# 17 Trabajar con API



En este capítulo, aprenderás a escribir un programa autónomo que genere una visualización basada en los datos que recupere. Tu programa utilizará una *application programming interface (API)* para solicitar automáticamente información específica de un sitio web y luego utilizará esa información para generar una visualización. Como los programas escritos así siempre utilizarán datos actuales para generar una visualización, incluso cuando esos datos puedan estar cambiando rápidamente, la visualización siempre estará actualizada.

## Utilizar una API

Una API es una parte de un sitio web diseñada para interactuar con programas. Esos programas utilizan URL muy específicas para solicitar determinada información. Este tipo de solicitud se denomina *API call*. Los datos solicitados se devolverán en un formato fácilmente procesable, como JSON o CSV. La mayoría de las aplicaciones que utilizan fuentes de datos externas, como las que se integran con sitios de redes sociales, se basan en llamadas a la API.

### Git y GitHub

Basaremos nuestra visualización en información de GitHub[(https://github.com)](https://github.com), un sitio que permite a los programadores colaborar en proyectos de codificación. Utilizaremos la API de GitHub para solicitar información sobre proyectos Python en el sitio, y luego generaremos una visualización interactiva de la popularidad relativa de estos proyectos utilizando Plotly.

GitHub toma su nombre de Git, un sistema distribuido de control de versiones. Git ayuda a las personas a gestionar su trabajo en un proyecto de forma que se evite que los cambios realizados por una persona interfieran con los cambios que están realizando otras personas. Cuando implementas una nueva función en un proyecto, Git rastrea los cambios que haces en cada archivo. Cuando tu nuevo código funciona, *commit* los cambios que has hecho, y Git registra el nuevo estado de tu proyecto. Si cometes un error y quieres revertir tus cambios, puedes volver fácilmente a cualquier estado que funcionara anteriormente. (Para saber más sobre el control de versiones mediante Git, consulta el Apéndice D.) Los proyectos en GitHub se almacenan en *repositories*, que contiene todo lo relacionado con el proyecto: su código, información sobre sus colaboradores, cualquier incidencia o informe de errores, etc.

Cuando a los usuarios de GitHub les gusta un proyecto, pueden ponerle una "estrella" para mostrar su apoyo y hacer un seguimiento de los proyectos que podrían querer utilizar. En este capítulo, escribiremos un programa para descargar automáticamente información sobre los proyectos Python con más estrellas en GitHub, y luego crearemos una visualización informativa de estos proyectos.

### Solicitar datos mediante una llamada a la API

La API de GitHub te permite solicitar una amplia gama de información a través de llamadas a la API. Para ver cómo es una llamada a la API, introduce lo siguiente en la barra de direcciones de tu navegador y pulsa INTRO:

https://api.github.com/search/repositories?q=language:python+sort:stars

Esta llamada devuelve el número de proyectos Python alojados actualmente en GitHub, así como información sobre los repositorios Python más populares. Examinemos la llamada. La primera parte, https://api.github.com/, dirige la petición a la parte de GitHub que responde a las llamadas a la API. La siguiente parte, search/repositories, indica a la API que realice una búsqueda en todos los repositorios de GitHub.

El signo de interrogación después de repositories indica que vamos a pasar un argumento. El q significa *query*, y el signo igual (=) nos permite empezar a especificar una consulta (q=). Al utilizar language:python, indicamos que sólo queremos información sobre repositorios que tengan Python como lenguaje principal. La parte final, +sort:stars, ordena los proyectos por el número de estrellas que han recibido.

El siguiente fragmento muestra las primeras líneas de la respuesta:

{  
❶ "total\_count": 8961993,  
❷ "incomplete\_results": true,  
❸ "items": [  
 {  
 "id": 54346799,  
 "node\_id": "MDEwOlJlcG9zaXRvcnk1NDM0Njc5OQ==",  
 "name": "public-apis",  
 "full\_name": "public-apis/public-apis",  
 --snip--

Puedes ver en la respuesta que esta URL no está pensada principalmente para ser introducida por humanos, ya que está en un formato pensado para ser procesado por un programa. GitHub ha encontrado algo menos de nueve millones de proyectos Python en el momento de escribir esto ❶. El valor de "incomplete\_results" es true, lo que nos indica que GitHub no procesó completamente la consulta ❷. GitHub limita el tiempo de ejecución de cada consulta para que la API responda a todos los usuarios. En este caso encontró algunos de los repositorios de Python más populares, pero no tuvo tiempo de encontrarlos todos; lo arreglaremos en un momento. Los "items" devueltos se muestran en la lista siguiente, que contiene detalles sobre los proyectos Python más populares en GitHub ❸.

