Capítulo 7: Resolución de problemas de cálculo

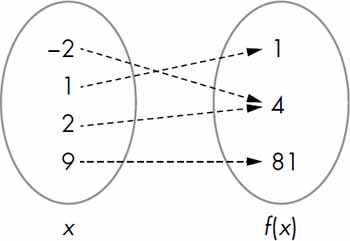
## **7** **Res**olver problemas de cálculo



En este último capítulo aprenderemos a resolver problemas de cálculo. Primero aprenderemos qué son las funciones matemáticas, seguido de una rápida visión general de las funciones matemáticas más comunes disponibles en la biblioteca estándar de Python y en SymPy. A continuación, aprenderemos a encontrar los límites de las funciones y a calcular derivadas e integrales, es decir, el tipo de cosas que harías en cualquier clase de cálculo. ¡Empecemos!

### **¿Qué es una función?**

Empecemos con algunas definiciones básicas. Una función es una *correspondencia* entre un conjunto de entrada y un conjunto de salida. La condición especial de una función es que un elemento del conjunto de entrada esté relacionado *exactamente* con *un* elemento del conjunto de salida. Por ejemplo, [la Figura 7-1](ch07.html#ch7fig1) muestra dos conjuntos tales que un elemento del conjunto de salida es el cuadrado de un elemento que pertenece al conjunto de entrada.



*Figura 7-1: Una función describe una correspondencia entre un conjunto de entrada y un conjunto de salida. Aquí, un elemento del conjunto de salida es el cuadrado de un elemento del conjunto de entrada.*

Utilizando la conocida notación de funciones, escribiríamos esta función como *f*(*x*) = x2, donde *x* es la cantidad variable independiente. Así, *f*(2) = 4, *f*(100) = 10000, etc. Nos referimos a *x* como la cantidad variable independiente porque somos libres de asumir un valor para ella siempre que ese valor esté dentro de su dominio (véase el siguiente apartado).

Las funciones también pueden definirse en términos de múltiples variables. Por ejemplo, *f(x*, *y*) = x2 + y2 define una función de dos variables, *x* e *y*.

#### ***Dominio y rango de una función***

El *dominio* de una función es el conjunto de valores de entrada que la variable independiente puede asumir válidamente. El conjunto de valores de salida de una función se llama *rango*.

Por ejemplo, el dominio de la función *f(x*) = *1/x* son todos los números reales y complejos distintos de cero, porque 1/0 no está definido. El rango está formado por el conjunto de valores obtenidos al sustituir cada número del dominio por *1/x*, por lo que en este caso también son todos los números reales y complejos distintos de cero.

**NOTA**

*El dominio y el rango de una función pueden ser ciertamente diferentes. Por ejemplo, para la funciónx2, el dominio son todos los números positivos y negativos, pero el rango son sólo los números positivos.*

#### ***Visión general de las funciones matemáticas más comunes***

Ya hemos utilizado varias funciones matemáticas comunes del módulo math de la biblioteca estándar de Python. Un par de ejemplos familiares son las funciones sin() y cos(), que corresponden a las funciones trigonométricas seno y coseno. También se definen otras funciones trigonométricas:tan() y los equivalentes inversos de estas funciones, asin(), acos() y atan().

El módulo math también incluye funciones que hallan el logaritmo de un número -la función logaritmo natural log(), el logaritmo de base-2 log2(), y el logaritmo de base-10 log10()-, así como la función exp(), que halla el valor de *ex*, donde *e* es el número de Euler (aproximadamente 2,71828).

Un inconveniente de todas estas funciones es que no son adecuadas para trabajar con expresiones simbólicas. Si queremos manipular una expresión matemática que incluya símbolos, tenemos que empezar a utilizar las funciones equivalentes definidas por SymPy.

Veamos un ejemplo rápido:

>>> import math  
>>> math.sin(math.pi/2)  
1.0

Aquí, encontramos el seno del ángulo *π/2* utilizando la función sin() definida por el módulo math de la biblioteca estándar. A continuación, podemos hacer lo mismo utilizando SymPy.

>>> import sympy  
>>> sympy.sin(math.pi/2)  
1.00000000000000

Al igual que la función sin() de la biblioteca estándar, la función sin() de SymPy espera que el ángulo se exprese en radianes. Ambas funciones devuelven 1.

Ahora, intentemos llamar a cada función con un símbolo en su lugar y veamos qué ocurre:

>>> from sympy import Symbol  
>>> theta = Symbol('theta')  
➊ >>> math.sin(theta) + math.sin(theta)  
Traceback (most recent call last):  
File "<pyshell#53>", line 1, in <module>  
math.sin(theta) + math.sin(theta)  
File "/usr/lib/python3.4/site-packages/sympy/core/expr.py", line 225, in  
\_\_float\_\_  
raise TypeError("can't convert expression to float")  
TypeError: can't convert expression to float  
  
➋ >>> sympy.sin(theta) + sympy.sin(theta)  
2\*sin(theta)

La función sin() de la biblioteca estándar no sabe qué hacer cuando la llamamos con theta en ➊, así que lanza una excepción para indicar que espera un valor numérico como argumento de la función sin(). En cambio, SymPy es capaz de realizar la misma operación en ➋, y devuelve como resultado la expresión  2\*sin(theta) . Esto apenas nos sorprende ahora, pero ilustra el tipo de tareas en las que las funciones matemáticas de la biblioteca estándar pueden quedarse cortas.

Consideremos otro ejemplo. Supongamos que queremos deducir la expresión del tiempo que tarda un cuerpo en movimiento de proyectil en alcanzar el punto más alto si se lanza con velocidad inicial u con un ángulo theta (ver "[Movimiento de proyectil](ch02.html#ch02lev2sec08)" en [la página 48](ch02.html#page_48)).

En el punto más alto, u\*sin(theta)-g\*t = 0, así que para hallar t, utilizaremos la función solve() que aprendimos en el [Capítulo 4](ch04.html#ch04):

>>> from sympy import sin, solve, Symbol  
>>> u = Symbol('u')  
>>> t = Symbol('t')  
>>> g = Symbol('g')  
>>> theta = Symbol('theta')  
>>> solve(u\*sin(theta)-g\*t, t)  
[u\*sin(theta)/g]

La expresión para t, como aprendimos antes, resulta ser u\*sin(theta)/g, e ilustra cómo la función solve() puede utilizarse también para encontrar soluciones a ecuaciones que contienen funciones matemáticas.

### **Suposiciones en SymPy**

En todos nuestros programas, hemos creado un objeto Symbol en SymPy, definiendo la variable así: x = Symbol('x'). Supongamos que, como resultado de una operación que le has pedido a SymPy que realice, SymPy necesita comprobar si la expresión *x* + 5 es mayor que 0. Veamos qué ocurriría:

>>> from sympy import Symbol  
>>> x = Symbol('x')  
>>> if (x+5) > 0:  
print('Do Something')  
else:  
print('Do Something else')  
  
Traceback (most recent call last):  
File "<pyshell#45>", line 1, in <module>  
if (x + 5) > 0:  
File "/usr/lib/python3.4/site-packages/sympy/core/relational.py", line 103,  
in \_\_nonzero\_\_  
raise TypeError("cannot determine truth value of\n%s" % self)  
TypeError: cannot determine truth value of  
x + 5 > 0

Como SymPy no sabe nada sobre el signo de *x*, no puede deducir si *x* + 5 es mayor que 0, por lo que muestra un error. Pero las matemáticas básicas nos dicen que si *x* es positivo, *x* + 5 siempre será positivo, y si *x* es negativo, sólo será positivo en determinados casos.

Así que si creamos un objeto Symbol especificando positive=True, le decimos a SymPy que asuma sólo valores positivos. Ahora sabe con seguridad que *x* + 5 es definitivamente mayor que 0:

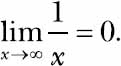
>>> x = Symbol('x', positive=True)  
>>> if (x+5) > 0:  
print('Do Something')  
else:  
print('Do Something else')  
  
Do Something

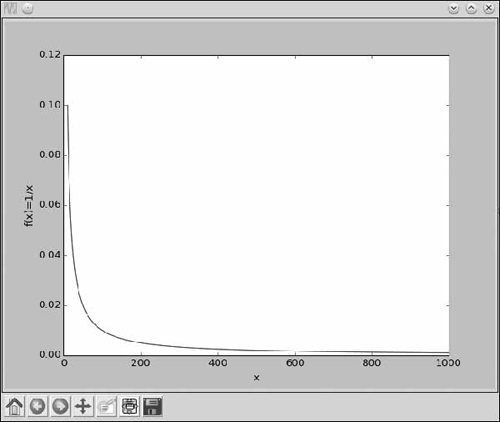
Ten en cuenta que si en su lugar hubiéramos especificado negative=True, podríamos obtener el mismo error que en el primer caso. Igual que podemos declarar un símbolo como positive y negative, también es posible especificarlo como real, integer, complex, imaginary, etc. Estas declaraciones se denominan *supuestos* en SymPy.

