

Python's urllib.request for HTTP Requests

by Ian Currie 4 Comments [api](#) [intermediate](#) [web-dev](#) [web-scraping](#)

Mark as Completed



[Tweet](#)

[Share](#)

[Email](#)

Table of Contents

- [Basic HTTP GET Requests With urllib.request](#)
- [The Nuts and Bolts of HTTP Messages](#)
 - [Understanding What an HTTP Message Is](#)

— FREE Email Series —



Python Tricks

```
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...

[Get Python Tricks »](#)



No spam. Unsubscribe any time.

[Browse Topics](#)

[Guided Learning Paths](#)

[Basics](#)

[Intermediate](#)

[Advanced](#)

[api](#) [best-practices](#) [community](#)

[databases](#) [data-science](#) [devops](#)

[django](#) [docker](#) [flask](#) [front-end](#)

[gamedev](#) [gui](#) [machine-learning](#)

[projects](#) [python](#) [testing](#) [tools](#)

[web-dev](#) [web-scraping](#)

- [Understanding How urllib.request Represents an HTTP Message](#)
- [Closing an HTTPResponse](#)
- [Exploring Text, Octets, and Bits](#)
- [Dealing With Character Encodings](#)
- [Going From Bytes to Strings](#)
- [Going From Bytes to File](#)
- [Going From Bytes to Dictionary](#)
- [Common urllib.request Troubles](#)
 - [Implementing Error Handling](#)
 - [Dealing With 403 Errors](#)
 - [Fixing the SSL CERTIFICATE_VERIFY_FAILED Error](#)
- [Authenticated Requests](#)
- [POST Requests With urllib.request](#)
- [The Request Package Ecosystem](#)
 - [What Are urllib2 and urllib3?](#)
 - [When Should I Use requests Over urllib.request?](#)
 - [Why Is requests Not Part of the Standard Library?](#)
- [Conclusion](#)



**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

If you need to make HTTP requests with Python, then you may find yourself directed to the brilliant [requests](#) library. Though it's a great library, you may have noticed that it's not a built-in part of Python. If you prefer, for whatever reason, to limit your dependencies and stick to standard-library Python, then you can reach for `urllib.request`!

In this tutorial, you'll:

- Learn how to make basic **HTTP requests** with `urllib.request`



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

Table of Contents

- [Basic HTTP GET Requests With urllib.request](#)
- [The Nuts and Bolts of HTTP Messages](#)
- [Common urllib.request Troubles](#)
- [Authenticated Requests](#)
- [POST Requests With urllib.request](#)
- [The Request Package Ecosystem](#)
- [Conclusion](#)

Mark as Completed



Tweet

Share

Email

Learn how to make basic [HTTP requests](#) with `urllib.request`

- Dive into the nuts and bolts of an **HTTP message** and how `urllib.request` represents it
- Understand how to deal with **character encodings** of HTTP messages
- Explore some **common errors** when using `urllib.request` and learn how to resolve them
- Dip your toes into the world of **authenticated requests** with `urllib.request`
- Understand why both `urllib` and the `requests` library exist and **when to use one or the other**

If you've heard of HTTP requests, including [GET](#) and [POST](#), then you're probably ready for this tutorial. Also, you should've already used Python to [read and write to files](#), ideally with a [context manager](#), at least once.

Ultimately, you'll find that making a request doesn't have to be a frustrating experience, although it does tend to have that reputation. Many of the issues that you tend to run into are due to the inherent complexity of this marvelous thing called the Internet. The good news is that the `urllib.request` module can help to demystify much of this complexity.

Learn More: Click here to join 290,000+ Python developers on the Real Python Newsletter and get new Python tutorials and news that will make you a more effective Pythonista.

Basic HTTP GET Requests With `urllib.request`

Before diving into the deep end of what an HTTP request is and how it works, you're going to get your feet wet by making a basic GET request to a [sample URL](#). You'll also make a GET request to a mock [REST API](#) for some [JSON](#) data. In case you're wondering about POST Requests, you'll be covering them [later in the tutorial](#), once you have some more knowledge of `urllib.request`.

Beware: Depending on your exact setup, you may find that some of these examples don't work. If so, skip ahead to the section on common [urllib.request errors](#) for troubleshooting.

If you're running into a problem that's not covered there, be sure to comment below with a precise and reproducible example.

To get started, you'll make a request to `www.example.com`, and the server will return an HTTP message. Ensure that you're using Python 3 or above, and then use the `urlopen()` function from `urllib.request`:

