

7

Enteros

En este capítulo exploraremos cómo manejar cálculos con números enteros en Python, resolviendo problemas comunes relacionados con la manipulación de cifras y la divisibilidad. Python proporciona una representación flexible de los enteros y evita los problemas de desbordamiento presentes en otros lenguajes de programación, lo que resulta especialmente útil cuando se trabaja con números muy grandes, como en cálculo científico, combinatoria o teoría de números.

7.1. Representación de números enteros en Python

En Python, los números enteros se representan con el tipo `int`, que tiene las siguientes características:

- No hay un límite de tamaño fijo: Python maneja enteros de **precisión arbitraria**. Esto significa que no se produce desbordamiento cuando se realizan operaciones matemáticas con números arbitrariamente grandes (el único límite es la memoria disponible).
- Los enteros se pueden escribir en base decimal, binaria (prefijo `0b`), octal (prefijo `0o`) o hexadecimal (prefijo `0x`).
- Soportan los operadores aritméticos estándar, como veremos en la siguiente sección.

Por ejemplo, el siguiente código define cuatro variables enteras escritas en distintas bases:

```
a = 2147483647    # Un entero en base 10
b = 0b1010        # Un entero en base 2 (binario)
c = 0o12           # Un entero en base 8 (octal)
d = 0xA           # Un entero en base 16 (hexadecimal)
```

```
print(a, b, c, d) # Salida: 2147483647 10 10 10
```

7.1.1. Conversiones entre bases

Aunque internamente todos los enteros se almacenan igual, es muy frecuente – sobre todo en programación de bajo nivel y manipulación de datos binarios – necesitar la representación textual de un entero en una base distinta. Python ofrece funciones específicas para ello:

- `bin(n)`: devuelve la representación binaria de `n` como cadena, prefijada con `0b`.
- `oct(n)`: devuelve la representación octal, prefijada con `0o`.
- `hex(n)`: devuelve la representación hexadecimal, prefijada con `0x`.
- `int(s, base)`: convierte una cadena `s` a entero interpretándola en la base indicada (entre 2 y 36).

Por ejemplo:

```
print(bin(255))          # Salida: 0b11111111
print(hex(255))          # Salida: 0xff
print(int("ff", 16))     # Salida: 255
print(int("1010", 2))    # Salida: 10
```

7.2. Operadores aritméticos

Python proporciona los operadores aritméticos habituales sobre enteros, recogidos en la Tabla 1.

Operador	Descripción	Ejemplo
+	Suma	$7 + 3 \rightarrow 10$
-	Resta	$7 - 3 \rightarrow 4$
*	Multipliación	$7 * 3 \rightarrow 21$
/	División real (<i>float</i>)	$7 / 3 \rightarrow 2.333\dots$
//	División entera	$7 // 3 \rightarrow 2$
%	Resto (módulo)	$7 \% 3 \rightarrow 1$
**	Exponenciación	$2 ** 10 \rightarrow 1024$

Tabla 1: Operadores aritméticos sobre enteros.

Cada uno de estos operadores tiene también su forma compuesta de asignación (`+=`, `-=`, `*=`, `//=`, etc.), de uso muy frecuente.

Además, Python ofrece dos funciones *built-in* muy útiles para trabajar con enteros:

- `divmod(a, b)`: devuelve la tupla `(a // b, a % b)` en una sola operación. Es más eficiente que calcular cociente y resto por separado cuando se necesitan ambos.
- `pow(base, exp)` y `pow(base, exp, mod)`: la primera forma es equivalente a `base ** exp`; la segunda calcula `(base ** exp) % mod` pero de manera mucho más eficiente. Esta segunda forma se denomina **exponenciación modular** y es fundamental en criptografía moderna (RSA, Diffie-Hellman).

```

print(divmod(17, 5))          # Salida: (3, 2)
print(pow(3, 4))              # Salida: 81
print(pow(3, 4, 5))           # Salida: 1    (porque 81 % 5 ==
    1)
print(pow(2, 1024, 1000003)) # Cálculo modular eficiente con
    exponente grande

```

Eficiencia de `pow(a, b, m)`

Calcular `(a ** b) % m` con `b` muy grande es prohibitivo: primero calcula la potencia entera completa (que puede tener millones de cifras) y luego le aplica el módulo. `pow(a, b, m)` usa *exponenciación rápida modular*, manteniendo en cada paso el resultado reducido módulo `m`. La diferencia de rendimiento es de varios órdenes de magnitud en cuanto los exponentes superan unos pocos miles de bits.

