Números aleatorios Python

Generar números aleatorios en Python es por suerte una tarea muy fácil usando la librería random.

Vamos a ver diferentes funciones:

randint

uniform

random

choice

Función randint

La función randint genera un número aleatorio entero entre 0 y n, siendo el segundo valor 30 en este caso.

from random import \*

print(randint(0, 30))

# 10

Función uniform

Si queremos generar valores aleatorios que sean decimales podemos hacer uso de uniform.

from random import \*

print(uniform(0,10))

# 6.68079620859125

Función random

Por otro lado tenemos la función random que no acepta parámetros y genera números aleatorios decimales de entre 0 y 1.

from random import \*

print(random())

# 0.00402817235037356

Función choice

Por último, la función choice nos permite elegir un elemento aleatorio de una lista.

from random import \*

print(choice(["A", "B", "C"]))

# C

Números primos Python

Los números primos son aquellos que sólo son divisibles por uno y por sí mismos. Se entiende por divisible a que el resultado sea un número entero, es decir, no decimal.

Por lo tanto los números 2, 3, 5, 7, 11 son primos, ya que no hay ningún número que los pueda dividir de manera entera (excluyendo al 1 y al propio número)

Sin embargo, el número 10 no es primo, ya que se puede dividir por 2 y por 5.

Los números primos son un concepto un tanto enigmático y muy estudiado. Algunas teorías como la Conjetura de Goldbach son de lo más interesantes.

Todo número entero mayor que 5 se puede escribir como suma de tres números primos.

Sabida ya la teoría, veamos como se puede implementar en Python una función que calcule si un número es primo o no. Veremos dos formas, una usando bucles for y otra empleando recursividad.

Números primos en Python (bucles)

Para implementar una calculadora de números primos en Python, lo primero es saber si dos números son divisibles. Usamos el operador módulo %.

if D%d != 0:

print("No es divisor")

El operador módulo nos devuelve el resto de la división entre dos números. Por lo tanto 5%3 es 2. Es decir, si el resto es distinto de cero, decimos que d no es divisor de D.

Para determinar si un número es primo, iteramos todos los números desde 2 hasta nuestro número, comprobando si ese número n puede dividir al nuestro. En el momento en el que encontramos a un divisor, ya sabemos que no es primo y devolvemos False.

Si por lo contrario ningún número consigue dividir al nuestro, devolvemos True indicando que el número es primo.

def es\_primo(num):

for n in range(2, num):

if num % n == 0:

print("No es primo", n, "es divisor")

return False

print("Es primo")

return True

Veamos un ejemplo de cómo usar nuestra función.

es\_primo(13) # Es primo

es\_primo(14) # No es primo 2 es divisor

es\_primo(887) # Es primo

es\_primo(1001) # No es primo 7 es divisor

Número primos en Python (recursividad)

Ciertos algoritmos que usan bucles, pueden ser implementados con recursividad, es decir, haciendo uso de una función que se llama repetidas veces a sí misma.

La siguiente función se llama a sí misma comprueba si un número es divisible por otro, empezando por n=2 hasta llegar al número en cuestión.

Si n es mayor que nuestro número, significa que ya hemos probado a dividir nuestro número con todos los anteriores, por lo que podemos considerar que es primo.

Si el número no es divisible entre n, continuamos llamando a es\_primo incrementando n.

Si detectamos que el número es divisible por n, salimos de la función indicando que no es primo.

def es\_primo(num, n=2):

if n >= num:

print("Es primo")

return True

elif num % n != 0:

return es\_primo(num, n + 1)

else:

print("No es primo", n, "es divisor")

return False

Fíjate en lo siguiente:

Resulta fácil comprobar que un número no es primo, ya que basta con encontrar un número que lo divida. Se puede ver a simple vista que 10029438 no es primo, ya que al ser par es divisible por 2.

Pero es difícil demostrar que un número es primo, ya que es necesario comprobar que no es divisible por ninguno, y esto requiere de más esfuerzo.

