# 15 Generar datos



*Data visualization* es el uso de representaciones visuales para explorar y presentar patrones en conjuntos de datos. Está estrechamente relacionado con *data analysis*, que utiliza código para explorar los patrones y conexiones de un conjunto de datos. Un conjunto de datos puede ser una pequeña lista de números que cabe en una sola línea de código, o pueden ser terabytes de datos que incluyen muchos tipos diferentes de información.

Crear visualizaciones de datos eficaces es algo más que hacer que la información tenga un aspecto agradable. Cuando la representación de un conjunto de datos es sencilla y visualmente atractiva, su significado resulta claro para los espectadores. La gente verá patrones y significados en tus conjuntos de datos que no sabían que existían.

Afortunadamente, no necesitas un superordenador para visualizar datos complejos. Python es tan eficiente que, con un simple ordenador portátil, puedes explorar rápidamente conjuntos de datos que contengan millones de puntos de datos individuales. Estos puntos de datos no tienen por qué ser números; con las nociones básicas que aprendiste en la primera parte de este libro, también puedes analizar datos no numéricos.

La gente utiliza Python para trabajos intensivos en datos en genética, investigación climática, análisis político y económico, y mucho más. Los científicos de datos han escrito una impresionante serie de herramientas de visualización y análisis en Python, muchas de las cuales también están a tu disposición. Una de las herramientas más populares es Matplotlib, una biblioteca de trazado matemático. En este capítulo, utilizaremos Matplotlib para hacer trazados sencillos, como gráficos de líneas y de dispersión. Luego crearemos un conjunto de datos más interesante basado en el concepto de paseo aleatorio: una visualización generada a partir de una serie de decisiones aleatorias.

También utilizaremos un paquete llamado Plotly, que crea visualizaciones que funcionan bien en dispositivos digitales, para analizar los resultados de tirar dados. Plotly genera visualizaciones que cambian automáticamente de tamaño para adaptarse a diversos dispositivos de visualización. Estas visualizaciones también pueden incluir una serie de características interactivas, como resaltar aspectos concretos del conjunto de datos cuando los usuarios pasan el ratón por encima de distintas partes de la visualización. Aprender a utilizar Matplotlib y Plotly te ayudará a empezar a visualizar los tipos de datos que más te interesan.

## Instalar Matplotlib

Para utilizar Matplotlib en tu conjunto inicial de visualizaciones, tendrás que instalarlo mediante pip, igual que hicimos con pytest en el Capítulo 11 (ver "Instalar pytest con pip" en la página 210).

Para instalar Matplotlib, introduce el siguiente comando en una terminal:

$ python -m pip install --user matplotlib

Si utilizas un comando distinto de python para ejecutar programas o iniciar una sesión de terminal, como python3, tu comando tendrá este aspecto:

$ python3 -m pip install --user matplotlib

Para ver los tipos de visualizaciones que puedes hacer con Matplotlib, visita la página principal de Matplotlib en <https://matplotlib.org> y haz clic en **Tipos de gráficos**. Cuando hagas clic en una visualización de la galería, verás el código utilizado para generar el gráfico.

## Trazado de un gráfico lineal simple

Vamos a trazar un gráfico lineal sencillo utilizando Matplotlib y luego lo personalizaremos para crear una visualización de datos más informativa. Utilizaremos la secuencia de números cuadrados 1, 4, 9, 16 y 25 como datos para el gráfico.

Para hacer un gráfico lineal simple, especifica los números con los que quieres trabajar y deja que Matplotlib haga el resto:

**mpl\_squares.py**

import matplotlib.pyplot as plt  
  
squares = [1, 4, 9, 16, 25]  
  
❶ fig, ax = plt.subplots()  
ax.plot(squares)  
  
plt.show()

Primero importamos el módulo pyplot utilizando el alias plt para no tener que escribir pyplot repetidamente. (Verás esta convención a menudo en los ejemplos en línea, así que la utilizaremos aquí.) El módulo pyplot contiene una serie de funciones que ayudan a generar gráficos y trazados.

Creamos una lista llamada squares para guardar los datos que vamos a trazar. A continuación, seguimos otra convención habitual de Matplotlib llamando a la función subplots() ❶. Esta función puede generar uno o más gráficos en la misma figura. La variable fig representa la totalidad de *figure*, que es la colección de tramas que se generan. La variable ax representa un único gráfico en la figura; ésta es la variable que utilizaremos la mayor parte del tiempo cuando definamos y personalicemos un único gráfico.

A continuación, utilizamos el método plot(), que intenta trazar los datos que se le dan de una forma significativa. La función plt.show() abre el visor de Matplotlib y muestra el gráfico, como se muestra en [la Figura 15-1](#figure15-1). El visor te permite hacer zoom y navegar por el gráfico, y puedes guardar las imágenes del gráfico que quieras haciendo clic en el icono del disco.

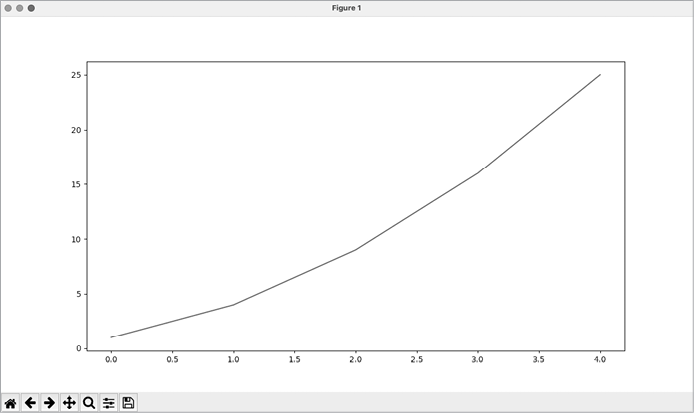


Figura 15-1: Uno de los trazados más sencillos que puedes hacer en Matplotlib

### Cambiar el tipo de etiqueta y el grosor de la línea

Aunque el gráfico de [la Figura 15-1](#figure15-1) muestra que los números están aumentando, el tipo de etiqueta es demasiado pequeño y la línea es un poco fina para leerla fácilmente. Afortunadamente, Matplotlib te permite ajustar todas las características de una visualización.

Utilizaremos algunas de las personalizaciones disponibles para mejorar la legibilidad de este gráfico. Empecemos por añadir un título y etiquetar los ejes:

**mpl\_squares.py**

import matplotlib.pyplot as plt  
  
squares = [1, 4, 9, 16, 25]  
  
fig, ax = plt.subplots()  
❶ ax.plot(squares, linewidth=3)  
  
# Set chart title and label axes.  
❷ ax.set\_title("Square Numbers", fontsize=24)  
❸ ax.set\_xlabel("Value", fontsize=14)  
ax.set\_ylabel("Square of Value", fontsize=14)  
  
# Set size of tick labels.  
❹ ax.tick\_params(labelsize=14)  
  
plt.show()

El parámetro linewidth controla el grosor de la línea que genera plot() ❶. Una vez generado el gráfico, hay muchos métodos disponibles para modificarlo antes de presentarlo. El método set\_title() establece un título general para el gráfico ❷. Los parámetros fontsize, que aparecen repetidamente a lo largo del código, controlan el tamaño del texto en varios elementos del gráfico.

Los métodos set\_xlabel() y set\_ylabel() te permiten establecer un título para cada uno de los ejes ❸, y el método tick\_params() da estilo a las marcas de graduación ❹. Aquí tick\_params() establece el tamaño de fuente de las etiquetas de las marcas de graduación en 14 en ambos ejes.

Como puedes ver en [la Figura 15-2](#figure15-2), el gráfico resultante es mucho más fácil de leer. El tipo de etiqueta es más grande, y el gráfico de líneas es más grueso. A menudo merece la pena experimentar con estos valores para ver qué funciona mejor en el gráfico resultante.

