

5

Diseño de programas elementales en Python

En este capítulo exploraremos la estructura y funcionamiento de programas básicos en Python, prestando especial atención a la modularidad y la comunicación de datos entre funciones. Para ello, vamos a diseñar e implementar un programa que escriba la tabla de multiplicar de un número dado por el usuario.

El programa permitirá al usuario solicitar la tabla de multiplicar de un número. El programa continuará funcionando hasta que el usuario introduzca el número 0, momento en el que finalizará su ejecución.

Un ejemplo de ejecución de lo que se desea es:

```
¿Qué tabla desea escribir? (0 para acabar): 7
LA TABLA DEL 7
7 x 0 = 0
7 x 1 = 7
7 x 2 = 14
...
7 x 10 = 70
```

```
¿Qué tabla desea escribir? (0 para acabar): 0
```

5.1. Funciones

Podemos resolver este problema directamente dentro de un bloque principal. El flujo del programa sería: (1) pedirle al usuario un número, (2) generar y mostrar la tabla de multiplicar de ese número, y (3) repetir este proceso hasta que el usuario ingrese un 0 para finalizar.

Por ejemplo:

```
1 # Solución sin funciones
2
3 while True:
4     numero = int(input("¿Qué tabla desea escribir? (0 para
↪     acabar): "))
5     if numero == 0:
6         break
7
8     print(f"\nLA TABLA DEL {numero}")
9     for i in range(11):
10        print(f"{numero} x {i} = {numero * i}")
```

En este caso, el programa hace todo en un único bloque. Primero, el programa solicita el número de la tabla (línea 4) y luego, en un bucle, genera las multiplicaciones desde 0 hasta 10 (líneas 9 y 10). Si el usuario ingresa 0, el programa termina (líneas 5 y 6).

Esta solución plantea realizar todo en un único bloque de código, sin necesidad de definir *funciones*. Una función es un conjunto de instrucciones que realizan una tarea específica. Cuando definimos una función, no se ejecuta inmediatamente; se ejecutará cuando la llamemos. Llamar a una función significa invocar o ejecutar el bloque de código que define esa función. Es decir, cuando “llamamos” a una función, el programa va a la parte de código donde está definida y ejecuta las instrucciones que contiene.

La solución sin funciones es rápida y sencilla, especialmente en programas pequeños o cuando no se anticipa que el código se reutilice en otros contextos. Sin embargo, a medida que el programa crece o se vuelve más complejo, este enfoque puede llevar a un código repetitivo y menos organizado. La falta de modularidad puede hacer que el código sea más difícil de leer y mantener, especialmente si necesitamos modificar el comportamiento del programa o añadir nuevas características. Además, la estructura del programa se vuelve más difícil de gestionar si se requiere hacer cambios en la lógica principal.

Ahora, vamos a reestructurar el programa utilizando funciones. Como se ha mencionado, esto no sólo organiza el código, sino que facilita la reutilización y mejora la legibilidad. La solución sería algo como esto:

```
1 # Solución con funciones
2
3 def presentar_tabla(n):
4     """Muestra por pantalla la tabla de multiplicar del nú
↪     mero n."""
5     print(f"\nLA TABLA DEL {n}")
6     for i in range(11):
7         print(f"{n} x {i} = {n * i}")
8
9 def main():
10    """Gestiona el diálogo con el usuario y llama a
```

```

11     presentar_tabla."""
12     while True:
13         numero = int(input("¿Qué tabla desea escribir? (0
14             para acabar): "))
15         if numero == 0:
16             break
17         presentar_tabla(numero)
18 if __name__ == "__main__":
19     main()

```

En esta versión del programa, hemos dividido el código en dos funciones:

- La función `presentar_tabla(n)` se encarga de imprimir la tabla de multiplicar para el número `n`.
- La función `main()` gestiona la interacción con el usuario y llama a `presentar_tabla()` para mostrar la tabla cuando sea necesario.

El enfoque con funciones es más estructurado y modular. Al tener funciones separadas para tareas específicas (como `presentar_tabla()`), el código es más limpio y fácil de entender. Además, la reutilización del código se vuelve mucho más fácil, porque cada función puede ser invocada en cualquier parte del programa sin duplicar la lógica. En este caso, por ejemplo, si quisiéramos agregar nuevas funcionalidades (como imprimir una tabla en un archivo o mostrar más detalles), sólo necesitaríamos hacer cambios en una función sin tocar la parte del código que maneja la interacción con el usuario.

Es conveniente descomponer nuestros programas en funciones siempre que sea posible. La principal desventaja de usar funciones en programas muy pequeños es que podría parecer innecesario dividir el código en funciones separadas cuando la lógica es tan simple. Esto podría hacer que el código sea un poco más largo de lo necesario, pero este “coste” se justifica rápidamente cuando el programa crece o se vuelve más complejo.

Respecto a la solución con funciones planteada, conviene explicar la estructura:

```

if __name__ == "__main__":
    main()

```

que aparece al final de la solución planteada. Este bloque es una forma común en Python de asegurarse de que un determinado bloque de código sólo se ejecute cuando el script se ejecuta directamente y no cuando es importado como módulo en otro programa.

En Python, `__name__` es una variable especial que indica el contexto en el que un script se está ejecutando. Cuando un script es ejecutado directamente (por ejemplo, con `python3 script.py` desde una terminal), Python asigna el valor `"__main__"` a la variable `__name__`. Por contra, cuando un script es importado como un módulo en otro programa, `__name__` tomará el nombre del módulo y no será `"__main__"`.

5.1.1. Definición de funciones

La definición de una función en Python comienza con la palabra clave `def`, seguida del nombre de la función, los paréntesis que pueden incluir parámetros (si los tiene), y dos puntos `:`. Los parámetros son valores que una función puede recibir cuando es llamada. Estos valores permiten que la función trabaje con datos específicos en lugar de realizar solo una tarea generalizada. Dentro de la función, el código se indenta para indicar que las instrucciones forman parte de la función.

