

2

Introducción a la programación y resolución automática de problemas con Python

En este capítulo exploraremos *cómo los programas pueden utilizarse para resolver problemas relacionados con el procesamiento de información cuando son ejecutados en un computador*. Con el entorno de trabajo ya configurado en el Capítulo 1, el siguiente paso es entender qué es un algoritmo, qué es un programa, y cómo se relacionan ambos conceptos. Veremos también los primeros elementos de la sintaxis de Python para que el lector pueda escribir y ejecutar sus primeros programas.

2.1. Programación y resolución automática de problemas

La programación es una disciplina que permite automatizar la resolución de problemas en ámbitos muy diversos: cálculo numérico, gestión de datos, procesamiento de imágenes, reconocimiento y traducción de lenguaje natural, control de sistemas industriales, robótica, conducción autónoma, análisis de señales biomédicas, ciberseguridad, entre muchos otros.

Los computadores (término que se usará en este libro para referirse a los equipos electrónicos programables, sean ordenadores personales, servidores o sistemas embebidos) están diseñados para procesar información de forma automática y eficiente. Su funcionamiento depende de programas que, al ejecutarse, deben proporcionar soluciones a los problemas planteados.

La pregunta central que aborda este libro es: *¿cómo escribir un programa para resolver un problema de procesamiento de información?*

Para resolver un problema, el primer paso es **identificar uno o varios métodos que puedan utilizarse para encontrar una solución**. Luego, es necesario **seleccionar el más adecuado** considerando aspectos como su eficiencia y su simplicidad. Posteriormente, **el método elegido debe formalizarse por escrito**, describiendo la información relevante y la forma en que debe ser procesada para obtener un resultado. Esa formalización se conoce como *algoritmo*.

Algoritmo: secuencia ordenada y finita de pasos, definida sin ambigüedades, que permite encontrar la solución a un problema de procesamiento de información.

Los algoritmos pueden expresarse en distintos lenguajes: lenguajes de programación como Python, C, C++, Java, JavaScript, Rust o Go, o lenguas naturales como español, inglés o francés. Los lenguajes de programación tienen la ventaja de ser comprensibles directamente por los computadores, algo que aún no es posible con los lenguajes naturales sin una capa de interpretación adicional.

Cuando un algoritmo se implementa en un lenguaje de programación, se convierte en un *programa*. Existen cientos de lenguajes de programación, surgen nuevos constantemente, y otros caen en desuso con el tiempo.

En este libro utilizaremos Python como lenguaje de referencia para expresar algoritmos. Sin embargo, el enfoque principal no será el dominio exclusivo de su sintaxis, sino la comprensión de los principios y técnicas de programación, y la aplicación de buenas prácticas de diseño de software. Al finalizar este libro, el lector dispondrá de los conocimientos y habilidades necesarios para analizar y resolver problemas de procesamiento de información, y será capaz de elegir el lenguaje más adecuado a cada contexto.

Un mini-ejemplo: del problema al algoritmo. Supongamos el siguiente problema: dada una lista de notas de un examen, calcular la nota media. Sin haber escrito todavía una sola línea de Python, podemos plantear el algoritmo en lenguaje natural:

1. Empezar con un acumulador con valor cero y un contador con valor cero.
2. Para cada nota de la lista:
 - a) sumar la nota al acumulador;
 - b) incrementar el contador en uno.
3. Si el contador es cero, no hay notas y la media no está definida.
4. En caso contrario, la media es el acumulador dividido entre el contador.

Esta secuencia ordenada y sin ambigüedades es un algoritmo. La traducción a Python (es decir, su conversión en programa) la veremos a lo largo de los próximos capítulos.

2.2. Propiedades de un algoritmo

Un algoritmo debe cumplir ciertas propiedades para garantizar su correcto funcionamiento y facilitar su comprensión y reutilización. Estas propiedades pueden agruparse en *imprescindibles* y *deseables*.

Las propiedades *imprescindibles* son:

- **Corrección:** un algoritmo debe proporcionar una solución correcta al problema planteado. Sus resultados deben ajustarse a la especificación dada y resolver el problema de manera adecuada.
- **Legibilidad:** la descripción de un algoritmo debe ser clara y fácil de entender, no solo para quien lo diseñó, sino también para cualquier persona con conocimientos de programación. Esto facilita su revisión, mantenimiento y posible modificación.

