Capítulo 6: Dibujo de formas geométricas y fractales

## **6** **Dibujar** formas geométricas y fractales



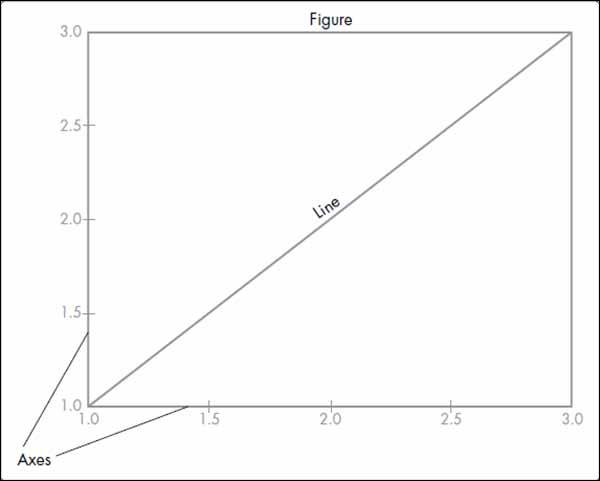
En este capítulo, empezaremos conociendo los parches de matplotlib que nos permiten dibujar formas geométricas, como círculos, triángulos y polígonos. A continuación, conoceremos el soporte de animación de matplotlib y escribiremos un programa para animar la trayectoria de un proyectil. En la última sección, aprenderemos a dibujar *fractales,*formas geométricas *complejas*creadas por la aplicación repetida de transformaciones geométricas simples. ¡Manos a la obra!

### **Dibujar formas geométricas con los parches de Matplotlib**

En matplotlib, *los par* ches nos permiten dibujar formas geométricas, a cada una de las cuales nos referimos como un *parche*. Puedes especificar, por ejemplo, el radio y el centro de un círculo para añadir el círculo correspondiente a tu gráfico. Esto es bastante diferente de cómo hemos utilizado matplotlib hasta ahora, que ha consistido en suministrar las *coordenadas* *x* e *y* de los puntos a trazar. Sin embargo, antes de que podamos escribir un programa para utilizar la función de parches, tendremos que entender un poco mejor cómo se crea un gráfico con matplotlib. Considera el siguiente programa, que traza los puntos (1, 1), (2, 2) y (3, 3) utilizando matplotlib:

>>> import matplotlib.pyplot as plt  
>>> x = [1, 2, 3]  
>>> y = [1, 2, 3]  
>>> plt.plot(x, y)  
[<matplotlib.lines.Line2D object at 0x7fe822d67a20>]  
>>> plt.show()

Este programa crea una ventana matplotlib que muestra una línea que pasa por los puntos dados. En realidad, cuando se llama a la función plt.plot(), se crea un objeto Figure, dentro del cual se crean los ejes y, finalmente, se trazan los datos dentro de los ejes (ver [Figura 6-1](ch06.html#ch6fig1))[.1](footnote.html#fn03)



*Figura 6-1: Arquitectura de un gráfico matplotlib*

El siguiente programa vuelve a crear este gráfico, pero también crearemos explícitamente el objeto Figure y le añadiremos ejes, en lugar de limitarnos a llamar a la función plot() y confiar en ella para crearlos:

>>> import matplotlib.pyplot as plt  
>>> x = [1, 2, 3]  
>>> y = [1, 2, 3]  
➊ >>> fig = plt.figure()  
➋ >>> ax = plt.axes()  
>>> plt.plot(x, y)  
[<matplotlib.lines.Line2D object at 0x7f9bad1dcc18>]  
>>> plt.show()  
>>>

Aquí, creamos el objeto Figure utilizando la función figure() en ➊, y luego creamos los ejes utilizando la función axes() en ➋. La función axes() también añade los ejes al objeto Figure. Las dos últimas líneas son iguales que en el programa anterior. Esta vez, cuando llamamos a la función plot(), ésta ve que ya existe un objeto Figure con un objeto Axes y procede directamente a trazar los datos que se le suministran.

Además de crear manualmente los objetos Figure y Axes, puedes utilizar dos funciones diferentes del módulo pyplot para obtener una referencia a los objetos Figure y Axes actuales. Cuando llamas a la función gcf(), te devuelve una referencia al Figure actual, y cuando llamas a la función gca(), te devuelve una referencia al Axes actual. Una característica interesante de estas funciones es que cada una creará el objeto respectivo si aún no existe. El funcionamiento de estas funciones quedará más claro cuando las utilicemos más adelante en este capítulo.

#### ***Dibujar un círculo***

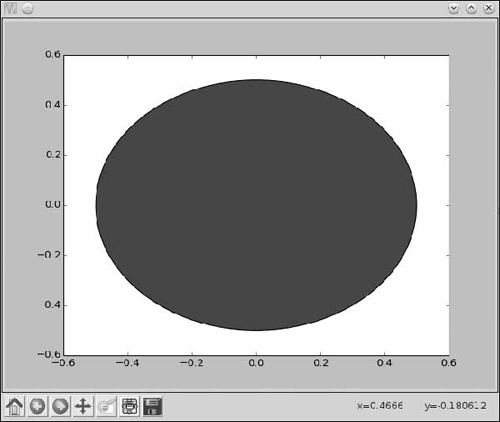
Para dibujar un círculo, puedes añadir el parche Circle al objeto actual Axes, como se demuestra en el siguiente ejemplo:

'''  
Example of using matplotlib's Circle patch  
'''  
import matplotlib.pyplot as plt  
  
def create\_circle():  
➊ circle = plt.Circle((0, 0), radius = 0.5)  
return circle  
  
def show\_shape(patch):  
➋ ax = plt.gca()  
ax.add\_patch(patch)  
plt.axis('scaled')  
plt.show()  
  
if \_\_name\_\_ == '\_\_main\_\_':  
➌ c = create\_circle()  
show\_shape(c)

En este programa, hemos separado la creación del objeto parche Circle y la adición del parche a la figura en dos funciones: create\_circle() y show\_shape(). En create\_circle(), hacemos un círculo con centro en (0, 0) y radio de 0,5 creando un objeto Circle con las coordenadas del centro (0, 0) pasadas como tupla y con el radio de 0,5 pasado mediante el argumento de palabra clave del mismo nombre en ➊. La función devuelve el objeto Circle creado.

