Capítulo 2: Visualizar datos con gráficos

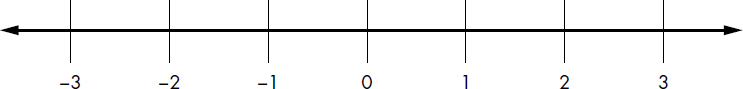
## **2** Visualizar datos con gráficos



En este capítulo aprenderás una potente forma de presentar datos numéricos: dibujando gráficas con Python. Empezaremos hablando de la recta numérica y el plano cartesiano. A continuación, conoceremos la potente biblioteca de trazado *matplotlib* y cómo podemos utilizarla para crear gráficos. Exploraremos cómo hacer gráficos que presenten los datos de forma clara e intuitiva. Por último, utilizaremos gráficos para explorar la ley de la gravitación universal de Newton y el movimiento de los proyectiles. ¡Empecemos!

### **Comprender el plano de coordenadas cartesianas**

Considera una *recta numérica*, como la que se muestra en [la Figura 2-1](ch02.html#ch2fig1). Los números enteros de -3 a 3 están marcados en la recta, pero entre cualquiera de estos dos números (digamos, 1 y 2) se encuentran todos los números posibles intermedios: 1,1, 1,2, 1,3, etc.

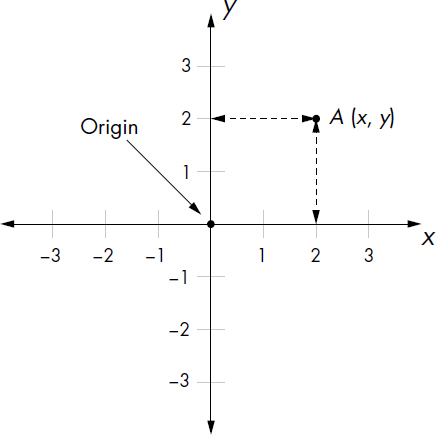


*Figura 2-1: Una recta numérica*

La recta numérica hace que ciertas propiedades sean visualmente intuitivas. Por ejemplo, todos los números situados a la derecha del 0 son positivos, y los situados a la izquierda son negativos. Cuando un número *a* se encuentra a la derecha de otro número *b*, *a* siempre es mayor que *b* y *b* siempre es menor que *a*.

Las flechas de los extremos de la recta numérica indican que la recta se extiende infinitamente, y cualquier punto de esta recta corresponde a algún número real, por grande que sea. Un solo número es suficiente para describir un punto de la recta numérica.

Considera ahora dos rectas numéricas dispuestas como se muestra en la [Figura 2-2](ch02.html#ch2fig2). Las rectas numéricas se cortan en ángulo recto y se cruzan en el punto 0 de cada recta. Esto forma un *plano de coordenadas cartesianas*, o plano *x-y*, en el que la recta numérica horizontal se llama *eje x* y la vertical *eje y*.



*Figura 2-2: El plano de coordenadas cartesianas*

Al igual que con la recta numérica, podemos tener infinitos puntos en el plano. Describimos un punto con un par de números en lugar de con un solo número. Por ejemplo, describimos el punto *A* de la figura con dos números, *x* e *y*, normalmente escritos como*(x*, *y*) y denominados *coordenadas* del punto. Como se muestra en [la Figura 2-2](ch02.html#ch2fig2), *x* es la distancia del punto desde el origen a lo largo del eje *x*, e *y* es la distancia a lo largo del *eje y*. El punto de intersección de los dos ejes se denomina *origen* y tiene las coordenadas (0, 0).

El plano de coordenadas cartesianas nos permite visualizar la relación entre dos conjuntos de números. Aquí utilizo el término *conjunto* en sentido amplio para referirme a una colección de números. (En el [Capítulo 5](ch05.html#ch05) aprenderemos sobre los conjuntos matemáticos y cómo trabajar con ellos en Python). No importa lo que representen los dos conjuntos de números -temperatura, resultados de béisbol o notas de los exámenes de clase-, todo lo que necesitas son los números en sí. Luego, puedes trazarlos, ya sea en papel cuadriculado o en tu ordenador con un programa escrito en Python. En el resto de este libro, utilizaré el término *trazar* como verbo para describir el acto de trazar dos conjuntos de números y el término *gráfico* para describir el resultado: una línea, una curva o simplemente un conjunto de puntos en el plano cartesiano.

### **Trabajar con listas y tuplas**

Al hacer gráficos con Python, trabajaremos con *listas* y *tuplas*. En Python, se trata de dos formas distintas de almacenar grupos de valores. Las tuplas y las listas son muy similares en su mayor parte, con una diferencia importante: después de crear una lista, es posible añadirle valores y cambiar el orden de los valores. En cambio, los valores de una tupla se fijan inmediatamente y no pueden modificarse. Utilizaremos listas para almacenar *las coordenadas* *x* e *y* de los puntos que queremos trazar. Las tuplas aparecerán en "[Personalizar gráficos](ch02.html#ch02lev2sec05)", en la [página 41](ch02.html#page_41), cuando aprendamos a personalizar el rango de nuestros gráficos. En primer lugar, repasemos algunas características de las listas.

Puedes crear una lista introduciendo valores, separados por comas, entre corchetes. La siguiente sentencia crea una lista y utiliza la etiqueta simplelist para referirse a ella:

>>> simplelist = [1, 2, 3]

Ahora puedes referirte a los números individuales-1, 2 y 3-utilizando la etiqueta y la posición del número en la lista, que se denomina *índice*. Así, simplelist[0] se refiere al primer número, simplelist[1] al segundo y simplelist[2] al tercero:

>>> simplelist[0]  
1  
>>> simplelist[1]  
2  
>>> simplelist[2]  
3

Observa que el primer elemento de la lista está en el índice 0, el segundo en el 1, y así sucesivamente, es decir, las posiciones de la lista empiezan a contar desde 0, no desde 1.

Las listas también pueden almacenar cadenas:

>>> stringlist = ['a string','b string','c string']  
>>> stringlist[0]  
'a string'  
>>> stringlist[1]  
'b string'  
>>> stringlist[2]  
'c string'

Una ventaja de crear una lista es que no tienes que crear una etiqueta distinta para cada valor; basta con crear una etiqueta para la lista y utilizar la posición del índice para referirte a cada elemento. Además, puedes añadir a la lista siempre que necesites almacenar nuevos valores, por lo que una lista es la mejor opción para almacenar datos si no sabes de antemano cuántos números o cadenas puedes necesitar almacenar.

Una lista *vacía* es precisamente eso -una lista sin elementos- y puede crearse así:

>>> emptylist = []

Las listas vacías son útiles sobre todo cuando no conoces de antemano ninguno de los elementos que habrá en tu lista, pero piensas rellenar los valores durante la ejecución de un programa. En ese caso, puedes crear una lista vacía y luego utilizar el método append() para añadir elementos más adelante:

➊ >>> emptylist  
[]  
➋ >>> emptylist.append(1)  
>>> emptylist  
[1]  
➌ >>> emptylist.append(2)  
>>> emptylist  
➍ [1, 2]

En ➊, emptylist comienza vacía. A continuación, añadimos el número 1 a la lista en ➋ y luego añadimos el 2 en ➌. En la línea ➍, la lista es ahora [1, 2]. Ten en cuenta que cuando utilizas .append(), el valor se añade al final de la lista. Ésta es sólo una forma de añadir valores a una lista. Hay otras, pero no las necesitaremos en este capítulo.

Crear una tupla es similar a crear una lista, pero en lugar de corchetes, utilizas paréntesis:

>>> simpletuple = (1, 2, 3)

Puedes referirte a un número individual en simpletuple utilizando el índice correspondiente entre paréntesis, igual que con las listas:

>>> simpletuple[0]  
1  
  
>>> simpletuple[1]  
2  
>>> simpletuple[2]  
3

También puedes utilizar *índices negativos* tanto con listas como con tuplas. Por ejemplo, simplelist[-1] y simpletuple[-1] se referirían al último elemento de la lista o de la tupla, simplelist[-2] y simpletuple[-2] se referirían al penúltimo elemento, y así sucesivamente.

Las tuplas, como las listas, pueden tener cadenas como valores, y puedes crear una *tupla vacía* sin elementos como emptytuple=(). Sin embargo, no existe ningún método append() para añadir un nuevo valor a una tupla existente, por lo que no puedes añadir valores a una tupla vacía. Una vez creada una tupla, su contenido no puede modificarse.

#### ***Iterar sobre una lista o tupla***

Podemos recorrer una lista o tupla utilizando un bucle for como se indica a continuación:

>>> l = [1, 2, 3]  
>>> for item in l:  
print(item)

Esto imprimirá los elementos de la lista:

1  
2  
3

Los elementos de una tupla pueden recuperarse del mismo modo.