Las propiedades *deseables*, aunque no estrictamente necesarias, mejoran la calidad y utilidad de un algoritmo:

- **Generalidad:** un algoritmo no debe restringirse a resolver un problema demasiado específico si, con un esfuerzo adicional asumible, puede aplicarse a una gama más amplia de problemas.
- **Reusabilidad:** un buen algoritmo debe poder reutilizarse en otros programas sin modificaciones o con ajustes mínimos. Esto mejora la eficiencia en el desarrollo y evita la redundancia de código.
- **Eficiencia:** un buen algoritmo debe utilizar los recursos de manera óptima, considerando tanto el tiempo de ejecución como el consumo de memoria. La eficiencia es crucial para resolver problemas de manera rápida y sin sobrecargar el sistema.
- **Independencia del lenguaje y del sistema:** un algoritmo debe ser lo suficientemente abstracto como para implementarse en distintos lenguajes de programación y ejecutarse en distintos entornos sin dificultades.

2.3. Estructura de un programa en Python

Python es un lenguaje de programación ampliamente utilizado por su sencillez y legibilidad. A continuación se presenta un programa muy simple que imprime un mensaje en pantalla:

```
1 | # Pre: ---
2 | # Post: Escribe en pantalla el mensaje "Hola, mundo desde
↪ | Python"
3 |
4 | print("Hola, mundo desde Python")
```

2. Introducción a la programación y resolución automática de problemas con Python

A pesar de su brevedad, este programa ya muestra los elementos básicos de un programa Python:

- **Comentarios:** las dos primeras líneas comienzan por el símbolo `#`, lo que las marca como comentarios. Python ignora todo lo que aparece tras un `#` hasta el final de la línea. En este caso, los comentarios documentan la *precondición* y la *postcondición* del programa: qué se asume al arrancar (**Pre:** `---`, es decir, ninguna), y qué garantiza al terminar (**Post:** `...`). Documentar pre y postcondiciones es una práctica recomendada al diseñar algoritmos.
- **Instrucción:** la última línea contiene una llamada a la función `print`, que muestra el mensaje recibido como argumento entre paréntesis.

En Jupyter Notebook, al ejecutar la celda, la salida aparece justo debajo de la celda de código, sin necesidad de invocaciones adicionales.

2.4. Desarrollo y ejecución de un programa en Python

El desarrollo de un programa en Python consta de tres pasos principales: (1) escribir el código en un archivo o celda; (2) ejecutarlo; e (3) interpretar los resultados.

La gestión de estos pasos depende del entorno en el que se trabaje. En un entorno interactivo como Jupyter Notebook, la ejecución se realiza directamente desde una celda de código. En un editor de texto o en un IDE, el flujo habitual es escribir el código en un archivo con extensión `.py` y ejecutarlo desde una terminal con:

```
python nombre_del_archivo.py
```

En un entorno interactivo, basta con ejecutar la celda donde se encuentra el código. Por ejemplo, ejecutar la celda:

```
1 | print("Hola, mundo desde Python")
```

mostrará en pantalla el mensaje `Hola, mundo desde Python`.

El desarrollo y la ejecución también pueden gestionarse desde un IDE. Un IDE es un software que facilita la edición de código, la ejecución y, cuando aparecen errores, su depuración. Existen numerosos IDEs para Python; algunos populares son PyCharm, Visual Studio Code, Spyder y Thonny. En este libro recomendamos VS Code (descrito en el Capítulo 1) por su ligereza y por su buena integración con Jupyter.

Es importante que cualquier código esté debidamente documentado mediante comentarios. Los comentarios ayudan a explicar la funcionalidad del programa, el propósito de las variables y las acciones realizadas. En Python, los comentarios se introducen con el símbolo `#` y duran hasta el final de la línea. Una buena documentación facilita el mantenimiento del código y permite que otros programadores lo entiendan y modifiquen en el futuro. Como regla general, los comentarios deben ser claros y concisos, evitando explicaciones innecesarias o redundantes.

