Capítulo 4: Álgebra y matemáticas simbólicas con SymPy

## **4** **Ál**gebra y Matemáticas Simbólicas con SymPy



Los problemas matemáticos y las soluciones de nuestros programas hasta ahora han implicado la manipulación de números. Pero hay otra forma de enseñar, aprender y practicar las matemáticas, y es en términos de símbolos y operaciones entre ellos. Piensa en todas las *"x"*e *"y"*de un típico problema de álgebra. Nos referimos a este tipo de matemáticas como *matemáticas simbólicas*. Seguro que recuerdas aquellos temidos problemas de "factorizar x3 + 3x2 + *3x* + 1" en tu clase de matemáticas. No temas más, porque en este capítulo aprenderemos a escribir programas capaces de resolver esos problemas y muchos más. Para ello, utilizaremos *SymPy, una*biblioteca de Python que te permite escribir expresiones que contengan símbolos y realizar operaciones con ellos. Como se trata de una biblioteca de terceros, tendrás que instalarla antes de poder utilizarla en tus programas. Las instrucciones de instalación se describen en [el Apéndice A](app01.html#app01).

### **Definición de símbolos y operaciones simbólicas**

*Los símbolos* forman los bloques de construcción de la matemática simbólica. El término *símbolo* no es más que un nombre general para las *x*, *y*, *as*y *b*que utilizas en ecuaciones y expresiones algebraicas. Crear y utilizar símbolos nos permitirá hacer las cosas de forma diferente a como las hacíamos antes. Considera las siguientes afirmaciones:

>>> x = 1  
>>> x + x + 1  
3

Aquí creamos una etiqueta, x, para referirnos al número 1. Luego, cuando escribimos la expresión x + x + 1, se evalúa por nosotros, y el resultado es 3. ¿Y si quisieras el resultado en términos del símbolo *x*? Es decir, ¿si en lugar de 3, quisieras que Python te dijera que el resultado es*2x* + 1? No podrías escribir x + x + 1 *sin* la sentencia x = 1, porque Python no sabría a qué se refiere x.

SymPy nos permite escribir programas en los que podemos expresar y evaluar expresiones matemáticas en términos de dichos símbolos. Para utilizar un símbolo en tu programa, tienes que crear un objeto de la clase Symbol, de la siguiente manera:

>>> from sympy import Symbol  
>>> x = Symbol('x')

Primero, importamos la clase Symbol de la biblioteca sympy. A continuación, creamos un objeto de esta clase pasando 'x' como parámetro. Observa que este 'x' se escribe como una cadena entre comillas. Ahora podemos definir expresiones y ecuaciones en términos de este símbolo. Por ejemplo, aquí tienes la expresión anterior:

>>> from sympy import Symbol  
>>> x = Symbol('x')  
>>> x + x + 1  
2\*x + 1

Ahora el resultado se da en términos del símbolo *x*. En la expresión x = Symbol('x'), el x de la izquierda es la etiqueta Python. Es el mismo tipo de etiqueta que hemos utilizado antes, sólo que esta vez se refiere al símbolo *x* en lugar de a un número -más concretamente, a un objeto Symbol que representa el símbolo 'x'. Esta etiqueta tampoco tiene que coincidir necesariamente con el símbolo; podríamos haber utilizado una etiqueta como a o var1. Por lo tanto, no hay ningún problema en escribir las declaraciones anteriores de la siguiente manera:

>>> a = Symbol('x')  
>>> a + a + 1  
2\*x + 1

Sin embargo, utilizar una etiqueta que no coincida puede resultar confuso, por lo que te recomiendo que elijas una etiqueta que tenga la misma letra que el símbolo al que se refiere.

**ENCONTRAR EL SÍMBOLO REPRESENTADO POR UN OBJETO SÍMBOLO**

Para cualquier objeto Symbol, su atributo name es una cadena que es el símbolo real que representa:

>>> x = Symbol('x')  
>>> x.name  
'x'  
>>> a = Symbol('x')  
>>> a.name  
'x'

Puedes utilizar .name en una etiqueta para recuperar el símbolo que almacena.

Para que quede claro, el símbolo que crees debe especificarse como una cadena. Por ejemplo, no puedes crear el símbolo *x* utilizando x = Symbol(x)-debes definirlo como x = Symbol('x').

Para definir varios símbolos, puedes crear objetos Symbol separados o utilizar la función symbols() para definirlos de forma más concisa. Supongamos que quieres utilizar tres símbolos *-x*, *y* y *z- en*tu programa. Podrías definirlos individualmente, como hemos hecho antes:

>>> x = Symbol('x')  
>>> y = Symbol('y')  
>>> z = Symbol('z')

Pero un método más breve sería utilizar la función symbols() para definir los tres a la vez:

>>> from sympy import symbols  
>>> x,y,z = symbols('x,y,z')

Primero, importamos la función symbols() de SymPy. Luego, la llamamos con los tres símbolos que queremos crear, escritos como una cadena con comas separándolas. Una vez ejecutada esta sentencia, x, y, y z harán referencia a los tres símbolos 'x', 'y', y 'z'.

Una vez que hayas definido los símbolos, puedes realizar operaciones matemáticas básicas con ellos, utilizando los mismos operadores que aprendiste en [el Capítulo 1](ch01.html#ch01) (+, -, /, \*, y \*\*). Por ejemplo, puedes hacer lo siguiente:

>>> from sympy import Symbol  
>>> x = Symbol('x')  
>>> y = Symbol('y')  
  
>>> s = x\*y + x\*y  
>>> s  
2\*x\*y

Veamos si podemos encontrar el producto de x(x + x):

>>> p = x\*(x + x)  
>>> p  
2\*x\*\*2

SymPy realizará automáticamente estos cálculos sencillos de suma y multiplicación, pero si introducimos una expresión más compleja, no cambiará. Veamos qué ocurre cuando introducimos la expresión (x + 2)\*(x + 3):

>>> p = (x + 2)\*(x + 3)  
>>> p  
(x + 2)\*(x + 3)

Es posible que esperaras que SymPy lo multiplicara todo e imprimiera x\*\*2 + 5\*x + 6. En lugar de eso, la expresión se imprimió exactamente como la introdujimos. SymPy sólo simplifica automáticamente las expresiones más básicas y deja que el programador exija explícitamente la simplificación en casos como el anterior. Si quieres multiplicar la expresión para obtener la versión expandida, tendrás que utilizar la función expand(), que veremos dentro de un momento.

### **Trabajar con expresiones**

Ahora que sabemos cómo definir nuestras propias expresiones simbólicas, vamos a aprender más sobre cómo utilizarlas en nuestros programas.

