

8

Flotantes (números reales)

En este capítulo exploraremos cómo Python maneja los números reales, centrándonos en su representación, las operaciones básicas y las limitaciones inherentes al uso de la *aritmética en coma flotante*. Veremos cómo Python utiliza el tipo `float` y cómo se gestionan casos especiales como el desbordamiento (`inf`) y los valores no definidos (`nan`). También analizaremos las herramientas que Python ofrece para realizar cálculos con números reales de forma fiable, como la biblioteca `math` y los módulos `decimal` y `fractions`. La aritmética en coma flotante es relevante en ciencia de datos y en muchos otros dominios, como la ciberseguridad: numerosos algoritmos numéricos y modelos de aprendizaje automático toman decisiones basadas en valores reales, y entender cómo (y cuándo) fallan estos cálculos resulta esencial.

8.1. Representación de números reales en Python

En Python, los números reales se representan con el tipo `float`, un número de **coma flotante de doble precisión** (64 bits). Sigue el estándar IEEE 754, que regula cómo se almacenan y manipulan los números de coma flotante en la mayoría de plataformas.

Internamente, los 64 bits se dividen en tres partes:

- El **bit de signo** (1 bit), que determina si el número es positivo o negativo.
- El **exponente** (11 bits), que define la potencia de 2 por la que se multiplica la parte significativa.
- La **mantisa** (52 bits), que representa la parte fraccionaria del número.

El rango de valores que puede representar el tipo `float` es amplio, aproximadamente $[-1,7 \cdot 10^{308}, 1,7 \cdot 10^{308}]$, con una precisión relativa de unos 15 a 16 dígitos decimales significativos. Para que te hagas una idea de la magnitud, este rango es suficiente para enumerar todas y cada una de las estrellas del Universo visible.

8. Flotantes (números reales)

Sin embargo, esta representación binaria **no permite expresar de forma exacta todos los números decimales**. Por ejemplo, 0,1 no tiene representación binaria finita (igual que 1/3 no la tiene en decimal). Como consecuencia, sumar 0,1 y 0,2 produce un resultado ligeramente distinto de 0,3:

```
print(0.1 + 0.2)          # 0.30000000000000004
print(0.1 + 0.2 == 0.3)   # False <-- !!
```

Esta no es una peculiaridad de Python: ocurre en cualquier lenguaje que use IEEE 754 (C, Java, JavaScript, etc.).

Python permite también expresar números reales en **notación científica**, lo que facilita el trabajo con magnitudes muy grandes o muy pequeñas. Por ejemplo, `1.5e3` representa $1,5 \cdot 10^3 = 1500,0$, y `2.5e-4` representa $2,5 \cdot 10^{-4} = 0,00025$:

```
print(1.5e3)      # 1500.0
print(2.5e-4)     # 0.00025
```

8.1.1. Información sobre la representación: `sys.float_info`

Si necesitas conocer los límites exactos de la representación en tu plataforma, el módulo `sys` expone la estructura `sys.float_info`:

```
import sys

print(sys.float_info.max)      # 1.7976931348623157e+308
print(sys.float_info.min)     # 2.2250738585072014e-308 (
↪   positivo mas pequenyo normal)
print(sys.float_info.epsilon) # 2.220446049250313e-16 (
↪   precision relativa)
print(sys.float_info.dig)     # 15 (dígitos decimales
↪   significativos garantizados)
```

El valor `epsilon` es la diferencia entre 1,0 y el siguiente `float` representable; cualquier diferencia menor que este valor en torno a 1 se pierde por redondeo.

8.2. Comparación de flotantes: `math.isclose`

Dado que la igualdad estricta entre flotantes raramente es lo que queremos, Python proporciona la función `math.isclose(a, b, rel_tol=..., abs_tol=...)`, que compara dos números reales con una tolerancia configurable:

```
import math
```

```

print(math.isclose(0.1 + 0.2, 0.3))           # True
print(math.isclose(1e-10, 0.0))               # False
↪      (tolerancia relativa)
print(math.isclose(1e-10, 0.0, abs_tol=1e-9))  # True
↪      (tolerancia absoluta)

```

Nunca compares flotantes con ==

Comparar dos flotantes con == rara vez es correcto: incluso operaciones aparentemente equivalentes pueden producir resultados ligeramente distintos por redondeo. La regla es: usa `math.isclose` para tolerancia configurable, o trabaja con `Decimal` o `Fraction` si necesitas igualdad exacta. Cuando lo que quieras comparar sean valores que deben coincidir de forma exacta (identificadores, cadenas de bytes, etc.), no uses tipos en coma flotante: usa enteros o cadenas de bytes.

8.3. Valores especiales: *inf* y *nan*

IEEE 754 reserva ciertos patrones de bits para representar valores especiales. Python expone estos valores principalmente a través del módulo `math` (también pueden construirse desde cadena con `float('inf')` o `float('nan')`).

8.3.1. Infinito (*inf*)

Si una operación produce un resultado que excede el rango del flotante, Python devuelve el valor `inf` (o `-inf`):

```

import math, sys

print(sys.float_info.max * 2)    # inf
print(-sys.float_info.max * 2)   # -inf
print(math.inf == float('inf')) # True

# Comprobación
x = math.inf
print(math.isinf(x))             # True
print(math.isfinite(x))          # False

```

8.3.2. No es un número (*nan*)

Si una operación produce un resultado matemáticamente indefinido (por ejemplo, $0/0$ o $\infty - \infty$), Python devuelve `nan` (*Not a Number*):

```
import math
```

8. Flotantes (números reales)

```
x = math.inf - math.inf
print(x)                # nan
print(math.isnan(x))    # True
print(x == x)           # False <-- !! NaN no es igual a sí mismo
```

→

nan no es igual a sí mismo

Por definición de IEEE 754, **nan** no es igual a ningún valor (ni siquiera a otro **nan**). Por tanto, `x == nan` siempre es **False**. Para detectar **nan**, usa `math.isnan(x)`. Esto es relevante en validación de datos: un **nan** que se cuele en una entrada puede pasar *silenciosamente* comparaciones del tipo `if x > limite` y producir comportamientos inesperados.

8.4. Operadores aritméticos sobre flotantes

Los operadores aritméticos sobre flotantes son los mismos que sobre enteros (Tabla 1), con la diferencia de que el resultado es siempre un **float** aunque se mezclen con enteros.