2.5. Guía de estilo en la programación

Al escribir algoritmos y programas conviene seguir una serie de reglas de estilo cuyo propósito es mejorar la calidad del código y su legibilidad. Mantener un estilo coherente facilita el trabajo del propio programador y la colaboración con otros desarrolladores.

En Python, la guía de estilo de referencia es **PEP 8**¹, escrita por Guido van Rossum (creador del lenguaje) y mantenida por la comunidad. Algunas reglas básicas de PEP 8 que conviene conocer desde el principio son:

- **Indentación:** usar 4 espacios por nivel de indentación. No mezclar tabuladores y espacios.
- **Longitud de línea:** limitar las líneas a 79 caracteres como máximo (en docstrings y comentarios, a 72).
- **Nombres de variables y funciones:** en `snake_case` (todo en minúsculas con guiones bajos para separar palabras), por ejemplo, `edad_alumno` o `calcular_media`.
- **Nombres de constantes:** en mayúsculas con guiones bajos, por ejemplo, `PI = 3.141592`.
- **Espacios alrededor de operadores:** `a = b + c` y no `a=b+c`.
- **Líneas en blanco:** separar funciones y clases con dos líneas en blanco; bloques lógicos dentro de funciones, con una línea en blanco.

Los ejemplos de este libro siguen estas reglas, y se anima al lector a adoptarlas desde el primer programa. Existen herramientas que comprueban automáticamente el cumplimiento de PEP 8 (por ejemplo, `flake8` o `pycodestyle`); otras como `black` reformatean el código directamente.

2.6. Sintaxis básica de Python: primeros pasos

La sintaxis de Python es sencilla y fácil de aprender, lo que lo hace un lenguaje excelente para principiantes. Vamos a comenzar con los conceptos básicos. Todos los ejemplos pueden probarse desde una celda de Jupyter Notebook o desde una terminal de Python.

En Python, las *sentencias* son instrucciones que indican al computador qué hacer. Cada línea de código es, en general, una sentencia. Python utiliza la *indentación* para estructurar los bloques de código y organizar el flujo del programa. A diferencia de otros lenguajes (C, C++, Java, JavaScript), Python *no* usa llaves { } para delimitar bloques: la indentación es semánticamente significativa.

¹PEP, del inglés *Python Enhancement Proposal*, número 8: <https://peps.python.org/pep-0008/>.

```
1 | print("Hola, mundo desde Python")
```

El ejemplo anterior es una sentencia que imprime el mensaje "Hola, mundo desde Python" en la consola. Veamos cómo se usa la indentación para definir un bloque:

```
1 | if 5 > 3:
2 |     print("5 es mayor que 3")
```

En este caso, la sentencia `print` está dentro del bloque del `if` porque está indentada con 4 espacios. Sin la indentación, Python no sabría que la llamada forma parte del cuerpo de la condición. Las sentencias condicionales se estudian en detalle en capítulos posteriores.

2.6.1. Comentarios y docstrings

Los comentarios sirven para añadir notas explicativas dentro del código. Python los ignora durante la ejecución. En Python, los comentarios se introducen con el símbolo `#` y se extienden hasta el final de la línea:

```
1 | # Esta línea es un comentario
2 | print("Hola, mundo") # Otro comentario al final de una línea
```

Para comentar varias líneas, lo idiomático en Python es prefijar cada una con `#`:

```
1 | # Este programa imprime un saludo
2 | # y termina su ejecución
3 | print("Hola, mundo")
```

A veces verás cadenas de varias líneas delimitadas por comillas triples (`""" ... """` o `''' ... '''`) usadas como si fueran comentarios:

```
1 | """
2 | Esto NO es un comentario en sentido estricto:
3 | es una cadena literal que el intérprete descarta.
4 | """
5 | print("Hola, mundo")
```

En realidad, lo que aparece entre comillas triples es un *literal de cadena de varias líneas*: si no se asigna a ninguna variable ni se imprime, simplemente se evalúa y se descarta. Su uso idiomático no es como comentario, sino como **docstring** (cadena de documentación) en la primera línea de un módulo, función o clase, donde sí tiene un significado especial: las herramientas de Python (como `help()`) la leen para describir el objeto. Los *docstrings* se estudiarán con mayor detalle al introducir funciones y módulos.

