17

WORKING WITH APIs

In this chapter, you'll learn how to write a self-contained program that generates a visualization based on data that it retrieves. Your program will use a web *application programming interface (API)* to automatically request specific information from a website—rather than entire pages—and then use that information to generate a visualization. Because programs written like this will always use current data to generate a visualization, even when that data might be rapidly changing, it will always be up to date.

Using a Web API

A web API is a part of a website designed to interact with programs. Those programs use very specific URLs to request certain information. This kind of request is called an *API call*. The requested data will be returned in an easily processed format, such as JSON or CSV. Most apps that rely on external data sources, such as apps that integrate with social media sites, rely on API calls.

Git and GitHub

We'll base our visualization on information from GitHub, a site that allows programmers to collaborate on coding projects. We'll use GitHub's API to request information about Python projects on the site, and then generate an interactive visualization of the relative popularity of these projects using Plotly.

GitHub (*https://github.com/*) takes its name from Git, a distributed version control system. Git helps people manage their work on a project, so changes made by one person won't interfere with changes other people are making. When you implement a new feature in a project, Git tracks the changes you make to each file. When your new code works, you *commit* the changes you've made, and Git records the new state of your project. If you make a mistake and want to revert your changes, you can easily return to any previously working state. (To learn more about version control using Git, see Appendix D.) Projects on GitHub are stored in *repositories*, which contain everything associated with the project: its code, information on its collaborators, any issues or bug reports, and so on.

When users on GitHub like a project, they can "star" it to show their support and keep track of projects they might want to use. In this chapter, we'll write a program to automatically download information about the most-starred Python projects on GitHub, and then we'll create an informative visualization of these projects.

Requesting Data Using an API Call

GitHub's API lets you request a wide range of information through API calls. To see what an API call looks like, enter the following into your browser's address bar and press ENTER:

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

This call returns the number of Python projects currently hosted on GitHub, as well as information about the most popular Python repositories. Let's examine the call. The first part, `https://api.github.com/`, directs the request to the part of GitHub that responds to API calls. The next part, `search/repositories`, tells the API to conduct a search through all repositories on GitHub.

The question mark after `repositories` signals that we're about to pass an argument. The `q` stands for *query*, and the equal sign (`=`) lets us begin spec-

ifying a query (`q=`). By using `language:python`, we indicate that we want information only on repositories that have Python as the primary language. The final part, `&sort=stars`, sorts the projects by the number of stars they've been given.

The following snippet shows the first few lines of the response.

```
 {
❶ "total_count": 3494012,
❷ "incomplete_results": false,
❸ "items": [
    {
     "id": 21289110,
     "node_id": "MDEwOlJlcG9zaXRvcnkyMTI4OTExMA==",
     "name": "awesome-python",
     "full_name": "vinta/awesome-python",
     --snip--
```

You can see from the response that this URL is not primarily intended to be entered by humans, because it's in a format that's meant to be processed by a program. GitHub found 3,494,012 Python projects as of this writing ❶. Because the value for `"incomplete_results"` is `false`, we know that the request was successful (it's not incomplete) ❷. If GitHub had been unable to fully process the API request, it would have returned `true` here. The `"items"` returned are displayed in the list that follows, which contains details about the most popular Python projects on GitHub ❸.

Installing Requests

The Requests package allows a Python program to easily request information from a website and examine the response. Use `pip` to install Requests:

```
$ python -m pip install --user requests
```

This line tells Python to run the `pip` module and install the Requests package to the current user's Python installation. If you use `python3` or a

different command when running programs or installing packages, make sure you use the same command here.

NOTE

If this command doesn't work on macOS, try running the command again without the `--user` flag.

Processing an API Response

Now we'll begin to write a program to automatically issue an API call and process the results by identifying the most starred Python projects on GitHub:

python_repos.py

```
❶ import requests

  # Make an API call and store the response.
❷ url = 'https://api.github.com/search/repositories?
q=language:python&sort=stars'
❸ headers = {'Accept': 'application/vnd.github.v3+json'}
❹ r = requests.get(url, headers=headers)
❺ print(f"Status code: {r.status_code}")
  # Store API response in a variable.
❻ response_dict = r.json()

  # Process results.
  print(response_dict.keys())
```

At ❶ we import the `requests` module. At ❷ we store the URL of the API call in the `url` variable. GitHub is currently on the third version of its API, so we define headers for the API call ❸ that ask explicitly to use this version of the API. Then we use `requests` to make the call to the API ❹.

