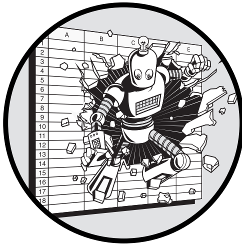
8

RETRIEVING DATA FROM THE INTERNET



The chapter focuses on APIs, which allow you to write scripts to retrieve data from external sources across the internet. We'll cover the basics of making API requests and processing the responses, and we'll consider some of the complexities that come up when working with APIs, like pagination, error handling, and authentication. In the end, we'll show how to write code for making API requests in clean, organized, and maintainable functions that you can use and reuse in whatever capacity you may need.

In Excel, your work is siloed; it lives only within the Excel application and the Microsoft applications that tie to it. Connections to the non-Microsoft world often don't exist, and that leaves you needing to manually bridge the gap. That might mean copying and pasting data from a website or clicking the Download button to extract data from an application. By contrast, APIs let you make these connections through code and bring external data right into your

workflow. And since Python allows you to automate these connections, APIs become one of the most essential tools for boosting productivity with code.

All kinds of data are retrievable through APIs, some of which you may not expect. Maybe you often send documents for e-signature; you can automate that with Python and the DocuSign API. Maybe you work in HR and use Workday; you can use the Workday API to automate all sorts of tasks. Or maybe you run a Shopify store selling T-shirts with relatable Excel-themed sayings like *CTRL+S and No Regrets*; you can keep track of your inventory and orders by using the Shopify API.

With an API, you can pull data from a vast array of external sources programmatically—that is, automatically from within a Python script. From there, you can pass the data along to a library like pandas for analysis. This allows you to leverage data from far and wide while streamlining your workflow, meaning you can do much more in much less time.

What Is an API?

An *application programming interface (API)* is a set of standards governing applications' interactions with one another. APIs describe how commands translate into actions, enabling different types of software to work together through code.

This definition is as nonspecific as it sounds. APIs can be any set of commands allowing pieces of software to exchange information with one another. There are lots of types of software that need to interact in lots of ways, so there are a lot of kinds of APIs.

APIs work by connecting a client and a server. The *client* is the entity that makes requests to access data or services from the API. This could be your Python script or any other application seeking information. The *server* is the entity that hosts the API, processes requests, and sends back the required data.

REST APIs

The primary API that we'll focus on in this chapter is the *Representational State Transfer (REST) API*. The name refers to the process of transferring representations of resources (data) between the client and the server. I say *representations* because the data the client receives may not be in the same format as it's stored on the server's side. The word *state* indicates that the request is for the data as it exists at a specific point in time. In the most direct terms, a REST API facilitates the transfer of a representation of a state.

The most salient advantage of REST APIs is their simple user experience. The interface that you, as the client, interact with, which defines how you structure requests and receive responses, is standardized. Therefore, once you know how to work with one REST API, applying those skills to other REST APIs is easy.

REST APIs also allow the API provider to entirely separate the client from all the complicated intermediate infrastructure that supports the API. This

means you ordinarily won't be able to tell whether you're connected directly to the end server or to an intermediary in the middle. This separation also allows the server to introduce multiple layers of infrastructure, such as load balancers, proxies, and gateways, to improve the API's performance. These additions can improve scalability, balance the workload more evenly, and enhance security without requiring the user to change anything on their side.

Other Types of APIs

REST is the most common type of API and serves the widest variety of applications and use cases, but several other relatively common types also exist:

Simple Object Access Protocol (SOAP) APIs Operate like REST APIs but provide more rigidity in protocol and messaging format, ensuring stricter standards and error handling. These are used in contexts like banking that require greater security and reliability.

WebSocket APIs Operate by opening a channel of communication and leaving it in place over a period of time. These are common for applications like real-time messaging or trading platforms and financial exchanges, where maintaining an open connection allows for instant data transmission and low latency.

GraphQL APIs Allow for more-complex querying than REST APIs. Developed by Meta (formerly Facebook), they're used when the user needs precise control over the data they fetch, like social media platforms with a wide variety of complex user data.

If you're not a software engineer, you likely won't encounter these other kinds of APIs as often in your everyday work, but it's helpful to know that they exist and their typical uses in case you do.

Making Requests and Handling Responses

Messages that go through REST APIs take the form of requests and responses, which are made over the internet via URLs. A client sends a *request* to a server, and the server returns a *response* with the requested data or an appropriate status message. These interactions typically involve one of a few standard methods established by the REST architecture.

REST APIs rely on *Hypertext Transfer Protocol (HTTP)*, a standard of communication used for transmitting web pages over the internet. You're familiar with this protocol if you've ever visited a website; it's the *http://* at the start of a URL. Making REST API requests also involves working with URLs, just like visiting web pages in your browser. To understand what that means exactly, it helps to talk a little about how the internet works.

