How to Download Files From URLs With Python

by Claudia Ng Jul 31, 2023 1 Comment basics

Mark as Completed

Share Share Email

Table of Contents

When it comes to file retrieval, Python offers a robust set of tools and packages that are useful in a variety of applications, from web scraping to automating scripts and analyzing retrieved data. Downloading files from a URL programmatically is a to learn for various programming and data projects and workflows.

In this tutorial, you'll learn how to:

- **Download** files from the Web using the standard library as well as third-party libraries in Python
- **Stream data** to download large files in manageable chunks
- Implement **parallel** downloads using a pool of threads
- Perform **asynchronous** downloads to fetch multiple files in bulk

In this tutorial, you'll be downloading a range of economic data from the World Bank Open Data platform. To get started on this example project, go ahead and grab the sample code below:

> **Free Bonus: Click here to download your sample code** for downloading files from the Web with Python.

Facilitating File Downloads With Python

While it's possible to download files from URLs using traditional command-line tools, Python provides several libraries that facilitate file retrieval. Using Python to download files offers several advantages.

One advantage is **flexibility**, as Python has a rich ecosystem of libraries, including ones that offer efficient ways to handle different file formats, protocols, and authentication methods. You can choose the most suitable Python tools to accomplish the task at hand and fulfill your specific requirements, whether you're downloading from a plain-text CSV file or a complex binary file.

Another reason is **portability**. You may encounter situations where you're working on cross-platform applications. In such cases, using Python is a good choice because it's a cross-platform programming language. This means that Python code can run consistently across different operating systems, such as Windows, Linux, and macOS.

Using Python also offers the possibility of **automating your processes**, saving you time and effort. Some examples include automating retries if a download fails, retrieving and saving multiple files from URLs, and processing and storing your data in designated locations.

These are just a few reasons why downloading files using Python is better than using traditional command-line tools. Depending on your project requirements, you can choose the approach and library that best suits your needs. In this tutorial, you'll learn approaches to some common scenarios requiring file retrievals.

Downloading a File From a URL in Python

In this section, you'll learn the basics of downloading a ZIP file containing gross domestic product (GDP) data from the World Bank Open Data platform. You'll use two common tools in Python, `urllib` and `requests`, to download **GDP by country**.

While the `urllib` package comes with Python in its standard library, it has some limitations. So, you'll also learn to use a popular third-party library, `requests`, that offers more features for making HTTP requests. Later in the tutorial, you'll see additional functionalities and use cases.

Using `urllib` From the Standard Library

Python ships with a package called `urllib`, which provides a convenient way to interact with web resources. It has a straightforward and user-friendly interface, making it suitable for quick prototyping and smaller projects. With `urllib`, you can perform different tasks dealing with network communication, such as parsing URLs, sending HTTP requests, downloading files, and handling errors related to network operations.

As a standard library package, `urllib` has no external dependencies and doesn't require installing additional packages, making it a convenient choice. For the same reason, it's readily accessible for development and deployment. It's also cross-platform compatible, meaning you can write and run code seamlessly using the `urllib` package across different operating systems without additional dependencies or configuration.

The `urllib` package is also very versatile. It integrates well with other modules in the Python standard library, such as `re` for building and manipulating regular expressions, as well as `json` for working with JSON data. The latter is particularly handy when you need to consume JSON APIs.

In addition, you can extend the `urllib` package and use it with other third-party libraries, like `requests`, `BeautifulSoup`, and `Scrapy`. This offers the possibility for more advanced operations in web scraping and interacting with web APIs.