We call `get()` and pass it the URL and the header that we defined, and we assign the response object to the variable `r`. The response object has an attribute called `status_code`, which tells us whether the request was successful. (A status code of 200 indicates a successful response.) At ❺ we print the value of `status_code` so we can make sure the call went through successfully.

The API returns the information in JSON format, so we use the `json()` method to convert the information to a Python dictionary ❻. We store the resulting dictionary in `response_dict`.

Finally, we print the keys from `response_dict` and see this output:

```
Status code: 200
dict_keys(['total_count', 'incomplete_results', 'items'])
```

Because the status code is 200, we know that the request was successful. The response dictionary contains only three keys: `'total_count'`, `'incomplete_results'`, and `'items'`. Let's take a look inside the response dictionary.

NOTE

Simple calls like this should return a complete set of results, so it's safe to ignore the value associated with `'incomplete_results'`. But when you're making more complex API calls, your program should check this value.

Working with the Response Dictionary

With the information from the API call stored as a dictionary, we can work with the data stored there. Let's generate some output that summarizes the information. This is a good way to make sure we received the information we expected and to start examining the information we're interested in:

python_repos.py

```
  import requests

  # Make an API call and store the response.
  --snip--

  # Store API response in a variable.
  response_dict = r.json()
❶ print(f"Total repositories: {response_dict['total_count']}")

  # 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)
```

At ❶ we print the value associated with `'total_count'`, which represents the total number of Python repositories on GitHub.

The value associated with `'items'` is a list containing a number of dictionaries, each of which contains data about an individual Python repository. At ❷ we store this list of dictionaries in `repo_dicts`. We then print the length of `repo_dicts` to see how many repositories we have information for.

To look closer at the information returned about each repository, we pull out the first item from `repo_dicts` and store it in `repo_dict` ❸. We then print the number of keys in the dictionary to see how much information we have ❹. At ❺ we print all the dictionary's keys to see what kind of information is included.

The results give us a clearer picture of the actual data:

```
Status code: 200
Total repositories: 3494030
Repositories returned: 30

❶ Keys: 73
  archive_url
  archived
  assignees_url
  --snip--
  url
  watchers
  watchers_count
```

GitHub's API returns a lot of information about each repository: there are 73 keys in `repo_dict` ❶. When you look through these keys, you'll get a sense of the kind of information you can extract about a project. (The only way to know what information is available through an API is to read the documentation or to examine the information through code, as we're doing here.)

Let's pull out the values for some of the keys in `repo_dict`:

python_repos.py

```
--snip--
# Explore information about the repositories.
repo_dicts = response_dict['items']
print(f"Repositories returned: {len(repo_dicts)}")

# 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']}")
```

Here, we print the values for a number of keys from the first repository's dictionary. At ❶ we print the name of the project. An entire dictionary represents the project's owner, so at ❷ we use the key `owner` to access the dictionary representing the owner, and then use the key `login` to get the owner's login name. At ❸ we print how many stars the project has earned and the URL for the project's GitHub repository. We then show when it was created ❹ and when it was last updated ❺. Finally, we print the repository's description; the output should look something like this:

```
Status code: 200
Total repositories: 3494032
Repositories returned: 30

Selected information about first repository:
Name: awesome-python
Owner: vinta
Stars: 61549
Repository: https://github.com/vinta/awesome-python
Created: 2014-06-27T21:00:06Z
Updated: 2019-02-17T04:30:00Z
Description: A curated list of awesome Python frameworks, libraries, soft-
ware
  and resources
```

We can see that the most-starred Python project on GitHub as of this writing is *awesome-python*, its owner is user *vinta*, and it has been starred by more than 60,000 GitHub users. We can see the URL for the project's repository, its creation date of June 2014, and that it was updated recently. Additionally, the description tells us that awesome-python contains a list of popular Python resources.

