

6

Desarrollo modular y diseño descendente

En este capítulo nos centraremos en el diseño de programas modulares en Python: aprenderemos a organizar el código en funciones reutilizables y módulos independientes, y a gestionar la interacción entre ellos. Este enfoque facilita la creación de aplicaciones más mantenibles, escalables y comprensibles.

La programación modular (o desarrollo modular) implica dividir un programa grande en componentes más pequeños, llamados *módulos*, que pueden ser desarrollados, probados y depurados de manera independiente. Cada módulo realiza una parte del trabajo del programa, y estos módulos interactúan entre sí para cumplir el objetivo global. En Python, los módulos son archivos `«.py»` que contienen funciones, clases o variables; los paquetes (que se introducirán más adelante) son directorios que agrupan varios módulos.

El desarrollo modular tiene múltiples ventajas: la *reutilización* del código (los módulos pueden reutilizarse en distintos programas o proyectos), la *facilidad de mantenimiento* (un fallo en un módulo puede corregirse sin afectar al resto del programa), el *trabajo en equipo* (módulos distintos pueden ser desarrollados en paralelo por distintas personas) y la *facilidad de pruebas* (cada módulo puede probarse por separado, asegurando su corrección antes de la integración).

El enfoque opuesto se denomina *programación monolítica*: el código se escribe en un solo bloque, sin una división clara en módulos independientes. Este enfoque puede ser útil para programas muy pequeños y rápidos de escribir, pero a medida que el código crece se vuelve difícil de gestionar y propenso a errores difíciles de localizar. En la práctica, salvo casos triviales, conviene adoptar siempre un enfoque modular.

6.1. Diseño descendente

El diseño descendente (también conocido como *top-down design*) es la técnica de diseño que mejor encaja con el desarrollo modular. Consiste en partir de una visión general del problema e ir descomponiéndolo progresivamente en partes más pequeñas y manejables. En programación, esto se traduce en definir primero la funcionalidad global del programa y, después, ir detallando los módulos o funciones necesarios para llevarla a cabo.

El proceso de diseño descendente consta de los siguientes pasos:

- 1.- Identificación de las necesidades:** comienza con una descripción general de lo que debe hacer el programa.
- 2.- Descomposición en partes más pequeñas:** divide las tareas globales en sub-tareas o funciones.
- 3.- Definición de módulos:** para cada sub-tarea, defines un módulo o función que resuelva un aspecto del problema global.
- 4.- Refinamiento continuo:** añade detalles de implementación gradualmente, a medida que profundizas en cada módulo.

Esta técnica favorece la claridad en la planificación (al partir de una visión general, se asegura que el programa cumpla los requisitos antes de entrar en detalles complejos), la organización (cada módulo tiene una tarea clara y bien definida) y la flexibilidad (al descomponer el problema en partes pequeñas, es más fácil modificar o mejorar una parte sin afectar al sistema completo).

En Python, el diseño descendente se implementa creando funciones y módulos que resuelvan cada parte del problema. Lo habitual es que un programa comience con una función principal, llamada por convención `main()`, que define el flujo general; a partir de ahí, se invocan funciones de niveles inferiores que se ocupan de tareas cada vez más específicas.

6.2. Uso de `import`: módulos y paquetes

Para que un programa se pueda partir en varios ficheros y aprovechar el código de otros, Python ofrece la sentencia `import`. La sentencia `import` se usa para *traer* código de otro archivo o módulo y poder utilizar las funciones, clases y variables definidas en él.

6.2.1. Importar un módulo completo

Si en un fichero llamado `«calculos.py»` tenemos definidas varias funciones, podemos importarlo desde otro fichero del mismo directorio con:

```
import calculos
```

Tras este `import`, todas las funciones del módulo están accesibles a través del nombre del módulo, seguido del operador punto y el nombre de la función:

```
calculos.es_primo(numero)
```

6.2.2. Importar nombres concretos

Si solo se necesitan algunas funciones de un módulo, se pueden importar de forma selectiva con la sintaxis `from ... import ...`:

```
from calculos import es_primo
```

Tras esto, la función `es_primo` se utiliza directamente, sin prefijar con el nombre del módulo:

```
es_primo(numero)
```

El antipatrón `from modulo import *`. Existe la variante `from calculos import *`, que importa todos los nombres públicos del módulo de golpe. **En general es una mala práctica:** contamina el espacio de nombres del programa con identificadores cuya procedencia no queda explícita, dificultando la lectura y aumentando el riesgo de colisiones (un nombre del módulo importado puede pisar otro nombre que ya tuvieras definido). Salvo en sesiones interactivas o casos muy concretos (módulos diseñados explícitamente para esto, como ciertas bibliotecas matemáticas), conviene evitar el `import *`.