### Instalar solicitudes

El paquete *Requests* permite a un programa Python solicitar fácilmente información a un sitio web y examinar la respuesta. Utiliza pip para instalar Peticiones:

$ python -m pip install --user requests

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

$ python3 -m pip install --user requests

### Procesar una respuesta de la API

Ahora escribiremos un programa para realizar automáticamente una llamada a la API y procesar los resultados:

**python\_repos.py**

import requests  
  
# Make an API call and check the response.  
❶ url = "https://api.github.com/search/repositories"  
url += "?q=language:python+sort:stars+stars:>10000"  
  
❷ headers = {"Accept": "application/vnd.github.v3+json"}  
❸ r = requests.get(url, headers=headers)  
❹ print(f"Status code: {r.status\_code}")  
  
# Convert the response object to a dictionary.  
❺ response\_dict = r.json()  
  
# Process results.  
print(response\_dict.keys())

Primero importamos el módulo requests. Luego asignamos la URL de la llamada a la API a la variable url ❶. Se trata de una URL larga, así que la dividimos en dos líneas. La primera línea es la parte principal de la URL, y la segunda es la cadena de consulta. Hemos incluido una condición más a la cadena de consulta original: stars:>10000, que indica a GitHub que sólo busque repositorios de Python que tengan más de 10.000 estrellas. Esto debería permitir a GitHub devolver un conjunto de resultados completo y coherente.

GitHub está actualmente en la tercera versión de su API, por lo que definimos cabeceras para la llamada a la API que piden explícitamente utilizar esta versión de la API, y devuelven los resultados en formato JSON ❷. A continuación, utilizamos requests para realizar la llamada a la API ❸. Llamamos a get() y le pasamos la URL y la cabecera que hemos definido, y asignamos el objeto respuesta a la variable r.

El objeto respuesta tiene un atributo llamado status\_code, que nos indica si la solicitud se ha realizado correctamente. (Un código de estado 200 indica una respuesta correcta.) Imprimimos el valor de status\_code para asegurarnos de que la llamada se ha realizado correctamente ❹. Pedimos a la API que devolviera la información en formato JSON, así que utilizamos el método json() para convertir la información en un diccionario Python ❺. Asignamos el diccionario resultante a response\_dict.

Por último, imprimimos las claves de response\_dict y vemos la siguiente salida:

Status code: 200  
dict\_keys(['total\_count', 'incomplete\_results', 'items'])

Como el código de estado es 200, sabemos que la solicitud se ha realizado correctamente. El diccionario de respuesta sólo contiene tres claves: 'total\_count', 'incomplete\_results', y 'items'. Echemos un vistazo al diccionario de respuesta.

### Trabajar con el diccionario de respuesta

Con la información de la llamada a la API representada como un diccionario, podemos trabajar con los datos allí almacenados. Vamos a generar alguna salida que resuma la información. Es una buena forma de asegurarnos de que hemos recibido la información que esperábamos y de empezar a examinar la información que nos interesa:

**python\_repos.py**

import requests  
  
# Make an API call and store the response.  
--snip--  
  
# Convert the response object to a dictionary.  
response\_dict = r.json()  
❶ print(f"Total repositories: {response\_dict['total\_count']}")  
print(f"Complete results: {not response\_dict['incomplete\_results']}")  
  
# Explore information about the repositories.  
❷ repo\_dicts = response\_dict['items']  
print(f"Repositories returned: {len(repo\_dicts)}")  
  
# Examine the first repository.  
❸ repo\_dict = repo\_dicts[0]  
❹ print(f"\nKeys: {len(repo\_dict)}")  
❺ for key in sorted(repo\_dict.keys()):  
 print(key)

Empezamos a explorar el diccionario de respuestas imprimiendo el valor asociado a 'total\_count', que representa el número total de repositorios Python devueltos por esta llamada a la API ❶. También utilizamos el valor asociado a 'incomplete\_results', para saber si GitHub pudo procesar completamente la consulta. En lugar de imprimir este valor directamente, imprimimos su opuesto: un valor de True indicará que hemos recibido un conjunto completo de resultados.

El valor asociado a 'items' es una lista que contiene una serie de diccionarios, cada uno de los cuales contiene datos sobre un repositorio Python individual. Asignamos esta lista de diccionarios a repo\_dicts ❷. A continuación, imprimimos la longitud de repo\_dicts para ver de cuántos repositorios tenemos información.

Para ver más de cerca la información devuelta sobre cada repositorio, sacamos el primer elemento de repo\_dicts y lo asignamos a repo\_dict ❸. A continuación, imprimimos el número de claves del diccionario para ver cuánta información tenemos ❹. Por último, imprimimos todas las claves del diccionario para ver qué tipo de información se incluye ❺.

Los resultados nos dan una idea más clara de los datos reales:

Status code: 200  
❶ Total repositories: 248  
❷ Complete results: True  
Repositories returned: 30  
  
❸ Keys: 78  
allow\_forking  
archive\_url  
archived  
--snip--  
url  
visiblity  
watchers  
watchers\_count

En el momento de escribir esto, sólo hay 248 repositorios de Python con más de 10.000 estrellas ❶. Podemos ver que GitHub fue capaz de procesar completamente la llamada a la API ❷. En esta respuesta, GitHub devolvió información sobre los primeros 30 repositorios que coinciden con las condiciones de nuestra consulta. Si queremos más repositorios, podemos solicitar páginas adicionales de datos.