La sintaxis básica para definir una función es:

```
def nombre_de_la_funcion(parametros):  
    # Código de la función  
    # Realiza alguna tarea
```

Veamos un ejemplo básico de definición de función muy simple que no toma parámetros y no devuelve nada:

```
1 def saludar():  
2     print(";Hola, mundo!")
```

5.1.2. Parámetros en la función

Los parámetros permiten que una función sea flexible y reutilizable con diferentes entradas sin tener que escribir el mismo código una y otra vez. Los parámetros son valores que la función puede recibir para realizar su tarea. Se definen dentro de los paréntesis de la función. Los parámetros son como “variables” dentro de la función, que se usan para realizar los cálculos o tareas.

Veamos un ejemplo básico de definición de función muy simple con parámetros:

```
1 def saludar_persona(nombre):  
2     print(f";Hola, {nombre}!")
```

En este caso, la función `saludar_persona` toma un parámetro llamado `nombre`. Ahora, cuando llamamos a la función, le pasamos un valor para ese parámetro:

```
saludar_persona("Carlos") # Imprime: ;Hola, Carlos!
```

Parámetros con valores por defecto

En Python, es posible asignar valores por defecto a los parámetros de una función. Esto significa que si no se proporciona un valor para ese parámetro al llamar a la función, se utilizará el valor predeterminado. Los valores por defecto permiten hacer que los parámetros sean opcionales, facilitando la reutilización de las funciones.

La sintaxis para definir parámetros con valores por defecto es la siguiente:

```
1 def saludar(nombre="Usuario"):  
2     print(f"Hola, {nombre}!")  
3
```

```

4 # Llamar a la función sin pasar un argumento para 'nombre'
5 saludar() # Salida: Hola, Usuario!
6
7 # Llamar a la función pasando un argumento
8 saludar("Carlos") # Salida: Hola, Carlos!

```

En este ejemplo, el parámetro `nombre` tiene un valor por defecto de `"Usuario"`. Si no se pasa un valor para `nombre`, se utiliza este valor predeterminado. Si se pasa un valor, ese será el que se use en lugar del predeterminado.

Los parámetros con valores por defecto deben ir al final de la lista de parámetros. Si una función tiene parámetros con valores por defecto, estos deben colocarse después de los parámetros sin valor por defecto. De lo contrario, Python generará un error de sintaxis (`SyntaxError: non-default argument follows default argument`):

```

# Correcto
def saludar(nombre, mensaje="¡Hola!"):
    print(f"{mensaje}, {nombre}")

# Incorrecto: SyntaxError en tiempo de compilación
def saludar(mensaje="¡Hola!", nombre):
    print(f"{mensaje}, {nombre}")

```

Si una función tiene tanto parámetros con valores por defecto como parámetros sin valores por defecto, los parámetros sin valores por defecto deben ser pasados al llamar a la función, y los valores por defecto solo se aplicarán a los parámetros que no se hayan pasado:

```

def saludar(nombre, mensaje="¡Hola!"):
    print(f"{mensaje}, {nombre}")

# Llamadas correctas
saludar("Carlos") # Salida: ¡Hola!, Carlos
saludar("Carlos", "Buenos días") # Salida: Buenos días,
→ Carlos

```

Este tipo de flexibilidad hace que las funciones sean más fáciles de manejar, especialmente cuando tienen muchos parámetros, pero algunos de ellos tienen valores predeterminados que sólo cambian en ciertos casos.

5.1.3. Llamar a una función

Cuando llamamos o invocamos una función, lo que hacemos es ejecutar el código que hemos definido dentro de ella. Para ello, simplemente escribimos su nombre seguido de paréntesis. Si la función tiene parámetros, los valores deben ser proporcionados dentro de los paréntesis, separados por comas.

En el caso de que la función tenga varios parámetros, los valores deben ser pasados en el mismo orden en que los definimos en la declaración de la función. Es decir, el primer

5. Diseño de programas elementales en Python

valor que pasamos corresponde al primer parámetro, el segundo valor corresponde al segundo parámetro, y así sucesivamente. Los valores deben estar separados por comas.

Por ejemplo, si tienes una función que imprime dos números y la defines de esta manera:

```
1 def imprime(a, b):  
2     print("a:", a, "; b:", b)
```

Para llamarla y pasarle los valores que mostrará, deberás proporcionar dos números en el orden correspondiente:

```
imprime(5, 3)
```

En este caso, el valor 5 se pasa al parámetro `a`, y el valor 3 se pasa al parámetro `b`. Cuando la función se ejecuta, se imprimirá por pantalla el valor 5 de `a` y el valor 3 de `b`.

Es importante recordar que el número de parámetros debe coincidir con el número de argumentos que pasamos en la llamada. Si intentamos llamar a la función con un número incorrecto de argumentos, Python generará un error.

Por ejemplo, si intentas pasar solamente un argumento a la función `imprime`, obtendrás un error durante la ejecución del programa:

```
↪ # Error, la función espera dos parametros  
imprime(5) # TypeError: missing 1 required positional  
           argument: 'b'
```

Python permite usar parámetros con nombre (también conocidos como argumentos con palabra clave). En lugar de pasar los valores en el orden en que fueron definidos, puedes especificar qué parámetro está recibiendo cada valor. Esto se hace utilizando el nombre del parámetro seguido del operador de asignación (`=`) y el valor que deseas pasar (no lo confundas con que el parámetro tenga un valor por defecto: aquí el `=` aparece en la llamada, no en la definición).