To download a file from a URL using the `urllib` package, you can call `urlretrieve()` from the `urllib.request` module. This function fetches a web resource from the specified URL and then saves the response to a local file. To start, import `urlretrieve()` from `urlllib.request`:

Python

```
>>> from urllib.request import urlretrieve
```

Next, define the URL that you want to retrieve data from. If you don't specify a path to a local file where you want to save the data, then the function will create a temporary file for you. Since you know that you'll be downloading a ZIP file from that URL, go ahead and provide an optional path to the target file:

Python

```
>>> url = (
...     "https://api.worldbank.org/v2/en/indicator/"
...     "NY.GDP.MKTP.CD?downloadformat=csv"
... )
>>> filename = "gdp_by_country.zip"
```

Because your URL is quite long, you rely on Python's **implicit concatenation** by splitting the string literal over multiple lines inside parentheses. The Python interpreter will automatically join the separate strings on different lines into a single string. You also define the location where you wish to save the file. When you only provide a filename without a path, Python will save the resulting file in your current working directory.

Then, you can download and save the file by calling `urlretrieve()` and passing in the URL and optionally your filename:

Python

```
>>> urlretrieve(url, filename)
('gdp_by_country.zip', <http.client.HTTPMessage object at 0x7f06ee7527d0>)
```

The function returns a tuple of two objects: the path to your output file and an HTTP message object. When you don't specify a custom filename, then you'll see a path to a temporary file that might look like this: `/tmp/tmps7qjl1tj`. The `HTTPMessage` object represents the HTTP headers returned by the server for the request, which can contain information like content type, content length, and other metadata.

You can unpack the tuple into the individual variables using an assignment statement and iterate over the headers as though they were a Python dictionary:

Python

```
>>> path, headers = urlretrieve(url, filename)
>>> for name, value in headers.items():
...     print(name, value)
...
Date Wed, 28 Jun 2023 11:26:18 GMT
Content-Type application/zip
Content-Length 128310
Connection close
Set-Cookie api_https.cookieCORS=76a6c6567ab12cea5dac4942d8df71cc; Path=/; SameSite=None; Secure
Set-Cookie api_https.cookie=76a6c6567ab12cea5dac4942d8df71cc; Path=/
Cache-Control public, must-revalidate, max-age=1
Expires Wed, 28 Jun 2023 11:26:19 GMT
Last-Modified Wed, 28 Jun 2023 11:26:18 GMT
Content-Disposition attachment; filename=API_NY.GDP.MKTP.CD_DS2_en_csv_v2_5551501.zip
Request-Context appId=cid-v1:da002513-bd8b-4441-9f30-737944134422
```

This information might be helpful when you're unsure about which file format you've just downloaded and how you're supposed to interpret its content. In this case, it's a ZIP file that's about 128 kilobytes in size. You can also deduce the original filename, which was `API_NY.GDP.MKTP.CD_DS2_en_csv_v2_5551501.zip`.

Now that you've seen how to download a file from a URL using Python's `urllib` package, it's time to tackle the same task using a third-party library. You'll find out which way is more convenient for you.

Using the Third-Party `requests` Library

While `urllib` is a good built-in option, there may be scenarios where you need to use third-party libraries to make more advanced HTTP requests, such as those requiring some form of authentication. The `requests` library is a popular, user-friendly, and Pythonic API for making HTTP requests in Python. It can handle the complexities of low-level network communication behind the curtain.

The `requests` library is also known for its flexibility and offers tighter control over the download process, allowing you to customize it according to your project requirements. Some examples include the ability to specify request headers, handle cookies, access data behind login-gated web pages, stream data in chunks, and more.

In addition, the library is designed to be efficient and performant by supporting various features that enhance the overall download performance. Its ability to automatically handle connection pooling and reuse optimizes network utilization and reduces overhead.

Now, you'll look into using the `requests` library to download that same ZIP file with **GDP by country** data from the World Bank Open Data platform. To begin, install the `requests` library into your active virtual environment using `pip`:

Shell

```
(venv) $ python -m pip install requests
```

This command installs the latest release of the `requests` library into your virtual environment. Afterward, you can start a new Python REPL session and import the `requests` library:

Python

```
>>> import requests
```

Before moving further, it's worth recalling the available HTTP methods because the `requests` library exposes them to you through Python functions. When you make HTTP requests to web servers, you have two commonly used methods to choose from:

1. HTTP GET
2. HTTP POST

You'll use the GET method to retrieve data by fetching a representation of the remote resource without modifying the server's state. Therefore, you'll commonly use it to retrieve files like images, HTML web pages, or raw data. You'll use the GET request in later steps.