La API de GitHub devuelve mucha información sobre cada repositorio: hay 78 claves en repo\_dict ❸. Cuando eches un vistazo a estas claves, te harás una idea del tipo de información que puedes extraer sobre un proyecto. (La única forma de saber qué información está disponible a través de una API es leer la documentación o examinar la información a través del código, como estamos haciendo aquí).

Vamos a extraer los valores de algunas de las claves de repo\_dict:

**python\_repos.py**

--snip--  
# Examine the first repository.  
repo\_dict = repo\_dicts[0]  
  
print("\nSelected information about first repository:")  
❶ print(f"Name: {repo\_dict['name']}")  
❷ print(f"Owner: {repo\_dict['owner']['login']}")  
❸ print(f"Stars: {repo\_dict['stargazers\_count']}")  
print(f"Repository: {repo\_dict['html\_url']}")  
❹ print(f"Created: {repo\_dict['created\_at']}")  
❺ print(f"Updated: {repo\_dict['updated\_at']}")  
print(f"Description: {repo\_dict['description']}")

Aquí imprimimos los valores de varias claves del diccionario del primer repositorio. Empezamos con el nombre del proyecto ❶. Un diccionario entero representa al propietario del proyecto, así que utilizamos la clave owner para acceder al diccionario que representa al propietario, y luego utilizamos la clave login para obtener el nombre de usuario del propietario ❷. A continuación, imprimimos cuántas estrellas ha obtenido el proyecto ❸ y la URL del repositorio GitHub del proyecto. A continuación, mostramos cuándo se creó ❹ y cuándo se actualizó por última vez ❺. Por último, imprimimos la descripción del repositorio.

El resultado debería ser algo parecido a esto

Status code: 200  
Total repositories: 248  
Complete results: True  
Repositories returned: 30  
  
Selected information about first repository:  
Name: public-apis  
Owner: public-apis  
Stars: 191493  
Repository: https://github.com/public-apis/public-apis  
Created: 2016-03-20T23:49:42Z  
Updated: 2022-05-12T06:37:11Z  
Description: A collective list of free APIs

Podemos ver que el proyecto Python con más estrellas en GitHub en el momento de escribir esto es *public-apis*. Su propietario es una organización con el mismo nombre, y ha recibido estrellas de casi 200.000 usuarios de GitHub. Podemos ver la URL del repositorio del proyecto, su fecha de creación en marzo de 2016, y que se ha actualizado recientemente. Además, la descripción nos dice que *public-apis* contiene una lista de API gratuitas que podrían interesar a los programadores.

### Resumen de los principales repositorios

Cuando hagamos una visualización de estos datos, querremos incluir más de un repositorio. Escribamos un bucle para imprimir la información seleccionada sobre cada repositorio que devuelva la llamada a la API, de forma que podamos incluirlos todos en la visualización:

**python\_repos.py**

--snip--  
# Explore information about the repositories.  
repo\_dicts = response\_dict['items']  
print(f"Repositories returned: {len(repo\_dicts)}")  
  
❶ print("\nSelected information about each repository:")  
❷ for repo\_dict in repo\_dicts:  
 print(f"\nName: {repo\_dict['name']}")  
 print(f"Owner: {repo\_dict['owner']['login']}")  
 print(f"Stars: {repo\_dict['stargazers\_count']}")  
 print(f"Repository: {repo\_dict['html\_url']}")  
 print(f"Description: {repo\_dict['description']}")

Primero imprimimos un mensaje introductorio ❶. Después hacemos un bucle a través de todos los diccionarios en repo\_dicts ❷. Dentro del bucle, imprimimos el nombre de cada proyecto, su propietario, cuántas estrellas tiene, su URL en GitHub y la descripción del proyecto:

Status code: 200  
Total repositories: 248  
Complete results: True  
Repositories returned: 30  
  
Selected information about each repository:  
  
Name: public-apis  
Owner: public-apis  
Stars: 191494  
Repository: https://github.com/public-apis/public-apis  
Description: A collective list of free APIs  
  
Name: system-design-primer  
Owner: donnemartin  
Stars: 179952  
Repository: https://github.com/donnemartin/system-design-primer  
Description: Learn how to design large-scale systems. Prep for the system  
 design interview. Includes Anki flashcards.  
--snip--  
  
Name: PayloadsAllTheThings  
Owner: swisskyrepo  
Stars: 37227  
Repository: https://github.com/swisskyrepo/PayloadsAllTheThings  
Description: A list of useful payloads and bypass for Web Application Security  
 and Pentest/CTF

En estos resultados aparecen algunos proyectos interesantes, y puede que merezca la pena echar un vistazo a unos cuantos. Pero no pases demasiado tiempo aquí, porque estamos a punto de crear una visualización que hará que los resultados sean mucho más fáciles de leer.

