Capítulo 5: Jugar con conjuntos y probabilidad

### **Probabilidad**

Los conjuntos nos permiten razonar sobre los conceptos básicos de la probabilidad. Empezaremos con algunas definiciones:

**Experimento** El *experimento* es simplemente la prueba que queremos realizar. Realizamos la prueba porque nos interesa la probabilidad de cada resultado posible. Lanzar un dado, lanzar una moneda y sacar una carta de una baraja son ejemplos de experimentos. Una única ejecución de un experimento se denomina *ensayo*.

*Espacio muestral* Todos los resultados posibles de un experimento forman un conjunto conocido como *espacio muestral*, que normalmente llamaremos *S* en nuestras fórmulas. Por ejemplo, cuando se lanza una vez un dado de seis caras, el espacio muestral es {1, 2, 3, 4, 5, 6}.

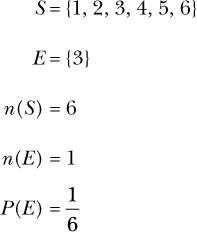
**Evento** Un *evento* es un conjunto de resultados de los que queremos calcular la probabilidad y que forman un *subconjunto* del espacio muestral. Por ejemplo, podemos querer conocer la probabilidad de un resultado concreto, como sacar un 3, o la probabilidad de un conjunto de resultados múltiples, como sacar un número par (2, 4 ó 6). Utilizaremos la letra *E* en nuestras fórmulas para representar un suceso.

Si existe una *distribución uniforme,*es decir, si cada resultado del espacio muestral tiene la misma probabilidad de producirse, la probabilidad de que se produzca un suceso, *P(E*), se calcula mediante la siguiente fórmula (hablaré de las distribuciones no uniformes un poco más adelante en este capítulo):

image

Aquí, *n*(*E*) y *n(S*) son la cardinalidad de los conjuntos *E*, el suceso, y *S*, el espacio muestral, respectivamente. El valor de *P(E*) oscila entre 0 y 1, y los valores más altos indican una mayor probabilidad de que ocurra el suceso.

Podemos aplicar esta fórmula a la tirada de un dado para calcular la probabilidad de una determinada tirada, por ejemplo, 3:



Esto confirma lo que era evidente desde el principio: la probabilidad de una determinada tirada es 1/6. Podrías hacer fácilmente este cálculo mentalmente, pero podemos utilizar esta fórmula para escribir la siguiente función en Python que calcula la probabilidad de cualquier suceso, event, en cualquier espacio muestral, space:

def probability(space, event):  
return len(event)/len(space)

En esta función, los dos argumentos space y event-el espacio muestral y el suceso- no tienen por qué ser conjuntos creados con FiniteSet. También pueden ser listas o, para el caso, cualquier otro objeto de Python que admita la función len().

Utilizando esta función, escribamos un programa para averiguar la probabilidad de que aparezca un número primo al lanzar un dado de 20 caras:

def probability(space, event):  
return len(event)/len(space)  
  
➊ def check\_prime(number):  
if number != 1:  
for factor in range(2, number):  
if number % factor == 0:  
return False  
else:  
return False  
return True  
  
if \_\_name\_\_ == '\_\_main\_\_':  
➋ space = FiniteSet(\*range(1, 21))  
primes = []  
for num in s:  
➌ if check\_prime(num):  
primes.append(num)  
➍ event= FiniteSet(\*primes)  
p = probability(space, event)  
  
print('Sample space: {0}'.format(space))  
print('Event: {0}'.format(event))  
print('Probability of rolling a prime: {0:.5f}'.format(p))

Primero creamos un conjunto que represente el espacio muestral, space, utilizando la función range() en ➋. Para crear el conjunto de sucesos, necesitamos encontrar los números primos del espacio muestral, así que definimos una función, check\_prime(), en ➊. Esta función toma un número entero y comprueba si es divisible (sin resto) por cualquier número entre 2 y él mismo. Si es así, devuelve False. Como un número primo sólo es divisible por 1 y por sí mismo, esta función devuelve True si un número entero es primo y False en caso contrario.

