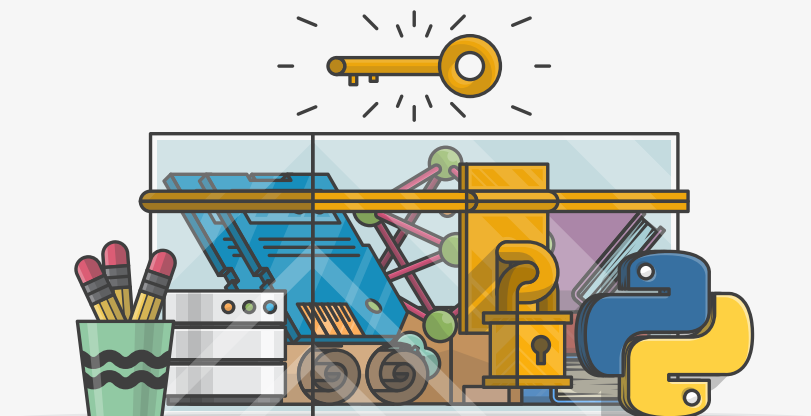


**Keep reading Real Python by  
creating a free account or  
signing in:**



[Continue »](#)

Already have an account? [Sign-In](#)

- Organize Your Site Connectivity Checker Project
- Step 2: Check Websites' Connectivity in Python
  - Implement a Connectivity Checker Function
  - Run Your First Connectivity Checks
- Step 3: Create Your Website Connectivity Checker's CLI
  - Parse Website URLs at the Command Line

- Load Website URLs From a File
- Display the Check Results
- Step 4: Put Everything Together in the App's Main Script
  - Create the Application's Entry-Point Script
  - Build the List of Target Website URLs
  - Check the Connectivity of Multiple Websites
  - Run Connectivity Checks From the Command Line
- Step 5: Check Websites' Connectivity Asynchronously
  - Implement an Asynchronous Connectivity Checker Function
  - Add an Asynchronous Option to the Application's CLI
  - Check the Connectivity of Multiple Websites Asynchronously
  - Add Asynchronous Checks to the App's Main Code
- Conclusion
- Next Steps


### Python Data Connectors

Connect to 250+ SaaS, NoSQL, & Big Data sources from pandas, SQLAlchemy, Dash, petl, and more!



**cdata**  
Learn More

 Remove ads

 This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Building a Site Connectivity Checker**

Building a site connectivity checker in Python is an interesting project to level up your skills. With this project, you'll integrate knowledge related to handling **HTTP requests**, creating **command-line interfaces (CLI)**, and organizing your application's code using common Python **project layout** practices.

By building this project, you'll learn how Python's **asynchronous features** can help you deal with multiple HTTP requests efficiently.

**In this tutorial, you'll learn how to:**

- Create command-line interfaces (CLI) using Python's `argparse`
- Check if a website is online using Python's `http.client` from the standard library
- Implement **synchronous checks** for multiple websites
- Check if a website is online using the `aiohttp` third-party library
- Implement **asynchronous checks** for multiple websites

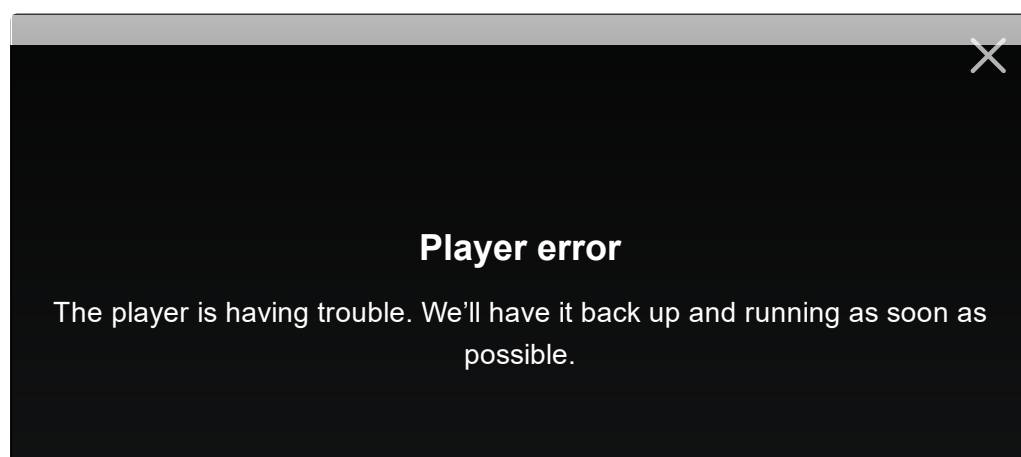
To get the most out of this project, you need to know the basics of handling [HTTP requests](#) and using [argparse](#) to create CLIs. You should also be familiar with the [asyncio](#) module and the [async and await](#) keywords.

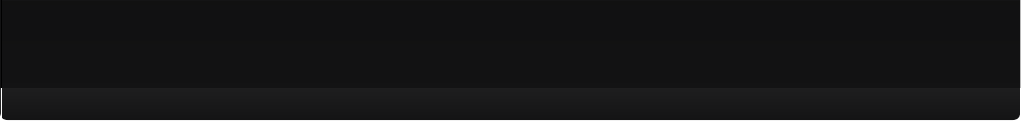
But don't worry! The topics throughout the tutorial will be introduced in a step-by-step fashion so that you can grasp them as you go. Additionally, you can download the complete source code and other resources for this project by clicking the link below:

**Get Source Code:** [Click here to get the source code you'll use to build your site connectivity checker app.](#)

## Demo: A Site Connectivity Checker

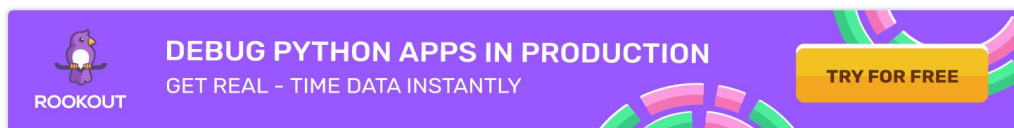
In this step-by-step project, you'll build an application that checks if one or more websites are online at a given moment. The app will take a list of target URLs at the command line and check them for connectivity either **synchronously** or **asynchronously**. The following video shows how the app works:





Your site connectivity checker can take one or more URLs at the command line. Then it creates an internal list of target URLs and checks them for connectivity by issuing HTTP requests and processing the corresponding responses.

