04 y 05 - Loops over Strings, Guess-and-Check, Binary & Floats and Approximation Methods

Enumeración exhaustiva

El siguiente código imprime la raíz cúbica entera, si existe, de un número entero. Si la entrada no es un cubo perfecto, imprime un mensaje a tal efecto. El operador `!=` significa no igual.

```
x = int(input("Ingresa un entero: "))
ans = 0
while ans ** 3 < abs(x):
    ans = ans + 1
if ans ** 3 != abs(x):
    print(x, "No es un cubo perfecto")
else:
    if x < 0:
        ans = -ans
    print("La raiz cubica de", x, "es", ans)
```

El código intenta primero establecer la variable `ans` en la raíz cúbica del valor absoluto de `x`. Si lo consigue, entonces establece `ans` en `-ans` si `x` es negativo. El trabajo pesado (tal como es) en este código se realiza en el bucle `while`. Cada vez que un programa contiene un bucle, es importante *entender lo que hace que el programa finalmente salga de este bucle*. ¿Para qué valores de `x` terminará este bucle `while`? La respuesta es «*todos los enteros*». Esto se puede argumentar de forma bastante sencilla.

- El valor de la expresión `ans ** 3` empieza en 0 y aumenta cada vez que se pasa por el bucle.
- Cuando alcanza o supera `abs(x)`, el bucle termina.
- Dado que `abs(x)` es siempre positivo, sólo hay un número finito de iteraciones antes de que el bucle deba terminar.

Este argumento se basa en la noción de *función decreciente*. Se trata de una función que tiene las siguientes propiedades:

- Asigna un conjunto de variables del programa a un número entero.
- Cuando se introduce en el bucle, su valor es no negativo.
- Cuando su valor es `≤ 0`, el bucle termina.
- Su valor disminuye cada vez que se pasa por el bucle.

¿Cuál es la función de decremento para el bucle `while`? Es `abs(x) - ans ** 3`.
Ahora, introduzcamos algunos errores y veamos qué ocurre. Primero, intenta comentar la sentencia `ans = 0`. El intérprete de Python imprime el mensaje de error:

```
NameError: name 'ans' is not defined
```

Porque el intérprete intenta encontrar el valor al que `ans` está ligado antes de que haya sido ligado a nada. Ahora, restaure la inicialización de `ans`, reemplace la sentencia `ans = ans + 1` por `ans = ans`, e intente encontrar la raíz cúbica de 8. Cuando se canse de esperar, introduzca «`Ctrl + C`» (mantenga pulsada la tecla `Ctrl` y la tecla `c` simultáneamente). Esto te devolverá al *prompt* de usuario en el shell.

Ahora, añade la sentencia

```
print('Value of the decrementing function abs(x) - ans ** 3 is', abs(x) - ans ** 3)
```

al principio del bucle, e intente ejecutarlo de nuevo. Esta vez se imprimirá

```
Value of the decrementing function abs(x) - ans ** 3 is 8
```

una y otra vez.

El programa se habría ejecutado eternamente porque el cuerpo del bucle ya no está reduciendo la distancia entre `ans ** 3` y `abs(x)`. Cuando nos enfrentamos a un programa que parece no terminar, los programadores experimentados a menudo insertan sentencias `print`, como la de aquí, para comprobar si la función decrementadora está siendo realmente decrementada.

La técnica algorítmica utilizada en este programa es una variante de guess and check llamada *enumeración exhaustiva*. Enumeramos todas las posibilidades hasta que llegamos a la respuesta correcta o agotamos el espacio de posibilidades. A primera vista, puede parecer una forma increíblemente estúpida de resolver un problema. Sin embargo, sorprendentemente, los algoritmos de enumeración exhaustiva suelen ser la forma más práctica de resolver un problema. Suelen ser fáciles de implementar y de entender. Y, en muchos casos, se ejecutan lo suficientemente rápido a efectos prácticos.

Elimine o comente la sentencia `print` que insertó para depurar y vuelva a insertar la sentencia `ans = ans + 1`. Ahora intente encontrar la raíz cúbica de `1957816251`. El programa terminará casi instantáneamente. Ahora, prueba `7406961012236344616`.

