# ¿Qué es la programación modular?

La **programación modular** es un paradigma de programación que consiste en dividir un programa en módulos o subprogramas con el fin de hacerlo más legible y manejable

Se presenta históricamente como una evolución de la programación estructurada para **solucionar problemas de programación más grandes y complejos** de lo que esta puede resolver.

Al aplicar la programación modular, un problema complejo debe ser dividido en varios subprogramas más simples, y estos a su vez en otros subprogramas más simples aún. Esto debe hacerse hasta obtener subprogramas lo suficientemente simples como para poder ser resueltos fácilmente con algún lenguaje de programación. **Esta técnica se llama refinamiento sucesivo, divide y vencerás o análisis descendente** (Top-Down).

**Un 'módulo' es cada una de las partes de un programa que resuelve uno de los subproblemas en que se divide el problema complejo original**. Cada uno de estos módulos tiene una tarea bien definida y algunos necesitan de otros para poder operar. En caso de que un módulo necesite de otro, puede comunicarse con este mediante una interfaz de comunicación que también debe estar bien definida.

Si bien un módulo puede entenderse como *una parte* de un programa en cualquiera de sus formas y variados contextos, en la práctica se los suele tomar como sinónimos de procedimientos y funciones. Pero no necesaria ni estrictamente un módulo es una función o un procedimiento, ya que el mismo puede contener muchos de ellos. No debe confundirse el término "módulo" (en el sentido de programación modular) con términos como "función" o "procedimiento", propios del lenguaje que lo soporte.

# Definición de funciones en Python

A la hora de definir una función en Python, comenzamos con la palabra def seguida del nombre de la función y los parámetros que tendrá, entre paréntesis. Para cada parámetro sólo debemos especificar su nombre (recuerda que en Python no se especifican los tipos de datos explícitamente).

Igual que ocurre con otras estructuras como if o while, el código perteneciente a una función debe estar tabulado. Además, si queremos que la función devuelva algún valor, podemos emplear la cláusula return como en otros lenguajes, aunque no es obligatoria si no queremos devolver nada. También podemos definir un return vacío para indicar que no se devuelve nada.

Veamos algunos ejemplos.

def maximo(num1, num2):

if num1 > num2:

return num1

else:

return num2

def imprimeTexto(texto):

print(texto)

return # Esta línea se podría omitir

A la hora de llamar a estas funciones desde otras partes del código, se hace igual que en muchos otros lenguajes:

print ("Escribe dos números")

n1 = int(input())

n2 = int(input())

print ("El máximo es", maximo(n1, n2))

texto = input("Escribe un texto:")

imprimeTexto(texto)

**AC301. Realiza las siguientes funciones.**

1. **sumaT (num).** Calcula la suma de los números enteros positivos menores o iguales a num. Si el número es negativo, debe devolver 0.
2. **sucFibo (tope).** Realiza una función que debe devolver una lista con la sucesión de Fibonacci menor o igual a tope.

La sucesión de Fibonacci es una secuencia de números en la cual cada número es la suma de los dos anteriores, comenzando con 0 y 1.

Entonces, la secuencia de Fibonacci comienza así: 0,1,1,2,3,5,8,13,21,34,55.

# Sobre los parámetros

En Python **todos los parámetros simples (números, booleanos y textos) se pasan por valor** (se pasa una copia del mismo), con lo que no podemos modificar el valor original del dato, y **todos los tipos complejos (listas, u objetos) se pasan por referencia**. Esto último implica que, siempre que se mantenga la referencia original, podemos modificar el valor del parámetro de forma persistente (se aplica a la variable original utilizada como parámetro).

**Caso de estudio.** Indica que se imprime en cada uno de los casos.

|  |  |
| --- | --- |
| def anyadirValores1(lista):  lista.append(30)  print ("Valores en la función:", lista)  return  #------------------  lista1 = [10, 20]  anyadirValores1(lista1)  print ("Valores fuera de la función:", lista1) | def anyadirValores2(lista):  lista = [30, 40]  print ("Valores en la función:", lista)  return  #------------------  lista1 = [10, 20]  anyadirValores2(lista1)  print ("Valores fuera de la función:", lista1) |
|  |  |

## Tipos de parámetros

Los parámetros definidos en una función pueden ser de distintos tipos, y se pueden especificar de distintas formas. Veremos aquí algunos ejemplos.

Por un lado, tenemos los parámetros **obligatorios**. Es el modo normal de pasar parámetros; si simplemente definimos el nombre de cada parámetro, entonces ese parámetro es obligatorio, y debemos darles valor al llamarles, en el mismo orden en que están definidos. Aquí podemos ver un ejemplo (el mismo visto anteriormente):

def maximo(num1, num2):

if num1 > num2:

return num1

else:

return num2

También podemos invocar a una función usando los nombres de los parámetros como **palabras clave.** De este modo no tenemos por qué seguir el mismo orden que cuando se definió dicha función. Por ejemplo:

def imprimirDatos(nombre, edad):

print ("Tu nombre es", nombre, "y tu edad es", edad)

return

...

imprimirDatos(edad = 28, nombre = "Juan")

Además, podemos asignar **valores por defecto** a los parámetros que queramos. Así, si queremos llamar a la función, podemos omitir los parámetros que tengan un valor por defecto asignado, si queremos. Por ejemplo:

def imprimirDatos(nombre, edad = 0):

print ("Tu nombre es", nombre, "y tu edad es", edad)

return

...

imprimirDatos("Juan") # Imprime "Tu nombre es Juan y tu edad es 0"

Es importante que los parámetros que tengan valores por defecto se coloquen todos al final de la lista de parámetros (tras los obligatorios), para que así no queden huecos si queremos llamar a la función omitiendo parámetros. También es importante que, cuando omitamos un parámetro, los que vayan detrás también se omitan para que no se desplace el orden y se asignen por error a otros parámetros.