The POST method allows you to send data for the server to process or use in creating or updating a resource. In POST requests, the data is typically sent in the request body in various formats like JSON or XML, and it's not visible in the URL. You can use POST requests for operations that modify server data, such as creating, updating, or submitting existing or new resources.

In this tutorial, you'll only use GET requests for downloading files.

Next, define the URL of the file that you want to download. To include additional query parameters in the URL, you'll pass in a dictionary of strings as key-value pairs:

Python

```
>>> url = "https://api.worldbank.org/v2/en/indicator/NY.GDP.MKTP.CD"
>>> query_parameters = {"downloadformat": "csv"}
```

In the example above, you define the same URL as before but specify the `downloadformat=csv` parameter separately using a Python dictionary. The library will append those parameters to the URL after you pass them to `requests.get()` using an optional `params` argument:

Python

```
>>> response = requests.get(url, params=query_parameters)
```

This makes a GET request to retrieve data from the constructed URL with optional query parameters. The function returns an HTTP response object with the server's response to the request. If you'd like to see the constructed URL with the optional parameters included, then use the response object's `.url` attribute:

Python

```
>>> response.url
'https://api.worldbank.org/v2/en/indicator/NY.GDP.MKTP.CD?downloadformat=csv'
```

The response object provides several other convenient attributes that you can check out. For example, these two will let you determine if the request was successful and what HTTP status code the server returned:

Python

```
>>> response.ok
True

>>> response.status_code
200
```

A status code of `200` indicates that your request has been completed successfully. Okay, but how do you access the data payload, usually in JSON format, that you've retrieved with the `requests` library? Read on to answer this question in the next section.

Saving Downloaded Content to a File

Now that you've retrieved content from a URL using the `requests` library, you can save it to your computer locally. When saving data to a file in Python, it's highly recommended to use the `with` statement. It ensures that Python properly manages resources, including files, and automatically closes them when you no longer need them.

There are a few ways in which you can access data retrieved with the `requests` library, depending on content type. In particular, when you want to save the original data to a local file, then you'll use the `.content` attribute of the returned response object. Because this attribute holds raw bytes, you'll open a new file in **binary mode** for writing (`'wb'`) and then write the downloaded content to this file:

Python

```
>>> with open("gdp_by_country.zip", mode="wb") as file:
...     file.write(response.content)
...
128310
```

What you see in the output is the number of bytes saved to the file. In this case, it's consistent with the expected content length that you saw earlier. Fine, but that file was only a hundred or so kilobytes long. How about downloading much larger files, which are so common in many data science projects?

Downloading a Large File in a Streaming Fashion

You've now seen how to download a single ZIP file using both the standard `urllib` package and the third-party `requests` library. If your project requires downloading a larger file, then you may run into issues using the steps above when you try to load the entire file into memory.

To overcome those issues, you can download large files in a **streaming fashion** to avoid reading the content of large responses all at once. Data streams enable you to process and handle the data in manageable chunks, making the download process more efficient and saving memory.

Data streaming also offers advantages in other scenarios when downloading files in Python, such as the ability to:

- **Download and process a file in small chunks**: This comes in handy when a network enforces restrictions on the size of data transfer in a single request. In these instances, streaming data can allow you to bypass these limitations to download and process the file in smaller chunks.
- **Process and consume the data in real time**: By processing the data as it arrives, you can use and extract insights from the downloaded content of the file while the remaining data continues to download.
- **Pause and resume the download process**: This enables you to download a portion of the file, pause the operation, and later resume where you left off, without having to restart the entire download.