Internet Communication Protocols

On the most basic level, the internet is just a network of connected computers around the world. Some of these computers are connected physically with

Python for Excel Users (Sample Chapter) © 7/30/2025 by Tracy Stephens

wires, and some are connected wirelessly, but since every computer in the network is connected somehow to at least one other computer in the network, it's possible to send a message between any two computers in the network no matter where they're located or what they're connected to. This system of connections and the rules that govern it is called the *Internet Protocol (IP)*. You might already be familiar with IP in the context of IP addresses, which are the identifiers devices use to locate each other on the network.

IP alone isn't enough to move data reliably between computers, since IP doesn't guarantee that information sent over the network will be delivered. It also doesn't ensure that information will arrive in any specific order, and it lacks any mechanism to adjust the flow of information when the network is congested. This is where the *Transmission Control Protocol (TCP)* comes into play.

TCP establishes a connection between the source and destination computers before data is sent over the internet. It systematically breaks information into smaller pieces, delivers each piece reliably and in the correct order, and applies checks to ensure everything works as intended. All in all, TCP adds necessary improvements on top of the IP network that make transmitting information through it more practical.

While TCP and IP work together to provide reliable data transfer between computers, neither specifies how data should be structured or interpreted. Enter HTTP. This protocol builds on top of TCP by providing a common language for clients and servers operating within the system. This common language standardizes how requests and responses should be formatted, how resources should be identified, and how data should be interpreted. Without HTTP, accessing and interacting with web content in a consistent and meaningful way would be impossible.

To explain these three protocols in simpler terms, if the World Wide Web were a restaurant, HTTP would be the menu, TCP would be the waitstaff, and IP would be the floor plan. HTTP defines what the clients (customers) can request (order) and how the server (kitchen) should respond. TCP handles the reliability and sequencing of the requests (dishes) being delivered. And IP lays out the routes and addresses, determining how to get from one computer (table) to another.

HTTPS

You'll notice that we use *https://* instead of *http://* at the start of the URL in all the example API requests in this chapter. *Hypertext Transfer Protocol Secure (HTTPS)* extends HTTP by adding extra security measures. Specifically, HTTPS encrypts the data exchanged between the client and the server. This encryption ensures that any sensitive data transmitted (such as API keys, login credentials, or personal information) is protected from any eavesdroppers who might attempt to listen in and that it can't be read if intercepted.

Request Types

Several types of requests can be transferred through a REST API. We'll focus on two of the most common: `GET` and `POST`. In "Simplifying Requests with the Requests Library" on page 169, we'll demonstrate how to make these requests through Python, but for these first few examples, we'll use `curl`, a simple tool for sending API requests directly from the command line. You don't need to do anything to install it, since `curl` is already built into both Windows and macOS, making it an excellent tool for practicing API interactions.

GET

A `GET` request retrieves data from a server. When you use a `GET` request, you're asking the server to send you information about a specific resource. When you visit a web page in your browser, the browser sends a `GET` request to the server to fetch the content of that page, which returns the necessary elements to display the page.

You can use `curl` in your console to make a `GET` request. As an example, we'll make a request to the Random User Generator API, an open source API that creates random user data for testing purposes:

```
curl -X GET "https://randomuser.me/api/?inc=name"
```

This request returns a long string of data including a randomly generated name and information about the response:

```
{
  "results": [
    {
      "name": {
          "title": "Mr",
          "first": "Paul",
          "last": "Allen"
      }
    }
  ],
  "info": {
    "seed": "abc123xyz789",
    "results": 1,
    "page": 1,
    "version": "1.4"
  }
}
```

In "Simplifying Requests with the Requests Library" on page 169, we'll discuss how to process this data in Python.

POST

A `POST` request sends data to a server. When you use a `POST` request, you're typically submitting data to the server for processing. For example, if you

were using an API to post a comment to a social media site, that would require a `POST` request. So would sending payment details to process a transaction or registering as a new user on a website. All these situations require sending data to the server, so `POST` is appropriate.

Say you want to send the following data about a cat to a server for storage in a database:

```
{
    "name": "whiskers",
    "age": 3,
    "color": "gray",
    "breed": "persian"
}
```

This data is in JSON format, which we'll discuss shortly. To illustrate how to send this data to an API, we'll use an API called JSONPlaceholder, a free service that allows you to test API calls. The `curl` command to send the cat data to JSONPlaceholder is shown here:

```
curl -X POST "https://jsonplaceholder.typicode.com/posts" \
    -H "Content-Type: application/json" \
    -d '{
        "name": "whiskers",
        "age": 3,
        "color": "gray",
        "breed": "persian"
    }'
```

This command uses the `POST` method to send the JSON data to the specified URL. The server processes the data and responds with a confirmation, including the data you sent along with a new ID assigned to it. The response would look something like this:

```
{
    "name": "whiskers",
    "age": 3,
    "color": "gray",
    "breed": "persian",
    "id": 101
}
```

This indicates that the server successfully received and processed the data.