7.3. Desbordamientos

A diferencia de otros lenguajes donde los enteros tienen un tamaño fijo (como C++, donde están limitados a 4 u 8 bytes según la arquitectura), en Python no hay desbordamiento de enteros: el tipo `int` ajusta dinámicamente su tamaño. Los enteros en Python crecen automáticamente según la memoria disponible, por lo que se puede trabajar con números muy grandes sin preocuparse por el desbordamiento. Sin embargo, el rendimiento puede verse afectado cuando se manejan números extremadamente grandes.

Por tanto, errores debidos a problemas de precisión derivados de desbordamientos no van a aparecer en Python. Por ejemplo, el siguiente código:

```

1 def factorial(n):
2     if n < 0:
3         raise ValueError("El factorial no esta definido para
    negativos.")
4     result = 1
5     for i in range(1, n + 1):
6         result *= i
7     return result

```

calculará correctamente el factorial de cualquier número pasado por parámetro sin producir errores de desbordamiento (mientras haya memoria disponible). Por ejemplo, `factorial(1000)` devuelve un entero con 2.568 cifras y Python lo maneja sin problema.

7.4. El tipo `bool` y su relación con los enteros

En Python, el tipo `bool` es un *subtipo* de los enteros. Los valores `True` y `False` se representan internamente como los enteros 1 y 0, respectivamente. Esto significa que se pueden usar en contextos aritméticos directamente, lo cual a veces resulta cómodo (por ejemplo, para contar cuántas condiciones se cumplen). Por ejemplo:

```
print(int(True))      # Salida: 1
print(int(False))     # Salida: 0
print(True + True)    # Salida: 2
```

7.5. Bibliotecas numéricas: el módulo `math`

Python proporciona un conjunto amplio de funciones matemáticas en el módulo `math`. Algunas de las más utilizadas con enteros son:

- `math.factorial(n)`: calcula el factorial de un número.
- `math.gcd(a, b, ...)`: calcula el máximo común divisor de los argumentos.
- `math.lcm(a, b, ...)`: calcula el mínimo común múltiplo.
- `math.isqrt(n)`: calcula la parte entera de la raíz cuadrada (sin pasar por `float`, evitando errores de redondeo).
- `math.comb(n, k)` y `math.perm(n, k)`: combinaciones y permutaciones.

Por ejemplo:

```
import math
print(math.factorial(5))  # Salida: 120
print(math.gcd(48, 18))  # Salida: 6
print(math.isqrt(10**18)) # Sin perdida de precision
```

`math.sqrt` vs `math.isqrt`

`math.sqrt(n)` devuelve un `float` y, para enteros grandes, puede sufrir errores de redondeo (por ejemplo, devolver 9.999999999999998 en lugar de 10). Si solo necesitas la parte entera de la raíz, usa `math.isqrt(n)`: trabaja siempre en aritmética entera y es exacta.

7.6. Manipulación de números enteros

Los siguientes problemas se enfocan en cómo analizar y manipular los dígitos de un número entero cuando se expresa en base 10.

7.6.1. Contar el número de cifras de un entero

Para contar cuántas cifras tiene un número, se puede simplemente utilizar `len(str(n))` (transformamos el número a una cadena, y miramos cuál es la longitud de la cadena). Para ilustrar el proceso manual, implementamos el siguiente algoritmo:

```

1 def num_cifras(n):
2     """Devuelve el número de cifras de n en base 10."""
3     n = abs(n) # Convertimos a positivo si es negativo
4     cuenta = 1
5     while n >= 10:
6         n //= 10 # Eliminamos la última cifra
7         cuenta += 1
8     return cuenta
9
10 print(num_cifras(65030)) # Salida: 5
11 print(num_cifras(-103044)) # Salida: 6

```

El código define una función llamada `num_cifras` que tiene como objetivo contar el número de cifras que tiene un número entero `n` cuando se expresa en base 10.