```
Python >>>
>>> from urllib.request import urlopen
>>> with urlopen("https://www.example.com") as response:
...     body = response.read()
...
>>> body[:15]
b'<!doctype html>'
```

In this example, you import `urlopen()` from `urllib.request`. Using the [context manager](#) `with`, you make a request and receive a response with `urlopen()`. Then you read the body of the response and close the response object. With that, you display the first fifteen positions of the body, noting that it looks like an HTML document.

There you are! You've successfully made a request, and you received a response. By inspecting the content, you can tell that it's likely an HTML document. Note that the printed output of the body is preceded by `b`. This indicates a [bytes literal](#), which you may need to decode. Later in the tutorial, you'll learn how to turn bytes into a [string](#), write them to a [file](#), or parse them into a [dictionary](#).

The process is only slightly different if you want to make calls to REST APIs to get JSON data. In the following example, you'll make a request to [{JSON} Placeholder](#) for some fake

to-do data:

Python

>>>

```
>>> from urllib.request import urlopen
>>> import json
>>> url = "https://jsonplaceholder.typicode.com/todos/1"
>>> with urlopen(url) as response:
...     body = response.read()
...
>>> todo_item = json.loads(body)
>>> todo_item
{'userId': 1, 'id': 1, 'title': 'delectus aut autem', 'completed': False}
```

In this example, you're doing pretty much the same as in the previous example. But in this one, you import `urllib.request` *and* `json`, using the `json.loads()` function with `body` to decode and parse the returned JSON bytes into a [Python dictionary](#). Voila!

If you're lucky enough to be using error-free [endpoints](#), such as the ones in these examples, then maybe the above is all that you need from `urllib.request`. Then again, you may find that it's not enough.

Now, before doing some `urllib.request` troubleshooting, you'll first gain an understanding of the underlying structure of HTTP messages and learn how `urllib.request` handles them. This understanding will provide a solid foundation for troubleshooting many different kinds of issues.



[Become a Python Expert »](#)

Remove ads

The Nuts and Bolts of HTTP Messages

The rules and bits of HTTP messages

To understand some of the issues that you may encounter when using `urllib.request`, you'll need to examine how a response is represented by `urllib.request`. To do that, you'll benefit from a high-level overview of what an **HTTP message** is, which is what you'll get in this section.

Before the high-level overview, a quick note on reference sources. If you want to get into the technical weeds, the [Internet Engineering Task Force \(IETF\)](#) has an extensive set of [Request for Comments \(RFC\)](#) documents. These documents end up becoming the actual specifications for things like HTTP messages. [RFC 7230, part 1: Message Syntax and Routing](#), for example, is all about the HTTP message.

If you're looking for some reference material that's a bit easier to digest than RFCs, then the [Mozilla Developer Network \(MDN\)](#) has a great range of reference articles. For example, their article on [HTTP messages](#), while still technical, is a lot more digestible.

Now that you know about these essential sources of reference information, in the next section you'll get a beginner-friendly overview of HTTP messages.

Understanding What an HTTP Message Is

In a nutshell, an HTTP message can be understood as text, transmitted as a stream of [bytes](#), structured to follow the guidelines specified by RFC 7230. A decoded HTTP message can be as simple as two lines:

Text

```
GET / HTTP/1.1
Host: www.google.com
```

This specifies a [GET](#) request at the root (/) using the HTTP/1.1 protocol. The one and only [header](#) required is the host, `www.google.com`. The target server has enough information to make a response with this information.

A response is similar in structure to a request. HTTP messages have two main parts, the **metadata** and the **body**. In the request example above, the message is all metadata with no body. The response, on the other hand, does have two parts:

Text

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=ISO-8859-1
Server: gws
(... other headers ...)

<!doctype html><html itemscope="" itemtype="http://schema.org/WebPage"
...
```

The response starts with a **status line** that specifies the HTTP protocol HTTP/1.1 and the status 200 OK. After the status line, you get many key-value pairs, such as `Server: gws`, representing all the response **headers**. This is the metadata of the response.

After the metadata, there's a blank line, which serves as the divider between the headers and the body. Everything that follows the blank line makes up the body. This is the part that gets read when you're using `urllib.request`.

Note: Blank lines are often technically referred to as **newlines**. A newline in an HTTP message has to be a Windows-style **carriage return** (`\r`) together with a **line ending** (`\n`). On **Unix-like** systems, newlines are typically just a line ending (`\n`).

You can assume that all HTTP messages follow these specifications, but it's possible that some may break these rules or follow an older specification. It's *exceptionally* rare for this to cause any issues, though. So, just keep it in the back of your mind in case you run into a strange bug!

In the next section, you'll see how `urllib.request` deals with raw HTTP messages.

Understanding How `urllib.request` Represents an HTTP Message

The main representation of an HTTP message that you'll be interacting with when using `urllib.request` is the `HTTPResponse` object. The `urllib.request` module itself depends on the low-level `http` module, which you don't need to interact with directly. You do end up using some of the data structures that `http` provides, though, such as `HTTPResponse` and `HTTPMessage`.

Note: The internal naming of objects representing HTTP responses and messages in Python can be a bit confusing. You generally only interact with instances of `HTTPResponse`, while the *request* end of things is taken care of internally.