### **Encontrar el límite de funciones**

Una tarea habitual en cálculo es encontrar el *valor límite* (o simplemente el *límite*) de la función, cuando se supone que el valor de la variable se aproxima a un determinado valor. Considera una función *f(x*) = *1/x*, cuya gráfica se muestra en [la Figura 7-2](ch07.html#ch7fig2).

A medida que aumenta el valor de *x*, el valor de *f*(*x*) se aproxima a 0. Utilizando la notación límite, escribiríamos esto como





*Figura 7-2: Gráfica de la función 1/x* *a* medida que *aumenta* el valor de x

Podemos encontrar límites de funciones en SymPy creando objetos de la clase Limit como se indica a continuación:

➊ >>> from sympy import Limit, Symbol, S  
➋ >>> x = Symbol('x')  
➌ >>> Limit(1/x, x, S.Infinity)  
Limit(1/x, x, oo, dir='-')

En ➊, importamos las clases Limit y Symbol, así como S, que es una clase especial de SymPy que contiene la definición de infinito (positivo y negativo) y otros valores especiales. A continuación, en ➋ creamos un objeto símbolo, x, para representar *x*. Creamos el objeto Limit en ➌, pasándole tres argumentos: 1/x, la variable x, y por último el valor en el que queremos calcular el límite de la función (infinito, dado por S.Infinity).

El resultado se devuelve como un objeto *no evaluado* con el símbolo oo que denota el infinito positivo y el símbolo dir='-' que especifica que nos acercamos al límite por el lado negativo.

Para encontrar el valor del límite, utilizamos el método doit():

>>> l = Limit(1/x, x, S.Infinity)  
>>> l.doit()  
0

Por defecto, el límite se encuentra desde una dirección positiva, a menos que el valor en el que se va a calcular el límite sea infinito positivo o negativo. En el caso del infinito positivo, la dirección es negativa, y viceversa. Puedes cambiar la dirección por defecto como se indica a continuación:

>>> Limit(1/x, x, 0, dir='-').doit()  
-oo

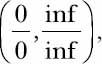
Aquí calculamos

image

y a medida que nos acercamos a 0 para *x* desde el lado negativo, el valor del límite se aproxima al infinito negativo. En cambio, si nos acercamos a 0 desde el lado positivo, el valor se aproxima al infinito positivo:

>>> Limit(1/x, x, 0, dir='+').doit()  
oo

La clase Limit también maneja funciones con límites de formas indeterminadas,



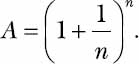
automáticamente:

>>> from sympy import Symbol, sin  
>>> Limit(sin(x)/x, x, 0).doit()  
1

Es muy probable que hayas utilizado la regla de l'Hôpital para encontrar dichos límites, pero como vemos aquí, la clase Limit se encarga de ello por nosotros.

#### ***Interés compuesto continuo***

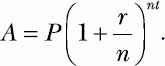
Supongamos que has depositado 1 $ en un banco. Este depósito es el *capital*, que te paga un interés *-en*este caso, un interés del 100% que se compone n veces al año durante 1 año. La cantidad que obtendrás al cabo de 1 año viene dada por



El destacado matemático James Bernoulli descubrió que, a medida que aumenta el valor de *n*, el término (1 + *1/n)n* se aproxima al valor de *e, la*constante que podemos verificar hallando el límite de la función:

>>> from sympy import Limit, Symbol, S  
>>> n = Symbol('n')  
>>> Limit((1+1/n)\*\*n, n, S.Infinity).doit()  
E

Para cualquier importe principal *p*, cualquier tipo *r* y cualquier número de años *t*, el interés compuesto se calcula mediante la fórmula



Suponiendo un interés compuesto continuo, podemos hallar la expresión de *A* de la siguiente manera:

>>> from sympy import Symbol, Limit, S  
>>> p = Symbol('p', positive=True)  
>>> r = Symbol('r', positive=True)  
>>> t = Symbol('t', positive=True)  
>>> Limit(p\*(1+r/n)\*\*(n\*t), n, S.Infinity).doit()  
p\*exp(r\*t)

Creamos tres objetos símbolo, que representan el importe principal, p, el tipo de interés, r, y el número de años, t. También le decimos a SymPy que estos símbolos asumirán valores positivos pasando el argumento de la palabra clave positive=True al crear los objetos Symbol. Si no lo especificamos, SymPy no sabe nada sobre los valores numéricos que puede asumir el símbolo y puede que no sea capaz de evaluar el límite correctamente. A continuación, introducimos la expresión del interés compuesto para crear el objeto Limit y lo evaluamos mediante el método doit(). El límite resulta ser p\*exp(r\*t), lo que nos indica que el interés compuesto crece exponencialmente con el tiempo para el tipo de interés fijo.

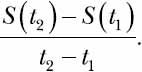
#### ***Tasa de variación instantánea***

Considera un coche que se desplaza por una carretera. Acelera uniformemente de forma que la distancia recorrida, *S*, viene dada por la función

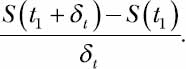
*S(t*) = 5t2 + *2t* + 8.

En esta función, la variable independiente es *t*, que representa el tiempo transcurrido desde que el coche empezó a moverse.

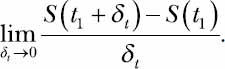
Si medimos la distancia recorrida en el tiempo t1 y en el tiempo t2 de forma que t2 > t1, podemos calcular la distancia recorrida por el coche en 1 unidad de tiempo mediante la expresión



También se denomina tasa de variación media de la función *S(t*) respecto a la variable *t*, o dicho de otro modo, velocidad media. Si escribimos t2 como t1 + δt *-dondeδt* es la diferencia entre t2 y t1 en unidades de tiempo- podemos reescribir la expresión de la velocidad media como



Esta expresión también es una función con t1 como variable. Ahora bien, si además suponemos que *δt* es realmente pequeña, de modo que se aproxima a 0, podemos utilizar la notación de límite para escribirla como



Ahora evaluaremos el límite anterior. En primer lugar, vamos a crear los distintos objetos de expresión:

>>> from sympy import Symbol, Limit  
>>> t = Symbol('t')  
➊ >>> St = 5\*t\*\*2 + 2\*t + 8  
  
>>> t1 = Symbol('t1')  
>>> delta\_t = Symbol('delta\_t')  
  
➋ >>> St1 = St.subs({t: t1})  
➌ >>> St1\_delta = St.subs({t: t1 + delta\_t})

Primero definimos la función *S(t*) en ➊. A continuación, definimos dos símbolos, t1 y delta\_t, que corresponden a t1 y *δt*. Utilizando el método subs(), hallamos entonces *S*(t1) y *S(*t1 + *δt*) sustituyendo el valor de t por t1 y t1\_delta\_t en ➋ y ➌, respectivamente.

Ahora, evaluemos el límite:

>>> Limit((St1\_delta-St1)/delta\_t, delta\_t, 0).doit()  
10\*t1 + 2

El límite resulta ser 10\*t1 + 2, y es la tasa de cambio de *S(t*) en el tiempo t1, o la tasa de cambio instantánea. Este cambio se denomina más comúnmente *velocidad instant* ánea del coche en el instante de tiempo t1.

El límite que hemos calculado aquí se denomina *derivada* de una función, y podemos calcularlo directamente utilizando la clase Derivative de SymPy.