Using the `-a` or `--asynchronous` option makes the application perform the connectivity checks asynchronously, potentially resulting in lower execution times, especially when you're processing a long list of websites.



 Remove ads

## Project Overview

Your website connectivity checker app will provide a few options through a minimal [command-line interface \(CLI\)](#). Here's a summary of these options:

- `-u` or `--urls` allows you to provide one or more target URLs at the command line.
- `-f` or `--input-file` allows you to supply a file containing a list of URLs to check.
- `-a` or `--asynchronous` allows you to run the connectivity checks asynchronously.

By default, your application will run the connectivity checks synchronously. In other words, the app will perform the checks one after another.

With the `-a` or `--asynchronous` option, you can modify this behavior and make the app run the connectivity checks concurrently. To do this, you'll take advantage of Python's [asynchronous features](#) and the [aiohttp](#) third-party library.

Running asynchronous checks can make your website connectivity checker faster and more efficient, especially when you have a long list of URLs to check.

Internally, your application will use the standard-library [http.client](#) module to create a connection to the target website. Once you have a connection, then you can make an HTTP request to the website, which will hopefully react with an appropriate response. If the request is successful, then you'll know that the site is online. Otherwise, you'll know that the site is offline.

To display the result of every connectivity check on your screen, you'll provide your app with a nicely formatted output that will make the app appealing to your users.

## Prerequisites

The project that you'll build in this tutorial will require familiarity with general Python programming. Additionally, it'll require basic knowledge of the following topics:

- Handling [exceptions](#) in Python
- Working with [files](#), the [with statement](#), and the [pathlib](#) module
- Handling [HTTP requests](#) with standard-library or third-party tools
- Creating CLI applications with the [argparse](#) module
- Using Python's [asynchronous features](#)

Knowing the basics of the [aiohttp](#) third-party library would also be a plus but not a requirement. However, if you don't have all this knowledge yet, then that's okay! You might learn more by going ahead and giving the project a shot. You can always stop and review the resources linked here if you get stuck.

With this short overview of your website connectivity checker project and the prerequisites, you're almost ready to start Pythoning and having fun while coding. But first, you need to create

a suitable working environment and set up your project's layout.

## Step 1: Set Up Your Site Connectivity Checker Project in Python

In this section, you'll get ready to start coding your site connectivity checker app. You'll start by creating a Python [virtual environment](#) for the project. This environment will allow you to isolate the project and its dependencies from other projects and your system Python installation.

The next step is to set up the project's [layout](#) by creating all the required files and the directory structure.

To download the code for this first step, click the following link and navigate to the `source_code_step_1/` folder:

**Get Source Code:** Click here to get the source code you'll use to build your site connectivity checker app.

## Set Up the Development Environment

Before you start coding a new project, you should do some preparation. In Python, you typically start by creating a **virtual environment** for the project. A virtual environment provides an isolated Python interpreter and a space to install your project's dependencies.

To kick things off, go ahead and create the project's root directory, called `rpchecker_project/`. Then move to this directory and run the following commands on your system's command line or terminal:

 Windows

 Linux + macOS

Windows PowerShell

```
PS> python -m venv venv
PS> venv\Scripts\activate
(venv) PS>
```


The first command creates a fully functional Python virtual environment called `venv` inside the project's root directory, while the second command activates the environment. Now run the following command to install the project's dependencies with `pip`, the standard Python package manager:

#### Shell


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

With this command, you install `aiohttp` into your virtual environment. You'll use this third-party library along with Python's `async` features to handle asynchronous HTTP requests in your site connectivity checker app.

Cool! You have a working Python virtual environment with all the dependencies that you'll need to start building your project. Now you can create the project's layout to organize your code following Python best practices.

Get usable transcripts with   
**understanding on any audio.**

\$150 in Free Credit



 Remove ads

## Organize Your Site Connectivity Checker Project

Python is surprisingly flexible when it comes to structuring applications, so you may find pretty different structures from project to project. However, small [installable Python projects](#) typically have a single [package](#), which is often named after the project itself.

Following this practice, you can organize your site connectivity checker app using the following directory structure:

```
rpchecker_project/  
|
```

```
├── rpchecker/
│   ├── __init__.py
│   ├── __main__.py
│   ├── checker.py
│   └── cli.py
├── README.md
└── requirements.txt
```

You can use any name for this project and its main package. In this tutorial, the project will be named `rpchecker` as a combination of Real Python (`rp`) and `checker`, which points out the app's main functionality.

The `README.md` file will contain the project's description and instructions for installing and running the application. Adding a `README.md` file to your projects is a best practice in programming, especially if you're planning to release the project as an open source solution. To learn more about writing good `README.md` files, check out [How to write a great README for your GitHub project](#).

The `requirements.txt` file will hold the list of your project's external dependencies. In this case, you only need the `aiohttp` library, because the rest of the tools and modules that you'll use are available out of the box in the Python [standard library](#). You can use this file to automatically reproduce the appropriate Python virtual environment for your app using `pip`, the standard package manager.

**Note:** You won't be adding content to the `README.md` and `requirements.txt` files in this tutorial. To get a taste of their content, download the bonus material provided in this tutorial and check out the corresponding files.

Inside the `rpchecker/` directory, you'll have the following files:

- `__init__.py` enables `rpchecker/` as a Python package.
- `__main__.py` works as an entry-point script for the app.
- `checker.py` provides the application's core functionalities.
- `cli.py` contains the command-line interface for the



application.

Now go ahead and create all these files as empty files. You can do this by using your favorite [code editor or IDE](#). Once you finish creating the project's layout, then you can start coding the app's main functionality: *checking if a website is online or not*.