#### ***Factorizar y expandir expresiones***

La función factor() descompone una expresión en sus factores, y la función expand() expande una expresión, expresándola como una suma de términos individuales. Probemos estas funciones con la identidad algebraica básica x2 - y2 =*(x* + *y*)*(x* - *y*). El lado izquierdo de la identidad es la versión expandida, y el lado derecho representa la factorización correspondiente. Como tenemos dos símbolos en la identidad, crearemos dos objetos Symbol:

>>> from sympy import Symbol  
>>> x = Symbol('x')  
>>> y = Symbol('y')

A continuación, importamos la función factor() y la utilizamos para convertir la versión expandida (en el lado izquierdo de la identidad) en la versión factorizada (en el lado derecho):

>>> from sympy import factor  
>>> expr = x\*\*2 - y\*\*2  
>>> factor(expr)  
(x - y)\*(x + y)

Como era de esperar, obtenemos la versión factorizada de la expresión. Ahora vamos a expandir los factores para recuperar la versión expandida original:

>>> factors = factor(expr)  
>>> expand(factors)  
x\*\*2 - y\*\*2

Almacenamos la expresión factorizada en una nueva etiqueta, factors, y luego llamamos a la función expand() con ella. Al hacerlo, recibimos la expresión original con la que empezamos. Probémoslo con la identidad más complicada x3 + *3x2y* + 3xy2 + y3 =*(x* + *y*)3:

>>> expr = x\*\*3 + 3\*x\*\*2\*y + 3\*x\*y\*\*2 + y\*\*3  
>>> factors = factor(expr)  
>>> factors  
(x + y)\*\*3  
  
>>> expand(factors)  
x\*\*3 + 3\*x\*\*2\*y + 3\*x\*y\*\*2 + y\*\*3

La función factor() es capaz de factorizar la expresión y, a continuación, la función expand() expande la expresión factorizada para volver a la expresión original.

Si intentas factorizar una expresión para la que no hay factorización posible, la función factor() devuelve la expresión original. Por ejemplo, mira lo siguiente:

>>> expr = x + y + x\*y  
>>> factor(expr)  
x\*y + x + y

Del mismo modo, si pasas una expresión a expand() que no puede expandirse más, devuelve la misma expresión.

#### ***Impresión bonita***

Si quieres que las expresiones con las que hemos estado trabajando tengan un aspecto más bonito cuando las imprimas, puedes utilizar la función pprint(). Esta función imprimirá la expresión de forma que se parezca más a como la escribiríamos normalmente en papel. Por ejemplo, aquí tienes una expresión:

>>> expr = x\*x + 2\*x\*y + y\*y

Si la imprimimos como hemos estado haciendo hasta ahora o utilizamos la función print(), este es su aspecto:

>>> expr  
x\*\*2 + 2\*x\*y + y\*\*2

Ahora, utilicemos la función pprint() para imprimir la expresión anterior:

>>> from sympy import pprint  
>>> pprint(expr)  
x2 + 2·x·y + y2

La expresión tiene ahora un aspecto mucho más limpio; por ejemplo, en lugar de tener un montón de feos asteriscos, los exponentes aparecen encima del resto de los números.

También puedes cambiar el orden de los términos al imprimir una expresión. Considera la expresión 1 +*2x* + 2x2:

>>> expr = 1 + 2\*x + 2\*x\*\*2  
>>> pprint(expr)  
2·x2 + 2·x + 1

Los términos están dispuestos en el orden de potencias de *x*, de mayor a menor. Si quieres la expresión en el orden inverso, con la mayor potencia de *x* en último lugar, puedes hacerlo con la función init\_printing(), como se indica a continuación:

>>> from sympy import init\_printing  
>>> init\_printing(order='rev-lex')  
>>> pprint(expr)  
1 + 2·x + 2·x2

Primero se importa la función init\_printing() y se llama con el argumento de palabra clave order='rev-lex'. Esto indica que queremos que SymPy imprima las expresiones de modo que estén en *orden lexicográfico inverso*. En este caso, el argumento de palabra clave indica a Python que imprima primero los términos de menor potencia.

**NOTA**

*Aunque aquí hemos utilizado la* función init\_printing() *para establecer el orden de impresión de las expresiones, esta función puede utilizarse de muchas otras formas para configurar cómo se imprime una expresión. Para conocer más opciones y saber más sobre la impresión en SymPy, consulta la documentación en* [http://docs.sympy.org/latest/tutorial/printing.html.](http://docs.sympy.org/latest/tutorial/printing.html)

Apliquemos lo que hemos aprendido hasta ahora para implementar un programa de impresión de series.

##### **Imprimir una serie**

Considera la siguiente serie:

image

Escribamos un programa que pida al usuario que introduzca un número, *n*, e imprima esta serie para ese número. En la serie, *x* es un símbolo y *n* es un número entero introducido por el usuario del programa. El término *n*de esta serie viene dado por

image

Podemos imprimir esta serie utilizando el siguiente programa:

'''  
Print the series:  
x + x\*\*2 + x\*\*3 + ... + x\*\*n  
\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_  
2 3 n  
'''  
  
from sympy import Symbol, pprint, init\_printing  
def print\_series(n):  
  
# Initialize printing system with reverse order  
init\_printing(order='rev-lex')  
  
x = Symbol('x')  
➊ series = x  
➋ for i in range(2, n+1):  
➌ series = series + (x\*\*i)/i  
pprint(series)  
  
if \_\_name\_\_ == '\_\_main\_\_':  
n = input('Enter the number of terms you want in the series: ')  
➍ print\_series(int(n))

La función print\_series() acepta como parámetro un número entero, n, que es el número de términos de la serie que se imprimirán. Observa que convertimos la entrada en un número entero utilizando la función int() al llamar a la función en ➍. A continuación, llamamos a la función init\_printing() para que la serie se imprima en orden lexicográfico inverso.

En ➊, creamos la etiqueta, series, y establecemos su valor inicial como x. A continuación, definimos un bucle for que iterará sobre los enteros de 2 a n en ➋. Cada vez que el bucle itera, añade cada término a series en ➌, como sigue:

i = 2, series = x + x\*\*2 / 2  
i = 3, series = x + x\*\*2/2 + x\*\*3/3  
  
--snip--

El valor de series comienza siendo simplemente x, pero con cada iteración, x\*\*i/i se añade al valor de series hasta completar la serie que queremos. Aquí puedes ver un buen uso de la suma SymPy. Por último, la función pprint() se utiliza para imprimir la serie.

Cuando ejecutas el programa, te pide que introduzcas un número y luego imprime la serie hasta ese término:

Enter the number of terms you want in the series: 5  
  
x2 x3 x4 x5  
x + -- + -- + -- + --  
2 3 4 5

Pruébalo cada vez con un número diferente de términos. A continuación veremos cómo calcular la suma de esta serie para un determinado valor de *x*.