A veces puedes necesitar conocer la posición o el índice de un elemento de una lista o tupla. Puedes utilizar la función enumerate() para iterar sobre todos los elementos de una lista y devolver el índice de un elemento, así como el propio elemento. Utilizamos las etiquetas index y item para referirnos a ellas:

>>> l = [1, 2, 3]  
>>> for index, item in enumerate(l):  
print(index, item)

Esto producirá el siguiente resultado:

0 1  
1 2  
2 3

Esto también funciona para tuplas.

### **Crear gráficos con Matplotlib**

Utilizaremos matplotlib para crear gráficos con Python. Matplotlib es un *paquete* de Python, lo que significa que es una colección de módulos con funcionalidades relacionadas. En este caso, los módulos sirven para trazar números y hacer gráficos. Matplotlib no viene incorporado con la biblioteca estándar de Python, así que tendrás que instalarlo. Las instrucciones de instalación se encuentran en [el Apéndice A](app01.html#app01). Una vez que lo tengas instalado, inicia un intérprete de comandos de Python. Como se explica en las instrucciones de instalación, puedes seguir utilizando el shell IDLE o utilizar el shell incorporado de Python.

Ahora estamos preparados para crear nuestro primer gráfico. Empezaremos con un gráfico sencillo con sólo tres puntos: (1, 2), (2, 4) y (3, 6). Para crear esta gráfica, primero haremos dos listas de números: una que almacene los valores de las coordenadas *x* de estos puntos y otra que almacene las *coordenadas y*. Las dos sentencias siguientes hacen exactamente eso, creando las dos listas x\_numbers y y\_numbers:

>>> x\_numbers = [1, 2, 3]  
>>> y\_numbers = [2, 4, 6]

A partir de aquí, podemos crear el gráfico:

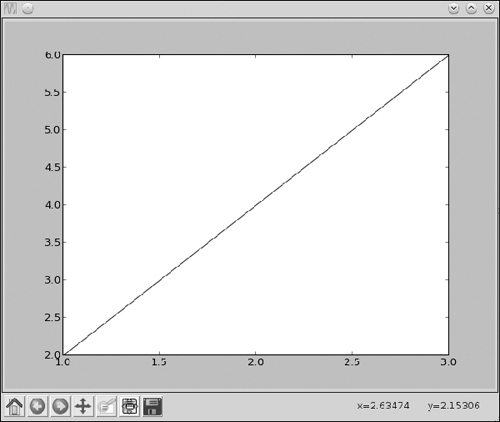
>>> from pylab import plot, show  
>>> plot(x\_numbers, y\_numbers)  
[<matplotlib.lines.Line2D object at 0x7f83ac60df10>]

En la primera línea, importamos las funciones plot() y show() del módulo pylab, que forma parte del paquete matplotlib. A continuación, llamamos a la función plot() en la segunda línea. El primer argumento de la función plot() es la lista de números que queremos representar en el eje *x*, y el segundo argumento es la lista correspondiente de números que queremos representar en el *eje y*. La función plot() devuelve un objeto o, más exactamente, una lista que contiene un objeto. Este objeto contiene la información sobre el gráfico que le hemos pedido a Python que cree. En este punto, puedes añadir más información, como un título, a la gráfica, o puedes simplemente mostrar la gráfica tal y como es. Por ahora sólo mostraremos el gráfico.

La función plot() sólo crea el gráfico. Para visualizarla realmente, tenemos que llamar a la función show():

>>> show()

Deberías ver la gráfica en una ventana de matplotlib, como se muestra en [la Figura 2-3](ch02.html#ch2fig3). (La ventana de visualización puede tener un aspecto diferente según tu sistema operativo, pero la gráfica debería ser la misma).



*Figura 2-3: Gráfica de una recta que pasa por los puntos (1, 2), (2, 4) y (3, 6)*

Observa que, en lugar de partir del origen (0, 0), el *eje x* parte del número 1 y el *eje y* parte del número 2. Estos son los números más bajos de cada uno de los ejes. Estos son los números más bajos de cada una de las dos listas. Además, puedes ver incrementos marcados en cada uno de los ejes (como 2,5, 3,0, 3,5, etc., en el *eje y*). En "[Personalizar gráficos](ch02.html#ch02lev2sec05)", en [la página 41](ch02.html#page_41), aprenderemos a controlar esos aspectos del gráfico, junto con la forma de añadir etiquetas a los ejes y un título para el gráfico.

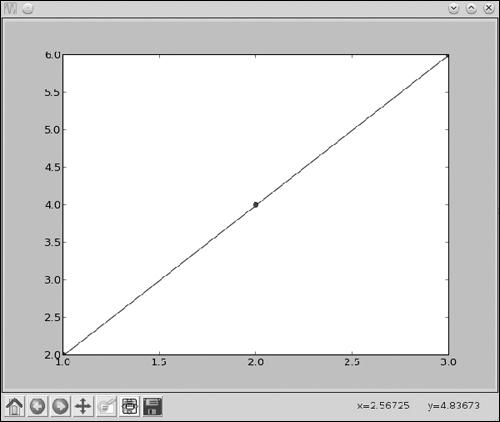
Observarás en el intérprete de comandos interactivo que no puedes introducir más sentencias hasta que cierres la ventana de matplotlib. Cierra la ventana del gráfico para poder seguir programando.

#### ***Marcar puntos en tu gráfico***

Si quieres que el gráfico marque los puntos que has suministrado para su trazado, puedes utilizar un argumento de palabra clave adicional al llamar a la función plot():

>>> plot(x\_numbers, y\_numbers, marker='o')

Introduciendo marker='o', le decimos a Python que marque cada punto de nuestras listas con un pequeño punto parecido a una *o*. Cuando vuelvas a introducir show(), verás que cada punto se marca con un punto (ver [Figura 2-4](ch02.html#ch2fig4)).

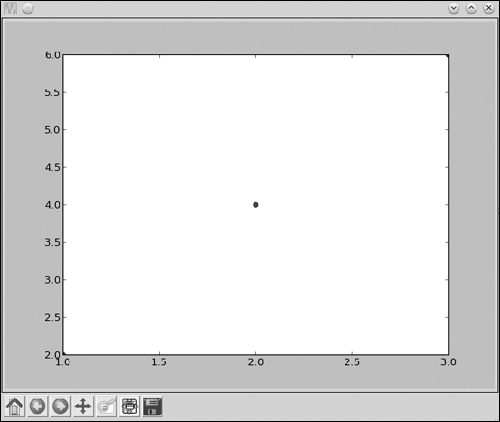


*Figura 2-4: Gráfico que muestra una recta que pasa por los puntos (1, 2), (2, 4) y (3, 6) con los puntos marcados con un punto*

El marcador en (2, 4) es fácilmente visible, mientras que los demás están ocultos en las esquinas del gráfico. Puedes elegir entre varias opciones de marker, incluyendo 'o', '\*', 'x' y '+'. El uso de marker= incluye una línea que conecta los puntos (es la opción por defecto). También puedes hacer un gráfico que marque sólo los puntos que has especificado, sin ninguna línea que los conecte, omitiendo marker=:

>>> plot(x\_numbers, y\_numbers, 'o')  
[<matplotlib.lines.Line2D object at 0x7f2549bc0bd0>]

Aquí, 'o' indica que cada punto debe marcarse con un punto, pero no debe haber ninguna línea que conecte los puntos. Llama a la función show() para mostrar el gráfico, que debería parecerse al que se muestra en la [Figura 2-5](ch02.html#ch2fig5).



*Figura 2-5: Gráfica que muestra los puntos (1, 2), (2, 4) y (3, 6)*

Como puedes ver, ahora sólo se muestran los puntos en la gráfica, sin ninguna línea que los conecte. Como en el gráfico anterior, el primer y el último punto apenas son visibles, pero pronto veremos cómo cambiar eso.

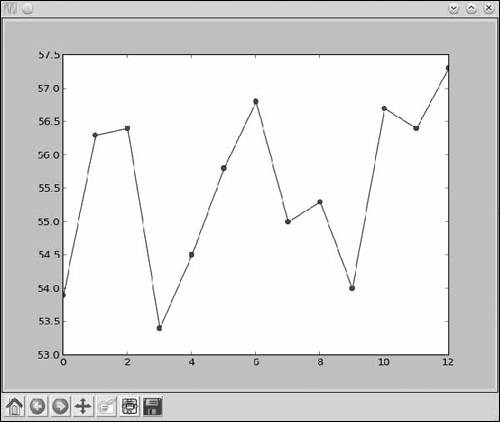
#### ***Gráfico de la temperatura media anual en Nueva York***

Echemos un vistazo a un conjunto de datos ligeramente mayor para poder explorar más funciones de matplotlib. Las temperaturas medias anuales de la ciudad de Nueva York -medidas en Central Park, concretamente- durante los años 2000 a 2012 son las siguientes: 53,9, 56,3, 56,4, 53,4, 54,5, 55,8, 56,8, 55,0, 55,3, 54,0, 56,7, 56,4 y 57,3 grados Fahrenheit. Ahora mismo, esto parece un revoltijo aleatorio de números, pero podemos trazar este conjunto de temperaturas en un gráfico para que la subida y bajada de la temperatura media de un año a otro sea mucho más clara:

>>> nyc\_temp = [53.9, 56.3, 56.4, 53.4, 54.5, 55.8, 56.8, 55.0, 55.3, 54.0, 56.7, 56.4, 57.3]  
>>> plot(nyc\_temp, marker='o')  
[<matplotlib.lines.Line2D object at 0x7f2549d52f90>]

Almacenamos las temperaturas medias en una lista, nyc\_temp. A continuación, llamamos a la función plot() pasándole sólo esta lista (y la cadena del marcador). Cuando utilizas plot() en una sola lista, esos números se trazan automáticamente en el *eje y*. Los valores correspondientes en el *eje x* se rellenan como las posiciones de cada valor de la lista. Es decir, al primer valor de temperatura, 53,9, le corresponde un valor 0 *en el eje x*, porque está en la posición 0 de la lista (recuerda que la posición de la lista empieza a contar desde 0, no desde 1). Como resultado, los números trazados en el eje *x* son los enteros de 0 a 12, que podemos considerar como correspondientes a los 13 años de los que tenemos datos de temperatura.