La función show\_shape() está escrita de forma que funcione con cualquier parche de matplotlib. Primero obtiene una referencia al objeto Axes actual utilizando la función gca() en ➋. Después, añade el parche que se le ha pasado utilizando la función add\_patch() y, por último, llama a la función show() para mostrar la figura. Aquí llamamos a la función axis() con el parámetro scaled, que básicamente indica a matplotlib que ajuste automáticamente los límites de los ejes. Necesitaremos esta declaración en todos los programas que utilicen parches para escalar automáticamente los ejes. Por supuesto, también puedes especificar valores fijos para los límites, como vimos en el [capítulo 2](ch02.html#ch02).

En ➌, llamamos a la función create\_circle() utilizando la etiqueta c para referirnos al objeto Circle devuelto. A continuación, llamamos a la función show\_shape(), pasando c como argumento. Cuando ejecutes el programa, verás una ventana matplotlib que muestra el círculo (ver [Figura 6-2](ch06.html#ch6fig2)).



*Figura 6-2: Un círculo con centro en (0, 0) y radio de 0,5*

Como puedes ver, el círculo no parece exactamente un círculo. Esto se debe a la relación de aspecto automática, que determina la relación entre la longitud de los *ejes* *x* e *y*. Si insertas la sentencia ax.set\_aspect('equal') después de ➋, verás que el círculo sí parece un círculo. La función set\_aspect() se utiliza para establecer la relación de aspecto del gráfico; utilizando el argumento equal, pedimos a matplotlib que establezca la relación de la longitud de los *ejes* *x* e y en 1:1.

Tanto el color de las aristas como el color de las caras (color de relleno) del parche pueden cambiarse utilizando los argumentos de palabra clave ec y fc. Por ejemplo, si pasas fc='g' y ec='r' crearás un círculo con un color de cara verde y un color de borde rojo.

Matplotlib admite otros parches, como Ellipse, Polygon y Rectangle.

#### ***Crear figuras animadas***

A veces podemos querer crear figuras con formas en movimiento. El soporte para animación de Matplotlib nos ayudará a conseguirlo. Al final de esta sección, crearemos una versión animada del programa de dibujo de trayectorias de proyectiles.

Primero, veamos un ejemplo más sencillo. Dibujaremos una figura de matplotlib con un círculo que empieza siendo pequeño y crece hasta un radio determinado indefinidamente (a menos que se cierre la ventana de matplotlib):

'''  
A growing circle  
'''  
  
from matplotlib import pyplot as plt  
from matplotlib import animation  
  
def create\_circle():  
circle = plt.Circle((0, 0), 0.05)  
return circle  
  
def update\_radius(i, circle):  
circle.radius = i\*0.5  
return circle,  
  
def create\_animation():  
➊ fig = plt.gcf()  
ax = plt.axes(xlim=(-10, 10), ylim=(-10, 10))  
ax.set\_aspect('equal')  
circle = create\_circle()  
➋ ax.add\_patch(circle)  
➌ anim = animation.FuncAnimation(  
fig, update\_radius, fargs = (circle,), frames=30, interval=50)  
plt.title('Simple Circle Animation')  
plt.show()  
  
if \_\_name\_\_ == '\_\_main\_\_':  
create\_animation()

Empezaremos importando el módulo animation del paquete matplotlib. La función create\_animation() realiza aquí la funcionalidad principal. Obtiene una referencia al objeto Figure actual utilizando la función gcf() en ➊ y luego crea los ejes con límites de -10 y 10 tanto para *el* *eje* *x* como para el *eje y*. Después, crea un objeto Circle que represente un círculo con radio 0,05 y centro en (0, 0) y añade este círculo a los ejes actuales en ➋. A continuación, creamos un objeto FuncAnimation ➌, que pasa los siguientes datos sobre la animación que queremos crear:

fig Este es el objeto Figure actual.

update\_radius Esta función se encargará de dibujar *cada* fotograma. Toma dos argumentos: un número de fotograma que se le pasa automáticamente al llamarla y el objeto parche que queremos actualizar cada fotograma. Esta función también debe devolver el objeto.

fargs Esta tupla consta de todos los argumentos que hay que pasar a la función update\_radius() aparte del número de fotograma. Si no hay argumentos que pasar, no es necesario especificar este argumento.

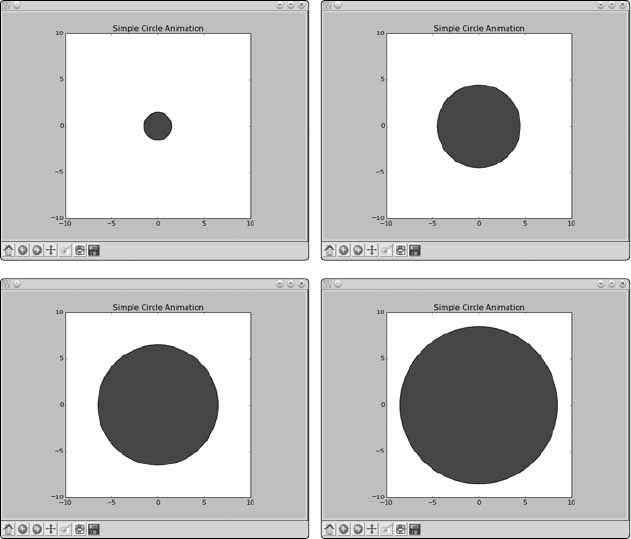
frames Es el número de fotogramas de la animación. Nuestra función update\_radius() es llamada así muchas veces. Aquí, hemos elegido arbitrariamente 30 fotogramas.

interval Este es el intervalo de tiempo en milisegundos entre dos fotogramas. Si tu animación parece demasiado lenta, disminuye este valor; si parece demasiado rápida, auméntalo.

A continuación, establecemos un título mediante la función title() y, por último, mostramos la figura mediante la función show().

Como ya hemos dicho, la función update\_radius() se encarga de actualizar la propiedad del círculo que cambiará en cada fotograma. Aquí, fijamos el radio en i\*0.5, donde i es el número de fotograma. Como resultado, verás un círculo que crece en cada fotograma durante 30 fotogramas, por lo que el radio del círculo más grande es 15. Como los límites de los ejes están fijados en -10 y 10, esto da el efecto de que el círculo sobrepasa las dimensiones de la figura. Cuando ejecutes el programa, verás tu primera figura animada, como se muestra en la [Figura 6-3](ch06.html#ch6fig3).

Observarás que la animación continúa hasta que cierras la ventana de matplotlib. Éste es el comportamiento por defecto, que puedes cambiar estableciendo el argumento de la palabra clave en repeat=False cuando crees el objeto FuncAnimation.



