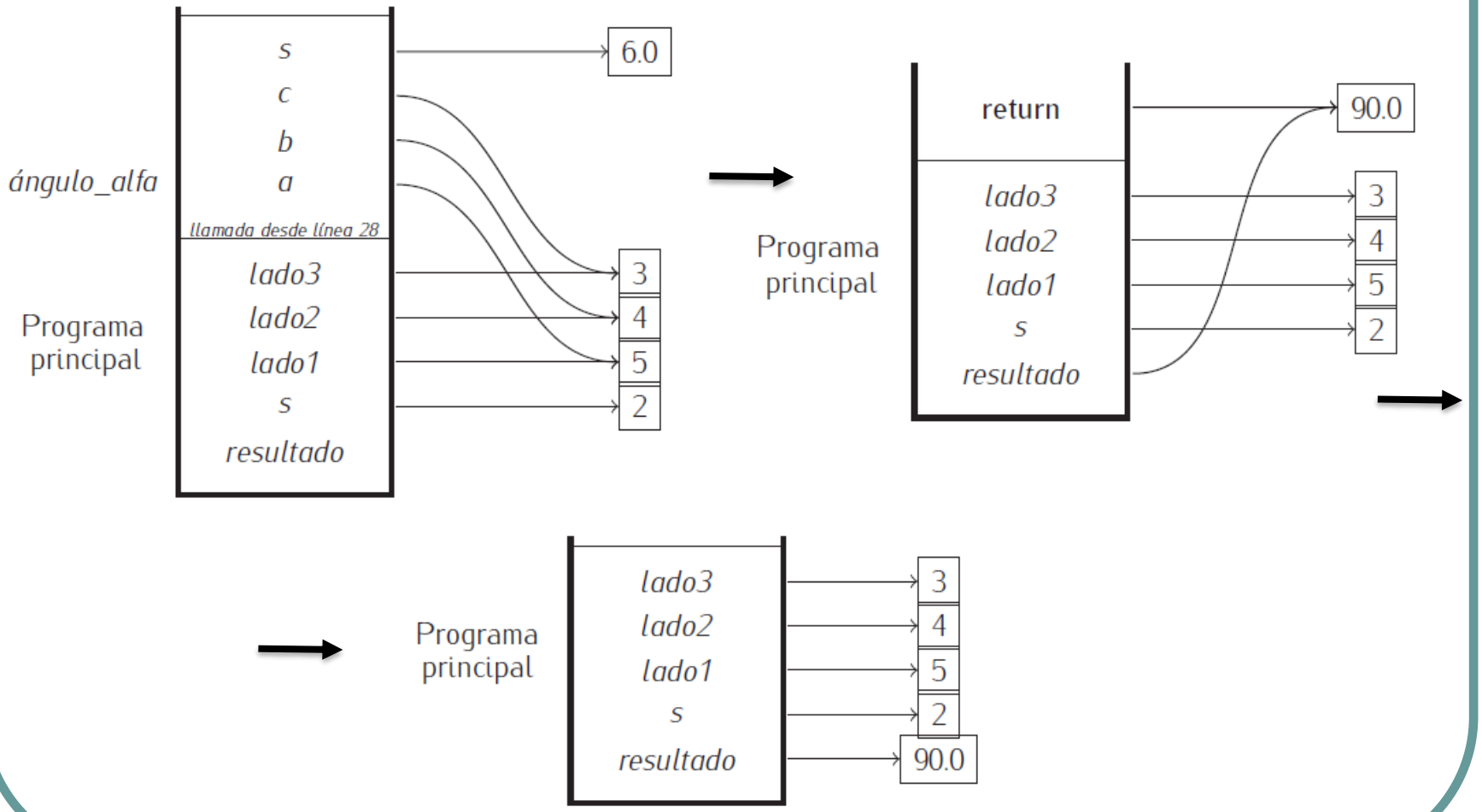
Ejemplo: 289_area_y_angulo.py



Ejemplo: 289_area_y_angulo.py



Ejemplo: 289_area_y_angulo.py



Más sobre el paso de parámetros

- ¿Qué ocurre con el dato original si se modifica un parámetro dentro de una función?

```
def incrementa(p):
    p = p + 1
    return p

a = 1
b = incrementa(a)
```

```
def modifica(a, b):
    a.append(4)
    b = b + [4]
    return b

l1 = [1, 2, 3]
l2 = [1, 2, 3]
l3 = modifica(l1, l2)
```