## Step 2: Check Websites' Connectivity in Python

At this point, you should have a suitable Python virtual environment with your project's dependencies installed in it. You should also have a project directory containing all the files that you'll use throughout this tutorial. It's time to start coding!

Before jumping into the really fun stuff, go ahead and add the application's version number to the `__init__.py` module in your `rpchecker` package:

Python

```
# __init__.py

__version__ = "0.1.0"
```

The `__version__` module-level constant holds your project's current version number. Because you're creating a brand-new app, the initial version is set to `0.1.0`. With this minimal setup, you can start implementing the connectivity-checking functionality.

To download the code for this step, click the following link and look inside the `source_code_step_2/` folder:

**Get Source Code:** [Click here to get the source code you'll use to build your site connectivity checker app.](#)

## Implement a Connectivity Checker Function

There are several Python tools and libraries that you can use to check if a website is online at a given time. For example, a popular option is the [requests](#) third-party library, which allows you to perform HTTP requests using a human-readable [API](#).

However, using `requests` has the drawback of installing an external library just to use a minimal part of its functionality. It'd be more efficient to find an appropriate tool in the Python standard library.

With a quick look at the standard library, you'll find the [urllib](#) package, which provides several modules for handling HTTP requests. For example, to check if a website is online, you can use the `urlopen()` function from the [urllib.request](#) module:

```
Python >>>

>>> from urllib.request import urlopen

>>> response = urlopen("https://python.org")
>>> response.read()
b'<!doctype html>\n<!--[if lt IE 7]>
...
```

The `urlopen()` function takes a URL and opens it, returning its content as a string or a [Request](#) object. But you just need to check if the website is online, so downloading the entire page would be wasteful. You need something more efficient.

What about a tool that gives you lower-level control over your HTTP request? That's where the [http.client](#) module comes in. This module provides the [HTTPConnection](#) class, representing a connection to a given HTTP server.

`HTTPConnection` has a `.request()` method that allows you to perform HTTP requests using the different [HTTP methods](#). For this project, you can use the [HEAD](#) HTTP method to ask for a response containing only the [headers](#) of the target website. This option will reduce the amount of data to download, making your connectivity checker app more efficient.

At this point, you have a clear idea of the tool to use. Now you can

do some quick tests. Go ahead and run the following code in a Python interactive session:

```
Python >>>

>>> from http.client import HTTPConnection

>>> connection = HTTPConnection("pypi.org", port=80, timeout=10)
>>> connection.request("HEAD", "/")

>>> response = connection.getresponse()
>>> response.getheaders()
[('Server', 'Varnish'), ..., ('X-Permitted-Cross-Domain-Policies', 'none;')]
>>>
```

In this example, you first create an `HTTPConnection` instance targeting the `pypi.org` website. The connection uses port 80, which is the default HTTP port. Finally, the `timeout` argument provides the number of seconds to wait before timing out connection attempts.

Then you perform a HEAD request on the site's root path, `/`, using `.request()`. To get the actual response from the server, you call `.getresponse()` on the connection object. Finally, you inspect the response's headers by calling `.getheaders()`.

Your website connectivity checker just needs to create a connection and make a HEAD request. If the request is successful, then the target website is online. Otherwise, the site is offline. In the latter case, it'd be appropriate to display an error message to the user.

Now go ahead and open the `checker.py` file in your code editor. Then add the following code to it:

```
Python

1  # checker.py
2
3  from http.client import HTTPConnection
4  from urllib.parse import urlparse
5
6  def site_is_online(url, timeout=2):
7      """Return True if the target URL is online.
8
9      Raise an exception otherwise.
10     """
```

```

11     error = Exception("unknown error")
12     parser = urlparse(url)
13     host = parser.netloc or parser.path.split("/")[0]
14     for port in (80, 443):
15         connection = HTTPConnection(host=host, port=port, timeout=timeout)
16         try:
17             connection.request("HEAD", "/")
18             return True
19         except Exception as e:
20             error = e
21         finally:
22             connection.close()
23     raise error

```

Here's a breakdown of what this code does line by line:

- **Line 3** imports `HTTPConnection` from `http.client`. You'll use this class to establish a connection with the target website and handle HTTP requests.
- **Line 4** imports `urlparse()` from `urllib.parse`. This function will help you parse the target URLs.
- **Line 6** defines `site_is_online()`, which takes a `url` and a `timeout` argument. The `url` argument will hold a string representing a website's URL. Meanwhile, `timeout` will hold the number of seconds to wait before timing out connection attempts.
- **Line 11** defines a generic `Exception` instance as a placeholder.
- **Line 12** defines a `parser` variable containing the result of parsing the target URL using `urlparse()`.
- **Line 13** uses the `or` operator to extract the hostname from the target URL.
- **Line 14** starts a `for` loop over the HTTP and HTTPS ports. This way, you can check if the website is available on either port.
- **Line 15** creates an `HTTPConnection` instance using `host`, `port`, and `timeout` as arguments.

- **Lines 16 to 22** define a try ... except ... finally statement. The try block attempts to make a HEAD request to the target website by calling `.request()`. If the request succeeds, then the function returns `True`. If an exception occurs, then the except block keeps a reference to that exception in `error`. The finally block closes the connection to free the acquired resources.
- **Line 23** raises the exception stored in `error` if the loop finishes without a successful request.

Your `site_is_online()` function returns `True` if the target website is available online. Otherwise, it raises an exception pointing out the problem it encountered. This latter behavior is convenient because you need to show an informative error message when the site isn't online. Now it's time to try out your new function.



**Master Real-World Python Skills**  
With a Community of Experts  
Level Up With Unlimited Access to Our Vast Library  
of Python Tutorials and Video Lessons

[Watch Now »](#)

 Remove ads

## Run Your First Connectivity Checks

To try out your `site_is_online()` function, go ahead and get back to your interactive session. Then run the following code:

```
Python >>>

>>> from rpchecker.checker import site_is_online

>>> site_is_online("python.org")
True

>>> site_is_online("non-existing-site.org")
Traceback (most recent call last):
...
socket.gaierror: [Errno -2] Name or service not known
```

In this code snippet, you first import `site_is_online()` from the `checker` module. Then you call the function with `"python.org"` as an argument. Because the function returns `True`, you know that the target site is online.