#### ***Sustitución de valores***

Veamos cómo podemos utilizar SymPy para introducir valores en una expresión algebraica. Esto nos permitirá calcular el valor de la expresión para determinados valores de las variables. Considera la expresión matemática x2 + *2xy* + y2, que puede definirse así:

>>> x = Symbol('x')  
>>> y = Symbol('y')  
>>> x\*x + x\*y + x\*y + y\*y  
x\*\*2 + 2\*x\*y + y\*\*2

Si quieres evaluar esta expresión, puedes sustituir los símbolos por números utilizando el método subs():

➊ >>> expr = x\*x + x\*y + x\*y + y\*y  
>>> res = expr.subs({x:1, y:2})

Primero, creamos una nueva etiqueta para referirnos a la expresión en ➊, y luego llamamos al método subs(). El argumento del método subs() es un *diccionario* Python que contiene las dos etiquetas de símbolos y los valores numéricos que queremos sustituir por cada símbolo. Comprobemos el resultado:

>>> res  
9

También puedes expresar un símbolo en términos de otro y sustituirlo en consecuencia, utilizando el método subs(). Por ejemplo, si supieras que *x* = 1 - *y*, así es como podrías evaluar la expresión anterior:

>>> expr.subs({x:1-y})  
y\*\*2 + 2\*y\*(-y + 1) + (-y + 1)\*\*2

**DICCIONARIOS PYTHON**

Un diccionario es otro tipo de estructura de datos en Python (las listas y las tuplas son otros ejemplos de estructuras de datos, que ya has visto antes). Los diccionarios contienen pares clave-valor dentro de llaves, donde cada clave se empareja con un valor, separados por dos puntos. En el listado de código anterior, hemos introducido el diccionario {x:1, y:2} como argumento del método subs(). Este diccionario tiene dos pares clave-valor -x:1 y y:2, donde x y y son las claves y 1 y 2 son los valores correspondientes. Puedes recuperar un valor de un diccionario introduciendo su clave asociada entre paréntesis, del mismo modo que recuperaríamos un elemento de una lista utilizando su índice. Por ejemplo, aquí creamos un diccionario simple y luego recuperamos el valor correspondiente a key1:

>>> sampledict = {"key1": 5, "key2": 20}  
>>> sampledict["key1"]  
5

Para saber más sobre diccionarios, consulta [el Apéndice B](app02.html#app02).

Si quieres simplificar aún más el resultado -por ejemplo, si hay términos que se anulan entre sí-, podemos utilizar la función simplify() de SymPy, como se indica a continuación:

➊ >>> expr\_subs = expr.subs({x:1-y})  
>>> from sympy import simplify  
➋ >>> simplify(expr\_subs)  
1

En ➊, creamos una nueva etiqueta, expr\_subs, para referirnos al resultado de sustituir *x* = 1 - *y* en la expresión. A continuación, importamos la función simplify() de SymPy y la llamamos en ➋. El resultado resulta ser 1 porque los demás términos de la expresión se anulan entre sí.

Aunque en el ejemplo anterior había una versión simplificada de la expresión, tuviste que pedirle a SymPy que la simplificara utilizando la función simplify(). Una vez más, esto se debe a que SymPy no realiza ninguna simplificación si no se le pide que lo haga.

La función simplify() también puede simplificar expresiones complicadas, como las que incluyen logaritmos y funciones trigonométricas, pero no entraremos en eso aquí.

##### **Calcular el valor de una serie**

Volvamos al programa de impresión de series. Además de imprimir la serie, queremos que nuestro programa sea capaz de hallar el valor de la serie para un valor concreto de *x*. Es decir, ahora nuestro programa tomará dos datos de entrada del usuario: el número de términos de la serie y el valor de *x* para el que se calculará el valor de la serie. A continuación, el programa mostrará tanto la serie como la suma. El siguiente programa amplía el programa de impresión de series para incluir estas mejoras:

'''  
Print the series:  
x + x\*\*2 + x\*\*3 + ... + x\*\*n  
\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_  
2 3 n  
'''  
  
from sympy import Symbol, pprint, init\_printing  
def print\_series(n, x\_value):  
  
# Initialize printing system with reverse order  
init\_printing(order='rev-lex')  
  
x = Symbol('x')  
series = x  
for i in range(2, n+1):  
series = series + (x\*\*i)/i  
  
pprint(series)  
  
# Evaluate the series at x\_value  
➊ series\_value = series.subs({x:x\_value})  
print('Value of the series at {0}: {1}'.format(x\_value, series\_value))  
  
if \_\_name\_\_ == '\_\_main\_\_':  
n = input('Enter the number of terms you want in the series: ')  
➋ x\_value = input('Enter the value of x at which you want to evaluate the series: ')  
  
print\_series(int(n), float(x\_value))

La función print\_series() toma ahora un argumento adicional, x\_value, que es el valor de x para el que debe evaluarse la serie. En ➊, utilizamos el método subs() para realizar la evaluación y la etiqueta series\_value para referirnos al resultado. En la línea siguiente, mostramos el resultado.

La sentencia de entrada adicional en ➋ pide al usuario que introduzca el valor de x utilizando la etiqueta x\_value para referirse a él. Antes de llamar a la función print\_series(), convertimos este valor en su equivalente en coma flotante utilizando la función float().

Si ejecutas ahora el programa, te pedirá las dos entradas e imprimirá la serie y el valor de la serie:

Enter the number of terms you want in the series: 5  
Enter the value of x at which you want to evaluate the series: 1.2  
  
x2 x3 x4 x5  
x + -- + -- + -- + --  
2 3 4 5  
Value of the series at 1.2: 3.51206400000000

En esta ejecución de ejemplo, pedimos cinco términos de la serie, con x fijado en 1,2, y el programa imprime y evalúa la serie.

#### ***Convertir cadenas en expresiones matemáticas***

Hasta ahora, hemos estado escribiendo expresiones individuales cada vez que queríamos hacer algo con ellas. Sin embargo, ¿qué pasaría si quisieras escribir un programa más general que pudiera manipular cualquier expresión proporcionada por el usuario? Para ello, necesitamos una forma de convertir la entrada del usuario, que es una cadena, en algo con lo que podamos realizar operaciones matemáticas. La función sympify() de SymPy nos ayuda a hacer exactamente eso. La función se llama así porque convierte la cadena en un objeto SymPy que permite aplicar las funciones de SymPy a la entrada. Veamos un ejemplo:

➊ >>> from sympy import sympify  
>>> expr = input('Enter a mathematical expression: ')  
Enter a mathematical expression: x\*\*2 + 3\*x + x\*\*3 + 2\*x  
➋ >>> expr = sympify(expr)

Primero importamos la función sympify() en ➊. A continuación, utilizamos la función input() para pedir una expresión matemática como entrada, utilizando la etiqueta expr para referirnos a ella. A continuación, llamamos a la función sympify() con expr como argumento en ➋ y utilizamos la misma etiqueta para referirnos a la expresión convertida.