Veamos como usar la función.

es\_primo(33) # No es primo 3 es divisor

es\_primo(10000000175489) # No es primo 599 es divisor

Por último, es importante tener en cuenta que Python pone un cierto límite a las veces que una función puede ser llamada recursivamente. En algunos casos podrías encontrarte con un RecursionError, indicando que has superado el límite.

es\_primo(1009)

# RecursionError: maximum recursion depth exceeded in comparison

Medir tiempo de ejecución

Medir el tiempo de ejecución de un programa es una tarea muy importante, ya que programar no sólo consiste en crear código que funcione, sino código que pueda ser ejecutado en un tiempo razonable. ¿Te imaginas que consultar tu saldo en la web del banco tardase 5 horas?

Imagina que tienes un determinado algoritmo que tarda en ejecutarse 1 segundo. A priori no parece demasiado, pero ¿y si tuviéramos a 1000 usuarios ejecutando ese algoritmo periódicamente? En este caso, podría ser interesante optimizarlo, ya que una simple reducción de 0.1 segundos, podría aliviar la carga de nuestro sistema notablemente.

Por lo contrario, si tenemos un determinado algoritmo o proceso que tarda 1 segundo, pero es usado muy de vez en cuanto, tal vez no sea interesante invertir recursos en optimizarlo.

Afortunadamente, Python nos proporciona diferentes formas de medir el tiempo de ejecución de un programa.

Tiempo de ejecución con time

La librería time es bastante completa, pero lo único que nos interesa para medir el tiempo de ejecución es el método time().

Este método nos devuelve el número de segundos, con precisión de microsegundos, que han pasado desde el 1 de Enero de 1970.

import time

print(time.time()) # 1609954317.943479

Para medir el tiempo de ejecución de nuestros programas, basta con saber el tiempo que ha pasado antes y después de ejecutar nuestro programa, y la diferencia será el tiempo que ha tardado nuestro código.

import time

inicio = time.time()

# Código a medir

time.sleep(1)

# -------------

fin = time.time()

print(fin-inicio) # 1.0005340576171875

En este caso podemos ver como el tiempo transcurrido es prácticamente 1, ya que time.sleep(1) tarda 1 segundo en ejecutarse.

Podemos ver también el tiempo que tardamos en calcular los primeros 10000000 números pares.

import time

inicio = time.time()

# Código a medir

lista = [i for i in range(10000000) if i%2==0]

# -------------

fin = time.time()

print(fin-inicio) # 1.5099220275878906

Por otro lado, también podemos crear un decorador que use time, lo que nos permitirá medir el tiempo de ejecución de nuestras funciones sin repetir tanto código. Creamos el decorador de la siguiente manera:

def mide\_tiempo(funcion):

def funcion\_medida(\*args, \*\*kwargs):

inicio = time.time()

c = funcion(\*args, \*\*kwargs)

print(time.time() - inicio)

return c

return funcion\_medida

Ahora podemos usar el decorador mide\_tiempo con @ antes de nuestra función, y cada vez que la llamemos se imprimirá por pantalla el tiempo que tardó en ejecutarse.

@mide\_tiempo

def calcula\_pares(n):

return [i for i in range(n) if i%2 == 0]

calcula\_pares(10000000)

# 1.4356

Tiempo de ejecución con timeit

Python nos ofrece también otra librería denominada timeit, y está pensada para medir tiempos de ejecución de fragmentos pequeños de código. Puede ser usada por línea de comandos o como una función dentro de Python.

El siguiente fragmento de código nos permite medir el tiempo de ejecución de la sentencia x=5. El argumento number es el número de veces que se ejecuta el fragmento de código.

import timeit

tiempo = timeit.timeit('x = 5', number=100000000)

print(tiempo) # 2.173444958

Es importante notar que el tiempo que se nos devuelve, es la suma de todas las ejecuciones. Es decir, el siguiente ejemplo tarda 2.17 segundos en ejecutar x=5 un total de 100000000 veces.