Summarizing the Top Repositories

When we make a visualization for this data, we'll want to include more than one repository. Let's write a loop to print selected information about each repository the API call returns so we can include them all in the visualization:

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']}")
```

We print an introductory message at ❶. At ❷ we loop through all the dictionaries in `repo_dicts`. Inside the loop, we print the name of each project, its owner, how many stars it has, its URL on GitHub, and the project's description, as shown here:

```
Status code: 200
Total repositories: 3494040
Repositories returned: 30

Selected information about each repository:

Name: awesome-python
Owner: vinta
Stars: 61549
```

```
Repository: https://github.com/vinta/awesome-python
Description: A curated list of awesome Python frameworks, libraries, soft-
ware
  and resources

Name: system-design-primer
Owner: donnemartin
Stars: 57256
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: python-patterns
Owner: faif
Stars: 19058
Repository: https://github.com/faif/python-patterns
Description: A collection of design patterns/idioms in Python
```

Some interesting projects appear in these results, and it might be worth looking at a few. But don't spend too much time, because shortly we'll create a visualization that will make the results much easier to read.

Monitoring API Rate Limits

Most APIs are rate limited, which means there's a limit to how many requests you can make in a certain amount of time. To see if you're approaching GitHub's limits, enter *https://api.github.com/rate_limit* into a web browser. You should see a response that begins like this:

```
{
  "resources": {
    "core": {
      "limit": 60,
      "remaining": 58,
      "reset": 1550385312
```

```
    },
❶   "search": {
❷     "limit": 10,
❸     "remaining": 8,
❹     "reset": 1550381772
    },
    --snip--
```

The information we're interested in is the rate limit for the search API ❶. We see at ❷ that the limit is 10 requests per minute and that we have 8 requests remaining for the current minute ❸. The reset value represents the time in *Unix* or *epoch time* (the number of seconds since midnight on January 1, 1970) when our quota will reset ❹. If you reach your quota, you'll get a short response that lets you know you've reached the API limit. If you reach the limit, just wait until your quota resets.

NOTE

Many APIs require you to register and obtain an API key to make API calls. As of this writing, GitHub has no such requirement, but if you obtain an API key, your limits will be much higher.

Visualizing Repositories Using Plotly

Let's make a visualization using the data we have now to show the relative popularity of Python projects on GitHub. We'll make an interactive bar chart: the height of each bar will represent the number of stars the project has acquired, and you can click the bar's label to go to that project's home on GitHub. Save a copy of the program we've been working on as *python_repos_visual.py*, and then modify it so it reads as follows:

python_repos_visual.py

```
import requests
```

```
❶ from plotly.graph_objs import Bar
  from plotly import offline

❷ # Make an API call and store the response.
  url = 'https://api.github.com/search/repositories?
q=language:python&sort=stars'
  headers = {'Accept': 'application/vnd.github.v3+json'}
  r = requests.get(url, headers=headers)
  print(f"Status code: {r.status_code}")

  # Process results.
  response_dict = r.json()
  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 data = [{
      'type': 'bar',
      'x': repo_names,
      'y': stars,
  }]
❺ my_layout = {
      'title': 'Most-Starred Python Projects on GitHub',
      'xaxis': {'title': 'Repository'},
      'yaxis': {'title': 'Stars'},
  }

  fig = {'data': data, 'layout': my_layout}
  offline.plot(fig, filename='python_repos.html')
```

We import the `Bar` class and the `offline` module from `plotly` ❶. We don't need to import the `Layout` class because we'll use the dictionary approach to define the layout, just as we did for the `data` list in the earthquake map-

ping project in Chapter 16. We continue to print the status of the API call response so we'll know if there is a problem ❷. We also remove some of the code that processes the API response, because we're no longer in the exploratory phase; we know we have the data we want.

We then create two empty lists ❸ to store the data we'll include in the initial chart. We'll need the name of each project to label the bars, and the number of stars to determine the height of the bars. In the loop, we append the name of each project and the number of stars it has to these lists.

Next, we define the `data` list ❹. This contains a dictionary, like we used in Chapter 16, which defines the type of the plot and provides the data for the x- and y-values. The x-values are the names of the projects, and the y-values are the number of stars each project has been given.

At ❺ we define the layout for this chart using the dictionary approach. Instead of making an instance of the `Layout` class, we build a dictionary with the layout specifications we want to use. We set a title for the overall chart, and we define a label for each axis.

Figure 17-1 shows the resulting chart. We can see that the first few projects are significantly more popular than the rest, but all of them are important projects in the Python ecosystem.

Figure 17-1: The most-starred Python projects on GitHub

Refining Plotly Charts

Let's refine the chart's styling. As you saw in Chapter 16, you can include all the styling directives as key-value pairs in the `data` and `my_layout` dictionaries.

Changes to the `data` object affect the bars. Here's a modified version of the `data` object for our chart that gives us a specific color and a clear border for each bar:

python_repos_visual.py

