Capítulo 5: Jugar con conjuntos y probabilidad

## **5** Jugar con conjuntos y probabilidad



En este capítulo, empezaremos aprendiendo cómo podemos hacer que nuestros programas comprendan y manipulen conjuntos de números. Luego veremos cómo los conjuntos pueden ayudarnos a entender conceptos básicos de probabilidad. Por último, aprenderemos a generar números aleatorios para simular sucesos aleatorios. ¡Vamos a empezar!

### **¿Qué es un conjunto?**

Un *conjunto* es una colección de objetos distintos, a menudo llamados *elementos* o *miembros*. Dos características de un conjunto lo diferencian de cualquier colección de objetos. Un conjunto está "bien definido", lo que significa que la pregunta "¿Está un objeto concreto en esta colección?" siempre tiene una respuesta clara de sí o no, normalmente basada en una regla o en algún criterio dado. La segunda característica es que no hay dos miembros iguales en un conjunto. Un conjunto puede contener cualquier cosa: números, personas, cosas, palabras, etc.

Repasemos algunas propiedades básicas de los conjuntos mientras aprendemos a trabajar con conjuntos en Python utilizando SymPy.

#### ***Construcción de conjuntos***

En notación matemática, un conjunto se representa escribiendo sus miembros entre llaves. Por ejemplo, {2, 4, 6} representa un conjunto con 2, 4 y 6 como miembros. Para crear un conjunto en Python, podemos utilizar la clase FiniteSet del paquete sympy, como se indica a continuación:

>>> from sympy import FiniteSet  
>>> s = FiniteSet(2, 4, 6)  
>>> s  
{2, 4, 6}

Aquí, primero importamos la clase FiniteSet de SymPy y luego creamos un objeto de esta clase pasando los miembros del conjunto como argumentos. Asignamos la etiqueta s al conjunto que acabamos de crear.

Podemos almacenar distintos tipos de números -incluidos enteros, números de coma flotante y fracciones- en el mismo conjunto:

>>> from sympy import FiniteSet  
>>> from fractions import Fraction  
>>> s = FiniteSet(1, 1.5, Fraction(1, 5))  
>>> s  
{1/5, 1, 1.5}

La *cardinalidad* de un conjunto es el número de miembros del conjunto, que puedes averiguar utilizando la función len():

>>> s = FiniteSet(1, 1.5, 3)  
>>> len(s)  
3

##### **Comprobar si un número está en un conjunto**

Para comprobar si un número es miembro de un conjunto existente, utiliza el operador in. Este operador pregunta a Python: "¿Este número está en este conjunto?". Devuelve True si el número pertenece al conjunto y False en caso contrario. Si, por ejemplo, quisiéramos comprobar si el 4 está en el conjunto anterior, haríamos lo siguiente:

>>> 4 in s  
False

Como el 4 no está presente en el conjunto, el operador devuelve False.

##### **Crear un conjunto vacío**

Si quieres crear un conjunto *vacío*, que es un conjunto que no tiene elementos ni miembros, crea un objeto FiniteSet sin pasarle ningún argumento. El resultado es un objeto EmptySet:

>>> s = FiniteSet()  
>>> s  
EmptySet()

##### **Crear conjuntos a partir de listas o tuplas**

También puedes crear un conjunto pasando una lista o tupla de miembros del conjunto como argumento a FiniteSet:

>>> members = [1, 2, 3]  
>>> s = FiniteSet(\*members)  
>>> s  
{1, 2, 3}

Aquí, en lugar de pasar los miembros del conjunto directamente a FiniteSet, primero los almacenamos en una lista, a la que llamamos members. Después, pasamos la lista a FiniteSet utilizando esta sintaxis especial de Python, que básicamente se traduce en crear un objeto FiniteSet que pasa los miembros de la lista como argumentos separados y no como una lista. Es decir, esta forma de crear un objeto FiniteSet es equivalente a FiniteSet(1, 2, 3). Utilizaremos esta sintaxis cuando se calculen los miembros del conjunto en tiempo de ejecución.

##### **Repetición y orden de los conjuntos**

Los conjuntos en Python (como los conjuntos matemáticos) ignoran cualquier repetición de un miembro, y no llevan la cuenta del orden de los miembros del conjunto. Por ejemplo, si creas un conjunto a partir de una lista que tiene varias instancias de un número, el número se añade al conjunto sólo una vez, y las demás instancias se descartan:

>>> from sympy import FiniteSet  
>>> members = [1, 2, 3, 2]  
>>> FiniteSet(\*members)  
{1, 2, 3}

Aquí, aunque hayamos pasado una lista que tenía dos instancias del número 2, el número 2 sólo aparece una vez en el conjunto creado a partir de esa lista.