Por ejemplo:

```
1 def saludar(nombre, edad):  
2     print(f"Hola, {nombre}. Tienes {edad} años.")
```

En este caso, al invocar la función, puedes pasar los parámetros por nombre:

```
saludar(edad=30, nombre="Carlos")
```

Aquí, aunque `nombre` se define primero en la función, estamos pasando el valor 30 a `edad` y el valor `"Carlos"` a `nombre`. Esto puede ser útil para hacer el código más legible y para asegurarte de que los valores son asignados correctamente a los parámetros, especialmente cuando tienes funciones con varios parámetros. Además, permite que no sea necesario seguir estrictamente el orden de los parámetros, lo que mejora la flexibilidad.

5.1.4. La sentencia `return`

Una función puede devolver un valor usando la sentencia `return`. Cuando una función tiene `return`, el valor especificado es enviado de vuelta a quien llamó a la función. Una

función que no tiene `return` devolverá `None` de forma predeterminada.

Veamos un ejemplo de función con `return`:

```
1 def sumar(a, b):
2     return a + b
```

Aquí, la función `sumar` toma dos parámetros, `a` y `b`, y devuelve la suma de ambos. Cuando llamamos a esta función, podemos “capturar” el valor devuelto:

```
resultado = sumar(5, 3) # Llamada a la función
print(resultado) # Imprime 8
```

Cuando la función se ejecuta, el resultado de evaluar $5 + 3$ (código de la función `sumar` con los parámetros recibidos) se almacena en la variable `resultado`, con lo que mostrará por pantalla el valor 8.

La sentencia `return` puede usarse en cualquier parte de la función. Si se encuentra un `return`, la ejecución de la función se detiene inmediatamente, y el valor se devuelve a la parte que hizo la llamada.

Por ejemplo:

```
1 def mayor_de_dos(a, b):
2     if a > b:
3         return a # Si 'a' es mayor, devolvemos 'a'
4     else:
5         return b # Si no, devolvemos 'b'
6
7 # Llamada a la función
8 print(mayor_de_dos(5, 10)) # Imprime 10
```

Aquí, la función devuelve el valor mayor de los dos números comparados. Si `a > b`, se devuelve `a`; si no, se devuelve `b`. La ejecución de la función se detiene en el primer `return` que se encuentre.

Si una función no tiene una sentencia `return`, entonces no devuelve un valor explícito. En este caso, el valor por defecto que Python retorna es `None`.

Por ejemplo:

```
1 def saludar_persona(nombre):
2     print(f";Hola, {nombre}!") # Imprime el saludo
3
4 # Llamamos a la función
5 saludar_persona("Carlos") # Imprime: ;Hola, Carlos!
6
7 # Esta llamada no tiene valor de retorno, así que:
8 saludo = saludar_persona("Carlos")
9 print(saludo) # Imprime: None
```

En este ejemplo, la función `saludar_persona` no tiene un `return`, por lo que cuando intentamos capturar su salida en la variable `saludo`, esta obtiene el valor `None`.

Retorno de múltiples valores

En Python, una función puede devolver más de un valor separándolos por comas. En realidad, lo que ocurre es que Python agrupa estos valores en una tupla¹. Por ejemplo:

```
1 def operaciones_basicas(a, b):
2     suma = a + b
3     resta = a - b
4     multiplicacion = a * b
5     return suma, resta, multiplicacion # Se devuelven tres
↪     valores
6
7 # Captura de múltiples valores en variables separadas
8 resultado_suma, resultado_resta, resultado_multiplicacion =
↪     operaciones_basicas(10, 5)
9
10 print(f"Suma: {resultado_suma}, Resta: {resultado_resta},
↪     Multiplicación: {resultado_multiplicacion}")
```

Esto facilita la programación, ya que permite retornar varios resultados sin necesidad de encapsularlos en una lista o diccionario (tipos de datos estructurados estudiados en un capítulo posterior). También se pueden capturar todos los valores en una única variable si no se conoce de antemano cuántos valores retornará la función:

```
1 resultado = operaciones_basicas(10, 5)
2 print(resultado) # Salida: (15, 5, 50)
```

Así, resultado contendrá una tupla con los tres valores, que luego pueden accederse mediante resultado[0], resultado[1], etcétera.

5.1.5. Documentar funciones: docstrings

Un *docstring* (cadena de documentación) es una cadena literal que aparece como **primera sentencia** de un módulo, función, clase o método. Aunque sintácticamente es una cadena cualquiera, Python la asocia automáticamente al objeto en el atributo especial `__doc__`, y herramientas como `help()` la utilizan para mostrar la documentación al usuario. Por ejemplo:

```
1 def calcular_area(radio):
2     """Calcula el área de un círculo dado su radio."""
3     return 3.1416 * radio * radio
4
5 print(calcular_area.__doc__)
6 # Salida: Calcula el área de un círculo dado su radio.
7
```

¹Este tipo de dato estructurado se estudia en detalle más adelante.

```

8 help(calcular_area)
9 # Salida:
10 # calcular_area(radio)
11 #     Calcula el área de un círculo dado su radio.

```

Para funciones simples basta con una línea entre comillas triples. Para funciones más complejas, es habitual seguir alguna convención de estilo: las más populares son *Google style*, *NumPy style* y *Sphinx/reST style*. A modo de ejemplo, una versión más completa al estilo Google sería:

```

1 def calcular_area(radio):
2     """Calcula el área de un círculo.
3
4     Args:
5         radio (float): radio del círculo, en metros.
6
7     Returns:
8         float: área del círculo, en metros cuadrados.
9     """
10    return 3.1416 * radio * radio

```

Adoptar siempre la misma convención dentro de un proyecto es importante para mantener la coherencia, y permite que generadores de documentación como *Sphinx* o *pdoc* produzcan automáticamente referencias a partir del código fuente.