```
--snip--
data = [{
    'type': 'bar',
    'x': repo_names,
    'y': stars,
    'marker': {
        'color': 'rgb(60, 100, 150)',
        'line': {'width': 1.5, 'color': 'rgb(25, 25, 25)'}
    },
    'opacity': 0.6,
}]
--snip--
```

The `marker` settings shown here affect the design of the bars. We set a custom blue color for the bars and specify that they'll be outlined with a dark gray line that's 1.5 pixels wide. We also set the opacity of the bars to 0.6 to soften the appearance of the chart a little.

Next, we'll modify `my_layout`:

python_repos_visual.py

```
--snip--
my_layout = {
    'title': 'Most-Starred Python Projects on GitHub',
❶   'titlefont': {'size': 28},
❷   'xaxis': {
        'title': 'Repository',
        'titlefont': {'size': 24},
        'tickfont': {'size': 14},
    },
❸   'yaxis': {
        'title': 'Stars',
        'titlefont': {'size': 24},
        'tickfont': {'size': 14},
    },
}
--snip--
```

We use the `'titlefont'` key to define the font size of the overall chart title ❶. Within the `'xaxis'` dictionary, we add settings to control the font size of the x-axis title (`'titlefont'`) and also of the tick labels (`'tickfont'`) ❷. Because these are individual nested dictionaries, you can include keys for the color and font family of the axis titles and tick labels. At ❸ we define similar settings for the y-axis.

Figure 17-2 shows the restyled chart.

Figure 17-2: The styling for the chart has been refined.

Adding Custom Tooltips

In Plotly, you can hover the cursor over an individual bar to show the information that the bar represents. This is commonly called a *tooltip*, and in this case, it currently shows the number of stars a project has. Let's create a custom tooltip to show each project's description as well as the project's owner.

We need to pull some additional data to generate the tooltips and modify the `data` object:

python_repos_visual.py

```
--snip--
# Process results.
response_dict = r.json()
repo_dicts = response_dict['items']
❶ repo_names, stars, labels = [], [], []
for repo_dict in repo_dicts:
    repo_names.append(repo_dict['name'])
    stars.append(repo_dict['stargazers_count'])

❷    owner = repo_dict['owner']['login']
```

```
    description = repo_dict['description']
❸   label = f"{owner}<br />{description}"
    labels.append(label)

# Make visualization.
data = [{
    'type': 'bar',
    'x': repo_names,
    'y': stars,
❹   'hovertext': labels,
    'marker': {
        'color': 'rgb(60, 100, 150)',
        'line': {'width': 1.5, 'color': 'rgb(25, 25, 25)'}
    },
    'opacity': 0.6,
}]
--snip--
```

We first define a new empty list, `labels`, to hold the text we want to display for each project ❶. In the loop where we process the data, we pull the owner and the description for each project ❷. Plotly allows you to use HTML code within text elements, so we generate a string for the label with a line break (`<br />`) between the project owner's username and the description ❸. We then store this label in the list `labels`.

In the `data` dictionary, we add an entry with the key `'hovertext'` and assign it the list we just created ❹. As Plotly creates each bar, it will pull labels from this list and only display them when the viewer hovers over a bar.

Figure 17-3 shows the resulting chart.

Figure 17-3: Hovering over a bar shows the project's owner and description.

Adding Clickable Links to Our Graph

Because Plotly allows you to use HTML on text elements, we can easily add links to a chart. Let's use the x-axis labels as a way to let the viewer visit any project's home page on GitHub. We need to pull the URLs from the data and use them when generating the x-axis labels:

python_repos_visual.py