Introduce show() para visualizar el gráfico, que se muestra en la [Figura 2-6](ch02.html#ch2fig6). El gráfico muestra que la temperatura media ha subido y bajado de un año a otro. Si echas un vistazo a los números que hemos trazado, en realidad no están muy alejados entre sí. Sin embargo, el gráfico hace que las variaciones parezcan bastante dramáticas. Entonces, ¿qué está pasando? La razón es que matplotlib elige el rango del *eje y* de modo que sea justo el suficiente para encerrar los datos suministrados para el trazado. Así, en este gráfico, el *eje* y empieza en 53,0 y su valor más alto es 57,5. Esto hace que incluso las pequeñas diferencias parezcan aumentadas, porque el rango del eje *y* es muy pequeño. Aprenderemos a controlar el rango de cada eje en "[Personalizar gráficos](ch02.html#ch02lev2sec05)" en [la página 41](ch02.html#page_41).



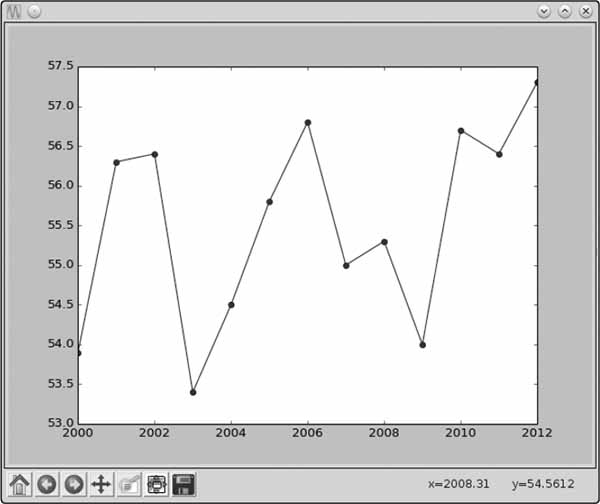
*Figura 2-6: Gráfico que muestra la temperatura media anual de la ciudad de Nueva York durante los años 2000-2012*

También puedes ver que los números del *eje y* son números de coma flotante (porque es lo que pedimos que se representara) y los del *eje x* son números enteros. Matplotlib puede manejar cualquiera de los dos.

Trazar la temperatura sin mostrar los años correspondientes es una forma rápida y sencilla de visualizar las variaciones entre los años. Sin embargo, si tuvieras pensado presentar este gráfico a alguien, te gustaría hacerlo más claro mostrando a qué año corresponde cada temperatura. Podemos hacerlo fácilmente creando otra lista con los años y llamando a la función plot():

>>> nyc\_temp = [53.9, 56.3, 56.4, 53.4, 54.5, 55.8, 56.8, 55.0, 55.3, 54.0, 56.7, 56.4, 57.3]  
>>> years = range(2000, 2013)  
>>> plot(years, nyc\_temp, marker='o')  
[<matplotlib.lines.Line2D object at 0x7f2549a616d0>]  
>>> show()

Utilizamos la función range() que conocimos en el [Capítulo](ch01.html#ch01) 1 para especificar los años 2000 a 2012. Ahora verás los años mostrados en el *eje x*(ver [Figura 2-7](ch02.html#ch2fig7)).



*Figura 2-7: Gráfico de la temperatura media anual de la ciudad de Nueva York, mostrando los* años en el *eje* x

#### ***Comparación de las tendencias mensuales de la temperatura en Nueva York***

Sin dejar de observar la ciudad de Nueva York, veamos cómo ha variado la temperatura media mensual a lo largo de los años. Esto nos dará la oportunidad de entender cómo trazar varias líneas en un mismo gráfico. Elegiremos tres años: 2000, 2006 y 2012. Para cada uno de estos años, trazaremos la temperatura media de los 12 meses.

En primer lugar, tenemos que crear tres listas para almacenar la temperatura (en grados Fahrenheit). Cada lista constará de 12 números correspondientes a la temperatura media de enero a diciembre de cada año:

>>> nyc\_temp\_2000 = [31.3, 37.3, 47.2, 51.0, 63.5, 71.3, 72.3, 72.7, 66.0, 57.0, 45.3, 31.1]  
>>> nyc\_temp\_2006 = [40.9, 35.7, 43.1, 55.7, 63.1, 71.0, 77.9, 75.8, 66.6, 56.2, 51.9, 43.6]  
>>> nyc\_temp\_2012 = [37.3, 40.9, 50.9, 54.8, 65.1, 71.0, 78.8, 76.7, 68.8, 58.0, 43.9, 41.5]

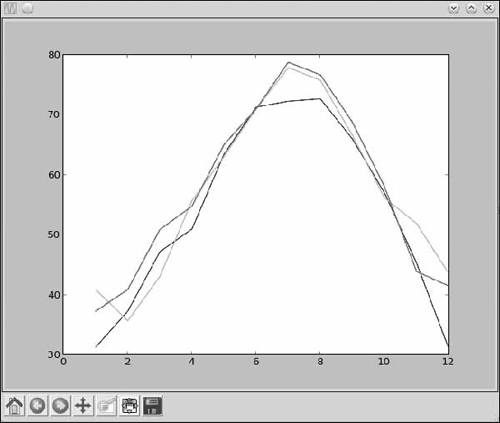
La primera lista corresponde al año 2000, y las dos siguientes a los años 2006 y 2012, respectivamente. Podríamos trazar los tres conjuntos de datos en tres gráficos diferentes, pero eso no facilitaría mucho ver cómo se compara cada año con los demás. ¡Prueba a hacerlo!

La forma más clara de comparar todas estas temperaturas es trazar los tres conjuntos de datos en un *solo* gráfico, como éste:

>>> months = range(1, 13)  
>>> plot(months, nyc\_temp\_2000, months, nyc\_temp\_2006, months, nyc\_temp\_2012)  
[<matplotlib.lines.Line2D object at 0x7f2549c1f0d0>, <matplotlib.lines.Line2D  
object at 0x7f2549a61150>, <matplotlib.lines.Line2D object at 0x7f2549c1b550>]

En primer lugar, creamos una lista (months) donde almacenamos los números 1, 2, 3, y así sucesivamente hasta el 12 utilizando la función range(). A continuación, llamamos a la función plot() con tres pares de listas. Cada par consiste en una lista de meses que se trazarán en el eje *x* y una lista de temperaturas medias mensuales (de 2000, 2006 y 2012, respectivamente) que se trazarán en el *eje y*. Hasta ahora, hemos utilizado plot() en un solo par de listas a la vez, pero en realidad puedes introducir varios pares de listas en la función plot(). Con cada lista separada por una coma, la función plot() trazará automáticamente una línea diferente para cada par.

La función plot() devuelve una lista de tres objetos en lugar de uno. Matplotlib considera que las tres curvas son distintas entre sí, y sabe que debe dibujarlas una encima de la otra cuando llamas a show(). Llamemos a show() para mostrar el gráfico, como se muestra en la [Figura 2-8](ch02.html#ch2fig8).



*Figura 2-8: Gráfico que muestra la temperatura media mensual de la ciudad de Nueva York durante los años 2000, 2006 y 2012*

Ahora tenemos tres gráficos en uno solo. Python elige automáticamente un color diferente para cada línea, para indicar que las líneas se han trazado a partir de conjuntos de datos diferentes.

En lugar de llamar a la función trazar con los tres pares a la vez, también podríamos llamar a la función trazar tres veces distintas, una para cada par:

>>> plot(months, nyc\_temp\_2000)  
[<matplotlib.lines.Line2D object at 0x7f1e51351810>]  
>>> plot(months, nyc\_temp\_2006)  
[<matplotlib.lines.Line2D object at 0x7f1e5ae8e390>]  
>>> plot(months, nyc\_temp\_2012)  
[<matplotlib.lines.Line2D object at 0x7f1e5136ccd0>]  
>>> show()

Matplotlib lleva la cuenta de los trazados que aún no se han mostrado. Así que, si esperamos a llamar a show() hasta después de llamar tres veces a plot(), todos los gráficos se mostrarán en el mismo gráfico.