6.2.3. Alias al importar

Para abreviar nombres largos o evitar colisiones, se puede dar un alias al módulo o a un nombre importado mediante la palabra clave `as`:

```
import calculos as calc
calc.es_primo(numero)
```

Esta forma es muy habitual al trabajar con bibliotecas de terceros: por ejemplo, `import numpy as np` o `import pandas as pd` son convenciones casi universales en el ecosistema científico de Python (Capítulo 12).

6.2.4. Dónde busca Python los módulos: `sys.path`

Cuando se ejecuta un `import nombre`, Python busca un fichero `«nombre.py»` (o un paquete `«nombre/»`) en una lista ordenada de directorios accesible mediante `sys.path`. Esa lista se inicializa con:

- el directorio del script que se está ejecutando,
- los directorios indicados en la variable de entorno `PYTHONPATH` (si está definida), y
- los directorios de la biblioteca estándar y de los paquetes instalados.

Si un `import` falla con `ModuleNotFoundError`, lo primero que conviene comprobar es si el fichero está en un directorio incluido en `sys.path`:

```
import sys
print(sys.path)
```

6.2.5. Paquetes: módulos agrupados en directorios

Cuando un proyecto crece, los módulos sueltos no bastan: conviene agruparlos en *paquetes*. Un paquete es, simplemente, un directorio que contiene varios módulos relacionados y un fichero especial llamado `«__init__.py»` (que puede estar vacío). Su presencia indica a Python que ese directorio debe tratarse como un paquete y permite importar sus módulos con la notación de punto:

```
mi_proyecto/
|-- main.py
'-- utilidades/
    |-- __init__.py
    |-- matematicas.py
    '-- cadenas.py
```

Desde `«main.py»`, podemos importar funciones del paquete así:

```
from utilidades.matematicas import factorial
import utilidades.cadenas as cadenas
```

A partir de Python 3.3, los *namespace packages* (paquetes sin `«__init__.py»`) también funcionan, pero el uso explícito de `«__init__.py»` sigue siendo la convención más clara y compatible.

6.3. Caso práctico: programa dirigido por menú

Una de las maneras más comunes de estructurar un programa interactivo es a través de un programa dirigido por menú, en el que se presenta al usuario un conjunto de opciones y el flujo del programa depende de la elección que haga. Aplicaremos a este caso el enfoque modular y descendente: comenzaremos con la estructura de alto nivel (presentar el menú, leer la opción, ejecutar la operación) y descompondremos en módulos cada operación específica.

Cada opción del menú estará asociada a una función independiente que realice una tarea concreta. Por ejemplo, una opción podría ser calcular el número de cifras de un entero, encapsulada en una función `num_cifras()`. La interacción con el usuario se gestiona de forma centralizada en un módulo principal, que delega cada operación específica al módulo que la implementa.

6.3.1. Análisis del problema

Vamos a desarrollar un programa interactivo que presente un menú con varias opciones para realizar cálculos con números enteros. El menú debe mostrarse iterativamente hasta que el usuario decida finalizar el programa. Un ejemplo de ejecución es:

Ejemplo de ejecución esperada

```

MENÚ DE OPERACIONES
=====
0 - Finalizar
1 - Calcular el número de cifras de un entero
2 - Sumar las cifras de un entero
3 - Extraer una cifra de un entero
4 - Calcular la imagen especular de un entero
5 - Comprobar si un número es primo

Seleccione una operación [0-5]: 4
Escriba un número entero: 8802361
El número imagen especular de 8802361 es 1632088.

```

Aplicando el diseño descendente, dividiremos el programa en dos ficheros:

- El módulo de funciones de cálculo (fichero «`calculos.py`»), que contiene las funciones que realizan las operaciones matemáticas con enteros.
- El módulo principal (fichero «`calculadora.py`»), encargado de manejar la interacción con el usuario y coordinar la ejecución de las operaciones.

6.3.2. Módulo de funciones de cálculo

Este módulo contiene únicamente las funciones de cálculo, sin lógica de interacción con el usuario:

```

1  # Fichero calculos.py
2
3  def num_cifras(n: int) -> int:
4      """Devuelve el número de cifras de `n` en base 10."""
5      return len(str(abs(n)))
6
7
8  def suma_cifras(n: int) -> int:
9      """Devuelve la suma de las cifras de `n` en base 10."""
10     return sum(int(digito) for digito in str(abs(n)))
11
12