En las listas y tuplas de Python, cada elemento se almacena en un orden determinado, pero no siempre ocurre lo mismo con los conjuntos. Por ejemplo, podemos imprimir cada miembro de un conjunto iterando a través de él como se indica a continuación:

>>> from sympy import FiniteSet  
>>> s = FiniteSet(1, 2, 3)  
>>> for member in s:  
print(member)  
  
2  
1  
3

Cuando ejecutes este código, los elementos podrían imprimirse en cualquier orden posible. Esto se debe a la forma en que Python almacena los conjuntos: mantiene un registro de los miembros del conjunto, pero no de ningún orden concreto de esos miembros.

Veamos otro ejemplo. Dos conjuntos son *iguales* cuando tienen los mismos elementos. En Python, puedes utilizar el operador de igualdad, ==, para comprobar si dos conjuntos son iguales:

>>> from sympy import FiniteSet  
>>> s = FiniteSet(3, 4, 5)  
>>> t = FiniteSet(5, 4, 3)  
>>> s == t  
True

Aunque los miembros de estos dos conjuntos aparezcan en distinto orden, los conjuntos siguen siendo iguales.

#### ***Subconjuntos, superconjuntos y conjuntos potentes***

Un conjunto, *s*, es *subconjunto* de otro conjunto, *t*, si todos los miembros de *s* son también miembros de *t*. Por ejemplo, el conjunto {1} es subconjunto del conjunto {1, 2}. Puedes comprobar si un conjunto es subconjunto de otro mediante el método is\_subset():

>>> s = FiniteSet(1)  
>>> t = FiniteSet(1,2)  
>>> s.is\_subset(t)  
True  
>>> t.is\_subset(s)  
False

Observa que un conjunto vacío es un subconjunto de todo conjunto. Además, cualquier conjunto es un subconjunto de sí mismo, como puedes ver a continuación:

>>> s.is\_subset(s)  
True  
>>> t.is\_subset(t)  
True

Del mismo modo, se dice que un conjunto, *t*, es *superconjunto* de otro conjunto, *s*, si *t* contiene todos los miembros contenidos en *s*. Puedes comprobar si un conjunto es superconjunto de otro utilizando el método is\_superset():

>>> s.is\_superset(t)  
False  
>>> t.is\_superset(s)  
True

El *conjunto potencia* de un conjunto, *s*, es el conjunto de todos los subconjuntos posibles de *s*. Cualquier conjunto, *s*, tiene precisamente *2|s|* subconjuntos, donde *|s|* es la cardinalidad del conjunto. Por ejemplo, el conjunto {1, 2, 3} tiene una cardinalidad de 3, por lo que tiene23 u 8 subconjuntos: {} (el conjunto vacío), {1}, {2}, {3}, {1, 2}, {2, 3}, {1, 3} y {1, 2, 3}.

El conjunto de todos estos subconjuntos forma el conjunto potencia, y podemos hallar el conjunto potencia utilizando el método powerset():

>>> s = FiniteSet(1, 2, 3)  
>>> ps = s.powerset()  
>>> ps  
{{1}, {1, 2}, {1, 3}, {1, 2, 3}, {2}, {2, 3}, {3}, EmptySet()}

Como el conjunto potencia es un conjunto en sí mismo, puedes hallar su cardinalidad utilizando la función len():

>>> len(ps)  
8

La cardinalidad del conjunto potencia es *2|s|*, que es23 = 8.

Según nuestra definición de subconjunto, dos conjuntos cualesquiera con exactamente los mismos miembros serían subconjuntos y superconjuntos entre sí. Por el contrario, un conjunto, *s*, es un *subconjunto* propio de *t* sólo si todos los miembros de *s* están también en *t* y *t* tiene al menos un miembro que no está en *s*. Así, si *s* = {1, 2, 3}, sólo es un subconjunto propio de *t* si *t* contiene 1, 2 y 3 más al menos un miembro más. Esto también significaría que *t* es un *superconjunto* adecuado de *s*. Puedes utilizar el método is\_proper\_subset() y el método is\_proper\_superset() para comprobar estas relaciones:

>>> from sympy import FiniteSet  
>>> s = FiniteSet(1, 2, 3)  
>>> t = FiniteSet(1, 2, 3)  
>>> s.is\_proper\_subset(t)  
False  
>>> t.is\_proper\_superset(s)  
False

Ahora, si volvemos a crear el conjunto t para incluir otro miembro, s se considerará un subconjunto propio de t y t un superconjunto propio de s:

>>> t = FiniteSet(1, 2, 3, 4)  
>>> s.is\_proper\_subset(t)  
True  
>>> t.is\_proper\_superset(s)  
True

**CONJUNTOS NUMÉRICOS COMUNES**

En el primer capítulo aprendimos que existen distintos tipos de números: enteros, de coma flotante, fracciones y complejos. Todos estos números forman diferentes conjuntos numéricos, que tienen nombres especiales.