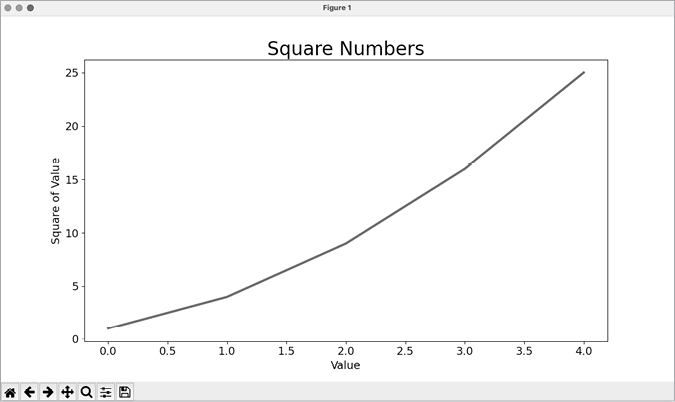


Figura 15-2: Ahora el gráfico es mucho más fácil de leer.

### Corregir el gráfico

Ahora que podemos leer mejor el gráfico, podemos ver que los datos no están trazados correctamente. Observa que al final del gráfico el cuadrado de 4,0 aparece como ¡25! Vamos a corregirlo.

Cuando das a plot() una única secuencia de números, asume que el primer punto de datos corresponde a un valor *x*-de 0, pero nuestro primer punto corresponde a un valor *x*-de 1. Podemos anular el comportamiento por defecto dando a plot() tanto los valores de entrada como los de salida utilizados para calcular los cuadrados:

**mpl\_squares.py**

import matplotlib.pyplot as plt  
  
input\_values = [1, 2, 3, 4, 5]  
squares = [1, 4, 9, 16, 25]  
  
fig, ax = plt.subplots()  
ax.plot(input\_values, squares, linewidth=3)  
  
# Set chart title and label axes.  
--snip--

Ahora plot() no tiene que hacer ninguna suposición sobre cómo se generaron los números de salida. El gráfico resultante, que se muestra en [la Figura 15-3](#figure15-3), es correcto.

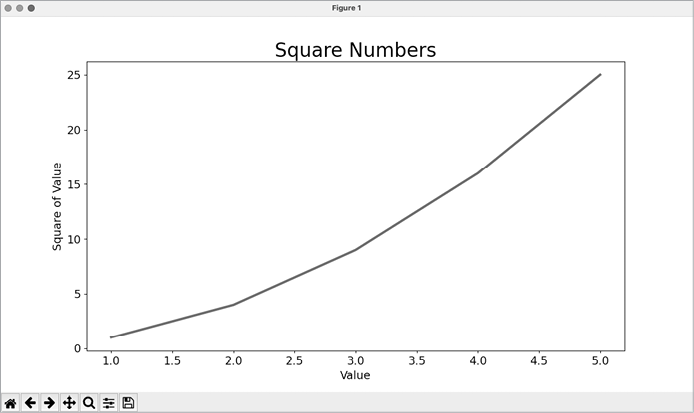


Figura 15-3: Ahora los datos se representan correctamente.

Puedes especificar una serie de argumentos al llamar a plot() y utilizar una serie de métodos para personalizar tus gráficos después de generarlos. Seguiremos explorando estos métodos de personalización a medida que trabajemos con conjuntos de datos más interesantes a lo largo de este capítulo.

### Uso de estilos incorporados

Matplotlib dispone de una serie de estilos predefinidos. Estos estilos contienen una variedad de ajustes predeterminados para colores de fondo, líneas de cuadrícula, anchos de línea, fuentes, tamaños de fuente y mucho más. Pueden hacer que tus visualizaciones sean atractivas sin requerir mucha personalización. Para ver la lista completa de estilos disponibles, ejecuta las siguientes líneas en una sesión de terminal:

>>> import matplotlib.pyplot as plt  
>>> plt.style.available  
['Solarize\_Light2', '\_classic\_test\_patch', '\_mpl-gallery',  
--snip--

Para utilizar cualquiera de estos estilos, añade una línea de código antes de llamar a subplots():

**mpl\_squares.py**

import matplotlib.pyplot as plt  
  
input\_values = [1, 2, 3, 4, 5]  
squares = [1, 4, 9, 16, 25]  
  
plt.style.use('seaborn')  
fig, ax = plt.subplots()  
--snip--

Este código genera el gráfico que se muestra en la [Figura 15-4](#figure15-4). Hay una gran variedad de estilos disponibles; juega con ellos para encontrar alguno que te guste.

|  |
| --- |
| Figura |



15-4: Estilo seaborn incorporado

### Trazar y estilizar puntos individuales con scatter()

A veces resulta útil trazar y estilizar puntos individuales en función de determinadas características. Por ejemplo, puedes trazar valores pequeños en un color y valores más grandes en otro. También puedes trazar un gran conjunto de datos con un conjunto de opciones de estilo y luego resaltar puntos individuales trazándolos de nuevo con opciones diferentes.

Para trazar un único punto, pasa los valores *x*- y *y*- del punto a scatter():

**scatter\_squares.py**

import matplotlib.pyplot as plt  
  
plt.style.use('seaborn')  
fig, ax = plt.subplots()  
ax.scatter(2, 4)  
  
plt.show()

Estilicemos la salida para hacerla más interesante. Añadiremos un título, etiquetaremos los ejes y nos aseguraremos de que todo el texto sea lo suficientemente grande como para poder leerlo:

import matplotlib.pyplot as plt  
  
plt.style.use('seaborn')  
fig, ax = plt.subplots()  
❶ ax.scatter(2, 4, s=200)  
  
# Set chart title and label axes.  
ax.set\_title("Square Numbers", fontsize=24)  
ax.set\_xlabel("Value", fontsize=14)  
ax.set\_ylabel("Square of Value", fontsize=14)  
  
# Set size of tick labels.  
ax.tick\_params(labelsize=14)  
  
plt.show()

Llamamos a scatter() y utilizamos el argumento s para establecer el tamaño de los puntos utilizados para dibujar el gráfico ❶. Cuando ejecutes ahora *scatter\_squares.py* , deberías ver un único punto en el centro del gráfico, como se muestra en la [Figura 15-5](#figure15-5).

|  |
| --- |
| Figura |



15-5:

Trazado de un único punto

### Trazar una serie de puntos con scatter()

Para trazar una serie de puntos, podemos pasar a scatter() listas separadas de valores *x*- y *y*-, de la siguiente manera:

**scatter\_squares.py**

import matplotlib.pyplot as plt  
  
x\_values = [1, 2, 3, 4, 5]  
y\_values = [1, 4, 9, 16, 25]  
  
plt.style.use('seaborn')  
fig, ax = plt.subplots()  
ax.scatter(x\_values, y\_values, s=100)  
  
# Set chart title and label axes.  
--snip--

La lista x\_values contiene los números que hay que elevar al cuadrado, y y\_values contiene el cuadrado de cada número. Cuando estas listas se pasan a scatter(), Matplotlib lee un valor de cada lista al trazar cada punto. Los puntos que se trazan son (1, 1), (2, 4), (3, 9), (4, 16) y (5, 25); [la Figura 15-6](#figure15-6) muestra el resultado.

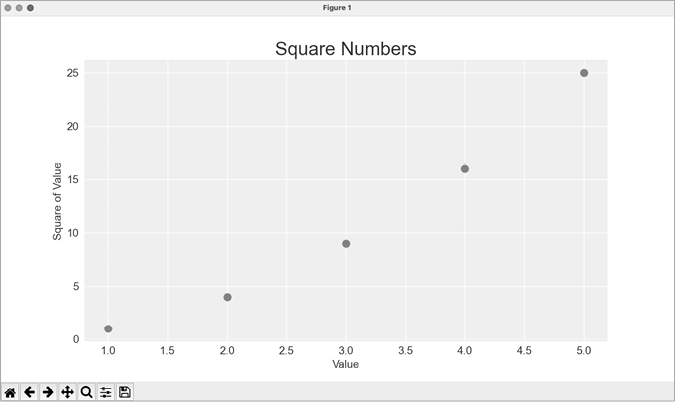


Figura 15-6: Diagrama de dispersión con múltiples puntos

### Calcular datos automáticamente

Escribir listas a mano puede resultar ineficaz, sobre todo cuando tenemos muchos puntos. En lugar de escribir cada valor, utilicemos un bucle que haga los cálculos por nosotros.