Sin embargo, tenemos un problema, porque no tenemos ninguna pista sobre qué color corresponde a cada año. Para solucionarlo, podemos utilizar la función legend(), que nos permite añadir una leyenda al gráfico. Una *leyenda* es un pequeño recuadro que identifica lo que significan las distintas partes del gráfico. Aquí utilizaremos una leyenda para indicar a qué año corresponde cada línea de color. Para añadir la leyenda, llama primero a la función plot() como antes:

>>> plot(months, nyc\_temp\_2000, months, nyc\_temp\_2006, months, nyc\_temp\_2012)  
[<matplotlib.lines.Line2D object at 0x7f2549d6c410>, <matplotlib.lines.Line2D  
object at 0x7f2549d6c9d0>, <matplotlib.lines.Line2D object at 0x7f2549a86850>]

A continuación, importa la función legend() del módulo pylab y llámala como se indica a continuación:

>>> from pylab import legend  
>>> legend([2000, 2006, 2012])  
<matplotlib.legend.Legend object at 0x7f2549d79410>

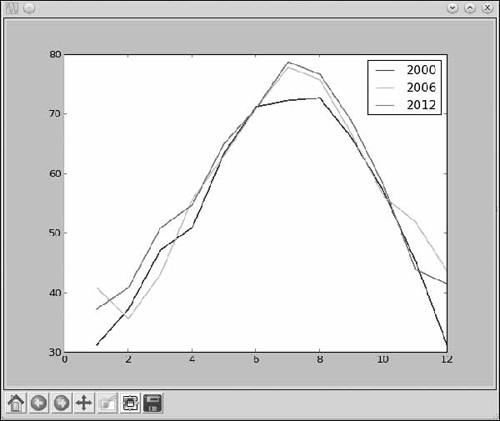
Llamamos a la función legend() con una lista de las etiquetas que queremos utilizar para identificar cada parcela del gráfico. Estas etiquetas se introducen en este orden para que coincidan con el orden de los pares de listas que se introdujeron en la función plot(). Es decir, 2000 será la etiqueta para el gráfico del primer par que introdujimos en la función plot(); 2006, para el segundo par; y 2012, para el tercero. También puedes especificar un segundo argumento a la función que especificará la posición de la leyenda. Por defecto, siempre se coloca en la parte superior derecha del gráfico. Sin embargo, puedes especificar una posición concreta, como 'lower center', 'center left', y 'upper left'. O puedes establecer la posición 'best', y la leyenda se colocará de forma que no interfiera con el gráfico.

Por último, llamamos a show() para mostrar el gráfico:

>>> show()

Como puedes ver en el gráfico (ver [Figura 2-9](ch02.html#ch2fig9)), ahora hay un cuadro de leyenda en la esquina superior derecha. Nos indica qué línea representa la temperatura media mensual del año 2000, qué línea representa el año 2006 y qué línea representa el año 2012.

Observando el gráfico, puedes concluir dos hechos interesantes: la temperatura más alta de los tres años se produjo en julio y alrededor de julio (correspondiente a 7 en el *eje x*), y ha ido aumentando desde el año 2000, con una subida más espectacular entre 2000 y 2006. Tener las tres líneas trazadas juntas en un gráfico hace que sea mucho más fácil ver este tipo de relaciones. Desde luego, es más claro que mirar unas largas listas de números o incluso que mirar tres líneas trazadas en tres gráficos distintos.



*Figura 2-9: Gráfico que muestra la temperatura media mensual de la ciudad de Nueva York, con una leyenda que muestra el año al que corresponde cada color*

#### ***Personalizar gráficos***

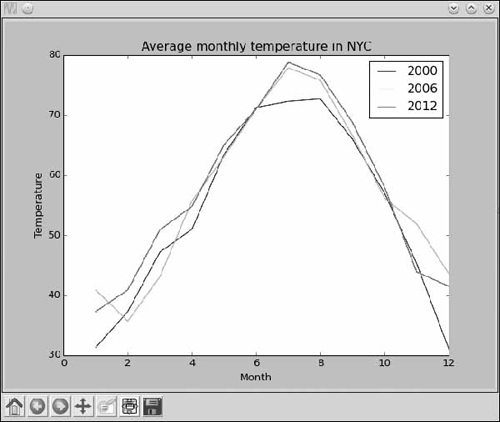
Ya hemos visto una forma de personalizar un gráfico: añadiendo una leyenda. Ahora aprenderemos otras formas de personalizar un gráfico y hacerlo más claro, añadiendo etiquetas a los ejes *x* e *y*, añadiendo un título al gráfico y controlando el rango y los pasos de los ejes.

##### **Añadir un título y etiquetas**

Podemos añadir un título a nuestro gráfico utilizando la función title() y añadir etiquetas para los *ejes* *x* e y utilizando las funciones xlabel() y ylabel(). Volvamos a crear el último gráfico y añadamos toda esta información adicional:

>>> from pylab import plot, show, title, xlabel, ylabel, legend  
>>> plot(months, nyc\_temp\_2000, months, nyc\_temp\_2006, months, nyc\_temp\_2012)  
[<matplotlib.lines.Line2D object at 0x7f2549a9e210>, <matplotlib.lines.Line2D  
object at 0x7f2549a4be90>, <matplotlib.lines.Line2D object at 0x7f2549a82090>]  
>>> title('Average monthly temperature in NYC')  
<matplotlib.text.Text object at 0x7f25499f7150>  
>>> xlabel('Month')  
<matplotlib.text.Text object at 0x7f2549d79210>  
>>> ylabel('Temperature')  
<matplotlib.text.Text object at 0x7f2549b8b2d0>  
  
>>> legend([2000, 2006, 2012])  
<matplotlib.legend.Legend object at 0x7f2549a82910>

Las tres funciones -title(), xlabel(), y ylabel()- se llaman con el texto correspondiente que queremos que aparezca en el gráfico introducido como cadenas. Al llamar a la función show() se mostrará el gráfico con toda esta información recién añadida (ver [Figura 2-10](ch02.html#ch2fig10)).



*Figura 2-10: Se han añadido al gráfico las etiquetas de los ejes y un título.*

Con los tres nuevos datos añadidos, el gráfico es más fácil de entender.

##### **Personalizar los Ejes**

Hasta ahora, hemos permitido que Python determine automáticamente los números de ambos ejes basándose en los datos suministrados a la función plot(). Esto puede estar bien para la mayoría de los casos, pero a veces este rango automático no es la forma más clara de presentar los datos, como vimos en el gráfico en el que trazamos la temperatura media anual de la ciudad de Nueva York (ver [Figura 2-7](ch02.html#ch2fig7)). Allí, incluso pequeños cambios en la temperatura parecían grandes porque el rango del eje *y* elegido automáticamente era muy estrecho. Podemos ajustar el rango de los ejes utilizando la función axis(). Esta función puede utilizarse tanto para recuperar el rango actual como para establecer un nuevo rango para los ejes.

Considera, una vez más, la temperatura media anual de la ciudad de Nueva York durante los años 2000 a 2012 y crea un gráfico como hicimos anteriormente.

>>> nyc\_temp = [53.9, 56.3, 56.4, 53.4, 54.5, 55.8, 56.8, 55.0, 55.3, 54.0, 56.7, 56.4, 57.3]  
>>> plot(nyc\_temp, marker='o')  
[<matplotlib.lines.Line2D object at 0x7f3ae5b767d0>]

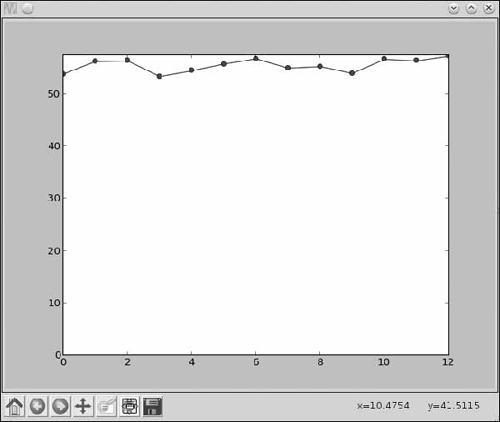
Ahora, importa la función axis() y llámala:

>>> from pylab import axis  
>>> axis()  
(0.0, 12.0, 53.0, 57.5)

La función devuelve una tupla con cuatro números correspondientes al rango para el *eje x*(0,0, 12,0) y el *eje y*(53,0, 57,5). Estos son los mismos valores de rango del gráfico que hicimos antes. Ahora, cambiemos el *eje* y para que empiece en 0 en lugar de en 53,0:

>>> axis(ymin=0)  
(0.0, 12.0, 0, 57.5)

Al llamar a la función axis() con el nuevo valor inicial del eje *y*(especificado por ymin=0) cambia el rango, y la tupla devuelta lo confirma. Si visualizas el gráfico llamando a la función show(), el *eje* y empieza en 0, y las diferencias entre los valores de los años consecutivos parecen menos drásticas (ver [Figura 2-11](ch02.html#ch2fig11)).