Response Types

REST APIs give responses in plaintext, but that text can come in many formats. Often a REST API may let you choose the format for your response. Some common choices are CSV and JSON.

CSV Files

Comma-separated values (CSV) is a simple text format where each line corresponds to a single record, and each record consists of fields separated by commas. The first line typically contains headers that indicate the names of the columns. Here's an example of some data in CSV format:

```
name,age,color,breed
whiskers,3,gray,persian
mittens,2,black,siamese
tiger,4,orange,bengal
sparkline,7,brown tabby,domestic shorthair
```

The first line contains the headers: `name`, `age`, `color`, and `breed`. Each subsequent line contains the data for one cat, with the individual fields separated by commas.

JSON Files

JSON is a format for text-based data that uses key-value pairs. Each piece of data is designated with a key, and this key is paired with a value that represents the data. Like CSV, the JSON format is easy for humans to read and write and easy for machines to parse and generate.

JSON stands for *JavaScript Object Notation*; the format was derived from the way objects are defined in the JavaScript programming language. JSON itself is language agnostic, however, and doesn't have much to do with JavaScript specifically. The format can handle keys and values that are strings (with quotes); numbers (without quotes); arrays, which are ordered lists of values; or other, nested JSON objects containing their own key-value combinations. Here's the same cat data packaged into a JSON object:

```
{
  "metadata": {
    "total_cats": 4,
    "source": "example dataset"
  },
  "cats": [
    {
      "name": "whiskers",
      "age": 3,
      "color": "gray",
      "breed": "persian"
    },
    {
      "name": "mittens",
      "age": 2,
      "color": "black",
      "breed": "siamese"
    },
    {
```

```
      "name": "tiger",
      "age": 4,
      "color": "orange",
      "breed": "bengal"
    },
    {
      "name": "sparkline",
      "age": 2,
      "color": "brown tabby",
      "breed": "domestic shorthair"
    }
  ]
}
```

At the top level, this object consists of two keys, `metadata` and `cats`. The value for `metadata` is a nested object (enclosed in curly brackets) with two keys of its own, while the value for `cats` is an array (enclosed in square brackets) of objects, each representing an individual cat.

Although CSV responses are commonly available for REST APIs, they don't allow for as much flexibility in terms of what you can put in a response as JSON does. CSV responses are limited to tabular data, where each row has values for the same set of attributes (columns). They can't easily represent nested or hierarchical data. By contrast, JSON allows for each element in the response to have an entirely different set of attributes associated with it, and it can easily include complex nested structures.

JSON isn't a native data type in Python, like a list or a dictionary. To work with JSON data in your scripts, you need to use the functions in the `json` module. It's part of the Python Standard Library, so there's no need to separately install `json` if you already have Python installed. Some of the module's functions for working with JSON data include the following:

`json.load()` Reads JSON data from a file and parses it into a Python object, like a dictionary, list, or string

`json.loads()` Parses a JSON string (not from a file) into a Python object

`json.dump()` Writes a Python object as JSON data to a file

`json.dumps()` Converts a Python object to a JSON string without writing it to a file

As an example, to save the ages of our four cats as a JSON file, you can use the `json.dump()` function:

```
import json

cats = {
    "whiskers": 3,
    "mittens": 2,
    "tiger": 4,
    "sparkline": 2,
```

```
}

with open("cats.json", "w") as file:
    json.dump(cats, file, indent=4)
```

We declare the cats and their ages as a Python dictionary, whose key-value pairs already have a JSON-like structure. Then we write the data to a new file called *cats.json*. Notice that we call `json.dump()` inside a `with` block to ensure that the file is properly closed after writing, even if an error occurs.

To read that data back into a Python script from the JSON file, you can use the `json.load()` function:

```
with open("cats.json", "r") as file:
    cats = json.load(file)
```

This code reads the contents of the *cats.json* file and loads it into a Python dictionary named `cats`. Once again, notice that we use a `with` block to ensure that the file is properly managed.

Simplifying Requests with the Requests Library

To work with APIs programmatically from Python scripts, we'll use the Requests library. It includes functions that simplify the process of making requests over the internet through HTTP. If you didn't already install Requests in Chapter 1 to run our first sample script, use `pip` (or `pip3`) to install it now:

```
pip install requests
```

You'll need to import the library with `import requests` to use it in your scripts.

Making an API Request

Let's try making an API request with Python and the Requests library. We'll send a request to the Random User Generator API introduced earlier.

When working with a new API, it's always helpful to look at the API's documentation so you can understand the kind of data the API provides and how to get it. In this case, the Random User Generator documentation (*https://randomuser.me/documentation*) indicates that a generic `GET` request to the API will yield data for one random user in JSON format. Here's how to make that request:

```
import requests

url = "https://randomuser.me/api/"
response = requests.get(url)
```

We use the Requests library's `get()` method to make a `GET` request to the specified URL, storing the server's response in the `response` variable. The

`response` object includes a status code and the retrieved data. If you just print the variable with `print(response)`, you'll simply see the status code, like this:

```
<Response [200]>
```

The status code alone doesn't tell us much, but `response` includes within it properties that give more information. For example, `response.elapsed` gives the amount of time taken between sending the request and receiving the response from the server, and `response.text` gives the content of the response as a string.