Así se vería esto con 1.000 puntos:

**scatter\_squares.py**

import matplotlib.pyplot as plt  
  
❶ x\_values = range(1, 1001)  
y\_values = [x\*\*2 for x in x\_values]  
  
plt.style.use('seaborn')  
fig, ax = plt.subplots()  
❷ ax.scatter(x\_values, y\_values, s=10)  
  
# Set chart title and label axes.  
--snip--  
  
# Set the range for each axis.  
❸ ax.axis([0, 1100, 0, 1\_100\_000])  
  
plt.show()

Empezamos con un rango de *x*-valores que contienen los números del 1 al 1.000 ❶. A continuación, una comprensión de listas genera los *y*-valores recorriendo en bucle los *x*-valores (for x in x\_values), elevando al cuadrado cada número (x\*\*2) y asignando los resultados a y\_values. Luego pasamos las listas de entrada y salida a scatter() ❷. Como se trata de un conjunto de datos grande, utilizamos un tamaño de punto más pequeño.

Antes de mostrar el gráfico, utilizamos el método axis() para especificar el rango de cada eje ❸. El método axis() requiere cuatro valores: los valores mínimo y máximo del eje *x* y del eje *y*. Aquí, ejecutamos el eje *x* de 0 a 1.100 y el eje *y* de 0 a 1.100.000. [La Figura 15-7](#figure15-7) muestra el resultado.

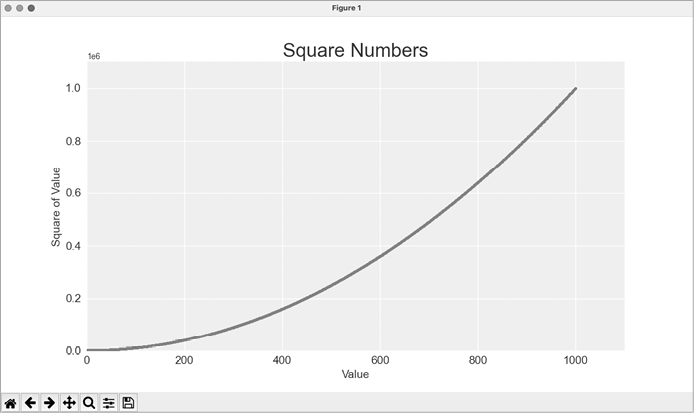


Figura 15-7: Python puede trazar 1.000 puntos tan fácilmente como traza 5 puntos.

### Personalizar las etiquetas de los puntos

Cuando los números de un eje son lo suficientemente grandes, Matplotlib utiliza por defecto la notación científica para las etiquetas de las marcas. Esto suele ser bueno, porque los números más grandes en notación simple ocupan mucho espacio innecesario en una visualización.

Casi todos los elementos de un gráfico son personalizables, así que puedes decirle a Matplotlib que siga utilizando la notación plana si lo prefieres:

--snip--  
# Set the range for each axis.  
ax.axis([0, 1100, 0, 1\_100\_000])  
ax.ticklabel\_format(style='plain')  
  
plt.show()

El método ticklabel\_format() te permite anular el estilo predeterminado de la etiqueta de marca para cualquier gráfico.

### Definir colores personalizados

Para cambiar el color de los puntos, pasa el argumento color a scatter() con el nombre de un color a utilizar entre comillas, como se muestra aquí:

ax.scatter(x\_values, y\_values, color='red', s=10)

También puedes definir colores personalizados utilizando el modelo de color RGB. Para definir un color, pasa al argumento color una tupla con tres valores flotantes (uno para el rojo, otro para el verde y otro para el azul, en ese orden), utilizando valores entre 0 y 1. Por ejemplo, la línea siguiente crea un gráfico con puntos de color verde claro:

ax.scatter(x\_values, y\_values, color=(0, 0.8, 0), s=10)

Los valores más cercanos a 0 producen colores más oscuros, y los valores más cercanos a 1 producen colores más claros.

### Utilizar un mapa de colores

Un *colormap* es una secuencia de colores en un gradiente que se mueve de un color inicial a uno final. En las visualizaciones, los mapas de colores se utilizan para resaltar patrones en los datos. Por ejemplo, puedes hacer que los valores bajos tengan un color claro y los valores altos un color más oscuro. El uso de un mapa de colores garantiza que todos los puntos de la visualización varíen suavemente y con precisión a lo largo de una escala de colores bien diseñada.

El módulo pyplot incluye un conjunto de mapas de color integrados. Para utilizar uno de estos mapas de colores, tienes que especificar cómo pyplot debe asignar un color a cada punto del conjunto de datos. A continuación se explica cómo asignar un color a cada punto, basándose en su valor *y*:

**scatter\_squares.py**

--snip--  
plt.style.use('seaborn')  
fig, ax = plt.subplots()  
ax.scatter(x\_values, y\_values, c=y\_values, cmap=plt.cm.Blues, s=10)  
  
# Set chart title and label axes.  
--snip--

El argumento c es similar a color, pero se utiliza para asociar una secuencia de valores a una asignación de colores. Pasamos la lista de *y*-valores a c, y luego le dice a pyplot qué mapa de color utilizar con el argumento cmap. Este código colorea de azul claro los puntos con valores *y* más bajos y de azul oscuro los puntos con valores *y* más altos. [La Figura 15-8](#figure15-8) muestra el gráfico resultante.

## Nota

Puedes ver todos los mapas de colores disponibles en pyplot en [https://matplotlib.org.](https://matplotlib.org) Ve a Tutoriales, desplázate hasta Colores y haz clic en **Elegir mapas de colores en Matplotlib**.

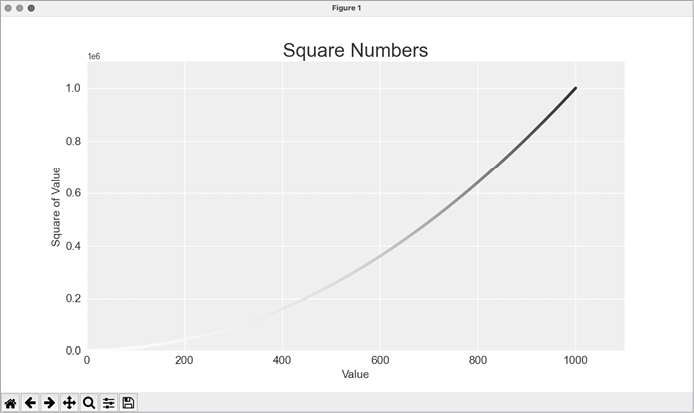


Figura 15-8: Un gráfico utilizando el mapa de colores Blues

### Guardar tus trazados automáticamente

Si quieres guardar el gráfico en un archivo en lugar de mostrarlo en el visor de Matplotlib, puedes utilizar plt.savefig() en lugar de plt.show():

plt.savefig('squares\_plot.png', bbox\_inches='tight')

El primer argumento es un nombre de archivo para la imagen del gráfico, que se guardará en el mismo directorio que *scatter\_squares.py*. El segundo argumento recorta los espacios en blanco sobrantes del gráfico. Si quieres el espacio en blanco extra alrededor del gráfico, puedes omitir este argumento. También puedes llamar a savefig() con un objeto Path, y escribir el archivo de salida en cualquier lugar que desees de tu sistema.

## Pruébalo tú mismo

15-1. Cubos: Un número elevado a la tercera potencia es un *cube*. Traza los cinco primeros números cúbicos, y luego traza los 5.000 primeros números cúbicos.

15-2. Cubos coloreados: Aplica un mapa de colores a tu gráfico de cubos.