En el siguiente ejemplo vemos como la siguiente list comprehension tarda en ejecutarse una media de 0.18 segundos.

import timeit

tiempo = timeit.timeit('lista = [i for i in range(1000000) if i%2==0]', number=5)

# Calculamos el tiempo medio

print(tiempo/5) # 0.18671

Consejos y conclusiones

Para medir el tiempo de ejecución de tus programas, haz la media de varias ejecuciones. Debido a diferentes factores, nunca obtendrás el mismo resultado, por eso es importante realizar la media. Observa los diferentes valores que obtienes, y si son muy dispares, plantéate las cosas.

Si mides fragmentos de código que tardan muy poco en ejecutarse, deberás promediar más valores.

Ten en cuenta que dependiendo de tu sistema operativo, la precisión que obtendrás de las diferentes librerías puede variar.

No pierdas tiempo en optimizar códigos que apenas son usados. Dedica tiempo a analizar el código que requiere de optimización y centra ahí tus esfuerzos.

Hash en Python

Una función hash es una función que dada una entrada de longitud variable devuelve una secuencia de longitud fija, con algunas propiedades interesantes. Si por ejemplo aplicamos como entrada la cadena El Libro De Python, la salida será la siguiente.

import hashlib

salida = hashlib.sha256(b"El Libro De Python").hexdigest()

print(salida)

# Salida

# f7b5c532807800c540f5e4476ea1f6d968294fc34c90f2e7e64435ea3c054ce6

Existen diferentes funciones hash, y en el caso anterior hemos usado la sha256. Una función hash se puede ver como una función resumen, ya que nos permite “resumir” un conjunto de datos de longitud variable en una secuencia de longitud fija (y relativamente corta). Podríamos también meter el libro entero de El Quijote un su función hash sha256 sería:

03f22ee1408a1bea9a7a9dfc0431051432c26a8a16fa6925d5246ff3235de3a4

Hemos por tanto resumido cientos de páginas en una sola línea.

A continuación veremos:

Las propiedades de las funciones hash.

Las aplicaciones de las funciones hash.

Los diferentes tipos de funciones hash.

Cómo usar funciones hash en Python con hashlib.

Propiedades de las Funciones Hash

Las funciones hash tienen unas propiedades que las hacen muy útiles en el mundo de la criptografía y blockchain:

A pesar de que la entrada tiene una longitud arbitraria, la salida tiene una longitud fija. Esta longitud vendrá determinada por el tipo de función hash que se use. Por ejemplo, la sha256 devuelve siempre 256 bits (o 32 bytes).

Las funciones hash suelen ser rápidas de calcular.

Siendo x la entrada y hash(x) su función hash, es imposible (o muy muy difícil) obtener x a partir de hash(x). Es decir, que la función hash no es reversible. Si tenemos el hash de El Quijote, no podemos reconstruir el libro a partir del hash. Esto se conoce como resistencia a la primera preimagen.

Tiene que ser muy complicado (por no decir imposible) encontrar una nueva entrada x' siendo x'!=x tal que hash(x) = hash(x'). Es decir, tiene que ser imposible encontrar dos entradas cuya función hash sea la misma. Esto se conoce como resistencia a la segunda imagen.

Por último, la resistencia a colisiones o collision resistance nos dice que debe ser imposible encontrar dos entradas diferentes y distintas cuyo hash sea el mismo.

Es importante notar que algunas de estas características son de vital importancia, y que si alguna de ellas dejara de cumplirse, el mundo de Internet estaría en serios problemas.

Por ejemplo, si se llegara a poder revertir una función hash, los pagos online que realizamos, contraseñas o incluso blockchains como bitcoin o ethereum podrían estar en problemas. Se dice que la computación cuántica podría romper las funciones hash, pero aún quedan años para eso.

Aplicaciones de Función Hash

Las funciones hash tienen aplicaciones en diferentes sectores. Explicamos a continuación sus casos de uso más relevantes:

Integridad de información: Podemos usar las funciones hash para asegurarnos de que un determinado contenido digital no ha sido modificado. Si por ejemplo calculamos el hash de un vídeo o un libro y lo almacenamos, tendremos una “huella digital” de dicho contenido. Si en un futuro nos envían ese mismo vídeo o libro, podemos calcular el hash otra vez y compararlo con el que teníamos almacenado anteriormente. Esto nos ahorra tener que ir fotograma a fotograma o página a página comparando ambos archivos.