In the final example, you call `site_is_online()` with a non-existing website as a target URL. In this case, the function raises an exception that you can later catch and process to display an error message to the user.

Great! You've implemented the application's main functionality of checking a website's connectivity. Now you can continue with your project by setting up its CLI.

## Step 3: Create Your Website Connectivity Checker's CLI

So far, you have a working function that allows you to check if a given website is online by performing an HTTP request using the `http.client` module from the standard library. At the end of this step, you'll have a minimal CLI that will allow you to run your website connectivity checker app from the command line.

The CLI will include options for taking a list of URLs at the command line and loading a list of URLs from a text file. The application will also display the connectivity check results with a user-friendly message.

To create the application's CLI, you'll use `argparse` from the Python standard library. This module allows you to build user-friendly CLIs without installing any external dependencies, such as [Click](#) or [Typer](#).

To get started, you'll write the required boilerplate code for working with `argparse`. You'll also code the option to read URLs from the command line.

Click the link below to download the code for this step so that you can follow along with the project. You'll find what you need in the `source_code_step_3/` folder:

**Get Source Code:** [Click here to get the source code you'll use to build your site connectivity checker app.](#)

# Parse Website URLs at the Command Line

To build the application's CLI with `argparse`, you need to create an `ArgumentParser` instance so that you can parse `arguments` provided at the command line. Once you have an argument parser, then you can start adding arguments and `options` to your app's CLI.

Now go ahead and open the `cli.py` file in your code editor. Then add the following code:

Python

```
# cli.py

import argparse

def read_user_cli_args():
    """Handle the CLI arguments and options."""
    parser = argparse.ArgumentParser(
        prog="rpchecker", description="check the availability of websites"
    )
    parser.add_argument(
        "-u",
        "--urls",
        metavar="URLs",
        nargs="+",
        type=str,
        default=[],
        help="enter one or more website URLs",
    )
    return parser.parse_args()
```

In this code snippet, you create `read_user_cli_args()` to keep the functionality related to the argument parser in a single place. To build the parser object, you use two arguments:

- **prog** defines the program's name.
- **description** provides a suitable description for the application. This description will be displayed when you call the app with the `--help` option.

After creating the argument parser, you add a first command-line argument using `.add_argument()`. This argument will allow the user

to enter one or more URLs at the command line. It'll use the `-u` and `--urls` switches.

The rest of the arguments to `.add_argument()` work as follows:

- **metavar** sets a name for the argument in usage or help messages.
- **nargs** tells argparse to accept a list of command-line arguments after the `-u` or `--urls` switch.
- **type** sets the data type of the command-line arguments, which is `str` in this argument.
- **default** sets the command-line argument to an empty list by default.
- **help** provides a help message for the user.

Finally, your function returns the result of calling `.parse_args()` on the parser object. This method returns a `Namespace` object containing the parsed arguments.



[Become a Python Expert »](#)

 Remove ads

## Load Website URLs From a File

Another valuable option to implement in your site connectivity checker is the ability to load a list of URLs from a text file on your local machine. To do this, you can add a second command-line argument with the `-f` and `--input-file` flags.

Go ahead and update `read_user_cli_args()` with the following code:

Python

```
# cli.py
# ...

def read_user_cli_args():
    # ...
```



```

parser.add_argument(
    "-f",
    "--input-file",
    metavar="FILE",
    type=str,
    default="",
    help="read URLs from a file",
)
return parser.parse_args()

```

To create this new command-line argument, you use `.add_argument()` with almost the same arguments as in the section above. In this case, you aren't using the `nargs` argument, because you want the application to accept only one input file at the command line.

## Display the Check Results

An essential component of every application that interacts with the user through the command line is the application's output. Your application needs to show the result of its operations to the user. This feature is vital for ensuring a pleasant [user experience](#).

Your site connectivity checker doesn't need a very complex output. It just needs to inform the user about the current status of the checked websites. To implement this functionality, you'll code a function called `display_check_result()`.

Now get back to the `cli.py` file and add the function at the end:

### Python

```

# cli.py
# ...

def display_check_result(result, url, error=""):
    """Display the result of a connectivity check."""
    print(f'The status of "{url}" is:', end=" ")
    if result:
        print('"Online!" 🟢')
    else:
        print(f'"Offline?" 🚫 \n Error: "{error}"')

```

This function takes the connectivity check result, the checked URL, and an optional error message. The [conditional statement](#) tests to see if `result` is true, in which case an "Online!" message is [printed](#) to the screen. If `result` is false, then the `else` clause prints "Offline?" along with an error report about the actual problem that has just occurred.

That's it! Your website connectivity checker has a command-line interface to allow the user to interact with the application. Now it's time to put everything together in the application's entry-point script.

## Step 4: Put Everything Together in the App's Main Script

So far, your site connectivity checker project has a function that checks if a given website is online. It also has a CLI that you quickly built using the `argparse` module from the Python standard library. In this step, you'll write the [glue code](#)—the code that will bring all these components together and make your application work as a full-fledged command-line app.

To kick things off, you'll start by setting up the application's main script or [entry-point](#) script. This script will contain the `main()` function and some high-level code that will help you connect the CLI in the [front-end](#) with the connectivity-checking functionality in the back-end.

To download the code for this step, click the link below, then check out the `source_code_step_4/` folder:

**Get Source Code:** [Click here to get the source code you'll use to build your site connectivity checker app.](#)

## Create the Application's Entry-Point Script

The next step in building your website connectivity checker app is to

define the entry-point script with a suitable `main()` function. To do this, you'll use the `__main__.py` file that lives in the `rpchecker` package. Including a `__main__.py` file in a Python package enables you to run the package as an executable program using the command `python -m <package_name>`.