## Recorridos aleatorios

En esta sección, utilizaremos Python para generar datos para un paseo aleatorio y luego utilizaremos Matplotlib para crear una representación visualmente atractiva de esos datos. Un *random walk* es un camino determinado por una serie de decisiones simples, cada una de las cuales se deja totalmente al azar. Podrías imaginar un paseo aleatorio como el camino que seguiría una hormiga confundida si diera cada paso en una dirección aleatoria.

Los paseos aleatorios tienen aplicaciones prácticas en la naturaleza, la física, la biología, la química y la economía. Por ejemplo, un grano de polen flotando en una gota de agua se mueve por la superficie del agua porque es empujado constantemente por las moléculas de agua. El movimiento molecular en una gota de agua es aleatorio, por lo que la trayectoria que traza un grano de polen sobre la superficie es un paseo aleatorio. El código que escribiremos a continuación modela muchas situaciones del mundo real.

### Creación de la clase RandomWalk

Para crear un paseo aleatorio, crearemos una clase RandomWalk, que tomará decisiones aleatorias sobre la dirección que debe tomar el paseo. La clase necesita tres atributos: una variable para registrar el número de puntos del paseo, y dos listas para almacenar las coordenadas *x*- y *y*- de cada punto del paseo.

Sólo necesitaremos dos métodos para la clase RandomWalk: el método \_\_init\_\_() y fill\_walk(), que calculará los puntos del paseo. Empecemos por el método \_\_init\_\_():

**random\_walk.py**

❶ from random import choice  
  
class RandomWalk:  
 """A class to generate random walks."""  
  
❷ def \_\_init\_\_(self, num\_points=5000):  
 """Initialize attributes of a walk."""  
 self.num\_points = num\_points  
  
 # All walks start at (0, 0).  
❸ self.x\_values = [0]  
 self.y\_values = [0]

Para tomar decisiones aleatorias, almacenaremos los posibles movimientos en una lista y utilizaremos la función choice() (del módulo random ) para decidir qué movimiento hacer cada vez que se dé un paso ❶. Establecemos el número predeterminado de puntos en un paseo en 5000, que es lo suficientemente grande como para generar algunos patrones interesantes, pero lo suficientemente pequeño como para generar paseos rápidamente ❷. A continuación, hacemos dos listas para guardar los valores *x*- y *y*-, y empezamos cada paseo en el punto (0, 0) ❸.

### Elegir direcciones

Utilizaremos el método fill\_walk() para determinar la secuencia completa de puntos del recorrido. Añade este método a *random\_walk.py*:

**random\_walk.py**

def fill\_walk(self):  
 """Calculate all the points in the walk."""  
  
 # Keep taking steps until the walk reaches the desired length.  
❶ while len(self.x\_values) < self.num\_points:  
  
 # Decide which direction to go, and how far to go.  
❷ x\_direction = choice([1, -1])  
 x\_distance = choice([0, 1, 2, 3, 4])  
❸ x\_step = x\_direction \* x\_distance  
  
 y\_direction = choice([1, -1])  
 y\_distance = choice([0, 1, 2, 3, 4])  
❹ y\_step = y\_direction \* y\_distance  
  
 # Reject moves that go nowhere.  
❺ if x\_step == 0 and y\_step == 0:  
 continue  
  
 # Calculate the new position.  
❻ x = self.x\_values[-1] + x\_step  
 y = self.y\_values[-1] + y\_step  
  
 self.x\_values.append(x)  
 self.y\_values.append(y)

Primero establecemos un bucle que se ejecuta hasta que el paseo se llena con el número correcto de puntos ❶. La parte principal de fill\_walk() le dice a Python cómo simular cuatro decisiones aleatorias: ¿Irá el paseo hacia la derecha o hacia la izquierda? ¿Qué distancia recorrerá en esa dirección? ¿Irá hacia arriba o hacia abajo? ¿Qué distancia recorrerá en esa dirección?

Utilizamos choice([1, -1]) para elegir un valor para x\_direction, que devuelve 1 para el movimiento hacia la derecha o -1 para el movimiento hacia la izquierda ❷. A continuación, choice([0, 1, 2, 3, 4]) selecciona aleatoriamente una distancia para moverse en esa dirección. Asignamos este valor a x\_distance. La inclusión de un 0 permite la posibilidad de pasos con movimiento a lo largo de un solo eje.

Determinamos la longitud de cada paso en las direcciones *x-* y *y-*multiplicando la dirección del movimiento por la distancia elegida ❸❹. Un resultado positivo para x\_step significa moverse hacia la derecha, un resultado negativo significa moverse hacia la izquierda, y 0 significa moverse verticalmente. Un resultado positivo para y\_step significa moverse hacia arriba, negativo significa moverse hacia abajo y 0 significa moverse horizontalmente. Si los valores tanto de x\_step como de y\_step son 0, el recorrido no va a ninguna parte; cuando esto ocurre, continuamos el bucle ❺.

Para obtener el siguiente valor de *x* para el recorrido, sumamos el valor de x\_step al último valor almacenado en x\_values ❻ y hacemos lo mismo con los valores de *y*. Cuando tengamos las coordenadas del nuevo punto, las añadimos a x\_values y y\_values.

### Trazar el recorrido aleatorio

Aquí tienes el código para trazar todos los puntos del paseo:

**rw\_visual.py**

import matplotlib.pyplot as plt  
  
from random\_walk import RandomWalk  
  
# Make a random walk.  
❶ rw = RandomWalk()  
rw.fill\_walk()  
  
# Plot the points in the walk.  
plt.style.use('classic')  
fig, ax = plt.subplots()  
❷ ax.scatter(rw.x\_values, rw.y\_values, s=15)  
❸ ax.set\_aspect('equal')  
plt.show()

Empezamos importando pyplot y RandomWalk. A continuación, creamos un paseo aleatorio y lo asignamos a rw ❶, asegurándonos de llamar a fill\_walk(). Para visualizar el paseo, introducimos los valores *x*- y *y*- del paseo en scatter() y elegimos un tamaño de punto ❷ adecuado . Por defecto, Matplotlib escala cada eje de forma independiente. Pero ese enfoque estiraría la mayoría de los recorridos horizontal o verticalmente. Aquí utilizamos el método set\_aspect() para especificar que ambos ejes deben tener el mismo espaciado entre marcas de graduación ❸.

[La Figura 15-9](#figure15-9) muestra el gráfico resultante con 5.000 puntos. Las imágenes de esta sección omiten el visor de Matplotlib, pero seguirás viéndolo cuando ejecutes *rw\_visual.py*.

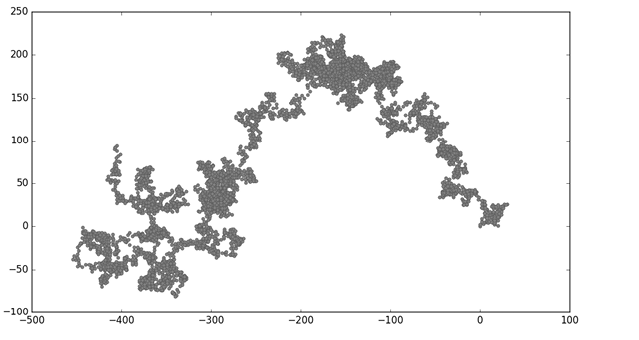


Figura 15-9: Un paseo aleatorio con 5.

000 puntos

### Generar múltiples paseos aleatorios

Cada paseo aleatorio es diferente, y es divertido explorar los distintos patrones que se pueden generar. Una forma de utilizar el código anterior para hacer múltiples paseos sin tener que ejecutar el programa varias veces es envolverlo en un bucle while, como éste:

**rw\_visual.py**

import matplotlib.pyplot as plt  
  
from random\_walk import RandomWalk  
  
# Keep making new walks, as long as the program is active.  
while True:  
 # Make a random walk.  
 --snip--  
 plt.show()  
  
 keep\_running = input("Make another walk? (y/n): ")  
 if keep\_running == 'n':  
 break

Este código genera un paseo aleatorio, lo muestra en el visor de Matplotlib y hace una pausa con el visor abierto. Cuando cierres el visor, se te preguntará si quieres generar otro paseo. Si generas unos cuantos paseos, deberías ver algunos que se quedan cerca del punto de partida, otros que se alejan sobre todo en una dirección, otros que tienen tramos delgados que conectan grupos de puntos más grandes, y muchos otros tipos de paseos. Cuando quieras terminar el programa, pulsa N.