En primer lugar, la función toma el valor absoluto del número (`n = abs(n)`) para asegurarse de que, si el número es negativo, se convierta en positivo, ya que el signo no afecta al número de cifras.

La variable `cuenta` se inicializa en 1 porque, al menos, el número tendrá una cifra. Luego, se utiliza un bucle `while` para dividir el número `n` por 10 repetidamente. En cada iteración, se elimina la última cifra usando el operador `//=`, que realiza una división entera y asigna el cociente de vuelta a `n`. Esto reduce el número por un factor de 10 en cada paso, eliminando la cifra menos significativa (la de más a la derecha).

En cada iteración, se incrementa la variable `cuenta` en 1 para llevar la cuenta de cuántas veces se ha dividido el número, lo que equivale a cuántas cifras tiene. El bucle continúa ejecutándose hasta que `n` sea menor que 10, lo que indica que solo queda la última cifra.

7.6.2. Suma de las cifras de un número entero

La suma de los dígitos de un número se puede calcular tal y como se muestra en el siguiente código:

```

1 def suma_cifras(n):
2     """Devuelve la suma de las cifras de n en base 10."""
3     n = abs(n)
4     suma = 0
5     while n > 0:
6         suma += n % 10 # Extraemos la última cifra y la
7             sumamos
8         n //= 10 # Eliminamos la última cifra
9     return suma
10 print(suma_cifras(65030)) # Salida: 14
11 print(suma_cifras(-103044)) # Salida: 12

```

El código define una función llamada `suma_cifras` que calcula la suma de las cifras de un número entero `n` cuando se expresa en base 10.

La función comienza tomando el valor absoluto del número (`n = abs(n)`), lo que garantiza que si `n` es negativo se convierte en positivo. Esto es importante porque el signo negativo no afecta a la suma de las cifras.

La variable `suma` se inicializa en 0 y acumulará la suma de las cifras del número.

El bucle `while n > 0` se ejecuta mientras `n` sea mayor que cero. Dentro del bucle, la operación `n % 10` extrae la última cifra del número: el operador `%` es el operador de módulo, y al hacer `n % 10` obtenemos el resto de dividir `n` entre 10, lo que equivale a la última cifra. Esa cifra extraída se acumula en `suma` con la instrucción `suma = suma + n % 10`.

Después de sumar la última cifra, el número `n` se divide por 10 con el operador de división entera y asignación `//=`. Esto elimina la última cifra. El bucle continúa ejecutándose hasta que `n` sea cero, es decir, hasta que no haya más cifras por procesar.

Una vez que el bucle termina, la función devuelve la variable `suma`, que contiene la suma total de las cifras del número original.

7.6.3. Obtener la i-ésima cifra menos significativa

Para extraer la *i*-ésima cifra menos significativa, eliminamos las cifras menos significativas hasta llegar a la deseada:

```

1 def cifra(n, i):
2     """Devuelve la i-esima cifra menos significativa de n."""
3     n = abs(n) # Trabajamos solo con valores positivos
4     for _ in range(i - 1):
5         n //= 10 # Eliminamos cifras
6     return n % 10 # Devolvemos la última cifra
7
8 print(cifra(6470431, 1)) # Salida: 1
9 print(cifra(6470431, 3)) # Salida: 4
10 print(cifra(-6470431, 5)) # Salida: 7

```

El código define una función llamada `cifra` que devuelve la i -ésima cifra menos significativa de un número entero n . La función recibe dos parámetros: n , que es el número del cual queremos extraer la cifra, e i , que es la posición de la cifra que queremos obtener, comenzando desde 1 (es decir, la primera cifra menos significativa corresponde a $i = 1$).

La primera línea dentro de la función es $n = \text{abs}(n)$, que convierte n en su valor absoluto. Esto asegura que trabajamos solo con valores positivos, ya que el signo de un número no afecta a sus cifras individuales.