### **Encontrar la derivada de funciones**

La derivada de una función *y* = *f(x*) expresa la tasa de cambio de la variable dependiente, *y*, con respecto a la variable independiente, *x*. Se denota como *f′(x*) o *dy/dx*. Podemos hallar la derivada de una función creando un objeto de la clase Derivative. Utilicemos como ejemplo la función anterior que representa el movimiento de un coche:

➊ >>> from sympy import Symbol, Derivative  
  
>>> t = Symbol('t')  
>>> St = 5\*t\*\*2 + 2\*t + 8  
  
➋ >>> Derivative(St, t)  
Derivative(5\*t\*\*2 + 2\*t + 8, t)

Importamos la clase Derivative en ➊. En ➋, creamos un objeto de la clase Derivative. Los dos argumentos que se pasan al crear el objeto son la función St y el símbolo t, que corresponde a la variable *t*. Al igual que con la clase Limit, se devuelve un objeto de la clase Derivative, y en realidad no se calcula la derivada. Llamamos al método doit() sobre el objeto no evaluado Derivative para hallar la derivada:

>>> d = Derivative(St, t)  
>>> d.doit()  
10\*t + 2

La expresión para la derivada resulta ser 10\*t + 2. Ahora, si queremos calcular el valor de la derivada en un valor concreto de *t -digamos*, *t* = t1 o *t* = 1- podemos utilizar el método subs():

>>> d.doit().subs({t:t1})  
10\*t1 + 2  
>>> d.doit().subs({t:1})  
12

Probemos con una función arbitraria complicada con *x* como única variable:*(*x3 + x2 + *x*) ×*(*x2 + *x*).

>>> from sympy import Derivative, Symbol  
>>> x = Symbol('x')  
>>> f = (x\*\*3 + x\*\*2 + x)\*(x\*\*2+x)  
>>> Derivative(f, x).doit()  
(2\*x + 1)\*(x\*\*3 + x\*\*2 + x) + (x\*\*2 + x)\*(3\*x\*\*2 + 2\*x + 1)

Puedes considerar esta función como el producto de dos funciones independientes, lo que significa que, a mano, tendríamos que hacer uso de la *regla del producto* de la diferenciación para hallar la derivada. Pero aquí no tenemos que preocuparnos de eso, porque podemos crear un objeto de la clase Derivative para que lo haga por nosotros.

Prueba con otras expresiones complicadas, como las que implican funciones trigonométricas.

#### ***Una calculadora de derivadas***

Ahora vamos a escribir un programa calculadora de derivadas, que tomará una función como entrada y luego imprimirá el resultado de diferenciarla respecto a la variable especificada:

'''  
Derivative calculator  
'''  
  
from sympy import Symbol, Derivative, sympify, pprint  
from sympy.core.sympify import SympifyError  
  
def derivative(f, var):  
var = Symbol(var)  
d = Derivative(f, var).doit()  
pprint(d)  
  
if \_\_name\_\_=='\_\_main\_\_':  
  
➊ f = input('Enter a function: ')  
var = input('Enter the variable to differentiate with respect to: ')  
try:  
➋ f = sympify(f)  
except SympifyError:  
print('Invalid input')  
else:  
➌ derivative(f, var)

En ➊, pedimos al usuario que introduzca una función para la que se debe hallar la derivada y, a continuación, le pedimos la variable con respecto a la cual se debe diferenciar la función. En ➋, convertimos la función de entrada en un objeto SymPy utilizando la función sympify(). Llamamos a esta función en un bloque try...except para poder mostrar un mensaje de error en caso de que el usuario introduzca una entrada no válida. Si la expresión de entrada es una expresión válida, llamamos a la función derivada en ➌, pasando como argumentos la expresión convertida y la variable con respecto a la cual se va a diferenciar la función.

En la función derivative(), creamos primero un objeto Symbol que corresponde a la variable con respecto a la cual se va a diferenciar la función. Utilizamos la etiqueta var para referirnos a esta variable. A continuación, creamos un objeto Derivative que pasa tanto la función a diferenciar como el objeto símbolo var. Inmediatamente llamamos al método doit() para evaluar la derivada, y a continuación utilizamos la función pprint() para imprimir el resultado de forma que se parezca a su homólogo matemático. A continuación se muestra un ejemplo de ejecución del programa:

Enter a function: 2\*x\*\*2 + 3\*x + 1  
Enter the variable to differentiate with respect to: x  
4·x + 3

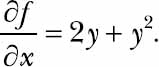
Aquí tienes un ejemplo de ejecución cuando se utiliza con una función de dos variables:

Enter a function: 2\*x\*\*2 + y\*\*2  
Enter the variable to differentiate with respect to: x  
4·x

#### ***Cálculo de derivadas parciales***

En el programa anterior vimos que es posible calcular la derivada de una función multivariable respecto a cualquier variable utilizando la clase Derivative. Este cálculo suele denominarse *diferenciación* parcial, indicando *parcial* que suponemos que sólo varía una variable, mientras que las demás son fijas.

Consideremos la función *f(x*, *y*) = *2xy* + xy2. La diferenciación parcial de *f*(*x*, *y*) respecto a *x* es



El programa anterior es capaz de hallar la derivada parcial porque sólo es cuestión de especificar la variable correcta:

Enter a function: 2\*x\*y + x\*y\*\*2  
Enter the variable to differentiate with respect to: x  
y2 + 2·y

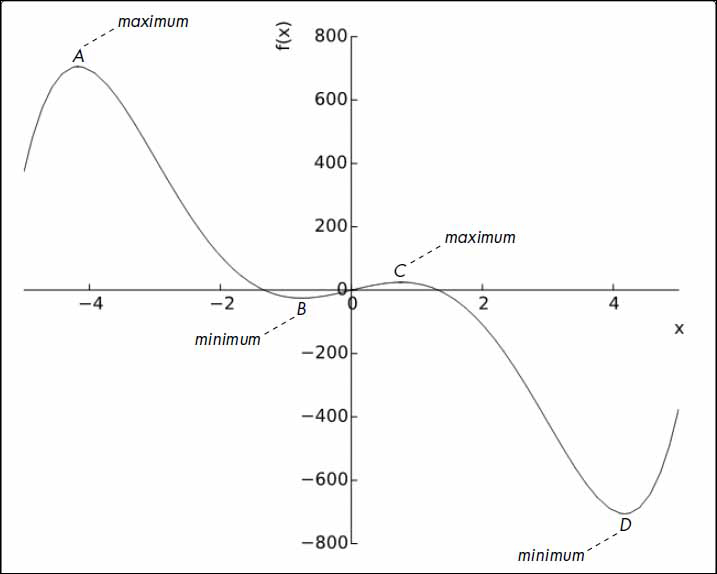
**NOTA**

*Una suposición clave que he hecho en este capítulo es que todas las funciones de las que estamos calculando la derivada son diferenciables en sus respectivos dominios.*

### **Derivadas de orden superior y búsqueda de máximos y mínimos**

Por defecto, al crear el objeto derivada mediante la clase Derivative se encuentra la derivada de primer orden. Para encontrar derivadas de orden superior, basta con especificar el orden de la derivada a calcular como tercer argumento al crear el objeto Derivative. En esta sección te mostraré cómo utilizar la derivada de primer y segundo orden de la función para encontrar sus máximos y mínimos en un intervalo.

Considera la función x5 - 30x3 + *50x*, definida en el dominio [-5, 5]. Observa que he utilizado corchetes para indicar un dominio cerrado, lo que indica que la variable *x* puede asumir cualquier valor real mayor o igual que -5 y menor o igual que 5 (ver [Figura 7-3](ch07.html#ch7fig3)).



Figura*7-3: Gráfica de la funciónx5-30x3* + *50x, donde -5* ≤ x ≤ *5*

En la gráfica podemos ver que la función alcanza su valor mínimo en el intervalo -2 ≤ *x* ≤ 0 en el punto *B*. Del mismo modo, alcanza su valor máximo en el intervalo 0 ≤ *x* ≤ 2 en el punto *C*. Por otra parte, la función alcanza sus valores máximo y mínimo en todo el dominio de *x* que hemos considerado aquí en los puntos *A* y *D*, respectivamente. Así, cuando consideramos la función en todo el intervalo [-5, 5], los puntos *B* y *C* se denominan *mínimo local* y *máximo local*, respectivamente, mientras que los puntos *A* y *D* son el *máximo global* y el *mínimo global*, respectivamente.