*Figura 2-11: Gráfico que muestra la temperatura media anual de la ciudad de Nueva York durante los años 2000-2012. El* eje y se ha*personalizado para que empiece en 0.*

Del mismo modo, puedes utilizar xmin, xmax, y ymax para establecer los valores mínimo y máximo del eje *x* y el valor máximo del *eje y*, respectivamente. Si vas a cambiar los cuatro valores, puede que te resulte más fácil llamar a la función axis() con los cuatro valores del rango introducidos como una lista, como axis([0, 10, 0, 20]). Esto establecería el rango del eje *x* en (0, 10) y el del *eje* y en (0, 20).

##### **Trazar con pyplot**

El módulo pylab es útil para crear gráficos en un intérprete de comandos interactivo, como el intérprete de comandos IDLE, como hemos hecho hasta ahora. Sin embargo, cuando se utiliza matplotlib fuera del shell IDLE -por ejemplo, como parte de un programa más amplio-, el módulo pyplot es más eficiente. No te preocupes: todos los métodos que conociste al utilizar pylab funcionarán igual con pyplot.

El siguiente programa recrea el primer gráfico de este capítulo utilizando el módulo pyplot:

'''  
Simple plot using pyplot  
'''  
  
➊ import matplotlib.pyplot  
  
➋ def create\_graph():  
x\_numbers = [1, 2, 3]  
y\_numbers = [2, 4, 6]  
  
matplotlib.pyplot.plot(x\_numbers, y\_numbers)  
matplotlib.pyplot.show()  
  
if \_\_name\_\_ == '\_\_main\_\_':  
create\_graph()

En primer lugar, importamos el módulo pyplot utilizando la sentencia import matplotlib .pyplot ➊. Esto significa que estamos importando todo el módulo pyplot del paquete matplotlib. Para hacer referencia a cualquier definición de función o clase definida en este módulo, tendrás que utilizar la sintaxis matplotlib.pyplot.itemdonde item es la función o clase que quieres utilizar.

Esto es diferente de importar una única función o clase cada vez, que es lo que hemos estado haciendo hasta ahora. Por ejemplo, en el primer capítulo importamos la clase Fraction como from fractions import Fraction. Importar un módulo entero es útil cuando vas a utilizar varias funciones de ese módulo. En lugar de importarlas individualmente, puedes importar todo el módulo de una vez y hacer referencia a las distintas funciones cuando las necesites.

En la función create\_graph() de ➋, creamos las dos listas de números que queremos trazar en el gráfico y luego pasamos las dos listas a la función plot(), del mismo modo que hicimos antes con pylab. Esta vez, sin embargo, llamamos a la función como matplotlib.pyplot.plot(), lo que significa que estamos llamando a la función plot() definida en el módulo pyplot del paquete matplotlib. A continuación, llamamos a la función show() para visualizar el gráfico. La única diferencia entre la forma de trazar los números aquí comparada con lo que hicimos antes es el mecanismo de llamada a las funciones.

Para ahorrarnos algo de tecleo, podemos importar el módulo pyplot introduciendo import matplotlib.pyplot as plt. Entonces, podemos referirnos a pyplot con la etiqueta plt en nuestros programas, en lugar de tener que escribir siempre matplotlib.pyplot:

'''  
Simple plot using pyplot  
'''  
import matplotlib.pyplot as plt  
  
def create\_graph():  
x\_numbers = [1, 2, 3]  
y\_numbers = [2, 4, 6]  
plt.plot(x\_numbers, y\_numbers)  
plt.show()  
  
if \_\_name\_\_ == '\_\_main\_\_':  
create\_graph()

Ahora podemos llamar a las funciones anteponiéndoles el prefijo plt en lugar de matplotlib.pyplot.

A partir de ahora, para el resto de este capítulo y de este libro, utilizaremos pylab en el intérprete de comandos interactivo y pyplot en los demás casos.

#### ***Guardar los gráficos***

Si necesitas guardar tus gráficos, puedes hacerlo utilizando la función savefig(). Esta función guarda el gráfico como un archivo de imagen, que puedes utilizar en informes o presentaciones. Puedes elegir entre varios formatos de imagen, como PNG, PDF y SVG.

Aquí tienes un ejemplo:

>>> from pylab import plot, savefig  
>>> x = [1, 2, 3]  
>>> y = [2, 4, 6]  
>>> plot(x, y)  
>>> savefig('mygraph.png')

Este programa guardará el gráfico en un archivo de imagen, *mygraph.png*, en tu directorio actual. En Microsoft Windows, suele ser *C:\Python33* (donde instalaste Python). En Linux, el directorio actual suele ser tu directorio personal *(* */home/<nombredeusuario>* *)*, donde *<nombredeusuario>* es el usuario con el que has iniciado sesión. En Mac, IDLE guarda los archivos en *~/Documentos* por defecto. Si quieres guardarlo en un directorio distinto, especifica la ruta completa. Por ejemplo, para guardar la imagen en *C:\Nen* Windows como *mygraph.png*, llamarías a la función savefig() como se indica a continuación:

>>> savefig('C:\mygraph.png')

Si abres la imagen en un programa de visualización de imágenes, verás el mismo gráfico que verías llamando a la función show(). (Verás que el archivo de imagen sólo contiene el gráfico, no toda la ventana que aparece con la función show() ). Para especificar un formato de imagen diferente, simplemente nombra el archivo con la extensión adecuada. Por ejemplo, mygraph.svg creará un archivo de imagen SVG.

Otra forma de guardar una figura es utilizar el botón Guardar de la ventana que aparece cuando llamas a show().

### **Trazar con fórmulas**

Hasta ahora, hemos estado trazando puntos en nuestros gráficos basándonos en medidas científicas observadas. En esas gráficas, ya teníamos todos nuestros valores para *x* e *y* establecidos. Por ejemplo, las temperaturas y fechas registradas ya estaban a nuestra disposición en el momento en que quisimos crear el gráfico de la ciudad de Nueva York, mostrando cómo variaba la temperatura a lo largo de meses o años. En esta sección, vamos a crear gráficos a partir de fórmulas matemáticas.

#### ***Ley de Gravitación Universal de Newton***

Según la ley de gravitación universal de Newton, un cuerpo de masa m1 atrae a otro cuerpo de masam2 con una fuerza *F* según la fórmula

image

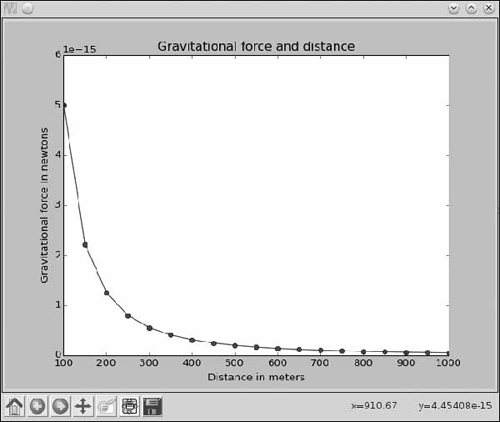
donde *r* es la distancia entre los dos cuerpos y *G* es la constante gravitatoria. Queremos ver qué ocurre con la fuerza a medida que aumenta la distancia entre los dos cuerpos.

Tomemos las masas de dos cuerpos: la masa del primer cuerpo*(*m1) es de 0,5 kg, y la masa del segundo cuerpo*(*m2) es de 1,5 kg. El valor de la constante gravitatoria es 6,674 × 10-11 Nm2 kg-2. Ahora estamos preparados para calcular la fuerza gravitatoria entre estos dos cuerpos a 19 distancias diferentes: 100 m, 150 m, 200 m, 250 m, 300 m, y así hasta 1000 m. El siguiente programa realiza estos cálculos y también dibuja la gráfica:

'''  
The relationship between gravitational force and  
distance between two bodies  
'''  
  
import matplotlib.pyplot as plt  
  
# Draw the graph  
def draw\_graph(x, y):  
plt.plot(x, y, marker='o')  
plt.xlabel('Distance in meters')  
  
plt.ylabel('Gravitational force in newtons')  
plt.title('Gravitational force and distance')  
plt.show()  
  
def generate\_F\_r():  
# Generate values for r  
➊ r = range(100, 1001, 50)  
# Empty list to store the calculated values of F  
F = []  
  
# Constant, G  
G = 6.674\*(10\*\*-11)  
# Two masses  
m1 = 0.5  
m2 = 1.5  
  
# Calculate force and add it to the list, F  
➋ for dist in r:  
force = G\*(m1\*m2)/(dist\*\*2)  
F.append(force)  
  
# Call the draw\_graph function  
➌ draw\_graph(r, F)  
  
if \_\_name\_\_=='\_\_main\_\_':  
generate\_F\_r()

La función generate\_F\_r() realiza la mayor parte del trabajo en el programa anterior. En ➊, utilizamos la función range() para crear una lista denominada r con distintos valores para la distancia, utilizando un valor de paso de 50. El valor final se especifica como 1001 porque queremos que también se incluya 1000. A continuación, creamos una lista vacía (F), donde almacenaremos la fuerza gravitatoria correspondiente a cada una de estas distancias. A continuación, creamos etiquetas referidas a la constante gravitatoria (G) y a las dos masas (m1 y m2). Utilizando un bucle for ➋, calculamos entonces la fuerza en cada uno de los valores de la lista de distancias (r). Utilizamos una etiqueta (force) para referirnos a la fuerza calculada y añadirla a la lista (F). Por último, llamamos a la función draw\_graph() en ➌ con la lista de distancias y la lista de las fuerzas calculadas. El *eje x* del gráfico muestra la fuerza, y el *eje y* muestra la distancia. El gráfico se muestra en la [Figura 2-12](ch02.html#ch2fig12).