2.6.2. Variables

En Python, una *variable* es un nombre que apunta a un valor almacenado en memoria. Ese valor puede cambiar a lo largo de la ejecución, según el programa lo manipule. Las variables son, por tanto, contenedores que almacenan datos que pueden ser utilizados y modificados durante la ejecución.

A diferencia de lenguajes como C o Java, en Python no es necesario declarar el tipo de una variable: el tipo lo determina el valor que se le asigna. Python es un *lenguaje dinámicamente tipado*, lo que significa que el tipo de una variable puede cambiar a lo largo de la ejecución.

Para usar una variable basta con asignarle un valor mediante el operador `=`. El nombre de la variable, llamado *identificador*, debe cumplir algunas reglas:

- Debe comenzar por una letra o un guion bajo (`_`).
- Puede continuar con letras, dígitos o guiones bajos.
- Es sensible a mayúsculas y minúsculas: `edad` y `Edad` son variables distintas.
- No puede coincidir con una *palabra reservada* del lenguaje (por ejemplo, `if`, `for`, `class`, `return`).

Veamos un ejemplo. Tras la sentencia:

```
1 | edad = 25
```

Python infiere automáticamente, a partir del valor asignado, que la variable `edad` contiene un entero. Si más adelante se asigna a la misma variable un valor de otro tipo:

```
1 | edad = "Veinticinco"
```

la variable pasa a contener una cadena de texto. Esta flexibilidad facilita el desarrollo rápido pero también **puede ser fuente de errores si no se tiene cuidado con los tipos de los datos que se manipulan**.

2.6.3. Tipos de datos básicos (introducción)

Los tipos de datos más básicos en Python son los enteros, los flotantes, las cadenas y los booleanos. Estos tipos son esenciales para realizar cálculos y manipular información. A continuación se presentan brevemente, junto con un ejemplo de literal y de asignación. Un *literal* es la representación directa de un valor en el código fuente, sin tener que calcularlo ni invocar funciones.

Enteros (`int`)

Los enteros son números sin decimales. Pueden ser positivos o negativos y, en Python, no tienen un tamaño fijo: pueden crecer hasta el límite que permita la memoria disponible.

```
1 | numero_entero = 42
```

En el ejemplo, se define una variable `numero_entero` y se le asigna el literal entero `42`.

Flotantes (float)

Los flotantes son números con parte decimal. Python utiliza una precisión doble de punto flotante para representarlos, lo que le permite manejar una amplia gama de valores reales.

```
1 | numero_flotante = 3.141592
```

Cadenas (str)

Las cadenas son secuencias de caracteres, es decir, textos. Se pueden delimitar con comillas simples (' ') o con comillas dobles (" "), pero no se pueden mezclar (las comillas que abren la cadena deben coincidir con las que la cierran). Ambas formas son equivalentes en Python.

```
1 | mensaje = "Hola, ¿cómo estás?"
```

Booleanos (bool)

Los valores booleanos sólo pueden tomar dos valores: **True** (verdadero) o **False** (falso). Son muy útiles en el control de flujo del programa, especialmente en condiciones y bucles, que se estudiarán en capítulos posteriores.

```
1 | es_mayor = True
```

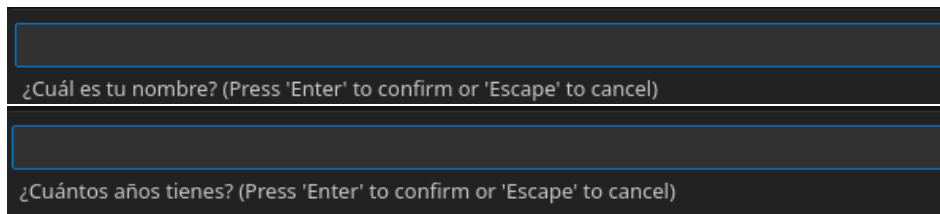
2.6.4. Entrada y salida de datos

Una de las primeras necesidades de un programa es interactuar con el usuario. En Python, la función `input()` se usa para leer información de la consola. Esta función **siempre devuelve una cadena de texto**, así que si lo que se necesita es un número, hay que realizar una *conversión explícita* usando funciones como `int()` (para enteros) o `float()` (para reales).