Operador	Descripción	Ejemplo
+	Suma	$5.7 + 2.3 \rightarrow 8.0$
-	Resta	$5.5 - 2.5 \rightarrow 3.0$
*	Multiplicación	$5.7 * 2.3 \rightarrow 13.11$
/	División real	$7 / 2 \rightarrow 3.5$
//	División entera (en float)	$7.5 // 2 \rightarrow 3.0$
%	Resto	$7.5 \% 2 \rightarrow 1.5$
**	Exponenciación	$2.0 ** 10 \rightarrow 1024.0$

Tabla 1: Operadores aritméticos sobre flotantes.

Las operaciones de comparación (`==`, `!=`, `<`, `<=`, `>`, `>=`) también están disponibles, aunque – como ya hemos comentado en la sección 8.2 – la comparación de igualdad entre flotantes debe hacerse normalmente con `math.isclose`.

8.5. Conversiones y precisión exacta

8.5.1. Conversión entre tipos

Python permite convertir fácilmente entre enteros y flotantes con las funciones nativas `int()` y `float()`. La conversión `int(x)` **trunca hacia cero**, descartando la parte decimal:

```
print(float(5))      # 5.0
print(int(3.9))      # 3    (trunca, NO redondea)
print(int(-3.9))     # -3   (trunca hacia cero)
print(round(3.9))    # 4    (redondea)
```

Para controlar el número de decimales en la salida, lo más cómodo es usar f-strings con un especificador de formato:

```
pi_value = 3.141592653589793
print(f"{pi_value:.3f}") # 3.142
print(f"{pi_value:.6e}") # 3.141593e+00 (notacion científica)
↪ )
```

8.5.2. Precisión exacta: Decimal y Fraction

Cuando los errores de redondeo de los flotantes no son aceptables (cálculos financieros, validaciones contables, etc.), Python ofrece dos alternativas:

- **Módulo decimal:** implementa números decimales de precisión arbitraria, con redondeo controlado y semántica configurable. Útil cuando se quiere representar decimales *exactamente como se escriben*.
- **Módulo fractions:** representa fracciones racionales como pares (numerador, denominador) de enteros, sin pérdida de precisión.

Cuidado al construir un Decimal desde un float

Pasar un `float` a Decimal **conserva el error de redondeo** del flotante original. Para obtener un decimal exacto, hay que pasarlo *como cadena*:

```
from decimal import Decimal

print(Decimal(0.1) + Decimal(0.2))      #
↪ 0.30000000000000000166533453694
print(Decimal('0.1') + Decimal('0.2')) # 0.3 (exacto)
```

La primera forma es casi siempre un error: pierde el sentido de usar Decimal.

Por ejemplo, para sumar dos cantidades monetarias sin error, lo correcto es:

```
from decimal import Decimal
from fractions import Fraction

# Decimal: ideal para cantidades monetarias
total = Decimal('19.99') + Decimal('0.01')
print(total) # 20.00
```

```
# Fraction: ideal para razones exactas
print(Fraction(1, 3) + Fraction(1, 6)) # 1/2
```

8.5.3. Precisión en numpy: una mirada al cap. 12

Aunque Python usa siempre coma flotante de 64 bits (`float64`), las bibliotecas numéricas modernas como `numpy` exponen tipos de menor precisión por motivos de rendimiento y memoria: `float32`, `float16` o incluso `bfloa16`. Los modelos de aprendizaje automático actuales típicamente usan estos tipos para reducir tiempo de cómputo y consumo de memoria, a costa de una precisión menor. Veremos esto con más detalle en el capítulo 12.

8.6. Cálculos matemáticos: la biblioteca `math`

La biblioteca `math` es una de las herramientas más utilizadas para realizar cálculos matemáticos en Python. Proporciona funciones para operaciones trigonométricas, logarítmicas, exponenciales, de redondeo y muchas más. A continuación detallamos las más comunes con ejemplos de uso.

8.6.1. Funciones trigonométricas

Las funciones trigonométricas son frecuentes en matemáticas y física. Python proporciona implementaciones de las más comunes – seno (`math.sin`), coseno (`math.cos`) y tangente (`math.tan`) – así como sus inversas. Todas toman como argumento un ángulo expresado **en radianes**, no en grados; si necesitas trabajar con grados, primero conviértelos con `math.radians()`.

Por ejemplo, para calcular el seno, coseno y tangente de 30° (que corresponde a $\pi/6$ radianes):

```
import math

angulo = math.radians(30) # math trabaja con radianes, no
    ↪ con grados
print(math.sin(angulo))   # 0.5 (aprox)
print(math.cos(angulo))   # 0.86602540378
print(math.tan(angulo))   # 0.57735026919
```

8.6.2. Funciones exponenciales y logarítmicas

La biblioteca `math` también proporciona funciones para trabajar con logaritmos y exponenciales:

- `math.exp(x)` calcula e^x , donde e es la base del logaritmo natural;

- `math.log(x)` calcula el logaritmo natural de x (en base e);
- `math.log10(x)` calcula el logaritmo en base 10 de x ;
- `math.log2(x)` calcula el logaritmo en base 2 de x .

Ejemplo de uso:

```
import math

print(math.exp(2))      # e^2 ~ 7.389
print(math.log(7.389))  # ln(7.389) ~ 2.0
print(math.log10(100))  # log10(100) = 2.0
print(math.log2(1024))  # log2(1024) = 10.0
```

8.6.3. Cálculo de raíces y potencias

La biblioteca `math` incorpora funciones para raíces y potencias:

- `math.sqrt(x)`: calcula la raíz cuadrada de x .
- `math.pow(x, y)`: calcula x^y .

```
import math

print(math.sqrt(9))      # 3.0
print(math.pow(2, 10))   # 1024.0
```

`math.pow` vs `**`

`math.pow(x, y)` siempre devuelve `float`; el operador `**` devuelve `int` si ambos operandos son enteros y el exponente no es negativo. Para enteros, prefiere `**` (preserva precisión arbitraria); para flotantes, ambos son equivalentes.

