Capítulo 5: Jugar con conjuntos y probabilidad

### **¿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}

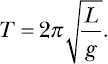
Aquí, por ejemplo, elevamos el conjunto s a la potencia de 3. Como estamos tomando el producto cartesiano de tres conjuntos, esto nos da un conjunto de todos los posibles tripletes que contienen un miembro de cada conjunto:

>>> 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.

[anterior](ch05_1.html)[Subtema 2 de 5: (Ver todo)](ch05.html)[siguiente](ch05_3.html)