### Supervisar los límites de velocidad de la API

La mayoría de las API tienen *rate limits*, lo que significa que hay un límite en el número de peticiones que puedes hacer en un determinado periodo de tiempo. Para ver si te estás acercando a los límites de GitHub, introduce [https://api.github.com/rate\_limit](https://api.github.com/rate_limit%20) en un navegador web. Deberías ver una respuesta que empiece así:

{  
 "resources": {  
 --snip--  
❶ "search": {  
❷ "limit": 10,  
❸ "remaining": 9,  
❹ "reset": 1652338832,  
 "used": 1,  
 "resource": "search"  
 },  
 --snip--

La información que nos interesa es el límite de tasa para la API de búsqueda ❶. Vemos que el límite es de 10 peticiones por minuto ❷ y que nos quedan 9 peticiones para el minuto actual ❸. El valor asociado a la clave "reset" representa el tiempo en *Unix* o *epoch time* (el número de segundos desde la medianoche del 1 de enero de 1970) en que se restablecerá nuestra cuota ❹. Si alcanzas tu cuota, recibirás una breve respuesta que te informará de que has llegado al límite de la API. Si alcanzas el límite, espera a que se restablezca tu cuota.

## Nota

Muchas API requieren que te registres y obtengas una clave API o un token de acceso para realizar llamadas a la API. En el momento de escribir esto, GitHub no tiene ese requisito, pero si obtienes un token de acceso, tus límites serán mucho mayores.

## Visualizar repositorios con Plotly

Vamos a hacer una visualización utilizando los datos que hemos recopilado para mostrar la popularidad relativa de los proyectos Python en GitHub. Haremos un gráfico de barras interactivo: la altura de cada barra representará el número de estrellas que ha adquirido el proyecto, y podrás hacer clic en la etiqueta de la barra para ir a la página principal de ese proyecto en GitHub.

Guarda una copia del programa en el que hemos estado trabajando como *python\_repos\_visual.py*, y luego modifícalo para que quede como sigue:

**python\_repos\_visual.py**

import requests  
import plotly.express as px  
  
# Make an API call and check the response.  
url = "https://api.github.com/search/repositories"  
url += "?q=language:python+sort:stars+stars:>10000"  
  
headers = {"Accept": "application/vnd.github.v3+json"}  
r = requests.get(url, headers=headers)  
❶ print(f"Status code: {r.status\_code}")  
  
# Process overall results.  
response\_dict = r.json()  
❷ print(f"Complete results: {not response\_dict['incomplete\_results']}")  
  
# Process repository information.  
repo\_dicts = response\_dict['items']  
❸ repo\_names, stars = [], []  
for repo\_dict in repo\_dicts:  
 repo\_names.append(repo\_dict['name'])  
 stars.append(repo\_dict['stargazers\_count'])  
  
# Make visualization.  
❹ fig = px.bar(x=repo\_names, y=stars)  
fig.show()

Importamos Plotly Express y realizamos la llamada a la API como hemos venido haciendo. Seguimos imprimiendo el estado de la respuesta de la llamada a la API para saber si hay algún problema ❶. Cuando procesemos los resultados globales, seguimos imprimiendo el mensaje que confirma que hemos obtenido un conjunto completo de resultados ❷. Eliminamos el resto de las llamadas a print() porque ya no estamos en la fase exploratoria; sabemos que tenemos los datos que queremos.

A continuación, creamos dos listas vacías ❸ para almacenar los datos que incluiremos en el gráfico inicial. Necesitaremos el nombre de cada proyecto para etiquetar las barras (repo\_names) y el número de estrellas para determinar la altura de las barras (stars). En el bucle, añadimos a estas listas el nombre de cada proyecto y el número de estrellas que tiene.

Hacemos la visualización inicial con sólo dos líneas de código ❹. Esto es coherente con la filosofía de Plotly Express de que debes poder ver tu visualización lo más rápidamente posible antes de refinar su aspecto. Aquí utilizamos la función px.bar() para crear un gráfico de barras. Pasamos la lista repo\_names como argumento x y stars como argumento y.

[La Figura 17-1](#figure17-1) muestra el gráfico resultante. Podemos ver que los primeros proyectos son significativamente más populares que el resto, pero todos ellos son proyectos importantes en el ecosistema Python.

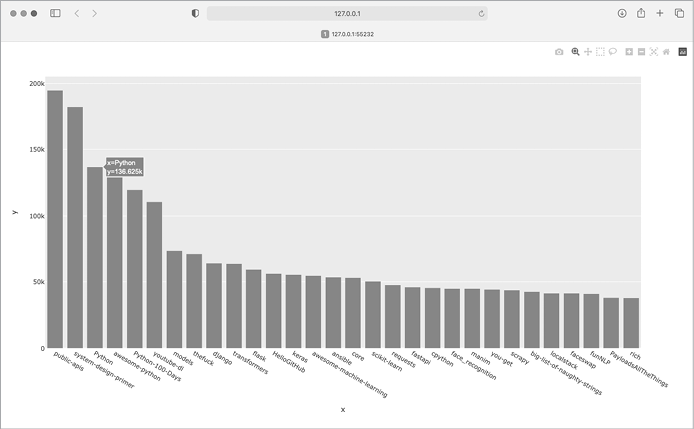


Figura 17-1: Los proyectos Python con más estrellas en GitHub

### Estilizar el gráfico

Plotly admite varias formas de dar estilo y personalizar los gráficos, una vez que sabes que la información del gráfico es correcta. Haremos algunos cambios en la llamada inicial a px.bar() y luego haremos otros ajustes en el objeto fig una vez creado.