```

```
13 def extraer_cifra(n: int, i: int) -> int:
14     """Devuelve la `i`-ésima cifra menos significativa de `n`
    ↪     `.`."""
15     n = str(abs(n))
16     if 1 <= i <= len(n):
17         return int(n[-i])
18     else:
19         raise ValueError("Índice fuera de rango")
20
21
22 def imagen_especular(n: int) -> int:
23     """Devuelve el número que representa la imagen especular
    ↪     de `n`."""
24     signo = -1 if n < 0 else 1
25     return signo * int(str(abs(n))[::-1])
26
27
28 def es_primo(n: int) -> bool:
29     """Devuelve True si `n` es un número primo, False en caso
    ↪     contrario."""
30     if n < 2:
31         return False
32     if n in (2, 3):
33         return True
34     if n % 2 == 0 or n % 3 == 0:
35         return False
36     i = 5
37     while i * i <= n:
38         if n % i == 0 or n % (i + 2) == 0:
39             return False
40         i += 6
41     return True
42
```

```

43 if __name__ == "__main__":
44     # Pruebas unitarias básicas
45     assert num_cifras(12345) == 5
46     assert num_cifras(-999) == 3
47     assert suma_cifras(123) == 6
48     assert suma_cifras(-555) == 15
49     assert extraer_cifra(98765, 1) == 5
50     assert extraer_cifra(98765, 3) == 7
51     assert imagen_especular(123) == 321
52     assert imagen_especular(-456) == -654
53     assert es_primo(2) is True
54     assert es_primo(9) is False
55     assert es_primo(17) is True
56
57     print("Todas las pruebas pasaron correctamente.")

```

Cada operación es una función simple que toma un parámetro (n), realiza la operación correspondiente y retorna el resultado. La función `es_primo()` está optimizada respecto a la forma elemental: tras descartar los casos pequeños y los múltiplos de 2 y 3, comprueba divisores de la forma $6k \pm 1$ hasta $\sqrt{n}$, lo que reduce significativamente el número de divisiones.

Algunas funciones realizan comprobaciones y, si no se cumplen, generan una excepción mediante la sentencia `raise` (introducida en el Capítulo 3).

Pruebas unitarias rápidas con `assert`. Al final del módulo aparecen una serie de sentencias `assert`. La sentencia `assert` verifica que una condición se cumple en tiempo de ejecución: si es `True`, el programa continúa normalmente; si es `False`, se genera una excepción `AssertionError` y la ejecución se detiene. Es útil para escribir pruebas unitarias rápidas sin necesidad de un marco de pruebas como `unittest` o `pytest`. En el módulo «`calculos.py`», cada `assert` comprueba que una función devuelve el resultado esperado para una entrada concreta, facilitando la detección temprana de fallos.

¡Atención! `assert` y modo optimizado

Cuando Python se ejecuta con la opción `-O` (modo optimizado, por ejemplo `python -O script.py`), **todas las sentencias `assert` se ignoran**. Por tanto, `assert` es adecuado para pruebas y comprobaciones de invariantes durante el desarrollo, pero **no debe usarse para validar entradas del usuario, datos externos o requisitos de seguridad**: en producción podría desactivarse y la validación sería ignorada. Para esos casos, lanzar excepciones explícitamente con `raise` es la opción correcta.

6.3.3. Módulo principal

El módulo principal gestiona la interacción con el usuario y delega los cálculos al módulo «`calculos.py`»:

```
1 # Fichero calculadora.py
2
3 import calculos
4
5
6 def mostrar_menu():
7     """Muestra el menú de opciones disponibles."""
8     print("\nMENÚ DE OPERACIONES")
9     print("=====")
10    print("0 - Finalizar")
11    print("1 - Calcular el número de cifras de un entero")
12    print("2 - Sumar las cifras de un entero")
13    print("3 - Extraer una cifra de un entero")
14    print("4 - Calcular la imagen especular de un entero")
15    print("5 - Comprobar si un número es primo")
16
17
18 def pedir_opcion() -> int:
19     """Solicita al usuario una opción del menú."""
20     while True:
21         try:
22             opcion = int(input("Seleccione una operación
23 ↪ [0-5]: "))
24             if 0 <= opcion <= 5:
25                 return opcion
26             else:
27                 print("Por favor, ingrese un número entre 0 y
28 ↪ 5.")
29         except ValueError:
30             print("Entrada no válida. Ingrese un número
31 ↪ entero.")
```

```
31 def ejecutar_operacion(opcion: int):
32     """Ejecuta la operación correspondiente según la opción
    ↪     seleccionada."""
33     if opcion == 0:
34         print("Finalizando el programa.")
35         return False
36
37     try:
38         numero = int(input("Escriba un número entero: "))
39     except ValueError:
40         print("Entrada no válida. Debe ingresar un número
    ↪         entero.")
41         return True
42
43     if opcion == 1:
44         print(f"El número {numero} tiene {calculos.num_cifras
    ↪         (numero)} cifras.")
45     elif opcion == 2:
46         print(f"Las cifras de {numero} suman {calculos.
    ↪         suma_cifras(numero)}.")
47     elif opcion == 3:
48         try:
49             posicion = int(input("Seleccione la posición de
    ↪             una cifra: "))
50             print(f"La cifra en la posición {posicion} de {
    ↪             numero} es {calculos.extraer_cifra(numero,
    ↪             posicion)}.")
51         except ValueError as e:
52             print(e)
53     elif opcion == 4:
54         print(f"El número imagen especular de {numero} es {
    ↪         calculos.imagen_especular(numero)}.")
55     elif opcion == 5:
56         es_primo = calculos.es_primo(numero)
57         print(f"El número {numero} {'es' if es_primo else 'no
    ↪         es'} primo.")
58
59     return True
60
61
```

```
62 def main():
63     """Función principal que ejecuta el programa interactivo.
    ↪     """
64     while True:
65         mostrar_menu()
66         opcion = pedir_opcion()
67         if not ejecutar_operacion(opcion):
68             break
69
70
71 if __name__ == "__main__":
72     main()
```