El siguiente paso es eliminar las primeras $i - 1$ cifras del número. Esto se logra mediante el bucle `for _ in range(i - 1)`. En este caso, se usa el carácter `_` como variable de control, lo que indica que la variable no se va a utilizar dentro del bucle (es una convención en Python para indicar que el valor de la variable no es relevante y que solo se necesita ejecutar el bucle un número determinado de veces). En cada iteración del bucle, se realiza una división entera de n entre 10 (sentencia $n //= 10$), eliminando la última cifra. El bucle se repite $i - 1$ veces, eliminando así las primeras $i - 1$ cifras menos significativas.

Por ejemplo, si estamos buscando la tercera cifra menos significativa ($i = 3$), el bucle se ejecutará dos veces, eliminando las dos primeras cifras menos significativas del número. Después de que el bucle ha eliminado las primeras $i - 1$ cifras, lo que queda en n es un número cuyo último dígito es la cifra que queremos obtener. Para obtenerla, utilizamos el operador de módulo ($n \% 10$), que devuelve el resto de la división de n por 10, es decir, su último dígito. Esta cifra es la que se devuelve finalmente con `return n \% 10`.

7.6.4. Invertir un número (imagen especular)

Para invertir un número, podemos usar la siguiente función:

```

1 def imagen(n):
2     """Devuelve la imagen especular de n."""
3     negativo = n < 0
4     n = abs(n)
5     invertido = 0
6     while n > 0:
7         invertido = invertido * 10 + n \% 10 # Construimos el
8         n //= 10                               número invertido
9     return -invertido if negativo else invertido
10
11 print(imagen(123456))    # Salida: 654321
12 print(imagen(-103044))  # Salida: -440301

```

El código define una función llamada `imagen` que devuelve la imagen especular de un número entero n , es decir, el número obtenido al invertir el orden de sus cifras.

La primera línea de la función evalúa si el número n es negativo. Si lo es, la variable `negativo` se establecerá como `True`, en caso contrario como `False`. Esto es importan-

te para que, al final del proceso, podamos devolver la imagen especular con el signo adecuado.

Después se convierte `n` a su valor absoluto (`n = abs(n)`) para que el proceso de inversión no se vea afectado por el signo. A continuación, comienza el proceso de inversión: la variable `invertido` se inicializa a 0; a medida que vamos procesando las cifras del número, iremos construyendo la versión invertida en esta variable.

El bucle principal extrae la última cifra de `n` con el operador de módulo y la añade a `invertido`, que previamente se ha multiplicado por 10 para desplazar a la izquierda las cifras ya procesadas. Después se elimina la última cifra de `n` usando la división entera (`n //= 10`). El bucle se repite hasta que `n` sea 0, lo que significa que ya hemos procesado todas las cifras.

Una vez invertido el número, comprobamos si el número original era negativo. Si lo era, devolvemos el invertido con signo negativo; si era positivo, lo devolvemos tal cual. Concretamente, la última línea de la función,

```
1 | return -invertido if negativo else invertido
```

es una expresión condicional (también conocida como *expresión ternaria*) que permite devolver uno de dos valores dependiendo de la condición que se evalúe. El mismo comportamiento podría lograrse con una sentencia `if` tradicional:

```
1 | if negativo:
2 |     return -invertido
3 | else:
4 |     return invertido
```

La expresión ternaria es simplemente una forma más concisa de lograr lo mismo.

7.6.5. Comprobar la primalidad de un número

Un número es primo si es mayor o igual a 2 y no es divisible más que por 1 y por sí mismo. Para mejorar la eficiencia, verificamos divisores hasta la raíz cuadrada del número:

```
1 | import math
2 |
```

```

3 def es_primo(n):
4     """Devuelve True si n es un número primo, False en caso
↪     contrario."""
5     if n < 2:
6         return False
7     if n == 2:
8         return True
9     if n % 2 == 0:
10        return False
11    for divisor in range(3, math.isqrt(n) + 1, 2):
12        if n % divisor == 0:
13            return False
14    return True
15
16 print(es_primo(29))    # Salida: True
17 print(es_primo(30))    # Salida: False

```

El código define una función llamada `es_primo` que determina si un número `n`, dado por parámetro, es primo o no. La función comienza verificando si `n` es menor que 2. Si es así, devuelve `False` porque no existen números primos menores que 2 (por convención, el 1 no se considera primo).

A continuación aparece el caso especial del número 2: el único par que es primo. Si `n` es 2, la función devuelve `True`.