Empezaremos a dar estilo al gráfico añadiendo un título y etiquetas para cada eje:

**python\_repos\_visual.py**

--snip--  
# Make visualization.  
title = "Most-Starred Python Projects on GitHub"  
labels = {'x': 'Repository', 'y': 'Stars'}  
fig = px.bar(x=repo\_names, y=stars, title=title, labels=labels)  
  
❶ fig.update\_layout(title\_font\_size=28, xaxis\_title\_font\_size=20,  
 yaxis\_title\_font\_size=20)  
  
fig.show()

Primero añadimos un título y etiquetas para cada eje, como hicimos en los Capítulos 15 y 16. A continuación, utilizamos el método fig.update\_layout() para modificar elementos específicos del gráfico ❶. Plotly utiliza una convención en la que los aspectos de un elemento del gráfico están conectados por guiones bajos. A medida que te familiarices con la documentación de Plotly, empezarás a ver patrones consistentes en cómo se nombran y modifican los distintos elementos de un gráfico. Aquí fijamos el tamaño de la fuente del título en 28 y el tamaño de la fuente del título de cada eje en 20. El resultado se muestra en la [Figura 17-2](#figure17-2).

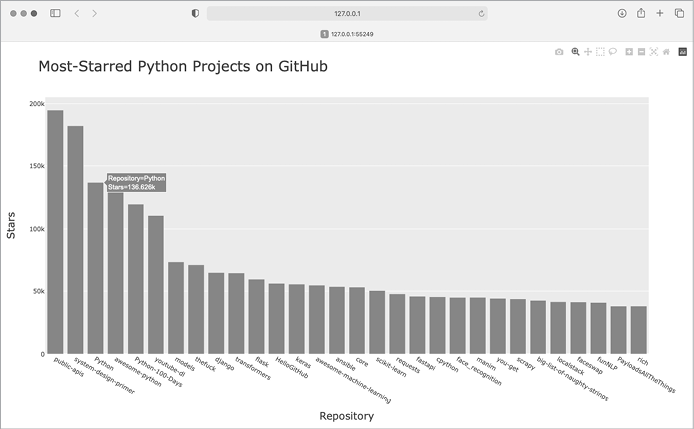


Figura 17-2: Se ha añadido un título al gráfico principal, y también a cada eje.

### Añadir información sobre herramientas personalizada

En Plotly, puedes situar el cursor sobre una barra individual para mostrar la información que representa. Esto se denomina comúnmente *tooltip*, y en este caso, actualmente muestra el número de estrellas que tiene un proyecto. Vamos a crear una información sobre herramientas personalizada para mostrar la descripción de cada proyecto, así como su propietario.

Necesitamos obtener algunos datos adicionales para generar la información sobre herramientas:

**python\_repos\_visual.py**

--snip--  
# Process repository information.  
repo\_dicts = response\_dict['items']  
❶ repo\_names, stars, hover\_texts = [], [], []  
for repo\_dict in repo\_dicts:  
 repo\_names.append(repo\_dict['name'])  
 stars.append(repo\_dict['stargazers\_count'])  
  
 # Build hover texts.  
❷ owner = repo\_dict['owner']['login']  
 description = repo\_dict['description']  
❸ hover\_text = f"{owner}<br />{description}"  
 hover\_texts.append(hover\_text)  
  
# Make visualization.  
title = "Most-Starred Python Projects on GitHub"  
labels = {'x': 'Repository', 'y': 'Stars'}  
❹ fig = px.bar(x=repo\_names, y=stars, title=title, labels=labels,  
 hover\_name=hover\_texts)  
  
fig.update\_layout(title\_font\_size=28, xaxis\_title\_font\_size=20,  
 yaxis\_title\_font\_size=20)  
  
fig.show()

Primero definimos una nueva lista vacía, hover\_texts, para contener el texto que queremos mostrar para cada proyecto ❶. En el bucle donde procesamos los datos, extraemos el propietario y la descripción de cada proyecto ❷. Plotly te permite utilizar código HTML dentro de los elementos de texto, así que generamos una cadena para la etiqueta con un salto de línea (<br />) entre el nombre de usuario del propietario del proyecto y la descripción ❸. A continuación, añadimos esta etiqueta a la lista hover\_texts.

En la llamada a px.bar(), añadimos el argumento hover\_name y le pasamos hover\_texts ❹. Este es el mismo enfoque que utilizamos para personalizar la etiqueta de cada punto en el mapa de la actividad sísmica mundial. A medida que Plotly crea cada barra, extraerá etiquetas de esta lista y sólo las mostrará cuando el espectador pase el ratón por encima de una barra. [La Figura 17-3](#figure17-3) muestra uno de estos tooltips personalizados.