La función `main()` gestiona el flujo general: muestra el menú con `mostrar_menu()`, lee la opción del usuario con `pedir_opcion()` y, si la opción no es la de finalización, llama a `ejecutar_operacion()` para ejecutar el cálculo correspondiente. Las excepciones se capturan a nivel de la interacción con el usuario para que un fallo de entrada no termine bruscamente el programa.

6.3.4. Niveles de abstracción

El diseño anterior sigue una metodología descendente con tres niveles de abstracción claramente diferenciados:

Nivel 1 (*nivel superior de abstracción*): la función `main()`, que interactúa con el usuario y gestiona el flujo de ejecución del programa.

Nivel 2 (*segundo nivel de abstracción*): las funciones `pedir_opcion()` y `ejecutar_operacion()`, que gestionan la entrada del usuario y la coordinación de las operaciones.

Nivel 3 (*nivel inferior de abstracción*): las funciones del módulo `«calculos.py»` (`num_cifras()`, `suma_cifras()`, etc.), que implementan los detalles técnicos de cada cálculo matemático.

Este tipo de descomposición es lo que hace que el código sea fácil de leer (se entiende primero a alto nivel y solo se baja al detalle cuando hace falta) y fácil de mantener: si más adelante se quisiera optimizar el cálculo de primalidad, solo habría que tocar la función `es_primo()` del módulo de cálculos, sin afectar al resto del programa.

6.4. Ejercicios

EJERCICIO 6.1

Escribe un programa en Python que verifique si una cadena de texto es un pangrama, es decir, si contiene todas las letras del alfabeto al menos una vez. El programa debe eliminar las mayúsculas y los espacios antes de verificar la condición. El flujo de trabajo del programa debe seguir un diseño descendente, comenzando con la función `main()` y desglosándose en módulos más pequeños para verificar si la cadena es un pangrama.

Ejemplo de ejecución esperada

Introduce una cadena de texto: El veloz murciélago hindú comía feliz un cardillo y kiwi,
La cadena es un pangrama.

Solución (girar para leer)

```

1 import string
2
3 def es_pangrama(cadena: str) -> bool:
4     """Verifica si una cadena es un pangrama (contiene
5     todas las letras del alfabeto)."""
6     cadena = cadena.lower() # Convertir a minúsculas
7     alfabeto = set(string.ascii_lowercase) # Alfabeto
8     en minúsculas
9     return set(cadena) >= alfabeto # Comparar si la
10    cadena contiene todas las letras
11
12 def main():
13     texto = input("Introduce una cadena de texto: ")
14     if es_pangrama(texto):
15         print("La cadena es un pangrama.")
16     else:
17         print("La cadena no es un pangrama.")
18     if __name__ == "__main__":
19         main()

```

Código 6.1: Solución al Ejercicio 6.1.

EJERCICIO 6.2

Escribe un programa en Python que reciba un texto y cuente cuántas veces aparece cada palabra en el texto. El programa debe ser modular, con una función `contar_palabras()` que se encargue de contar las palabras y una función principal `main()` que gestione la entrada y salida del usuario.

Ejemplo de ejecución esperada

Introduce un texto: hola mundo hola

La frecuencia de las palabras es: 'hola': 2, 'mundo': 1

Solución (girar para leer)

```

1  import string
2
3  def contar_palabras(texto: str) -> dict:
4      """Cuenta la frecuencia de cada palabra en el texto"""
5      texto = texto.lower().translate(str.maketrans("", "",
6      " , string.punctuation)) # Eliminar puntuación
7      palabras = texto.split() # Dividir el texto en
8      frecuencia = {}
9      for palabra in palabras:
10         if palabra in frecuencia:
11             frecuencia[palabra] += 1
12         else:
13             frecuencia[palabra] = 1
14         return frecuencia
15
16 def main():
17     texto = input("Introduce un texto: ")
18     resultado = contar_palabras(texto)
19     print("La frecuencia de las palabras es:", resultado)
20
21 if __name__ == "__main__":
22     main()

```

Código 6.2: Solución al Ejercicio 6.2.