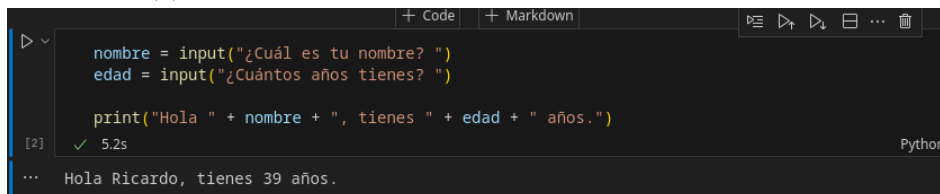
A continuación se muestra un ejemplo de lectura de datos del usuario y muestra de un saludo:

```
1 | nombre = input("¿Cuál es tu nombre? ")
2 | edad = input("¿Cuántos años tienes? ")
3 |
4 | print("Hola " + nombre + ", tienes " + edad + " años.")
```

La función `print()` muestra información al usuario. Conviene destacar que en el ejemplo la variable `edad` no es un entero, sino una cadena, ya que los datos leídos con `input()` siempre son cadenas. Si se necesitara como entero, habría que convertirlo explícitamente con `edad = int(edad)`. La diferencia entre la *conversión implícita* (la que Python hace automáticamente, como al sumar un `int` con un `float`) y la *conversión explícita* (la que el programador realiza llamando a `int()`, `float()`, etc.) se estudiará en detalle en el siguiente capítulo. Para concatenar cadenas y valores de variables se puede



(a) Ventanas emergentes para petición de nombre y edad



(b) Ejecución de la celda tras la introducción de datos

Figura 1: Interacción con el usuario desde Jupyter Notebook en VS Code.

usar el operador `+`, o emplear *f-strings* (cadenas formateadas), que se verán en detalle también en el siguiente capítulo.

Si se trabaja con VS Code y Jupyter Notebook, al ejecutar una celda con interacción, los datos se solicitan mediante una ventana flotante en la parte central superior de la ventana (Figura 1).

2.6.5. Operadores aritméticos

Python soporta un conjunto básico de operadores aritméticos para realizar operaciones matemáticas simples. Estos operadores son esenciales en cualquier programa que realice cálculos. A continuación se describe cada uno con un breve ejemplo.

Suma (+)

Suma dos valores.

```
1 resultado = 5 + 3
2 print(resultado) # Imprime 8
```

Resta (-)

Resta el segundo valor del primero.

```
1 resultado = 5 - 3
2 print(resultado) # Imprime 2
```

Multiplicación (*)

Multiplica dos valores.

```
1 resultado = 5 * 3
2 print(resultado) # Imprime 15
```

División (/)

Realiza la división de dos valores. Es importante recordar que **en Python 3, la división con / (división verdadera) siempre devuelve un valor de tipo float, incluso si los dos operandos son enteros**. Esto difiere del comportamiento de otros lenguajes de programación.

```
1 resultado = 5 / 2
2 print(resultado) # Imprime 2.5; resultado es de tipo float
```

División entera (//)

Realiza la división eliminando la parte decimal del cociente. El tipo del resultado depende de los operandos: si ambos son enteros, el resultado es entero; si alguno de los operandos es flotante, el resultado es flotante.

```
1 print(5 // 2) # Imprime 2 (int)
2 print(5.0 // 2) # Imprime 2.0 (float)
```

Módulo (%)

Calcula el resto de una división. Es muy útil para verificar, por ejemplo, si un número es par o impar.

```
1 resultado = 5 % 2
2 print(resultado) # Imprime 1 (resto de la división)
```

Exponenciación (**)

Eleva un número a una potencia.

```
1 resultado = 5 ** 2
2 print(resultado) # Imprime 25
```

Estos operadores permiten escribir programas que realicen cálculos numéricos básicos. En programación más avanzada, se combinan para resolver problemas matemáticos más complejos, desde la física hasta las ciencias computacionales y la ciencia de datos.