Como puede ver, aunque se necesiten millones de conjeturas, el tiempo de ejecución no suele ser un problema. *Los ordenadores modernos son asombrosamente rápidos*. Tardan menos de un nanosegundo -la milmillonésima parte de un segundo- en ejecutar una instrucción. Es difícil apreciar lo rápido que es eso. Por ejemplo, la luz tarda algo más de un nanosegundo en recorrer un pie (0,3 metros). Otra forma de verlo es que un ordenador moderno puede ejecutar millones de instrucciones en el tiempo que tarda el sonido de tu voz en recorrer 30 metros.

Sólo por diversión, intente ejecutar el código

```
max_val = int(input('Enter a postive integer: '))
i = 0
while i < max_val:
    i = i + 1
print(i)
```

Vea qué tamaño de número entero necesita introducir antes de que se produzca una pausa perceptible antes de que se imprima el resultado.

Veamos otro ejemplo de enumeración exhaustiva: comprobar si un número entero es primo y devolver el divisor más pequeño si no lo es. Un número primo es un número entero mayor que 1 que sólo es divisible por sí mismo y por 1. Por ejemplo, 2, 3, 5 y 111.119 son primos, y 4, 6, 8 y 62.710.561 no lo son.

La forma más sencilla de averiguar si un número entero, `x`, mayor que 3 es primo, es dividir `x` por cada número entero comprendido entre `2` y, `x - 1`. Si el resto de cualquiera de esos números está entre `2` y, `x - 1`, **es primo**. Si el resto de cualquiera de esas divisiones es 0, `x` **no es primo**, de lo contrario `x` **es primo**.

```
x = int(input("Ingresa un entero mayor que 2: "))
smallest_divisor = None
for guess in range(2, x):
    if x % guess == 0:
        smallest_divisor = guess
        break
if smallest_divisor != None:
    print("El divisor mas pequeño de", x, "es", smallest_divisor)
else:
    print(x, "es un numero primo")
```

Primero pide al usuario que introduzca un entero, convierte la cadena devuelta a un `int` y asigna ese entero a la variable `x`. A continuación, establece las condiciones iniciales para una enumeración exhaustiva inicializando `guess` a `2` y la variable `smallest_divisor` a `None`, *indicando que hasta que se demuestre lo contrario, el código asume que x es primo*.
La enumeración exhaustiva se realiza dentro de un bucle `for`. El bucle termina cuando se han probado todos los posibles divisores enteros de `x` o se ha descubierto un entero que es divisor de `x`.

Después de salir del bucle, el código comprueba el valor de `smallest_divisor` e imprime el texto apropiado. El truco de inicializar una variable antes de entrar en un bucle, y luego comprobar si ese valor ha cambiado al salir, es muy común.

Ejercicio manual

Cambia el codigo para que devuelva el divisor mayor en lugar del menor.

```
x = int(input("Ingresa un entero mayor que 2: "))
smallest_divisor = None
    for guess in range(2, x):
        if x % guess == 0:
            smallest_divisor = guess
            break
    if smallest_divisor != None:
        print("El divisor mas pequeño de", x, "es", smallest_divisor)
    else:
        print(x, "es un numero primo")
```

Pista: si `y * z = x` e `y` es el menor divisor de `x`, `z` es el mayor divisor de `x`.

Mi aproximación al ejercicio:

```
x = int(input("Ingresa un entero mayor que 2: "))
biggest_divisor = None

for guess in range(2, x):
    if x % guess == 0:
        biggest_divisor = guess
if biggest_divisor != None:
    print("El divisor mas grande de", x, "es", biggest_divisor)
else:
    print(x, "es un numero primo")
```

Si el menor divisor propio es el primer numero que da resto cero, el ultimo en el bucle será el mayor.