Puedes realizar varias operaciones con esta expresión. Por ejemplo, probemos a multiplicar la expresión por 2:

>>> 2\*expr  
2\*x\*\*3 + 2\*x\*\*2 + 10\*x

¿Qué ocurre cuando el usuario proporciona una expresión no válida? Veámoslo:

>>> expr = input('Enter a mathematical expression: ')  
Enter a mathematical expression: x\*\*2 + 3\*x + x\*\*3 + 2x  
>>> expr = sympify(expr)  
Traceback (most recent call last):  
File "<pyshell#146>", line 1, in <module>  
expr = sympify(expr)  
File "/usr/lib/python3.3/site-packages/sympy/core/sympify.py", line 180, in sympify  
raise SympifyError('could not parse %r' % a)  
sympy.core.sympify.SympifyError: SympifyError: "could not parse 'x\*\*2 + 3\*x + x\*\*3 + 2x'"

La última línea nos dice que sympify() no es capaz de convertir la expresión de entrada suministrada. Como este usuario no añadió un operador entre 2 y x, SymPy no entiende lo que significa. Tu programa debería esperar una entrada no válida de este tipo e imprimir un mensaje de error si se produce. Veamos cómo podemos hacerlo capturando la excepción SympifyError:

>>> from sympy import sympify  
>>> from sympy.core.sympify import SympifyError  
>>> expr = input('Enter a mathematical expression: ')  
Enter a mathematical expression: x\*\*2 + 3\*x + x\*\*3 + 2x  
>>> try:  
expr = sympify(expr)  
except SympifyError:  
print('Invalid input')  
  
Invalid input

Los dos cambios en el programa anterior son que importamos la clase de excepción SympifyError del módulo sympy.core.sympify y llamamos a la función sympify() en un bloque try...except. Ahora, si se produce una excepción SympifyError, se imprime un mensaje de error.

##### **Multiplicador de expresiones**

Apliquemos la función sympify() para escribir un programa que calcule el producto de dos expresiones:

'''  
Product of two expressions  
'''  
  
from sympy import expand, sympify  
from sympy.core.sympify import SympifyError  
  
def product(expr1, expr2):  
prod = expand(expr1\*expr2)  
print(prod)  
  
if \_\_name\_\_=='\_\_main\_\_':  
➊ expr1 = input('Enter the first expression: ')  
➋ expr2 = input('Enter the second expression: ')  
  
try:  
expr1 = sympify(expr1)  
expr2 = sympify(expr2)  
except SympifyError:  
print('Invalid input')  
else:  
➌ product(expr1, expr2)

En ➊ y ➋, pedimos al usuario que introduzca las dos expresiones. A continuación, las convertimos a una forma comprensible para SymPy utilizando la función sympify()  en un bloque try...except. Si la conversión tiene éxito (indicado por el bloque else ), llamamos a la función product() en ➌. En esta función, calculamos el producto de las dos expresiones y lo imprimimos. Observa cómo utilizamos la función expand() para imprimir el producto de forma que todos sus términos se expresen como una suma de sus términos constituyentes.

Aquí tienes un ejemplo de ejecución del programa:

Enter the first expression: x\*\*2 + x\*2 + x  
Enter the second expression: x\*\*3 + x\*3 + x  
x\*\*5 + 3\*x\*\*4 + 4\*x\*\*3 + 12\*x\*\*2

La última línea muestra el producto de las dos expresiones. La entrada también puede tener más de un símbolo en cualquiera de las expresiones:

Enter the first expression: x\*y+x  
Enter the second expression: x\*x+y  
x\*\*3\*y + x\*\*3 + x\*y\*\*2 + x\*y

### **Resolución de ecuaciones**

La función solve() de SymPy puede utilizarse para encontrar soluciones a ecuaciones. Cuando introduces una expresión con un símbolo que representa una variable, como *x*, solve() calcula el valor de ese símbolo. Esta función siempre realiza su cálculo suponiendo que la expresión que introduces es igual a cero, es decir, imprime el valor que, al sustituir al símbolo, hace que toda la expresión sea igual a cero. Empecemos con la sencilla ecuación *x* - 5 = 7. Si queremos utilizar solve() para hallar el valor de x, primero tenemos que hacer que uno de los lados de la ecuación sea igual a cero*(x* - 5 - 7 = 0). Entonces, estamos preparados para utilizar solve(), como se indica a continuación:

>>> from sympy import Symbol, solve  
>>> x = Symbol('x')  
>>> expr = x - 5 - 7  
>>> solve(expr)  
[12]

Cuando utilizamos solve(), calcula el valor de 'x' como 12 porque ése es el valor que hace que la expresión*(x* - 5 - 7) sea igual a cero.

Observa que el resultado 12 se devuelve en una lista. Una ecuación puede tener varias soluciones; por ejemplo, una ecuación cuadrática tiene dos soluciones. En ese caso, la lista tendrá todas las soluciones como miembros. También puedes pedir a la función solve() que devuelva el resultado de forma que cada miembro sea un diccionario. Cada diccionario está compuesto por el símbolo (nombre de la variable) y su valor (la solución). Esto es especialmente útil cuando resolvemos ecuaciones simultáneas en las que tenemos más de una variable que resolver, porque cuando la solución se devuelve como diccionario, sabemos qué solución corresponde a cada variable.

#### ***Resolver ecuaciones cuadráticas***

En el [Capítulo 1](ch01.html#ch01), encontramos las raíces de la ecuación cuadrática ax2 + *bx* + *c* = 0 escribiendo las fórmulas de las dos raíces y sustituyendo después los valores de las constantes *a*, *b* y *c*. Ahora aprenderemos a utilizar la función solve() de SymPy para encontrar las raíces sin necesidad de escribir las fórmulas. Veamos un ejemplo:

➊ >>> from sympy import solve  
>>> x = Symbol('x')  
➋ >>> expr = x\*\*2 + 5\*x + 4  
➌ >>> solve(expr, dict=True)  
➍ [{x: -4}, {x: -1}]

Primero importamos la función solve() en ➊. A continuación, definimos un símbolo, x, y una expresión correspondiente a la ecuación cuadrática, x\*\*2 + 5\*x + 4, en ➋. A continuación, llamamos a la función solve() con la expresión anterior en ➌. El segundo argumento de la función solve() (dict=True) especifica que queremos que el resultado se devuelva como una lista de diccionarios Python.

Cada solución de la lista devuelta es un diccionario que utiliza el símbolo como clave emparejado con su valor correspondiente. Si la solución está vacía, se devuelve una lista vacía. Las raíces de la ecuación anterior son -4 y -1, como puedes ver en ➍.

En el primer capítulo descubrimos que las raíces de la ecuación

x2 + *x* + 1 = 0

son números complejos. Intentemos hallarlas utilizando solve():

>>> x=Symbol('x')  
>>> expr = x\*\*2 + x + 1  
>>> solve(expr, dict=True)  
[{x: -1/2 - sqrt(3)\*I/2}, {x: -1/2 + sqrt(3)\*I/2}]

Ambas raíces son imaginarias, como era de esperar, con la componente imaginaria indicada por el símbolo I.