*Figura 6-3: Animación de un círculo simple*

**OBJETO FUNCANIMACIÓN Y PERSISTENCIA**

Probablemente hayas observado en el programa del círculo animado que asignamos el objeto FuncAnimation creado a la etiqueta anim aunque no volvamos a utilizarlo en ningún otro lugar. Esto se debe a un problema con el comportamiento actual de matplotlib: no almacena ninguna referencia al objeto FuncAnimation, por lo que está sujeto a la recolección de basura por parte de Python. Esto significa que la animación no se creará. Crear una etiqueta que haga referencia al objeto evita que esto ocurra.

Para saber más sobre este tema, puedes seguir las discusiones en [*https://github.com/matplotlib/matplotlib/issues/1656/.*](https://github.com/matplotlib/matplotlib/issues/1656/)

#### ***Animar la trayectoria de un proyectil***

En el [Capítulo 2](ch02.html#ch02), dibujamos la trayectoria de una pelota en movimiento de proyectil. Aquí, nos basaremos en este dibujo, haciendo uso del soporte de animación de matplotlib para animar la trayectoria, de modo que se acerque más a la demostración de cómo verías viajar una pelota en la vida real:

'''  
Animate the trajectory of an object in projectile motion  
'''  
  
from matplotlib import pyplot as plt  
from matplotlib import animation  
import math  
  
g = 9.8  
  
def get\_intervals(u, theta):  
  
t\_flight = 2\*u\*math.sin(theta)/g  
intervals = []  
start = 0  
interval = 0.005  
while start < t\_flight:  
intervals.append(start)  
start = start + interval  
return intervals  
  
def update\_position(i, circle, intervals, u, theta):  
  
t = intervals[i]  
x = u\*math.cos(theta)\*t  
y = u\*math.sin(theta)\*t - 0.5\*g\*t\*t  
circle.center = x, y  
return circle,  
  
def create\_animation(u, theta):  
  
intervals = get\_intervals(u, theta)  
  
xmin = 0  
xmax = u\*math.cos(theta)\*intervals[-1]  
ymin = 0  
t\_max = u\*math.sin(theta)/g  
➊ ymax = u\*math.sin(theta)\*t\_max - 0.5\*g\*t\_max\*\*2  
fig = plt.gcf()  
➋ ax = plt.axes(xlim=(xmin, xmax), ylim=(ymin, ymax))  
  
circle = plt.Circle((xmin, ymin), 1.0)  
ax.add\_patch(circle)  
  
➌ anim = animation.FuncAnimation(fig, update\_position,  
fargs=(circle, intervals, u, theta),  
frames=len(intervals), interval=1,  
repeat=False)  
  
plt.title('Projectile Motion')  
plt.xlabel('X')  
plt.ylabel('Y')  
plt.show()  
  
if \_\_name\_\_ == '\_\_main\_\_':  
try:  
u = float(input('Enter the initial velocity (m/s): '))  
theta = float(input('Enter the angle of projection (degrees): '))  
except ValueError:  
print('You entered an invalid input')  
else:  
theta = math.radians(theta)  
create\_animation(u, theta)

La función create\_animation() acepta dos argumentos: u y theta. Estos argumentos corresponden a la velocidad inicial y al ángulo de proyección*(θ*), que se suministraron como entrada al programa. La función get\_intervals() se utiliza para encontrar los intervalos de tiempo en los que calcular las *coordenadas* *x* e *y*. Esta función se implementa haciendo uso de la misma lógica que utilizamos en el [Capítulo 2](ch02.html#ch02), cuando implementamos una función independiente, frange(), para ayudarnos.

Para establecer los límites de los ejes de la animación, necesitaremos encontrar los valores mínimo y máximo de *x* e *y*. El valor mínimo de cada uno es 0, que es el valor inicial de cada uno. El valor máximo de la coordenada *x* es el valor de la coordenada al final del vuelo de la pelota, que es el último intervalo de tiempo de la lista intervals. El valor máximo de la coordenada *y* es cuando la bola está en su punto más alto, es decir, en ➊, donde calculamos ese punto mediante la fórmula

image

Una vez que tenemos los valores, creamos los ejes en ➋, pasando los límites de eje adecuados. En las dos sentencias siguientes, creamos una representación de la bola y la añadimos al objeto Axes de la figura creando un círculo de radio 1.0 en (xmin, ymin)-las coordenadas mínimas de los ejes *x* e *y*, respectivamente.

A continuación, creamos el objeto FuncAnimation ➌, suministrándole el objeto figura actual y los siguientes argumentos:

update\_position Esta función cambiará el centro del círculo en cada fotograma. La idea aquí es que se crea un nuevo fotograma por cada intervalo de tiempo, por lo que fijamos el número de fotogramas al tamaño de los intervalos de tiempo (consulta la descripción de frames en esta lista). Calculamos las *coordenadas* *x e* *y* de la bola en el instante de tiempo en el iintervalo de tiempo, y fijamos el centro del círculo a estos valores.

fargs La función update\_position() necesita acceder a la lista de intervalos de tiempo, intervalos, velocidad inicial y theta, que se especifican mediante este argumento de palabra clave.

frames Como dibujaremos un fotograma por intervalo de tiempo, fijamos el número de fotogramas al tamaño de la lista intervals.

repeat Como ya comentamos en el primer ejemplo de animación, la animación se repite indefinidamente por defecto. No queremos que eso ocurra en este caso, así que fijamos esta palabra clave en False.

Cuando ejecutes el programa, te pedirá las entradas iniciales y luego creará la animación, como se muestra en la [Figura 6-4](ch06.html#ch6fig4).



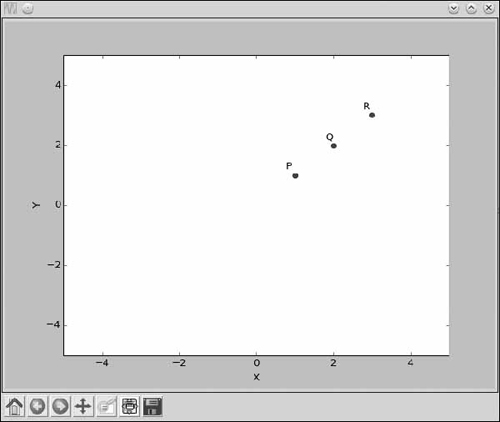
*Figura 6-4: Animación de la trayectoria de un proyectil*

### **Dibujar fractales**

Los fractales son patrones o formas geométricas complejas que surgen de fórmulas matemáticas sorprendentemente sencillas. Comparado con las formas geométricas, como los círculos y los rectángulos, un fractal parece irregular y sin ningún patrón o descripción obvia, pero si lo observas de cerca, verás que surgen patrones y que toda la forma está compuesta por numerosas copias de sí misma. Como los fractales implican la aplicación repetitiva de la misma *transformación geométrica* de puntos en un plano, los programas informáticos son muy adecuados para crearlos. En este capítulo aprenderemos a dibujar el helecho de Barnsley, el triángulo de Sierpiński y el conjunto de Mandelbrot (los dos últimos en los desafíos), ejemplos populares de fractales estudiados en este campo. Los fractales también abundan en la naturaleza: las costas, los árboles y los copos de nieve son ejemplos populares.