To make the data useful, we can use the object's `json()` method to convert the JSON data into a Python dictionary:

```
result = response.json()
print(result)
```

This prints the JSON data retrieved from the API, which includes details about a randomly generated user, such as their name, location, email, and other attributes, as well as metadata about the request. The result is a nested dictionary, like this:

```
{
  "results": [
    {
      "gender": "male",
      "name": {
          "title": "Mr",
          "first": "Satya",
          "last": "Nadella"
      },
      "location": {
      --snip--
```

Since the data is now in dictionary format, it's much easier to use however you like in the rest of your Python script.

Adding Arguments

You can often customize API requests by adding arguments to the URL that alter the API's behavior or response. For example, the Random User Generator API takes various arguments to specify the number of users you want to generate, their nationality, and their gender, among other details.

The arguments are specified as key-value pairs at the end of the URL in the form *argument_name=value*. They're separated from the rest of the URL by a question mark. For example, to get multiple random users instead of one, you can add the `results` argument to the Random User Generator URL:

```
import requests

url = "https://randomuser.me/api/?results=5"
```

Python for Excel Users (Sample Chapter) © 7/30/2025 by Tracy Stephens

```
response = requests.get(url)
result = response.json()
print(lenresult)
```

This returns the number of users, which is five. You can also attach multiple arguments to a single request by separating the key-value pairs with ampersands (&). For example, here we generate three female users from the United Kingdom (`nat=gb`):

```
url = "https://randomuser.me/api/?results=3&gender=female&nat=gb"
response = requests.get(url)
result = response.json()
```

To find out what other arguments are accepted by any given API and the options available for each, see the API documentation.

Perhaps you can see how these API URLs can get out of hand as you customize your requests with more and more arguments. To make your code cleaner and more manageable, you can use the `params` argument of the `get()` method to separate the arguments from the URL. This allows you to put all your arguments in a Python dictionary and then let the Requests library do the work of tacking them onto the URL for you:

```
url = "https://randomuser.me/api/"
params = {"results": 3, "gender": "female", "nat": "gb"}

response = requests.get(url, params=params)
result = response.json()
```

This version of the request has exactly the same result as the previous version, where all the arguments were coded directly into the URL, but now the `params` dictionary contains the query arguments instead. These are automatically appended to the URL by the `get()` method when it sends the request. With this approach, it's easy to edit the code in the future to make different requests by simply adding, changing, or removing arguments from the `params` dictionary.

Building in Error Handling

API requests add a level of uncertainty to a Python script that you don't typically encounter when you're working with local files or in-memory data. This uncertainty stems from all the possible new kinds of errors that may arise. These errors may be entirely out of your control and can include internet connectivity issues, rate limits, unexpected response formats, invalid credentials, and more.

With this added room for potential problems, handling errors and validating results is a much more important factor when you're writing code that interacts with APIs. In this section, we'll look at Python's built-in mechanisms for handling errors generally, and then we'll see how to apply them to API calls.

Python for Excel Users (Sample Chapter) © 7/30/2025 by Tracy Stephens

try...except Blocks

In Python, you can explicitly tell your script how to handle errors by using `try...except` blocks. This technique allows you to manage and respond to errors as they occur during the execution of your program rather than letting them crash your program unexpectedly. With `try...except`, you define two possible paths for your code: the path you want to take assuming everything works and the path to take if an error occurs along the first path. The basic structure is as follows:

```
try:
    # Code that might raise an exception
except:
    # Code that says what to do if there is an exception
```

The `try` block contains the code that you want to execute, and the `except` block defines what to do in case of an error (called an *exception*) during the `try` block. If the `try` block code runs without any exceptions, the `except` block is skipped. However, if an error does occur within the `try` block, the rest of the `try` block is skipped and the code in the `except` block is executed. This is called *catching* the exception. If an error occurs in code that isn't contained within the `try` block, that error won't be caught by the `except` block and will propagate up, potentially crashing the program.

Many kinds of errors can arise in a Python program, each represented by a different kind of `Exception` object (we'll discuss some of these errors shortly). By default, an `except` block will catch all kinds of exceptions, but you can also specify a particular kind of exception to catch with the following syntax:

```
try:
    # Code that might raise an exception
except SomeException as e:
    # Code that says what to do if there is an exception
```

Here, *`SomeException`* represents the particular kind of exception you want the `except` block to handle, and `as e` makes the associated `Exception` object available within the `except` block as the `e` variable (usually for the purpose of printing out a custom error message). If the type of error encountered in the `try` block doesn't match the error specified by the `except` block, the error won't be caught, and again the program could crash.