Generar números aleatorios: Podemos usar las funciones hash para generar números aleatorios, o para ser más preciso para generar números pseudoaleatorios.

Firma digital: En la firma digital se suele firmar sólo el hash del mensaje en vez del contenido entero, lo que resulta más eficiente y reduce ciertos vectores de ataque.

Merkle Trees: Los merkle trees también pueden ser usados para resumir información, donde la misma es dividida en pequeños trozos y su hash es calculado recursivamente hasta obtener un único hash llamado merkle root. Estos son muy utilizados en la blockchain.

Tipos de Funciones Hash

Existen diferentes funciones hash, donde cada una tiene sus casos de uso. Algunas de las características más importantes son la longitud de la salida y el algoritmo que usan:

BLAKE: Tiene variantes como la BLAKE-2, BLAKE-3, siendo la última anunciada en 2020. Existen diferentes variantes en función del número de bits de su salida.

MD: Tiene múltiples variantes como la MD1, MD2, MD3, MD4 y MD5. El MD5 es muy usado para integridad de datos y fue introducido en la RFC 1321.

SHA: Tiene variantes como la SHA256, SHA512, SHA224, SHA384. El SHA256 es el usado por la criptomoneda Bitcoin.

KECCAK-256: Usado por la criptomoneda Ethereum.

Funciones Hash en Python

Gracias a la librería hashlib de Python disponemos de prácticamente todas las funciones hash que existen. Veamos por ejemplo como usar la sha256.

import hashlib

m = hashlib.sha256()

m.update(b"El Libro De Python")

salida = m.hexdigest()

print(salida)

# f7b5c532807800c540f5e4476ea1f6d968294fc34c90f2e7e64435ea3c054ce6

También podemos acceder al digest\_size, es decir a la longitud de la salida. Este será un valor fijo dentro de cada función hash, y por ejemplo en el caso de sha256 es 32 bytes, o lo que viene siendo lo mismo, 256 bits. De ahí viene su nombre.

print(m.digest\_size)

# Salida

# 32

También podemos hacer el hash de múltiples entradas:

import hashlib

m = hashlib.sha256()

m.update(b"Secuencia 1")

m.update(b"Secuencia 2")

m.update(b"Secuencia 3")

salida = m.hexdigest()

print(salida)

# Salida:

# 8bfe6a71680cdf4cc4c4024b3808f0d865a729c58695b145e7408555c24aab29

A continuación podemos ver ejemplos para varias funciones hash, donde todas usan la misma entrada. Podemos ver como las salidas son diferentes y tienen distinta longitud.

import hashlib

print(hashlib.sha256(b"El Libro de Python").hexdigest())

print(hashlib.sha224(b"El Libro de Python").hexdigest())

print(hashlib.sha512(b"El Libro de Python").hexdigest())

print(hashlib.blake2b(b"El Libro de Python").hexdigest())

print(hashlib.blake2s(b"El Libro de Python").hexdigest())

print(hashlib.blake2s(b"El Libro de Python").hexdigest())

print(hashlib.md5(b"El Libro de Python").hexdigest())

# Salida:

# 3f0eb88c12b73f8235f3bc5a19336d32a41bbe6743291c97e8fb00ea2d3520e0

# a46e8c9207522d305f79b818f37170c8b3094acb607ed3645854ea38

# 01a489b59eb18cc2d297e3be6918ba1cff2875514900ce781a95c006a05eac68c9dffc0ae9bb64c00dbc628fa1b2a28159e8cee2875e86157d82c0998a786beb

# 2728dcb655d2572bff0f0ecad1518ddc005a7896ea18e0f47c07cf54efcb639266c3056ebd4ef62abad34d8ccd074925012cde3da4a351f0f14831b7c36f6a48

# cae14ea7acc2827b66212f3da5ea35d8d65fe248fc00030ddb9f9e5113f120a7

# cae14ea7acc2827b66212f3da5ea35d8d65fe248fc00030ddb9f9e5113f120a7

# 77dd74c5fa2c54e26cc67b5ab47b38fd