Todos los números enteros positivos y negativos forman el conjunto de los *números enteros*. Todos los números enteros positivos forman el conjunto de los números *naturales* (a veces el 0 se incluye en este conjunto de números aunque no sea positivo, pero a veces no). Esto significa que el conjunto de los números naturales es un subconjunto propio del conjunto de los números enteros.

El conjunto de los *números racionales* incluye cualquier número que pueda expresarse como fracción, lo que incluye todos los números enteros, además de cualquier número con terminación decimal que termine o se repita (incluyendo números como 1/4 o 0,25, y 1/3 o 0,33333 ...). Por el contrario, los números decimales que no se repiten ni terminan se conocen como *números irracionales*. Tanto la raíz cuadrada de 2 como *π* son ejemplos de números irracionales, porque se eternizan sin repetirse.

Si juntas todos los números racionales e irracionales, obtienes el conjunto de los *números reales*. Pero aún mayor es el conjunto de los *números complejos*, que incluye todos los números reales y todos los números con componente imaginario.

Todos estos conjuntos de números son conjuntos infinitos porque tienen infinitos miembros. En cambio, los conjuntos de los que hemos hablado en este capítulo tienen un número finito de miembros, por eso la clase SymPy que utilizamos se llama FiniteSet.

#### ***Operaciones con conjuntos***

Las operaciones con conjuntos, como la unión, la intersección y el producto cartesiano, te permiten combinar conjuntos de determinadas formas metódicas. Estas operaciones con conjuntos son muy útiles en situaciones reales de resolución de problemas, cuando tenemos que considerar varios conjuntos juntos. Más adelante en este capítulo, veremos cómo utilizar estas operaciones para aplicar una fórmula a varios conjuntos de datos y calcular las probabilidades de sucesos aleatorios.

##### **Unión e intersección**

La *unión* de dos conjuntos es un conjunto que contiene todos los miembros *distintos* de los dos conjuntos. En teoría de conjuntos, utilizamos el símbolo ∪ para referirnos a la operación de unión. Por ejemplo, {1, 2} ∪ {2, 3} dará como resultado un nuevo conjunto, {1, 2, 3}. En SymPy, la unión de estos dos conjuntos puede crearse utilizando el método union():

>>> from sympy import FiniteSet  
>>> s = FiniteSet(1, 2, 3)  
>>> t = FiniteSet(2, 4, 6)  
>>> s.union(t)  
{1, 2, 3, 4, 6}

Encontramos la unión de s y t aplicando el método union a s y pasando t como argumento. El resultado es un tercer conjunto con todos los miembros distintos de los dos conjuntos. En otras palabras, cada miembro de este tercer conjunto es miembro de uno o de los dos primeros conjuntos.

La *intersección* de dos conjuntos crea un nuevo conjunto a partir de los elementos comunes a ambos conjuntos. Por ejemplo, la intersección de los conjuntos {1, 2} y {2, 3} dará como resultado un nuevo conjunto con el único elemento común, {2}. Matemáticamente, esta operación se escribe como {1, 2} ∩ {2, 3}.

En SymPy, utiliza el método intersect() para hallar la intersección:

>>> s = FiniteSet(1, 2)  
>>> t = FiniteSet(2, 3)  
>>> s.intersect(t)  
{2}

Mientras que la operación de unión encuentra los miembros que están en uno *u otro* conjunto, la operación de intersección encuentra los elementos que están presentes en ambos. Ambas operaciones también pueden aplicarse a más de dos conjuntos. Por ejemplo, así es como encontrarías la unión de tres conjuntos:

>>> from sympy import FiniteSet  
>>> s = FiniteSet(1, 2, 3)  
>>> t = FiniteSet(2, 4, 6)  
>>> u = FiniteSet(3, 5, 7)  
>>> s.union(t).union(u)  
{1, 2, 3, 4, 5, 6, 7}

Del mismo modo, así es como encontrarías la intersección de tres conjuntos:

>>> s.intersect(t).intersect(u)  
EmptySet()

La intersección de los conjuntos s, t, y u resulta ser un conjunto vacío porque no hay elementos que compartan los tres conjuntos.

##### **Producto cartesiano**

El producto *cartesiano* de dos conjuntos crea un conjunto formado por todas las parejas posibles formadas tomando un elemento de cada conjunto. Por ejemplo, el producto cartesiano de los conjuntos {1, 2} y {3, 4} es {(1, 3), (1, 4), (2, 3), (2, 4)}. En SymPy, puedes hallar el producto cartesiano de dos conjuntos utilizando simplemente el operador multiplicación:

>>> from sympy import FiniteSet  
>>> s = FiniteSet(1, 2)  
>>> t = FiniteSet(3, 4)  
>>> p = s\*t  
>>> p  
{1, 2} x {3, 4}

Éste toma el producto cartesiano de los conjuntos s y t y lo almacena como p. Para ver realmente cada par en ese producto cartesiano, podemos iterar a través de ellos e imprimirlos como sigue:

>>> for elem in p:  
print(elem)  
(1, 3)  
(1, 4)  
(2, 3)  
(2, 4)

Cada elemento del producto es una tupla formada por un miembro del primer conjunto y un miembro del segundo conjunto.