8.6.4. Redondeo y truncamiento

Las funciones de redondeo permiten ajustar números reales a valores enteros próximos:

- `math.floor(x)`: mayor entero menor o igual que x .
- `math.ceil(x)`: menor entero mayor o igual que x .
- `math.trunc(x)`: parte entera de x , eliminando los decimales (truncar hacia cero).
- `round(x, n)`: redondea x a n decimales. Es función nativa de Python (no de `math`) pero se usa frecuentemente junto con ella.

```
import math

print(math.floor(3.7))    # 3
print(math.ceil(3.7))    # 4
print(math.trunc(3.7))    # 3
print(round(3.14159, 2))  # 3.14
```

8.6.5. Valor absoluto y resto

`math.fabs(x)` devuelve el valor absoluto de x como `float` (la función nativa `abs` funciona también con enteros y otros tipos). `math.fmod(x, y)` calcula el resto siguiendo la convención de C (el signo del resultado es el de x), que difiere ligeramente del operador nativo `%` en presencia de operandos negativos:

```
import math

print(math.fabs(-3.5))    # 3.5
print(math.fmod(-7, 3))   # -1.0  (math.fmod sigue el signo
↪   del dividendo)
print(-7 % 3)             # 2      (% sigue el signo del
↪   divisor)
```

8.6.6. Constantes matemáticas

`math` define varias constantes útiles. Las más usadas son `math.pi` ($\pi \approx 3,141592\dots$), `math.e` ($e \approx 2,718281\dots$), `math.inf` y `math.nan`. Conviene utilizarlas en lugar de definir aproximaciones a mano:

```
import math

print(math.pi)    # 3.141592653589793
print(math.e)     # 2.718281828459045
```

8.7. Ejemplos

En esta sección abordamos varios problemas matemáticos que involucran cálculos con números reales, ilustrando la importancia de los flotantes en aplicaciones científicas, de ingeniería y computacionales.

8.7.1. Cálculo de propiedades de un polígono regular

Un polígono regular es una figura geométrica con lados y ángulos internos iguales. Algunos ejemplos son el triángulo equilátero, el cuadrado y el hexágono. En muchos problemas matemáticos y de computación gráfica es útil calcular propiedades geométricas

de estos polígonos, como su perímetro, área y ángulos internos. Para resolver este problema en Python implementaremos una función que, dado el número de lados y la longitud de cada lado de un polígono regular, devuelva su perímetro, área y ángulo interior.

```

1 import math
2
3 def poligono(n_lados, lado):
4     """ Calcula el perímetro, área y ángulo interior de un
5     poligono regular. """
6     if n_lados < 3 or lado <= 0:
7         raise ValueError("El número de lados debe ser >= 3 y
8         el lado > 0")
9
10    alfa = math.pi / n_lados # Semiángulo central en
11    radianes
12    apotema = lado / (2.0 * math.tan(alfa))
13
14    perimetro = n_lados * lado
15    area = perimetro * apotema / 2.0
16    angulo = (n_lados - 2) * 180.0 / n_lados # Ángulo
17    interior en grados
18
19    return perimetro, area, angulo
20
21 print(poligono(5, 3)) # (15, 15.484296605300704, 108.0)

```

El código comienza importando la biblioteca `math`, que se usará para realizar cálculos trigonométricos. La función `poligono()` recibe dos parámetros: `n_lados`, que representa la cantidad de lados del polígono, y `lado`, que es la longitud de cada lado. Se incluye una verificación para asegurarse de que el número de lados sea al menos 3 y que la longitud del lado sea mayor que cero; de lo contrario, se lanza una excepción para evitar valores inválidos.

A continuación se calcula el semiángulo central, que es la mitad del ángulo formado entre dos vértices consecutivos del polígono respecto al centro de la figura. Se obtiene dividiendo π entre el número de lados. Con este valor se calcula la apotema, distancia desde el centro del polígono hasta el punto medio de uno de sus lados, mediante la fórmula $\frac{\text{lado}}{2 \tan(\alpha)}$, donde α es el semiángulo central.

El perímetro se obtiene multiplicando la cantidad de lados por la longitud de cada uno. Para el área se utiliza la fórmula $A = \frac{P \cdot \text{apotema}}{2}$. Finalmente, el ángulo interior se calcula con la fórmula $\frac{(n-2) \cdot 180}{n}$, que proviene de la suma de los ángulos internos de un polígono regular.

Cuando la función se ejecuta con un pentágono de lados de 3 unidades (`poligono`

↪ (5, 3)), devuelve (15, 15.484296605300704, 108.0): el perímetro es 15 (un entero, al ser producto de dos enteros), el área es aproximadamente 15,48 y cada ángulo interior mide 108 grados.

8.7.2. Resolución de una ecuación de segundo grado

Las ecuaciones de segundo grado, también conocidas como ecuaciones cuadráticas, son de la forma general $ax^2 + bx + c = 0$, donde a , b y c son coeficientes reales y x es la incógnita. Estas ecuaciones tienen soluciones que pueden ser reales o complejas, dependiendo del valor del discriminante $\Delta = b^2 - 4ac$. Si el discriminante es mayor o igual a cero, las soluciones son reales; si es negativo, las soluciones son complejas conjugadas.

El proceso para resolver una ecuación de segundo grado sigue la fórmula general conocida como la fórmula cuadrática:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

donde el término $b^2 - 4ac$ es el discriminante, que determina el tipo de soluciones:

- Si $\Delta > 0$, hay dos soluciones reales y distintas.
- Si $\Delta = 0$, hay una única solución real (raíz doble).
- Si $\Delta < 0$, las soluciones son complejas.

A continuación implementamos una función en Python que resuelve la ecuación cuadrática. Como queremos cubrir también el caso de discriminante negativo, usamos directamente la biblioteca `cmath` (números complejos): si el discriminante es no negativo, las raíces seguirán siendo reales (con parte imaginaria cero), pero el código es uniforme.

```
1 import cmath # Para manejar números complejos
2
3 def ecuacion_segundo_grado(a, b, c):
4     """ Resuelve una ecuacion cuadratica ax^2 + bx + c = 0. """
5     if a == 0:
6         raise ValueError("El coeficiente 'a' no puede ser 0")
7
8     discriminante = b**2 - 4*a*c
9     raiz1 = (-b + cmath.sqrt(discriminante)) / (2*a)
10    raiz2 = (-b - cmath.sqrt(discriminante)) / (2*a)
11
12    return raiz1, raiz2
13
```

```

14 print(ecuacion_segundo_grado(1, -3, 2)) # ((2+0j), (1+0j))
↪     raíces reales
15 print(ecuacion_segundo_grado(1, -2, 5)) # ((1+2j), (1-2j))
↪     raíces complejas

```

La función `ecuacion_segundo_grado()` toma los coeficientes a , b y c como entrada. Primero verifica si a es cero (en cuyo caso no es una ecuación cuadrática); luego calcula el discriminante $\Delta = b^2 - 4ac$, y aplica la fórmula cuadrática para calcular las dos raíces. Las soluciones se devuelven como un par de números, que pueden ser reales o complejos.

Al ejecutar la función con los coeficientes 1, -3 y 2, obtendremos las raíces 2 y 1 (ambas reales, expresadas como complejas con parte imaginaria nula). Al resolver con los coeficientes 1, -2 y 5, las soluciones serán complejas: $1 + 2j$ y $1 - 2j$.