El código del ejemplo anterior funciona, pero es innecesariamente ineficiente. Por ejemplo, no hay necesidad de comprobar los números pares más allá de 2, ya que si un número entero es divisible por cualquier número par, es divisible por 2. El código siguiente se aprovecha de este hecho comprobando primero si x es un número par. Si no lo es, utiliza un bucle para comprobar si x es divisible por algún número impar.

```
# Test if an int > 2 is prime. If not, print smallest divisor

x = int(input("Ingresa un entero mayor que 2: "))

if x % 2 == 0:
    smallest_divisor = 2
else:
    for guess in range(3, x, 2):
        if x % guess == 0:
            smallest_divisor = guess
            break

if smallest_divisor != None:
    print("El divisor mas grande de", x, "es", smallest_divisor)
else:
    print(x, "es un numero primo")
```

Aunque el código es ligeramente más complejo que el anterior, es considerablemente más rápido, ya que *se comprueban la mitad de números dentro del bucle*. La oportunidad de intercambiar complejidad de código por eficiencia en tiempo de ejecución es un fenómeno común. Pero **más rápido no siempre significa mejor**. Hay mucho que decir a favor de un código sencillo que sea obviamente correcto y lo suficientemente rápido como para ser útil.

Ejercicio manual

Escribe un programa que pida al usuario que introduzca un entero e imprima dos enteros, raíz y pwr, tales que `1 < pwr < 6` y `raíz ** pwr` sea igual al entero introducido por el usuario. Si no existe tal par de enteros, debe imprimir un mensaje a tal efecto.

Mi aproximación al ejercicio:

```python
num = int(input("Introduce un número entero: "))
encontrado = False

for pwr in range(2, 6):
    raiz = round(num ** (1 / pwr))
    if raiz ** pwr == num:
        print(f"{raiz} ** {pwr} = {num}")
        encontrado = True
        break

if encontrado == False:
    print("No existe una combinación de raíz y pwr tal que raíz ** pwr sea igual al número ingresado.")
```

Inicializo una variable que preguntara al usuario por un numero. También inicializo una variable booleana para saber si se encontró o no la combinación que necesito.

Mediante un bucle for recorro `pwr` tal que `1 < pwr < 6`, calculo la raíz `num ** (1 / pwr)` redondeándola porque me interesan las raíces enteras, si la raíz entera elevada a `pwr` me da como resultado el numero ingresado entonces imprimo y mi variable booleana será verdadera, de lo contrario imprimo que no existe tal combinación.

Soluciones aproximadas y búsqueda por bisección

Imagina que alguien te pide que escribas un programa que imprima la raíz cuadrada de cualquier número no negativo. ¿Qué debería hacer? Probablemente deberías empezar diciendo que necesitas un mejor enunciado del problema. Por ejemplo, ¿Qué debería hacer el programa si se le pide que encuentre la raíz cuadrada de 2? La raíz cuadrada de 2 no es un número racional. Esto significa que no hay manera de representar con precisión su valor como una cadena finita de dígitos (o como un flotador), por lo que el problema tal como se planteó inicialmente no puede ser resuelto.

Lo que sí puede hacer un programa es encontrar una aproximación a la raíz cuadrada, es decir, una respuesta que se acerque lo suficiente a la raíz cuadrada real como para ser útil. Pero por ahora, vamos a pensar en «lo suficientemente cerca» como una respuesta que se encuentra dentro de alguna constante, llámese épsilon, de la respuesta real.

El siguiente código implementa un algoritmo que imprime una aproximación a la raíz cuadrada de x.

```
epsilon = 0
step = epsilon ** 2
num_guesses = 0
ans = 0.0

while abs(ans ** 2 - x) >= epsilon and ans <= x:
    ans += step
    num_guesses += 1

print("Num of guesses =", num_guesses)

if abs(ans ** 2 - x) >= epsilon:
    print("Failed on square root of", x)
else:
    print(ans, "is close to square root of", x)
```

Una vez más, utilizamos la *enumeración exhaustiva*. Observa que este método para hallar la raíz cuadrada no tiene nada en común con la forma de hallar raíces cuadradas utilizando un lápiz que podrías haber aprendido en la escuela media. A menudo ocurre que la mejor manera de resolver un problema con un ordenador es bastante diferente de cómo se abordaría el problema a mano.

