Capítulo 4: Álgebra y matemáticas simbólicas con SymPy

### **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  
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

[anterior](ch04_2.html)[Subtema 3 de 7: (Ver todo)](ch04.html)[siguiente](ch04_4.html)