La cardinalidad del producto cartesiano es el producto de la cardinalidad de los conjuntos individuales. Podemos demostrarlo en Python:

>>> len(p) == len(s)\*len(t)  
True

Si aplicamos el operador exponencial (\*\*) a un conjunto, obtendremos el producto cartesiano de ese conjunto multiplicado por sí mismo el número de veces especificado.

>>> from sympy import FiniteSet  
>>> s = FiniteSet(1, 2)  
>>> p = s\*\*3  
>>> p  
{1, 2} x {1, 2} x {1, 2}

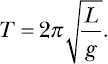
Aquí, por ejemplo, elevamos el conjunto s a la potencia de 3. Como estamos tomando el producto cartesiano de tres conjuntos, esto nos da un conjunto de todos los posibles tripletes que contienen un miembro de cada conjunto:

>>> for elem in p:  
print(elem)  
(1, 1, 1)  
(1, 1, 2)  
(1, 2, 1)  
(1, 2, 2)  
(2, 1, 1)  
(2, 1, 2)  
(2, 2, 1)  
(2, 2, 2)

Encontrar el producto cartesiano de conjuntos es útil para encontrar todas las combinaciones posibles de los miembros del conjunto, que exploraremos a continuación.

##### **Aplicación de una fórmula a múltiples conjuntos de variables**

Considera un péndulo simple de longitud *L*. El *período de tiempo*, *T*, de este péndulo -es decir, la cantidad de tiempo que tarda el péndulo en completar una oscilación completa- viene dado por la fórmula



Aquí, *π* es la constante matemática, *pi*, y *g* es la aceleración gravitatoria local, que es de unos 9,8 m/s2 en la Tierra. Como *π* y *g* son constantes, la longitud, *L*, es la única variable del lado derecho de la ecuación que no tiene un valor constante.

Si quisieras ver cómo varía el período de tiempo de un péndulo simple con su longitud, asumirías distintos valores para la longitud y medirías el período de tiempo correspondiente a cada uno de estos valores utilizando la fórmula. Un experimento típico de bachillerato consiste en comparar el periodo de tiempo que obtienes utilizando la fórmula anterior, que es el resultado teórico, con el que mides en el laboratorio, que es el resultado experimental. Por ejemplo, elijamos cinco valores diferentes: 15, 18, 21, 22,5 y 25 (todos ellos expresados en centímetros). Con Python, podemos escribir un programa rápido que agilizará los cálculos de los resultados teóricos:

from sympy import FiniteSet, pi  
➊ def time\_period(length):  
g = 9.8  
T = 2\*pi\*(length/g)\*\*0.5  
return T  
  
if \_\_name\_\_ == '\_\_main\_\_':  
➋ L = FiniteSet(15, 18, 21, 22.5, 25)  
for l in L:  
➌ t = time\_period(l/100)  
print('Length: {0} cm Time Period: {1:.3f} s'. format(float(l), float(t)))

Primero definimos la función time\_period en ➊. Esta función aplica la fórmula mostrada anteriormente a una longitud dada, que se pasa como length. A continuación, nuestro programa define un conjunto de longitudes en ➋ y aplica la función time\_period a cada valor en ➌. Observa que cuando pasamos los valores de longitud a time\_period, los dividimos por 100. Esta operación convierte las longitudes de centímetros a metros para que coincidan con la unidad de aceleración gravitatoria, que se expresa en unidades de metros/segundo2. Por último, imprimimos el periodo de tiempo calculado. Cuando ejecutes el programa, verás la siguiente salida:

Length: 15.0 cm Time Period: 0.777 s  
Length: 18.0 cm Time Period: 0.852 s  
Length: 21.0 cm Time Period: 0.920 s  
Length: 22.5 cm Time Period: 0.952 s  
Length: 25.0 cm Time Period: 1.004 s

##### **Gravedad diferente, resultados diferentes**

Ahora, imagina que realizamos este experimento en tres lugares distintos: mi ubicación actual, Brisbane, Australia; el Polo Norte; y el ecuador. La fuerza de la gravedad varía ligeramente en función de la latitud de tu ubicación: es un poco menor (aproximadamente 9,78 m/s2) en el ecuador y mayor (9,83 m/s2) en el Polo Norte. Esto significa que podemos tratar la fuerza de la gravedad como una variable en nuestra fórmula, en lugar de como una constante, y calcular los resultados para tres valores diferentes de aceleración gravitatoria: {9.8, 9.78, 9.83}.