It's good practice to explicitly define which error you think you might encounter and would like to handle with your `except` block, or to plan for multiple types of errors by adding multiple `except` blocks after the `try` block. If you don't, your code may hit a kind of error that you didn't anticipate without you noticing, and this could cause unexpected results. Here's an example:

```
try:
    value = int("abc")
except ValueError as e:
    print(f"ValueError occurred: {e}")
```

We try to convert the string `abc` into an integer, which obviously isn't possible, so the code will run into a `ValueError`. When you run this, the interpreter executes the code inside the `except` block and prints the following to the console:

```
ValueError occurred: invalid literal for int() with base 10: 'abc'
```

By specifying this error in particular, we handle only that specific exception. This approach makes the code more predictable and easier to debug than using a "naked" `except` and handling all errors generally. If you catch all exceptions without distinction, you might inadvertently catch exceptions you didn't intend to handle, masking other issues in your code.

Common Errors

As mentioned earlier, you may encounter many kinds of errors in Python. Some are built into the language, while others come from libraries and external sources. Python's built-in errors are automatically recognized by the interpreter, regardless of which version of the language you've installed or which libraries you've imported. You can see a full list of Python's built-in errors in the Python documentation at *https://docs.python.org/3/library/exceptions.html*.

We've already seen an example of `ValueError`, which occurs when a function receives an argument with an inappropriate value. Another built-in error is `KeyError`, which occurs when you attempt to access a key that doesn't exist in a dictionary. Because working with APIs in Python typically involves dictionaries, `KeyError` tends to come up often. Here's an example:

```
my_dict = {"a": 7, "b": 5}
try:
    print(my_dict["c"])
except KeyError:
    my_dict["c"] = 1
    print("Added the value for c.")
```

We try to access the nonexistent key `c` in the dictionary, causing a `KeyError`. The `except` block catches the error and adds a new key-value pair to the dictionary, and a message is printed.

Python for Excel Users (Sample Chapter) © 7/30/2025 by Tracy Stephens

Several errors that are unique to the Requests library often get triggered when working with APIs. These are found in `requests.exceptions`, which is a submodule of the library:

`ConnectionError` A network problem exists, like a faulty internet connection.

`Timeout` The server takes too long to respond.

`HTTPError` HTTP request returns an unsuccessful status code like 404.

`InvalidURL` provided URL is malformed or contains invalid characters.

Another common error, called `JSONDecodeError`, comes from the `json` module and occurs when decoding a JSON response fails because the data isn't valid JSON.

Here's an example of how to handle `ConnectionError` when making an API request. To see this in action, try running the code with your Wi-Fi turned off:

```
import requests

try:
    url = "https://randomuser.me/api/"
    response = requests.get(url)
    result = response.json()
    print(result)
except requests.exceptions.ConnectionError:
    print("Connection error: Please check your internet connection.")
```

If there's no internet connection, calling the `get()` method within the `try` block will trigger a connection error. Since the `except` block is designed to handle this type of error (`requests.exceptions.ConnectionError`), the exception is caught instead of allowing the program to crash. The `except` block then prints a message to check the internet connection.

Breaking Large Requests into Smaller Parts

APIs often place limits on the size of requests and responses, or on the number of requests you can send within a certain time period, so they aren't inundated and can maintain consistent performance. One way to work within these limits, or to just make a response more feasible to process, is *pagination*. In this technique, you divide a request into smaller, more manageable requests called *batches*. This breaks the desired dataset into individual chunks that can be reconstituted on the receiving end, just as the text of a long novel is divided across the individual pages of a book.

As an example, the Random User Generator API sets an upper limit of 5,000 users per request. To retrieve a larger number of users (say, 10,050), we'll need to paginate through the API. The common pattern is to use a

while loop to repeatedly make smaller requests until the desired number is reached, like so:

```
import requests

url = "https://randomuser.me/api/"

total_users = 10050
users_per_request = 5000
retrieved_users = []

while len(retrieved_users) < total_users:
    remaining_users = total_users - len(retrieved_users)
    current_batch_size = min(users_per_request, remaining_users)

    params = {"results": current_batch_size}

    response = requests.get(url, params=params)
    if "error" in response.json():
        print(response.json())
        break

    data = response.json()
    retrieved_users.extend(data["results"])
```

We define three variables to help manage the pagination: `total_users` is the total number of users we'd like to retrieve, `users_per_request` is the batch size, and `retrieved_users` is an empty list for accumulating the results as we receive them. The condition for the `while` loop, which needs to be `True` at the start of each repetition for the loop to keep iterating, is `len(retrieved_users) < total_users`, meaning that we'll start a new iteration only if we haven't retrieved enough users yet. Within the `while` loop, we identify the number of users left to retrieve (`remaining_users`) and the number of results we need to ask the API for this time (the lesser of `users_per_request` or `remaining_users`). Then we make the request and append the results to the `retrieved_users` list. Just in case any of our requests results in an error from the API, we add an `if` statement to print the error.