Si no se cumple ninguna de las condiciones anteriores, la función comprueba si `n` es divisible entre 2, lo que significa que `n` es un número par. Todos los pares, excepto el 2, no son primos; por lo tanto, si `n` es divisible entre 2, la función devuelve `False`.

Si el número no es menor que 2, no es 2, y no es divisible entre 2, entonces verificamos si tiene divisores impares. Para hacer esto, la función utiliza un bucle `for` que comienza desde 3 y va hasta la raíz cuadrada de `n`. Esto se debe a que si un número `n` tiene un divisor mayor que su raíz cuadrada, también tendrá uno menor, por lo que no es necesario comprobar divisores más grandes que la raíz cuadrada. Además, solo verificamos divisores impares, ya que hemos descartado los pares (excepto el 2): por eso el paso es 2 en la función `range`.

Hemos utilizado `math.isqrt(n)` para obtener la parte entera exacta de la raíz cuadrada de `n`, evitando así los errores de redondeo en coma flotante que tendríamos con `int(math.sqrt(n))` para enteros muy grandes. Se verifica la divisibilidad con `n % divisor == 0`. Si en algún momento encontramos que `n` es divisible por alguno de los divisores impares, la función devuelve `False`, indicando que `n` no es primo. Si no se encuentran divisores durante la iteración (lo que significa que `n` no es divisible por ningún número distinto de 1 y de sí mismo), la función devuelve `True`.

7.6.6. Cálculo del máximo común divisor

El máximo común divisor (MCD) de dos números se puede calcular de forma eficiente usando el **algoritmo de Euclides**. Este algoritmo se basa en una propiedad fundamental de los números enteros: el MCD de dos números a y b es el mismo que el MCD de b y el resto de a dividido por b , es decir, $MCD(a, b) = MCD(b, a \bmod b)$. Este proceso se repite hasta que el resto sea cero, momento en el cual el MCD será el último valor distinto de cero. El algoritmo es eficiente porque, a medida que se calculan los restos, los números involucrados se hacen más pequeños, lo que reduce significativamente la cantidad de iteraciones necesarias.

El siguiente código implementa este algoritmo:

```
1 def mcd(a, b):
2     """Devuelve el máximo común divisor de a y b usando el
   ↪     algoritmo de Euclides."""
3     a, b = abs(a), abs(b) # Trabajamos con valores absolutos
4     while b:
5         a, b = b, a % b # Aplicamos iterativamente el
   ↪         algoritmo de Euclides
6     return a
7
8 print(mcd(48, 18))          # Salida: 6
9 print(mcd(-103044, 65030)) # Salida: 2
```

En la primera línea de la función, los números a y b se convierten a sus valores absolutos utilizando la función `abs()`. Esto se hace porque el MCD es el mismo si los números son negativos o positivos, y el algoritmo de Euclides funciona sin importar el signo.

El bucle `while b` continúa ejecutándose mientras b no sea 0. En cada iteración, el valor de a se actualiza a b , y el valor de b se actualiza al resto de $a \% b$. Este proceso se repite hasta que b se convierte en 0, momento en el que termina el bucle. El valor final de a es el MCD de los dos números originales.

El algoritmo de Euclides es la base sobre la que se construye el *algoritmo de Euclides extendido*, que además de calcular $MCD(a, b)$ encuentra los coeficientes de Bézout (x, y tales que $ax + by = MCD(a, b)$). Esta extensión se utiliza, por ejemplo, para calcular el inverso modular, una operación que aparece en numerosos algoritmos numéricos.

7.7. Aplicación: representación binaria de enteros

La precisión arbitraria de los enteros en Python, combinada con sus operaciones de conversión a bytes, hace que el lenguaje sea especialmente cómodo para trabajar con representaciones binarias: serializar y deserializar enteros, interpretar cabeceras de ficheros o de protocolos, o manipular datos a bajo nivel. En esta sección recogemos algunas funcionalidades del lenguaje que aparecen de forma recurrente en ese tipo de tareas.