#### ***Transformaciones de puntos en un plano***

Una idea básica en la creación de fractales es la de la transformación de un punto. Dado un punto, *P(x*, *y*), en un plano *x-y*, un ejemplo de transformación es *P(x*, *y*) → *Q(x* + 1, *y* + 1), lo que significa que, tras aplicar la transformación , se crea un nuevo punto, *Q*, que está una unidad por encima y una unidad a la derecha de *P*. Si consideras entonces *Q* como punto de partida, obtendrás otro punto, *R*, que está una unidad por encima y una unidad a la derecha de *Q*. Considera que el punto de partida, *P*, es (1, 1). [La figura 6-5](ch06.html#ch6fig5) muestra el aspecto que tendrían los puntos.



*Figura 6-5: Los* puntos Q *y* R *se han obtenido aplicando una transformación al punto* P *durante dos iteraciones.*

Esta transformación es, por tanto, una regla que describe cómo se desplaza un punto en el plano *x-y*, partiendo de una posición inicial y desplazándose a un punto distinto en cada iteración. Podemos pensar en una transformación como la *trayectoria* del punto en el plano. Ahora, considera que en lugar de una regla de transformación, hay dos reglas de este tipo y que una de estas transformaciones se elige *al azar* en cada paso. Consideremos estas reglas:

Regla 1: *P* 1*(x*, *y*) → *P* 2*(x* + 1, *y* - 1)

Regla 2: *P* 1*(x*, *y*) → *P* 2 (*x* + 1, *y* + 1)

Considera que *P1*(1, 1) es el punto de partida. Si realizamos cuatro iteraciones, podríamos tener la siguiente secuencia de puntos

*P* 1 (1, 1) → *P* 2 (2, 0) (Regla 1)

*P* 2 (2, 0) → *P* 3 (3, 1) (Regla 2)

*P* 3 (3, 1) → *P* 4 (4, 2) (Regla 2)

*P* 4 (4, 2) → *P* 5 (5, 1) (Regla 1)

... y así sucesivamente.

La regla de transformación se elige al azar, y cada regla tiene la misma probabilidad de ser seleccionada. Se elija la que se elija, los puntos avanzarán hacia la derecha, porque aumentamos la *coordenada x* en ambos casos. A medida que los puntos avanzan hacia la derecha, se mueven hacia arriba o hacia abajo, creando así una trayectoria en zigzag. El siguiente programa traza la trayectoria de un punto al someterlo a una de estas transformaciones durante un número determinado de iteraciones:

'''  
Example of selecting a transformation from two equally probable  
transformations  
'''  
import matplotlib.pyplot as plt  
import random  
  
def transformation\_1(p):  
x = p[0]  
y = p[1]  
return x + 1, y - 1  
  
def transformation\_2(p):  
x = p[0]  
y = p[1]  
return x + 1, y + 1  
  
def transform(p):  
➊ # List of transformation functions  
transformations = [transformation\_1, transformation\_2]  
# Pick a random transformation function and call it  
➋ t = random.choice(transformations)  
➌ x, y = t(p)  
return x, y  
  
def build\_trajectory(p, n):  
x = [p[0]]  
y = [p[1]]  
for i in range(n):  
p = transform(p)  
x.append(p[0])  
y.append(p[1])  
  
return x, y  
  
if \_\_name\_\_ == '\_\_main\_\_':  
# Initial point  
p = (1, 1)  
n = int(input('Enter the number of iterations: '))  
➍ x, y = build\_trajectory(p, n)  
# Plot  
➎ plt.plot(x, y)  
plt.xlabel('X')  
plt.ylabel('Y')  
plt.show()

Definimos dos funciones, transformation\_1() y transformation\_2(), correspondientes a las dos transformaciones anteriores. En la función transform(), creamos una lista con estos dos nombres de función en ➊ y utilizamos la función random.choice() para elegir una de las transformaciones de la lista en ➋. Ahora que hemos elegido la transformación a aplicar, la llamamos con el punto, *P*, y almacenamos las coordenadas del punto transformado en las etiquetas x, y ➌ y las devolvemos.

**SELECCIONAR UN ELEMENTO ALEATORIO DE UNA LISTA**

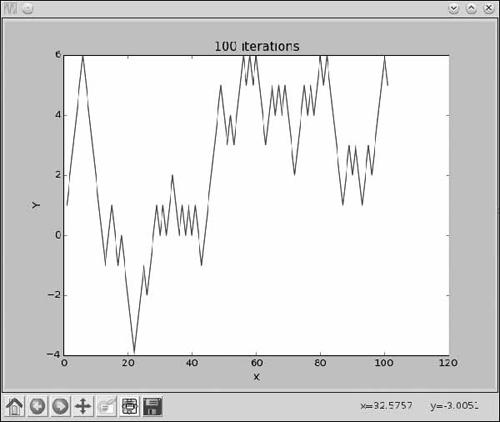
La función random.choice() que vimos en nuestro primer programa fractal se puede utilizar para seleccionar un elemento aleatorio de una lista. Cada elemento tiene *la misma* probabilidad de ser devuelto. Aquí tienes un ejemplo:

>>> import random  
>>> l = [1, 2, 3]  
>>> random.choice(l)  
3  
>>> random.choice(l)  
1  
>>> random.choice(l)  
1  
>>> random.choice(l)  
3  
>>> random.choice(l)  
3  
>>> random.choice(l)  
2

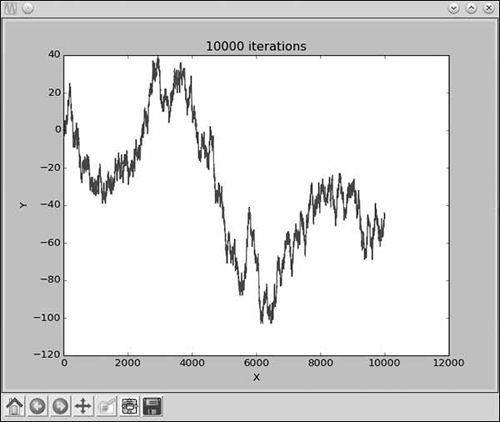
La función también funciona con tuplas y cadenas. En este último caso, devuelve un carácter aleatorio de la cadena.