Si x es 25, el código imprimirá

```
number of guesses = 49990
4.999000000001688 is close to square root of 25
```

¿Debería decepcionarnos que el programa no se diera cuenta de que 25 es un cuadrado perfecto e imprimiera 5? No. El programa ha hecho lo que debía hacer. Aunque hubiera estado bien imprimir 5, hacerlo no es mejor que imprimir cualquier valor lo suficientemente cercano a 5. ¿Qué crees que pasará si ponemos `x = 0.25`? ¿Encontrará una raíz cercana a 0,5? No. Por desgracia, informará

```
number of guesses = 2501
Failed on square root of 0.25
```

La *enumeración exhaustiva es una técnica de búsqueda que sólo funciona si el conjunto de valores buscados incluye la respuesta*. En este caso, estamos enumerando los valores entre 0 y el valor de `x`. Cuando `x` está entre 0 y 1, la raíz cuadrada de x no se encuentra en este intervalo. Una forma de arreglar esto es cambiar el segundo operando de `y` en la primera línea del bucle `while` para obtener

```
```

```
while abs(ans ** 2 - x) >= epsilon and ans * ans <= x:
```

Cuando ejecutamos nuestro código después de este cambio, nos informa de que

```
0.48989999999996237 is close to square root of 0.25
```

Ahora, pensemos en cuánto tardará en ejecutarse el programa. El número de iteraciones depende de lo cerca que esté la respuesta de nuestro punto de partida, 0, y del tamaño de los pasos. A grandes rasgos, el programa ejecutará el bucle `while` como máximo `x/paso` veces. Probemos el código con algo más grande, por ejemplo, `x = 123456`. Se ejecutará durante un buen rato, y luego imprimirá

```
number of guesses = 3513631
Failed on square root of 123456
```

¿Qué crees que ha pasado? Seguramente existe un número en coma flotante que se aproxima a la raíz cuadrada de 123456 con una precisión de 0,01.

¿Por qué no lo ha encontrado nuestro programa? El problema es que nuestro tamaño de paso era demasiado grande, y el programa se saltó todas las respuestas adecuadas. Una vez más, estamos buscando exhaustivamente en un espacio que no contiene una solución. Intente hacer el paso igual a `epsilon ** 3` y ejecute el programa. Al final encontrará una respuesta adecuada, pero puede que no tengas paciencia para esperar a que lo haga.

Aproximadamente, ¿Cuántas conjeturas tendrá que hacer? El tamaño del paso será de 0,000001 y la raíz cuadrada de 123456 es de 351,36 aproximadamente. Esto significa que el programa tendrá que hacer alrededor de 351.000.000 conjeturas para encontrar una respuesta satisfactoria. Podríamos intentar acelerarlo empezando más cerca de la respuesta, pero eso supone que conocemos la vecindad de la respuesta.

Ha llegado el momento de buscar una forma diferente de atacar el problema. Tenemos que elegir un algoritmo mejor en lugar de ajustar el actual. Pero antes de hacerlo, veamos un problema que, a primera vista, parece completamente distinto de la búsqueda de raíces.

Consideremos el problema de descubrir si una palabra que empieza por una determinada secuencia de letras aparece en un diccionario impreso. En principio, la enumeración exhaustiva funcionaría. Se podría empezar por la primera palabra y examinar cada una de ellas hasta que se

encontrara una palabra que empezara por la secuencia de letras o hasta que se agotaran las palabras por examinar. Si el diccionario contuviera n palabras, se necesitaría una media de `n/2` sondeos para encontrar la palabra. Si la palabra no estuviera en el diccionario, se necesitarían n pruebas. Por supuesto, quienes han tenido el placer de buscar una palabra en un diccionario físico (y no en línea) nunca procederían de este modo.

Afortunadamente, quienes publican diccionarios en papel se toman la molestia de ordenar las palabras por orden lexicográfico. Esto nos permite abrir el libro por la página en la que creemos que se encuentra la palabra (por ejemplo, cerca de la mitad para las palabras que empiezan por la letra m). Si la secuencia de letras precede lexicográficamente a la primera palabra de la página, sabemos que debemos retroceder. Si la secuencia de letras sigue a la última palabra de la página, sabemos que debemos avanzar. En caso contrario, comprobamos si la secuencia de letras coincide con una palabra de la página.

Tomemos la misma idea y apliquémosla al problema de encontrar la raíz cuadrada de x. Supongamos que sabemos que una buena aproximación a la raíz cuadrada de x se encuentra en algún punto entre 0 y `max`. Podemos aprovechar el hecho de que los números están totalmente ordenados.