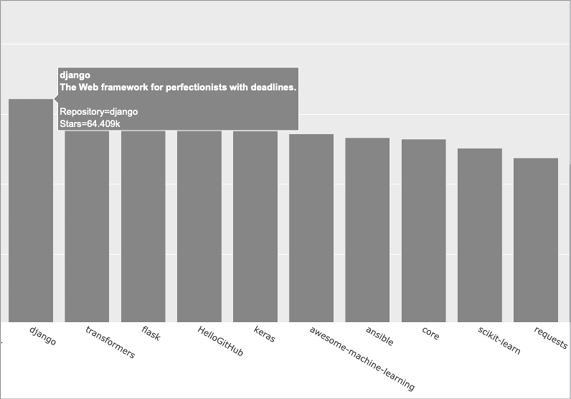


Figura 17-3: Al

pasar el ratón por encima de



una barra se muestra el propietario y la descripción del proyecto.

### Añadir enlaces clicables

Como Plotly te permite utilizar HTML en elementos de texto, podemos añadir fácilmente enlaces a un gráfico. Utilicemos las etiquetas del eje *x* para que el espectador pueda visitar la página de inicio de cualquier proyecto en GitHub. Tenemos que extraer las URL de los datos y utilizarlas al generar las etiquetas del eje *x*:

**python\_repos\_visual.py**

--snip--  
# Process repository information.  
repo\_dicts = response\_dict['items']  
❶ repo\_links, stars, hover\_texts = [], [], []  
for repo\_dict in repo\_dicts:  
 # Turn repo names into active links.  
 repo\_name = repo\_dict['name']  
❷ repo\_url = repo\_dict['html\_url']  
❸ repo\_link = f"<a href='{repo\_url}'>{repo\_name}</a>"  
 repo\_links.append(repo\_link)  
  
 stars.append(repo\_dict['stargazers\_count'])  
 --snip--  
  
# Make visualization.  
title = "Most-Starred Python Projects on GitHub"  
labels = {'x': 'Repository', 'y': 'Stars'}  
fig = px.bar(x=repo\_links, y=stars, title=title, labels=labels,  
 hover\_name=hover\_texts)  
  
fig.update\_layout(title\_font\_size=28, xaxis\_title\_font\_size=20,  
 yaxis\_title\_font\_size=20)  
  
fig.show()

Actualizamos el nombre de la lista que estamos creando de repo\_names a repo\_links para comunicar con mayor precisión el tipo de información que estamos reuniendo para el gráfico ❶. A continuación, extraemos la URL del proyecto de repo\_dict y la asignamos a la variable temporal repo\_url ❷. A continuación, generamos un enlace al proyecto ❸. Utilizamos la etiqueta de anclaje HTML, que tiene la forma <a href='URL'>link text</a>, para generar el enlace. A continuación, añadimos este enlace a repo\_links.

Cuando llamamos a px.bar(), utilizamos repo\_links para los valores *x* del gráfico. El resultado es el mismo que antes, pero ahora el espectador puede hacer clic en cualquiera de los nombres de proyecto de la parte inferior del gráfico para visitar la página de inicio de ese proyecto en GitHub. ¡Ahora tenemos una visualización interactiva e informativa de datos obtenidos a través de una API!

### Personalizar los colores de los marcadores

Una vez creado un gráfico, se puede personalizar casi cualquier aspecto del gráfico mediante un método de actualización. Ya hemos utilizado anteriormente el método update\_layout(). Otro método, update\_traces(), puede utilizarse para personalizar los datos que se representan en un gráfico.

Cambiemos las barras a un azul más oscuro, con cierta transparencia:

--snip--  
fig.update\_layout(title\_font\_size=28, xaxis\_title\_font\_size=20,  
 yaxis\_title\_font\_size=20)  
  
fig.update\_traces(marker\_color='SteelBlue', marker\_opacity=0.6)  
  
fig.show()

En Plotly, un *trace* se refiere a una colección de datos en un gráfico. El método update\_traces() puede recibir varios argumentos diferentes; cualquier argumento que empiece por marker\_ afecta a los marcadores del gráfico. Aquí establecemos el color de cada marcador en 'SteelBlue'; cualquier color CSS con nombre funcionará aquí. También establecemos la opacidad de cada marcador en 0.6. Una opacidad de 1,0 será totalmente opaca, y una opacidad de 0 será totalmente invisible.

### Más información sobre Plotly y la API de GitHub

La documentación de Plotly es extensa y está bien organizada; sin embargo, puede ser difícil saber por dónde empezar a leer. Un buen lugar para empezar es el artículo "Plotly Express en Python", en [https://plotly.com/python/plotly-express.](https://plotly.com/python/plotly-express) Es un resumen de todos los gráficos que puedes hacer con Plotly Express, y puedes encontrar enlaces a artículos más largos sobre cada tipo de gráfico.

Si quieres entender cómo personalizar mejor los gráficos de Plotly, el artículo "Estilizar gráficos de Plotly Express en Python" ampliará lo que has visto en los Capítulos15-17. Puedes encontrar este artículo en <https://plotly.com/python/styling-plotly-express>.

Para más información sobre la API de GitHub, consulta su documentación en [https://docs.github.com/en/rest.](https://docs.github.com/en/rest) Aquí aprenderás a extraer una amplia variedad de información de GitHub. Para ampliar lo que has visto en este proyecto, busca la sección Buscar de la referencia en la barra lateral. Si tienes una cuenta de GitHub, puedes trabajar con tus propios datos, así como con los datos disponibles públicamente de los repositorios de otros usuarios.