EJERCICIO 6.3

Escribe un programa en Python que calcule la suma de los dígitos de un número de manera recursiva. El programa debe tener un diseño modular, con una función `suma_digitos()` que realice la recursión y una función principal `main()` que gestione la entrada del usuario.

Ejemplo de ejecución esperada

Introduce un número: 9875

La suma de los dígitos es 29

Solución (girar para leer)

```

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

```

Código 6.3: Solución al Ejercicio 6.3.

EJERCICIO 6.4

Escribe un programa en Python que determine si un número tiene exactamente tres divisores (es decir, si es un número casi primo). Además, el programa debe contar cuántos números casi primos existen hasta un valor límite proporcionado por el usuario. El programa debe ser modular, con una función `es_casi_primo()` que determine si un número es casi primo, y una función principal `main()` para gestionar el flujo general.

Ejemplo de ejecución esperada

Introduce el límite: 10

La cantidad de números casi primos hasta 10 es: 2

Solución (girar para leer)

```

1  def es_casi_primo(n: int) -> bool:
2      """Determina si un número tiene exactamente
3      divisores (es casi primo)."""
4      divisores = 0
5      for i in range(1, n + 1):
6          if n % i == 0:
7              divisores += 1
8          return divisores == 3
9  def contar_casi_primos(límite: int) -> int:
10     """Cuenta cuántos números casi primos hay hasta el
11     límite proporcionado."""
12     contador = 0
13     for i in range(1, límite + 1):
14         if es_casi_primo(i):
15             contador += 1
16     return contador
17 def main():
18     límite = int(input("Introduce el límite: "))
19     cantidad = contar_casi_primos(límite)
20     print(f"La cantidad de números casi primos hasta {
21     límite} es: {cantidad}")
22 if __name__ == "__main__":
23     main()

```

Código 6.4: Solución al Ejercicio 6.4.

EJERCICIO 6.5

Escribe un programa en Python que reciba un número entero y determine la mayor secuencia de dígitos consecutivos dentro de ese número. El programa debe estar diseñado de manera modular, con una función `mayor_secuencia_consecutiva()` que encuentre la secuencia y una función principal `main()` que gestione la entrada y salida de datos.

Ejemplo de ejecución esperada

Introduce un número: 123456789987654

La mayor secuencia de dígitos consecutivos es 123456789

Solución (girar para leer)

```

1 def mayor-secuencia-consecutiva(n: int) -> str:
2     """Determina la mayor secuencia de dígitos
3     consecutivos en un número."""
4     num_str = str(n)
5     max-secuencia = ""
6     secuencia-actual = num_str[0]
7     for i in range(1, len(num_str)):
8         if int(num_str[i]) == int(num_str[i-1]) + 1:
9             secuencia-actual += num_str[i]
10        else:
11            if len(secuencia-actual) > len(
12                max-secuencia):
13                max-secuencia = secuencia-actual
14            secuencia-actual = num_str[i]
15        if len(secuencia-actual) > len(max-secuencia):
16            max-secuencia = secuencia-actual
17    return max-secuencia
18
19 def main():
20     numero = int(input("Introduce un número: "))
21     secuencia = mayor-secuencia-consecutiva(numero)
22     print(f"La mayor secuencia de dígitos consecutivos
23     es {secuencia}")
24
25 if __name__ == "__main__":
26     main()

```

Código 6.5: Solución al Ejercicio 6.5.

EJERCICIO 6.6

↳ Escribe un programa en Python que implemente una biblioteca de cifrado César y descifrado. La biblioteca debe contener dos funciones: `cifrar_cesar(mensaje: str, desplazamiento: int)` y `descifrar_cesar(mensaje_cifrado: str, desplazamiento: int)`. Luego, el programa debe solicitar al usuario un mensaje, cifrarlo con un desplazamiento dado, mostrar el mensaje cifrado y luego descifrarlo para verificar que se obtiene el mensaje original.