Es decir, para cualquier par de números distintos, `n1` y `n2`, o bien `n1 < n2` o bien `n1 > n2`. Por tanto, podemos pensar que la raíz cuadrada de x se encuentra en algún punto de la recta.

```
0_____________________________________________________max
```

y empezar a buscar en ese intervalo. Como no sabemos necesariamente dónde empezar a buscar, empecemos por el medio.

```
0____________________guess____________________max
```

Si no es la respuesta correcta (y no lo será la mayoría de las veces), *pregúntate si es demasiado grande o demasiado pequeña*. Si es demasiado grande, sabemos que la respuesta debe estar a la izquierda. Si es demasiado pequeño, sabemos que la respuesta debe estar a la derecha. A continuación, repetimos el proceso en el intervalo más pequeño.

```
epsilon = 0.01
num_guesses, low = 0, 0
high = max(1, x)
ans = (high + low) / 2

```

```
while abs(ans ** 2 - x) >= epsilon:
    print("low =", low, "High =", high, "ans =", ans)
    num_guesses += 1

    if ans ** 2 < x:
        low = ans
    else:
        high = ans
    ans = (high + low) / 2

print("num of guesses =", num_guesses)
print(ans, "is close to square root of", x)
```

Cuando se ejecuta para x = 25, imprime

```
low = 0.0 high = 25 ans = 12.5
low = 0.0 high = 12.5 ans = 6.25
low = 0.0 high = 6.25 ans = 3.125
low = 3.125 high = 6.25 ans = 4.6875
low = 4.6875 high = 6.25 ans = 5.46875
low = 4.6875 high = 5.46875 ans = 5.078125
low = 4.6875 high = 5.078125 ans = 4.8828125
low = 4.8828125 high = 5.078125 ans = 4.98046875
low = 4.98046875 high = 5.078125 ans = 5.029296875
low = 4.98046875 high = 5.029296875 ans = 5.0048828125
low = 4.98046875 high = 5.0048828125 ans = 4.99267578125
low = 4.99267578125 high = 5.0048828125 ans = 4.998779296875
low = 4.998779296875 high = 5.0048828125 ans = 5.0018310546875

numGuesses = 13

5.00030517578125 is close to square root of 25
```

Observa que encuentra una respuesta diferente a la de nuestro algoritmo anterior. Eso está perfectamente bien, ya que sigue cumpliendo la especificación del problema.

Y lo que es más importante, observa que *en cada iteración del bucle, el tamaño del espacio a buscar se reduce a la mitad*. Por este motivo, el algoritmo se denomina **búsqueda por bisección**. La búsqueda por bisección es una mejora enorme respecto a nuestro algoritmo anterior, que reducía el espacio de búsqueda sólo un poco en cada iteración.

Probemos de nuevo con `x = 123456`. Esta vez el programa sólo tarda 30 intentos en encontrar una respuesta aceptable. ¿Qué tal `x = 123456789`? Sólo tarda 45 intentos.

No hay nada especial en utilizar este algoritmo para hallar raíces cuadradas. Por ejemplo, cambiando un par de 2's a 3's, podemos usarlo para aproximar una raíz cúbica de un número no negativo.

La búsqueda por bisección es una técnica muy útil para muchas otras cosas. Por ejemplo, el siguiente código utiliza la búsqueda por bisección para encontrar una aproximación al logaritmo de base 2 de x (es decir, un número, `ans`, tal que `2 ** ans` se aproxime a `x`). Está estructurado exactamente igual que el código utilizado para encontrar una aproximación a una raíz cuadrada. Primero encuentra un intervalo que contiene una respuesta adecuada, y luego utiliza la búsqueda por bisección para explorar eficientemente ese intervalo.

```
# Find lower bound on ans
lower_bound = 0

while 2 ** lower_bound < x:
    lower_bound += 1

low = lower_bound - 1
high = lower_bound + 1

# Perform bisection search

ans = (high + low) / 2

while abs(2 ** ans - x) >= epsilon:
    if 2 ** ans < x:
        low = ans
    else:
        high = ans
    ans = (high + low) / 2

print(ans, "is close to the long base 2 of", x)
```

La búsqueda por bisección es un ejemplo de **método de aproximación sucesiva**. Estos métodos funcionan realizando una secuencia de conjeturas con la propiedad de que cada conjetura está más cerca de una respuesta correcta que la conjetura anterior.