To start populating `__main__.py` with code, go ahead and open the file in your code editor. Then add the following:

```
Python

1  # __main__.py
2
3  import sys
4
5  from rpchecker.cli import read_user_cli_args
6
7  def main():
8      """Run RP Checker."""
9      user_args = read_user_cli_args()
10     urls = _get_websites_urls(user_args)
11     if not urls:
12         print("Error: no URLs to check", file=sys.stderr)
13         sys.exit(1)
14     _synchronous_check(urls)
```

After importing `read_user_cli_args()` from the `cli` module, you define the app's `main()` function. Inside `main()`, you'll find a few lines of code that don't work yet. Here's what this code should do after you provide the missing functionality:

- **Line 9** calls `read_user_cli_args()` to parse the command-line arguments. The resulting `Namespace` object is then stored in the `user_args` local [variable](#).
- **Line 10** puts together a list of target URLs by calling a **helper function** named `_get_websites_urls()`. You'll be coding this function in a moment.
- **Line 11** defines an `if` statement to check if the list of URLs is empty. If that's the case, then the `if` block prints an error message to the user and exits the application.

- **Line 14** invokes a function called `_synchronous_check()`, which takes the list of target URLs as an argument and runs the connectivity check over each URL. As the name points out, this function will run the connectivity checks synchronously, or one after the other. Again, you'll be coding this function in a moment.

With `main()` in place, you can start coding the missing pieces to make it work correctly. In the following sections, you implement `_get_websites_urls()` and `_synchronous_check()`. Once they're ready to go, you'll be able to run your website connectivity checker app for the first time.



[Learn Python »](#)

 Remove ads

## Build the List of Target Website URLs

Your site connectivity checker app will be able to check multiple URLs in every execution. Users will feed URLs into the app by listing them at the command line, providing them in a text file, or both. To create the internal list of target URLs, the app will first process URLs provided at the command line. Then it'll add additional URLs from a file, if any.

Here's the code that accomplishes these tasks and returns a list of target URLs that combines both sources, the command line and an optional text file:

Python

```
1  # __main__.py
2
3  import pathlib
4  import sys
5
6  from rpchecker.cli import read_user_cli_args
7
8  def main():
9      # ...
```

```

10
11 def _get_websites_urls(user_args):
12     urls = user_args.urls
13     if user_args.input_file:
14         urls += _read_urls_from_file(user_args.input_file)
15     return urls
16
17 def _read_urls_from_file(file):
18     file_path = pathlib.Path(file)
19     if file_path.is_file():
20         with file_path.open() as urls_file:
21             urls = [url.strip() for url in urls_file]
22             if urls:
23                 return urls
24             print(f"Error: empty input file, {file}", file=sys.stderr)
25     else:
26         print("Error: input file not found", file=sys.stderr)
27     return []

```

The first update in this code snippet is to import `pathlib` to manage the path to the optional URLs file. The second update is to add the `_get_websites_urls()` helper function, which does the following:

- **Line 12** defines `urls`, which initially stores the list of URLs provided at the command line. Note that if the user doesn't supply any URLs, then `urls` will store an empty list.
- **Line 13** defines a conditional that checks if the user has provided a URLs file. If so, then the `if` block augments the list of target URLs resulting from calling `_read_urls_from_file()` with the file provided in the `user_args.input_file` command-line argument.
- **Line 15** returns the resulting list of URLs.

At the same time, `_read_urls_from_file()` runs the following actions:

- **Line 18** turns the `file` argument into a `pathlib.Path` object to facilitate further processing.
- **Line 19** defines a conditional statement that checks if the current file is an actual file in the local file system. To perform

this check, the conditional calls `.is_file()` on the Path object. Then the `if` block opens the file and reads its content using a list comprehension. This comprehension strips any possible leading and ending whitespace from every line in the file to prevent processing errors later.

- **Line 22** defines a nested conditional to check if any URL has been gathered. If so, then line 23 returns the resulting list of URLs. Otherwise, line 24 prints an error message to inform the reader that the input file is empty.

The `else` clause on lines 25 to 26 prints an error message to point out that the input file doesn't exist. If the function runs without returning a valid list of URLs, it returns an empty list.

Wow! That was a lot, but you made it to the end! Now you can continue with the final part of `__main__.py`. In other words, you can implement the `_synchronous_check()` function so that the app can perform connectivity checks on multiple websites.

## Check the Connectivity of Multiple Websites

To run connectivity checks over multiple websites, you need to iterate through the list of target URLs, do the checks, and display the corresponding results. That's what the `_synchronous_check()` function below does:

Python

```
1  # __main__.py
2
3  import pathlib
4  import sys
5
6  from rpchecker.checker import site_is_online
7  from rpchecker.cli import display_check_result, read_user_c
8
9  # ...
10
11 def _synchronous_check(urls):
12     for url in urls:
```

```

13         error = ""
14         try:
15             result = site_is_online(url)
16         except Exception as e:
17             result = False
18             error = str(e)
19         display_check_result(result, url, error)
20
21 if __name__ == "__main__":
22     main()

```

In this piece of code, you first update your imports by adding `site_is_online()` and `display_check_result()`. Then you define `_synchronous_check()`, which takes a list of URLs as arguments. The function's body works like this:

- **Line 12** starts a for loop that iterates over the target URLs.
- **Line 13** defines and initializes `error` which will hold the message that will be displayed if the app doesn't get a response from the target website.
- **Lines 14 to 18** define a try ... except statement that catches any exception that may occur during the connectivity checks. These checks run on line 15, which calls `site_is_online()` with the target URL as an argument. Then lines 17 and 18 update the `result` and `error` variables if a connection problem happens.
- **Line 19** finally calls `display_check_result()` with appropriate arguments to display the connectivity check result to the screen.

To wrap up the `__main__.py` file, you add the typical Python `if __name__ == "__main__":` boilerplate code. This snippet calls `main()` when the module is [run as a script](#) or executable program. With these updates, your application is now ready for a test flight!