2.7. Multiparadigma en Python

Python es un lenguaje **multiparadigma**: permite programar bajo distintos enfoques. En estos primeros capítulos trabajaremos bajo el **paradigma imperativo**, en el que se define una secuencia de instrucciones que el intérprete ejecuta de arriba a abajo, modificando el estado de las variables.

A medida que avance el aprendizaje, se introducirán otros paradigmas. Python soporta también la **programación orientada a objetos**, que organiza el código en clases y objetos, favoreciendo la modularidad y la reutilización; y la **programación funcional**, que trata las funciones como valores de primera clase y permite aplicar técnicas como las funciones de orden superior y construcciones como `map`, `filter` o `reduce`. Posteriormente, se trabajará también bajo los enfoques de **programación estructurada** y **programación modular**, que se explican en capítulos posteriores.

2.8. Ejercicios

EJERCICIO 2.1

Escribe un programa en Python que solicite al usuario su nombre y su ciudad de origen, y luego lo salude de manera personalizada mencionando ambos datos.

Ejemplo de ejecución esperada:

```
¿Cuál es tu nombre? Alberto
¿De qué ciudad eres? Teruel
Hola, Alberto de Teruel.
¡Bienvenido a la programación con Python!
```

Modifica ahora el programa para que también pregunte por el país de origen y lo incluya en el mensaje de bienvenida.

Ejemplo de ejecución esperada:

```
¿Cuál es tu nombre? Ana
¿De qué ciudad eres? Buenos Aires
¿De qué país eres? Argentina
Hola, Ana de Buenos Aires, Argentina.
¡Bienvenido a la programación con Python!
```

Solución (girar para leer)

```
1 # Pre: El usuario debe ingresar su nombre y su ciudad
2 # Post: Se muestra un mensaje personalizado de
3 #       bienvenida con la ciudad de origen.
4 # Solicita el nombre y la ciudad del usuario
5 nombre = input("¿Cuál es tu nombre? ")
6 ciudad = input("¿De qué ciudad eres? ")
7 # Solicita el país de origen
8 país = input("¿De qué país eres? ")
9
10 # Muestra el mensaje de bienvenida
11 print("Hola, " + nombre + " de " + ciudad + ", " + país
12 print("!Bienvenido a la programación con Python!")
```

Código 2.1: Solución al Ejercicio 2.1.

EJERCICIO 2.2

Escribe un programa en Python que solicite el nombre y la edad del usuario, y luego lo salude indicando cuántos años tendrá dentro de 5 años.

Ejemplo de ejecución esperada:

```
¿Cuál es tu nombre? Sofía
¿Cuántos años tienes? 30
Hola, Sofía. Dentro de 5 años tendrás 35 años.
```

Modifica ahora el programa para que pregunte cuántos años en el futuro quiere calcular el usuario y muestre el resultado basado en su elección.

Ejemplo de ejecución esperada:

```
¿Cuál es tu nombre? Marcos
¿Cuántos años tienes? 25
¿Dentro de cuántos años quieres saber tu edad? 10
Hola, Marcos. Dentro de 10 años tendrás 35 años.
```

Solución (girar para leer)

```

1 # Pre: El usuario debe ingresar su nombre y edad.
2 # Post: Se muestra un mensaje con su edad dentro de 5 a
3     ños.
4 # Solicita el nombre y la edad del usuario
5 nombre = input("¿Cuál es tu nombre? ")
6 edad = int(input("¿Cuántos años tienes? "))
7
8 # Solicita los años que el usuario quiere calcular en
9     el futuro
10 años_futuro = int(input("¿Dentro de cuántos años
11     quieres saber tu edad? "))
12 # Calcula la edad en el futuro especificado
13 edad_futura = edad + años_futuro
14 # Muestra el mensaje con la edad futura
15 print("Hola, " + nombre + " . Dentro de " + str(
16     edad_futura) + " años tendrás " + str(edad_futura)
17     + " años.")

```

Código 2.2: Solución al Ejercicio 2.2.

EJERCICIO 2.3

Escribe un programa en Python que solicite al usuario una temperatura en grados Celsius y la convierta a grados Fahrenheit, usando la fórmula:

$$F = \left(C \cdot \frac{9}{5} \right) + 32$$

Muestra el resultado en pantalla.