Llamamos a esta función para cada uno de los números del espacio muestral en ➌ y añadimos los números primos a una lista, primes. A continuación, creamos nuestro conjunto de sucesos, event, a partir de esta lista en ➍. Por último, llamamos a la función probability() que hemos creado antes. Al ejecutar el programa obtenemos el siguiente resultado:

Sample space: {1, 2, 3, ..., 18, 19, 20}  
Event: {2, 3, 5, 7, 11, 13, 17, 19}  
Probability of rolling a prime: 0.40000

Aquí, *n(E*) = 8 y *n(S*) = 20, por lo que la probabilidad, *P*, es 0,4.

En nuestro programa del dado de 20 caras, realmente no necesitábamos crear los conjuntos; en su lugar, podríamos haber llamado a la función probability() con el espacio muestral y los sucesos como listas:

if \_\_name\_\_ == '\_\_main\_\_':  
space = range(1, 21)  
primes = []  
for num in space:  
if check\_prime(num):  
primes.append(num)  
p = probability(space, primes)

La función probability() funciona igual de bien en este caso.

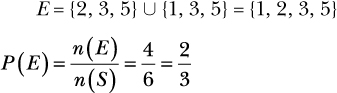
#### ***Probabilidad del suceso A o del suceso B***

Supongamos que nos interesan dos posibles sucesos y queremos hallar la probabilidad de que ocurra *alguno de* ellos. Por ejemplo, volviendo a una simple tirada de dado, consideremos los dos sucesos siguientes:

A = El número es un número primo.

B = El número es impar.

Como antes, el espacio muestral, *S*, es {1, 2, 3, 4, 5, 6}. El suceso A puede representarse como el subconjunto {2, 3, 5}, el conjunto de números primos del espacio muestral, y el suceso B puede representarse como {1, 3, 5}, los números impares del espacio muestral. Para calcular la probabilidad de cualquiera de los dos conjuntos de resultados, podemos hallar la probabilidad de la *unión* de los dos conjuntos. En nuestra notación, podríamos decir



Ahora vamos a realizar este cálculo en Python:

>>> from sympy import FiniteSet  
>>> s = FiniteSet(1, 2, 3, 4, 5, 6)  
>>> a = FiniteSet(2, 3, 5)  
>>> b = FiniteSet(1, 3, 5)  
➊ >>> e = a.union(b)  
>>> len(e)/len(s)  
0.6666666666666666

Primero creamos un conjunto, s, que representa el espacio muestral, seguido de los dos conjuntos a y b. A continuación, en ➊, utilizamos el método union() para encontrar el conjunto de sucesos, e. Por último, calculamos la probabilidad de la unión de los dos conjuntos utilizando la fórmula anterior.

#### ***Probabilidad del suceso A y del suceso B***

Supongamos que tienes dos sucesos en mente y quieres calcular las probabilidades de que ocurran *ambos*; por ejemplo, las probabilidades de que la tirada de un dado sea a la vez primo e impar. Para determinarlo, calcula la probabilidad de la intersección de los dos conjuntos de sucesos:

*E* = *A* ∩ *B* = {2, 3, 5} ∩ {1, 3, 5} = {3, 5}

Podemos calcular la probabilidad de que ocurran tanto A como B utilizando el método intersect(), que es similar al que hicimos en el caso anterior:

>>> from sympy import FiniteSet  
>>> s = FiniteSet(1, 2, 3, 4, 5, 6)  
>>> a = FiniteSet(2, 3, 5)  
>>> b = FiniteSet(1, 3, 5)  
>>> e = a.intersect(b)  
>>> len(e)/len(s)  
0.3333333333333333

#### ***Generar números aleatorios***

Los conceptos de probabilidad nos permiten razonar y calcular la posibilidad de que ocurra un suceso. Para simular realmente tales sucesos -como un simple juego de dados- utilizando programas informáticos, necesitamos una forma de generar números aleatorios.