### Estilizar el recorrido

En esta sección, personalizaremos nuestros trazados para resaltar las características importantes de cada paseo y restar importancia a los elementos que distraen. Para ello, identificamos las características que queremos resaltar, como dónde empezó el paseo, dónde terminó y el camino recorrido. A continuación, identificamos las características a las que queremos restar importancia, como las marcas de graduación y las etiquetas. El resultado debe ser una representación visual sencilla que comunique claramente el camino recorrido en cada paseo aleatorio.

#### Colorear los puntos

Utilizaremos un mapa de colores para mostrar el orden de los puntos en el paseo, y eliminaremos el contorno negro de cada punto para que el color de los puntos sea más claro. Para colorear los puntos según su posición en el recorrido, pasamos al argumento c una lista que contiene la posición de cada punto. Como los puntos se trazan en orden, esta lista sólo contiene los números de 0 a 4.999:

**rw\_visual.py**

--snip--  
while True:  
 # Make a random walk.  
 rw = RandomWalk()  
 rw.fill\_walk()  
  
 # Plot the points in the walk.  
 plt.style.use('classic')  
 fig, ax = plt.subplots()  
❶ point\_numbers = range(rw.num\_points)  
 ax.scatter(rw.x\_values, rw.y\_values, c=point\_numbers, cmap=plt.cm.Blues,  
 edgecolors='none', s=15)  
 ax.set\_aspect('equal')  
 plt.show()  
 --snip--

Utilizamos range() para generar una lista de números igual al número de puntos del recorrido ❶. Asignamos esta lista a point\_numbers, que utilizaremos para establecer el color de cada punto del paseo. Pasamos point\_numbers al argumento c, utilizamos el mapa de colores Blues, y luego pasamos edgecolors='none' para deshacernos del contorno negro alrededor de cada punto. El resultado es un gráfico que varía de azul claro a azul oscuro, mostrando exactamente cómo se mueve el paseo desde su punto inicial hasta su punto final. Esto se muestra en la [Figura 15](#figure15-10)-10.

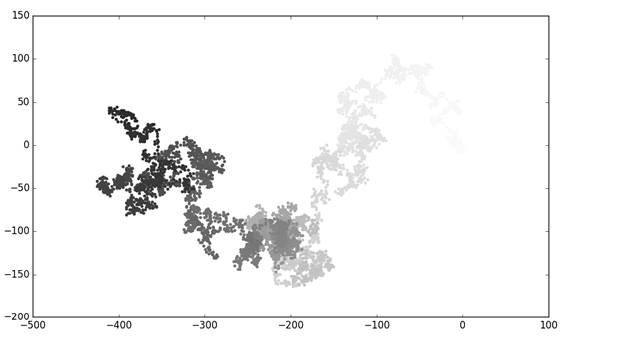


Figura15-10: Un paseo aleatorio coloreado con el mapa de colores Blues

#### Trazar los puntos inicial y final

Además de colorear los puntos para mostrar su posición a lo largo del recorrido, sería útil ver exactamente dónde empieza y acaba cada recorrido. Para ello, podemos trazar los puntos inicial y final individualmente, una vez trazada la serie principal. Haremos los puntos finales más grandes y los colorearemos de forma diferente para que destaquen:

**rw\_visual.py**

--snip--  
while True:  
 --snip--  
 ax.scatter(rw.x\_values, rw.y\_values, c=point\_numbers, cmap=plt.cm.Blues,  
 edgecolors='none', s=15)  
 ax.set\_aspect('equal')  
  
 # Emphasize the first and last points.  
 ax.scatter(0, 0, c='green', edgecolors='none', s=100)  
 ax.scatter(rw.x\_values[-1], rw.y\_values[-1], c='red', edgecolors='none',  
 s=100)  
  
 plt.show()  
 --snip--

Para mostrar el punto inicial, trazamos el punto (0, 0) en verde y en un tamaño mayor (s=100) que el resto de los puntos. Para marcar el punto final, trazamos los últimos valores *x*- y *y*- también en rojo y con un tamaño de 100. Asegúrate de insertar este código justo antes de la llamada a plt.show() para que los puntos inicial y final se dibujen encima de todos los demás puntos.

Cuando ejecutes este código, deberías ser capaz de distinguir exactamente dónde empieza y acaba cada paseo. Si estos puntos finales no destacan con suficiente claridad, ajusta su color y tamaño hasta que lo hagan.

#### Limpiar los ejes

Eliminemos los ejes de este gráfico para que no distraigan de la trayectoria de cada paseo. He aquí cómo ocultar los ejes:

**rw\_visual.py**

--snip--  
while True:  
 --snip--  
 ax.scatter(rw.x\_values[-1], rw.y\_values[-1], c='red', edgecolors='none',  
 s=100)  
  
 # Remove the axes.  
 ax.get\_xaxis().set\_visible(False)  
 ax.get\_yaxis().set\_visible(False)  
  
 plt.show()  
 --snip--

Para modificar los ejes, utilizamos los métodos ax.get\_xaxis() y ax.get\_yaxis() para obtener cada eje, y luego encadenamos el método set\_visible() para hacer invisible cada eje. A medida que sigas trabajando con visualizaciones, verás con frecuencia este encadenamiento de métodos para personalizar distintos aspectos de una visualización.

Ejecuta ahora *rw\_visual.py*; deberías ver una serie de gráficos sin ejes.

#### Añadir puntos de trazado

Aumentemos el número de puntos, para tener más datos con los que trabajar. Para ello, aumentamos el valor de num\_points cuando hagamos una instancia de RandomWalk y ajustamos el tamaño de cada punto al dibujar el gráfico:

**rw\_visual.py**

--snip--  
while True:  
 # Make a random walk.  
 rw = RandomWalk(50\_000)  
 rw.fill\_walk()  
  
 # Plot the points in the walk.  
 plt.style.use('classic')  
 fig, ax = plt.subplots()  
 point\_numbers = range(rw.num\_points)  
 ax.scatter(rw.x\_values, rw.y\_values, c=point\_numbers, cmap=plt.cm.Blues,  
 edgecolors='none', s=1)  
 --snip--

Este ejemplo crea un paseo aleatorio con 50.000 puntos y traza cada punto con el tamaño s=1. El paseo resultante es difuso y con forma de nube, como se muestra en la [Figura 15-11](#figure15-11). ¡Hemos creado una obra de arte a partir de un simple gráfico de dispersión!

Experimenta con este código para ver cuánto puedes aumentar el número de puntos de un recorrido antes de que tu sistema empiece a ralentizarse significativamente o el gráfico pierda su atractivo visual.

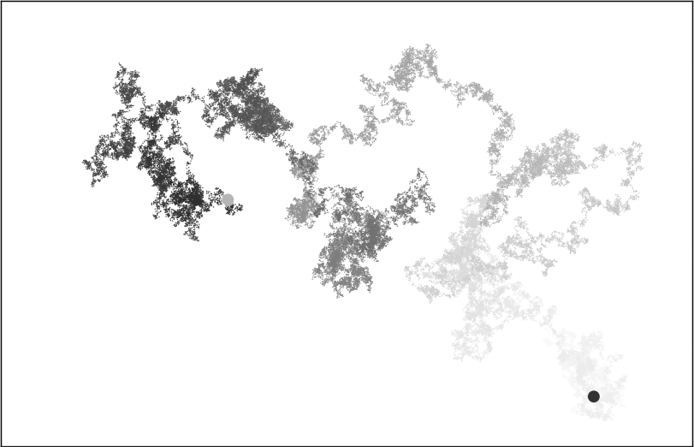


Figura 15-11: Un paseo con 50.000

puntos

#### Alterar el tamaño para llenar la pantalla

Una visualización es mucho más eficaz a la hora de comunicar patrones en los datos si se ajusta bien a la pantalla. Para que la ventana de trazado se ajuste mejor a tu pantalla, puedes ajustar el tamaño de la salida de Matplotlib. Esto se hace en la llamada subplots():

fig, ax = plt.subplots(figsize=(15, 9))

Al crear un gráfico, puedes pasar a subplots() un argumento figsize, que establece el tamaño de la figura. El parámetro figsize toma una tupla que indica a Matplotlib las dimensiones de la ventana de trazado en pulgadas.