7.7.1. Bits, bytes y enteros: `bit_length`, `to_bytes`, `from_bytes`

Todo entero en Python lleva asociadas tres operaciones muy útiles para trabajar con representaciones binarias:

- `n.bit_length()`: devuelve el número mínimo de bits necesarios para representar `n` en binario, sin contar el signo ni los ceros a la izquierda.
- `n.to_bytes(longitud, byteorder)`: convierte el entero `n` en una secuencia de `longitud` bytes, en el orden indicado (`'big'` o `'little'`).
- `int.from_bytes(secuencia, byteorder)`: la operación inversa, reconstruye el entero a partir de los bytes.

Por ejemplo, para consultar cuántos bits ocupa un número y serializarlo como una secuencia de bytes:

```
clave = 0xDEADBEEFCAFEBABE
print(clave.bit_length())           # Salida: 64

paquete = clave.to_bytes(8, 'big') # b'\xde\xad\xbe\xef\xca\xba\xfe\xbe'
print(int.from_bytes(paquete, 'big') == clave) # Salida: True
```

Este es el mecanismo natural para serializar y deserializar enteros cuando interpretamos cabeceras de protocolos, ficheros binarios o cualquier formato a bajo nivel.

7.8. Ejercicios

EJERCICIO 7.1

Escribe una función en Python que reciba un número entero n y determine el número total de divisores de n (incluyendo el número n mismo). El programa debe ser eficiente y calcular los divisores sólo hasta la raíz cuadrada de n para optimizar el tiempo de ejecución.

Ejemplo de ejecución esperada

Introduce un número: 36
El número 36 tiene 9 divisores

```

1 import math
2
3 def numero_divisores(n):
4     """Devuelve el número total de divisores de n
5     incluyendo n mismo)."""
6     count = 0
7     # Iteramos solo hasta la raíz cuadrada de n
8     for i in range(1, int(math.sqrt(n)) + 1):
9         if n % i == 0:
10             # Si i es divisor de n
11             count += 1
12             # Contamos el divisor
13             # Si i no es la raíz
14             # cuadrada de n
15             count += 1
16         # Contamos el divisor
17         //
18     return count
19
20 # Interacción con el usuario
21 n = int(input("Introduce un número: "))
22 print(f"El número {n} tiene {numero_divisores(n)} divisores")

```

Código 7.1: Solución al Ejercicio 7.1.

EJERCICIO 7.2

Escribe una función en Python que reciba un número entero n y calcule la suma de los primeros n números primos. El programa debe ser eficiente en cuanto al tiempo de ejecución y manejar correctamente valores grandes de n . Al final, debe mostrar el resultado de la suma.

Ejemplo de ejecución esperada

Introduce un número: 5

La suma de los primeros 5 números primos es 28

Solución (girar para leer)

```

1  def es-primo(num):
2      """Devuelve True si el número es primo, False en
3      caso contrario."""
4      if num < 2:
5          return False
6      for i in range(2, int(num**0.5) + 1):
7          # comprobamos si el número es divisible por algún
8          # número hasta su raíz cuadrada
9          if num % i == 0:
10             return False
11         return True
12     def suma-primos(n):
13         """Devuelve la suma de los primeros n números
14         primos."""
15         suma = 0
16         num = 2
17         count = 0
18         while count < n:
19             if es-primo(num):
20                 # Usamos la función es-primo
21                 # para verificar si num es primo
22                 suma += num
23                 count += 1
24             num += 1
25         return suma
26     # Interacción con el usuario
27     n = int(input("Introduce un número: "))
28     print(f"La suma de los primeros {n} números primos es {
29     suma-primos(n)}")

```

Código 7.2: Solución al Ejercicio 7.2.

EJERCICIO 7.3

Escribe una función en Python que reciba un número entero n y determine si n es un cuadrado perfecto. Un número es un cuadrado perfecto si puede expresarse como el cuadrado de un número entero. Es decir, existe un número entero m tal que $n = m^2$.

———— Ejemplo de ejecución esperada ————

Introduce un número: 25

El número 25 es un cuadrado perfecto.

Solución (girar para leer)

```

1 import math
2
3 def cuadrado-perfecto(n):
4     """Devuelve True si n es un cuadrado perfecto,
5     False en caso contrario."""
6     raiz = int(math.sqrt(n))
7     return raiz * raiz == n
8
9 # Interacción con el usuario
10 n = int(input("Introduce un número: "))
11 if cuadrado-perfecto(n):
12     print(f"El número {n} es un cuadrado perfecto.")
13 else:
14     print(f"El número {n} no es un cuadrado perfecto.")