5.1.6. Anotaciones de tipo

En Python, podemos indicar los tipos de datos de los parámetros de una función utilizando las anotaciones de tipo (del inglés, *type hints*). Esto significa que podemos especificar qué tipo de datos se espera que reciba cada parámetro de la función. Por ejemplo, si queremos indicar que una función debería recibir un número entero como parámetro, podemos anotarlo de la siguiente manera:

```

1 def sumar(a: int, b: int) -> int:
2     return a + b

```

En este ejemplo, las anotaciones `a: int` y `b: int` indican que esperamos que ambos parámetros sean de tipo entero (`int`), y la anotación `-> int` después de los paréntesis de la función indica que esperamos que la función devuelva un valor de tipo entero.

Aunque esta anotación de tipo proporciona claridad sobre lo que esperamos como entrada y salida de la función, es importante recordar que Python es un lenguaje de tipado dinámico. Esto significa que el intérprete de Python **no** valida en tiempo de ejecución que los tipos coincidan con los anotados: las anotaciones son simplemente metadatos asociados al objeto función.

Por ejemplo, en el caso de la función `sumar`, podríamos llamarla con parámetros de tipo incorrecto y el intérprete no rechazaría la llamada:

```
↪ sumar(5, "10") # Python no verifica las anotaciones; aceptar  
    ía la llamada
```

Aunque Python no genera error inmediatamente, esta función intentará sumar un entero (5) y una cadena ("10"), lo que resultará en un error en tiempo de ejecución:

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Si se desea verificar las anotaciones *antes* de ejecutar el programa, existen herramientas externas de análisis estático como `mypy` (<https://mypy-lang.org/>) o `pyright` (<https://github.com/microsoft/pyright>), que examinan el código fuente y avisan de inconsistencias entre las anotaciones y los usos reales. Estas herramientas son la forma habitual de aprovechar las anotaciones de tipo en proyectos grandes.

5.1.7. Ámbito de las variables

En Python, cada variable tiene un *ámbito* (también llamado *alcance* o *scope*), que determina dónde puede ser accedida en el código, y un *tiempo de vida*, que define cuánto tiempo permanece disponible en memoria durante la ejecución. Existen dos ámbitos básicos: variables *locales* y variables *globales*.

Una variable local es aquella definida dentro de una función y solo es accesible dentro de ella. Por ejemplo:

```
1 def ejemplo_local():  
2     x = 10 # Variable local  
3     print(x) # Accesible dentro de la función  
4  
5 ejemplo_local()  
6 # print(x) # Generaría un error: 'x' no existe fuera de la  
↪ función
```

Las variables globales son aquellas definidas fuera de cualquier función y pueden leerse desde cualquier parte del programa:

```
1 mensaje = "Hola, mundo" # Variable global  
2  
3 def mostrar_mensaje():  
4     print(mensaje) # Lee la variable global  
5  
6 mostrar_mensaje()
```

Sin embargo, si dentro de una función se asigna un valor a una variable global sin declararla explícitamente como global, Python creará una variable local en su lugar. Para modificar la variable global dentro de la función, hay que declararla con la palabra clave `global`:

```
1 contador = 0 # Variable global  
2
```

```

3 def incrementar():
4     global contador # Se declara explícitamente como global
5     contador += 1
6
7 incrementar()
8 print(contador) # Muestra 1

```

Regla LEGB. Cuando una función usa un nombre, Python lo busca siguiendo cuatro ámbitos en orden: **L**ocal (variables definidas dentro de la propia función), **E**nclosing (variables de funciones que la engloban si las hay), **G**lobal (variables del módulo) y **B**uilt-in (nombres del lenguaje, como `print` o `len`). El primer ámbito en el que se encuentra el nombre es el que se usa.

Para modificar una variable del ámbito *enclosing* desde una función anidada se utiliza la palabra clave `nonlocal`, análoga a `global` pero referida al ámbito de la función contenedora:

```

1 def contador_externo():
2     n = 0
3     def incrementar():
4         nonlocal n # Se refiere a la 'n' de contador_externo
5         n += 1
6         return n
7     return incrementar

```

Las variables globales son una mala práctica. Compartir información entre funciones a través de variables globales debe evitarse: genera efectos secundarios indeseados, dificulta la depuración y reduce la modularidad y la reutilización del código. La forma recomendada de comunicar funciones es a través de *parámetros* (datos de entrada) y *valores de retorno* (datos de salida), tal como se explica en la siguiente sección. Este enfoque permite que cada función sea independiente, facilita su prueba, su reutilización y su mantenimiento, y elimina el riesgo de que modificaciones accidentales en un sitio afecten al resto del programa.

5.2. Comunicación entre funciones

La comunicación entre funciones se refiere al proceso mediante el cual una función interactúa con otras dentro de un programa. Esto se puede lograr de diferentes maneras, y el método más común es pasar datos entre funciones. La comunicación puede hacerse mediante el uso de parámetros y valores de retorno.

Cuando una función necesita usar información proporcionada por otra, generalmente se hace mediante los parámetros. Los parámetros actúan como entradas de la función, y permiten que esta realice operaciones o cálculos utilizando datos específicos. Estos

parámetros se definen cuando la función es creada, y se rellenan cuando se invoca la función.

Por ejemplo, imagina que tienes una función para calcular el área de un círculo. La función necesitaría el radio del círculo como parámetro para poder realizar el cálculo:

```
1 def calcular_area(radio):  
2     return 3.1416 * radio * radio
```

Aquí, `radio` es un parámetro que se pasa a la función cuando se llama. El valor de `radio` es utilizado dentro de la función para calcular el área, y el resultado se devuelve con la sentencia `return`.

Una vez realizado el cálculo, la función puede devolver un valor mediante la sentencia `return`. Este valor puede ser utilizado por otra parte del programa, ya sea para realizar más cálculos o para mostrar el resultado al usuario. Cuando se llama a una función, el programa va a esa función, ejecuta su código y luego vuelve al lugar donde la función fue invocada, trayendo consigo el valor que la función ha retornado.