```
--snip--
# Process results.
response_dict = r.json()
repo_dicts = response_dict['items']
❶ repo_links, stars, labels = [], [], []
for repo_dict in repo_dicts:
    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.
  data = [{
      'type': 'bar',
❹     'x': repo_links,
      'y': stars,
      --snip--
  }]
  --snip--
```

We update the name of the list we're creating from `repo_names` to `repo_links` to more accurately communicate the kind of information we're putting together for the chart ❶. We then pull the URL for the project from `repo_dict` and assign it to the temporary variable `repo_url` ❷. At ❸ we generate a link to the project. We use the HTML anchor tag, which has the form `<a href='URL'>link text</a>`, to generate the link. We then append this link to the list `repo_links`.

At ❹ we use this list for the x-values in the chart. The result looks the same as before, but now the viewer can click any of the project names at the bottom of the chart to visit that project's home page on GitHub. Now we have an interactive, informative visualization of data retrieved through an API!

More About Plotly and the GitHub API

To read more about working with Plotly charts, there are two good places to start. You can find the *Plotly User Guide in Python* at *https://plot.ly/python/user-guide/*. This resource gives you a better understanding of how Plotly uses your data to construct a visualization and why it approaches defining data visualizations in this way.

The *python figure reference* at *https://plot.ly/python/reference/* lists all the settings you can use to configure Plotly visualizations. All the possible chart types are listed as well as all the attributes you can set for every configuration option.

For more about the GitHub API, refer to its documentation at *https://developer.github.com/v3/*. Here you'll learn how to pull a wide variety of specific information from GitHub. If you have a GitHub account, you can work with your own data as well as the publicly available data for other users' repositories.

The Hacker News API

To explore how to use API calls on other sites, let's take a quick look at Hacker News (*http://news.ycombinator.com/*). On Hacker News, people share articles about programming and technology, and engage in lively discussions about those articles. The Hacker News API provides access to data about all submissions and comments on the site, and you can use the API without having to register for a key.

The following call returns information about the current top article as of this writing:

```
https://hacker-news.firebaseio.com/v0/item/19155826.json
```

When you enter this URL in a browser, you'll see that the text on the page is enclosed by braces, meaning it's a dictionary. But the response is difficult to examine without some better formatting. Let's run this URL through the `json.dump()` method, like we did in the earthquake project in Chapter 16, so we can explore the kind of information that's returned about an article:

hn_article.py

```
import requests
import json

# Make an API call, and store the response.
url = 'https://hacker-news.firebaseio.com/v0/item/19155826.json'
r = requests.get(url)
print(f"Status code: {r.status_code}")
```

```
# Explore the structure of the data.
response_dict = r.json()
readable_file = 'data/readable_hn_data.json'
with open(readable_file, 'w') as f:
    json.dump(response_dict, f, indent=4)
```

Everything in this program should look familiar, because we've used it all in the previous two chapters. The output is a dictionary of information about the article with the ID 19155826:

readable_hn_data.json

```
  {
    "by": "jimktrains2",
❶   "descendants": 220,
    "id": 19155826,
❷   "kids": [
      19156572,
      19158857,
      --snip--
    ],
    "score": 722,
    "time": 1550085414,
❸   "title": "Nasa's Mars Rover Opportunity Concludes a 15-Year Mission",
    "type": "story",
❹   "url": "https://www.nytimes.com/.../mars-opportunity-rover-
dead.html"
  }
```

The dictionary contains a number of keys we can work with. The key `'descendants'` tells us the number of comments the article has received ❶. The key `'kids'` provides the IDs of all comments made directly in response to this submission ❷. Each of these comments might have comments of their own as well, so the number of descendants a submission has is usually greater than its number of kids. We can see the title of the article being discussed ❸, and a URL for the article that's being discussed as well ❹.

The following URL returns a simple list of all the IDs of the current top articles on Hacker News:

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

We can use this call to find out which articles are on the home page right now, and then generate a series of API calls similar to the one we just examined. With this approach, we can print a summary of all the articles on the front page of Hacker News at the moment:

hn_submissions.py