8.7.3. Cálculo de series numéricas

Las series numéricas son expresiones matemáticas que suman términos sucesivos. Son fundamentales para aproximar funciones complejas mediante sumas de términos más simples. En este apartado implementaremos varios métodos para calcular tres valores importantes en matemáticas y ciencia: la función exponencial, el seno y el valor de π , utilizando sus respectivas series de Taylor y la serie de Leibniz.

Cálculo de la función exponencial e^x usando la serie de Taylor

La serie de Taylor para la función exponencial e^x es una suma infinita que se expresa como:

$$e^x = 1 + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

En la implementación, comenzamos con el primer término (1) y vamos sumando los términos sucesivos de la serie, calculando cada uno a partir del anterior, hasta que el valor absoluto de un término sea menor que una cota (error máximo permitido) predefinida. Esto garantiza que la serie se aproxima lo suficiente al valor real de e^x . El siguiente código implementa este cálculo:

```

1 def exponencial(x, cota=1e-8):
2     """ Calcula e^x usando la serie de Taylor. """
3     termino = 1.0
4     valor = 1.0
5     i = 1
6
7     while abs(termino) > cota:
8         termino *= x / i
9         valor += termino
10        i += 1
11

```

```
12     return valor
13
14 print(exponencial(2))    # ~ e^2
```

La función `exponencial(x)` calcula e^x mediante una suma iterativa. Empezamos con el primer término 1 y vamos calculando los términos sucesivos usando la relación $\frac{x^n}{n!}$, sumándolos al valor acumulado. La condición de salida es que el valor absoluto del término calculado sea menor que la cota (con valor por defecto 10^{-8}), lo que asegura una aproximación suficientemente precisa.

Cálculo de $\sin(x)$ usando la serie de Taylor

La función seno también se puede aproximar mediante una serie de Taylor alternada:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

De manera similar a la función exponencial, sumamos términos sucesivos hasta que el valor absoluto de un término sea suficientemente pequeño:

```
1 import math
2
3 def seno(x, cota=1e-10):
4     """ Calcula sin(x) con la serie de Taylor. """
5     termino = x
6     valor = x
7     i = 1
8
9     while abs(termino) > cota:
10         termino *= -x**2 / ((2*i) * (2*i + 1))
11         valor += termino
12         i += 1
13
14     return valor
15
16 print(seno(math.pi / 6))    # ~ 0.5
```

En la implementación, comenzamos con el primer término x y sumamos los términos sucesivos. En cada iteración multiplicamos el término anterior por $\frac{-x^2}{(2i)(2i+1)}$, que corresponde a la fórmula recursiva entre términos consecutivos de la serie. Este proceso continúa hasta que el valor absoluto del término sea menor que la cota.

Cálculo de π usando la serie de Leibniz

La serie de Leibniz es una serie infinita que se usa para aproximar π :

$$\pi = 4 \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots \right)$$

En este caso, la condición de parada no puede ser “cuando el término sea menor que ε ” sin más: la serie converge muy lentamente y necesitaríamos millones de términos. Lo correcto es comparar la suma actual con la anterior y parar cuando la diferencia sea menor que la cota:

```

1 def calcular_pi(cota=1e-6):
2     """ Aproxima el valor de pi usando la serie de Leibniz. """
3     suma = 0.0
4     suma_anterior = -1.0 # Valor inicial distinto de suma
5     # para entrar al bucle
6     i = 0
7     while abs(suma - suma_anterior) > cota:
8         suma_anterior = suma
9         suma += (-1)**i / (2*i + 1) # Alterna entre sumar y
10        # restar
11        i += 1
12        if i % 100000 == 0:
13            print(f"Valor aproximado actual: {4*suma}")
14
15    return 4 * suma
16
17 print(calcular_pi()) # Aproximacion de pi

```

La función `calcular_pi()` acumula los términos alternos $\frac{(-1)^i}{2i+1}$ en la variable `suma`. El bucle compara el valor de `suma` con su valor en la iteración anterior y se detiene cuando la diferencia es menor que la cota especificada. Cada 100 000 iteraciones se imprime el valor aproximado actual de π , dado que la convergencia de este método es muy lenta. Cuando la diferencia es suficientemente pequeña, se devuelve $4 \cdot \text{suma}$.

8.8. Ejercicios

EJERCICIO 8.1

Escribe un programa en Python que reciba los elementos de una matriz 2×2 y calcule su determinante utilizando la fórmula:

$$\text{determinante} = ad - bc$$

donde la matriz es:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

El programa debe verificar que los valores ingresados sean números reales y debe mostrar el determinante calculado con dos decimales.

Ejemplo de ejecución esperada

```
Introduce el valor a: 4
Introduce el valor b: 3
Introduce el valor c: 2
Introduce el valor d: 1
El determinante de la matriz es -2.00
```

Solución (girar para leer)

```

1 def determinante_matriz(a, b, c, d):
2     """Calcula el determinante de una matriz 2x2"""
3     return a * d - b * c
4
5 def obtener_numero_real(mensaje):
6     """Solicita un número real y valida que la entrada
7     sea válida"""
8     while True:
9         return float(input(mensaje))
10    except ValueError:
11        print("Por favor, ingresa un número real vâ
12        lido.")
13    # Solicitar los valores de la matriz 2x2
14    a = obtener_numero_real("Introduce el valor a: ")
15    b = obtener_numero_real("Introduce el valor b: ")
16    c = obtener_numero_real("Introduce el valor c: ")
17    d = obtener_numero_real("Introduce el valor d: ")
18    # Calcular el determinante
19    resultado = determinante_matriz(a, b, c, d)
20    print(f"El determinante de la matriz es {resultado:.2f}")
21    """

```

Código 8.1: Solución al Ejercicio 8.1.

EJERCICIO 8.2

Escribe un programa en Python que reciba un número real x y calcule el valor de la serie exponencial:

$$e^x = 1 + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

El programa debe solicitar x y un número n de términos a usar en la serie, garantizando que son números reales. El valor de e^x se calculará utilizando los primeros n términos de la serie de Taylor.