To download a large file in a streaming manner, you'd keep the request connection open and download only the response headers by setting the `stream` keyword argument in the `requests.get()` function. Go ahead and try an example by downloading a large ZIP file, which is around 72 megabytes, containing the World Development Indicators from the World Bank Open Data platform:

Python

```
>>> url = "https://databank.worldbank.org/data/download/WDI_CSV.zip"
>>> response = requests.get(url, stream=True)
```

The `stream=True` parameter makes the `requests` library send a GET request at the specified URL in a streaming fashion, which downloads only the HTTP response headers first. You can view those response headers by accessing the `.headers` attribute of the received object:

Python

```
>>> response.headers
{'Date': 'Wed, 28 Jun 2023 12:53:58 GMT',
 'Content-Type': 'application/x-zip-compressed',
 'Content-Length': '71855385',
 'Connection': 'keep-alive',
 'Last-Modified': 'Thu, 11 May 2023 14:56:30 GMT',
 'ETag': '0x8DB522FE768EA66',
 'x-ms-request-id': '8490ea74-101e-002f-73bf-a9210b000000',
 'x-ms-version': '2009-09-19',
 'x-ms-lease-status': 'unlocked',
 'x-ms-blob-type': 'BlockBlob',
 'Cache-Control': 'public, max-age=3600',
 'x-azure-ref': '20230628T125357Z-99z2qrefc90b99ypt8spyt0dn40000...8dfa',
 'X-Cache': 'TCP_MISS',
 'Accept-Ranges': 'bytes'}
```

As you can see, one of the headers tells you that the server keeps the connection **alive** for you. This is an HTTP persistent connection, which allows you to potentially send multiple HTTP requests within a single network connection. Otherwise, you'd have to establish a new TCP/IP connection for each outgoing request, which is an expensive operation that takes time.

Another advantage of the streaming mode in the `requests` library is that you can download data in chunks even when you send only one request. To do so, use the `.iter_content()` method provided by the response object. This enables you to iterate through the response data in manageable chunks. In addition, you can specify the chunk size using the `chunk_size` parameter, which represents the number of bytes that it should read into memory.

With data streaming, you'll want to save the downloaded content locally as you progress through the download process:

Python

```
>>> with open("WDI_CSV.zip", mode="wb") as file:
...     for chunk in response.iter_content(chunk_size=10 * 1024):
...         file.write(chunk)
...
```

You specify a desired filename or path and open the file in binary mode (`wb`) using the `with` statement for better resource management. Then, you iterate through the response data using `response.iter_content()`, choosing an optional chunk size, which is `10` kilobytes in this case. Finally, you write each chunk to the file within the loop's body.

Note: When you don't intend to consume the entire message body by reading all of the chunks, then you should also use the `with` statement for the request. Doing so will ensure that you have access to the response until you're done reading all of the desired content:

Python

```
>>> with requests.get(url, stream=True) as response:
...     # ...
...
```

This will ensure that the library gracefully closes the underlying connection and releases it to a shared pool for subsequent requests.

You're getting better at downloading a single file under different scenarios using Python. However, in real life, you'll often want to download more than one file at the same time. For example, you may need to fetch a set of invoice documents from a given time period. In the next section, you'll explore a few different ways to download multiple files at once in Python.

Performing Parallel File Downloads

Downloading multiple files is yet another common scenario in Python. In such a case, you can speed up the download process by getting the files in parallel. There are two popular approaches to downloading several files simultaneously:

1. Using a **pool of threads** with the `requests` library
2. Using **asynchronous downloads** with the `aiohttp` library

You'll start with the first approach now.

Using a Pool of Threads With the `requests` Library

When you want to make many HTTP requests to the same or different servers, you can take advantage of **multithreading** to decrease the overall execution time of your code.

In a multithreaded program, you typically spawn multiple threads of execution, each with its own instruction sequence that runs independently and in parallel with the other. These threads can perform different tasks or execute other parts of the program concurrently, improving performance and maximizing the use of the available CPU cores. For example, each thread can make its own HTTP request.