#### ***Resolución de una variable en función de otras***

Además de hallar las raíces de las ecuaciones, podemos aprovechar la matemática simbólica para utilizar la función solve() para expresar una variable de una ecuación en términos de las demás. Veamos cómo encontrar las raíces de la ecuación cuadrática genérica ax2 + *bx* + *c* = 0. Para ello, definiremos *x* y tres símbolos *adicionales-a*, *b* y *c*, que corresponden a las tres constantes:

>>> x = Symbol('x')  
>>> a = Symbol('a')  
>>> b = Symbol('b')  
>>> c = Symbol('c')

A continuación, escribimos la expresión correspondiente a la ecuación y utilizamos sobre ella la función solve():

>>> expr = a\*x\*x + b\*x + c  
>>> solve(expr, x, dict=True)  
[{x: (-b + sqrt(-4\*a\*c + b\*\*2))/(2\*a)}, {x: -(b + sqrt(-4\*a\*c + b\*\*2))/(2\*a)}]

Aquí, tenemos que incluir un argumento adicional, x, a la función solve(). Como hay más de un símbolo en la ecuación, tenemos que decirle a solve() qué símbolo debe resolver, que es lo que indicamos pasando x como segundo argumento. Como era de esperar, solve() imprime la fórmula cuadrática: la fórmula genérica para encontrar el valor o valores de *x* en una expresión polinómica.

Para que quede claro, cuando utilizamos solve() en una ecuación con más de un símbolo, especificamos el símbolo a resolver como segundo argumento (y ahora el tercer argumento especifica cómo queremos que se devuelvan los resultados).

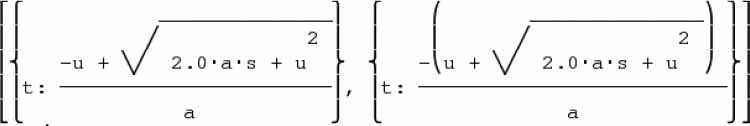
A continuación, veamos un ejemplo de física. Según una de las ecuaciones del movimiento, la distancia recorrida por un cuerpo que se mueve con una aceleración constante *a*, con una velocidad inicial *u*, en un tiempo *t*, viene dada por

image

Sin embargo, dados *u* y *a*, si quisieras hallar el tiempo necesario para recorrer una distancia dada, *s*, tendrías que expresar primero *t* en términos de las otras variables. He aquí cómo podrías hacerlo utilizando la función solve() de SymPy:

>>> from sympy import Symbol, solve, pprint  
>>> s = Symbol('s')  
>>> u = Symbol('u')  
>>> t = Symbol('t')  
>>> a = Symbol('a')  
>>> expr = u\*t + (1/2)\*a\*t\*t - s  
>>> t\_expr = solve(expr,t, dict=True)  
>>> pprint(t\_expr)

El resultado tiene este aspecto:



Ahora que tenemos la expresión para *t* (a la que se refiere la etiqueta t\_expr), podemos utilizar el método subs() para sustituir los valores de *s*, *u* y *a* y encontrar los dos valores posibles de *t*.

#### ***Resolución de un sistema de ecuaciones lineales***

Considera las dos ecuaciones siguientes

*2x* + *3y* = 6

*3x* + *2y* = 12

Supongamos que queremos encontrar el par de valores*(x*, *y*) que satisface las dos ecuaciones. Podemos utilizar la función solve() para encontrar la solución de un sistema de ecuaciones como éste.

En primer lugar, definimos los dos símbolos y creamos las dos ecuaciones:

>>> x = Symbol('x')  
>>> y = Symbol('y')  
>>> expr1 = 2\*x + 3\*y - 6  
>>> expr2 = 3\*x + 2\*y – 12

Las dos ecuaciones están definidas por las expresiones expr1 y expr2, respectivamente. Observa cómo hemos reordenado las expresiones para que ambas sean iguales a cero (hemos desplazado el lado derecho de las ecuaciones dadas al lado izquierdo). Para encontrar la solución, llamamos a la función solve() con las dos expresiones formando una tupla:

>>> solve((expr1, expr2), dict=True)  
[{y: -6/5, x: 24/5}]

Como he dicho antes, obtener la solución en forma de diccionario es útil en este caso. Podemos ver que el valor de x es 24/5 y el valor de y es -6/5. Verifiquemos si la solución que hemos obtenido satisface realmente las ecuaciones. Para ello, primero crearemos una etiqueta, soln, para referirnos a la solución que hemos obtenido y, a continuación, utilizaremos el método subs() para sustituir los valores correspondientes de x y y en las dos expresiones:

>>> soln = solve((expr1, expr2), dict=True)  
>>> soln = soln[0]  
>>> expr1.subs({x:soln[x], y:soln[y]})  
0  
>>> expr2.subs({x:soln[x], y:soln[y]})  
0

El resultado de sustituir los valores de x y y correspondientes a la solución en las dos expresiones es cero.

### **Trazar con SymPy**

En [el Capítulo 2](ch02.html#ch02) aprendimos a hacer gráficas en las que especificábamos explícitamente los números que queríamos representar. Por ejemplo, para trazar el gráfico de la fuerza gravitatoria frente a la distancia entre dos cuerpos, tenías que calcular la fuerza gravitatoria para cada valor de distancia y proporcionar las listas de distancias y fuerzas a matplotlib. Con SymPy, en cambio, sólo tienes que decirle la ecuación de la recta que quieres trazar y el gráfico se creará por ti. Vamos a trazar una recta cuya ecuación viene dada por *y* =*2x* + 3:

>>> from sympy.plotting import plot  
>>> from sympy import Symbol  
>>> x = Symbol('x')  
>>> plot(2\*x+3)

Todo lo que tuvimos que hacer fue importar plot y Symbol de sympy.plotting, crear un símbolo, x, y llamar a la función plot() con la expresión 2\*x+3. SymPy se encarga de todo lo demás y traza la gráfica de la función, como se muestra en la [Figura 4-1](ch04.html#ch4fig1).

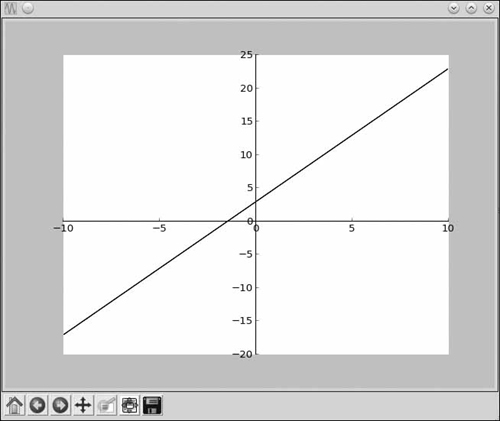


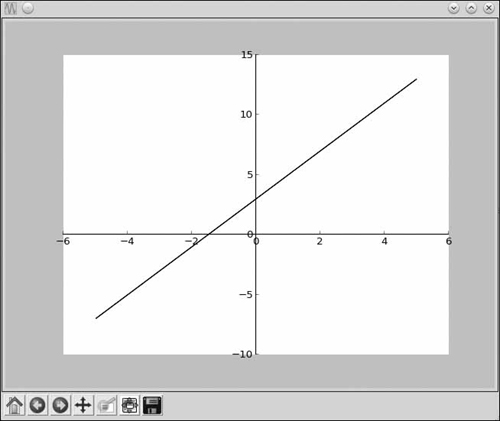
Figura 4-1*: Gráfico de la recta* y = *2x*+ *3*

La gráfica muestra que se ha elegido automáticamente un intervalo predeterminado de valores de *x*: de -10 a 10. Puede que notes que la ventana de la gráfica tiene un aspecto muy similar a las que viste en [los Capítulos 2](ch02.html#ch02) y [3](ch03.html#ch03). Esto se debe a que SymPy utiliza un lenguaje de programación muy sencillo. Esto se debe a que SymPy utiliza matplotlib entre bastidores para dibujar los gráficos. Observa también que no hemos tenido que llamar a la función show() para mostrar las gráficas, porque SymPy lo hace automáticamente.