## Run Connectivity Checks From the Command Line

You've written a ton of code without having the chance to see it in action. You've coded the site connectivity checker's CLI and its entry-point script. Now it's time to give your application a try. Before doing that, make sure you've downloaded the bonus material mentioned at the beginning of this step, especially the `sample-urls.txt` file.

Now get back to your command line and execute the following commands:

#### Shell

```
$ python -m rpchecker -h
python -m rpchecker -h
usage: rpchecker [-h] [-u URLs [URLs ...]] [-f FILE] [-a]

check the availability of web sites

options:
  -h, --help            show this help message and exit
  -u URLs [URLs ...], --urls URLs [URLs ...]
                        enter one or more website URLs
  -f FILE, --input-file FILE
                        read URLs from a file

$ python -m rpchecker -u python.org pypi.org peps.python.org
The status of "python.org" is: "Online!" 🟢
The status of "pypi.org" is: "Online!" 🟢
The status of "peps.python.org" is: "Online!" 🟢

$ python -m rpchecker --urls non-existing-site.org
The status of "non-existing-site.org" is: "Offline?" 🟡
Error: "[Errno -2] Name or service not known"

$ cat sample-urls.txt
python.org
pypi.org
docs.python.org
peps.python.org

$ python -m rpchecker -f sample-urls.txt
The status of "python.org" is: "Online!" 🟢
The status of "pypi.org" is: "Online!" 🟢
The status of "docs.python.org" is: "Online!" 🟢
The status of "peps.python.org" is: "Online!" 🟢
```



Your website connectivity checker works great! When you run `rpchecker` with the `-h` or `--help` option, you get a usage message that explains how to use the app.

The application can take several URLs at the command line or from a text file and check them for connectivity. If an error occurs during the check, then you get a message on the screen with information about what's causing the error.

Go ahead and try some other URLs and features. For example, try to combine URLs at the command line with URLs from a file using the `-u` and `-f` switches. Additionally, check what happens when you provide a URLs file that's empty or nonexistent.

Cool! Your site connectivity checker app works nicely and smoothly, doesn't it? However, it has a hidden issue. The execution time can be overwhelming when you run the application with a long list of target URLs, because all the connectivity checks run synchronously.

To work around this issue and improve the application's performance, you can implement asynchronous connectivity checks. That's what you'll do in the following section.



 Remove ads

## Step 5: Check Websites' Connectivity Asynchronously

By performing the connectivity checks on multiple websites concurrently through asynchronous programming, you can improve the overall performance of your application. To do this, you can take advantage of Python's asynchronous features and the `aiohttp` third-party library, which you already have installed in your project's virtual environment.

Python supports asynchronous programming with the `asyncio`

module and the `async` and `await` keywords. In the following sections, you'll write the required code to make your app run the connectivity checks asynchronously using these tools.

To download the code for this final step, click the following link and look into the `source_code_step_5/` folder:

**Get Source Code:** Click here to get the source code you'll use to build your site connectivity checker app.

## Implement an Asynchronous Connectivity Checker Function

The first step in the process of making your website connectivity checker work concurrently is to write an `async` function that allows you to perform a single connectivity check on a given website. This will be the asynchronous equivalent to your `site_is_online()` function.

Get back to the `checker.py` file and add the following code:

Python

```
1 # checker.py
2
3 import asyncio
4 from http.client import HTTPConnection
5 from urllib.parse import urlparse
6
7 import aiohttp
8 # ...
9
10 async def site_is_online_async(url, timeout=2):
11     """Return True if the target URL is online.
12
13     Raise an exception otherwise.
14     """
15     error = Exception("unknown error")
16     parser = urlparse(url)
17     host = parser.netloc or parser.path.split("/")[0]
18     for scheme in ("http", "https"):
19         target_url = scheme + "://" + host
20         async with aiohttp.ClientSession() as session:
```

```

21         try:
22             await session.head(target_url, timeout=time
23             return True
24         except asyncio.exceptions.TimeoutError:
25             error = Exception("timed out")
26         except Exception as e:
27             error = e
28     raise error

```

In this update, you first add the required imports, `asyncio` and `aiohttp`. Then you define `site_is_online_async()` on line 10. It's an async function that takes two arguments: the URL to check and the number of seconds before the requests time out. The function's body does the following:

- **Line 15** defines a generic `Exception` instance as a placeholder.
- **Line 16** defines a parser variable containing the result of parsing the target URL using `urlparse()`.
- **Line 17** uses the `or operator` to extract the hostname from the target URL.
- **Line 18** defines a for loop over the HTTP and HTTPS schemes. This will allow you to check if the website is available on either one.
- **Line 19** builds a URL using the current scheme and the hostname.
- **Line 20** defines an `async with statement` to handle an `aiohttp.ClientSession` instance. This class is the recommended interface for making HTTP requests with `aiohttp`.
- **Lines 21 to 27** define a try ... except statement. The try block performs and awaits a HEAD request to the target website by calling `.head()` on the session object. If the request succeeds, then the function returns `True`. The first except clause catches `TimeoutError` exceptions and sets `error` to a new `Exception` instance. The second except clause catches any other

exceptions and updates the error variable accordingly.

- **Line 28** raises the exception stored in `error` if the loop finishes without a successful request.

This implementation of `site_is_online_async()` is similar to the implementation of `site_is_online()`. It returns `True` if the target website is online. Otherwise, it raises an exception pointing out the encountered problem.

The main difference between these functions is that `site_is_online_async()` performs the HTTP requests asynchronously using the `aiohttp` third-party library. This distinction can help you optimize your app's performance when you have a long list of websites to check.

With this function in place, you can proceed to update your application's CLI with a new option that allows you to run the connectivity checks asynchronously.

## Add an Asynchronous Option to the Application's CLI

Now you need to add an option to your site connectivity checker app's CLI. This new option will tell the app to run the checks asynchronously. The option can just be a [Boolean](#) flag. To implement this type of option, you can use the `action` argument of `.add_argument()`.