Ejercicio manual

¿Qué haría el código si x = -25?

```
epsilon = 0.01
num_guesses, low = 0, 0
high = max(1, x)
ans = (high + low) / 2

while abs(ans ** 2 - x) >= epsilon:
    print("low =", low, "High =", high, "ans =", ans)
    num_guesses += 1

    if ans ** 2 < x:
        low = ans
```

```
        else:
            high = ans
        ans = (high + low) / 2

    print("num of guesses =", num_guesses)
    print(ans, "is close to square root of", x)
```

Mi aproximación al ejercicio:

El valor de `ans` tendera a 0 y el valor de `abs(ans ** 2 - (-25))` siempre va a ser mayor a 25, por tanto también mayor a `epsilon`, por lo cual se dará un bucle infinito.

> **Ejercicio manual**
>
> ¿Qué habría que cambiar para que el código anterior funcionara para encontrar una aproximación a la raíz cúbica de números tanto negativos como positivos? Pista: piensa en cambiar `low` para asegurar que la respuesta se encuentra dentro de la región buscada.

Mi aproximación al ejercicio:

```
epsilon = 0.01
num_guesses = 0
x = float(input("Introduce un número para calcular la raíz cúbica: "))

# Definir límites de búsqueda según el signo de x
if x < 0:
    low = min(-1.0, x)
    high = 0.0
else:
    low = 0.0
    high = max(1.0, x)

ans = (high + low) / 2.0

while abs(ans ** 3 - x) >= epsilon:
    print("low =", low, "high =", high, "ans =", ans)
    num_guesses += 1
    if ans ** 3 < x:
        low = ans
    else:
        high = ans
    ans = (high + low) / 2.0

print("num of guesses =", num_guesses)
print(ans, "is close to the cube root of", x)
```

Primero verificamos si el código es negativo o positivo y definimos los limites de búsqueda.
Cambiamos el `abs(ans ** 2 - x)` por `abs(ans ** 3 - x)` y aplicamos la búsqueda por bisección o binaria.

> **Ejercicio manual**
>
> El Empire State Building tiene 102 pisos de altura. Un hombre quiere saber desde qué piso más alto puede tirar un huevo sin que se rompa. Se propuso dejar caer un huevo desde el último piso. Si se rompía, bajaba un piso y volvía a intentarlo. Así hasta que el huevo no se rompiera. En el peor de los casos, este método requiere 102 huevos. Implementa un método que en el peor de los casos utilice siete huevos.

Mi aproximación al ejercicio:

```python
low, high = 1, 102
num_guesses = 0
eggs = 7
ans = (high + low) / 2
death = 80

while low <= high and eggs > 0:
    ans = (low + high) / 2
    num_guesses += 1
    print(f"Tirando desde el piso {ans}...")

    if ans >= death:
        print(f"💥 Se rompió en el piso {ans}")
        high = ans - 1
        eggs -= 1
    else:
        print(f"✅ No se rompió en el piso {ans}")
        low = ans + 1

print(f'Usamos {num_guesses} intentos y quedaron {eggs} huevos, y el piso seguro mas alto es el {round(high)}')
```

Unas palabras sobre el uso de flotantes

La mayoría de las veces, los números de tipo `float` proporcionan una aproximación razonablemente buena a los números reales. Pero «la mayor parte del tiempo» no es todo el tiempo, y cuando no lo hacen, puede llevar a consecuencias sorprendentes. Por ejemplo, intente ejecutar el código

```
x = 0.0
for i in range(10):
    x = x + 0.1
if x == 1.0:
    print(x, '= 1.0')
else:
    print(x, 'is not 1.0')
```

Quizá a usted, como a la mayoría, le sorprenda que se imprima,

```
0.9999999999999999 is not 1.0
```

¿Por qué llega a la cláusula else en primer lugar? Para entender por qué ocurre esto, tenemos que entender cómo se representan los números de coma flotante en el ordenador durante un cálculo. Para entender eso, necesitamos entender los números binarios.