##### **Simular la tirada de un dado**

Para simular la tirada de un dado de seis caras, necesitamos una forma de generar un número entero aleatorio entre 1 y 6. El módulo random de la biblioteca estándar de Python nos proporciona varias funciones para generar números aleatorios. Dos funciones que utilizaremos en este capítulo son la función randint(), que genera un entero aleatorio en un rango dado, y la función random(), que genera un número de coma flotante entre 0 y 1. Veamos un ejemplo rápido de cómo funciona la función randint():

>>> import random  
>>> random.randint(1, 6)  
4

La función randint() toma dos números enteros como argumentos y devuelve un número entero aleatorio que cae entre estos dos números (ambos inclusive). En este ejemplo, pasamos el rango (1, 6), y nos devolvió el número 4, pero si volvemos a llamarla, es muy probable que obtengamos un número diferente:

>>> random.randint(1, 6)  
6

Llamar a la función randint() nos permite simular la tirada de nuestro dado virtual. Cada vez que llamemos a esta función, obtendremos un número entre 1 y 6, igual que si lanzáramos un dado de seis caras. Ten en cuenta que randint() espera que proporciones primero el número más bajo, por lo que randint(6, 1) no es válido.

##### **¿Puedes sacar esa puntuación?**

Nuestro siguiente programa simulará un sencillo juego de lanzamiento de dados, en el que seguimos lanzando el dado de seis caras hasta que hayamos sacado un total de 20:

'''  
Roll a die until the total score is 20  
'''  
  
import matplotlib.pyplot as plt  
import random  
  
target\_score = 20  
  
def roll():  
return random.randint(1, 6)  
  
if \_\_name\_\_ == '\_\_main\_\_':  
score = 0  
num\_rolls = 0  
➊ while score < target\_score:  
die\_roll = roll()  
num\_rolls += 1  
print('Rolled: {0}'.format(die\_roll))  
score += die\_roll  
  
print('Score of {0} reached in {1} rolls'.format(score, num\_rolls))

Primero, definimos la misma función roll() que creamos antes. A continuación, utilizamos un bucle while en ➊ para llamar a esta función, llevar la cuenta del número de tiradas, imprimir la tirada actual y sumar la puntuación total. El bucle se repite hasta que la puntuación llega a 20, y entonces el programa imprime la puntuación total y el número de tiradas.

Aquí tienes un ejemplo de ejecución:

Rolled: 6  
Rolled: 2  
Rolled: 5  
Rolled: 1  
Rolled: 3  
Rolled: 4  
Score of 21 reached in 6 rolls

Si ejecutas el programa varias veces, te darás cuenta de que el número de tiradas que tarda en llegar a 20 varía.

##### **¿Es posible alcanzar la puntuación objetivo?**

Nuestro siguiente programa es similar, pero nos dirá si es posible alcanzar una determinada puntuación objetivo con un número máximo de tiradas:

from sympy import FiniteSet  
import random  
  
def find\_prob(target\_score, max\_rolls):  
  
die\_sides = FiniteSet(1, 2, 3, 4, 5, 6)  
# Sample space  
➊ s = die\_sides\*\*max\_rolls  
# Find the event set  
if max\_rolls > 1:  
success\_rolls = []  
➋ for elem in s:  
if sum(elem) >= target\_score:  
success\_rolls.append(elem)  
else:  
if target\_score > 6:  
➌ success\_rolls = []  
else:  
success\_rolls = []  
for roll in die\_sides:  
➍ if roll >= target\_score:  
success\_rolls.append(roll)  
➎ e = FiniteSet(\*success\_rolls)  
# Calculate the probability of reaching target score  
return len(e)/len(s)  
  
if \_\_name\_\_ == '\_\_main\_\_':  
  
target\_score = int(input('Enter the target score: '))  
max\_rolls = int(input('Enter the maximum number of rolls allowed: '))  
  
p = find\_prob(target\_score, max\_rolls)  
print('Probability: {0:.5f}'.format(p))

Cuando ejecutas este programa, te pide la puntuación objetivo y el número máximo de tiradas permitidas, y luego imprime la probabilidad de alcanzarla.

Aquí tienes dos ejemplos de ejecución:

Enter the target score: 25  
Enter the maximum number of rolls allowed: 4  
Probability: 0.00000  
  
Enter the target score: 25  
Enter the maximum number of rolls allowed: 5  
Probability: 0.03241

Vamos a entender el funcionamiento de la función find\_prob(), que realiza el cálculo de la probabilidad. El espacio muestral aquí es el producto cartesiano, die\_sidesmax\_rolls ➊, donde die\_sides es el conjunto {1, 2, 3, 4, 5, 6}, que representa los números de un dado de seis caras, y max\_rolls es el número máximo de tiradas de dado permitidas.