Ejemplo de ejecución esperada

```
Introduce el mensaje: este es un mensaje secreto
Introduce el desplazamiento: 3
Mensaje cifrado: hvwh hv xq phqvmh vhfuhwr
Mensaje descifrado: este es un mensaje secreto
```

Solución (girar para leer)

```

1  def cifrar_cesar(mensaje: str, desplazamiento: int) -> str
2  """Cifra un mensaje usando el cifrado César con un
3  desplazamiento dado."""
4  for char in mensaje:
5      if char.isalpha():
6          # Determina si la letra es mayúscula o minú
7          scula
8          base = 65 if char.isupper() else 97
9          # Realiza el desplazamiento y asegura que
10         se mantenga dentro del alfabeto
11         mensaje_cifrado += chr((ord(char) - base +
12         desplazamiento) % 26 + base)
13     else:
14         mensaje_cifrado += char # Si no es una
15         letra, lo añade tal cual
16     return mensaje_cifrado
17
18 def descifrar_cesar(mensaje_cifrado: str,
19                     desplazamiento: int) -> str:
20     """Descifra un mensaje cifrado con el cifrado César
21     ."""
22     return cifrar_cesar(mensaje_cifrado, -
23                         desplazamiento)
24
25 def main():
26     mensaje = input("Introduce el mensaje: ")
27     desplazamiento = int(input("Introduce el
28     desplazamiento: "))
29     # Cifrar el mensaje
30     mensaje_cifrado = cifrar_cesar(mensaje,
31     desplazamiento)
32     # Descifrar el mensaje
33     mensaje_descifrado = descifrar_cesar(
34     mensaje_cifrado, desplazamiento)
35     print(f"Mensaje cifrado: {mensaje_cifrado}")
36     print(f"Mensaje descifrado: {mensaje_descifrado}")
37
38 if __name__ == "__main__":
39     main()

```

EJERCICIO 6.7

↳ Escribe un programa en Python que implemente una biblioteca para manipulación de series numéricas. La biblioteca debe contener tres funciones: `es_fibonacci(n: int)` que determina si un número pertenece a la serie de Fibonacci, `es_primo(n: int)` que verifica si un número es primo, y `serie_fibonacci(n: int)` que imprima por pantalla los primeros n números de la serie de Fibonacci. Luego, el programa debe solicitar al usuario un número para verificar si pertenece a la serie de Fibonacci y si es primo, y mostrar los primeros n números de la serie de Fibonacci.

Ejemplo de ejecución esperada

```
Introduce un número: 13
El número 13 es parte de la serie de Fibonacci.
El número 13 es primo.
Introduce cuántos términos de la serie Fibonacci deseas ver: 8
Serie de Fibonacci: [0, 1, 1, 2, 3, 5, 8, 13]
```

Solución (girar para leer)

```

1 import math
2
3 def es_fibonacci(n: int) -> bool:
4     """Verifica si un número pertenece a la serie de
5     Fibonacci usando la fórmula matemática."""
6     if n > 0:
7         return False
8     # Verifica si  $5*n^2 + 4$  o  $5*n^2 - 4$  son cuadrados
9     # perfectos
10    return es_cuadrado_perfecto(5 * n * n + 4) or
11    es_cuadrado_perfecto(5 * n * n - 4)
12
13 def es_cuadrado_perfecto(n: int) -> bool:
14     """Verifica si un número es un cuadrado perfecto."""
15     return int(math.sqrt(n)) ** 2 == n
16
17 def es_primo(n: int) -> bool:
18     """Verifica si un número es primo."""
19     if n <= 1:
20         return False
21     for i in range(2, int(math.sqrt(n)) + 1):
22         if n % i == 0:
23             return False
24         return True

```

Código 6.7: Solución al Ejercicio 6.7.

Solución (girar para leer)

```

22 def serie_fibonacci(n: int) -> list:
23     """Genera los primeros `n` números de la serie de
24     Fibonacci."""
25     if n <= 0:
26         return []
27     elif n == 1:
28         return [0]
29     fibonacci = [0, 1]
30     for i in range(2, n):
31         fibonacci.append(fibonacci[-1] + fibonacci[-2])
32     return fibonacci
33
34 def main():
35     numero = int(input("Introduce un número: "))
36
37     # Comprobar si el número es parte de la serie de
38     # Fibonacci
39     if es_fibonacci(numero):
40         print(f"El número {numero} es parte de la serie
41         de Fibonacci.")
42     else:
43         print(f"El número {numero} no es parte de la
44         serie de Fibonacci.")
45     # Comprobar si el número es primo
46     if es-primo(numero):
47         print(f"El número {numero} es primo.")
48     else:
49         print(f"El número {numero} no es primo.")
50
51     # Mostrar los primeros n términos de la serie de
52     # Fibonacci
53     n = int(input("Introduce cuántos términos de la
54     serie Fibonacci deseas ver: "))
55     print(f"Serie de Fibonacci: {serie_fibonacci(n)}")
56
57 if __name__ == "__main__":
58     main()

```

Código 6.8: Solución al Ejercicio 6.7 (continuación).

EJERCICIO 6.8

Escribe un programa en Python que calcule el área de diferentes formas geométricas (círculo, cuadrado y rectángulo), mostrándola con dos decimales. El cálculo de cada área debe estar en un módulo independiente llamado «formas_geometricas.py». Luego, en el archivo principal, importa este módulo y usa las funciones para calcular el área de cada forma geométrica según la opción que elija el usuario.