Cuando aprendiste por primera vez sobre números decimales -es decir, números de base 10- aprendiste que *cualquier número decimal puede ser representado por una secuencia de los dígitos 0123456789*. El dígito más a la derecha es el lugar 10^0, el siguiente dígito hacia la izquierda es el lugar 10^1, etc. Por ejemplo, la secuencia de dígitos decimales 302 representa `3*100 + 0*10 + 2*1`.

¿Cuántos números diferentes puede representar una secuencia de longitud n? Una secuencia de longitud 1 puede representar cualquiera de los 10 números (0-9); una secuencia de longitud 2 puede representar 100 números diferentes (0-99). En general, una secuencia de longitud n puede representar 10^n números diferentes.

Los **números binarios** (números de base 2) funcionan de forma similar. Un número binario se representa mediante una secuencia de dígitos, cada uno de los cuales es 0 ó 1. Estos dígitos suelen denominarse **bits**. Estos dígitos suelen denominarse bits. El dígito situado más a la derecha es el lugar 2^0, el siguiente dígito hacia la izquierda es el lugar 2^1, etc. Por ejemplo, la secuencia de dígitos binarios 10^1 representa `1*4 + 0*2 + 1*1 = 5`.
¿Cuántos números distintos puede representar una secuencia de longitud n? 2^n.

Ejercicio manual

¿Cuál es el equivalente decimal del número binario 10011?

La secuencia de dígitos seria `1*2^4 + 0*2^3 + 0*2^2 + 1*2^1 + 1*2^0 = 16 + 0 + 0 + 2 + 1 = 19`

Quizá porque la mayoría de la gente tiene diez dedos, nos gusta utilizar decimales para representar números. En cambio, todos los sistemas informáticos modernos representan los números en binario. Esto no se debe a que los ordenadores nazcan con dos dedos de frente. Es porque es fácil construir interruptores de hardware, es decir, dispositivos que sólo pueden estar en uno de dos estados, encendido o apagado. Que los ordenadores utilicen una representación binaria y las personas una decimal puede provocar alguna que otra disonancia cognitiva.

En los lenguajes de programación modernos, los números no enteros se implementan mediante una representación denominada **coma flotante**. De momento, hagamos como si la representación interna fuera en decimal.

Representaríamos un número como un par de enteros: los dígitos significativos del número y un exponente. Por ejemplo, el número *1,949* se representaría como el par (1949, -3), que representa el producto $1949 * 10^{-3}$.

El número de cifras significativas determina la precisión con la que pueden representarse los números. Si, por ejemplo, sólo hubiera dos cifras significativas, el número 1,949 no podría representarse exactamente. Tendría que convertirse en alguna aproximación de 1,949, en este caso 1,9. Esa aproximación se denomina *valor redondeado*.

Los ordenadores modernos utilizan representaciones binarias, no decimales.
Representan los dígitos significativos y los exponentes en binario en lugar de en decimal, y elevan 2 en lugar de 10 al exponente. Por ejemplo, el número representado por los dígitos decimales **0,625** (5/8) se representaría como el par (101, -11); como 101 es la representación binaria del número 5 y -11 es la representación binaria de -3, el par (101, -11) representa $(5 * 2)^{-3} = 5/8 = 0,625$.

¿Qué ocurre con la fracción decimal 1/10, que en Python se escribe como 0,1? Lo mejor que podemos hacer con cuatro dígitos binarios significativos es (0011, -101). Esto equivale a 3/32, es decir, 0,09375. Si tuviéramos cinco dígitos binarios significativos, representaríamos 0,1 como (11001, -1000), lo que equivale a 25/256, es decir, 0,09765625.

¿Cuántos dígitos significativos necesitaríamos para obtener una representación exacta en coma flotante de 0,1? **Un número infinito de**

cifras. No existen enteros sig y exp tales que $sig * 2^{-exp}$ sea igual a 0.1. Por tanto, no importa cuántos bits use Python (o cualquier otro lenguaje) para representar números en coma flotante, sólo puede representar una aproximación a 0.1. En la mayoría de las implementaciones de Python, hay 53 bits de precisión disponibles para números en coma flotante, por lo que los dígitos significativos almacenados para el número decimal 0.1 serán

```
1   1100110011001100110011001100110011001100110011001100110011001
```