Cuando ejecutas el programa, te pide el número de iteraciones, n-es decir, el número de veces que se aplicaría la transformación. A continuación, llama a la función build\_trajectory() con n y el punto inicial, *P*, que se fija en (1, 1) ➍. La función build\_trajectory() llama repetidamente a la función transform() n veces, utilizando dos listas, x y y, para almacenar la *coordenada x* y la *coordenada y* de todos los puntos transformados. Por último, devuelve las dos listas, que luego se trazan ➎.

[Las figuras 6-6](ch06.html#ch6fig6) y [6-7](ch06.html#ch6fig7) muestran la trayectoria del punto durante 100 y 10.000 iteraciones, respectivamente. El movimiento en zigzag es bastante evidente en ambas figuras. Esta trayectoria en zigzag suele denominarse *paseo aleatorio sobre una línea*.



*Figura 6-6: La trayectoria en zigzag trazada por el punto (1, 1) cuando se somete a una u otra de las dos transformaciones de forma aleatoria durante 100 iteraciones*



*Figura 6-7: La trayectoria en zigzag trazada por el punto (1, 1) cuando se somete a una u otra de las dos transformaciones aleatoriamente durante 10.000 iteraciones.*

Este ejemplo demuestra una idea básica en la creación de fractales: partir de un punto inicial y aplicar una transformación a ese punto repetidamente. A continuación, veremos un ejemplo de aplicación de las mismas ideas para dibujar el helecho de *Barnsley*.

#### ***Dibujar el helecho de Barnsley***

El matemático británico Michael Barnsley describió cómo crear estructuras parecidas a helechos utilizando aplicaciones repetidas de una transformación simple sobre un punto (ver [Figura 6](ch06.html#ch6fig8)-8).



*Figura 6-8: Helechos en forma de* [*dama2*](footnote.html#fn04)

Propuso los siguientes pasos para crear estructuras parecidas a helechos: empieza con el punto (0, 0) y selecciona *al azar* una de las siguientes transformaciones con la *probabilidad* asignada:

**Transformación 1** (probabilidad 0,85):

*xn+1* = 0,*85xn* + 0,*04yn*

*yn+1* = -0,*04yn* + 0,*85yn* + 1,6

**Transformación 2** (probabilidad 0,07):

*xn+1* = 0,*2xn* - 0,*26yn*

*yn+1* = 0,*23yn* + 0,*22yn* + 1,6

**Transformación** 3 (0,07 de probabilidad):

*xn+1* = -0,*15xn* - 0,*28xn*

*yn+1* = 0,*26yn* + 0,*24yn* + 0,44

**Transformación** 4 (probabilidad 0,01):

*xn+1* = 0

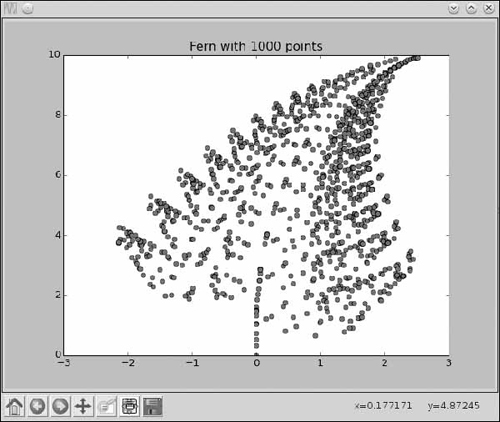
*yn+1* = 0,*16yn*

Cada una de estas transformaciones es responsable de crear una parte del helecho. La primera transformación seleccionada con la mayor probabilidad -y, por tanto, el mayor número de veces- crea el tallo y las frondas inferiores del helecho. La segunda y tercera transformaciones crean la fronda inferior de la izquierda y de la derecha, respectivamente, y la cuarta transformación crea el tallo del helecho.

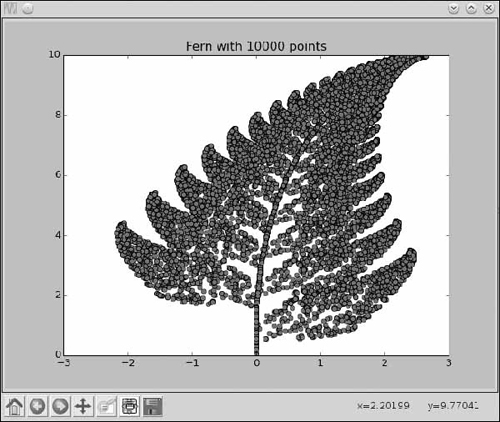
Éste es un ejemplo de selección probabilística no uniforme, sobre la que aprendimos por primera vez en [el Capítulo 5](ch05.html#ch05). El siguiente programa dibuja el helecho de Barnsley para el número de puntos especificado:

'''  
Draw a Barnsley Fern  
'''  
import random  
import matplotlib.pyplot as plt  
  
def transformation\_1(p):  
x = p[0]  
y = p[1]  
x1 = 0.85\*x + 0.04\*y  
y1 = -0.04\*x + 0.85\*y + 1.6  
return x1, y1  
  
def transformation\_2(p):  
x = p[0]  
y = p[1]  
x1 = 0.2\*x - 0.26\*y  
y1 = 0.23\*x + 0.22\*y + 1.6  
return x1, y1  
  
def transformation\_3(p):  
x = p[0]  
y = p[1]  
x1 = -0.15\*x + 0.28\*y  
y1 = 0.26\*x + 0.24\*y + 0.44  
return x1, y1  
  
def transformation\_4(p):  
x = p[0]  
y = p[1]  
x1 = 0  
y1 = 0.16\*y  
return x1, y1  
  
def get\_index(probability):  
r = random.random()  
c\_probability = 0  
sum\_probability = []  
for p in probability:  
c\_probability += p  
sum\_probability.append(c\_probability)  
for item, sp in enumerate(sum\_probability):  
if r <= sp:  
return item  
return len(probability)-1  
  
def transform(p):  
# List of transformation functions  
transformations = [transformation\_1, transformation\_2,  
transformation\_3, transformation\_4]  
➊ probability = [0.85, 0.07, 0.07, 0.01]  
# Pick a random transformation function and call it  
tindex = get\_index(probability)  
➋ t = transformations[tindex]  
x, y = t(p)  
return x, y  
  
def draw\_fern(n):  
# We start with (0, 0)  
x = [0]  
y = [0]  
  
x1, y1 = 0, 0  
for i in range(n):  
x1, y1 = transform((x1, y1))  
x.append(x1)  
y.append(y1)  
return x, y  
  
if \_\_name\_\_ == '\_\_main\_\_':  
n = int(input('Enter the number of points in the Fern: '))  
x, y = draw\_fern(n)  
# Plot the points  
plt.plot(x, y, 'o')  
plt.title('Fern with {0} points'.format(n))  
plt.show()

Cuando ejecutas este programa, te pide que especifiques el número de puntos del helecho y, a continuación, crea el helecho. Las Figuras [6-9](ch06.html#ch6fig9) y [6-10](ch06.html#ch6fig10) muestran helechos con 1.000 y 10.000 puntos, respectivamente.