Using a more straightforward `for` loop instead of a `while` loop would be technically possible in this example, since we know the exact number of users that will be included in each API response. However, often you don't know how many acceptable results will be retrieved from the API and therefore how many iterations you'll need to make. In those cases, a `while` loop is your only option, since it doesn't require you to specify the number of iterations you'll be making up front.

For example, say we'd like to retrieve 20 users who specifically live in Texas. The Random User Generator API doesn't offer an argument to request a particular state, only a country, so there's no way to know how many

users we'll get from Texas in any given set of results. Therefore, we'll have to use `while` to make an indefinite number of API requests and filter the results on our end until we've accumulated 20 users from Texas:

```
url = "https://randomuser.me/api/"

total_users = 20
users_per_request = 5000
retrieved_users = []

while len(retrieved_users) < total_users:
    remaining_users = total_users - len(retrieved_users)
    current_batch_size = min(users_per_request, remaining_users)

    params = {"results": current_batch_size, "nat": "us"}

    response = requests.get(url, params=params)
    if "error" in response.json():
        print(response.json())
        break

    data = response.json()
    users = data["results"]

    texas_users = [user for user in users if user["location"]["state"].lower() == "texas"] ❶
    retrieved_users.extend(texas_users)
```

This `while` loop will continue to fetch users in batches of `5000` until the total number of users from Texas reaches `total_users`, which is `20`. Each time through the loop, we use a list comprehension to keep only the users from the correct state ❶. The total number of iterations the code makes will depend on how many users from Texas randomly appear in the results of each request. Once the target number of users from Texas is reached, the `while` loop stops iterating, and the code moves on.

Handling Authentication

Many APIs, even publicly accessible ones, require users to authenticate before interacting with them. Authentication allows the API to identify who's making requests, what requests they're making, and how often they're making them. With this information, the API provider can impose restrictions on who can submit and receive data, as well as how much data they can process. Authentication also lets providers customize the data and services delivered to each user, and log and audit access to meet legal and regulatory requirements.

APIs use two primary mechanisms to authenticate users: API keys and Open Authorization (OAuth). Let's look at what's involved in each.

API Keys

An *API key* is a unique identifier, typically a random alphanumeric string, used to authenticate a client making requests to an API. You obtain a key by registering with an API, and then you include that key when you make requests to that API. This security measure lets API providers know who's accessing their API, how often they're doing it, and what information they can and can't receive.

API keys are sensitive information. An API key getting into the hands of a bad actor, or just someone who doesn't know what they're doing, opens the door for all sorts of nefarious activity. Depending on the API's purpose, this could mean significant financial costs or data getting into the wrong hands. At the very least, a compromised API key can mean rate limits are hit faster and more often, so it's important to keep API keys in a place that's out of view from everyone else.

A place that's *not* out of view from everyone else is in your Python scripts. If your scripts are version controlled (and they should be), they'll be tracked and likely saved in repositories that multiple people can access. Even if a repository is private, or your scripts aren't version controlled at all, they can still live on; you never know when files might be shared or content copied and pasted into places you don't intend it to be. A much safer approach is to store your API keys in environment variables.

Storing Keys in Environment Variables

The environment variables stored locally on your computer are out of view from everyone else. They meet the two necessary qualifications for safely storing an API key:

- They aren't stored within your Python scripts.
- They can be easily accessed by your Python script when it's executed.

To illustrate how to authenticate with an API key stored in an environment variable, we'll use a REST API that provides weather data maintained by a company called OpenWeather. The API provides weather data and uses API keys for authentication. To start, go to *https://openweathermap.org/api* and sign up to create an API key. There's no cost for signing up and creating a key, but the free tier limits you to 1,000 calls per day. That should be plenty for practice.

Once you've received your API key, save it as an environment variable called `WEATHER_API_KEY`. On macOS, add the following line to your shell configuration file:

```
export WEATHER_API_KEY=your_api_key_here
```

On Windows, use the Environment Variables section of Advanced System Settings to create the new environment variable. To verify that your API key has been stored, run **`echo $WEATHER_API_KEY`** (macOS) or **`echo %WEATHER_API_KEY%`** (Windows) at the command line. (See Chapter 1 for a more detailed review of how to set environment variables for your operating system.)

Python for Excel Users (Sample Chapter) © 7/30/2025 by Tracy Stephens

Now that you've saved your key as an environment variable, you'll need to retrieve it from within your Python script. To do that, use the `getenv()` function from the `os` module, which is part of the Python Standard Library:

```
import os

api_key = os.getenv("WEATHER_API_KEY")
```

This makes your API key available in your Python script as the `api_key` variable but only when the script is run on your local computer where the environment variable is stored.