El término *extremo* (plural *extremos*) se refiere a los puntos en los que la función alcanza un máximo o un mínimo local o global. Si *x* es un extremo de la función *f(x*), entonces la derivada de primer orden de *f* en *x*, denominada *f′(x*), debe desaparecer. Esta propiedad muestra que una buena forma de encontrar posibles extremos es intentar resolver la ecuación *f′(x*) = 0. Tales soluciones se denominan *puntos críticos* de la función. Vamos a probarlo:

>>> from sympy import Symbol, solve, Derivative  
>>> x = Symbol('x')  
>>> f = x\*\*5 - 30\*x\*\*3 + 50\*x  
>>> d1 = Derivative(f, x).doit()

Ahora que hemos calculado la derivada de primer orden, *f′*(*x*), vamos a resolver *f′*(*x*) = 0 para encontrar los puntos críticos:

>>> critical\_points = solve(d1)  
>>> critical\_points  
[-sqrt(-sqrt(71) + 9), sqrt(-sqrt(71) + 9), -sqrt(sqrt(71) + 9),  
sqrt(sqrt(71) + 9)]

Los números de la lista critical\_points que se muestran aquí corresponden a los puntos *B*, *C*, *A* y *D*, respectivamente. Crearemos etiquetas para referirnos a estos puntos, y luego podremos utilizar las etiquetas en nuestros comandos:

>>> A = critical\_points[2]  
>>> B = critical\_points[0]  
>>> C = critical\_points[1]  
>>> D = critical\_points[3]

Como todos los puntos críticos de esta función se encuentran dentro del intervalo considerado, todos ellos son relevantes para nuestra búsqueda del máximo y mínimo globales de *f(x*). Ahora podemos aplicar la llamada *prueba de la segunda derivada* para acotar qué puntos críticos podrían ser máximos o mínimos globales.

En primer lugar, calculamos la derivada de segundo orden de la función *f*(*x*). Observa que para ello introducimos 2 como tercer argumento:

>>> d2 = Derivative(f, x, 2).doit()

Ahora, hallamos el valor de la segunda derivada sustituyendo el valor de cada uno de los puntos críticos uno a uno en lugar de *x*. Si el valor resultante es menor que 0, el punto es un máximo local; si el valor es mayor que 0, es un mínimo local. Si el valor resultante es 0, la prueba no es concluyente y no podemos deducir nada sobre si el punto crítico *x* es un mínimo local, un máximo o ninguno de los dos.

>>> d2.subs({x:B}).evalf()  
127.661060789073  
>>> d2.subs({x:C}).evalf()  
-127.661060789073  
>>> d2.subs({x:A}).evalf()  
-703.493179468151  
>>> d2.subs({x:D}).evalf()  
703.493179468151

La evaluación de la prueba de la segunda derivada en los puntos críticos nos dice que los puntos *A* y *C* son máximos locales y los puntos *B* y *D* son mínimos locales.

El máximo y el mínimo globales de *f(x*) en el intervalo [-5, 5] se alcanzan en un punto crítico *x* o en uno de los puntos extremos del dominio*(x* = -5 y *x* = 5). Ya hemos encontrado todos los puntos críticos, que son los puntos *A*, *B*, *C* y *D*. La función no puede alcanzar su mínimo global en ninguno de los puntos críticos *A* o *C* porque son máximos locales. Por lógica similar, la función no puede alcanzar su máximo global en *B* ni en *D*.

Por lo tanto, para encontrar el máximo global, debemos calcular el valor de *f(x*) en los puntos *A*, *C*, -5 y 5. Entre estos puntos, el lugar donde f*(*x *)* tenga el mayor valor debe ser el máximo global.

Crearemos dos etiquetas, x\_min y x\_max, para referirnos a los límites del dominio y evaluaremos la función en los puntos A, C, x\_min y x\_max:

>>> x\_min = -5  
>>> x\_max = 5  
  
>>> f.subs({x:A}).evalf()  
705.959460380365  
>>> f.subs({x:C}).evalf()  
25.0846626340294  
>>> f.subs({x:x\_min}).evalf()  
375.000000000000  
>>> f.subs({x:x\_max}).evalf()  
-375.000000000000

Mediante estos cálculos, así como examinando el valor de la función en todos los puntos críticos y en los límites del dominio[(Figura 7-3](ch07.html#ch7fig3)), vemos que el punto *A* resulta ser el máximo global.

Del mismo modo, para determinar el mínimo global, debemos calcular los valores de *f(x*) en los puntos *B*, *D*, -5 y 5:

>>> f.subs({x:B}).evalf()  
-25.0846626340294  
>>> f.subs({x:D}).evalf()  
-705.959460380365  
>>> f.subs({x:x\_min}).evalf()  
375.000000000000  
>>> f.subs({x:x\_max}).evalf()  
-375.000000000000

El punto donde *f*(*x*) tiene el valor más pequeño debe ser el mínimo global de la función; éste resulta ser el punto *D*.

Este método para hallar los extremos de una función -considerando el valor de la función en todos los puntos críticos (tras descartar potencialmente algunos mediante la prueba de la segunda derivada) y los valores límite- funcionará siempre que la función sea doblemente diferenciable. Es decir, tanto la primera como la segunda derivada deben existir en todas las partes del dominio.

Para una función como *ex*, puede que no haya ningún punto crítico en el dominio, pero en este caso el método funciona bien: simplemente nos dice que los extremos se dan en la frontera del dominio.

### **Encontrar el máximo global mediante el ascenso del gradiente**

A veces sólo nos interesa encontrar el máximo global de una función en lugar de todos los máximos y mínimos locales y globales. Por ejemplo, podemos querer descubrir el ángulo de proyección para el que una bola recorrerá la máxima distancia horizontal. Vamos a aprender un nuevo enfoque más práctico para resolver un problema de este tipo. Este enfoque hace uso únicamente de la primera derivada, por lo que sólo es aplicable a funciones para las que pueda calcularse la primera derivada.

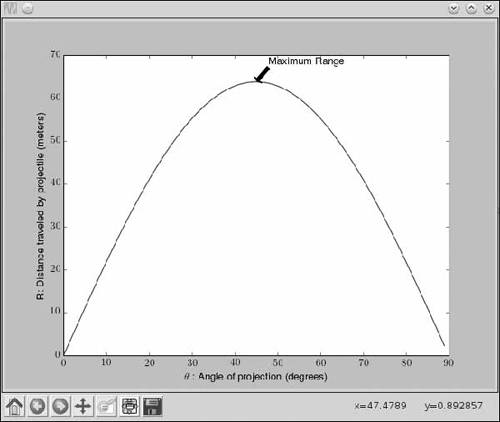
Este método se denomina *método* del gradiente ascendente, que es un enfoque iterativo para encontrar el máximo global. Como el método del ascenso gradiente implica muchos cálculos, es perfecto para resolverlo mediante programación y no a mano. Vamos a probarlo utilizando el problema de ejemplo de encontrar el ángulo de proyección. En el [Capítulo 2](ch02.html#ch02), dedujimos la expresión

image

para calcular el tiempo de vuelo de un cuerpo en movimiento de proyectil que es lanzado con una velocidad *u* a un ángulo *θ*. El *alcance* de un proyectil, *R*, es la distancia horizontal total recorrida por el proyectil y viene dada por el producto de *ux* *×*tflight. Aquí, *ux* es la componente horizontal de la velocidad inicial y es igual a *u* *cosθ*. Sustituyendo las fórmulas de *ux* y tflight, obtenemos la expresión

image

El gráfico de [la Figura 7-4](ch07.html#ch7fig4) muestra valores de *θ* entre 0 y 90 grados y el alcance correspondiente (distancia recorrida) para cada ángulo. En el gráfico podemos ver que el alcance máximo se obtiene cuando el ángulo de proyección es de unos 45 grados. Ahora aprenderemos a utilizar el método del gradiente ascendente para hallar numéricamente este valor de *θ*.



*Figura 7-4: Alcance de un proyectil lanzado con una velocidad inicial de 25 m/s con distintos ángulos de proyección*

El método de ascenso por gradiente es un método iterativo: empezamos con un valor inicial de θ *-digamos*, 0,001, o θold = 0,001- y nos acercamos gradualmente al valor de *θ* que corresponde al alcance máximo[(Figura 7-5](ch07.html#ch7fig5)). El paso que nos acerca es la ecuación

image

donde *λ* es el *tamaño del paso* y

image

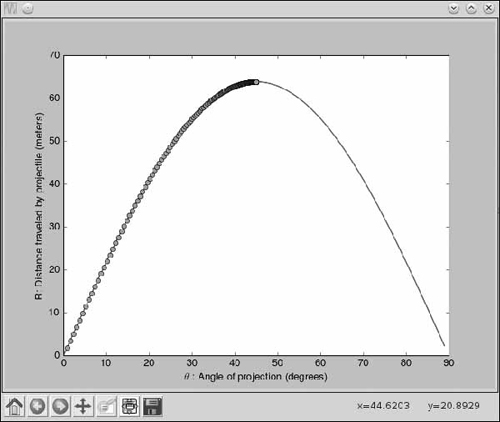
es la derivada de *R* respecto a *θ*. Una vez fijado θold = 0,001, hacemos lo siguiente

1. Calcula θnuevo utilizando la ecuación anterior.

2. Si la diferencia absoluta θnew - θold es mayor que un valor, *ε*, fijamos θold = θnew y volvemos al paso 1. En caso contrario, vamos al paso 3.