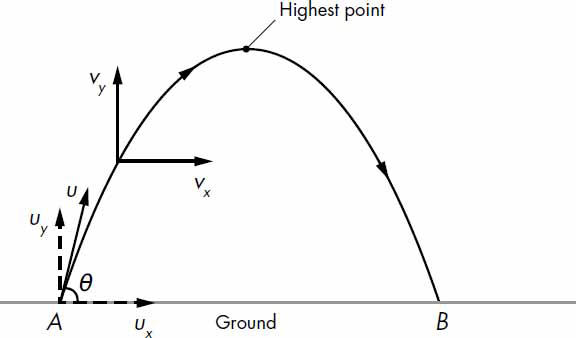
A medida que aumenta la distancia (r), disminuye la fuerza gravitatoria. Con este tipo de relación, decimos que la fuerza gravitatoria es *inversamente proporcional* a la distancia entre los dos cuerpos. Además, ten en cuenta que cuando cambia el valor de una de las dos variables, la otra variable no cambiará necesariamente en la misma proporción. A esto lo llamamos *relación no lineal*. Como resultado, acabamos teniendo una línea curva en la gráfica en lugar de una recta.



*Figura 2-12: Visualización de la relación entre la fuerza gravitatoria y la distancia al cuadrado*

#### ***Movimiento de proyectiles***

Ahora vamos a representar gráficamente algo que te resultará familiar de la vida cotidiana. Si lanzas una pelota a través de un campo, seguirá una trayectoria como la que se muestra en la [Figura 2](ch02.html#ch2fig13)-13.



*Figura 2-13: Movimiento de una pelota lanzada en* el punto A -con un*ángulo (*θ) *y una velocidad (*U)*- y que cae al suelo en el punto* B

En la figura, la pelota es lanzada desde el punto *A* y cae en el punto *B*. Este tipo de movimiento se denomina movimiento de *proyectil*. Nuestro objetivo aquí es utilizar las ecuaciones del movimiento de proyectil para representar gráficamente la trayectoria de un cuerpo, mostrando la posición de la pelota desde el punto en que es lanzada hasta que vuelve a tocar el suelo.

Cuando lanzas la pelota, ésta tiene una velocidad inicial y la dirección de esa velocidad crea un determinado ángulo con el suelo. Llamemos a la velocidad inicial *u* y al ángulo que forma con el suelo *θ* (theta), como se muestra en la [Figura 2-13](ch02.html#ch2fig13). La bola tiene dos componentes de velocidad: una en la dirección *x*, calculada por *ux* = *u* *cosθ*, y otra en la dirección *y*, donde *uy* = *u* *sinθ*.

A medida que la bola se mueve, su velocidad cambia, y representaremos ese cambio de velocidad mediante *v*: la componente horizontal es *vx* y la componente vertical es *vy*. Para simplificar, supongamos que la componente horizontal*(vx*) no cambia durante el movimiento del cuerpo, mientras que la componente vertical*(vy*) disminuye debido a la fuerza de la gravedad según la ecuación *vy* = *uy* *- gt*. En esta ecuación, *g* es la aceleración gravitatoria y *t* es el tiempo en el que se mide la velocidad. Como *uy* = *u* *sinθ*, podemos sustituir para obtener

*vy* = *u* *sinθ* - *gt*.

Como la componente horizontal de la velocidad permanece constante, la distancia horizontal recorrida*(Sx*) viene dada por *Sx* = *u*(*cosθ)t*. Sin embargo, la componente vertical de la velocidad cambia, y la distancia vertical recorrida viene dada por la fórmula

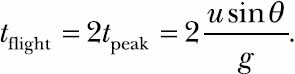
image

En otras palabras, *Sx* y *Sy* nos dan las *coordenadas* *x* e *y* de la bola en un momento dado de su vuelo. Utilizaremos estas ecuaciones cuando escribamos un programa para dibujar la trayectoria. Cuando utilicemos estas ecuaciones, el tiempo*(t*) se expresará en segundos, la velocidad se expresará en m/s, el ángulo de proyección*(θ*) se expresará en grados y la aceleración gravitatoria*(g*) se expresará en m/s2.

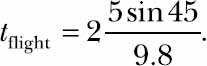
Sin embargo, antes de escribir nuestro programa, tendremos que averiguar cuánto tiempo estará la pelota en vuelo antes de tocar el suelo, para saber cuándo nuestro programa debe dejar de trazar la trayectoria de la pelota. Para ello, primero averiguaremos cuánto tarda la pelota en alcanzar su punto más alto. La pelota alcanza su punto más alto cuando la componente vertical de la velocidad*(vy*) es 0, es decir, cuando *vy* = *u* sen *θ* - *gt* = 0. Por tanto, buscamos el valor *t* mediante la fórmula

image

Llamaremos a este tiempo t\_peak. Tras alcanzar su punto más alto, la pelota tocará el suelo después de estar en el aire durante otros t\_peak segundos, por lo que el tiempo total de vuelo (t\_flight) de la pelota es



Tomemos una pelota lanzada con una velocidad inicial*(u*) de 5 m/s en un ángulo*(θ*) de 45 grados. Para calcular el tiempo total de vuelo, sustituimos *u* = 5, *θ* = 45 y *g* = 9,8 en la ecuación que vimos anteriormente:



En este caso, el tiempo de vuelo de la pelota resulta ser de 0,72154 segundos (redondeado a cinco decimales). La pelota estará en el aire durante este periodo de tiempo, así que para dibujar la trayectoria, calcularemos sus *coordenadas* *x* e y a intervalos regulares durante este periodo de tiempo. ¿Con qué frecuencia debemos calcular las coordenadas? Idealmente, con la mayor frecuencia posible. En este capítulo, calcularemos las coordenadas cada 0,001 segundos.

##### **Generación de números en coma flotante equidistantes**

Hemos utilizado la función range() para generar números enteros igualmente espaciados; es decir, si quisiéramos una lista de números enteros entre 1 y 10 con cada número entero separado por 1, utilizaríamos range(1, 10). Si quisiéramos un valor de paso diferente, podríamos especificarlo a la función rango como tercer argumento. Por desgracia, no existe una función incorporada de este tipo para los números de coma flotante. Así que, por ejemplo, no hay ninguna función que nos permita crear una lista de los números de 0 a 0,72 con dos números consecutivos separados por 0,001. Podemos utilizar un bucle while como el siguiente para crear nuestra propia función para esto:

'''  
Generate equally spaced floating point  
numbers between two given values  
'''  
  
def frange(start, final, increment):  
  
numbers = []  
➊ while start < final:  
➋ numbers.append(start)  
start = start + increment  
  
return numbers

Hemos definido una función frange() (rango "punto flotante") que recibe tres parámetros: start y final se refieren a los puntos inicial y final del rango de números, y increment se refiere a la diferencia entre dos números consecutivos. Inicializamos un bucle while en ➊, que continúa la ejecución mientras el número apuntado por start sea menor que el valor de final. Almacenamos el número apuntado por start en la lista numbers ➋ y luego añadimos el valor que hemos introducido como increment en cada iteración del bucle. Por último, devolvemos la lista numbers.

Utilizaremos esta función para generar instantes de tiempo igualmente espaciados en el programa de dibujo de trayectorias que se describe a continuación.