Making Requests with API Keys

With most REST APIs that use keys for authentication, you can apply those keys to requests in two ways. One is to include your key as an argument in the URL, just like any other argument. The second way is to add your key to the header of the HTTP request. *Headers* are key-value pairs that provide metadata or additional context or information about the request.

For the OpenWeatherMap API, both techniques are an option, so we'll try both. First, here's a script that authenticates with the API by including the key as an argument in the URL:

```
import os
import requests

api_key = os.getenv('WEATHER_API_KEY')

url = "https://api.openweathermap.org/data/2.5/weather"
params = {"lat": 47.6458, "lon": 122.1320, "appid": api_key}

response = requests.get(url, params=params)
result = response.json()
humidity = result["main"]["humidity"]
```

We build a request for current weather data with three arguments: `lat` and `lon`, which set the latitude and longitude of the desired location, and `appid` for the API key, as specified in the API's documentation. Then we attach those arguments to the request URL by using `get()`. After that, we store the JSON version of the response into a new variable called `result`. From there, we can use any part of the response that we like. In this example, we extracted the humidity by running `result["main"]["humidity"]`.

To see the results, simply run the following:

```
print(humidity)
```

This prints the relative humidity at the specified location that was retrieved from the API.

This approach is straightforward but opens up security concerns. If your API key is included in the URL, it can be logged by any server handling your

Python for Excel Users (Sample Chapter) © 7/30/2025 by Tracy Stephens

request, including your own server, proxy servers, and the API provider's server. Anyone with access to those servers can view the logs and discover your key. URLs are also often shared in error messages that include debugging output, and anyone who sees that information could access the keys. Because of these security concerns, and in order to keep your code more organized, it's generally a better practice to send your API key as part of the HTTP header rather than in the URL, as shown here:

```
api_key = os.getenv('WEATHER_API_KEY')

url = "https://api.openweathermap.org/data/2.5/weather"
headers = {"x-api-key": api_key}
params = {"lat": 47.6458, "lon": 122.1320}

response = requests.get(url, headers=headers, params=params)
result = response.json()
humidity = result["main"]["humidity"]
```

In this example, the API key is paired with the `x-api-key` header, which is added to the request by using the `headers` argument of the `get()` function.

OAuth

Instead of API keys, many major APIs that you might encounter in your day-to-day work, like GitHub, LinkedIn, and Slack, use *Open Authorization (OAuth)*. This authorization standard allows third-party applications to grant limited access to a user without exposing their credentials. OAuth provides a secure way for users to authorize applications (like Python scripts) to interact with APIs on the user's behalf.

In simple terms, OAuth provides a mechanism that allows you to input credentials on the server side in order to receive a temporary key that you can store on the client side. Here's a general overview of the typical steps involved:

1. Register with an API provider and establish a client ID and password.
2. Visit a server-side authorization URL to authenticate. Here, you can log in with your credentials and grant permission for your Python script to send requests.
3. After server-side authentication is complete, the authorization server automatically redirects back to the application you're looking to authenticate with an authorization code.
4. Exchange the authorization code for an access token by making a `POST` request to the API.
5. Use the temporary access key to make an authorized `GET` request to the API from your Python script.

The exact process, and the Python code you'll need to complete the process, will differ by API. This may seem like a daunting and complex task, but

major API providers typically have up-to-date documentation that details exactly how the authentication process works.

Streamlining Your Requests with Reusable Functions

The code to run API calls can quickly become complex as you apply features like additional arguments, error handling, pagination, and authentication. By encapsulating the code to make requests into functions, you can separate all that complexity from the bread and butter of your script—specifying the data you need and retrieving it. In this section, we'll revisit the Random User Generator and OpenWeather APIs to illustrate two examples of functionalizing code to make API requests.

First, let's create a function to fetch a list of random users from the Random User Generator API:

```
import requests

def fetch_random_users(total_users, params={}):
    url = "https://randomuser.me/api/"
    users_per_request = 5000

    retrieved_users = []
    while len(retrieved_users) < total_users:
        remaining_users = total_users - len(retrieved_users)
        current_batch_size = min(users_per_request, remaining_users)
        params["results"] = current_batch_size
        try:
            response = requests.get(url, params=params)
            data = response.json()
            users = data["results"]
            retrieved_users.extend(users)
        except requests.exceptions.HTTPError as http_err:
            print(f"HTTP error occurred: {http_err}")
            break
        except requests.exceptions.RequestException as req_err:
            print(f"Request error occurred: {req_err}")
            break
        except ValueError as json_err:
            print(f"JSON decode error occurred: {json_err}")
            break

    return retrieved_users
```

This `fetch_random_users()` function takes in two arguments: one that's required, called `total_users`, that specifies the number of users you'd like to retrieve, and one that's optional, called `params`, for any additional request arguments. All the API-specific information like `url` and `users_per_request` is contained within the function, meaning it's abstracted away from the code to

call the function. The while loop for pagination and the try...except blocks for error handling are also encapsulated within the function, so all this complex functionality will unfold seamlessly when the function is called. If one of the errors covered by the except blocks is caught, a statement to that effect will be printed in the console.