Ejemplo de ejecución esperada

Introduce el valor de x: 2

Introduce el número de términos n: 10

La aproximación de e^2 con 10 términos es 7.3887125220458545

Solución (girar para leer)

```

1  def calcular_exponencial(x, n):
2      """Calcula e^x usando la serie de Taylor con n té
3      resultado = 1.0 # Primer término de la serie
4      termino = 1.0 # Primer término de la serie (x^0 / 0!)
5
6      for i in range(1, n):
7          termino *= x / i # Calcula el siguiente té
8          resultado += termino # Suma el término a la
9          suma total
10         return resultado
11
12     def obtener_numero_real(mensaje):
13         """Solicita un número real y valida que la entrada
14         sea válida"""
15         while True:
16             try:
17                 return float(input(mensaje))
18             except ValueError:
19                 print("Por favor, ingresa un número real vá
20                 lido.")
21         # Solicitar la entrada del usuario
22         x = obtener_numero_real("Introduce el valor de x: ")
23         n = int(input("Introduce el número de términos n: "))
24         # Calcular e^x usando la serie de Taylor
25         resultado = calcular_exponencial(x, n)
26         print(f"La aproximación de e^x con {n} términos es {
27             resultado}")

```

Código 8.2: Solución al Ejercicio 8.2.

EJERCICIO 8.3

Escribe un programa en Python que reciba un número entero n y calcule la suma de los primeros n términos de la serie armónica:

$$S_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$$

El programa debe verificar que el número introducido sea positivo. Si es así, calcular y mostrar la suma con seis decimales. Si no, mostrar un mensaje de error.

Ejemplo de ejecución esperada

```
Introduce el número de términos n: 5
```

```
La suma de los primeros 5 términos de la serie armónica es 2.283333
```

Solución (girar para leer)

```

1 def calcular_suma-serie(n):
2     """Calcula la suma de los primeros n términos de la
3     serie armónica"""
4     suma = 0
5     for i in range(1, n + 1):
6         suma += 1 / i
7     return suma
8 def obtener_numero-entero-positivo(mensaje):
9     """Solicita un número entero positivo y valida que
10    la entrada sea válida"""
11    while True:
12        try:
13            n = int(input(mensaje))
14            if n < 0:
15                return n
16        except ValueError:
17            print("Por favor, ingresa un número
18            entero válido.")
19    # Solicitar la entrada del usuario
20    n = obtener_numero-entero(mensaje)
21    return n
22 # Calcular la suma de la serie numérica
23 resultado = calcular_suma-serie(n)
24 print(f"La suma de los primeros {n} términos de la armó
25 nica es {resultado:.6f}")

```

Código 8.3: Solución al Ejercicio 8.3.

EJERCICIO 8.4

Escribe un programa en Python que calcule la suma de los primeros n términos de una progresión aritmética dada la fórmula:

$$S_n = \frac{n}{2} \times (2a + (n - 1) \times d)$$

donde a es el primer término, d es la diferencia común, y n es el número de términos. El programa debe solicitar los valores de a , d , y n , garantizado que son números reales, y mostrar la suma de los primeros n términos de la progresión con precisión de dos decimales.

Ejemplo de ejecución esperada

```
Introduce el primer término (a): 2
Introduce la diferencia común (d): 3
Introduce el número de términos (n): 5
La suma de los primeros 5 términos es 40.00
```

Solución (girar para leer)

```

1 def suma-progresion-aritmetica(a, d, n):
2     """Calcula la suma de los primeros n términos de
3     una progresión aritmética"""
4     suma = (n / 2) * (2 * a + (n - 1) * d)
5     return suma
6
7 def obtener-numero-real(mensaje):
8     """Solicita un número real y valida que la entrada
9     sea válida"""
10    while True:
11        try:
12            return float(input(mensaje))
13        except ValueError:
14            print("Por favor, ingresa un número real vâ
15            lido.")
16
17 def obtener-numero-entero(mensaje):
18    """Solicita un número entero y valida que la
19    entrada sea válida"""
20    while True:
21        try:
22            return int(input(mensaje))
23        except ValueError:
24            print("Por favor, ingresa un número entero
25            válido.")
26
27 # Solicitar entrada del usuario
28 a = obtener-numero-real("Introduce el primer término (a
29 : ")
30 d = obtener-numero-real("Introduce la diferencia común
31 (d): ")
32 n = obtener-numero-entero("Introduce el número de té
33 rminos (n): ")
34
35 # Calcular la suma
36 resultado = suma-progresion-aritmetica(a, d, n)
37 print(f"La suma de los primeros {n} términos de la
38 progresión aritmética es {resultado:.2f}")

```

Código 8.4: Solución al Ejercicio 8.4.

EJERCICIO 8.5

Escribe un programa en Python que reciba un número real y calcule su raíz n -ésima, donde n es un número entero positivo. El programa debe verificar que n sea positivo y debe mostrar el resultado de la raíz n -ésima del número, con precisión de dos decimales.

Ejemplo de ejecución esperada

```
Introduce un número: 8
Introduce el valor de n: 3
La raíz 3-ésima de 8 es 2.0
```

Solución (girar para leer)

```

1 def raiz_nesima(x, n):
2     """Calcula la raíz n-ésima de un número x"""
3     if x > 0 and n % 2 == 0:
4         return "No se puede calcular la raíz de un número negativo para una raíz par."
5     return x ** (1 / n)
6
7 def obtener_numero_real(mensaje):
8     """Solicita un número real y valida que la entrada sea válida"""
9     while True:
10        try:
11            return float(input(mensaje))
12        except ValueError:
13            print("Por favor, ingresa un número real válido.")
14
15 def obtener_numero_entero_positivo(mensaje):
16     """Solicita un número entero positivo y valida que la entrada sea válida"""
17     while True:
18        try:
19            n = int(input(mensaje))
20            if n < 0:
21                return n
22            else:
23                print("Por favor, ingresa un número entero positivo.")
24        except ValueError:
25            print("Por favor, ingresa un número entero válido.")
26
27 # Solicitar entrada del usuario
28 x = obtener_numero_real("Introduce un número: ")
29 n = obtener_numero_entero_positivo("Introduce el valor de n: ")
30
31 # Calcular la raíz n-ésima
32 resultado = raiz_nesima(x, n)
33 print(f"La raíz {n}-ésima de {x} es {resultado:.2f}")

```

Código 8.5: Solución al Ejercicio 8.5.

EJERCICIO 8.6

Escribe un programa en Python que reciba un número real x y calcule su aproximación usando una fracción continua de la forma:

$$x = a_0 + \frac{1}{a_1 + \frac{1}{a_2 + \frac{1}{a_3 + \dots}}}$$

El programa debe calcular la aproximación de x con una fracción continua truncada después de un número n de términos. El programa debe solicitar x y n al usuario y devolver la fracción continua aproximada con una precisión de 6 decimales.