##### **Dibujar la trayectoria**

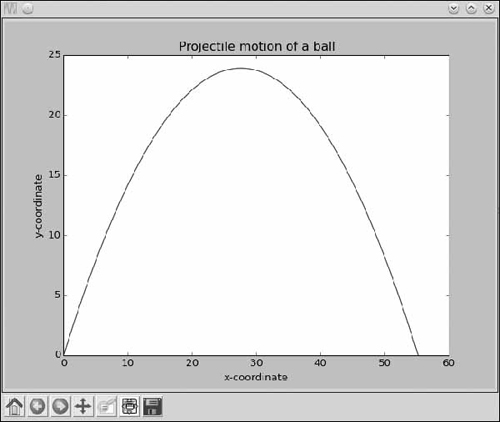
El siguiente programa dibuja la trayectoria de una pelota lanzada con una velocidad y un ángulo determinados, ambos suministrados como entrada al programa:

'''  
Draw the trajectory of a body in projectile motion  
'''  
  
from matplotlib import pyplot as plt  
import math  
  
def draw\_graph(x, y):  
plt.plot(x, y)  
plt.xlabel('x-coordinate')  
plt.ylabel('y-coordinate')  
plt.title('Projectile motion of a ball')  
  
def frange(start, final, interval):  
  
numbers = []  
while start < final:  
numbers.append(start)  
start = start + interval  
  
return numbers  
  
def draw\_trajectory(u, theta):  
  
➊ theta = math.radians(theta)  
g = 9.8  
  
# Time of flight  
➋ t\_flight = 2\*u\*math.sin(theta)/g  
# Find time intervals  
intervals = frange(0, t\_flight, 0.001)  
  
# List of x and y coordinates  
x = []  
y = []  
➌ for t in intervals:  
x.append(u\*math.cos(theta)\*t)  
y.append(u\*math.sin(theta)\*t - 0.5\*g\*t\*t)  
  
draw\_graph(x, y)  
  
if \_\_name\_\_ == '\_\_main\_\_':  
➍ 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:  
draw\_trajectory(u, theta)  
plt.show()

En este programa, utilizamos las funciones radians(), cos(), y sin() definidas en el módulo math de la biblioteca estándar, por lo que importamos dicho módulo al principio. La función draw\_trajectory() acepta dos argumentos, u y theta, correspondientes a la velocidad y al ángulo con el que se lanza la bola. Las funciones seno y coseno del módulo math esperan que el ángulo se proporcione en radianes, así que en ➊, convertimos el ángulo (theta) de grados a radianes utilizando la función math.radians(). A continuación, creamos una etiqueta (g) para referirnos al valor de la aceleración debida a la gravedad, 9,8 m/s2. En ➋, calculamos el tiempo de vuelo y luego llamamos a la función frange() con los valores de start, final, y increment fijados en 0, t\_flight, y 0,001, respectivamente. A continuación, calculamos las coordenadas *x* e *y* de la trayectoria en cada uno de los instantes de tiempo y las almacenamos en dos listas distintas, x y y ➌. Para calcular estas coordenadas, utilizamos las fórmulas para las distancias *Sx* y *Sy* que hemos discutido antes.

Por último, llamamos a la función draw\_graph() con las coordenadas *x* e *y* para dibujar la trayectoria. Observa que la función draw\_graph() no llama a la función show() (veremos por qué en el siguiente programa). Utilizamos un bloque try...except ➍ para informar de un mensaje de error en caso de que el usuario introduzca una entrada no válida. La entrada válida para este programa es cualquier número entero o de coma flotante. Cuando ejecutas el programa, te pide estos valores como entrada y luego dibuja la trayectoria (ver [Figura 2](ch02.html#ch2fig14)-14):

Enter the initial velocity (m/s): 25  
Enter the angle of projection (degrees): 60



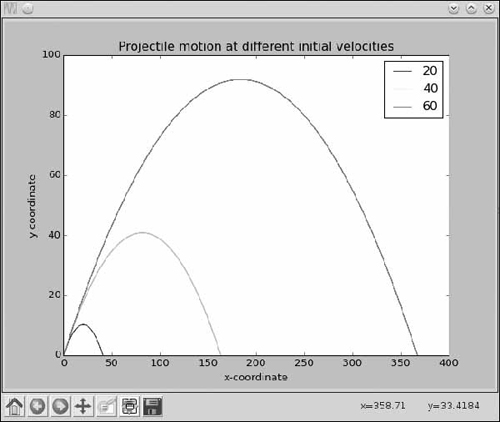
*Figura 2-14: Trayectoria de una pelota lanzada con una velocidad de 25 m/s y un ángulo de 60 grados*

##### **Comparación de la trayectoria con distintas velocidades iniciales**

El programa anterior te permite realizar experimentos interesantes. Por ejemplo, ¿cómo sería la trayectoria de tres pelotas lanzadas a distintas velocidades pero con el mismo ángulo inicial? Para representar gráficamente tres trayectorias a la vez, podemos sustituir el bloque de código main de nuestro programa anterior por el siguiente:

if \_\_name\_\_ == '\_\_main\_\_':  
  
# List of three different initial velocities  
➊ u\_list = [20, 40, 60]  
theta = 45  
for u in u\_list:  
draw\_trajectory(u, theta)  
  
# Add a legend and show the graph  
➋ plt.legend(['20', '40', '60'])  
plt.show()

Aquí, en lugar de pedir al usuario del programa que introduzca la velocidad y el ángulo de proyección, creamos una lista (u\_list) con las velocidades 20, 40 y 60 en ➊ y fijamos el ángulo de proyección en 45 grados (utilizando la etiqueta theta). A continuación, llamamos a la función draw\_trajectory() con cada uno de los tres valores de u\_list utilizando el mismo valor para theta, que calcula la lista de *coordenadas* *x* e *y* y llama a la función draw\_graph(). Cuando llamamos a la función show(), las tres gráficas se muestran en el mismo gráfico. Como ahora tenemos una gráfica con múltiples trazados, añadimos una leyenda a la gráfica en ➋ antes de llamar a show() para mostrar la velocidad de cada línea. Cuando ejecutes el programa anterior, verás la gráfica que se muestra en la [Figura 2-15](ch02.html#ch2fig15).



*Figura 2-15: La trayectoria de una pelota lanzada en un ángulo de 60 grados, con una velocidad de 20, 40 y 60 m/s*

### **Lo que has aprendido**

En este capítulo has aprendido lo básico para crear gráficos con matplotlib. Has visto cómo trazar un único conjunto de valores, cómo crear varios trazados en el mismo gráfico y cómo etiquetar varias partes de un gráfico para hacerlo más informativo. Utilizaste gráficas para analizar la variación de temperatura de una ciudad, estudiar la ley de gravitación universal de Newton y estudiar el movimiento de proyectil de un cuerpo. En el próximo capítulo, utilizarás Python para empezar a explorar la estadística, y verás cómo dibujar un gráfico puede ayudar a comprender mejor las relaciones entre conjuntos de números.

### **Retos de programación**

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

#### ***#nº 1: ¿Cómo varía la temperatura durante el día?***

Si introduces un término de búsqueda como "el tiempo en Nueva York" en el buscador de Google, verás, entre otras cosas, un gráfico que muestra la temperatura en diferentes momentos del día actual. Tu tarea aquí es recrear ese gráfico.

Utilizando una ciudad de tu elección, encuentra la temperatura en diferentes momentos del día. Utiliza los datos para crear dos listas en tu programa y para crear un gráfico con la hora del día en el *eje x* y la temperatura correspondiente en el *eje y*. El gráfico debe indicarte cómo varía la temperatura con la hora del día. Prueba con otra ciudad y observa cómo se comparan las dos ciudades trazando ambas líneas en el mismo gráfico.

La hora del día puede indicarse mediante cadenas como '10:11 AM' o '09:21 PM'.

#### ***#nº 2: Exploración visual de una función cuadrática***

En el [capítulo 1](ch01.html#ch01), aprendiste a encontrar las raíces de una ecuación cuadrática, como x2 +*2x* + 1 = 0. Podemos convertir esta ecuación en una función escribiéndola como *y* = x2 +*2x* + 1. Para cualquier valor de *x*, la función cuadrática produce *algún* valor para *y*. Por ejemplo, cuando *x* = 1, *y* = 4. Aquí tienes un programa que calcula el valor de *y* para seis valores distintos de *x*:

'''  
Quadratic function calculator  
'''  
  
# Assume values of x  
➊ x\_values = [-1, 1, 2, 3, 4, 5]  
➋ for x in x\_values:  
# Calculate the value of the quadratic function  
y = x\*\*2 + 2\*x + 1  
print('x={0} y={1}'.format(x, y))

En ➊, creamos una lista con seis valores diferentes para x. El bucle for que comienza en ➋ calcula el valor de la función anterior para cada uno de estos valores y utiliza la etiqueta y para referirse a la lista de resultados. A continuación, imprimimos el valor de x y el valor correspondiente de y. Cuando ejecutes el programa, deberías ver la siguiente salida:

x=-1 y=0  
x=1 y=4  
x=2 y=9  
  
x=3 y=16  
x=4 y=25  
x=5 y=36

Observa que la primera línea de la salida es una raíz de la ecuación cuadrática porque es un valor de x que hace que la función sea igual a 0.

Tu reto de programación es mejorar este programa para crear una gráfica de la función. Intenta utilizar al menos 10 valores para x en lugar de los 6 anteriores. Calcula los valores correspondientes de y utilizando la función y luego crea una gráfica utilizando estos dos conjuntos de valores.

Una vez que hayas creado la gráfica, dedica algún tiempo a analizar cómo varía el valor de *y* con respecto a *x*. ¿La variación es lineal o no lineal?