Now go ahead and update `read_user_cli_args()` on the `cli.py` file with the following code:

Python

```
# cli.py
# ...

def read_user_cli_args():
    """Handles the CLI user interactions."""
    # ...
    parser.add_argument(
        "-a",
```

```

        "--asynchronous",
        action="store_true",
        help="run the connectivity check asynchronously",
    )
    return parser.parse_args()

# ...

```

This call to `.add_argument()` on the parser object adds a new `-a` or `--asynchronous` option to the application's CLI. The `action` argument is set to `"store_true"`, which tells `argparse` that `-a` and `--asynchronous` are Boolean flags that will store `True` when provided at the command line.

With this new option in place, it's time to write the logic for checking the connectivity of multiple websites asynchronously.

**Write Cleaner & More Pythonic Code**

realpython.com



Remove ads

## Check the Connectivity of Multiple Websites Asynchronously

To check the connectivity of multiple websites asynchronously, you'll write an `async` function that calls and awaits `site_is_online_async()` from the `checker` module. Get back to the `__main__.py` file and add the following code to it:

Python

```

1  # __main__.py
2  import asyncio
3  import pathlib
4  import sys
5
6  from rpchecker.checker import site_is_online, site_is_onlin
7  from rpchecker.cli import display_check_result, read_user_c
8  # ...
9
10 async def _asynchronous_check(urls):
11     async def _check(url):

```

```

12         error = ""
13         try:
14             result = await site_is_online_async(url)
15         except Exception as e:
16             result = False
17             error = str(e)
18         display_check_result(result, url, error)
19
20     await asyncio.gather(*(_check(url) for url in urls))
21
22 def _synchronous_check(urls):
23     # ...

```

In this piece of code, you first update your imports to access `site_is_online_async()`. Then you define `_asynchronous_check()` on line 10 as an asynchronous function using the `async` keyword. This function takes a list of URLs and checks their connectivity asynchronously. Here's how it does that:

- **Line 11** defines an `inner` async function called `_check()`. This function allows you to reuse the code that checks a single URL for connectivity.
- **Line 12** defines and initializes a placeholder error variable, which will be used in the call to `display_check_result()` later.
- **Lines 13 to 17** define a `try ... except` statement to wrap the connectivity check. The `try` block calls and awaits `site_is_online_async()` with the target URL as an argument. If the call succeeds, then `result` ends up being `True`. If the call raises an exception, then `result` will be `False`, and `error` will hold the resulting error message.
- **Line 18** calls `display_check_result()` using `result`, `url`, and `error` as arguments. This call displays information about the website's availability.
- **Line 20** calls and awaits the `gather()` function from the `asyncio` module. This function runs a list of `awaitable objects` concurrently and returns an aggregated list of resulting values if all the awaitable objects complete successfully. To provide

the list of awaitable objects, you use a [generator expression](#) that calls `_check()` for each target URL.

Okay! You're almost ready to try out the asynchronous capabilities of your site connectivity checker app. Before doing that, you need to take care of a final detail: updating the `main()` function to integrate this new feature.

## Add Asynchronous Checks to the App's Main Code

To add the asynchronous functionality to your application's `main()` function, you'll use a conditional statement to check if the user provided the `-a` or `--asynchronous` flag at the command line. This conditional will allow you to run the connectivity checks with the right tool according to the user's input.

Go ahead and open the `__main__.py` file again. Then update `main()` like in the following code snippet:

Python

```
1 # __main__.py
2 # ...
3
4 def main():
5     """Run RP Checker."""
6     user_args = read_user_cli_args()
7     urls = _get_websites_urls(user_args)
8     if not urls:
9         print("Error: no URLs to check", file=sys.stderr)
10        sys.exit(1)
11
12     if user_args.asynchronous:
13         asyncio.run(_asynchronous_check(urls))
14     else:
15         _synchronous_check(urls)
16
17 # ...
```

The conditional statement on lines 12 to 15 checks if the user has provided the `-a` or `--asynchronous` flag at the command line. If that's

the case, `main()` runs the connectivity checks asynchronously using `asyncio.run()`. Otherwise, it runs the checks synchronously using `_synchronous_check()`.

That's it! You can now test this new feature of your website connectivity checker in practice. Get back to your command line and run the following:

#### Shell

```
$ python -m rpchecker -h
usage: rpchecker [-h] [-u URLs [URLs ...]] [-f FILE] [-a]

check the availability of web sites

options:
  -h, --help            show this help message and exit
  -u URLs [URLs ...], --urls URLs [URLs ...]
                        enter one or more website URLs
  -f FILE, --input-file FILE
                        read URLs from a file
  -a, --asynchronous    run the connectivity check asynchronously
```

```
$ # Synchronous execution
$ python -m rpchecker -u python.org pypi.org docs.python.org
The status of "python.org" is: "Online!" 👍
The status of "pypi.org" is: "Online!" 👍
The status of "docs.python.org" is: "Online!" 👍
```

```
$ # Asynchronous execution
$ python -m rpchecker -u python.org pypi.org docs.python.org -a
The status of "pypi.org" is: "Online!" 👍
The status of "docs.python.org" is: "Online!" 👍
The status of "python.org" is: "Online!" 👍
```

The first command shows that your application now has a new `-a` or `--asynchronous` option that will run the connectivity checks asynchronously.

The second command makes `rpchecker` run the connectivity checks synchronously, just like you did in the previous section. This is because you don't provide the `-a` or `--asynchronous` flags. Note that the URLs are checked in the same order that they're entered at the command line.



Finally, in the third command, you use the `-a` flag at the end of the line. This flag makes `rpchecker` run the connectivity checks concurrently. Now the check results aren't displayed in the same order as the URLs are entered but in the order in which the responses come from the target websites.

As an exercise, you can try to run your site connectivity checker application with a long list of target URLs and compare the execution time when the app runs the checks synchronously and asynchronously.

## Conclusion

You've built a functional site connectivity checker application in Python. Now you know the basics of handling **HTTP requests** to a given website. You also learned how to create a minimal yet functional **command-line interface (CLI)** for your application and how to organize a real-world Python project. Additionally, you've tried out Python's asynchronous features.