3. θnew es un valor aproximado de *θ* para el que *R* tiene el valor máximo.

El valor de *épsilon(ε*) determina cuándo decidimos detener la iteración del algoritmo. Se trata en "[El papel del tamaño del paso y](ch07.html#ch07lev2sec09)épsilon" en [la página 197](ch07.html#page_197).



*Figura 7-5: El método de ascenso gradiente nos lleva iterativamente hacia el punto máximo de la función.*

La siguiente función grad\_ascent() implementa el algoritmo de ascenso gradiente. El parámetro x0 es el valor inicial de la variable en el que comenzar la iteración, f1x es la derivada de la función cuyo máximo queremos encontrar, y x es el objeto Symbol correspondiente a la variable de la función.

'''  
Use gradient ascent to find the angle at which the projectile  
has maximum range for a fixed velocity, 25 m/s  
'''  
  
import math  
from sympy import Derivative, Symbol, sin  
  
def grad\_ascent(x0, f1x, x):  
➊ epsilon = 1e-6  
➋ step\_size = 1e-4  
➌ x\_old = x0  
➍ x\_new = x\_old + step\_size\*f1x.subs({x:x\_old}).evalf()  
➎ while abs(x\_old - x\_new) > epsilon:  
x\_old = x\_new  
x\_new = x\_old + step\_size\*f1x.subs({x:x\_old}).evalf()  
  
return x\_new  
  
➏ def find\_max\_theta(R, theta):  
# Calculate the first derivative  
R1theta = Derivative(R, theta).doit()  
theta0 = 1e-3  
theta\_max = grad\_ascent(theta0, R1theta, theta)  
➐ return theta\_max  
  
if \_\_name\_\_ == '\_\_main\_\_':  
  
g = 9.8  
# Assume initial velocity  
u = 25  
# Expression for range  
theta = Symbol('theta')  
➑ R = u\*\*2\*sin(2\*theta)/g  
  
➒ theta\_max = find\_max\_theta(R, theta)  
print('Theta: {0}'.format(math.degrees(theta\_max)))  
print('Maximum Range: {0}'.format(R.subs({theta:theta\_max})))

Establecemos el valor épsilon en 1e-6 y el tamaño del paso en 1e-4 en ➊ y ➋, respectivamente. El valor épsilon debe ser siempre un valor positivo muy pequeño cercano a 0, y el tamaño del paso debe elegirse de forma que la variable se incremente en pequeñas cantidades en cada iteración del algoritmo. La elección del valor de épsilon y del tamaño del paso se trata con un poco más de detalle en "[El papel del tamaño del paso y de épsilon](ch07.html#ch07lev2sec09)" en [la página 197](ch07.html#page_197).

Ajustamos x\_old a x0 en ➌ y calculamos x\_new por primera vez en ➍. Utilizamos el método subs() para sustituir el valor de x\_old en lugar de la variable y luego utilizamos evalf() para calcular el valor numérico. Si la diferencia absoluta abs(x\_old – x\_new) es mayor que epsilon, el bucle while en ➎ sigue ejecutándose, y seguimos actualizando el valor de x\_old y x\_new según los pasos 1 y 2 del algoritmo de ascenso gradiente. Una vez que salimos del bucle -es decir, de abs(x\_old – x\_new) > epsilon- devolvemos x\_new, el valor de la variable correspondiente al valor máximo de la función.

Empezamos a definir la función find\_max\_theta() en ➏. En esta función, calculamos la derivada de primer orden de R; creamos una etiqueta, theta0, y la establecemos en 1e-3; y llamamos a la función grad\_ascent() con estos dos valores como argumentos, además de un tercer argumento, el objeto símbolo theta. Una vez obtenido el valor de *θ* correspondiente al valor máximo de la función (theta\_max), lo devolvemos en ➐.

Por último, creamos la expresión que representa el rango horizontal en ➑, habiendo establecido la velocidad inicial, u = 25, y el objeto símbolo theta correspondiente al ángulo *θ*. A continuación, llamamos a la función find\_max\_theta() con R y theta en ➒.

Cuando ejecutes este programa, deberías ver la siguiente salida:

Theta: 44.99999978475661  
Maximum Range: 63.7755102040816

El valor de *θ* se imprime en grados y resulta ser próximo a 45 grados, como era de esperar. Si cambias la velocidad inicial por otros valores, verás que el ángulo de proyección en el que se alcanza el alcance máximo es siempre próximo a 45 grados.

#### ***Un programa genérico para el ascenso por gradiente***

Podemos modificar ligeramente el programa anterior para hacer un programa genérico de ascenso por gradiente:

'''  
Use gradient ascent to find the maximum value of a  
single-variable function  
'''  
  
from sympy import Derivative, Symbol, sympify  
  
def grad\_ascent(x0, f1x, x):  
epsilon = 1e-6  
step\_size = 1e-4  
x\_old = x0  
x\_new = x\_old + step\_size\*f1x.subs({x:x\_old}).evalf()  
while abs(x\_old - x\_new) > epsilon:  
x\_old = x\_new  
x\_new = x\_old + step\_size\*f1x.subs({x:x\_old}).evalf()  
  
return x\_new  
  
if \_\_name\_\_ == '\_\_main\_\_':  
  
f = input('Enter a function in one variable: ')  
var = input('Enter the variable to differentiate with respect to: ')  
var0 = float(input('Enter the initial value of the variable: '))  
try:  
f = sympify(f)  
except SympifyError:  
print('Invalid function entered')  
else:  
➊ var = Symbol(var)  
➋ d = Derivative(f, var).doit()  
➌ var\_max = grad\_ascent(var0, d, var)  
print('{0}: {1}'.format(var.name, var\_max))  
print('Maximum value: {0}'.format(f.subs({var:var\_max})))

La función grad\_ascent() sigue siendo la misma. Ahora, sin embargo, el programa pide al usuario que introduzca la función, la variable de la función y el valor inicial de la variable, donde comenzará el ascenso gradiente. Una vez que estamos seguros de que SymPy puede reconocer la entrada del usuario, creamos un objeto Símbolo correspondiente a la variable en ➊, hallamos la primera derivada con respecto a ella en ➋, y llamamos a la función grad\_ascent() con estos tres argumentos. El valor máximo se obtiene en ➌.

Aquí tienes un ejemplo de ejecución:

Enter a function in one variable: 25\*25\*sin(2\*theta)/9.8  
Enter the variable to differentiate with respect to: theta  
Enter the initial value of the variable: 0.001  
theta: 0.785360029379083  
Maximum value: 63.7755100185965

La entrada de la función es la misma que en nuestra primera implementación del gradiente ascendente, y el valor de *θ* se imprime en radianes.

Aquí tienes otra ejecución del programa, que encontrará el valor máximo de *cosy*:

Enter a function in one variable: cos(y)  
Enter the variable to differentiate with respect to: y  
Enter the initial value of the variable: 0.01  
y: 0.00999900001666658  
Maximum value: 0.999950010415832

El programa también funciona correctamente para una función como cos(y) + k, donde k es una constante:

Enter a function in one variable: cos(y) + k  
Enter the variable to differentiate with respect to: y  
Enter the initial value of the variable: 0.01  
y: 0.00999900001666658  
Maximum value: k + 0.999950010415832

Sin embargo, una función como cos(ky) no funcionará porque su derivada de primer orden, kcos(ky), aún contiene k, y SymPy no sabe nada sobre su valor. Por lo tanto, SymPy no puede realizar un paso clave en el algoritmo de ascenso gradiente, a saber, la comparación abs(x\_old - x\_new) > epsilon.