#### ***#nº 3: Programa de comparación de trayectorias de proyectiles mejoradas***

Tu reto aquí es mejorar el programa de comparación de trayectorias de varias maneras. En primer lugar, tu programa debe imprimir el tiempo de vuelo, la distancia horizontal máxima y la distancia vertical máxima recorrida para cada una de las combinaciones de velocidad y ángulo de proyección.

La otra mejora consiste en hacer que el programa funcione con cualquier número de valores iniciales de velocidad y ángulo de proyección, suministrados por el usuario. Por ejemplo, así es como el programa debería pedir las entradas al usuario:

How many trajectories? 3  
Enter the initial velocity for trajectory 1 (m/s): 45  
Enter the angle of projection for trajectory 1 (degrees): 45  
Enter the initial velocity for trajectory 2 (m/s): 60  
Enter the angle of projection for trajectory 2 (degrees): 45  
Enter the initial velocity for trajectory(m/s) 3: 45  
Enter the angle of projection for trajectory(degrees) 3: 90

Tu programa también debería asegurarse de que las entradas erróneas se gestionan adecuadamente mediante un bloque try...except, igual que en el programa original.

#### ***#nº 4: Visualizar tus gastos***

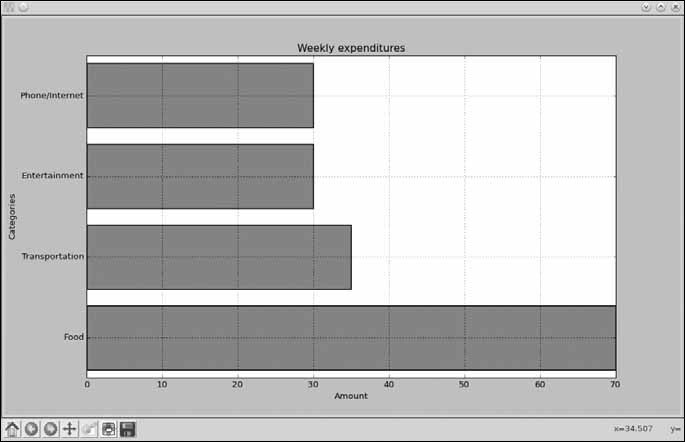
Siempre me pregunto a final de mes: "¿Adónde ha ido a parar todo ese dinero?". Estoy seguro de que no es un problema al que me enfrente sólo yo.

Para este reto, escribirás un programa que cree un gráfico de barras para comparar fácilmente los gastos semanales. El programa debe pedir primero el número de categorías para los gastos y el gasto total semanal en cada categoría, y luego debe crear el gráfico de barras que muestre estos gastos.

Aquí tienes un ejemplo de cómo debería funcionar el programa:

Enter the number of categories: 4  
Enter category: Food  
Expenditure: 70  
  
Enter category: Transportation  
Expenditure: 35  
Enter category: Entertainment  
Expenditure: 30  
Enter category: Phone/Internet  
Expenditure: 30

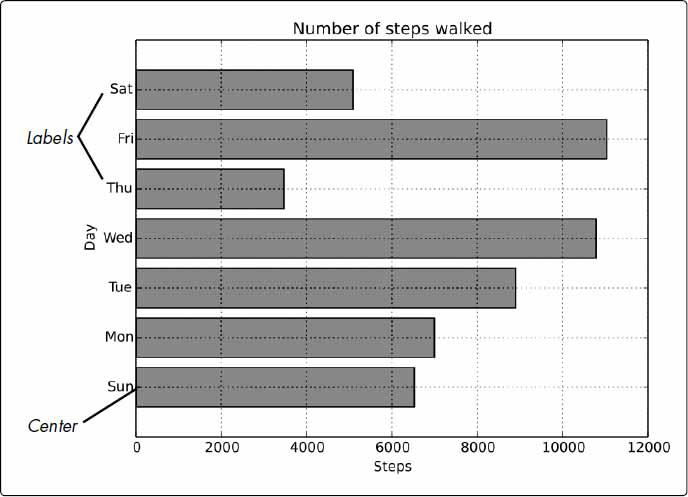
[La Figura 2-16](ch02.html#ch2fig16) muestra el gráfico de barras que se creará para comparar los gastos. Si guardas el gráfico de barras para cada semana, al final del mes podrás ver cómo han variado los gastos entre las semanas para las distintas categorías.



*Figura 2-16: Gráfico de barras que muestra los gastos por categoría durante la semana*

No hemos hablado de la creación de un gráfico de barras con matplotlib, así que probemos con un ejemplo.

Se puede crear un gráfico de barras utilizando la función barh() de matplotlib, que también está definida en el módulo pyplot. [La Figura 2-17](ch02.html#ch2fig17) muestra un gráfico de barras que ilustra el número de pasos que he dado durante la semana pasada. Los días de la semana -domingo, lunes, martes, etc.- se denominan *etiquetas*. Cada barra horizontal parte del *eje y*, y tenemos que especificar la *coordenada y* del *centro* de esta posición para cada una de las barras. La longitud de cada barra corresponde al número de pasos especificado.



*Figura 2-17: Gráfico de barras que muestra el número de pasos caminados durante una semana*

El siguiente programa crea el gráfico de barras:

'''  
Example of drawing a horizontal bar chart  
'''  
import matplotlib.pyplot as plt  
def create\_bar\_chart(data, labels):  
# Number of bars  
num\_bars = len(data)  
# This list is the point on the y-axis where each  
# Bar is centered. Here it will be [1, 2, 3...]  
➊ positions = range(1, num\_bars+1)  
➋ plt.barh(positions, data, align='center')  
# Set the label of each bar  
plt.yticks(positions, labels)  
plt.xlabel('Steps')  
plt.ylabel('Day')  
plt.title('Number of steps walked')  
# Turns on the grid which may assist in visual estimation  
plt.grid(  
plt.show()  
  
if \_\_name\_\_ == '\_\_main\_\_':  
# Number of steps I walked during the past week  
steps = [6534, 7000, 8900, 10786, 3467, 11045, 5095]  
# Corresponding days  
labels = ['Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat']  
create\_bar\_chart(steps, labels)

La función create\_bar\_chart() acepta dos parámetros:data, que es una lista de números que queremos representar mediante las barras y etiquetas, y la correspondiente lista labels. Hay que especificar el centro de cada barra, y yo he elegido arbitrariamente los centros 1, 2, 3, 4, etc. utilizando la ayuda de la función range() en ➊.

A continuación, llamamos a la función barh(), pasando positions y data como los dos primeros argumentos y luego el argumento de palabra clave, align='center', en ➋. El argumento de la palabra clave especifica que las barras se centren en las posiciones del *eje y* especificadas por la lista. A continuación, establecemos las etiquetas de cada barra, las etiquetas de los ejes y el título mediante la función yticks(). También llamamos a la función grid() para activar la cuadrícula, que puede ser útil para una estimación visual del número de pasos. Por último, llamamos a la función show().

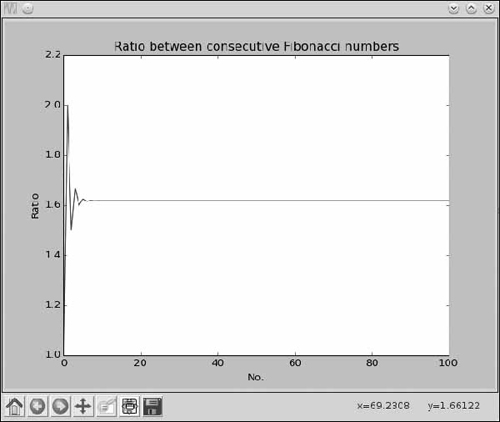
#### ***#5: Exploración de la relación entre la secuencia de Fibonacci y la proporción áurea***

La secuencia de Fibonacci (1, 1, 2, 3, 5, ...) es la serie de números en la que el *i-ésimo*número de la serie es la suma de los dos números anteriores, es decir, los números en las posiciones*(i* - 2) y*(i* - 1). Los números sucesivos de esta serie muestran una relación interesante. A medida que aumentas el número de términos de la serie, los cocientes de los pares de números consecutivos son casi iguales entre sí. Este valor se aproxima a un número especial denominado *razón* áurea. Numéricamente, la proporción áurea es el número 1,618033988 ..., y ha sido objeto de amplios estudios en música, arquitectura y naturaleza. Para este reto, escribe un programa que trace en un gráfico la relación entre números de Fibonacci consecutivos para, digamos, 100 números, lo que demostrará que los valores se aproximan a la proporción áurea.

La siguiente función, que devuelve una lista de los *n* primeros números de Fibonacci, puede resultarte útil para implementar tu solución:

def fibo(n):  
if n == 1:  
return [1]  
if n == 2:  
return [1, 1]  
# n > 2  
a = 1  
b = 1  
# First two members of the series  
series = [a, b]  
for i in range(n):  
c = a + b  
series.append(c)  
a = b  
b = c  
  
return series

La salida de tu solución debe ser un gráfico, como el que se muestra en la [Figura 2](ch02.html#ch2fig18)-18.



*Figura 2-18: La relación entre los números de Fibonacci consecutivos se aproxima a la proporción áurea.*