Si queremos calcular el período de un péndulo para cada una de nuestras cinco longitudes en cada uno de estos tres lugares, una forma sistemática de calcular todas estas combinaciones de los valores es tomar el producto cartesiano, como se muestra en el siguiente programa:

from sympy import FiniteSet, pi  
  
def time\_period(length, g):  
  
T = 2\*pi\*(length/g)\*\*0.5  
return T  
  
if \_\_name\_\_ == '\_\_main\_\_':  
  
L = FiniteSet(15, 18, 21, 22.5, 25)  
g\_values = FiniteSet(9.8, 9.78, 9.83)  
➊ print('{0:^15}{1:^15}{2:^15}'.format('Length(cm)', 'Gravity(m/s^2)', 'Time Period(s)'))  
➋ for elem in L\*g\_values:  
➌ l = elem[0]  
➍ g = elem[1]  
t = time\_period(l/100, g)  
  
➎ print('{0:^15}{1:^15}{2:^15.3f}'.format(float(l), float(g), float(t)))

En ➋, tomamos el producto cartesiano de nuestros dos conjuntos de variables, L y g\_values, e iteramos por cada combinación de valores resultante para calcular el periodo de tiempo. Cada combinación se representa como una tupla, y de cada tupla extraemos el primer valor, la longitud, en ➌ y el segundo valor, la gravedad, en ➍. A continuación, igual que antes, llamamos a la función time\_period() con estas dos etiquetas como parámetros, e imprimimos los valores de longitud (l), gravedad (g), y el periodo de tiempo correspondiente (T).

La salida se presenta en una tabla para facilitar su seguimiento. La tabla se formatea mediante las sentencias print en ➊ y ➎. La cadena de formato {0:^15} {1:^15}{2:^15.3f} crea tres campos, cada uno de 15 espacios de ancho, y el símbolo ^ centra cada entrada en cada campo. En el último campo de la sentencia print en ➎, '.3f' limita a tres el número de dígitos después del punto decimal.

Cuando ejecutes el programa, verás la siguiente salida:

Length(cm) Gravity(m/s^2) Time Period(s)  
15.0 9.78 0.778  
15.0 9.8 0.777  
15.0 9.83 0.776  
  
18.0 9.78 0.852  
18.0 9.8 0.852  
18.0 9.83 0.850  
21.0 9.78 0.921  
21.0 9.8 0.920  
21.0 9.83 0.918  
22.5 9.78 0.953  
22.5 9.8 0.952  
22.5 9.83 0.951  
25.0 9.78 1.005  
25.0 9.8 1.004  
25.0 9.83 1.002

Este experimento presenta un escenario sencillo en el que necesitas todas las combinaciones posibles de los elementos de varios conjuntos (o un grupo de números). En este tipo de situación, el producto cartesiano es exactamente lo que necesitas.

### **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}.

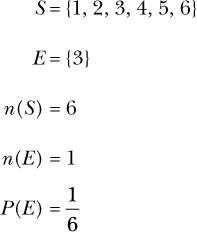
**Evento** Un *evento* es un conjunto de resultados de los que queremos calcular la probabilidad y que forman un *subconjunto* del espacio muestral. Por ejemplo, podemos querer conocer la probabilidad de un resultado concreto, como sacar un 3, o la probabilidad de un conjunto de resultados múltiples, 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.

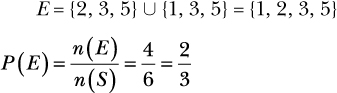
#### ***Probabilidad del suceso A o del suceso B***

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.

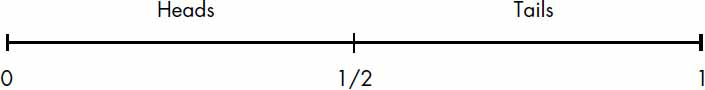
En ➎, calculamos el conjunto de sucesos, e, a partir de la lista success\_rolls que construimos anteriormente y, a continuación, devolvemos la probabilidad de alcanzar la puntuación objetivo.

#### ***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*.

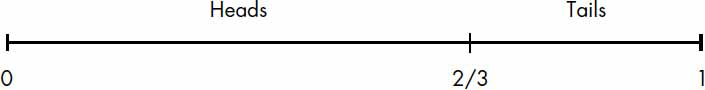
Antes de escribir el programa para hacerlo, repasaremos la idea que hay detrás.

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.



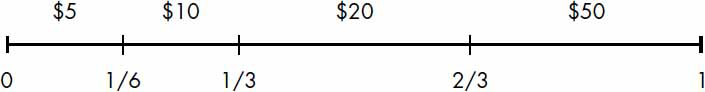
*Figura 5-2: Una recta numérica con una longitud de 1 dividida en dos intervalos desiguales correspondientes a la probabilidad de cara o cruz en el lanzamiento sesgado de una moneda*

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.