To call the function, you need to define the values you'd like to use as the total_users and params arguments:

```
total_users = 105
params = {"nat": "us", "gender": "female"}

all_users = fetch_random_users(total_users, params=params)
```

After running this code, all_users should contain a list of users outputted from the API request. Notice how concise this main script is now that all the nitty-gritty code for making the request is delegated to the fetch_random_users() function.

Next, we'll functionalize the code for fetching weather forecast data from the OpenWeather API:

```
import os
import requests

def fetch_weather_forecast(params={}):
    url = "https://api.openweathermap.org/data/2.5/forecast"
    api_key = os.getenv("WEATHER_API_KEY")

    headers = {"x-api-key": api_key}

    try:
        response = requests.get(url, headers=headers, params=params)
        data = response.json()
        forecasts = data["list"]  # Assume "list" contains the forecast data
        return forecasts
    except requests.exceptions.HTTPError as http_err:
        print(f"HTTP error occurred: {http_err}")
    except requests.exceptions.RequestException as req_err:
        print(f"Request error occurred: {req_err}")
    except ValueError as json_err:
        print(f"JSON decode error occurred: {json_err}")
    return []
```

Similar to fetch_random_users(), this fetch_weather_forecast() function encapsulates all the logic for making an OpenWeather API request and handling potential errors. This includes the os.getenv() function to retrieve the API key from the environment variable. The function uses two return statements, one in the try block to return the retrieved weather data if the request was a success, and one at the end to return an empty list if the request failed and one of the except blocks was triggered.

To call the function, define a dictionary containing the API request arguments and pass it through the `params` argument of `fetch_weather_forecast()`:

```
params = {"lat": 47.6458, "lon": 122.1320, "cnt": 50, "units": "metric"}

all_forecasts = fetch_weather_forecast(params=params)
```

If all goes well, `all_forecasts` should contain the weather forecast data from the OpenWeather API.

To make these two functions easy to find, reuse, and maintain, let's create a Python file called *api_utils.py* and place both function definitions within it. This will enable us to easily import and use these functions in other scripts:

```
import os
import requests

def fetch_random_users(params={}):
    # Code inside the fetch_random_users function

def fetch_weather_forecast(params={}):
    # Code inside the fetch_weather_forecast function
```

Don't forget to include an *__init__.py* file in the same directory as *api_utils.py* if you plan to import the functions to a file in another directory or if the directory containing the functions isn't included in the PATH environment variable.

To use these functions in another script, you can import them and call them from within that script's main block, as we discussed in Chapter 3. For example, you may have *random_users.py* that imports `fetch_random_users()`, makes a request, and saves the results to a JSON file:

```
from api_utils import fetch_random_users
import json

if __name__ == "__main__":
    total_users = 105
    params = {"nat": "us", "gender": "female"}

    all_users = fetch_random_users(total_users, params=params)
    with open("users.json", "w") as f:
        json.dump(all_users, f)
```

Python for Excel Users (Sample Chapter) © 7/30/2025 by Tracy Stephens

Similarly, you may also have a *weather_forecast.py* file that saves some weather data to a JSON file:

```
from api_utils import fetch_weather_forecast
import json

if __name__ == "__main__":
    params = {"lat": 47.6458, "lon": 122.1320, "cnt": 50, "units": "metric"}

    all_forecasts = fetch_weather_forecast(params=params)
    with open("forecast.json", "w") as f:
        json.dump(all_forecasts, f)
```

Packaging your API request code into reusable functions like this provides several advantages. First, it makes your code easier to maintain. You can import these functions into as many other scripts as you like, but anytime you fix an error or add another feature to the function, you'll need to make the change in only the original *api_utils.py*. This approach also makes the *random_users.py* and *weather_forecast.py* files much shorter and easier to read and understand. If you want to change the arguments of either API call, locating and updating the `params` argument is quick and easy. Without functionalizing the API request part of the code, the code would be far more cumbersome and chaotic.

Conclusion

This chapter covered the basics and benefits of working with REST APIs. We explored the architecture of a REST API request, including the role of HTTP and its integration into the broader structure of the internet. We discussed the essential components of a robust `GET` request using the Requests library, such as adding arguments, handling errors, implementing pagination, and managing authentication. Finally, we demonstrated how to functionalize these requests to promote simple, readable, and maintainable code.

Knowing how to retrieve data from APIs will open up opportunities for automating the work you currently do in Excel. With simple Python scripts like the ones we've explored in this chapter, you can eliminate much of the manual data collection that an Excel-based workflow tends to rely on. Once you have that data in hand, you can analyze, visualize, or process it—whether in Excel or with the Python-based analytical tools covered in this book.