## La API de Hacker News

Para explorar cómo utilizar las llamadas a la API en otros sitios, echemos un vistazo rápido a Hacker News[(https://news.ycombinator.com)](https://news.ycombinator.com). En Hacker News, la gente comparte artículos sobre programación y tecnología y participa en animados debates sobre esos artículos. La API de Hacker News proporciona acceso a datos sobre todos los envíos y comentarios del sitio, y puedes utilizar la API sin tener que registrarte para obtener una clave.

La siguiente llamada devuelve información sobre el artículo principal actual en el momento de escribir este artículo:

https://hacker-news.firebaseio.com/v0/item/31353677.json

Cuando introduzcas esta URL en un navegador, verás que el texto de la página está encerrado entre llaves, lo que significa que es un diccionario. Pero la respuesta es difícil de examinar sin un formato mejor. Pasemos esta URL por el método json.dumps(), como hicimos en el proyecto terremoto del Capítulo 16, para poder explorar el tipo de información que se devuelve sobre un artículo:

**hn\_article.py**

import requests  
import json  
  
# Make an API call, and store the response.  
url = "https://hacker-news.firebaseio.com/v0/item/31353677.json"  
r = requests.get(url)  
print(f"Status code: {r.status\_code}")  
  
# Explore the structure of the data.  
response\_dict = r.json()  
response\_string = json.dumps(response\_dict, indent=4)  
❶ print(response\_string)

Todo en este programa debería resultarte familiar, porque lo hemos utilizado todo en los dos capítulos anteriores. La principal diferencia aquí es que podemos imprimir la cadena de respuesta formateada ❶ en lugar de escribirla en un archivo, porque la salida no es especialmente larga.

La salida es un diccionario de información sobre el artículo con el ID 31353677:

{  
 "by": "sohkamyung",  
❶ "descendants": 302,  
 "id": 31353677,  
❷ "kids": [  
 31354987,  
 31354235,  
 --snip--  
 ],  
 "score": 785,  
 "time": 1652361401,  
❸ "title": "Astronomers reveal first image of the black hole  
 at the heart of our galaxy",  
 "type": "story",  
❹ "url": "https://public.nrao.edu/news/.../"  
}

El diccionario contiene una serie de claves con las que podemos trabajar. La clave "descendants" nos indica el número de comentarios que ha recibido el artículo ❶. La clave "kids" proporciona los ID de todos los comentarios realizados directamente en respuesta a este envío ❷. Cada uno de estos comentarios puede tener también comentarios propios, por lo que el número de descendientes que tiene un envío suele ser mayor que su número de hijos. Podemos ver el título del artículo que se está discutiendo ❸ y también una URL del artículo que se está discutiendo ❹.

La siguiente URL devuelve una lista simple de todos los ID de los principales artículos actuales de Hacker News:

https://hacker-news.firebaseio.com/v0/topstories.json

Podemos utilizar esta llamada para averiguar qué artículos están en la página principal en este momento, y luego generar una serie de llamadas a la API similares a la que acabamos de examinar. Con este enfoque, podemos imprimir un resumen de todos los artículos que están en la portada de Hacker News en este momento:

**hn\_submissions.py**

from operator import itemgetter  
  
import requests  
  
# Make an API call and check the response.  
❶ url = "https://hacker-news.firebaseio.com/v0/topstories.json"  
r = requests.get(url)  
print(f"Status code: {r.status\_code}")  
  
# Process information about each submission.  
❷ submission\_ids = r.json()  
❸ submission\_dicts = []  
for submission\_id in submission\_ids[:5]:  
 # Make a new API call for each submission.  
❹ url = f"https://hacker-news.firebaseio.com/v0/item/{submission\_id}.json"  
 r = requests.get(url)  
 print(f"id: {submission\_id}\tstatus: {r.status\_code}")  
 response\_dict = r.json()  
  
 # Build a dictionary for each article.  
❺ submission\_dict = {  
 'title': response\_dict['title'],  
 'hn\_link': f"https://news.ycombinator.com/item?id={submission\_id}",  
 'comments': response\_dict['descendants'],  
 }  
❻ submission\_dicts.append(submission\_dict)  
  
❼ submission\_dicts = sorted(submission\_dicts, key=itemgetter('comments'),  
 reverse=True)  
  
❽ for submission\_dict in submission\_dicts:  
 print(f"\nTitle: {submission\_dict['title']}")  
 print(f"Discussion link: {submission\_dict['hn\_link']}")  
 print(f"Comments: {submission\_dict['comments']}")

Primero, hacemos una llamada a la API e imprimimos el estado de la respuesta ❶. Esta llamada a la API devuelve una lista que contiene los ID de hasta 500 de los artículos más populares de Hacker News en el momento en que se realiza la llamada. A continuación, convertimos el objeto respuesta en una lista Python ❷, que asignamos a submission\_ids. Utilizaremos estos ID para construir un conjunto de diccionarios, cada uno de los cuales contiene información sobre uno de los envíos actuales.