Por ejemplo:

```
1 def calcular_area(radio):  
2     return 3.1416 * radio * radio  
3  
4 area = calcular_area(5)  
5 print(area) # Imprime el área calculada
```

En este caso, la función `calcular_area` recibe el parámetro 5 como el valor de `radio`, realiza el cálculo, y devuelve el resultado. Este valor se captura en la variable `area` en el programa principal, y luego se imprime.

Paso de parámetros: paso por asignación. A diferencia de lenguajes como C (donde los parámetros se pasan por valor) o C++ (donde se pueden pasar por valor o por referencia), **en Python los parámetros se pasan por asignación** (también llamado *call by sharing* o *pass-by-object-reference*). Esto significa que cuando se llama a una función, el parámetro de la función queda *ligado al mismo objeto* al que apunta el argumento de la llamada: no se copia el objeto ni se crea un alias nuevo distinto.

La consecuencia práctica depende de si el objeto es *mutable* (listas, diccionarios, conjuntos, objetos definidos por el usuario) o *immutable* (enteros, flotantes, cadenas, tuplas, booleanos):

- **Si el objeto es immutable**, no puede modificarse en sitio. Cualquier asignación dentro de la función crea un nuevo objeto y reasigna el nombre local, pero el objeto original no cambia.
- **Si el objeto es mutable**, los cambios que se hagan a través de los métodos del objeto (por ejemplo, `append()` en una lista) *sí* se ven desde fuera, porque el objeto al que apuntan ambos nombres es el mismo.

Por ejemplo, una función que modifica una lista:

```

1 def modificar_lista(lista):
2     """Añade un nuevo elemento a la lista recibida."""
3     lista.append(100)
4
5 mi_lista = [1, 2, 3]
6 modificar_lista(mi_lista)
7 print(mi_lista)  # [1, 2, 3, 100]

```

Para evitar que una función modifique accidentalmente una estructura mutable que recibe, se puede trabajar con una copia y devolverla:

```

1 def modificar_lista_segura(lista):
2     """Devuelve una copia con un elemento adicional, sin
3     tocar la original."""
4     copia = lista.copy()
5     copia.append(100)
6     return copia
7
8 mi_lista = [1, 2, 3]
9 nueva_lista = modificar_lista_segura(mi_lista)
10 print(mi_lista)      # [1, 2, 3] (la original no cambia)
11 print(nueva_lista)   # [1, 2, 3, 100]

```

5.3. Ejercicios

EJERCICIO 5.1

Escribe un programa en Python que convierta temperaturas de grados Celsius a grados Fahrenheit. El programa debe solicitar al usuario la temperatura en Celsius y mostrar el valor equivalente en Fahrenheit (con un único decimal). Recuerda que la fórmula para calcular los grados Fahrenheit a partir de grados Celsius es

$$F = C \cdot \frac{9}{5} + 32.$$

Ejemplo de ejecución esperada

```

Introduce la temperatura en Celsius: 25
La temperatura en Fahrenheit es: 77.0

```

Solución (girar para leer)

```

1  def convertir_a_fahrenheit(celsius):
2      return (celsius * 9/5) + 32
3
4  def main():
5      celsius = float(input("Introduce la temperatura en
6      fahrenheit = convertir_a_fahrenheit(celsius)
7      print(f"La temperatura en fahrenheit es: {
8      fahrenheit:.1f}")
9
10     if __name__ == "__main__":
11         main()

```

Código 5.1: Solución al Ejercicio 5.1.

EJERCICIO 5.2

Escribe un programa en Python que reciba un número entero y determine si es par o impar. El programa debe mostrar un mensaje indicando si el número es par o impar.

Ejemplo de ejecución esperada

Introduce un número: 7
El número 7 es impar.

Solución (girar para leer)

```

13  def main():
12      if __name__ == "__main__":
11          print(f"El número {numero} es {resultado}")
10      resultado = es-par-o-impar(numero)
9          numero = int(input("Introduce un número: "))
8      def main():
7          :
6          :
5          return "impar"
4          else:
3              return "par"
2          if numero % 2 == 0:
1              def es-par-o-impar(numero):

```

Código 5.2: Solución al Ejercicio 5.2.

EJERCICIO 5.3

Escribe un programa en Python que calcule el área de un rectángulo. El programa debe solicitar al usuario la base y la altura del rectángulo y mostrar el área (con dos decimales).

Ejemplo de ejecución esperada

```
Introduce la base del rectángulo: 5
Introduce la altura del rectángulo: 8
El área del rectángulo es: 40.00
```

Solución (girar para leer)

```
1  def calcular_area_rectangulo(base, altura):
2      return base * altura
3
4  def main():
5      base = float(input("Introduce la base del rectángulo: "))
6      altura = float(input("Introduce la altura del rectángulo: "))
7      area = calcular_area_rectangulo(base, altura)
8      print(f"El área del rectángulo es: {area:.2f}")
9
10 if __name__ == "__main__":
11     main()
```

Código 5.3: Solución al Ejercicio 5.3.

EJERCICIO 5.4

Escribe un programa en Python que reciba tres números y determine cuál es el mayor de los tres. El programa debe solicitar al usuario los tres números y mostrar el número mayor. Incorpora un control de errores ante entradas incorrectas por parte del usuario.