### Funciones con un número variable de parámetros

### Las funciones en Python también admiten un **número variable de parámetros**. Esto lo podemos especificar como último parámetro de la función un tipo especial que permite pasar tantos parámetros como necesitemos. Por ejemplo:

def imprimirTodo(num1, \*numeros):

print("Primer número:", num1)

for num in numeros:

print num

return

Lo que hará la función en este caso es recibir un primer parámetro obligatorio (*num1*) y el resto, opcionales, se recibirán en forma de **tupla** con sus valores. Podemos invocar a la función así:

imprimirTodo(1, 2, 3, 4)

# La tupla sería (2, 3, 4) en este caso

De forma alternativa podemos indicar un doble asterisco en ese último parámetro:

def imprimirTodo(num1, \*\*valores):

print("Primer número:", num1)

for num in valores:

print valores[num]

return

En este otro caso lo que se recibe como parámetro adicional es un **mapa** donde a cada parámetro (valor) se le asocia un nombre (clave). Podríamos invocar a la función de este modo:

imprimirTodo(1, a = 2, b = 3)

**AC302. Realiza las siguientes funciones.**

1. Realiza una función llamada area\_rectangulo que reciba dos parámetros: la bae y la altura de un rectángulo. Si no se proporciona la altura, debe asumir que es cuadrado, por lo que la altura tendrá el misma valor que la base. La función debe devolver el área de rectángulo.

print (area\_rectangulo(4,8)) # salida 32

print (area\_rectangulo(4)) # salida 16

1. Potencia de un número por defecto.

Escribe una función llamada elevar\_potencia que reciba dos parámetros: base y exponente. El exponente debe tener un valor por defecto de 2, lo que significa que, si no se especifica un exponente, la función debe calcular el cuadrado de la base. La función debe devolver el resultado de la operación base \*\* exponente

1. Escribe una función denominada suma\_numeros que reciba un número indefinido de argumentos y calcule la suma de todos ellos.

sumar\_numeros(1, 2, 3)

sumar\_numeros(4, 5, 6, 7, 8)

1. Realiza una función denominada sumaFibo(tope), que calcule la suma de la serie de Fibonacci hasta un número menor o igual a tope. No se puede llamar a la función de ejemplo sucFibo
2. Realiza una función que obtenga el máximo común divisor (MCD) de dos números utilizando el algoritmo de Euclides. El mcd es el divisor mayor que tienen en común dos números. El algoritmo de Euclides se basa en:
3. Dividir el mayor de n1 y n2 entre el menor
4. Si la división es exacta (resto 0), el MCD es el número menor
5. Si no, se sustituye el número mayor por el resto de la división, y se vuelve al paso 1

Por ejemplo, para 20 y 12 haríamos algo así:

* Dividimos 20 / 12. No es exacta, y el resto es 8. Reemplazamos 20 por 8
* Dividimos 12 / 8. No es exacta, y el resto es 4. Reemplazamos 12 por 4
* Dividimos 8 / 4. Es exacta, con lo que el MCD es 4.

# Recursividad

La recursividad es una técnica en programación en la que una función se llama a sí misma para resolver un problema. El problema se resuelve dividíendolo en subproblemas más pequeños y aplicando la misma función de manera repetitiva.

## Elementos claves de la recursividad

1. **Caso base:** Es la condición que detiene la recursividad. Sin un caso base, la función seguiría llamándose indefinidamente, lo que llevaría a un desbordamiento de la pila de llamadas (stack overflow).
2. **Llamada recursiva:** Es el momento en que la función se llama a sí misma con un argumento modificado. Este argumento generalmente está diseñado para acercar el problema al caso base.

**Ejemplo. Factorial de un número**

El factorial de un número entero n (denotado como n!) se define como el producto de todos los números enteros positivos menores o iguales a n. El factorial de 0 es 1 por definición.

n!=n×(n−1)×(n−2)×⋯×1

Podemos definir el factorial de forma recursiva:

n!=n×(n−1)!

Esta función en Python sería:

def factorial(n):

# Caso base: si n es 0, devuelve 1

if n == 0:

return 1

# Llamada recursiva

else:

return n \* factorial(n - 1)

**AC303. Establece el caso base y caso general de las siguientes funciones.**

|  |  |
| --- | --- |
| **Sumatorio de un número** sumaT(num)   * + Caso base:   + Caso general: | **En Python.** |
| **Inverso de un número,** invertir(num)  invertir(3452) =2543 |  |
| **Máximo común divisor,** mcd(a,b) |  |
| **Sucesión de Fibonacci,** sucFibo(pos), debe devolver una lista con los elementos de la sucesión.  sucFibo(0)=[0] sucFibo(1)=[0,1] sucFibo(2)=[0,1,1] sucFibo(3)=[0,1,1,2] sucFibo(4)=[0,1,1,2,3] sucFibo(5)=[0,1,1,2,3,5] |  |
| **Suma de los elementos de la sucesión de Fibonacci,** sumaFibo(pos), siendo pos la posición del número de la serie.  sumaFibo(0)=0, sumaFibo(1)=1, sumaFibo(2)=2, sumaFibo(3)=4, sumaFibo(4)=7, sumaFibo(5)=12 |  |

**AE304. Realiza una función iterativa (sin recursividad) denominada sumaDigital(numero), cuyo resultado sea la suma de los dígitos que lo conforman**

sumaDigital(13245) 🡪 15

sumaDigital(32) 🡪 5

sumaDigital(10) 🡪 1

sumaDigital(4) 🡪 4

**AE305. Realiza el ejercicio del AE304 como función recursiva.**