### **Lo que has aprendido**

En este capítulo, has empezado por aprender a representar un conjunto en Python. Después, discutimos los distintos conceptos de conjunto y aprendiste sobre la unión, la intersección y el producto cartesiano de conjuntos. Aplicaste algunos de los conceptos de conjunto para explorar los fundamentos de la probabilidad y, por último, aprendiste a simular sucesos aleatorios uniformes y no uniformes en tus programas.

### **Retos de programación**

A continuación, tienes que resolver algunos retos de programación que te darán la oportunidad de aplicar lo que has aprendido en este capítulo.

#### ***#nº 1: Utilizar diagramas de Venn para visualizar relaciones entre conjuntos***

Un *diagrama* de Venn es una forma sencilla de ver gráficamente la relación entre conjuntos. Nos dice cuántos elementos son comunes entre los dos conjuntos, cuántos elementos están sólo en un conjunto y cuántos elementos no están en ninguno de los dos conjuntos. Consideremos un conjunto, *A*, que representa el conjunto de los números impares positivos menores que 20 -es decir, *A* = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19}- y consideremos otro conjunto, *B*, que representa el conjunto de los números primos menores que 20 -es decir, *B* = {2, 3, 5, 7, 11, 13, 17, 19}. Podemos dibujar diagramas de Venn con Python utilizando el paquete matplotlib\_venn (consulta el [Apéndice A](app01.html#app01) para ver las instrucciones de instalación de este paquete). Una vez instalado, puedes dibujar el diagrama de Venn como se indica a continuación:

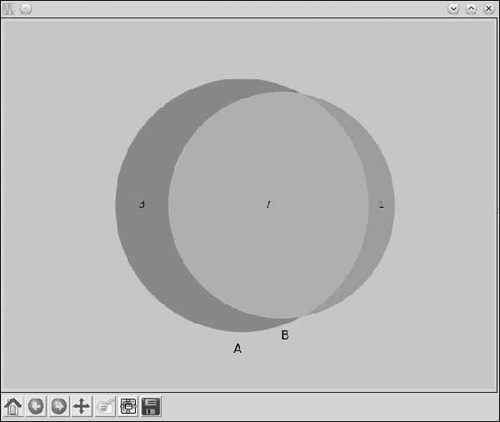
'''  
Draw a Venn diagram for two sets  
'''  
  
from matplotlib\_venn import venn2  
import matplotlib.pyplot as plt  
from sympy import FiniteSet  
  
def draw\_venn(sets):  
  
venn2(subsets=sets)  
plt.show()  
  
if \_\_name\_\_ == '\_\_main\_\_':  
  
s1 = FiniteSet(1, 3, 5, 7, 9, 11, 13, 15, 17, 19)  
s2 = FiniteSet(2, 3, 5, 7, 11, 13, 17, 19)  
  
draw\_venn([s1, s2])

Una vez que importamos todos los módulos y funciones necesarios (la función venn2(), matplotlib.pyplot, y la clase FiniteSet ), todo lo que tenemos que hacer es crear los dos conjuntos y luego llamar a la función venn2(), utilizando el argumento subsets palabra clave para especificar los conjuntos como una tupla.

[La Figura 5-4](ch05.html#ch5fig4) muestra el diagrama de Venn creado por el programa anterior. Los conjuntos *A* y *B* comparten siete elementos comunes, por lo que se escribe 7 en el área común. Cada uno de los conjuntos también tiene elementos únicos, por lo que el número de elementos únicos -3 y 1, respectivamente- se escribe en las áreas individuales. Las etiquetas debajo de los dos conjuntos se muestran como *A* y *B*. Puedes especificar tus propias etiquetas utilizando el argumento de la palabra clave set\_labels:

>>> venn2(subsets=(a,b), set\_labels=('S', 'T'))

Esto cambiaría las etiquetas de los conjuntos a S y T.

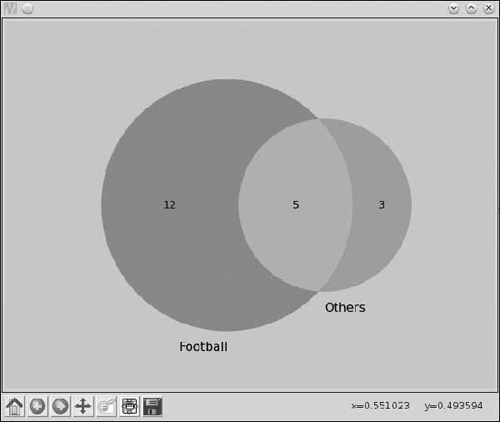


*Figura 5-4: Diagrama de Venn que muestra la relación entre dos* conjuntos, A *y* B

Para tu reto, imagina que has creado un cuestionario online en el que planteas a tus compañeros la siguiente pregunta: *¿Practicas fútbol, otro deporte o ningún deporte?* Una vez que tengas los resultados, crea un archivo CSV, *deportes.csv*, como el siguiente

StudentID,Football,Others  
1,1,0  
2,1,1  
3,0,1  
--snip--

Crea 20 filas de este tipo para los 20 alumnos de tu clase. La primera columna es el ID del alumno (la encuesta no es anónima), la segunda columna tiene un 1 si el alumno ha marcado "fútbol" como el deporte que más le gusta practicar, y la tercera columna tiene un 1 si el alumno practica cualquier otro deporte o ninguno. Escribe un programa para crear un diagrama de Venn que represente los resultados resumidos de la encuesta, como se muestra en la [Figura 5-5](ch05.html#ch5fig5).