You'll now try concurrently downloading three ZIP files from the World Bank Open Data platform:

1. Total population by country
2. GDP by country
3. Population density by country

In this example, you'll use a **pool of threads** with the `requests` library to understand how to perform parallel file downloads. First, import the `ThreadPoolExecutor` class from the `concurrent.futures` module and the `requests` library again:

Python

```
>>> from concurrent.futures import ThreadPoolExecutor
>>> import requests
```

Next, write a function that you'll execute within each thread to download a single file from a given URL:

Python

```
>>> def download_file(url):
...     response = requests.get(url)
...     if "content-disposition" in response.headers:
...         content_disposition = response.headers["content-disposition"]
...         filename = content_disposition.split("filename=")[1]
...     else:
...         filename = url.split("/")[-1]
...     with open(filename, mode="wb") as file:
...         file.write(response.content)
...     print(f"Downloaded file {filename}")
...
```

This function takes a URL as an argument, makes a GET request using the `requests` library, and saves the retrieved data in a local file. In this specific example, it first attempts to extract the filename from the `Content-Disposition` response header, which contains information on which items on the page are displayed inline versus as attachments. If that's not available, then it gets the filename from a part of the URL.

> **Note**: Another response header element that's worth knowing is the status code. This indicates the status of the request. For example, `200` is successful, `301` is a redirect, and `404` means page not found.

You'll be downloading three separate files from the same API, so create a URL template and populate a Python list with it:

Python

```
>>> template_url = (
...     "https://api.worldbank.org/v2/en/indicator/"
...     "{resource}?downloadformat=csv"
... )

>>> urls = [
...     # Total population by country
...     template_url.format(resource="SP.POP.TOTL"),
...
...     # GDP by country
...     template_url.format(resource="NY.GDP.MKTP.CD"),
...
...     # Population density by country
...     template_url.format(resource="EN.POP.DNST"),
... ]
```

Here, you call the `.format()` method on the template string with different resource names that correspond to ZIP files with CSV data on a remote server.

To download these files concurrently using multiple threads of execution, create a new thread pool and map your `download_file()` function onto each item from the list:

Python

```
>>> with ThreadPoolExecutor() as executor:
...     executor.map(download_file, urls)
...
<generator object Executor.map.<locals>.result_iterator at 0x7fc9c90f0640>
Downloaded file API_SP.POP.TOTL_DS2_en_csv_v2_5551506.zip
Downloaded file API_NY.GDP.MKTP.CD_DS2_en_csv_v2_5551501.zip
Downloaded file API_EN.POP.DNST_DS2_en_csv_v2_5552158.zip
```

As before, you use the `with` statement to help with resource management for your threads. Within the executor's context, you call the `.map()` method with the function that you want to execute. You also pass it an iterable, which is the list of URLs in this case. The executor object allocates a pool of threads up front and assigns a separate thread to each download task.

Because downloading data from a network is an I/O-bound task, which is limited by the speed at which the data can be read rather than the CPU speed, you achieve true parallelism. Despite Python's global interpreter lock (GIL), which would otherwise prevent that, the threads in the executor perform multiple tasks in parallel, resulting in faster overall completion time.

The sample ZIP files are relatively small and similar in size, so they finish downloading at roughly the same time. But what if you used a `for` loop to download the files instead? Go ahead and call `download_file()` with each URL on the main thread:

Python

```
>>> for url in urls:
...     download_file(url)
...
```

You may have noticed that this operation took much longer. That's because the files were downloaded sequentially instead of concurrently. So when the first file starts downloading, the second one won't start until it finishes, and so on. This means that the total time required to download all the files is the sum of the download times for each individual file.

> **Notes**: While timing your code is beyond the scope of this tutorial, there are plenty of ways to do so in Python. Check out Python Timer Functions: Three Ways to Monitor Your Code to learn what they are.

Unfortunately, using threads isn't always desirable due to their complexity. The example in this section was fairly straightforward because you didn't have to worry about thread synchronization, coordination, or resource management. But when working with multithreaded code, you should be careful about thread safety to ensure that you safely access and modify shared resources to avoid data corruption.