```

Código 7.3: Solución al Ejercicio 7.3.

EJERCICIO 7.4

Escribe una función en Python que reciba un número entero n y una base b , y calcule el número de dígitos de n cuando se representa en la base b . La función debe devolver la cantidad de dígitos en la representación de n en base b .

Ejemplo de ejecución esperada

Introduce un número: 255

Introduce la base: 16

El número 255 en base 16 tiene 2 dígitos.

Solución (girar para leer)

```

1 def numero_digitos_base(n, b):
2     """Devuelve el número de dígitos de n en base b."""
3     count = 0
4     while n > 0:
5         n //= b # Dividimos el número entre la base b
6         count += 1 # Incrementamos el contador de dígitos
7     return count if count > 0 else 1 # Si count es 0,
8     el número tiene 1 dígito (para el caso n=0)
9 # Interacción con el usuario
10 n = int(input("Introduce un número: "))
11 b = int(input("Introduce la base: "))
12 print(f"El número {n} en base {b} tiene {numero_digitos_base(n, b)} dígitos.")

```

Código 7.4: Solución al Ejercicio 7.4.

EJERCICIO 7.5

Escribe una función en Python que reciba un número entero n y determine si n es un número triangular. Un número triangular es aquel que puede representarse como la suma de los primeros k números naturales, es decir, $n = 1 + 2 + 3 + \dots + k$. El k -ésimo número triangular se puede calcular como $n = k(k + 1)/2$.

———— Ejemplo de ejecución esperada ————

Introduce un número: 10

El número 10 es triangular.

Solución (girar para leer)

Código 7.5: Solución al Ejercicio 7.5.

```

1  def es_triangular(n):
2      """Devuelve True si n es un número triangular,
3      False en caso contrario."""
4      suma = 0
5      while suma < n:
6          suma += k
7          k += 1
8      return (suma == n)
9
10 # Interacción con el usuario
11 n = int(input("Introduce un número: "))
12 if es_triangular(n):
13     print(f"El número {n} es triangular.")
14 else:
15     print(f"El número {n} no es triangular.")

```

EJERCICIO 7.6

Escribe un programa en Python que reciba dos números enteros a y b y determine si son **números amistosos**. Dos números son amistosos si la suma de los divisores propios de a es igual a b y la suma de los divisores propios de b es igual a a . Por ejemplo, los números 220 y 284 son números amistosos, porque la suma de los divisores de 220 (excluyendo 220) es 284, y la suma de los divisores de 284 (excluyendo 284) es 220.

Ejemplo de ejecución esperada

Introduce el primer número: 220
 Introduce el segundo número: 284
 Los números 220 y 284 son amistosos.

Solución (girar para leer)

```

1  def suma_divisores(n):
2      """Devuelve la suma de los divisores propios de n (
3      excluyendo n mismo)."""
4      suma = 0
5      for i in range(1, int(n**0.5) + 1):
6          if n % i == 0:
7              suma += i
8              if i != n // i:
9                  suma += n // i
10         return suma
11     def son_amistosos(a, b):
12         """Devuelve True si a y b son números amistosos,
13         False en caso contrario."""
14         return suma_divisores(a) == b and suma_divisores(b) == a
15     # Interacción con el usuario
16     a = int(input("Introduce el primer número: "))
17     b = int(input("Introduce el segundo número: "))
18     if son_amistosos(a, b):
19         print(f"Los números {a} y {b} son amistosos.")
20     else:
21         print(f"Los números {a} y {b} no son amistosos.")

```

Código 7.6: Solución al Ejercicio 7.6.

EJERCICIO 7.7

Escribe un programa en Python que reciba un número entero n y determine si n es un número práctico. Un número práctico es un número entero positivo tal que todos los números menores que él son la suma de un subconjunto de sus divisores propios. Por ejemplo, el número 6 es práctico, porque sus divisores son 1, 2 y 3, y todos los números menores que 6 pueden formarse sumando los divisores de 6. Los números prácticos son una generalización de los números perfectos.