El conjunto de sucesos son todos los conjuntos del espacio muestral que nos ayudan a alcanzar esta puntuación objetivo. Aquí hay dos casos: cuando el número de turnos restantes es mayor que 1 y cuando estamos en el último turno. En el primer caso, en ➋, iteramos sobre cada una de las tuplas del producto cartesiano y sumamos las que sumen o superen target\_score en la lista success\_rolls. El segundo caso es especial: nuestro espacio muestral es sólo el conjunto {1, 2, 3, 4, 5, 6}, y sólo nos queda un lanzamiento del dado. Si el valor de la puntuación objetivo es mayor que 6, no es posible alcanzarla, y establecemos success\_rolls como lista vacía en ➌. Si, por el contrario, target\_score es menor o igual que 6, iteramos sobre cada tirada posible y sumamos las que sean mayores o iguales que el valor de target\_score en ➍.

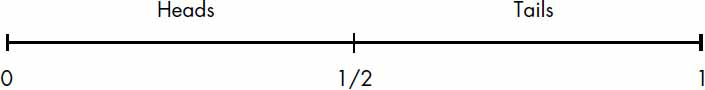
En ➎, calculamos el conjunto de sucesos, e, a partir de la lista success\_rolls que construimos anteriormente y, a continuación, devolvemos la probabilidad de alcanzar la puntuación objetivo.

#### ***Números aleatorios no uniformes***

Hasta ahora, en nuestras discusiones sobre la probabilidad hemos supuesto que cada uno de los resultados del espacio muestral tiene la misma probabilidad. La función random.randint(), por ejemplo, devuelve un número entero en el rango especificado suponiendo que cada número entero es *igualmente* probable. Nos referimos a dicha probabilidad como *probabilidad uniforme* y a los números aleatorios generados por la función randint() como *números aleatorios uniformes*. Digamos, sin embargo, que queremos simular un lanzamiento de moneda sesgado: una moneda cargada para la que la probabilidad de que salga cara es el doble que la de que salga cruz. Entonces necesitaríamos una forma de generar números aleatorios *no uniformes*.

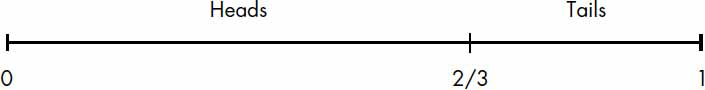
Antes de escribir el programa para hacerlo, repasaremos la idea que hay detrás.

Considera una recta numérica con una longitud de 1 y con dos intervalos divididos por igual, como se muestra en la [Figura 5-1](ch05.html#ch5fig1).



*Figura 5-1: Recta numérica con una longitud de 1 dividida en dos intervalos iguales correspondientes a la probabilidad de cara o cruz en el lanzamiento de una moneda*

Nos referiremos a esta recta como la *recta* numérica de la probabilidad, en la que cada división representa un resultado igualmente posible; por ejemplo, cara o cruz al lanzar una moneda. Ahora, en la [Figura 5-2](ch05.html#ch5fig2), considera una versión diferente de esta recta numérica.



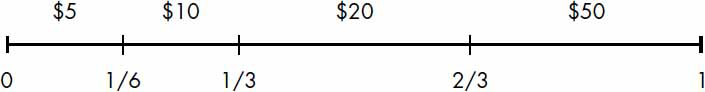
*Figura 5-2: Una recta numérica con una longitud de 1 dividida en dos intervalos desiguales correspondientes a la probabilidad de cara o cruz en el lanzamiento sesgado de una moneda*

Aquí, la división correspondiente a cara es 2/3 de la longitud total y la correspondiente a cruz es 1/3. Esto representa la situación de una moneda que probablemente salga cara en 2/3 de los lanzamientos y cruz sólo en 1/3 de los lanzamientos. La siguiente función de Python simulará un lanzamiento de moneda de este tipo, considerando esta probabilidad desigual de que salga cara o cruz:

import random  
  
def toss():  
# 0 -> Heads, 1-> Tails  
➊ if random.random() < 2/3:  
return 0  
else:  
return 1

Suponemos que la función devuelve 0 para indicar cara y 1 para indicar cruz, y luego generamos un número aleatorio entre 0 y 1 en ➊ utilizando la función random.random(). Si el número generado es menor que 2/3 -la probabilidad de lanzar cara con nuestra moneda sesgada- el programa devuelve 0; en caso contrario, devuelve 1 (cruz).