*Figura 6-9: Un helecho con 1.000 puntos*



*Figura 6-10: Un helecho con 10.000 puntos*

Las cuatro reglas de transformación se definen en las funciones transformation\_1(), transformation\_2(), transformation\_3(), y transformation\_4(). La probabilidad de que se seleccione cada una se declara en una lista en ➊, y luego se selecciona una de ellas ➋ para aplicarla cada vez que la función transform() es llamada por la función draw\_fern().

El número de veces que se transforma el punto inicial (0, 0) es el mismo que el número de puntos del helecho especificados como entrada al programa.

### **Lo que has aprendido**

En este capítulo, has empezado por aprender a dibujar formas geométricas básicas y a animarlas. Este proceso te introdujo en una serie de nuevas funciones de matplotlib. Luego aprendiste sobre las transformaciones geométricas y viste cómo las transformaciones simples repetitivas te ayudan a dibujar formas geométricas complejas llamadas *fractales*.

### **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/)

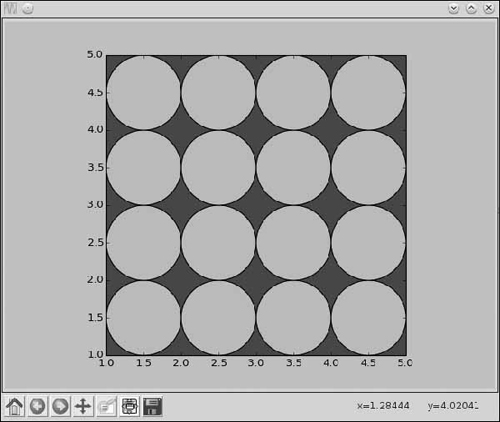
#### ***#1: Empaquetar círculos en un cuadrado***

Antes he mencionado que matplotlib admite la creación de otras formas geométricas. El parche Polygon es especialmente interesante, ya que te permite dibujar polígonos con diferentes números de lados. He aquí cómo podemos dibujar un cuadrado (cada lado de longitud 4):

'''  
Draw a square  
'''  
  
from matplotlib import pyplot as plt  
  
def draw\_square():  
ax = plt.axes(xlim = (0, 6), ylim = (0, 6))  
square = plt.Polygon([(1, 1), (5, 1), (5, 5), (1, 5)], closed = True)  
ax.add\_patch(square)  
plt.show()  
  
if \_\_name\_\_ == '\_\_main\_\_':  
draw\_square()

El objeto Polygon se crea pasando la lista de coordenadas de los vértices como primer argumento. Como estamos dibujando un cuadrado, pasamos las coordenadas de los cuatro vértices: (1, 1), (5, 1), (5, 5) y (1, 5). Al pasar closed=True le decimos a matplotlib que queremos dibujar un polígono cerrado, en el que los vértices inicial y final sean iguales.

En este reto, intentarás una versión muy simplificada del problema "círculos empaquetados en un cuadrado". ¿Cuántos círculos de radio 0,5 caben en el cuadrado producido por este código? ¡Dibuja y averígualo! La [Figura 6-11](ch06.html#ch6fig11) muestra el aspecto que tendrá la imagen final.



*Figura 6-11: Círculos empaquetados en un cuadrado*

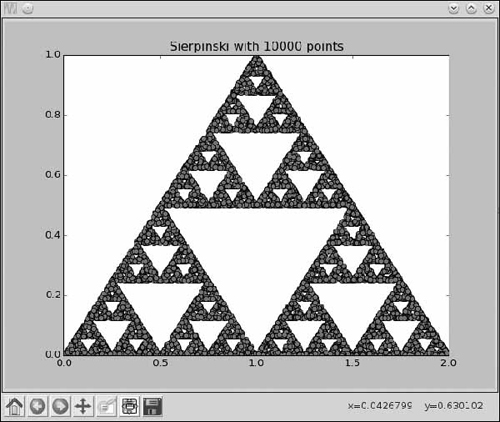
El truco está en empezar por la esquina inferior izquierda del cuadrado -es decir, (1, 1)- y seguir añadiendo círculos hasta llenar todo el cuadrado. El siguiente fragmento muestra cómo puedes crear los círculos y añadirlos a la figura:

y = 1.5  
while y < 5:  
x = 1.5  
while x < 5:  
c = draw\_circle(x, y)  
ax.add\_patch(c)  
  
x += 1.0  
y += 1.0

Cabe señalar que ésta *no* es la forma más óptima ni, en realidad, la única de introducir círculos en un cuadrado, y que los entusiastas de las matemáticas prefieren encontrar otras formas de resolver este problema.

#### ***#nº 2: Dibujo del triángulo de Sierpiński***

El triángulo de Sierpiński, llamado así por el matemático polaco Wacław Sierpiński, es un fractal que consiste en un triángulo equilátero compuesto por triángulos equiláteros más pequeños incrustados en su interior. [La Figura 6-12](ch06.html#ch6fig12) muestra un triángulo de Sierpiński compuesto por 10.000 puntos.



*Figura 6-12: Triángulo de Sierpiński con 10.000 puntos*

Lo interesante aquí es que el mismo proceso que utilizamos para dibujar un helecho también dibujará el triángulo de Sierpiński; sólo cambiarán las reglas de transformación y su probabilidad. Así es como puedes dibujar el triángulo de Sierpiński: empieza con el punto (0, 0) y aplica una de las siguientes transformaciones:

**Transformación 1:**

*xn+1* = 0,*5xn*

*yn+1* = 0,*5yn*

**Transformación** 2:

*xn+1* = 0,*5xn* + 0,5

*yn+1* = 0,*5yn* + 0,5

**Transformación** 3:

*xn+1* = 0,*5xn* + 1

*yn+1* = 0,*5yn*

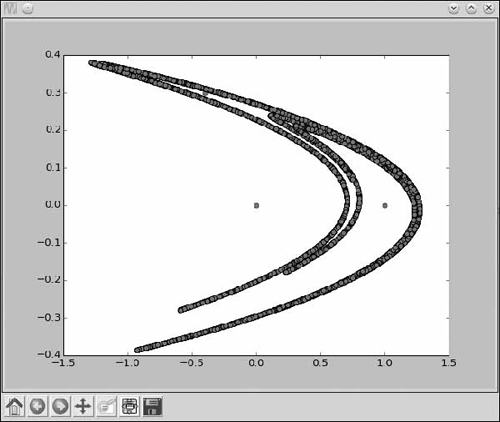
Cada una de las transformaciones tiene la misma probabilidad de ser seleccionada-1/3. Tu reto aquí es escribir un programa que dibuje el triángulo de Sierpiński compuesto por un determinado número de puntos especificados como entrada.