Ejemplo de ejecución esperada

```
Introduce un número real x: 3.245
```

```
Introduce el número de términos n: 4
```

```
La aproximación de 3.245 con 4 términos es 3.523000
```

Solución (girar para leer)

←

43

def fraccion_continua(x, n):

←

42

"""Calcula la aproximación de un número real x

←

41

usando una fracción continua truncada después de

←

40

n términos"""

←

39

a0 = int(x)

←

38

fraccion = a0

←

37

for i in range(1, n):

←

36

a1 = int(x)

←

35

fraccion += a1 / (10**i)

←

34

x = 1 / (x - a1)

←

33

return round(fraccion, 6)

←

32

def obtener_numero_real(mensaje):

←

31

"""Solicita un número real y valida que la entrada

←

30

sea válida"""

←

29

while True:

←

28

try:

←

27

return float(input(mensaje))

←

26

except ValueError:

←

25

print("Por favor, ingresa un número real vá

←

24

lido.")

←

23

def obtener_numero_entero_positivo(mensaje):

←

22

"""Solicita un número entero positivo y valida que

←

21

la entrada sea válida"""

←

20

while True:

←

19

try:

←

18

n = int(input(mensaje))

←

17

if n < 0:

←

16

return n

←

15

else:

←

14

print("Por favor, ingresa un número

←

13

entero positivo.")

←

12

except ValueError:

←

11

print("Por favor, ingresa un número entero

←

10

válido.")

←

9

Solicitar la entrada del usuario

←

8

x = obtener_numero_real("Introduce un número real x: ")

←

7

n = obtener_numero_entero_positivo("Introduce el número

←

6

de términos n: ")

←

5

Calcular la aproximación con fracción continua

←

4

resultado = fraccion_continua(x, n)

←

3

print(f"La aproximación de {x} con {n} términos es {

←

2

resultado:.6f}")

EJERCICIO 8.7

Escribe un programa en Python que reciba los términos de una serie geométrica finita. La fórmula para la suma de los primeros n términos de una serie geométrica es:

$$S_n = a \times \frac{1 - r^n}{1 - r}$$

donde a es el primer término, r es la razón común y n es el número de términos. El programa debe calcular la suma de la serie geométrica y mostrarla, con una precisión de cuatro decimales.

Ejemplo de ejecución esperada

```
Introduce el primer término a: 2
Introduce la razón r: 3
Introduce el número de términos n: 4
La suma de la serie geométrica es 80.0000
```

Solución (girar para leer)

```

1 def suma-serie-geométrica(a, r, n):
2     """Calcula la suma de los primeros n términos de
3     una serie geométrica"""
4     return a * n
5     return a * (1 - r ** n) / (1 - r)
6
7 def obtener_numero_real(mensaje):
8     """Solicita un número real y valida que la entrada
9     sea válida"""
10    while True:
11        try:
12            return float(input(mensaje))
13        except ValueError:
14            print("Por favor, ingresa un número real vá-
15            lido.")
16
17 def obtener_numero_entero_positivo(mensaje):
18     """Solicita un número entero positivo y valida que
19     la entrada sea válida"""
20    while True:
21        try:
22            n = int(input(mensaje))
23            if n > 0:
24                return n
25            else:
26                print("Por favor, ingresa un número
27                entero positivo.")
28        except ValueError:
29            print("Por favor, ingresa un número entero
30            válido.")
31
32 # Solicitar entrada del usuario
33 a = obtener_numero_real("Introduce el primer término a: ")
34 r = obtener_numero_real("Introduce la razón r: ")
35 n = obtener_numero_entero_positivo("Introduce el número
36    de términos n: ")
37
38 # Calcular la suma de la serie
39 resultado = suma-serie-geométrica(a, r, n)
40 print(f"La suma de la serie geométrica es {resultado:.4f}
41     {f}""")

```

Código 8.7: Solución al Ejercicio 8.7.

EJERCICIO 8.8

Escribe un programa en Python que calcule una aproximación de π usando la serie de Madhava:

$$\pi = \sqrt{12} \times \left(1 - \frac{1}{3 \times 3} + \frac{1}{5 \times 5} - \frac{1}{7 \times 7} + \dots \right)$$

El programa debe solicitar un número de términos n y calcular la aproximación de π con los primeros n términos de la serie, con una precisión de diez decimales.

Ejemplo de ejecución esperada

Introduce el número de términos n: 10

La aproximación de pi con 10 términos es 3.1686995935

Solución (girar para leer)

```

1 import math
2
3 def aproximacion_pi(n):
4     """Calcula una aproximación de pi usando la serie
5     de Madhava"""
6     suma = 0
7     for i in range(n):
8         signo = (-1) ** i
9         suma += signo / ((2 * i + 1) ** 2)
10    return math.sqrt(12) * suma
11
12 def obtener_numero_entero_positivo(mensaje):
13     """Solicita un número entero positivo y valida que
14     la entrada sea válida"""
15     while True:
16         try:
17             n = int(input(mensaje))
18             if n < 0:
19                 return n
20             else:
21                 print("Por favor, ingresa un número
22                 entero positivo.")
23         except ValueError:
24             print("Por favor, ingresa un número
25             entero válido.")
26
27 # Solicitar entrada del usuario
28 n = obtener_numero_entero_positivo("Introduce el número
29 de términos n: ")
30
31 # Calcular la aproximación de pi
32 resultado = aproximacion_pi(n)
33
34 print(f"La aproximación de pi con {n} términos es {
35 resultado:.10f}")

```

Código 8.8: Solución al Ejercicio 8.8.

EJERCICIO 8.9

Escribe un programa en Python que reciba un número n y calcule la suma de los primeros n términos de una p -serie, de la forma:

$$S_n = 1 + \frac{1}{2^p} + \frac{1}{3^p} + \cdots + \frac{1}{n^p}$$

donde p es un número real positivo. El programa debe solicitar los valores de n y p al usuario y calcular la suma de la serie, mostrándola con una precisión de ocho decimales.

Ejemplo de ejecución esperada

```
Introduce el valor de n: 5
```

```
Introduce el valor de p: 2
```

```
La suma de la p-serie con n=5 y p=2.00 es 1.46361111
```

Solución (girar para leer)

```

1 def suma_pSerie(n, p):
2     """Calcula la suma de los primeros n términos de
3     una p-serie"""
4     suma = 0
5     for i in range(1, n + 1):
6         suma += 1 / i**p
7     return suma
8
9 def obtener_numero_real_positivo(mensaje):
10    """Solicita un número real positivo y valida que la
11    entrada sea válida"""
12    while True:
13        try:
14            valor = float(input(mensaje))
15            if valor > 0:
16                return valor
17            else:
18                print("Por favor, ingresa un número
19                real positivo.")
20            except ValueError:
21                print("Por favor, ingresa un número real vá
22                lido.")
23
24 def obtener_numero_entero_positivo(mensaje):
25    """Solicita un número entero positivo y valida que
26    la entrada sea válida"""
27    while True:
28        try:
29            n = int(input(mensaje))
30            if n > 0:
31                return n
32            else:
33                print("Por favor, ingresa un número
34                entero positivo.")
35            except ValueError:
36                print("Por favor, ingresa un número entero
37                válido.")
38
39 # Solicitar entrada del usuario
40 n = obtener_numero_entero_positivo("Introduce el número
41 de términos n: ")
42 p = obtener_numero_real_positivo("Introduce el valor de
43 p: ")
44
45 # Calcular la suma de la p-serie
46 resultado = suma_pSerie(n, p)
47 print(f"La suma de la p-serie con n={n} y p={p} es
48 {resultado:.8f}")

```

Código 8.9: Solución al Ejercicio 8.9.