Acceso a variables globales

- Desde las funciones se puede acceder a las variables globales haciendo uso de la palabra reservada **global**
- Ejemplo: gestión de un cajero automático (p. 276 de Marzal)
- **Regla** que aplica el intérprete para determinar el ámbito de una variable: es local si participa en una asignación
- Modificar variables globales dentro de una función es **poco recomendable**

Recursividad

- Una función puede llamar a otra, ésta a una tercera, etc. (*anidamiento* de llamadas)
- Una función puede invocar a otra y ésta a la primera
- Una función que se llama a sí misma es una **función recursiva**

Ejemplo de recursividad: factorial

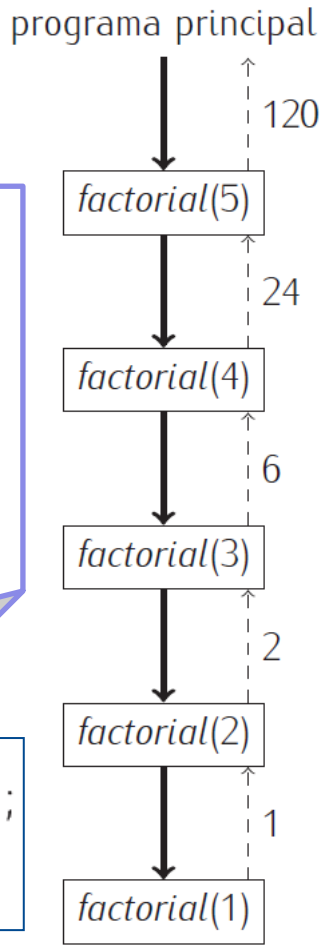
Versión convencional

```
def factorial(n):
    f = 1
    for i in range(1,n+1):
        f *= i
    return f
```

Versión recursiva

```
def fact(n):
    if n == 0 or n == 1:
        res = 1
    elif n > 1:
        res = n * fact(n-1)
    return res
```

$$n! = \begin{cases} 1, & \text{si } n = 0 \text{ o } n = 1; \\ n \cdot (n-1)!, & \text{si } n > 1. \end{cases}$$



Diseño modular

- ¿Por qué es conveniente estructurar un programa usando funciones?
- Los programadores tratan de escribir el menor número posible de líneas
- En programas sencillos no se aprecian ventajas, pero en programas complejos se repiten cálculos desde diferentes puntos
- Encapsular el código en funciones evita errores y mejora la legibilidad

Diseño modular

- ¿Cuándo construir una función?
  1. Encapsular los fragmentos que vayan a utilizarse más de una vez
  2. Integrar el código que se encargue de acciones fácilmente nombrables
  3. Limitar el tamaño de las funciones a 30 o 40 líneas como máximo
- Principio de *divide y vencerás*

Diseño descendente y ascendente

- Diseño **descendente** (*top-down*):
  - Establecer objetivo del programa y crear un esquema con las acciones a ejecutar
  - Definir una función para cada acción
  - Analizar la complejidad de cada función y decidir si se precisa dividirla en funciones más sencillas
- Diseño **ascendente** (*bottom-up*) es una propuesta complementaria de la anterior

Módulos

- Colecciones de funciones para construir programas por **aproximación ascendente**
- Biblioteca estándar de Python proporciona una gran cantidad de módulos predefinidos
- El programador también puede crear **módulos propios**
- Principio de *no reinventar la rueda*
- Propiedades de un módulo **bien formado**:
  - Alta cohesión
  - Bajo acoplamiento

Ejemplo de módulo propio

```
def min(a, b):
    if a < b:
        return a
    else:
        return b

def max(a, b):
    if a > b:
        return a
    else:
        return b
```

```
from minmax import min, max

x = 5
y = 3

maximo = max(x, y)
print(maximo)
```

Prueba de los módulos

```
def min(a, b):
    if a < b: return a
    else: return b

def max(a, b):
    if a > b: return a
    else: return b

if __name__ == '__main__':
    print('máximo(3, -7) =', max(3, -7))
    print('mínimo(3, -7) =',  min(3, -7))
```

El cuerpo de este condicional sólo se ejecuta si el módulo se interpreta directamente