Ahora, supongamos que quieres limitar los valores de 'x' en el gráfico anterior para que se encuentren en el intervalo de -5 a 5 (en lugar de -10 a 10). Lo harías de la siguiente manera

>>> plot((2\*x + 3), (x, -5, 5))

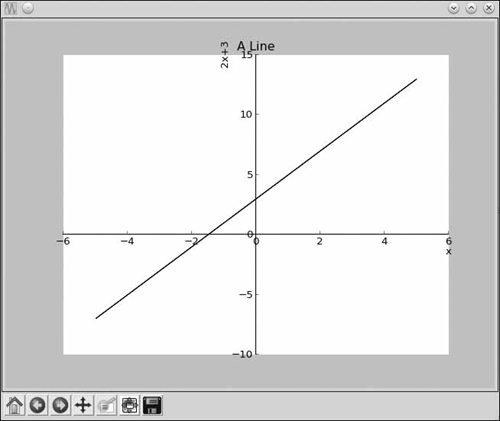
Aquí, una tupla formada por el símbolo, el límite inferior y el límite superior del intervalo -(x, -5, 5)- se especifica como segundo argumento de la función plot(). Ahora, el gráfico sólo muestra los valores de *y* correspondientes a los valores de *x* comprendidos entre -5 y 5 (ver [Figura 4-2](ch04.html#ch4fig2)).



*Figura 4-2: Gráfico de la recta* y = *2x*+ *3 con los valores de* x *restringidos al intervalo de -5 a* 5

Puedes utilizar otros argumentos de palabra clave en la función plot(), como title para introducir un título o xlabel y ylabel para etiquetar el eje *x* y el *eje y*, respectivamente. La siguiente función plot() especifica los tres argumentos de palabra clave anteriores (véase el gráfico correspondiente en la [Figura 4-3](ch04.html#ch4fig3)):

>>> plot(2\*x + 3, (x, -5, 5), title='A Line', xlabel='x', ylabel='2x+3')



*Figura 4-3: Gráfico de la recta* y = *2x*+ *3 con el rango de* x *y otros atributos especificados*

El gráfico de la Figura [4-3](ch04.html#ch4fig3) tiene ahora un título y etiquetas en el eje *x* y en el *eje y*. Puedes especificar otros argumentos de palabra clave a la función plot() para personalizar el comportamiento de la función y del propio gráfico. El argumento de palabra clave show nos permite especificar si queremos que se muestre la gráfica. Si pasas show=False, el gráfico no se mostrará cuando llames a la función plot():

>>> p = plot(2\*x + 3, (x, -5, 5), title='A Line', xlabel='x', ylabel='2x+3', show=False)

Verás que no se muestra ninguna gráfica. La etiqueta p hace referencia a la gráfica que se crea, por lo que ahora puedes llamar a p.show() para mostrar la gráfica. También puedes guardar la gráfica como un archivo de imagen utilizando el método save(), como se indica a continuación:

>>> p.save('line.png')

Esto guardará el gráfico en un archivo *line.png* en el directorio actual.

#### ***Trazado de expresiones introducidas por el usuario***

La expresión que pases a la función plot() debe expresarse sólo en términos de *x*. Por ejemplo, antes hemos trazado *y* =*2x* + 3, que hemos introducido en la función trazar como simplemente*2x* + 3. Si la expresión no estuviera originalmente en esta forma, tendríamos que reescribirla. Por supuesto, podríamos hacerlo manualmente, fuera del programa. Pero, ¿y si quieres escribir un programa que permita a sus usuarios representar gráficamente cualquier expresión? Si el usuario introduce una expresión en forma de*2x* + *3y* - 6, digamos, primero tenemos que convertirla. La función solve() nos ayudará en este caso. Veamos un ejemplo:

>>> expr = input('Enter an expression: ')  
Enter an expression: 2\*x + 3\*y - 6  
➊ >>> expr = sympify(expr)  
➋ >>> y = Symbol('y')  
>>> solve(expr, y)  
➌ [-2\*x/3 + 2]

En ➊, utilizamos la función sympify() para convertir la expresión de entrada en un objeto SymPy. En ➋, creamos un objeto Symbol para representar 'y', de modo que podamos decirle a SymPy para qué variable queremos resolver la ecuación. A continuación, resolvemos la expresión para encontrar y en términos de x especificando y como segundo argumento de la función solve(). En ➌, esto devuelve la ecuación en términos de x, que es lo que necesitamos para el trazado.

Observa que esta expresión final está almacenada en una lista, así que antes de poder utilizarla, tendremos que extraerla de la lista:

>>> solutions = solve(expr, 'y')  
➍ >>> expr\_y = solutions[0]  
>>> expr\_y  
-2\*x/3 + 2

Creamos una etiqueta, solutions, para referirnos al resultado devuelto por la función solve(), que es una lista con un solo elemento. Luego, extraemos ese elemento en ➍. Ahora podemos llamar a la función plot() para representar gráficamente la expresión. El siguiente listado muestra un programa completo de trazado de gráficos:

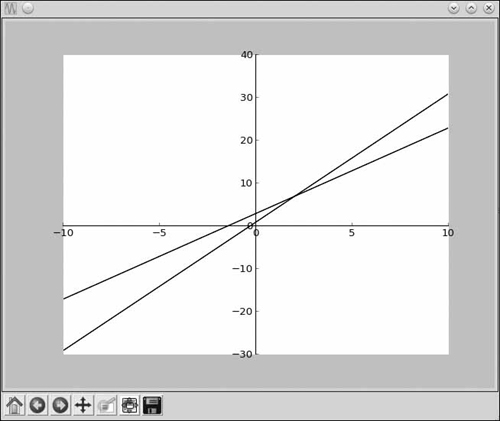
'''  
Plot the graph of an input expression  
'''  
  
from sympy import Symbol, sympify, solve  
from sympy.plotting import plot  
  
def plot\_expression(expr):  
  
y = Symbol('y')  
solutions = solve(expr, y)  
expr\_y = solutions[0]  
plot(expr\_y)  
  
if \_\_name\_\_=='\_\_main\_\_':  
  
expr = input('Enter your expression in terms of x and y: ')  
  
try:  
expr = sympify(expr)  
except SympifyError:  
print('Invalid input')  
else:  
plot\_expression(expr)

Observa que el programa anterior incluye un bloque try...except para comprobar si la entrada no es válida, como hemos hecho antes con sympify(). Cuando ejecutes el programa, te pedirá que introduzcas una expresión y creará el gráfico correspondiente.