*Figura 5-5: Diagrama de Venn que muestra el número de alumnos a los que les gusta jugar al fútbol y el número de alumnos a los que les gusta practicar otros deportes*

Dependiendo de los datos del archivo *sports.csv* que hayas creado, los números de cada conjunto variarán. La siguiente función lee un archivo CSV y devuelve dos listas correspondientes a los ID de los alumnos que practican fútbol y otros deportes:

def read\_csv(filename):  
football = []  
others = []  
with open(filename) as f:  
reader = csv.reader(f)  
next(reader)  
for row in reader:  
if row[1] == '1':  
football.append(row[0])  
if row[2] == '1':  
others.append(row[0])  
  
return football, others

#### ***#2: Ley de los grandes números***

Nos hemos referido a la tirada de un dado y al lanzamiento de una moneda como dos ejemplos de sucesos aleatorios que podemos simular utilizando números aleatorios. Hemos utilizado el término *suceso* para referirnos a la aparición de un número determinado en la tirada de un dado o a la aparición de cara o cruz en el lanzamiento de una moneda, y cada suceso tiene un valor de probabilidad asociado. En probabilidad, una *variable aleatoria -denotada normalmente*por *X- describe*un suceso. Por ejemplo, *X* = 1 describe el suceso de que aparezca 1 en la tirada de un dado, y *P(X* = 1) describe la probabilidad asociada. Hay dos tipos de variables aleatorias: (1) las variables aleatorias *discretas*, que sólo toman valores integrales y son el único tipo de variables aleatorias que veremos en este capítulo, y (2) las variables aleatorias *continuas*, que -como su nombre indica- pueden tomar cualquier valor real.

La *expectativa*, *E*, de una variable aleatoria discreta es el equivalente de la media o promedio que conocimos en el [Capítulo 3](ch03.html#ch03). La expectativa se puede calcular de la siguiente manera:

*E* = *x1P(*x1) + *x2P(*x2) + *x3P(*x3) + ... + *xnP*(*xn*)

Así, para un dado de seis caras, el *valor esperado* de una tirada puede calcularse así:

>>> e = 1\*(1/6) + 2\*(1/6) + 3\*(1/6) + 4\*(1/6) + 5\*(1/6) + 6\*(1/6)  
>>> e  
3.5

Según la *ley de los grandes números*, el valor medio de los resultados en múltiples ensayos se aproxima al valor esperado a medida que aumenta el número de ensayos. Tu reto en esta tarea es verificar esta ley al lanzar un dado de seis caras para el siguiente número de ensayos 100, 1000, 10000, 100000 y 500000. Aquí tienes un ejemplo de ejecución de tu programa completo:

Expected value: 3.5  
Trials: 100 Trial average 3.39  
Trials: 1000 Trial average 3.576  
Trials: 10000 Trial average 3.5054  
Trials: 100000 Trial average 3.50201  
Trials: 500000 Trial average 3.495568

#### ***#3: ¿Cuántas tiradas antes de quedarte sin dinero?***

Consideremos un juego sencillo jugado con un lanzamiento de moneda justo. Un jugador gana 1$ por cara y pierde 1,50$ por cruz. El juego termina cuando el saldo del jugador llega a 0 $. Dada una determinada cantidad inicial especificada por el usuario como entrada, tu reto es escribir un programa que simule este juego. Supón que el ordenador -tu oponente aquí- tiene una reserva de dinero ilimitada. He aquí una posible sesión de juego:

Enter your starting amount: 10  
Tails! Current amount: 8.5  
Tails! Current amount: 7.0  
Tails! Current amount: 5.5  
Tails! Current amount: 4.0  
Tails! Current amount: 2.5  
Heads! Current amount: 3.5  
Tails! Current amount: 2.0  
Tails! Current amount: 0.5  
Tails! Current amount: -1.0  
Game over :( Current amount: -1.0. Coin tosses: 9

#### ***#4: Barajar una baraja de cartas***

Considera una baraja estándar de 52 cartas. Tu reto aquí es escribir un programa para simular el barajado de esta baraja. Para simplificar la implementación, te sugiero que utilices los números enteros 1, 2, 3, ..., 52 para representar la baraja. Cada vez que ejecutes el programa, debe salir una baraja barajada, en este caso, una lista de enteros barajados.