Matplotlib asume que la resolución de tu pantalla es de 100 píxeles por pulgada; si este código no te da un tamaño de trazado exacto, ajusta los números según sea necesario. O, si conoces la resolución de tu sistema, puedes pasar a subplots() la resolución utilizando el parámetro dpi:

fig, ax = plt.subplots(figsize=(10, 6), dpi=128)

Esto debería ayudarte a hacer un uso más eficiente del espacio disponible en tu pantalla.

## Pruébalo tú mismo

15-3. Movimiento molecular: Modifica *rw\_visual.py* sustituyendo ax.scatter() por ax.plot(). Para simular la trayectoria de un grano de polen sobre la superficie de una gota de agua, pasa rw.x\_values y rw.y\_values, e incluye un argumento linewidth. Utiliza 5.000 puntos en lugar de 50.000 para que el gráfico no quede demasiado recargado.

15-4. Paseos aleatorios modificados: En la clase RandomWalk, x\_step y y\_step se generan a partir del mismo conjunto de condiciones. La dirección se elige aleatoriamente de la lista [1, -1] y la distancia de la lista [0, 1, 2, 3, 4]. Modifica los valores de estas listas para ver qué ocurre con la forma general de tus paseos. Prueba con una lista más larga de opciones para la distancia, como de 0 a 8, o elimina el -1 de la lista *x-* o *y*-dirección.

15-5. Refactorización: El método fill\_walk() es largo. Crea un nuevo método llamado get\_step() para determinar la dirección y la distancia de cada paso, y luego calcula el paso. Deberías acabar con dos llamadas a get\_step() en fill\_walk():

x\_step = self.get\_step()  
y\_step = self.get\_step()

Esta refactorización debería reducir el tamaño de fill\_walk() y facilitar la lectura y comprensión del método.

## Tirar dados con Plotly

En esta sección, utilizaremos Plotly para producir visualizaciones interactivas. Plotly es especialmente útil cuando creas visualizaciones que se mostrarán en un navegador, porque las visualizaciones se escalarán automáticamente para ajustarse a la pantalla del espectador. Estas visualizaciones también son interactivas; cuando el usuario pasa el ratón por encima de determinados elementos de la pantalla, se resalta la información sobre esos elementos. Construiremos nuestra visualización inicial en sólo un par de líneas de código utilizando *Plotly Express*, un subconjunto de Plotly que se centra en generar gráficos con el menor código posible. Una vez que sepamos que nuestro trazado es correcto, personalizaremos la salida igual que hicimos con Matplotlib.

En este proyecto, analizaremos los resultados de tirar dados. Cuando lanzas un dado normal de seis caras, tienes la misma probabilidad de sacar cualquiera de los números del 1 al 6. Sin embargo, cuando utilizas dos dados, tienes más probabilidades de sacar ciertos números que otros. Intentaremos determinar qué números tienen más probabilidades de salir generando un conjunto de datos que represente el lanzamiento de dados. Luego trazaremos los resultados de un gran número de tiradas para determinar qué resultados son más probables que otros.

Este trabajo ayuda a modelar los juegos con dados, pero las ideas centrales también se aplican a los juegos que implican azar de cualquier tipo, como los juegos de cartas. También se relaciona con muchas situaciones del mundo real en las que el azar desempeña un factor importante.

### Instalar Plotly

Instala Plotly con pip, igual que hiciste con Matplotlib:

$ python -m pip install --user plotly  
$ python -m pip install --user pandas

Plotly Express depende de *pandas*, que es una biblioteca para trabajar eficientemente con datos, así que también tenemos que instalarla. Si utilizaste python3 u otra cosa al instalar Matplotlib, asegúrate de utilizar el mismo comando aquí.

Para ver qué tipo de visualizaciones son posibles con Plotly, visita la galería de tipos de gráficos en [https://plotly.com/python.](https://plotly.com/python) Cada ejemplo incluye el código fuente, para que puedas ver cómo Plotly genera las visualizaciones.

### Creación de la clase troquel

Crearemos la siguiente clase Die para simular la tirada de un dado:

**die.py**

from random import randint  
  
class Die:  
 """A class representing a single die."""  
  
❶ def \_\_init\_\_(self, num\_sides=6):  
 """Assume a six-sided die."""  
 self.num\_sides = num\_sides  
  
 def roll(self):  
 """"Return a random value between 1 and number of sides."""  
❷ return randint(1, self.num\_sides)

El método \_\_init\_\_() toma un argumento opcional ❶. Con la clase Die, cuando se cree una instancia de nuestro dado, el número de caras será seis si no se incluye ningún argumento. Si se incluye un argumento *is*, ese valor fijará el número de caras del dado. (Los dados se nombran por su número de caras: un dado de seis caras es un D6, un dado de ocho caras es un D8, etc.)

El método roll() utiliza la función randint() para devolver un número aleatorio entre 1 y el número de caras ❷. Esta función puede devolver el valor inicial (1), el valor final (num\_sides), o cualquier número entero entre ambos.

### Tirar el dado

Antes de crear una visualización basada en la clase Die, vamos a lanzar un D6, imprimir los resultados y comprobar que los resultados parecen razonables:

**die\_visual.py**

from die import Die  
  
# Create a D6.  
❶ die = Die()  
  
# Make some rolls, and store results in a list.  
results = []  
❷ for roll\_num in range(100):  
 result = die.roll()  
 results.append(result)  
  
print(results)

Creamos una instancia de Die con los seis lados por defecto ❶. A continuación, lanzamos el dado 100 veces ❷ y almacenamos el resultado de cada tirada en la lista results. Aquí tienes un ejemplo de resultados:

[4, 6, 5, 6, 1, 5, 6, 3, 5, 3, 5, 3, 2, 2, 1, 3, 1, 5, 3, 6, 3, 6, 5, 4, 1, 1, 4, 2, 3, 6, 4, 2, 6, 4, 1, 3, 2, 5, 6, 3, 6, 2, 1, 1, 3, 4, 1, 4, 3, 5, 1, 4, 5, 5, 2, 3, 3, 1, 2, 3, 5, 6, 2, 5, 6, 1, 3, 2, 1, 1, 1, 6, 5, 5, 2, 2, 6, 4, 1, 4, 5, 1, 1, 1, 4, 5, 3, 3, 1, 3, 5, 4, 5, 6, 5, 4, 1, 5, 1, 2]

Un examen rápido de estos resultados muestra que la clase Die parece funcionar. Vemos los valores 1 y 6, por lo que sabemos que se están devolviendo los valores más pequeño y más grande posibles, y como no vemos 0 ni 7, sabemos que todos los resultados están en el rango apropiado. También vemos cada número del 1 al 6, lo que indica que están representados todos los resultados posibles. Determinemos exactamente cuántas veces aparece cada número.