Ejemplo de ejecución esperada

```

Selecciona la forma geométrica:
1. Círculo
2. Cuadrado
3. Rectángulo
Introduce tu opción: 1
Introduce el radio del círculo: 5
El área del círculo es 78.54

```

Solución (girar para leer)

Código 6.9: Solución al Ejercicio 6.8 (archivo «formas_geometricas.py»)

```

1 import math
2
3 def area_circulo(radio: float) -> float:
4     """Calcula el área de un círculo dado su radio."""
5     return math.pi * radio ** 2
6
7 def area_cuadrado(lado: float) -> float:
8     """Calcula el área de un cuadrado dado su lado."""
9     return lado ** 2
10
11 def area_rectangulo(base: float, altura: float) -> float:
12     """Calcula el área de un rectángulo dado su base y
13     altura."""
14     return base * altura

```

Solución (girar para leer)

```

1  import formas_geometricas
2
3  def main():
4      print("Selecciona la forma geométrica:")
5      print("1. Círculo")
6      print("2. Cuadrado")
7      print("3. Rectángulo")
8
9      opcion = int(input("Introduce tu opción: "))
10
11     if opcion == 1:
12         radio = float(input("Introduce el radio del círculo: "))
13         area = formas_geometricas.area_circulo(radio)
14         print(f"El área del círculo es {area:.2f}")
15
16     elif opcion == 2:
17         lado = float(input("Introduce el lado del cuadrado: "))
18         area = formas_geometricas.area_cuadrado(lado)
19         print(f"El área del cuadrado es {area:.2f}")
20
21     elif opcion == 3:
22         base = float(input("Introduce la base del rectángulo: "))
23         altura = float(input("Introduce la altura del rectángulo: "))
24         area = formas_geometricas.area_rectangulo(base, altura)
25         print(f"El área del rectángulo es {area:.2f}")
26
27     else:
28         print("Opción no válida.")
29
30     if __name__ == "__main__":
31         main()

```

Código 6.10: Solución al Ejercicio 6.8 (fichero principal).

EJERCICIO 6.9

Escribe un programa en Python que convierta temperaturas entre Celsius, Fahrenheit y Kelvin. El cálculo de la conversión debe realizarse en un módulo independiente llamado «`conversion.temperaturas.py`». Luego, en el archivo principal, importa este módulo y usa las funciones para convertir una temperatura proporcionada por el usuario entre las tres unidades. Todas las temperaturas calculadas se mostrarán con dos decimales.

Ejemplo de ejecución esperada

```
Introduce la temperatura en grados Celsius: 25
La temperatura en Fahrenheit es 77.00
La temperatura en Kelvin es 298.15
```

Solución (girar para leer)

```

1  def celsius_a_fahrenheit(celsius: float) -> float:
2      """Convierte temperatura de grados Celsius a
3      Fahrenheit."""
4      return (celsius * 9/5) + 32
5
6  def celsius_a_kelvin(celsius: float) -> float:
7      """Convierte temperatura de grados Celsius a Kelvin
8      ."""
9      return celsius + 273.15
10
11 def fahrenheit_a_celsius(fahrenheit: float) -> float:
12     """Convierte temperatura de grados Fahrenheit a
13     Celsius."""
14     return (fahrenheit - 32) * 5/9
15
16 def fahrenheit_a_kelvin(fahrenheit: float) -> float:
17     """Convierte temperatura de grados Fahrenheit a
18     Kelvin."""
19     return kelvin - 273.15
20
21 def kelvin_a_fahrenheit(kelvin: float) -> float:
22     """Convierte temperatura de Kelvin a Fahrenheit."""
23     return (kelvin - 273.15) * 9/5 + 32

```

Código 6.11: Solución al Ejercicio 6.9 (archivo «conversion-temperaturas.py»).

Solución (girar para leer)

```

1  import conversion_temperaturas
2
3  def main():
4      celsius = float(input("Introduce la temperatura en
5      grados Celsius: "))
6      fahrenheit = conversion_temperaturas.
7      celsius_a_fahrenheit(celsius)
8      kelvin = conversion_temperaturas.celsius_a_kelvin(
9      celsius)
10     print(f"La temperatura en Fahrenheit es {fahrenheit
11     :.2f}")
12     print(f"La temperatura en Kelvin es {kelvin:.2f}")
13
14 if __name__ == "__main__":
15     main()

```

Código 6.12: Solución al Ejercicio 6.9 (archivo principal).

EJERCICIO 6.10

Escribe un programa en Python que convierta entre diferentes unidades de longitud, como metros, kilómetros, millas y yardas. El programa debe estar estructurado en dos partes: (1) un módulo independiente llamado «`conversion_unidades.py`» que contenga funciones para convertir entre las unidades; y (2) el archivo principal, que debe importar este módulo y usar las funciones para convertir entre las unidades de longitud, solicitando al usuario que elija la conversión deseada.