Aquí tienes una posible salida de tu programa:

[3, 9, 21, 50, 32, 4, 20, 52, 7, 13, 41, 25, 49, 36, 23, 45, 1, 22, 40, 19, 2,  
35, 28, 30, 39, 44, 29, 38, 48, 16, 15, 18, 46, 31, 14, 33, 10, 6, 24, 5, 43,  
47, 11, 34, 37, 27, 8, 17, 51, 12, 42, 26]

El módulo random de la biblioteca estándar de Python tiene una función, shuffle(), para esta misma operación:

>>> import random  
>>> x = [1, 2, 3, 4]  
➊ >>> random.shuffle(x)  
>>> x  
[4, 2, 1, 3]

Crea una lista, x, formada por los números [1, 2, 3, 4]. A continuación, llama a la función shuffle() ➊, pasando esta lista como argumento. Verás que los números de x se han barajado. Observa que la lista se baraja "en su sitio". Es decir, se pierde el orden original.

Pero, ¿y si quisieras utilizar este programa en un juego de cartas? En ese caso, no basta con obtener la lista de enteros barajada. También necesitarás una forma de volver a asignar los enteros al palo y rango específicos de cada carta. Una forma de hacerlo es crear una clase Python que represente una sola carta:

class Card:  
def \_\_init\_\_(self, suit, rank):  
self.suit = suit  
self.rank = rank

Para representar el as de tréboles, crea un objeto tarjeta, card1 = Card('clubs', 'ace'). A continuación, haz lo mismo con las demás cartas. A continuación, crea una lista formada por cada uno de los objetos carta y baraja esta lista. El resultado será una baraja de cartas barajada de la que también sabrás el palo y el rango de cada carta. El resultado del programa debería ser algo parecido a esto:

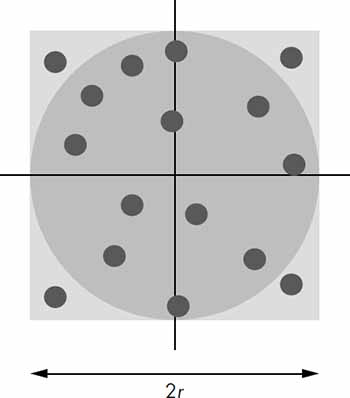
10 of spades  
6 of clubs  
jack of spades  
9 of spades

#### ***#5: Estimación del área de un círculo***

Considera una diana con un círculo de radio *r* inscrito en un cuadrado de lado *2r*. Supongamos que empiezas a lanzarle un gran número de dardos. Algunos de ellos caerán en la diana dentro del círculo -digamos, *N- y*otros fuera de él -digamos, *M*. Si consideramos la fracción de dardos que caen dentro del círculo,

image

entonces el valor de *f* × *A*, donde *A* es el área del cuadrado, sería aproximadamente igual al área del círculo (ver [Figura 5-6](ch05.html#ch5fig6)). Los dardos están representados por los pequeños puntos circulares de la figura. Denominaremos área estimada al valor de *f* × *A*. El área real es, por supuesto, πr2.



*Figura 5-6: Un círculo de radio* r *inscrito en un tablero cuadrado de lado 2r. Los puntos representan dardos lanzados al azar contra el tablero.*

Como parte de este reto, escribe un programa que encuentre el área estimada de un círculo, dado cualquier radio, utilizando este enfoque. El programa debe imprimir el área estimada del círculo para tres valores diferentes del número de dardos:103,105 y106. ¡Son muchos dardos! Verás que al aumentar el número de dardos el área estimada se aproxima al área real. Aquí tienes un ejemplo de salida de la solución completada:

Radius: 2  
Area: 12.566370614359172, Estimated (1000 darts): 12.576  
Area: 12.566370614359172, Estimated (100000 darts): 12.58176  
Area: 12.566370614359172, Estimated (1000000 darts): 12.560128

El lanzamiento de dardos puede simularse mediante una llamada a la función random.uniform(a, b), que devolverá un número aleatorio entre *a* y *b*. En este caso, utiliza los valores *a* = 0, *b* = *2r* (el lado del cuadrado).

##### **Estimación del valor de Pi**

Considera de nuevo [la Figura 5-6](ch05.html#ch5fig6). El área del cuadrado es 4r2, y el área del círculo inscrito es πr2. Si dividimos el área del círculo por el área del cuadrado, obtenemos *π/4*. La fracción *f* que hemos calculado antes

image

es por tanto una aproximación de *π/4*, lo que a su vez significa que el valor de

image

debe ser próximo al valor de *π*. Tu siguiente reto es escribir un programa que estime el valor de *π* suponiendo cualquier valor para el radio. A medida que aumentes el número de dardos, el valor estimado de *π* debería acercarse al valor conocido de la constante.