Creamos una lista vacía llamada submission\_dicts para almacenar estos diccionarios ❸. A continuación, recorremos en bucle los ID de las 30 mejores presentaciones. Hacemos una nueva llamada a la API para cada envío generando una URL que incluya el valor actual de submission\_id ❹. Imprimimos el estado de cada solicitud junto con su ID, para saber si se ha realizado correctamente.

A continuación, creamos un diccionario para el envío que se está procesando actualmente ❺. Almacenamos el título del envío, un enlace a la página de discusión de ese artículo y el número de comentarios que ha recibido hasta el momento. Después añadimos cada submission\_dict a la lista submission\_dicts ❻.

Cada envío en Hacker News se clasifica según una puntuación general basada en una serie de factores, entre ellos cuántas veces ha sido votado, cuántos comentarios ha recibido y lo reciente que es el envío. Queremos ordenar la lista de diccionarios por el número de comentarios. Para ello, utilizamos una función llamada itemgetter() ❼, que procede del módulo operator. Le pasamos a esta función la clave 'comments', y extrae el valor asociado a esa clave de cada diccionario de la lista. A continuación, la función sorted() utiliza este valor como base para ordenar la lista. Ordenamos la lista en orden inverso, para colocar primero las historias más comentadas.

Una vez ordenada la lista, la recorremos en bucle ❽ e imprimimos tres datos sobre cada uno de los envíos principales: el título, un enlace a la página de discusión y el número de comentarios que tiene actualmente el envío:

Status code: 200  
id: 31390506 status: 200  
id: 31389893 status: 200  
id: 31390742 status: 200  
--snip--  
  
Title: Fly.io: The reclaimer of Heroku's magic  
Discussion link: https://news.ycombinator.com/item?id=31390506  
Comments: 134  
  
Title: The weird Hewlett Packard FreeDOS option  
Discussion link: https://news.ycombinator.com/item?id=31389893  
Comments: 64  
  
Title: Modern JavaScript Tutorial  
Discussion link: https://news.ycombinator.com/item?id=31390742  
Comments: 20  
--snip--

Utilizarías un proceso similar para acceder a la información y analizarla con cualquier API. Con estos datos, podrías hacer una visualización que mostrara qué envíos han inspirado los debates recientes más activos. Esta es también la base de las aplicaciones que proporcionan una experiencia de lectura personalizada para sitios como Hacker News. Para saber más sobre a qué tipo de información puedes acceder a través de la API de Hacker News, visita la página de documentación en [https://github.com/HackerNews/API.](https://github.com/HackerNews/API)

## Nota

En ocasiones, Hacker News permite a las empresas a las que apoya realizar publicaciones especiales sobre contratación, y los comentarios están desactivados en estas publicaciones. Si ejecutas este programa mientras uno de estos posts está presente, obtendrás un KeyError. Si esto causa un problema, puedes envolver el código que construye submission\_dict en un bloque try-except y saltarte estos posts.

## Pruébalo tú mismo

17-1. Otros idiomas: Modifica la llamada a la API en *python\_repos.py* para que genere un gráfico que muestre los proyectos más populares en otros idiomas. Prueba con idiomas como *JavaScript*, *Ruby*, *C*, *Java*, *Perl*, *Haskell*, y *Go*.

17-2. Discusiones activas: Utilizando los datos de *hn\_submissions.py*, haz un gráfico de barras que muestre los debates más activos que se están produciendo actualmente en Hacker News. La altura de cada barra debe corresponder al número de comentarios que tiene cada envío. La etiqueta de cada barra debe incluir el título del envío y actuar como enlace a la página de discusión de ese envío. Si obtienes un KeyError al crear un gráfico, utiliza un bloque try-except para saltarte los envíos promocionales.

17-3.Probando python\_repos.py: En *python\_repos.py*, imprimimos el valor de status\_code para asegurarnos de que la llamada a la API se realizó correctamente. Escribe un programa llamado *test\_python\_repos.py* que utilice pytest para afirmar que el valor de status\_code es 200. Piensa en otras afirmaciones que puedas hacer: por ejemplo, que el número de elementos devueltos sea el esperado y que el número total de repositorios sea superior a una cantidad determinada.

17-4. Exploración adicional: Visita la documentación de Plotly y de la API de GitHub o de la API de Hacker News. Utiliza parte de la información que encuentres allí para personalizar el estilo de los gráficos que ya hemos hecho o para obtener información diferente y crear tus propias visualizaciones. Si tienes curiosidad por explorar otras API, echa un vistazo a las API mencionadas en el repositorio de GitHub en [https://github.com/public-apis.](https://github.com/public-apis)

## Resumen

En este capítulo has aprendido a utilizar las API para escribir programas autónomos que recopilan automáticamente los datos que necesitan y los utilizan para crear una visualización. Utilizaste la API de GitHub para explorar los proyectos Python con más estrellas en GitHub, y también echaste un breve vistazo a la API de Hacker News. Aprendiste a utilizar el paquete Peticiones para realizar automáticamente una llamada a la API y a procesar los resultados de esa llamada. También introdujimos algunos ajustes de Plotly que personalizan aún más el aspecto de los gráficos que generas.

En el próximo capítulo, utilizarás Django para construir una aplicación web como proyecto final.