El programa debe permitir la conversión entre las siguientes unidades:

- Metros a Kilómetros
- Kilómetros a Metros
- Millas a Kilómetros
- Kilómetros a Millas
- Yardas a Metros
- Metros a Yardas

Ejemplo de ejecución esperada

Selecciona la conversión que deseas realizar:

1. Metros a Kilómetros
2. Kilómetros a Metros
3. Millas a Kilómetros
4. Kilómetros a Millas
5. Yardas a Metros
6. Metros a Yardas

Introduce tu opción: 1

Introduce el valor en metros: 500

500.0 metros son 0.5 kilómetros

Solución (girar para leer)

Código 6.13: Solución al Ejercicio 6.10 (archivo «conversion-unidades.py»).

```

1  def metros_a_kilometros(metros: float) -> float:
2      """Convierte metros a kilómetros."""
3      return metros / 1000
4
5  def kilometros_a_metros(kilometros: float) -> float:
6      """Convierte kilómetros a metros."""
7      return kilometros * 1000
8
9  def millas_a_kilometros(millas: float) -> float:
10     """Convierte millas a kilómetros."""
11     return millas * 1.60934
12
13 def kilometros_a_millas(kilometros: float) -> float:
14     """Convierte kilómetros a millas."""
15     return kilometros / 1.60934
16
17 def yardas_a_metros(yardas: float) -> float:
18     """Convierte yardas a metros."""
19     return yardas * 0.9144
20
21 def metros_a_yardas(metros: float) -> float:
22     """Convierte metros a yardas."""
23     return metros / 0.9144

```

Solución (girar para leer)

```
def main():
    print("Selecciona la conversión que desees realizar")
    print("1. Metros a Kilómetros")
    print("2. Kilómetros a Metros")
    print("3. Millas a Kilómetros")
    print("4. Kilómetros a Millas")
    print("5. Yardas a Metros")
    print("6. Metros a Yardas")
    option = int(input("Introduce tu opción: "))

    if option == 1:
        metros = float(input("Introduce el valor en metros: "))
        resultado = conversion_unidades.metros_a_kilometros(metros)
        print(f"{metros} metros son {resultado} kilo metros")

    elif option == 2:
        kilometros = float(input("Introduce el valor en kilometros: "))
        resultado = conversion_unidades.kilometros_a_metros(kilometros)
        print(f"{kilometros} kilometros son {resultado} metros")

    elif option == 3:
        millas = float(input("Introduce el valor en millas: "))
        resultado = conversion_unidades.millas_a_kilometros(millas)
        print(f"{millas} millas son {resultado} kilo metros")

    elif option == 4:
        kilometros = float(input("Introduce el valor en kilometros: "))
        resultado = conversion_unidades.kilometros_a_millas(kilometros)
        print(f"{kilometros} kilometros son {resultado} millas")

    elif option == 5:
        yardas = float(input("Introduce el valor en yardas: "))
        resultado = conversion_unidades.yardas_a_metros(yardas)
        print(f"{yardas} yardas son {resultado} metros")

    elif option == 6:
        metros = float(input("Introduce el valor en metros: "))
        resultado = conversion_unidades.metros_a_yardas(metros)
        print(f"{metros} metros son {resultado} yardas")

    else:
        print("Opción no válida")
```

Solución (girar para leer)

```

48 main()
47 if __name__ == "__main__":
46
45     print("Opción no válida.")
44 else:
43
42     print(f"{metros} metros son {resultado} yardas")
41     (metros)
40     resultado = conversion_unidades.metros-a-yardas
39     metros: ""
38     if opcion == 6:
37         print(f"{yardas} yardas son {resultado} metros")
36         (yardas)
35         resultado = conversion_unidades.yardas-a-metros
34         yardas: ""
33         if opcion == 5:
32             yardas = float(input("Introduce el valor en
31             yardas: "))
30             resultado = conversion_unidades.yardas-a-metros
29             (yardas)
28             print(f"{yardas} yardas son {resultado} metros")
27         )
26         if opcion == 5:
25             yardas = float(input("Introduce el valor en
24             yardas: "))
23             resultado = conversion_unidades.yardas-a-metros
22             (yardas)
21             print(f"{yardas} yardas son {resultado} metros")
20         )
19         if opcion == 6:
18             metros = float(input("Introduce el valor en
17             metros: ""
16             resultado = conversion_unidades.metros-a-yardas
15             (metros)
14             print(f"{metros} metros son {resultado} yardas")
13         )
12     )
11
10
9
8
7
6
5
4
3
2
1

```