#### ***Advertencia sobre el valor inicial***

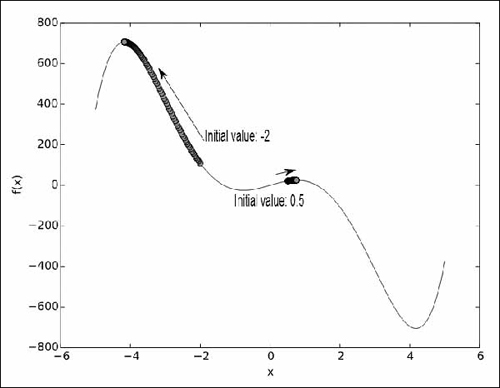
El valor inicial de la variable a partir de la cual iniciamos la iteración del método de ascenso gradiente desempeña un papel muy importante en el algoritmo. Consideremos la función x5 - 30x3 + *50x*, que hemos utilizado como ejemplo en [la Figura 7-3](ch07.html#ch7fig3). Vamos a encontrar el máximo utilizando nuestro programa genérico de ascenso por gradiente:

Enter a function in one variable: x\*\*5 - 30\*x\*\*3 + 50\*x  
Enter the variable to differentiate with respect to: x  
Enter the initial value of the variable: -2  
x: -4.17445116397103  
Maximum value: 705.959460322318

El algoritmo de ascenso por gradiente se detiene cuando encuentra el *pico más cercano*, que no siempre es el máximo global. En este ejemplo, cuando parte del valor inicial -2, se detiene en el pico que también corresponde al máximo global (aproximadamente 706) en el dominio considerado. Para comprobarlo mejor, probemos con otro valor inicial:

Enter a function in one variable: x\*\*5 - 30\*x\*\*3 + 50\*x  
Enter the variable to differentiate with respect to: x  
Enter the initial value of the variable: 0.5  
x: 0.757452532565767  
Maximum value: 25.0846622605419

En este caso, el pico más cercano en el que se detiene el algoritmo de ascenso gradiente no es el verdadero máximo global de la función. [La figura 7-6](ch07.html#ch7fig6) muestra el resultado del algoritmo de ascenso por gradiente en ambos casos.



*Figura 7-6: Resultados del algoritmo de ascenso por gradiente con distintos valores iniciales. El ascenso por gradiente siempre nos lleva al pico más cercano.*

Por tanto, al utilizar este método, el valor inicial debe elegirse con cuidado. Algunas variaciones del algoritmo intentan resolver esta limitación.

#### ***El papel del tamaño del paso y el épsilon***

En el algoritmo de ascenso gradiente, el siguiente valor de la variable se calcula mediante la ecuación

image

donde *λ* es el tamaño del *paso*. El tamaño del paso determina la distancia del siguiente paso. Debe ser pequeño para evitar *pasar por encima de* un pico. Es decir, si el valor actual de *x* está cerca del valor que corresponde al valor máximo de la función, el siguiente paso no debería ir más allá del pico. En ese caso, el algoritmo no tendrá éxito. Por otra parte, los valores muy pequeños tardarán más en calcularse. Hemos utilizado un tamaño de paso fijo de 10-3, pero puede que éste *no* sea el valor más adecuado para todas las funciones.

El valor de épsilon*(ε*) que determina cuándo decidimos detener la iteración del algoritmo debe ser un valor lo suficientemente pequeño como para que estemos convencidos de que el valor de *x* no está cambiando. Esperamos que la primera derivada, *f′(x*), sea 0 en el punto máximo, y lo ideal es que la diferencia absoluta |θnew - θold| sea 0 (véase el paso 2 del algoritmo de ascenso gradiente en [la página 192](ch07.html#page_192)). Sin embargo, debido a imprecisiones numéricas, puede que no obtengamos exactamente una diferencia de 0; por ello, el valor de épsilon se elige para que sea un valor cercano a 0, que, a efectos prácticos, nos diría que el valor de *x* ya no está cambiando. He utilizado 10-6 como épsilon para todas las funciones. Este valor, aunque suficientemente pequeño y adecuado para las funciones que tienen solución para *f′(x*) = 0, como sin(x), puede no ser el valor adecuado para otras funciones. Por lo tanto, es una buena idea verificar el valor máximo al final para asegurarse de que es correcto y, si es necesario, ajustar el valor de epsilon en consecuencia.

El paso 2 del algoritmo de ascenso gradiente también implica que, para que el algoritmo termine, la ecuación *f′(x*) = 0 debe tener solución, lo que no ocurre con una función como *ex* o log*(x*). Por tanto, si proporcionas una de estas funciones como entrada al programa anterior, el programa no te dará una solución y seguirá ejecutándose. Podemos hacer que el programa de ascenso gradiente sea más útil para estos casos incorporando una comprobación para saber si *f′(x*) = 0 tiene solución. Aquí tienes el programa modificado:

'''  
Use gradient ascent to find the maximum value of a  
single-variable function. This also checks for the existence  
of a solution for the equation f'(x)=0.  
'''  
  
from sympy import Derivative, Symbol, sympify, solve  
  
def grad\_ascent(x0, f1x, x):  
# Check if f1x=0 has a solution  
➊ if not solve(f1x):  
print('Cannot continue, solution for {0}=0 does not exist'.format(f1x))  
return  
epsilon = 1e-6  
step\_size = 1e-4  
x\_old = x0  
x\_new = x\_old + step\_size\*f1x.subs({x:x\_old}).evalf()  
while abs(x\_old - x\_new) > epsilon:  
x\_old = x\_new  
x\_new = x\_old + step\_size\*f1x.subs({x:x\_old}).evalf()  
  
return x\_new  
  
if \_\_name\_\_ == '\_\_main\_\_':  
  
f = input('Enter a function in one variable: ')  
var = input('Enter the variable to differentiate with respect to: ')  
var0 = float(input('Enter the initial value of the variable: '))  
try:  
f = sympify(f)  
except SympifyError:  
print('Invalid function entered')  
else:  
var = Symbol(var)  
d = Derivative(f, var).doit()  
var\_max = grad\_ascent(var0, d, var)  
➋ if var\_max:  
print('{0}: {1}'.format(var.name, var\_max))  
print('Maximum value: {0}'.format(f.subs({var:var\_max})))

En esta modificación de la función grad\_ascent(), llamamos a la función solve() de SymPy en ➊ para determinar si la ecuación *f′*(*x*) = 0, aquí f1x, tiene solución. Si no es así, imprimimos un mensaje y volvemos. Otra modificación aparece en el bloque \_\_main\_\_ en ➋. Comprobamos si la función grad\_ascent() ha devuelto correctamente un resultado; si es así, procedemos a imprimir el valor máximo de la función y el valor correspondiente de la variable.

Estos cambios permiten al programa manejar funciones como log*(x*) y *ex*:

Enter a function in one variable: log(x)  
Enter the variable to differentiate with respect to: x  
Enter the initial value of the variable: 0.1  
Cannot continue, solution for 1/x=0 does not exist

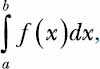
Verás lo mismo para *ex*.

**ALGORITMO DE DESCENSO DE GRADIENTE**

El algoritmo inverso del algoritmo de ascenso por gradiente es el algoritmo de *descenso* por gradiente, que es un método para encontrar el valor mínimo de una función. Es similar al algoritmo de ascenso por gradiente, pero en lugar de "subir" por la función, "bajamos". El Desafío nº 2 de la [página 205](ch07.html#page_205) analiza la diferencia entre estos dos algoritmos y te da la oportunidad de poner en práctica el inverso.

### **Hallar las integrales de funciones**

La *integral indefinida*, o la *antiderivada*, de una función *f(x*) es otra función *F(x*), tal que *F′(x*) = *f(x*). Es decir, la integral de una función es otra función cuya derivada es la función original. Matemáticamente, se escribe como *F(x*) = ∫ *f(x)dx*. La *integral definida*, por su parte, es la integral



que en realidad es *F*(*b*) - *F(a*), donde *F(b*) y *F(a*) son los valores de la antiderivada de la función en *x* = *b* y en *x* = *a*, respectivamente. Podemos hallar ambas integrales creando un objeto Integral.

Así es como podemos hallar la integral ∫ *kxdx*, donde *k* es un término constante:

>>> from sympy import Integral, Symbol  
>>> x = Symbol('x')  
>>> k = Symbol('k')  
>>> Integral(k\*x, x)  
Integral(k\*x, x)

Importamos las clases Integral y Symbol y creamos dos objetos Symbol correspondientes a k y x. A continuación, creamos un objeto Integral con la función kx, especificando la variable a integrar con respecto a x. De forma similar a las clases Limit y Derivative, ahora podemos evaluar la integral utilizando el método doit():

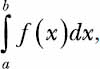
>>> Integral(k\*x, x).doit()  
k\*x\*\*2/2

La integral resulta ser kx2/2. Si calculas la derivada de kx2/2, obtendrás la función original, *kx*.