#### ***#3: Explorando la función de Hénon***

En 1976, Michel Hénon introdujo la función de Hénon, que describe una regla de transformación para un punto *P(x*, *y*) como sigue

*P(*x, *y*) → *Q(y* + 1 - 1,4x2, 0,*3x*)

Independientemente del punto inicial (siempre que no esté muy lejos del origen), verás que a medida que creas más puntos, éstos empiezan a situarse a lo largo de líneas curvas, como se muestra en la [Figura 6](ch06.html#ch6fig13)-13.



*Figura 6-13: Función de Hénon con 10.000 puntos*

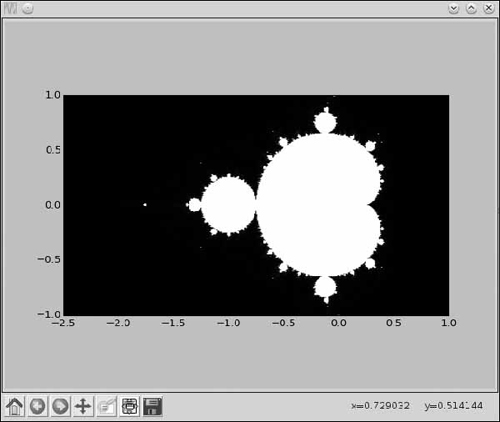
Tu reto aquí es escribir un programa para crear una gráfica que muestre 20.000 iteraciones de esta transformación, empezando por el punto (1, 1).

¡Mérito extra si escribes otro programa para crear una figura animada que muestre los puntos que empiezan a situarse a lo largo de las curvas! Consulta un ejemplo en [*https://www.youtube.com/watch?v=76ll818RlpQ*](https://www.youtube.com/watch?v=76ll818RlpQ).

Éste es un ejemplo de sistema dinámico, y las líneas curvas a las que parecen atraídos todos los puntos se denominan *atractores*. Para saber más sobre esta función, los sistemas dinámicos y los fractales en general, puedes consultar *Fractales: A Very Short Introduction*, de Kenneth Falconer (Oxford University Press, 2013).

#### ***#4: Dibujar el Conjunto de Mandelbrot***

Tu reto aquí es escribir un programa para dibujar el conjunto de Mandelbrot *, otro*ejemplo de aplicación de reglas sencillas que conducen a una forma de aspecto complicado (ver [Figura 6-14](ch06.html#ch6fig14)). Sin embargo, antes de establecer los pasos para hacerlo, primero aprenderemos sobre la función imshow() de matplotlib.

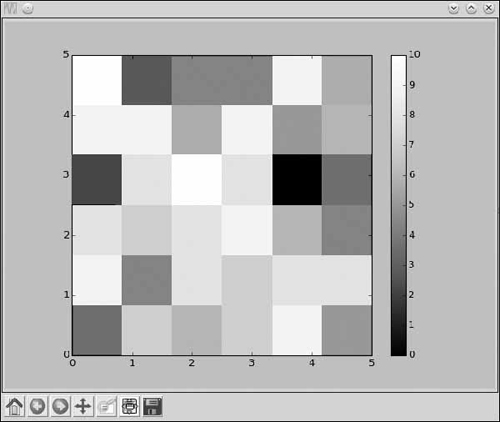


*Figura 6-14: Conjunto de Mandelbrot en el plano entre (-2,5, -1,0) y (1,0, 1,0)*

##### **La función imshow()**

La función imshow() suele utilizarse para mostrar una imagen externa, como una imagen JPEG o PNG. Puedes ver un ejemplo en [*http://matplotlib.org/users/image\_tutorial.html.*](http://matplotlib.org/users/image_tutorial.html) Aquí, sin embargo, utilizaremos la función para dibujar una nueva imagen de nuestra propia creación mediante matplotlib.

Considera la parte del plano cartesiano donde *x* e *y* van de 0 a 5. Ahora, considera seis puntos equidistantes a lo largo de cada eje: (0, 1, 2, 3, 4, 5) a lo largo del *eje x* y el mismo conjunto de puntos a lo largo del *eje* y. Si tomamos el producto cartesiano de estos puntos, obtenemos 36 puntos equidistantes en el plano *x-y* con las coordenadas (0, 0), (0, 1) ... (0, 5), (1, 0), (1, 1) ... (1, 5) ... (5, 5). Digamos ahora que queremos colorear cada uno de estos puntos de con un tono de gris, es decir, algunos de estos puntos serán negros, otros blancos y otros se colorearán con un tono intermedio, elegido al azar. La [Figura 6-15](ch06.html#ch6fig15) ilustra el escenario.



*Figura 6-15: Parte del* *plano* x-y *con* x *e* y *comprendidas entre 0 y 5. Hemos considerado 36 puntos en la región equidistantes entre sí y hemos coloreado cada uno con un tono de gris.*

Para crear esta figura, tenemos que hacer una lista de seis listas. Cada una de estas seis listas constará a su vez de seis números enteros que van de 0 a 10. Cada número corresponderá al color de cada punto, siendo 0 el negro y 10 el blanco. A continuación, pasaremos esta lista a la función imshow() junto con otros argumentos necesarios.

##### **Crear una lista de listas**

Una lista también puede contener listas como miembros:

>>> l1 = [1, 2, 3]  
>>> l2 = [4, 5, 6]  
>>> l = [l1, l2]

Aquí creamos una lista, l, formada por dos listas, l1 y l2. Así, el primer elemento de la lista, l[0], es igual que la lista l1 y el segundo elemento de la lista, l[1], es igual que la lista l2:

>>> l[0]  
[1, 2, 3]  
  
>>> l[1]  
[4, 5, 6]

Para referirnos a un elemento individual dentro de una de las listas de miembros, tenemos que especificar dos índices:l[0][1] se refiere al segundo elemento de la primera lista, l[1][2] se refiere al tercer elemento de la segunda lista, y así sucesivamente.