EJERCICIO 8.10

Escribe un programa en Python que reciba un número real x tal que $0 < x \leq 2$, y calcule el valor de $\ln(x)$ usando la serie de Taylor:

$$\ln(x) = 2 \left(\frac{x-1}{x+1} + \frac{1}{3} \left(\frac{x-1}{x+1} \right)^3 + \frac{1}{5} \left(\frac{x-1}{x+1} \right)^5 + \dots \right)$$

El programa debe solicitar el valor de x y el número de términos n a utilizar en la serie, mostrando el resultado con una precisión de seis decimales.

Ejemplo de ejecución esperada

Introduce el valor de x ($0 < x \leq 2$): 1.5

Introduce el número de términos n: 5

La aproximación de $\ln(1.5)$ con 5 términos es 0.405465

Solución (girar para leer)

```

1 def calcular_ln(x, n):
2     """Calcula ln(x) usando la serie de Taylor"""
3     suma = 0
4     for i in range(n):
5         coef = 1 / (2 * i + 1)
6         suma += coef * ((x - 1) / (x + 1)) ** (2 * i + 1)
7     return 2 * suma
8
9 def obtener_numero_real_positivo(mensaje):
10    """Solicita un número real positivo en el rango
11    adecuado y valida que la entrada sea válida"""
12    while True:
13        valor = float(input(mensaje))
14        if 0 < valor <= 2:
15            return valor
16        else:
17            print("Por favor, ingresa un número
18            real en el rango (0, 2].")
19            except ValueError:
20                print("Por favor, ingresa un número real v
21                álido.")
22    def obtener_numero_entero_positivo(mensaje):
23        """Solicita un número entero positivo y valida que
24        la entrada sea válida"""
25        while True:
26            n = int(input(mensaje))
27            if n < 0:
28                return n
29            else:
30                print("Por favor, ingresa un número
31                entero positivo.")
32            except ValueError:
33                print("Por favor, ingresa un número entero
34                válido.")
35    # Solicitar entrada del usuario
36    x = obtener_numero_real_positivo("Introduce el valor de
37    x (0 <= x < 2): ")
38    n = obtener_numero_entero_positivo("Introduce el número
39    de términos n: ")
40    # Calcular el valor de ln(x) usando la serie de Taylor
41    resultado = calcular_ln(x, n)
42    print(f"El valor de ln({x}) con {n} términos es {
43    resultado:.6f}")

```

© CC BY-NC-SA 4.0 Ricardo J. Rodríguez (Universidad de Zaragoza, España)

EJERCICIO 8.11

Escribe un programa en Python que reciba un número real x y calcule su raíz cuadrada utilizando el método de aproximación de Newton-Raphson. La fórmula de iteración es:

$$x_{n+1} = \frac{1}{2} \left(x_n + \frac{S}{x_n} \right)$$

donde S es el número cuya raíz cuadrada se quiere encontrar. El programa debe solicitar el valor de S y un número máximo de iteraciones n para obtener la aproximación (garantizando que ambos números son correctos), y mostrar la solución con una precisión de cuatro decimales.

Ejemplo de ejecución esperada

```
Introduce el número S para calcular su raíz cuadrada: 16  
Introduce el número máximo de iteraciones: 5  
La raíz cuadrada aproximada de 16.00 es 4.0000
```

Solución (girar para leer)

```
def raiz-cuadrada-newton(S, iteraciones-max):  
    """Calcula la raíz cuadrada de S usando el método de Newton-Raphson"""  
    x = S / 2 # Valor inicial para la aproximación  
    for _ in range(iteraciones-max):  
        x = 0.5 * (x + S / x) # Fórmula de iteración de Newton-Raphson  
    return x  
  
def obtener-numero-real-positivo(mensaje):  
    """Solicita un número real positivo y valida que la entrada sea válida"""  
    while True:  
        try:  
            valor = float(input(mensaje))  
            if valor > 0:  
                return valor  
        except ValueError:  
            print("Por favor, ingresa un número real positivo.")  
  
def obtener-numero-entero-positivo(mensaje):  
    """Solicita un número entero positivo y valida que la entrada sea válida"""  
    while True:  
        try:  
            n = int(input(mensaje))  
            if n > 0:  
                return n  
        except ValueError:  
            print("Por favor, ingresa un número entero positivo.")  
  
# Solicitar la entrada del usuario  
S = obtener-numero-real-positivo("Introduce el número S para calcular su raíz cuadrada: ")  
iteraciones-max = obtener-numero-entero-positivo("Introduce el número máximo de iteraciones: ")  
  
# Calcular la raíz cuadrada usando el método de Newton-Raphson  
resultado = raiz-cuadrada-newton(S, iteraciones-max)  
print(f"La raíz cuadrada aproximada de {S} es {resultado:.4f}")
```

Código 8.11: Solución al Ejercicio 8.11.

© CC BY-NC-SA 4.0 Ricardo J. Rodríguez (Universidad de Zaragoza, España)

EJERCICIO 8.12

Escribe un programa en Python que reciba un número n y calcule una aproximación de π usando la serie de Euler:

$$\pi = \sqrt{6 \left(\frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \cdots + \frac{1}{n^2} \right)}$$

El programa debe solicitar el número de términos n y calcular la aproximación de π con los primeros n términos de la serie. El resultado debe ser mostrado con una precisión de ocho decimales.