Para hallar la integral *definida*, basta con especificar la variable, el límite inferior y el límite superior como una tupla al crear el objeto Integral:

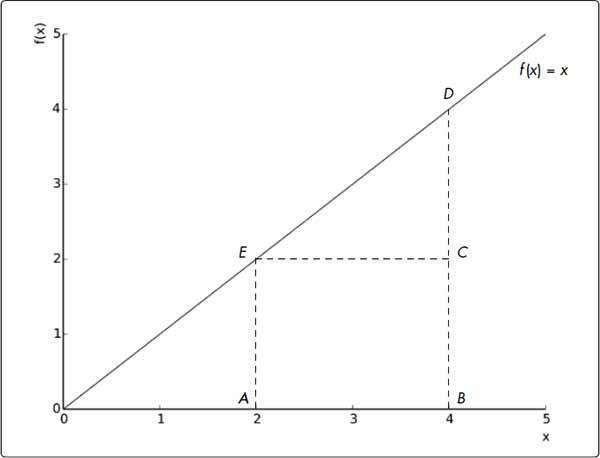
>>> Integral(k\*x, (x, 0, 2)).doit()  
2\*k

El resultado devuelto es la integral definida



Puede ser útil visualizar las integrales definidas analizándolas en un contexto geométrico. Considera [la Figura 7-7](ch07.html#ch7fig7), que muestra la gráfica de la función *f(x*) = *x* entre *x* = 0 y *x* = 5.

Considera ahora la región bajo la gráfica *ABDE*, que está delimitada por el *eje x*, entre los puntos *x* = 2 y *x* = 4 puntos *A* y *B*, respectivamente. El área de la región se puede hallar sumando el área del cuadrado *ABCE* y el triángulo rectángulo *ECD*, que es 2 × 2 + (1/2) × 2 × 2 = 6.



*Figura 7-7: La integral definida de una función entre dos puntos es el área encerrada por la gráfica de la función limitada por el* eje x*.*

Calculemos ahora la integral image:

>>> from sympy import Integral, Symbol  
>>> x = Symbol('x')  
>>> Integral(x, (x, 2, 4)).doit()  
6

El valor de la integral resulta ser el mismo que el área de la región *ABDE*. Esto no es una coincidencia; verás que esto es cierto para cualquier función de *x* para la que se pueda determinar la integral.

Comprender que la integral definida es el área encerrada por la función entre puntos especificados del *eje* x es clave para entender los cálculos de probabilidad en sucesos aleatorios que implican variables aleatorias continuas.

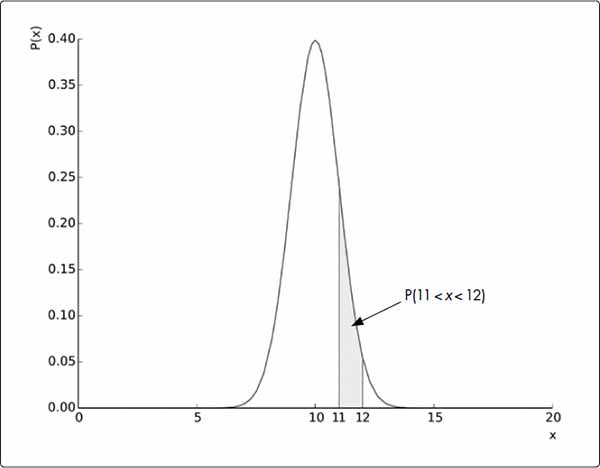
### **Funciones de densidad de probabilidad**

Consideremos una clase ficticia de estudiantes y sus calificaciones en un examen de matemáticas. Cada alumno puede obtener una nota entre 0 y 20, incluidas las notas fraccionarias. Si tratamos la nota como un suceso aleatorio, la nota en sí es una *variable aleatoria continua* porque puede tener *cualquier* valor entre 0 y 20. Si en queremos calcular la probabilidad de que un alumno obtenga una nota entre 11 y 12, no podemos aplicar la estrategia que aprendimos en el [Capítulo 5](ch05.html#ch05). Para ver por qué, consideremos la fórmula, suponiendo una probabilidad uniforme,

image

donde *E* es el conjunto de todas las calificaciones posibles entre 11 y 12 y *S* es el conjunto de todas las calificaciones posibles, es decir, todos los números reales entre 1 y 20. Según nuestra definición del problema anterior, *n(E*) es infinito porque es imposible contar todos los números reales posibles entre 11 y 12; lo mismo ocurre con *n(S*). Por tanto, necesitamos un enfoque diferente para calcular la probabilidad.

Una *función de densidad de probabilidad*, *P(x*), expresa la probabilidad de que el valor de una variable aleatoria se *aproxime* a *x*, un valor arbitrario[.1](footnote.html#fn05) También puede indicarnos la probabilidad de que *x* caiga dentro de un intervalo. Es decir, si conociéramos la función de densidad de probabilidad que representa la probabilidad de las notas en nuestra clase ficticia, calcular *P*(11 < *x* < 12) nos daría la probabilidad que buscamos. Pero, ¿cómo la calculamos? Resulta que esta probabilidad es el área encerrada por la gráfica de la función de densidad de probabilidad y el *eje x* entre los puntos *x* = 11 y *x* = 12. Suponiendo una función de densidad de probabilidad arbitraria, [la Figura 7-8](ch07.html#ch7fig8) lo demuestra.

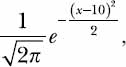


*Figura 7-8: Una función de densidad de probabilidad para las notas de un examen de matemáticas*

Ya sabemos que esta área es igual al valor de la integral,

image

por tanto, tenemos una forma fácil de hallar la probabilidad de que la nota esté entre 11 y 12. Con las matemáticas fuera del camino, ahora podemos averiguar cuál es la probabilidad. La función de densidad de probabilidad que hemos supuesto antes es la función



donde *x* es la nota obtenida. Esta función se ha elegido de modo que la probabilidad de que la nota sea próxima a 10 (mayor o menor que) sea alta, pero luego disminuya bruscamente.

Ahora, vamos a calcular la integral

image

siendo *p(x*) la función anterior:

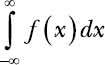
>>> from sympy import Symbol, exp, sqrt, pi, Integral  
>>> x = Symbol('x')  
>>> p = exp(-(x - 10)\*\*2/2)/sqrt(2\*pi)  
>>> Integral(p, (x, 11, 12)).doit().evalf()  
0.135905121983278

Creamos el objeto Integral para la función, con p representando la función de densidad de probabilidad que especifica que queremos calcular la integral definida entre 11 y 12 en el *eje x*. Evaluamos la función con doit() y hallamos el valor numérico con evalf(). Así, la probabilidad de que una nota se sitúe entre 11 y 12 es cercana a 0,14.

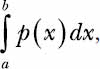
**LA FUNCIÓN DE DENSIDAD DE PROBABILIDAD: UNA ADVERTENCIA**

En sentido estricto, esta función de densidad asigna una probabilidad distinta de cero a las notas inferiores a 0 o superiores a 20. Sin embargo, como puedes comprobar utilizando las ideas de esta sección, la probabilidad de que se produzca este hecho es tan pequeña que resulta despreciable para nuestros fines.

Una función de densidad de probabilidad tiene dos propiedades especiales: (1) el valor de la función para cualquier *x* es siempre mayor que 0, ya que la probabilidad no puede ser menor que 0, y (2) el valor de la integral definida



es igual a 1. La segunda propiedad merece cierta discusión. Como *p(x*) es una función de densidad de probabilidad, el área encerrada por ella, que es también la integral

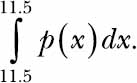


entre dos puntos cualesquiera, *x* = *a* y *x* = *b*, nos da la probabilidad de que *x* se encuentre entre *x* = *a* y *x* = *b*. Esto significa también que, sean cuales sean los valores de *a* y *b*, el valor de la integral no debe ser superior a 1, porque la probabilidad no puede ser mayor que 1 por definición. Por tanto, aunque *a* y *b* sean valores muy grandes, de modo que tiendan a -∞ y ∞, respectivamente, el valor de la integral seguirá siendo 1, como podemos comprobar nosotros mismos:

>>> from sympy import Symbol, exp, sqrt, pi, Integral, S  
>>> x = Symbol('x')  
>>> p = exp(-(x – 10)\*\*2/2)/sqrt(2\*pi)  
>>> Integral(p, (x, S.NegativeInfinity, S.Infinity)).doit().evalf()  
1.00000000000000

S.NegativeInfinity y S.Infinity denotan el infinito negativo y positivo que luego especificamos como límites inferior y superior, respectivamente, al crear el objeto Integral.