Ejemplo de ejecución esperada

```
Introduce el primer número: 8
Introduce el segundo número: 12
Introduce el tercer número: 5
El número mayor es 12
```

Solución (girar para leer)

Código 5.4: Solución al Ejercicio 5.4.

```

1 def numero_mayor(a, b, c):
2     return max(a, b, c)
3     # max es una función ya existente en Python,
4     # el máximo de los parámetros pasados
5     devuelve
6 def main():
7     try:
8         a = float(input("Introduce el primer número: "))
9         b = float(input("Introduce el segundo número: "))
10        c = float(input("Introduce el tercer número: "))
11        mayor = numero_mayor(a, b, c)
12        print(f"El número mayor es {mayor}")
13    except ValueError:
14        print("Por favor, ingresa un número válido.")
15
16 if __name__ == "__main__":
17     main()
```

EJERCICIO 5.5

Escribe un programa en Python que determine si un número es primo. Un número es primo si sólo tiene dos divisores positivos: 1 y el mismo número. El programa debe solicitar al usuario un número y mostrar si es primo o no.

Ejemplo de ejecución esperada

Introduce un número: 17

El número 17 es primo.

Solución (girar para leer)

```

1  def es_primo(numero):
2      if numero < 2:
3          return False
4      for i in range(2, int(numero ** 0.5) + 1):
5          if numero % i == 0:
6              return False
7          return True
8
9  def main():
10     numero = int(input("Introduce un número: "))
11     if es_primo(numero):
12         print(f"El número {numero} es primo.")
13     else:
14         print(f"El número {numero} no es primo.")
15
16 if __name__ == "__main__":
17     main()

```

Código 5.5: Solución al Ejercicio 5.5.

EJERCICIO 5.6

Escribe un programa en Python que cuente cuántos dígitos tiene un número entero. El programa debe solicitar al usuario un número y mostrar la cantidad de dígitos.

Ejemplo de ejecución esperada

Introduce un número: 5421
El número tiene 4 dígitos.

Solución (girar para leer)

```

1 def contar_dígitos(numero):
2     return len(str(abs(numero)))
3     # lo transformamos a cadena y miramos su longitud
4
5 def main():
6     numero = input("Introduce un número: ")
7     cantidad_dígitos = contar_dígitos(numero)
8     print(f"El número tiene {cantidad_dígitos} dígitos.")
9
10 if __name__ == "__main__":
11     main()

```

Código 5.6: Solución al Ejercicio 5.6.

EJERCICIO 5.7

Escribe un programa en Python que genere un número aleatorio entre 1 y 100, y el usuario debe adivinarlo. El programa debe indicar si el número ingresado por el usuario es mayor o menor que el número aleatorio.

Ejemplo de ejecución esperada

```
Adivina el número (entre 1 y 100): 50
El número es mayor.
Adivina el número: 75
El número es menor.
Adivina el número: 62
¡Correcto! El número era 62.
```

Solución (girar para leer)

```
1 import random
2
3 def adivina_el_numero():
4     numero_aleatorio = random.randint(1, 100)
5     intento = None
6     while intento != numero_aleatorio:
7         intento = int(input("Adivina el número (entre 1
8         y 100): "))
9         if intento > numero_aleatorio:
10            print("El número es mayor.")
11        elif intento < numero_aleatorio:
12            print("El número es menor.")
13        print(f"¡Correcto! El número era {numero_aleatorio}
14        ")
15    def main():
16        adivina_el_numero()
17    if __name__ == "__main__":
18        main()
```

Código 5.7: Solución al Ejercicio 5.7.

EJERCICIO 5.8

Escribe un programa en Python que calcule el interés compuesto de una inversión a lo largo del tiempo. La fórmula es:

$$A = P \left(1 + \frac{r}{n} \right)^{nt}$$

Donde P es el capital inicial, r es la tasa de interés anual (en decimal), n es el número de veces que se aplica el interés por año, y t es el número de años.

El programa debe solicitar al usuario estos valores y mostrar el monto final de la inversión (con dos decimales). Recuerda que si el usuario introduce el porcentaje en formato entero (por ejemplo, 5 para un 5%), tendrás que dividirlo entre 100 antes de aplicar la fórmula.

Ejemplo de ejecución esperada

```
Introduce el capital inicial: 1000
Introduce la tasa de interés anual (en porcentaje): 5
Introduce el número de veces que se aplica el interés al año: 12
Introduce la cantidad de años: 10
El monto final después de 10 años será: 1647.01
```

Solución (girar para leer)

```

14 main()
13 if __name__ == "__main__":
12
11     {monto_final:.2f}"
10     print(f"El monto final después de {años} años será:
9     monto_final = calcular_interes_compuesto(principal,
8     años = int(input("Introduce la cantidad de años: "))
7     veces_por_año = int(input("Introduce el número de
6     tasa = float(input("Introduce la tasa de interés
5     inicial: "))
4     principal = float(input("Introduce el capital
3     def main():
2     return principal * (1 + tasa / veces_por_año) **
1     def calcular_interes_compuesto(principal, tasa,
    veces_por_año * años)

```

Código 5.8: Solución al Ejercicio 5.8.

EJERCICIO 5.9

Escribe un programa en Python que determine si un número ingresado por el usuario es capicúa. Un número capicúa se lee igual de izquierda a derecha que de derecha a izquierda.

Ejemplo de ejecución esperada

Introduce un número: 12321

El número 12321 es capicúa.

Solución (girar para leer)

```

1  def es_capicua(numero):
2      return str(numero) == str(numero)[::-1]
3
4  def main():
5      numero = int(input("Introduce un número: "))
6      if es_capicua(numero):
7          print(f"El número {numero} es capicúa.")
8      else:
9          print(f"El número {numero} no es capicúa.")
10
11  if __name__ == "__main__":
12      main()

```

Código 5.9: Solución al Ejercicio 5.9.

EJERCICIO 5.10

Escribe un programa en Python que calcule el Máximo Común Divisor (MCD) y el Mínimo Común Múltiplo (MCM) de dos números utilizando el algoritmo de Euclides.

Ejemplo de ejecución esperada

Introduce el primer número: 56
 Introduce el segundo número: 98
 El MCD de 56 y 98 es 14.
 El MCM de 56 y 98 es 392.