#### ***Trazar múltiples funciones***

Puedes introducir varias expresiones al llamar a la función SymPy plot para trazar más de una expresión en el mismo gráfico. Por ejemplo, el siguiente código traza dos líneas a la vez (ver [Figura 4-4](ch04.html#ch4fig4)):

>>> from sympy.plotting import plot  
>>> from sympy import Symbol  
>>> x = Symbol('x')  
>>> plot(2\*x+3, 3\*x+1)

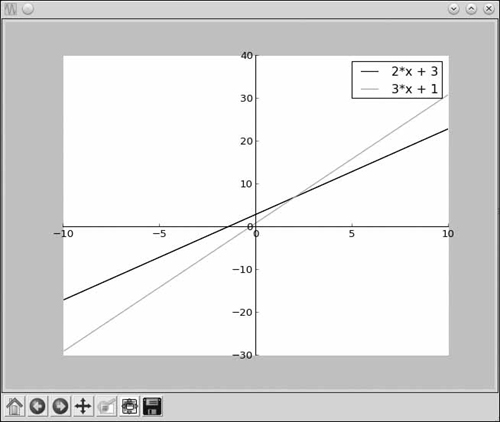


*Figura 4-4: Trazado de dos líneas en el mismo gráfico*

Este ejemplo pone de manifiesto otra diferencia entre el trazado en matplotlib y en SymPy. Aquí, con SymPy, las dos líneas tienen el mismo color, mientras que matplotlib las habría hecho automáticamente de colores diferentes. Para establecer colores diferentes para cada línea con SymPy, tendremos que realizar algunos pasos adicionales, como se muestra en el código siguiente, que también añade una leyenda al gráfico:

>>> from sympy.plotting import plot  
>>> from sympy import Symbol  
>>> x = Symbol('x')  
➊ >>> p = plot(2\*x+3, 3\*x+1, legend=True, show=False)  
➋ >>> p[0].line\_color = 'b'  
➌ >>> p[1].line\_color = 'r'  
>>> p.show()

En ➊, llamamos a la función plot() con las ecuaciones de las dos líneas, pero le pasamos dos argumentos clave adicionales:legend y show. Si establecemos el argumento leyenda en True, añadiremos una leyenda al gráfico, como vimos en el [Capítulo 2](ch02.html#ch02). Ten en cuenta, sin embargo, que el texto que aparezca en la leyenda coincidirá con las expresiones que hayas trazado; no puedes especificar ningún otro texto. También establecemos show=False porque queremos establecer el color de las líneas antes de dibujar el gráfico. La expresión en ➋, p[0], se refiere a la primera línea,*2x* + 3, y fijamos su atributo line\_color en 'b', lo que significa que queremos que esta línea sea azul. Del mismo modo, establecemos el color del segundo trazado en rojo utilizando la cadena 'r' ➌. Por último, llamamos a show() para que muestre el gráfico (véase [la Figura 4-5](ch04.html#ch4fig5)).



*Figura 4-5: Gráfico de las dos líneas con cada línea dibujada en un color diferente*

Además de rojo y azul, puedes trazar las líneas en verde, cian, magenta, amarillo, negro y blanco (utilizando la primera letra del color en cada caso).

### **Lo que has aprendido**

En este capítulo, has aprendido los fundamentos de la matemática simbólica utilizando SymPy. Has aprendido a declarar símbolos, a construir expresiones utilizando símbolos y operadores matemáticos, a resolver ecuaciones y a trazar gráficos. Aprenderás más funciones de SymPy en capítulos posteriores.

### **Retos de programación**

Aquí tienes algunos retos de programación que te ayudarán a aplicar mejor lo que has aprendido. Puedes encontrar ejemplos de soluciones en [*http://www.nostarch.com/doingmathwithpython/*](http://www.nostarch.com/doingmathwithpython/).

#### ***#nº 1: Buscador de factores***

Has aprendido sobre la función factor(), que imprime los factores de una expresión. Ahora que sabes cómo puede manejar tu programa las expresiones introducidas por un usuario, escribe un programa que pida al usuario que introduzca una expresión, calcule sus factores y los imprima. Tu programa debe ser capaz de manejar entradas no válidas haciendo uso del manejo de excepciones.

#### ***#2: Solucionador gráfico de ecuaciones***

Antes has aprendido a escribir un programa que pida al usuario que introduzca una expresión como *3x* + *2y* - 6 y cree la gráfica correspondiente. Escribe un programa que pida al usuario dos expresiones y luego grafique ambas, como se indica a continuación:

>>> expr1 = input('Enter your first expression in terms of x and y: ')  
>>> expr2 = input('Enter your second expression in terms of x and y: ')

Ahora, expr1 y expr2 almacenarán las dos expresiones introducidas por el usuario. Deberás convertir ambas en objetos SymPy utilizando el paso sympify() en un bloque try...except.

Todo lo que tienes que hacer a partir de aquí es trazar estas dos expresiones en lugar de una.

Una vez hecho esto, mejora tu programa para que imprima la solución, es decir, el par de valores *x* e *y* que satisface ambas ecuaciones. Éste será también el punto de intersección de las dos rectas de la gráfica. (Sugerencia: consulta cómo hemos utilizado antes la función solve() para hallar la solución de un sistema de dos ecuaciones lineales).

#### ***#3: Suma de una serie***

Ya vimos cómo hallar la suma de una serie en "[Imprimir una serie](ch04.html#ch04lev3sec01)", en [la página 99](ch04.html#page_99). Allí sumamos manualmente los términos de la serie haciendo un bucle sobre todos los términos. Aquí tienes un fragmento de ese programa:

for i in range(2, n+1):  
series = series + (x\*\*i)/i

La función summation() de SymPy puede utilizarse directamente para hallar dichas sumas. El siguiente ejemplo imprime la suma de los cinco primeros términos de la serie que hemos considerado antes:

>>> from sympy import Symbol, summation, pprint  
>>> x = Symbol('x')  
>>> n = Symbol('n')  
➊ >>> s = summation(x\*\*n/n, (n, 1, 5))  
>>> pprint(s)  
x5 x4 x3 x2  
-- + -- + -- + -- + x  
5 4 3 2

Llamamos a la función summation() en ➊, siendo el primer argumento el término *n*de la serie y el segundo una tupla que indica el rango de *n*. Aquí queremos la suma de los cinco primeros términos, así que el segundo argumento es (n, 1, 5).