Ahora veremos cómo podemos extrapolar la función anterior para simular un suceso no uniforme con múltiples resultados posibles. Consideremos un cajero automático ficticio que dispensa un billete de 5$, 10$, 20$ o 50$ cuando se pulsa su botón. Las distintas denominaciones tienen distintas probabilidades de ser dispensadas, como se muestra en la [Figura 5-3](ch05.html#ch5fig3).



*Figura 5-3: Recta numérica de longitud 1 dividida en cuatro intervalos de distinta longitud correspondientes a la probabilidad de dispensar billetes de distintas denominaciones*

Aquí, la probabilidad de que se dispense un billete de 5$ o de 10$ es de 1/6, y la probabilidad de que se dispense un billete de 20$ o de 50$ es de 1/3.

Creamos una lista para almacenar la suma móvil de las probabilidades, y luego generamos un número aleatorio entre 0 y 1. Empezamos por el extremo izquierdo de la lista que almacena la suma y devolvemos el primer índice de esta lista para el que la suma correspondiente sea menor o igual que el número aleatorio generado. La función get\_index() pone en práctica esta idea:

'''  
Simulate a fictional ATM that dispenses dollar bills  
of various denominations with varying probability  
'''  
  
import random  
  
def get\_index(probability):  
c\_probability = 0  
➊ sum\_probability = []  
for p in probability:  
c\_probability += p  
sum\_probability.append(c\_probability)  
➋ r = random.random()  
for index, sp in enumerate(sum\_probability):  
➌ if r <= sp:  
return index  
➍ return len(probability)-1  
  
def dispense():  
  
dollar\_bills = [5, 10, 20, 50]  
probability = [1/6, 1/6, 1/3, 2/3]  
bill\_index = get\_index(probability)  
return dollar\_bills[bill\_index]

Llamamos a la función get\_index() con una lista que contiene la probabilidad de que se produzca el suceso en la posición correspondiente. A continuación, en ➊, construimos la lista sum\_probability, donde el iº elemento es la suma de los primeros i elementos de la lista probability. Es decir, el primer elemento de sum\_probability es igual al primer elemento de probability, el segundo elemento es igual a la suma de los dos primeros elementos de probability, y así sucesivamente. En ➋, se genera un número aleatorio entre 0 y 1 utilizando la etiqueta r. A continuación, en ➌, recorremos sum\_probability y devolvemos el índice del primer elemento que supere r.

La última línea de la función, en ➍, se ocupa de un caso especial que se ilustra mejor con un ejemplo. Considera una lista de tres sucesos con porcentajes de ocurrencia cada uno expresados como 0,33. En este caso, la lista sum\_probability quedaría como [0.33, 0.66, 0.99]. Ahora, considera que el número aleatorio generado, r, es 0.99314. Para este valor de r, queremos que se elija el último elemento de la lista de sucesos. Puedes argumentar que esto no es exactamente correcto porque el último evento tiene una probabilidad superior al 33% de ser seleccionado. Según la condición de ➌, no hay ningún elemento en sum\_probability que sea mayor que r; por lo tanto, la función no devolvería ningún índice. La sentencia ➍ se encarga de esto y devuelve el último índice.

Si llamas a la función dispense() para simular un gran número de billetes de dólar desembolsados por el cajero automático, verás que la relación entre el número de veces que aparece cada billete obedece fielmente a la probabilidad especificada. Esta técnica nos resultará útil para crear *fractales* en el próximo capítulo.

[anterior](ch05_2.html)[Subtema 3 de 5: (Ver todo)](ch05.html)[siguiente](ch05_4.html)