Ejemplo de ejecución esperada

Introduce un número: 6
El número 6 es práctico.

Solución (girar para leer)

```

1  def es_practico(n):
2      """Devuelve True si n es un número práctico, False
3      en caso contrario."""
4      suma = 0
5      for i in range(1, n):
6          if n % i == 0:
7              suma += i
8          return suma >= n
9      # Interacción con el usuario
10     n = int(input("Introduce un número: "))
11     if es_practico(n):
12         print(f"El número {n} es práctico.")
13     else:
14         print(f"El número {n} no es práctico.")

```

Código 7.7: Solución al Ejercicio 7.7.

EJERCICIO 7.8

Escribe un programa en Python que reciba una lista de números enteros y calcule el máximo común divisor de todos ellos utilizando el algoritmo de Euclides.

Ejemplo de ejecución esperada

Introduce los números (separados por espacios): 48 180 36
El máximo común divisor de los números es 12

Solución (girar para leer)

```

1 import math
2
3 def mcd(a, b):
4     """Devuelve el máximo común divisor de a y b usando
5     el algoritmo de Euclides."""
6     while b:
7         a, b = b, a % b
8     return a
9
10 # Interacción con el usuario
11 entrada = input("Introduce los números separados por
12 espacios: ")
13 mcd_resultado = mcd(int(entrada[0]), # El primer número se
14 asigna a mcd_resultado
15 for num in entrada[1:]:
16     mcd_resultado = mcd(mcd_resultado, int(num))
17     # Calculamos el MCD con cada número posterior
18
19 print(f"El máximo común divisor de los números es {
20 mcd_resultado}")

```

Código 7.8: Solución al Ejercicio 7.8.

EJERCICIO 7.9

Escribe una función en Python que reciba un número entero n y calcule el n -ésimo número de Fibonacci de manera eficiente utilizando programación dinámica^a.

Ejemplo de ejecución esperada

Introduce un número: 100

El 100-ésimo número de Fibonacci es 354224848179261915075

^aLa programación dinámica es una técnica de optimización de algoritmos que se utiliza para resolver problemas que pueden ser descompuestos en subproblemas solapados. Es especialmente útil para problemas donde el mismo subproblema se resuelve varias veces durante la ejecución de un algoritmo. La idea clave en programación dinámica es almacenar los resultados de los subproblemas para evitar volver a calcularlos, lo que mejora la eficiencia del algoritmo.

Solución (girar para leer)

```

1 def fibonacci_grande(n):
2     """Devuelve el n-ésimo número de Fibonacci usando
3     programación dinámica (tabulación)."""
4     if n <= 1:
5         return n
6     a, b = 0, 1
7     for _ in range(2, n + 1):
8         a, b = b, a + b
9     return b
10 # Interacción con el usuario
11 n = int(input("Introduce un número: "))
12 print(f"El {n}-ésimo número de Fibonacci es {
13     fibonacci_grande(n)}")

```

Código 7.9: Solución al Ejercicio 7.9.

EJERCICIO 7.10

Escribe un programa en Python que reciba un número entero n y calcule su factorización prima, es decir, descomponga n en los factores primos que lo multiplican.

Ejemplo de ejecución esperada

Introduce un número: 60

La factorización prima de 60 es: 2, 2, 3, 5

Solución (girar para leer)

```

1 def factorizacion-prima(n):
2     """Imprime la factorización prima de n."""
3     # Comprobamos los divisores para los números
4     menores que n
5     # Comenzamos con el número 2
6     while n % 2 == 0:
7         print(2, end=" ", " ")
8         n //= 2
9     # Ahora verificamos los números impares desde 3 en
10    adelante
11    divisor = 3
12    while divisor * divisor <= n:
13        while n % divisor == 0:
14            print(divisor, end=" ", " ")
15            n //= divisor
16        divisor += 2
17    # Si n es mayor que 2, entonces n es un número
18    primo
19    if n > 2:
20        print(n)
21    # Interacción con el usuario
22    print(f"La factorización prima de {n} es: ", end=" ")
    factorizacion-prima(n)

```

Código 7.10: Solución al Ejercicio 7.10.