Una vez que tengas la suma, puedes utilizar el método subs() para sustituir *x* por un valor y hallar el valor numérico de la suma:

>>> s.subs({x:1.2})  
3.51206400000000

Tu reto es escribir un programa que sea capaz de hallar la suma de una serie arbitraria cuando le proporcionas el término *enésimo*de la serie y el número de términos que la componen. Aquí tienes un ejemplo de cómo funcionaría el programa:

Enter the nth term: a+(n-1)\*d  
Enter the number of terms: 3  
3·a + 3·d

En este ejemplo, el *enésimo*término suministrado es el de una *progresión aritmética*. Partiendo de a y d como *diferencia común*, el número de términos hasta el que se debe calcular la suma es 3. La suma resulta ser 3a + 3d, lo que concuerda con la fórmula conocida para la misma.

#### ***#nº 4: Resolución de inecuaciones de una sola variable***

Ya has visto cómo resolver una ecuación utilizando la función solve() de SymPy. Pero SymPy también es capaz de resolver desigualdades de una sola variable, como *x* + 5 > 3 y *sinx* - 0,6 > 0. Es decir, SymPy puede resolver relaciones además de la igualdad, como >, <, etc. Para este reto, crea una función, isolve(), que tome cualquier desigualdad, la resuelva y devuelva la solución.

En primer lugar, vamos a conocer las funciones SymPy que te ayudarán a ponerlo en práctica. Las funciones de resolución de desigualdades están disponibles como tres funciones distintas para desigualdades polinómicas, racionales y todas las demás. Tendremos que elegir la función adecuada para resolver las distintas desigualdades, o obtendremos un error.

Un *polinomio* es una expresión algebraica formada por una variable y coeficientes y en la que sólo intervienen las operaciones de suma, resta y multiplicación y sólo potencias positivas de la variable. Un ejemplo de desigualdad polinómica es x2 + 4 < 0.

Para resolver una desigualdad polinómica, utiliza la función solve\_poly\_inequality():

>>> from sympy import Poly, Symbol, solve\_poly\_inequality  
>>> x = Symbol('x')  
➊ >>> ineq\_obj = -x\*\*2 + 4 < 0  
➋ >>> lhs = ineq\_obj.lhs  
➌ >>> p = Poly(lhs, x)  
➍ >>> rel = ineq\_obj.rel\_op  
>>> solve\_poly\_inequality(p, rel)  
[(-oo, -2), (2, oo)]

Primero, crea la expresión que representa una desigualdad, -x2 + 4 < 0, en ➊ y refiérete a esta expresión con la etiqueta ineq\_obj. A continuación, extrae el lado izquierdo de la desigualdad -es decir, la expresión algebraica -x2 + 4- utilizando el atributo lhs en ➋. A continuación, crea un objeto Poly en ➌ para representar el polinomio que hemos extraído en ➋. El segundo argumento que se pasa al crear el objeto es el objeto símbolo que representa la variable, x. En ➍, extrae el operador relacional del objeto desigualdad utilizando el atributo rel. Por último, llama a la función solve\_poly\_inequality() con el objeto polinomio, p, y rel como los dos argumentos. El programa devuelve la solución como una lista de tuplas, en la que cada tupla representa una solución para la desigualdad como el límite inferior y el límite superior del rango de números. Para esta desigualdad, la solución son todos los números menores que -2 y todos los números mayores que 2.

Una *expresión* racional es una expresión algebraica en la que el numerador y el denominador son polinomios. Aquí tienes un ejemplo de desigualdad racional:

image

Para las inecuaciones racionales, utiliza la función solve\_rational\_inequalities():

>>> from sympy import Symbol, Poly, solve\_rational\_inequalities  
>>> x = Symbol('x')  
➊ >>> ineq\_obj = ((x-1)/(x+2)) > 0  
>>> lhs = ineq\_obj.lhs  
➋ >>> numer, denom = lhs.as\_numer\_denom()  
>>> p1 = Poly(numer)  
>>> p2 = Poly(denom)  
>>> rel = ineq\_obj.rel\_op  
➌ >>> solve\_rational\_inequalities([[((p1, p2), rel)]])  
(-oo, -2) U (1, oo)

Crea un objeto desigualdad que represente nuestra desigualdad racional de ejemplo en ➊ y, a continuación, extrae la expresión racional utilizando el atributo lhs. Separa el numerador y el denominador en las etiquetas numer y denom utilizando el método as\_numer\_denom() en ➋, que devuelve una tupla con el numerador y el denominador como los dos miembros. A continuación, crea dos objetos polinomio, p1 y p2, que representen el numerador y el denominador, respectivamente. Recupera el operador relacional y llama a la función solve\_rational\_inequalities(), pasándole los dos objetos polinomio -p1 y p2- y el operador relacional.

El programa devuelve la solución (-oo, -2) U (1, oo), donde U denota que la solución es la *unión* de los dos *conjuntos* de soluciones formados por todos los números menores que -2 y todos los números mayores que 1. (Aprenderemos sobre conjuntos en el [Capítulo 5](ch05.html#ch05).)

Por último, *sinx* - 0,6 > 0 es un ejemplo de desigualdad que no pertenece ni a la categoría de los polinomios ni a la de las expresiones racionales. Si tienes que resolver una desigualdad de este tipo, utiliza la función solve\_univariate\_inequality():

>>> from sympy import Symbol, solve, solve\_univariate\_inequality, sin  
>>> x = Symbol('x')  
>>> ineq\_obj = sin(x) - 0.6 > 0  
>>> solve\_univariate\_inequality(ineq\_obj, x, relational=False)  
(0.643501108793284, 2.49809154479651)

Crea un objeto desigualdad que represente la desigualdad sin(x) – 0.6 > 0 y luego llama a la función solve\_univariate\_inequality() con los dos primeros argumentos como el objeto desigualdad, ineq\_obj, y el objeto símbolo, x. El argumento de palabra clave relational=False especifica a la función que queremos que la solución se devuelva como un *conjunto*. La solución de esta desigualdad son todos los números que se encuentran entre el primer y el segundo miembro de la tupla que devuelve el programa.

##### **Consejos: Funciones útiles**

Ahora recuerda: tu reto es (1) crear una función, isolve(), que tome cualquier desigualdad y (2) elegir una de las funciones adecuadas que se han comentado en esta sección para resolverla y devolver la solución. Las siguientes pistas pueden ser útiles para implementar esta función.

El método is\_polynomial() puede utilizarse para comprobar si una expresión es un polinomio o no:

>>> x = Symbol('x')  
>>> expr = x\*\*2 - 4  
>>> expr.is\_polynomial()  
True  
>>> expr = 2\*sin(x) + 3  
>>> expr.is\_polynomial()  
False

El método is\_rational\_function() puede utilizarse para comprobar si una expresión es una expresión racional:

>>> expr = (2+x)/(3+x)  
>>> expr.is\_rational\_function()  
True  
>>> expr = 2+x  
>>> expr.is\_rational\_function()  
True  
>>> expr = 2+sin(x)  
>>> expr.is\_rational\_function()  
False

La función sympify() puede convertir una desigualdad expresada como cadena en un objeto desigualdad:

>>> from sympy import sympify  
>>> sympify('x+3>0')  
x + 3 > 0

Cuando ejecutes tu programa, debe pedir al usuario que introduzca una expresión de desigualdad e imprimir la solución.