### Analizar los resultados

Analizaremos los resultados de tirar un D6 contando cuántas veces sale cada número:

**die\_visual.py**

--snip--  
# Make some rolls, and store results in a list.  
results = []  
❶ for roll\_num in range(1000):  
 result = die.roll()  
 results.append(result)  
  
# Analyze the results.  
frequencies = []  
❷ poss\_results = range(1, die.num\_sides+1)  
for value in poss\_results:  
❸ frequency = results.count(value)  
❹ frequencies.append(frequency)  
  
print(frequencies)

Como ya no vamos a imprimir los resultados, podemos aumentar el número de tiradas simuladas a 1000 ❶. Para analizar las tiradas, creamos la lista vacía frequencies para almacenar el número de veces que sale cada valor. A continuación, generamos todos los resultados posibles que podríamos obtener; en este ejemplo, son todos los números desde 1 hasta el número de caras que tenga die ❷. Recorremos en bucle los valores posibles, contamos cuántas veces aparece cada número en results ❸, y luego añade este valor a frequencies ❹. Imprimimos esta lista antes de hacer una visualización:

[155, 167, 168, 170, 159, 181]

Los resultados parecen razonables: vemos seis frecuencias, una para cada número posible al tirar un D6. También vemos que ninguna frecuencia es significativamente mayor que otra. Ahora vamos a visualizar estos resultados.

### Hacer un histograma

Ahora que tenemos los datos que queremos, podemos generar una visualización en sólo un par de líneas de código utilizando Plotly Express:

**die\_visual.py**

import plotly.express as px  
  
from die import Die  
--snip--  
  
for value in poss\_results:  
 frequency = results.count(value)  
 frequencies.append(frequency)  
  
# Visualize the results.  
fig = px.bar(x=poss\_results, y=frequencies)  
fig.show()

Primero importamos el módulo plotly.express, utilizando el alias convencional px. A continuación, utilizamos la función px.bar() para crear un gráfico de barras. En el uso más sencillo de esta función, sólo necesitamos pasar un conjunto de valores *x-*y un conjunto de valores *y-*. Aquí, los valores *x* son los posibles resultados de lanzar un solo dado, y los valores *y* son las frecuencias de cada resultado posible.

La última línea llama a fig.show(), que indica a Plotly que muestre el gráfico resultante como un archivo HTML y lo abra en una nueva pestaña del navegador. El resultado se muestra en la [Figura 15-12](#figure15-12).

Este es un gráfico muy simple, y desde luego no está completo. Pero así es exactamente como debe utilizarse Plotly Express: escribes un par de líneas de código, miras el gráfico y te aseguras de que representa los datos como tú quieres. Si te gusta lo que ves, puedes pasar a personalizar elementos del gráfico como etiquetas y estilos. Pero si quieres explorar otros tipos de gráficos posibles, puedes hacerlo ahora, sin haber dedicado tiempo extra a las tareas de personalización. Puedes probarlo ahora cambiando px.bar() por algo como px.scatter() o px.line(). Encontrarás una lista completa de los tipos de gráficos disponibles en [https://plotly.com/python/plotly-express.](https://plotly.com/python/plotly-express)

Este gráfico es dinámico e interactivo. Si cambias el tamaño de la ventana de tu navegador, el gráfico cambiará de tamaño para adaptarse al espacio disponible. Si pasas el ratón por encima de cualquiera de las barras, verás una ventana emergente que resalta los datos específicos relacionados con esa barra.

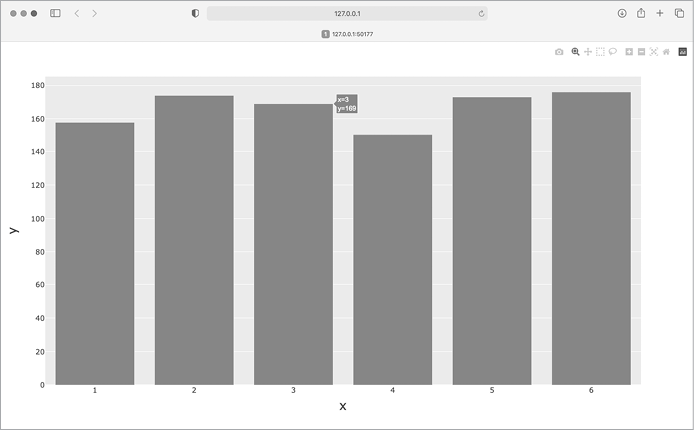


Figura 15-12: Gráfico inicial producido por Plotly Express

### Personalizar el gráfico

Ahora que sabemos que tenemos el tipo correcto de gráfico y que nuestros datos se representan con precisión, podemos centrarnos en añadir las etiquetas y estilos adecuados para el gráfico.

La primera forma de personalizar un gráfico con Plotly es utilizar algunos parámetros opcionales en la llamada inicial que genera el gráfico, en este caso, px.bar(). He aquí cómo añadir un título general y una etiqueta para cada eje:

**die\_visual.py**

--snip--  
# Visualize the results.  
❶ title = "Results of Rolling One D6 1,000 Times"  
❷ labels = {'x': 'Result', 'y': 'Frequency of Result'}  
fig = px.bar(x=poss\_results, y=frequencies, title=title, labels=labels)  
fig.show()

Primero definimos el título que queremos, aquí asignado a title ❶. Para definir las etiquetas de los ejes, escribimos un diccionario ❷. Las claves del diccionario se refieren a las etiquetas que queremos personalizar, y los valores son las etiquetas personalizadas que queremos utilizar. Aquí damos al eje *x* la etiqueta Result y al eje *y* la etiqueta Frequency of Result. La llamada a px.bar() incluye ahora los argumentos opcionales title y labels.

Ahora, cuando se genera el gráfico, incluye un título apropiado y una etiqueta para cada eje, como se muestra en la [Figura 15](#figure15-13)-13.

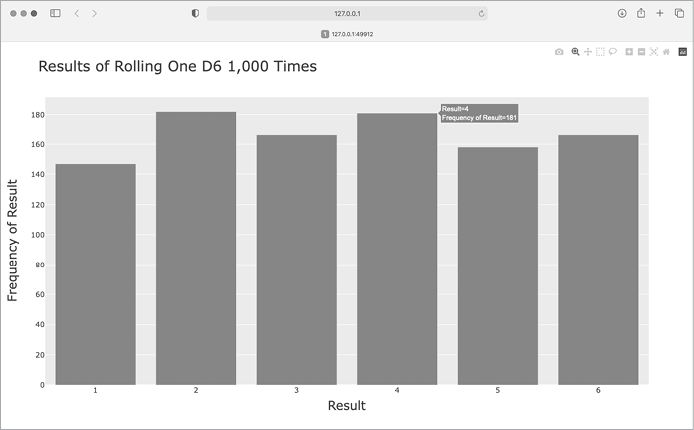


Figura 15-13: Gráfico de barras sencillo creado con Plotly

### Tirar dos dados

Al lanzar dos dados se obtienen números más grandes y una distribución diferente de los resultados. Modifiquemos nuestro código para crear dos dados D6 que simulen la forma en que lanzamos un par de dados. Cada vez que lancemos el par, sumaremos los dos números (uno de cada dado) y almacenaremos la suma en results. Guarda una copia de *die\_visual.py* como *dice\_visual.py* y realiza los siguientes cambios:

**dice\_visual.py**

import plotly.express as px  
  
from die import Die  
  
# Create two D6 dice.  
die\_1 = Die()  
die\_2 = Die()  
  
# Make some rolls, and store results in a list.  
results = []  
for roll\_num in range(1000):  
❶ result = die\_1.roll() + die\_2.roll()  
 results.append(result)  
  
# Analyze the results.  
frequencies = []  
❷ max\_result = die\_1.num\_sides + die\_2.num\_sides  
❸ poss\_results = range(2, max\_result+1)  
for value in poss\_results:  
 frequency = results.count(value)  
 frequencies.append(frequency)  
  
# Visualize the results.  
title = "Results of Rolling Two D6 Dice 1,000 Times"  
labels = {'x': 'Result', 'y': 'Frequency of Result'}  
fig = px.bar(x=poss\_results, y=frequencies, title=title, labels=labels)  
fig.show()

Tras crear dos instancias de Die, tiramos los dados y calculamos la suma de los dos dados para cada tirada ❶. El menor resultado posible (2) es la suma del menor número de cada dado. El mayor resultado posible (12) es la suma del mayor número de cada dado, que asignamos a max\_result ❷. La variable max\_result facilita mucho la lectura del código para generar poss\_results ❸. Podríamos haber escrito range(2, 13), pero esto sólo funcionaría para dos dados D6. Al modelar situaciones del mundo real, es mejor escribir un código que pueda modelar fácilmente diversas situaciones. Este código nos permite simular el lanzamiento de un par de dados con cualquier número de caras.