> **Note:** It's also important not to run into deadlocks, livelocks, resource starvation, and many more problems related to

threads and concurrent programming in general.

There are some conditions where multithreading may not improve your performance at all. If the underlying problem is inherently sequential, then there's no way to parallelize it. Moreover, if your tasks involve CPU-bound operations, then Python won't be able to take advantage of multiple CPU cores because of the GIL from earlier. The additional cost of context switching may actually reduce performance!

If working with threads makes your head spin, then you might be interested in exploring alternative solutions, like Python's **asynchronous programming** capabilities.

Using the Asynchronous `aiohttp` Library

In addition to multithreading, another method to download multiple files concurrently is by using the `async/await` pattern in Python. It involves running multiple non-blocking tasks asynchronously in an event loop by allowing them to suspend and resume execution voluntarily as a form of cooperative multitasking. This is different from threads, which require a preemptive scheduler to manage context switching between them.

Asynchronous tasks also differ from multithreading in that they execute concurrently within a *single* thread instead of multiple threads. Therefore, they must periodically give up their execution time to other tasks without hogging the CPU. By doing so, **I/O-bound tasks** such as asynchronous downloads allow for concurrency, as the program can switch between tasks and make progress on each in parallel.

A popular Python package to perform asynchronous downloads when retrieving multiple files is the `aiohttp` library. It's built on top of the standard library's `asyncio` module, which provides a framework for asynchronous programming in Python.

The `aiohttp` library takes advantage of the concurrency features in the `asyncio` package, allowing you to write asynchronous code that can handle multiple requests concurrently. The library can perform **non-blocking network operations**, meaning it'll let other code run while another task waits for data to arrive from the network.

By using the `aiohttp` API along with the `async def` and `await` keywords, you can write asynchronous code that makes concurrent HTTP requests. In addition, the `aiohttp` library supports **connection pooling**, a feature that allows multiple requests to use the same underlying connection. This helps to optimize and improve performance.

To begin, install the `aiohttp` library using `pip` in the command line:

Shell

```
(venv) $ python -m pip install aiohttp
```

This installs the `aiohttp` library into your active virtual environment.

In addition to this third-party library, you'll also need the `asyncio` package from the Python standard library to perform asynchronous downloads. So, import both packages now:

Python

```
>>> import asyncio
>>> import aiohttp
```

The next step is defining an asynchronous function to download a file from a URL. You can do so by creating an `aiohttp.ClientSession` instance, which holds a connector reused for multiple connections. It automatically keeps them alive for a certain time period to reuse them in subsequent requests to the same server whenever possible. This improves performance and reduces the overhead of establishing new connections for each request.

The following function performs an **asynchronous download** using the `ClientSession` class from the `aiohttp` package:

Python

```
>>> async def download_file(url):
...     async with aiohttp.ClientSession() as session:
...         async with session.get(url) as response:
...             if "content-disposition" in response.headers:
...                 header = response.headers["content-disposition"]
...                 filename = header.split("filename=")[1]
...             else:
...                 filename = url.split("/")[-1]
...             with open(filename, mode="wb") as file:
...                 while True:
...                     chunk = await response.content.read()
...                     if not chunk:
...                         break
...                     file.write(chunk)
...                 print(f"Downloaded file {filename}")
...
```

The function defined above takes a URL as an argument. It then creates a client session using the `async with` statement, ensuring that the session is properly closed and resources are released after the program exits this code block. With the session context, it makes an HTTP GET request to the specified URL and obtains the response object using the `async with` statement.

Inside the infinite loop, you read data in chunks, breaking out of the loop when there are no more chunks. The `await` keyword indicates that this operation is asynchronous and other tasks can execute in parallel until the data is available. After deriving the filename from the response object, you save the downloaded chunk of data in a local file.