Solución (girar para leer)

```

1  def mcd(a, b):
2      while b != 0:
3          a, b = b, a % b
4      return a
5
6  def mcm(a, b, mcd):
7      return a * b // mcd
8
9  def main():
10     a = int(input("Introduce el primer número: "))
11     b = int(input("Introduce el segundo número: "))
12     resultado = mcd(a, b)
13     print(f"El MCD de {a} y {b} es {resultado}.")
14     resultado = mcm(a, b, resultado)
15     print(f"El MCM de {a} y {b} es {resultado}.")
16
17 if __name__ == "__main__":
18     main()

```

Código 5.10: Solución al Ejercicio 5.10.

EJERCICIO 5.11

Escribe un programa en Python que calcule la potencia de un número sin utilizar el operador **. El programa debe solicitar al usuario la base y el exponente, y calcular la base elevada al exponente mediante un bucle o recursión.

Ejemplo de ejecución esperada

```
Introduce la base: 3
Introduce el exponente: 4
El resultado de 3^4 es 81.
```

Solución (girar para leer)

```

1  def potencia(base, exponente):
2      resultado = 1
3      for _ in range(exponente):
4          # si la variable de iteración no se va a
5          # utilizar, puede
6          # usarse - en su lugar
7          resultado *= base
8      return resultado
9
10 def main():
11     base = int(input("Introduce la base: "))
12     exponente = int(input("Introduce el exponente: "))
13     resultado = potencia(base, exponente)
14     print(f"El resultado de {base}^{exponente} es {
15         resultado}")
16     if __name__ == "__main__":
17         main()

```

Código 5.11: Solución al Ejercicio 5.11.

EJERCICIO 5.12

Escribe un programa en Python que convierta un número entre 0 y 99 en su equivalente en texto. El programa debe manejar correctamente los números del 0 al 15 y las decenas (20, 30, etc.).

Ejemplo de ejecución esperada

Introduce un número entre 0 y 99: 42
El número en texto es: cuarenta y dos.

Solución (girar para leer)

```

1 def convertir_a_texto(numero):
2     # para resolver este ejercicio usamos listas, se
3     unidades = ["", "uno", "dos", "tres", "cuatro", "
4     cinco", "seis", "siete", "ocho", "nueve"]
5     decenas = ["", "diez", "veinte", "treinta", "
6     cuarenta", "cincuenta", "sesenta", "setenta", "
7     ochenta", "noventa"]
8     if numero < 10:
9         return unidades[numero]
10    elif numero > 30:
11        return f"veinti{unidades[numero-20]}"
12    else:
13        return f"{decenas[numero // 10]} y {unidades[
14        numero % 10]}"
15    def main():
16        numero = int(input("Introduce un número entre 0 y
17        99: "))
18        texto = convertir_a_texto(numero)
19        print(f"El número en texto es: {texto}.")
20    if __name__ == "__main__":
21        main()

```

Código 5.12: Solución al Ejercicio 5.12.

EJERCICIO 5.13

Escribe un programa en Python que determine si un número es perfecto. Un número perfecto es aquel cuya suma de sus divisores propios (sin incluir el propio número) es igual al número.

Ejemplo de ejecución esperada

Introduce un número: 28

El número 28 es perfecto porque $1 + 2 + 4 + 7 + 14 = 28$.

Solución (girar para leer)

```

1  def obtener_divisores(numero):
2      return [i for i in range(1, numero) if numero % i
3              == 0]
4  def es_numero_perfecto(numero):
5      return sum(obtener_divisores(numero)) == numero
6
7  def main():
8      numero = input("Introduce un número: ")
9      if es_numero_perfecto(numero):
10         divisores = obtener_divisores(numero)
11         suma = " + ".join(str(d) for d in divisores)
12         print(f"El número {numero} es perfecto porque {
13             suma} = {numero}.")
14     else:
15         print(f"El número {numero} no es perfecto.")
16 if __name__ == "__main__":
17     main()

```

Código 5.13: Solución al Ejercicio 5.13.

EJERCICIO 5.14

Escribe un programa en Python que reciba un número entero positivo y calcule la suma de sus dígitos. El programa debe solicitar al usuario un número y mostrar la suma de sus dígitos.

Ejemplo de ejecución esperada

Introduce un número: 5421

La suma de los dígitos es: 12

Solución (girar para leer)

```
1 def suma_digitos(numero):
2     suma = 0
3     while numero > 0:
4         suma += numero % 10
5         numero = numero // 10
6     return suma
7
8 def main():
9     numero = int(input("Introduce un número: "))
10    resultado = suma_digitos(numero)
11    print(f"La suma de los dígitos es: {resultado}")
12
13 if __name__ == "__main__":
14     main()
```

Código 5.14: Solución al Ejercicio 5.14.

EJERCICIO 5.15

Escribe un programa en Python que reciba dos matrices de 2x2 y calcule su suma. El programa debe solicitar al usuario los elementos de las dos matrices y mostrar la matriz resultante de la suma.