Después de ejecutar este código, deberías ver un gráfico parecido al de [la Figura 15-14](#figure15-14).

|  |
| --- |
| Figura |



15-14: Resultados simulados de lanzar dos dados de seis caras 1.000 veces

Este gráfico muestra la distribución aproximada de resultados que es probable que obtengas al lanzar un par de dados D6. Como puedes ver, lo menos probable es que saques un 2 o un 12 y lo más probable es que saques un 7. Esto ocurre porque hay seis formas de sacar un 7: 1 y 6, 2 y 5, 3 y 4, 4 y 3, 5 y 2, y 6 y 1.

### Otras personalizaciones

Hay una cuestión que debemos abordar con la trama que acabamos de generar. Ahora que hay 11 barras, la configuración por defecto del eje *x* deja algunas barras sin etiquetar. Aunque la configuración por defecto funciona bien para la mayoría de las visualizaciones, este gráfico tendría mejor aspecto con todas las barras etiquetadas.

Plotly tiene un método update\_layout() que puede utilizarse para realizar una amplia variedad de actualizaciones en una figura después de haberla creado. He aquí cómo decirle a Plotly que dé a cada barra su propia etiqueta:

**dice\_visual.py**

--snip--  
fig = px.bar(x=poss\_results, y=frequencies, title=title, labels=labels)  
  
# Further customize chart.  
fig.update\_layout(xaxis\_dtick=1)  
  
fig.show()

El método update\_layout() actúa sobre el objeto fig, que representa el gráfico general. Aquí utilizamos el argumento xaxis\_dtick, que especifica la distancia entre las marcas de graduación en el eje *x*. Fijamos ese espaciado en 1, para que cada barra esté etiquetada. Cuando vuelvas a ejecutar *dice\_visual.py*, deberías ver una etiqueta en cada barra.

### Lanzar dados de distintos tamaños

Vamos a crear un dado de seis caras y otro de diez caras, y a ver qué ocurre cuando los lanzamos 50.000 veces:

**dice\_visual\_d6d10.py**

import plotly.express as px  
  
from die import Die  
  
# Create a D6 and a D10.  
die\_1 = Die()  
❶ die\_2 = Die(10)  
  
# Make some rolls, and store results in a list.  
results = []  
for roll\_num in range(50\_000):  
 result = die\_1.roll() + die\_2.roll()  
 results.append(result)  
  
# Analyze the results.  
--snip--  
  
# Visualize the results.  
❷ title = "Results of Rolling a D6 and a D10 50,000 Times"  
labels = {'x': 'Result', 'y': 'Frequency of Result'}  
--snip--

Para crear un D10, pasamos el argumento 10 al crear la segunda instancia de Die ❶ y cambiamos el primer bucle para simular 50.000 tiradas en lugar de 1.000. También cambiamos el título del gráfico ❷.

[La Figura 15-15](#figure15-15) muestra el gráfico resultante. En lugar de un resultado más probable, hay cinco resultados de este tipo. Esto ocurre porque sigue habiendo una sola forma de sacar el valor más pequeño (1 y 1) y el valor más grande (6 y 10), pero el dado más pequeño limita el número de formas en que puedes generar los números intermedios. Hay seis formas de sacar un 7, 8, 9, 10 u 11, que son los resultados más comunes, y tienes las mismas probabilidades de sacar cualquiera de ellos.

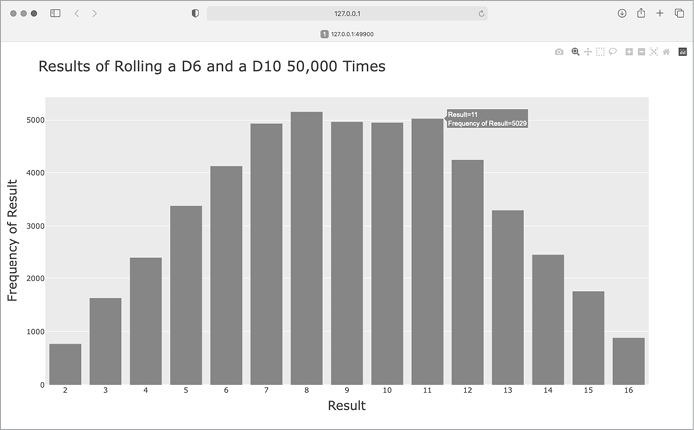


Figura 15-15: Los resultados de lanzar un dado de seis caras y un dado de diez caras 50.000 veces

La posibilidad de utilizar Plotly para modelizar el lanzamiento de dados nos da una libertad considerable para explorar este fenómeno. En sólo unos minutos, puedes simular un enorme número de tiradas utilizando una gran variedad de dados.

### Guardar cifras

Cuando tengas una figura que te guste, siempre puedes guardar el gráfico como un archivo HTML a través de tu navegador. Pero también puedes hacerlo mediante programación. Para guardar tu gráfico como un archivo HTML, sustituye la llamada a fig.show() por una llamada a fig.write\_html():

fig.write\_html('dice\_visual\_d6d10.html')

El método write\_html() requiere un argumento: el nombre del archivo en el que escribir. Si sólo proporcionas un nombre de archivo, el archivo se guardará en el mismo directorio que el archivo *.py*. También puedes llamar a write\_html() con un objeto Path, y escribir el archivo de salida en cualquier lugar que desees de tu sistema.

## Pruébalo tú mismo

15-6. Dos D8: Crea una simulación que muestre lo que ocurre cuando tiras dos dados de ocho caras 1.000 veces. Intenta imaginarte cómo crees que será la visualización antes de ejecutar la simulación, y luego comprueba si tu intuición era correcta. Aumenta gradualmente el número de tiradas hasta que empieces a ver los límites de las capacidades de tu sistema.

15-7. Tres dados: Cuando lanzas tres dados D6, el número más pequeño que puedes sacar es 3 y el más grande 18. Crea una visualización que muestre lo que ocurre cuando tiras tres dados D6.

15-8. Multiplicación: Cuando tiras dos dados, normalmente sumas los dos números para obtener el resultado. Crea una visualización que muestre qué ocurre si, en lugar de eso, multiplicas esos números entre sí.

15-9. Comprensión de dados: Para mayor claridad, los listados de esta sección utilizan la forma larga de los bucles for. Si te sientes cómodo utilizando comprensiones de listas, intenta escribir una comprensión para uno o ambos bucles de cada uno de estos programas.

15-10. Practicando con ambas bibliotecas: Intenta utilizar Matplotlib para hacer una visualización de una tirada de dados, y utiliza Plotly para hacer la visualización de un paseo aleatorio. (Tendrás que consultar la documentación de cada biblioteca para completar este ejercicio).

## Resumen

En este capítulo, has aprendido a generar conjuntos de datos y a crear visualizaciones de esos datos. Creaste gráficos sencillos con Matplotlib y utilizaste un gráfico de dispersión para explorar los paseos aleatorios. También creaste un histograma con Plotly, y lo utilizaste para explorar los resultados de lanzar dados de diferentes tamaños.

Generar tus propios conjuntos de datos con código es una forma interesante y potente de modelar y explorar una amplia variedad de situaciones del mundo real. A medida que sigas trabajando en los proyectos de visualización de datos que siguen, estate atento a las situaciones que puedas modelar con código. Observa las visualizaciones que ves en los medios de comunicación, y mira si puedes identificar las que se generaron utilizando métodos similares a los que estás aprendiendo en estos proyectos.

En el Capítulo 16, descargarás datos de fuentes online y seguirás utilizando Matplotlib y Plotly para explorar esos datos.