Ahora que sabemos cómo trabajar con una lista de listas, podemos escribir el programa para crear una figura similar a la de [la Figura 6-15](ch06.html#ch6fig15):

import matplotlib.pyplot as plt  
import matplotlib.cm as cm  
import random  
  
➊ def initialize\_image(x\_p, y\_p):  
image = []  
for i in range(y\_p):  
x\_colors = []  
for j in range(x\_p):  
x\_colors.append(0)  
image.append(x\_colors)  
return image  
  
def color\_points():  
x\_p = 6  
y\_p = 6  
image = initialize\_image(x\_p, y\_p)  
for i in range(y\_p):  
for j in range(x\_p):  
➋ image[i][j] = random.randint(0, 10)  
➌ plt.imshow(image, origin='lower', extent=(0, 5, 0, 5),  
cmap=cm.Greys\_r, interpolation='nearest')  
plt.colorbar()  
plt.show()  
  
if \_\_name\_\_ == '\_\_main\_\_':  
color\_points()

La función initialize\_image() en ➊ crea una lista de listas con cada uno de los elementos inicializados a 0. Acepta dos argumentos, x\_p y y\_p, que corresponden al número de puntos a lo largo del *eje x* y del *eje y*, respectivamente. Esto significa que la imagen de la lista inicializada estará formada por listas x\_p, cada una de las cuales contendrá y\_p ceros.

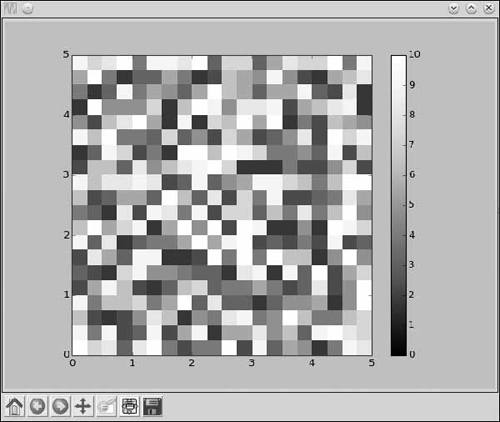
En la función color\_points(), una vez recuperada la lista imagen de initialize\_image(), asigna un entero aleatorio entre 0 y 10 al elemento image[i][j] en ➋. Cuando asignamos este número entero aleatorio al elemento, estamos asignando un color al punto del plano cartesiano que está a *i* pasos a lo largo del *eje y* y a *j* pasos a lo largo del *eje x* desde el origen. Es importante señalar que la función imshow() deduce automáticamente el color de un punto a partir de su posición en la lista image y no se preocupa de sus *coordenadas* *x* e *y* concretas.

A continuación, llama a la función imshow() en ➌, pasando image como primer argumento. El argumento de palabra clave origin='lower' especifica que el número de image[0][0] corresponde al color del punto (0, 0). El argumento de palabra clave extent=(0, 5, 0, 5) establece las esquinas inferior izquierda y superior derecha de la imagen en (0, 0) y (5, 5), respectivamente. El argumento de palabra clave cmap=cm.Greys\_r especifica que vamos a crear una imagen en escala de grises.

El último argumento de palabra clave, interpolation='nearest', especifica que matplotlib debe colorear un punto para el que no se haya especificado el color con el mismo color que el más cercano a él. ¿Qué significa esto? Observa que sólo consideramos y especificamos el color para 36 puntos de la región (0, 5) y (5, 5). Como hay un número infinito de puntos en esta región, le decimos a matplotlib que establezca el color de un punto no especificado en el de su punto más cercano. Esta es la razón por la que ves "recuadros" de color alrededor de cada punto en la figura.

Llama a la función colorbar() para mostrar una barra de color en la figura que muestre qué número entero corresponde a cada color. Por último, llama a show() para mostrar la imagen. Ten en cuenta que, debido al uso de la función random.randint(), tu imagen se coloreará de forma diferente a la de la [Figura 6-15](ch06.html#ch6fig15).

Si aumentas el número de puntos a lo largo de cada eje ajustando x\_p y y\_p a, digamos, 20 en color\_points(), verás una figura similar a la de la Figura [6-16](ch06.html#ch6fig16). Observa que los cuadros de color se hacen más pequeños. Si aumentas aún más el número de puntos, verás que el tamaño de los recuadros se reduce aún más, dando la ilusión de que cada punto tiene un color diferente.



*Figura 6-16: Parte del* *plano* x-y *con* x *e* y *comprendidas entre 0 y 5. Hemos considerado 400 puntos en la región equidistantes entre sí y hemos coloreado cada uno con un tono de gris.*

##### **Dibujar el Conjunto de Mandelbrot**

Consideraremos el área del plano *x-y* comprendida entre (-2,5, -1,0) y (1,0, 1,0) y dividiremos cada eje en 400 puntos equidistantes. El producto cartesiano de estos puntos nos dará 1.600 puntos igualmente espaciados en esta región. Nos referiremos a estos puntos como*(*x1, y1),*(*x1, y2) ...*(*x400, y400).

Crea una lista, image, llamando a la función initialize\_image() que vimos antes con x\_p y y\_p fijados en 400. A continuación, sigue estos pasos para *cada uno* de los puntos generados*(xi*, *yk*):

1. En primer lugar, crea dos números complejos, z1 = 0 + *0j* y *c* = *xi* + *yk j*. (Recuerda que utilizamos *j* para image.)

2. Crea una iteración de etiqueta y ponla a 0, es decir, iteration=0.

3. Crea un número complejo, image.

4. Incrementa el valor almacenado en iteration en 1, es decir, iteration = iteration + 1.

5. Si abs(z1) < 2 y iteration < max\_iteration, entonces vuelve al paso 3; si no, ve al paso 6. Cuanto mayor sea el valor de max\_iteration, más detallada será la imagen, pero más tiempo tardará en crearse. Ajusta aquí max\_iteration a 1.000.

6. Ajusta el color del punto*(xi*, *yk*) al valor de iteration-es decir, image[k][i] = iteration.

Una vez que tengas la lista image completa, llama a la función imshow() con el argumento de la palabra clave extent cambiado para indicar la región delimitada por (-2,5, -1,0) y (1,0, 1,0).

Este algoritmo suele denominarse *algoritmo del tiempo de escape*. Cuando se alcanza el número máximo de iteraciones antes de que la magnitud de un punto supere 2, ese punto pertenece al conjunto de Mandelbrot y se colorea de blanco. Los puntos que superan la magnitud en menos iteraciones se dice que "escapan"; no pertenecen al conjunto de Mandelbrot y se colorean de negro. Puedes experimentar disminuyendo y aumentando el número de puntos a lo largo de cada eje. Disminuyendo el número de puntos obtendrás una imagen granulada, mientras que aumentándolos obtendrás una imagen más detallada.