Ejemplo de ejecución esperada

Introduce los elementos de la primera matriz:

Introduce el elemento [1][1]: 1

Introduce el elemento [1][2]: 2

Introduce el elemento [2][1]: 3

Introduce el elemento [2][2]: 4

Introduce los elementos de la segunda matriz:

Introduce el elemento [1][1]: 5

Introduce el elemento [1][2]: 6

Introduce el elemento [2][1]: 7

Introduce el elemento [2][2]: 8

La suma de las matrices es:

6 8

10 12

Solución (girar para leer)

```

1  def suma_matrices(matriz1, matriz2):
2      matriz_resultante = []
3      for i in range(2):
4          fila = []
5          for j in range(2):
6              fila.append(matriz1[i][j] + matriz2[i][j])
7          matriz_resultante.append(fila)
8      return matriz_resultante
9
10 def main():
11     matriz1 = []
12     matriz2 = []
13
14     print("Introduce los elementos de la primera matriz")
15     for i in range(2):
16         fila = []
17         for j in range(2):
18             fila.append(int(input(f"Introduce el
19 elemento [{i+1}][{j+1}]: ")))
20         matriz1.append(fila)
21
22     print("Introduce los elementos de la segunda
23 matriz:")
24     for i in range(2):
25         fila.append(int(input(f"Introduce el
26 elemento [{i+1}][{j+1}]: ")))
27         matriz2.append(fila)
28
29     matriz_resultante = suma_matrices(matriz1, matriz2)
30     print("\nLa suma de las matrices es:")
31     for fila in matriz_resultante:
32         print(" ".join(map(str, fila)))
33
34 if __name__ == "__main__":
35     main()

```

Código 5.15: Solución al Ejercicio 5.15.

EJERCICIO 5.16

Escribe un programa en Python que genere la secuencia de Fibonacci hasta el n-ésimo término. El programa debe solicitar al usuario el valor de n y mostrar la secuencia de Fibonacci hasta ese término.

Ejemplo de ejecución esperada

Introduce un número: 6

La secuencia de Fibonacci hasta el término 6 es: 0 1 1 2 3 5

Solución (girar para leer)

```

1  def fibonacci(n):
2      a, b = 0, 1
3      secuencia = []
4      for _ in range(n):
5          secuencia.append(a)
6          a, b = b, a + b
7      return secuencia
8
9  def main():
10     n = int(input("Introduce un número: "))
11     secuencia = fibonacci(n)
12     print(f"La secuencia de Fibonacci hasta el término
    {n} es: {' '.join(map(str, secuencia))}")
13     # esto se podría haber hecho con un bucle for:
14     # for elemento in secuencia:
15         # print(elemento, end=' ')
16
17     if __name__ == "__main__":
18         main()

```

Código 5.16: Solución al Ejercicio 5.16.

EJERCICIO 5.17

Escribe un programa en Python que cuente cuántas vocales (mayúsculas y minúsculas) contiene una cadena de texto. El programa debe solicitar al usuario una cadena y mostrar la cantidad de vocales encontradas.

Ejemplo de ejecución esperada

Introduce una cadena: Hola Mundo
La cadena contiene 4 vocales.

Solución (girar para leer)

```

1  def contar_vocales(cadena):
2      contador = 0
3      for char in cadena:
4          if char in "aeiouAEIOU":
5              contador += 1
6      return contador
7
8  def main():
9      cadena = input("Introduce una cadena: ")
10     cantidad_vocales = contar_vocales(cadena)
11     print(f"La cadena contiene {cantidad_vocales} vocales.")
12
13     if __name__ == "__main__":
14         main()

```

Código 5.17: Solución al Ejercicio 5.17.

EJERCICIO 5.18

Escribe un programa en Python que reciba una cadena de texto e imprima la cadena invertida. El programa debe solicitar al usuario una cadena y mostrar la versión invertida de la misma.

Ejemplo de ejecución esperada

Introduce una cadena: Python

La cadena invertida es: nohtyP

Solución (girar para leer)

```

1  def invertir_cadena(cadena):
2      cadena_invertida = ""
3      for i in range(len(cadena) - 1, -1, -1):
4          cadena_invertida += cadena[i]
5      return cadena_invertida
6
7  def main():
8      cadena = input("Introduce una cadena: ")
9      cadena_invertida = invertir_cadena(cadena)
10     print(f"La cadena invertida es: {cadena_invertida}")
11
12     if __name__ == "__main__":
13         main()

```

Código 5.18: Solución al Ejercicio 5.18.

EJERCICIO 5.19

Escribe un programa en Python que convierta un número binario ingresado por el usuario a su equivalente en octal. El programa debe solicitar al usuario un número binario y mostrar el valor en octal.

Ejemplo de ejecución esperada

Introduce un número binario: 1011

El número binario 1011 en octal es: 13

Solución (girar para leer)

```

1  def binario_a_octal(binario):
2      decimal = 0
3      for i in range(len(binario)):
4          decimal += int(binario[i]) * (2 ** (len(binario)
5              - i - 1))
6          decimal > 0:
7              octal = decimal // 8
8              decimal = decimal % 8
9              octal = octal * 10
10         return octal
11     def main():
12         binario = input("Introduce un número binario: ")
13         octal = binario_a_octal(binario)
14         print(f"El número binario {binario} en octal es: {
15             octal}")
16     if __name__ == "__main__":
17         main()

```

Código 5.19: Solución al Ejercicio 5.19.

EJERCICIO 5.20

Escribe un programa en Python que reciba una lista de números y calcule su promedio. El programa debe solicitar al usuario los números, los almacenará en una lista y luego calculará el promedio de los mismos.

Ejemplo de ejecución esperada

Introduce una lista de números separados por espacio: 3 2 1 4 55 12 42 8
El promedio de los números introducidos es: 15.88

Solución (girar para leer)

```

1 def calcular_promedio(lista):
2     suma = 0
3     for num in lista:
4         suma += num
5     promedio = suma / len(lista)
6     return promedio
7
8 def main():
9     numeros = input("Introduce una lista de números
10    separados por espacio: ").split()
11    # lee los números, separados por el espacio, guarda
12    # los en una lista
13    numeros = [float(num) for num in numeros]
14    # convierte a número real
15    # devuelve siempre una cadena
16    if len(numeros) == 0:
17        print("No se introdujeron números.")
18    else:
19        promedio = calcular_promedio(numeros)
20        print(f"El promedio de los números introducidos
21        es: {promedio:.2f}")
22    if __name__ == "__main__":
23        main()

```

Código 5.20: Solución al Ejercicio 5.20.