Equivale al número decimal

```
1   0.1000000000000000055511151231257827021181583404541015625
```

Bastante cerca de 1/10, pero no exactamente 1/10.
Volviendo al misterio original, porque

```
1   x = 0.0
2   for i in range(10):
3       x = x + 0.1
4   if x == 1.0:
5       print(x, '= 1.0')
6   else:
7       print(x, 'is not 1.0')
```

Imprime

```
1   0.9999999999999999 is not 1.0
```

Ahora vemos que la prueba `x == 1.0` produce el resultado Falso porque el valor al que está ligado `x` *no es exactamente 1.0*. Esto explica por qué se ejecutó la cláusula `else`. Pero, ¿por qué decidió que x era menor que 1.0 cuando la representación en coma flotante de 0.1 es ligeramente mayor que 0.1? Porque durante alguna iteración del bucle, Python se quedó sin dígitos significativos e hizo algún redondeo, que resultó ser hacia abajo. ¿Qué se imprime si añadimos al final de la cláusula `else` el código `print(x == 10.0 * 0.1)`? Se imprime False. No es lo que nos enseñaron nuestros profesores de primaria, pero sumar 0,1 diez veces no produce el mismo valor que multiplicar 0,1 por 10.

Por cierto, si quieres redondear explícitamente un número en coma flotante, utiliza la función round. La expresión `round(x, num_digits)` devuelve el número en coma flotante equivalente a redondear el valor de `x` a `num_digits` dígitos después del punto decimal. Por ejemplo, `print(round(2**0.5, 3))` imprimirá `1.414` como aproximación a la raíz cuadrada de 2.

¿Realmente importa la diferencia entre números reales y de coma flotante? *La mayoría de las veces, por suerte, no*. Hay pocas situaciones en las que importe la diferencia entre 0,9999999999999999, 1,0 y 1,0000000000000001. Sin embargo, algo de lo que casi siempre merece la pena preocuparse es de las pruebas de igualdad. Como hemos visto, usar `==` para comparar dos valores de coma flotante puede producir un resultado sorprendente. Casi siempre es más apropiado preguntarse si dos valores de coma flotante están lo suficientemente cerca el uno del otro, no si son idénticos. Así, por ejemplo, es mejor escribir `abs(x-y) < 0.0001` en lugar de `x == y`.

Otro motivo de preocupación es la acumulación de errores de redondeo. La mayoría de las veces las cosas funcionan bien, porque a veces el número almacenado en el ordenador es un poco mayor de lo previsto, y a veces es un poco menor de lo previsto. Sin embargo, en algunos programas, los errores irán todos en la misma dirección y se acumularán con el tiempo.

Ejercicio manual de la lectura

Suponga que se le da una variable entera positiva llamada N. Escriba un fragmento de código Python que encuentre la raíz cúbica de N. El código imprime la raíz cúbica si N es un cubo perfecto o imprime un error si N no es un cubo perfecto. Sugerencia: utiliza un bucle que incremente un contador - tú decides cuándo debe detenerse el contador.

Mi aproximación

```python
N = int(input("Ingrese un número entero positivo: "))
encontrado = False
x = 0

while x ** 3 <= N:
    if x ** 3 == N:
        print(f"La raíz cúbica de {N} es {x}")
        encontrado = True
        break
    x += 1

if not encontrado:
    print(f"{N} no es un cubo perfecto.")
```

Mediante el bucle que se itera mientras que $x^3 <= N$, si x^3 es igual a N, entonces encontré el cubo perfecto y rompo el bucle. Si no se

encuentra ningún cubo perfecto devuelvo un mensaje de error.

Ejercicio manual de la lectura

Suponga que se le da una variable de cadena llamada `mi_cadena`. Escribe un fragmento de código Python que imprima una nueva cadena que contenga los caracteres pares de `mi_cadena`. Por ejemplo, si `mi_cadena = «abcdefg»` entonces tu código debería imprimir `aceg`.

Mi aproximación

```
1  mi_cadena = input("Ingrese un texto para devolver solo las cadenas
   pares:")
2  nueva_cadena = mi_cadena[::2]
3  print(nueva_cadena)
```

Mediante el método de slicing inicio desde la posición 0, hasta el final y con 2 pasos.