```
from operator import itemgetter

import requests

# Make an API call and store 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[:30]:
    # Make a separate 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"http://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']}")
```

First, we make an API call, and then print the status of the response ❶. This API call returns a list containing the IDs of up to the 500 most popular articles on Hacker News at the time the call is issued. We then convert the response object to a Python list at ❷, which we store in `submission_ids`. We'll use these IDs to build a set of dictionaries that each store information about one of the current submissions.

We set up an empty list called `submission_dicts` at ❸ to store these dictionaries. We then loop through the IDs of the top 30 submissions. We make a new API call for each submission by generating a URL that includes the current value of `submission_id` ❹. We print the status of each request along with its ID, so we can see whether it's successful.

At ❺ we create a dictionary for the submission currently being processed, where we store the title of the submission, a link to the discussion page for that item, and the number of comments the article has received so far. Then we append each `submission_dict` to the list `submission_dicts` ❻.

Each submission on Hacker News is ranked according to an overall score based on a number of factors including how many times it's been voted up, how many comments it's received, and how recent the submission is. We want to sort the list of dictionaries by the number of comments. To do this, we use a function called `itemgetter()` ❼, which comes

from the `operator` module. We pass this function the key `'comments'`, and it pulls the value associated with that key from each dictionary in the list. The `sorted()` function then uses this value as its basis for sorting the list. We sort the list in reverse order to place the most-commented stories first.

Once the list is sorted, we loop through the list at ❽ and print out three pieces of information about each of the top submissions: the title, a link to the discussion page, and the number of comments the submission currently has:

```
Status code: 200
id: 19155826    status: 200
id: 19180181    status: 200
id: 19181473    status: 200
--snip--

Title: Nasa's Mars Rover Opportunity Concludes a 15-Year Mission
Discussion link: http://news.ycombinator.com/item?id=19155826
Comments: 220

Title: Ask HN: Is it practical to create a software-controlled model rocket?
Discussion link: http://news.ycombinator.com/item?id=19180181
Comments: 72

Title: Making My Own USB Keyboard from Scratch
Discussion link: http://news.ycombinator.com/item?id=19181473
Comments: 62
--snip--
```

You would use a similar process to access and analyze information with any API. With this data, you could make a visualization showing which submissions have inspired the most active recent discussions. This is also the basis for apps that provide a customized reading experience for sites like Hacker News. To learn more about what kind of information

you can access through the Hacker News API, visit the documentation page at *https://github.com/HackerNews/API/*.

TRY IT YOURSELF

17-1. Other Languages: Modify the API call in *python_repos.py* so it generates a chart showing the most popular projects in other languages. Try languages such as *JavaScript*, *Ruby*, *C*, *Java*, *Perl*, *Haskell*, and *Go*.

17-2. Active Discussions: Using the data from *hn_submissions.py*, make a bar chart showing the most active discussions currently happening on Hacker News. The height of each bar should correspond to the number of comments each submission has. The label for each bar should include the submission's title and should act as a link to the discussion page for that submission.

17-3. Testing python_repos.py: In *python_repos.py*, we printed the value of `status_code` to make sure the API call was successful. Write a program called *test_python_repos.py* that uses `unittest` to assert that the value of `status_code` is 200. Figure out some other assertions you can make—for example, that the number of items returned is expected and that the total number of repositories is greater than a certain amount.

17-4. Further Exploration: Visit the documentation for Plotly and either the GitHub API or the Hacker News API. Use some of the information you find there to either customize the style of the plots we've already made or pull some different information and create your own visualizations.

Summary

In this chapter, you learned how to use APIs to write self-contained programs that automatically gather the data they need and use that data to create a visualization. You used the GitHub API to explore the most-starred Python projects on GitHub, and you also looked briefly at the Hacker News API. You learned how to use the Requests package to auto-

matically issue an API call to GitHub and how to process the results of that call. Some Plotly settings were also introduced that further customize the appearance of the charts you generate.

In the next chapter, you'll use Django to build a web application as your final project.