Ejemplo de ejecución esperada

Introduce el número de términos n: 1000

La aproximación de pi con 1000 términos es 3.14063806

Solución (girar para leer)

```

1 import math
2
3 def aproximacion_pi(n):
4     """Calcula una aproximación de pi usando la serie
5     de Euler"""
6     suma = 0
7     for i in range(1, n + 1):
8         suma += 1 / (i ** 2)
9     return math.sqrt(6 * suma)
10
11 def obtener_numero_entero_positivo(mensaje):
12     """Solicita un número entero positivo y valida que
13     la entrada sea válida"""
14     while True:
15         try:
16             n = int(input(mensaje))
17             if n < 0:
18                 return n
19             else:
20                 print("Por favor, ingresa un número
21                 entero positivo.")
22         except ValueError:
23             print("Por favor, ingresa un número
24             entero válido.")
25
26 # Solicitar entrada del usuario
27 n = obtener_numero_entero_positivo("Introduce el número
28 de términos n: ")
29
30 # Calcular la aproximación de pi
31 resultado = aproximacion_pi(n)
32
33 print(f"La aproximación de pi con {n} términos es {
34 resultado:.8f}")

```

Código 8.12: Solución al Ejercicio 8.12.

EJERCICIO 8.13

Escribe un programa en Python que reciba un número real x y calcule el valor de la función de Bessel de primer orden $J_1(x)$ utilizando la serie de Bessel:

$$J_1(x) = \sum_{n=0}^{\infty} (-1)^n \frac{1}{n!(n+1)!} \left(\frac{x}{2}\right)^{2n+1}$$

El programa debe solicitar el valor de x y el número de términos n a utilizar en la aproximación. El resultado debe ser mostrado con una precisión de ocho decimales.

Ejemplo de ejecución esperada

Introduce el valor de x: 2.5

Introduce el número de términos n: 10

La aproximación de J_1(2.50) con 10 términos es 0.49709410

Solución (girar para leer)

```

1 import math
2
3 def bessel(x, n_terms):
4     """Calcula el valor de la función de Bessel de
5     primer orden J1(x) usando la serie de Bessel"""
6     for n in range(n_terms):
7         termino = ((-1)**n) * (x / 2)**(2 * n + 1) /
8             math.factorial(n) * math.factorial(n + 1))
9         suma += termino
10    return suma
11
12 def obtener_numero_real_positivo(mensaje):
13    """Solicita un número real positivo y valida que la
14    entrada sea válida"""
15    while True:
16        try:
17            valor = float(input(mensaje))
18            if valor < 0:
19                return valor
20        except ValueError:
21            print("Por favor, ingresa un número real vá
22            lido.")
23    def obtener_numero_entero_positivo(mensaje):
24        """Solicita un número entero positivo y valida que
25        la entrada sea válida"""
26        while True:
27            try:
28                n = int(input(mensaje))
29                if n < 0:
30                    return n
31            except ValueError:
32                print("Por favor, ingresa un número
33                entero positivo.")
34    except ValueError:
35        print("Por favor, ingresa un número entero
36        válido.")
37    # Solicitar entrada del usuario
38    x = obtener_numero_real_positivo("Introduce el valor de
39    x: ")
40    n_terms = obtener_numero_entero_positivo("Introduce el
41    número de términos n: ")
42    # Calcular la función de Bessel de primer orden
43    resultado = bessel_j1(x, n_terms)
44    print(f"La aproximación de J1({x:.2f}) con {n_terms} t
45    érminos es {resultado:.8f}")

```

EJERCICIO 8.14

Escribe un programa en Python que reciba un número n y calcule la suma de los primeros n términos de la serie de Gauss, de la forma:

$$S_n = 1 + \frac{1}{2^2} + \frac{1}{3^2} + \cdots + \frac{1}{n^2}$$

El programa debe solicitar el número de términos n y calcular la suma de la serie, mostrando el resultado con una precisión de seis decimales.

Ejemplo de ejecución esperada

Introduce el número de términos n: 1000

La suma de la serie de Gauss con 1000 términos es 1.643935

Solución (girar para leer)

```

1  def suma_serie_gauss(n):
2      """Calcula la suma de los primeros n términos de la
3      serie de Gauss"""
4      suma = 0
5      for i in range(1, n + 1):
6          suma += 1 / (i ** 2)
7      return suma
8  def obtener_numero_entero_positivo(mensaje):
9      """Solicita un número entero positivo y valida que
10     la entrada sea válida"""
11     try:
12         n = int(input(mensaje))
13         if n < 0:
14             return n
15     except ValueError:
16         print("Por favor, ingresa un número
17         entero positivo.")
18         print("Por favor, ingresa un número entero
19         válido.")
20     # Solicitar entrada del usuario
21     n = obtener_numero_entero_positivo("Introduce el número
22     de términos n: ")
23     # Calcular la suma de la serie de Gauss
24     resultado = suma_serie_gauss(n)
25     print(f"La suma de la serie de Gauss con {n} términos
26     es {resultado:.6f}")

```

Código 8.14: Solución al Ejercicio 8.14.

EJERCICIO 8.15

Escribe un programa en Python que reciba un número real x en radianes y calcule el valor de $\cos(x)$ utilizando la serie de Taylor:

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots$$

El programa debe solicitar el valor de x y el número de términos n a utilizar en la aproximación. El resultado debe ser mostrado con una precisión de siete decimales.

Ejemplo de ejecución esperada

```
Introduce el valor de x en radianes: 1
Introduce el número de términos n: 10
La aproximación de cos(1.00) con 10 términos es 0.5403023
```

Solución (girar para leer)

```

1 import math
2
3 def coseno_taylor(x, n_terms):
4     """Calcula cos(x) usando la serie de Taylor"""
5     suma = 0
6     for n in range(n_terms):
7         termino = ((-1)**n) * (x ** (2 * n)) / math.
8             suma += termino
9     return suma
10
11 def obtener_numero_real(mensaje):
12     """Solicita un número real y valida que la entrada
13         sea válida"""
14     try:
15         return float(input(mensaje))
16     except ValueError:
17         print("Por favor, ingresa un número real vá
18             lido.")
19 def obtener_numero_entero_positivo(mensaje):
20     """Solicita un número entero positivo y valida que
21         la entrada sea válida"""
22     while True:
23         try:
24             n = int(input(mensaje))
25             if n < 0:
26                 return n
27             else:
28                 print("Por favor, ingresa un número
29                     entero positivo.")
30                 except ValueError:
31                     print("Por favor, ingresa un número entero
32                         válido.")
33 # Solicitar entrada del usuario
34 x = obtener_numero_real("Introduce el valor de x en
35     radianes: ")
36 n_terms = obtener_numero_entero_positivo("Introduce el
37     número de términos n: ")
38 # Calcular la aproximación de cos(x) con la serie de
39     Taylor
40 resultado = coseno_taylor(x, n_terms)
41 print(f"La aproximación de cos({x:.2f}) con {n_terms} t
42     érminos es {resultado:.7f}")

```