Ejemplo de ejecución esperada:

```
Introduce la temperatura en grados Celsius: 25
25 grados Celsius son 77.0 grados Fahrenheit.
```

Modifica ahora el programa para que también convierta la temperatura a Kelvin utilizando la fórmula:

$$K = C + 273.15$$

Ejemplo de ejecución esperada:

```
Introduce la temperatura en grados Celsius: 30
30 grados Celsius son 86.0 grados Fahrenheit y 303.15 grados Kelvin.
```

Solución (girar para leer)

```

1 # Pre: El usuario debe ingresar una temperatura en
  grados Celsius.
2 # Post: Se muestra la conversión a grados Fahrenheit.
3
4 # Solicita la temperatura en Celsius
5 Celsius = float(input("Introduce la temperatura en
  grados Celsius: "))
6
7 # Convierte a Fahrenheit
8 Fahrenheit = (Celsius * 9/5) + 32
9 # Convierte a Kelvin
10 Kelvin = Celsius + 273.15
11
12 # Muestra los resultados en ambas unidades
13 print(f"{int(Celsius)} grados Celsius son {Fahrenheit}
  grados Fahrenheit y {Kelvin} grados Kelvin.")

```

Código 2.3: Solución al Ejercicio 2.3.

EJERCICIO 2.4

Escribe un programa en Python que solicite al usuario la base y la altura de un rectángulo y calcule su área usando la fórmula:

$$A = b \cdot h$$

donde b es la base y h es la altura. Muestra el resultado en pantalla.

Ejemplo de ejecución esperada:

```
Introduce la base del rectángulo: 10
Introduce la altura del rectángulo: 5
El área del rectángulo es 50.
```

Modifica ahora el programa para que también calcule el perímetro del rectángulo usando la fórmula:

$$P = 2 \cdot (b + h)$$

Ejemplo de ejecución esperada:

```
Introduce la base del rectángulo: 8
Introduce la altura del rectángulo: 4
El área del rectángulo es 32 y el perímetro es 24.
```

Solución (girar para leer)

```

1 # Pre: El usuario debe ingresar la base y la altura de
  ↩ un rectángulo.
2 # Post: Se muestra el área calculada.
3
4 # Solicita la base y la altura del rectángulo
5 base = float(input("Introduce la base del rectángulo: "))
  ↩
6 altura = float(input("Introduce la altura del rectá
  ↩ ngulo: "))
7
8 # Calcula el área
9 area = base * altura
10
11 # Calcula el perímetro
12 perimetro = 2 * (base + altura)
13
14 # Muestra el resultado con el perímetro incluido
15 print(f"El área del rectángulo es {int(area)} y el perí
  ↩ metro es {int(perimetro)}.")

```

Código 2.4: Solución al Ejercicio 2.4.

EJERCICIO 2.5

Escribe un programa en Python que calcule el área de un círculo. El programa debe solicitar al usuario el radio del círculo e imprimir el resultado del área utilizando la fórmula:

$$A = \pi \cdot r^2$$

donde A es el área y r es el radio del círculo. Usa el valor de π como 3.141592.

Ejemplo de ejecución esperada:

```
Introduce el radio del círculo: 5
El área del círculo es 78.5398
```

Modifica ahora el programa para que también calcule el perímetro del círculo utilizando la fórmula:

$$P = 2 \cdot \pi \cdot r$$

Ejemplo de ejecución esperada:

```
Introduce el radio del círculo: 5
El área del círculo es 78.5398 y el perímetro es 31.41592
```

Solución (girar para leer)

```

1 # Pre: El usuario debe ingresar el radio del círculo.
2 # Post: Se muestra el área del círculo calculada con la
3     fórmula  $A = \pi r^2$ .
4 # Solicita el radio del círculo
5 radio = float(input("Introduce el radio del círculo: "))
6
7 # Define el valor de pi
8 pi = 3.141592
9
10 # Calcula el área del círculo
11 area = pi * radio ** 2
12
13 # Calcula el perímetro del círculo
14 perimetro = 2 * pi * radio
15
16 # Muestra el resultado con el área y el perímetro
17 print("El área del círculo es " + str(area) + " y el
    perímetro es " + str(perimetro))

```

Código 2.5: Solución al Ejercicio 2.5.