Afterward, you can perform concurrent downloads using the asynchronous capabilities of the `aiohttp` and `asyncio` libraries. You may reuse the code from an earlier example based on multithreading to prepare a list of URLs:

Python

```
>>> template_url = (
...     "https://api.worldbank.org/v2/en/indicator/"
...     "{resource}?downloadformat=csv"
... )

>>> urls = [
...     # Total population by country
...     template_url.format(resource="SP.POP.TOTL"),
...
...     # GDP by country
...     template_url.format(resource="NY.GDP.MKTP.CD"),
...
...     # Population density by country
...     template_url.format(resource="EN.POP.DNST"),
... ]
```

Finally, define and run an asynchronous `main()` function that will download files concurrently from those URLs:

Python

```
>>> async def main():
...     tasks = [download_file(url) for url in urls]
...     await asyncio.gather(*tasks)
...
>>> asyncio.run(main())
Downloaded file API_SP.POP.TOTL_DS2_en_csv_v2_5551506.zip
Downloaded file API_EN.POP.DNST_DS2_en_csv_v2_5552158.zip
Downloaded file API_NY.GDP.MKTP.CD_DS2_en_csv_v2_5551501.zip
```

In the snippet above, the `main()` function defines a list comprehension to create a list of tasks, whereby each task calls the `download_file()` function for each URL in the global variable `urls`. Note that you define the function with the `async` keyword to make it asynchronous. Then, you use `asyncio.gather()` to wait until all tasks in the list are completed.

You run the `main()` function using `asyncio.run()`, which initiates and executes the asynchronous tasks in an event loop. This code downloads files concurrently from the specified URLs without blocking the execution of other tasks.

Deciding Which Option to Choose

At this point, you've learned how to use several tools to download files in Python. Depending on the task at hand, you may want to choose one option over the others. Some factors to consider are the size and number of files that you'll be working with, as well as the tool's ease of use, flexibility, and additional features.

File Sizes to Download

If you're looking to download large files, then the `requests` library is a good option that will handle them efficiently. It can **stream data**, letting you iterate over the message body in chunks for a more efficient process and better memory use.

When downloading multiple files, you can either use the `requests` library together with **multithreading** or the `aiohttp` library with **asynchronous downloads** to perform concurrent downloads. When used properly, either can improve performance and optimize the download process.

User-Friendliness

If you're looking for something quick and straightforward, then the `urllib` package is already included in Python's standard library, requiring no additional installation. This is a good option if you want to download small files that can fit entirely into memory without any issues.

While you could build many functionalities using `urllib`, it may be harder to use and require more manual setup for more involved use cases, such as streaming data and downloading multiple files in parallel. In fact, there are third-party libraries with functions that readily support these features, such as the `requests` library, which can support data streaming with an argument in the `requests.get()` method.

Additional Features and Flexibility

The `requests` library has a rich set of features that can help you in numerous other download scenarios. Although you didn't extensively cover these tasks, the `requests` library has features that can handle authentication, redirects, session management, and more. These features can give you more control and flexibility for more advanced tasks.

If you'd like to learn more about a project that might require extra features supported by the `requests` library, then check out this project on building a content aggregator. Creating a content aggregator involves steps to download data from multiple websites, some of which may require authentication and session management, so `requests` would really come in handy.

Conclusion

You can use Python to automate your file downloads or to have better control and flexibility over this process. Python offers several options for downloading files from URLs that cater to different scenarios, such as downloading large files, multiple files, or files behind gated web pages that require authentication.

In this tutorial, you've learned the steps to download files in Python, including how to:

- **Download** files from the Internet using both built-in and external libraries in Python
- Perform **data streaming** and download large files in smaller, more manageable chunks
- Use a pool of threads to fetch multiple files **concurrently**
- Download multiple files **asynchronously** when performing bulk downloads

In addition, you've seen how to use the `requests` library for its **streaming** and **parallel downloading** capabilities. You've also seen examples using the `aiohttp` and `asyncio` libraries for concurrent requests and **asynchronous downloads** to improve download speeds for multiple files.