Cuando tratamos con variables aleatorias continuas, puede surgir una situación complicada. En probabilidad discreta, la probabilidad de que ocurra un suceso como que un dado justo de seis caras saque un 7 es 0. A un suceso cuya probabilidad es 0 lo llamamos suceso *imposible*. En el caso de las variables aleatorias continuas, la probabilidad de que la variable asuma cualquier valor exacto es 0, aunque sea un suceso *posible*. Por ejemplo, que la nota de un alumno sea exactamente 11,5 es posible, pero debido a la naturaleza de las variables aleatorias continuas, la probabilidad es 0. Para ver por qué, considera que la probabilidad será el valor de la integral



Como esta integral tiene los mismos límites inferior y superior, su valor es 0. Esto es bastante poco intuitivo y paradójico, así que intentemos comprenderlo.

Considera el intervalo de calificaciones que hemos abordado antes: de 0 a 20. La nota que puede obtener un alumno puede ser cualquier número de este intervalo, lo que significa que hay un número infinito de números. Si cada número tuviera la misma probabilidad de ser seleccionado, ¿cuál sería esa probabilidad? Según la fórmula de la probabilidad discreta, debería ser 1/∞, es decir, un número muy pequeño. De hecho, este número es tan pequeño que, a efectos prácticos, se considera 0. Por lo tanto, la probabilidad de que la nota sea 11,5 es 0.

### **Lo que has aprendido**

En este capítulo has aprendido a hallar los límites, las derivadas y las integrales de funciones. Has conocido el método de ascenso por gradiente para hallar el valor máximo de una función y has visto cómo puedes aplicar los principios de integración para calcular la probabilidad de variables aleatorias continuas. A continuación, tienes algunas tareas que intentar.

### **Retos de programación**

Los siguientes retos se basan en lo que has aprendido en este capítulo. Puedes encontrar ejemplos de soluciones en [*http://www.nostarch.com/doingmathwithpython/*](http://www.nostarch.com/doingmathwithpython/).

#### ***#nº 1: Verificar la continuidad de una función en un punto***

Una condición necesaria, pero no suficiente, para que una función sea diferenciable en un punto es que sea continua en ese punto. Es decir, la función debe estar definida en ese punto y su límite izquierdo y su límite derecho deben existir y ser iguales al valor de la función en ese punto. Si *f(x*) es la función y *x* = *a* es el punto que nos interesa evaluar, esto se expresa matemáticamente como

image

Tu reto aquí es escribir un programa que (1) acepte una función de una sola variable y un valor de esa variable como entradas y (2) compruebe si la función de entrada es continua en el punto en el que la variable asume el valor de entrada.

Aquí tienes un ejemplo de funcionamiento de la solución completada:

Enter a function in one variable: 1/x  
Enter the variable: x  
Enter the point to check the continuity at: 1  
1/x is continuous at 1.0

La función *1/x* es discontinua en 0, así que vamos a comprobarlo:

Enter a function in one variable: 1/x  
Enter the variable: x  
Enter the point to check the continuity at: 0  
1/x is not continuous at 0.0

#### ***#2: Implementar el descenso gradiente***

El método de descenso gradiente se utiliza para encontrar el valor mínimo de una función. Al igual que el método de ascenso por gradiente, el método de descenso por gradiente es un método iterativo: empezamos con un valor inicial de la variable y nos acercamos gradualmente al valor de la variable que corresponde al valor mínimo de la función. El paso que nos acerca es la ecuación

image

donde *λ* es el tamaño del paso y

image

es el resultado de diferenciar la función. Por tanto, la única diferencia con el método de ascenso gradiente es cómo obtenemos el valor de x\_new a partir de x\_old.

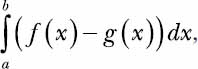
Tu reto es implementar un programa genérico que utilice el algoritmo de descenso gradiente para encontrar el valor mínimo de una función de una sola variable especificada como entrada por el usuario. El programa también debe crear una gráfica de la función y mostrar todos los valores intermedios que encontró antes de hallar el mínimo. (Puedes consultar la [Figura 7-5](ch07.html#ch7fig5) de la [página 193](ch07.html#page_193)).

#### ***#3: Área entre dos curvas***

Hemos aprendido que la integral

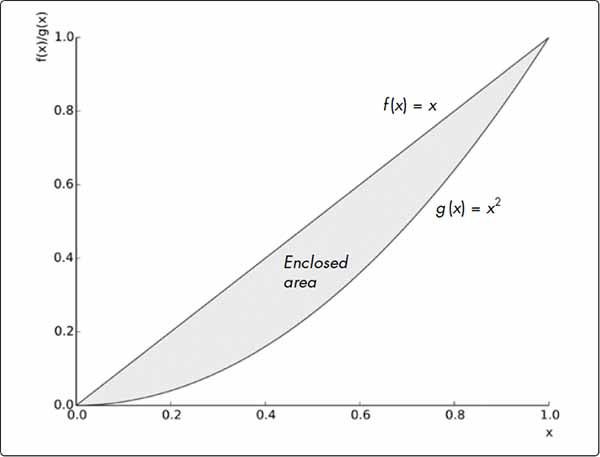


expresa el área encerrada por la función *f(x*), con el *eje x* entre *x* = *a* y *x* = *b*. El área entre dos curvas se expresa, pues, como la integral



donde *a* y *b* son los puntos de intersección de las dos curvas con *a* < *b*. La función *f*(*x*) se denomina *función superior* y *g(x*) *función inferior*. [La figura 7-9](ch07.html#ch7fig9) lo ilustra, suponiendo que *f*(*x*) = *x* y *g(x*) = x2, con *a* = 0 y *b* = 1.

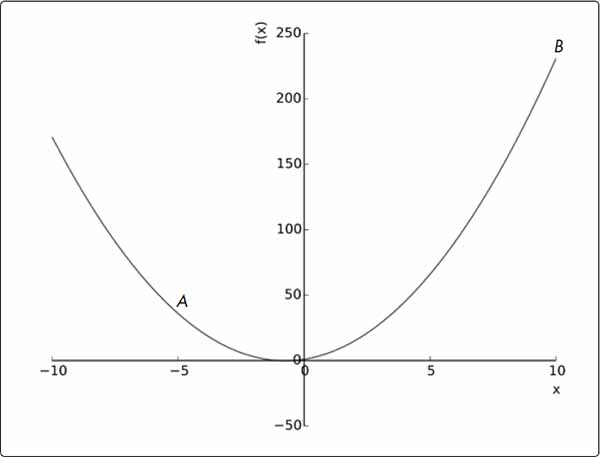
Tu reto aquí es escribir un programa que permita al usuario introducir dos funciones cualesquiera de una sola variable de *x* e imprimir el área encerrada entre ambas. El programa debe dejar claro que la primera función introducida debe ser la superior, y también debe pedir los valores de *x* entre los que hallar el área.



*Figura 7-9: Las funciones* f*(*x*) =* x *y* g*(*x*) =x2* encierran*un* área comprendida entre x = *0 y* x = *1,0.*

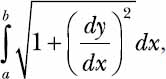
#### ***#4: Hallar la longitud de una curva***

Supongamos que acabas de recorrer en bicicleta una carretera que se parece aproximadamente a [la Figura 7-10](ch07.html#ch7fig10). Como no llevabas cuentakilómetros, quieres saber si hay alguna forma matemática de determinar la distancia que has recorrido en bicicleta. En primer lugar, tendremos que encontrar una ecuación -incluso una aproximación servirá- que describa este camino.



*Figura 7-10: Una aproximación de la trayectoria ciclista*

¿Te das cuenta de que se parece mucho a las funciones cuadráticas de las que hemos hablado en los capítulos anteriores? De hecho, para este reto, vamos a suponer que la ecuación es *y* = *f(x*) = 2x2 + *3x* + 1 y que has ido en bicicleta desde el punto *A* (-5, 36) hasta el punto *B* (10, 231). Para hallar la longitud de este arco -es decir, la distancia que has recorrido- tendremos que calcular la integral



donde *y* describe la función anterior. Tu reto aquí es escribir un programa que calcule la longitud del arco, *AB*.

También puedes generalizar tu solución para que te permita hallar la longitud del arco entre dos puntos cualesquiera para cualquier función arbitraria, *f(x*).