### In this tutorial, you learned how to:

- Create command-line interfaces (CLI) in Python with `argparse`
- Check if a website is online using Python's `http.client`
- Run **synchronous** checks on multiple websites
- Check if a website is online using `aiohttp`
- Check the connectivity of multiple websites **asynchronously**

With this foundation, you're ready to continue leveling up your skills by creating more complex command-line applications. You're also better prepared to continue learning about HTTP requests with Python.

To review what you've done to build your app, you can download the complete source code below:

**Get Source Code:** [Click here to get the source code you'll use to build your site connectivity checker app.](#)



 Remove ads

## Next Steps

Now that you've finished building your site connectivity checker application, you can go a step further by implementing a few additional features. Adding new features by yourself will push you to learn about new and exciting coding concepts and topics.

Here are some ideas for new features:

- **Timing support:** Measure the response time of every target website.
- **Check-scheduling support:** Schedule multiple rounds of connectivity checks in case some websites are offline.

To implement these features, you can take advantage of Python's [time](#) module, which will allow you to measure the execution time of your code.

Once you implement these new features, then you can change gears and jump into other cool and more complex projects. Here are some great next steps for you to continue learning Python and building projects:

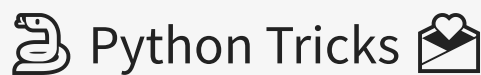
- [Build a Dice-Rolling Application With Python](#): In this step-by-step project, you'll build a dice-rolling simulator app with a minimal text-based user interface using Python. The app will simulate the rolling of up to six dice. Each individual die will have six sides.
- [Raining Outside? Build a Weather CLI App With Python](#): In this tutorial, you'll write a nicely formatted Python CLI app that displays information about the current weather in any city that you provide the name for.

- [Build a Python Directory Tree Generator for the Command Line](#): In this step-by-step project, you'll create a Python directory tree generator application for your command line. You'll code the command-line interface with `argparse` and traverse the file system using `pathlib`.
- [Build a Command-Line To-Do App With Python and Typer](#): In this step-by-step project, you'll create a to-do application for your command line using Python and `Typer`. While you build this app, you'll learn the basics of `Typer`, a modern and versatile library for building command-line interfaces (CLIs).

Mark as Completed



[▶ Watch Now](#) This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Building a Site Connectivity Checker](#)



Get a short & sweet **Python Trick** delivered to your inbox every couple of days. No spam ever. Unsubscribe any time. Curated by the Real Python team.

```
1# How to merge two dicts
2# in Python 3.5+
3
4>>> x = {'a': 1, 'b': 2}
5>>> y = {'b': 3, 'c': 4}
6
7>>> z = {**x, **y}
8
9>>> z
10{'c': 4, 'a': 1, 'b': 3}
```

Email Address

Send Me Python Tricks »

## About **Leodanis Pozo Ramos**



Leodanis is an industrial engineer who loves Python and software development. He's a self-taught Python developer with 6+ years of experience. He's an avid technical writer with a growing number of articles published on Real Python and other sites.

[» More about Leodanis](#)

*Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:*



Aldren



Geir  
Arne



Kate



Martin



Philipp

## Master Real-World Python Skills With Unlimited Access to Real Python





Join us and get access to thousands of tutorials,  
hands-on video courses, and a community of  
expert Pythonistas:

Level Up Your Python Skills »

## What Do You Think?

Rate this article:



Tweet

Share

Share

Email

What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

**Commenting Tips:** The most useful comments are those written with the goal of learning from or helping out other students. [Get tips for asking good questions](#) and [get answers to common questions in our support portal](#).

Looking for a real-time conversation? Visit the [Real Python Community Chat](#) or join the next “Office Hours” [Live Q&A Session](#). Happy Pythoning!

## Keep Learning

Related Tutorial Categories: [intermediate](#) [projects](#) [python](#)

Recommended Video Course: [Building a Site Connectivity Checker](#)

— FREE Email Series —



```
1 # How to merge two dicts
2 # in Python 3.5+
3
4 >>> x = {'a': 1, 'b': 2}
5 >>> y = {'b': 3, 'c': 4}
6
7 >>> z = {**x, **y}
8
9 >>> z
10 {'c': 4, 'a': 1, 'b': 3}
```

[Get Python Tricks »](#)

No spam. Unsubscribe any time.

## All Tutorial Topics

[advanced](#) [api](#) [basics](#) [best-practices](#) [community](#) [databases](#)  
[data-science](#) [devops](#) [django](#) [docker](#) [flask](#) [front-end](#) [gamedev](#) [gui](#)  
[intermediate](#) [machine-learning](#) [projects](#) [python](#) [testing](#) [tools](#)  
[web-dev](#) [web-scraping](#)

Python Data Connectors

# Python Data Connectors

Connect to 250+ SaaS, NoSQL, & Big Data sources  
from pandas, SQLAlchemy, Dash, petl, and more!

[Learn More](#)

## Table of Contents

- [Demo: A Site Connectivity Checker](#)
- [Project Overview](#)
- [Prerequisites](#)
- [Step 1: Set Up Your Site Connectivity Checker Project in Python](#)
- [Step 2: Check Websites' Connectivity in Python](#)
- [Step 3: Create Your Website Connectivity Checker's CLI](#)
- [Step 4: Put Everything Together in the App's Main Script](#)
- [Step 5: Check Websites' Connectivity Asynchronously](#)
- [Conclusion](#)
- [Next Steps](#)

[Mark as Completed](#)[Tweet](#)[Share](#)[Email](#)[▶ Recommended Video Course](#)[Building a Site Connectivity Checker](#)



Your **Practical Introduction to Python 3** »

 [Remove ads](#)

© 2012–2022 Real Python · [Newsletter](#) ·  
[Podcast](#) · [YouTube](#) · [Twitter](#) · [Facebook](#) ·  
[Instagram](#) · [Python Tutorials](#) · [Search](#) · [Privacy](#)  
[Policy](#) · [Energy Policy](#) · [Advertise](#) · [Contact](#)  
♥ Happy Pythoning!