You might think that `HTTPMessage` is a sort of base class, which `HTTPResponse` inherits from, but it's not. `HTTPResponse` inherits directly from `io.BufferedIOBase`, while the `HTTPMessage` class inherits from `email.message.EmailMessage`.

The `EmailMessage` is defined in the source code as an object that contains a bunch of headers and a payload, so it doesn't necessarily have to be an email. `HTTPResponse` simply uses `HTTPMessage` as a container for its headers.

However, if you're talking about HTTP itself and not its Python implementation, then you'd be right to think about an HTTP response as a kind of HTTP message.

When you make a request with `urllib.request.urlopen()`, you get an `HTTPResponse` object in return. Spend some time exploring the `HTTPResponse` object with `pprint()` and `dir()` to see all the different methods and properties that belong to it:

Python

>>>

```
>>> from urllib.request import urlopen
>>> from pprint import pprint
>>> with urlopen("https://www.example.com") as response:
...     pprint(dir(response))
...
```

To reveal the output of this code snippet, click to expand the collapsible section below:

Output

Show/Hide

That's a lot of methods and properties, but you'll only end up using a handful of these . Apart from `.read()`, the important ones usually involve getting information about the **headers**.

One way to inspect all the headers is to access the `.headers` attribute of the `HTTPResponse` object. This will return an `HTTPMessage` object. Conveniently, you can treat an `HTTPMessage` like a dictionary by calling `.items()` on it to get all the headers as tuples:

Python

>>>

```
>>> with urlopen("https://www.example.com") as response:
...     pass
...
>>> response.headers
<http.client.HTTPMessage object at 0x000001E029D9F4F0>
>>> pprint(response.headers.items())
[('Accept-Ranges', 'bytes'),
 ('Age', '398424'),
 ('Cache-Control', 'max-age=604800'),
 ('Content-Type', 'text/html; charset=UTF-8'),
 ('Date', 'Tue, 25 Jan 2022 12:18:53 GMT'),
 ('Etag', '"3147526947"'),
 ('Expires', 'Tue, 01 Feb 2022 12:18:53 GMT'),
 ('Last-Modified', 'Thu, 17 Oct 2019 07:18:26 GMT'),
 ('Server', 'ECS (nyb/1D16)'),
 ('Vary', 'Accept-Encoding'),
 ('X-Cache', 'HIT'),
 ('Content-Length', '1256'),
 ('Connection', 'close')]
```

Now you have access to all the response headers! You probably won't need most of this information, but rest assured that some applications do use it. For example, your browser might use the headers to read the response, set cookies, and determine an appropriate [cache](#) lifetime.

There are convenience methods to get the headers from an `HTTPResponse` object because it's quite a common operation. You can call `.getheaders()` directly on the `HTTPResponse` object, which will return exactly the same list of tuples as above. If you're only interested in one header, say the `Server` header, then you can use the singular `.getheader("Server")` on `HTTPResponse` or use the square bracket `[]` syntax on `.headers` from `HTTPMessage`:

Python

>>>

```
>>> response.getheader("Server")  
'ECS (nyb/1D16)'  
>>> response.headers["Server"]  
'ECS (nyb/1D16)'
```

Truth be told, you probably won't need to interact with the headers directly like this. The information that you're most likely to need will probably already have some built-in helper methods, but now you know, in case you ever need to dig deeper!



[Learn Python »](#)

 Remove ads

Closing an HTTPResponse

The HTTPResponse object has a lot in common with the [file object](#). The HTTPResponse class inherits from the [IOBase class](#), as do file objects, which means that you have to be mindful of opening and closing.

In simple programs, you're not likely to notice any issues if you forget to close HTTPResponse objects. For more complex projects, though, this can significantly slow execution and cause bugs that are difficult to pinpoint.

Problems arise because [input/output](#) (I/O) streams are limited. Each HTTPResponse requires a stream to be kept clear while it's being read. If you never close your streams, this will eventually prevent any other stream from being opened, and it might interfere with other programs or even your operating system.

So, make sure you close your HTTPResponse objects! For your convenience, you can use a context manager, as you've seen in the examples. You can also achieve the same result by explicitly calling `.close()` on the response object:

Python

>>>

```
>>> from urllib.request import urlopen
>>> response = urlopen("https://www.example.com")
>>> body = response.read()
>>> response.close()
```

In this example, you don't use a context manager, but instead close the response stream explicitly. The above example still has an issue, though, because an exception may be raised before the call to `.close()`, preventing the proper teardown. To make this call unconditional, as it should be, you can use a `try ... except` block with both an `else` and a `finally` clause:

Python

>>>

```
>>> from urllib.request import urlopen
>>> response = None
>>> try:
...     response = urlopen("https://www.example.com")
... except Exception as ex:
...     print(ex)
... else:
...     body = response.read()
... finally:
...     if response is not None:
...         response.close()
```

In this example, you achieve an unconditional call to `.close()` by using the `finally` block, which will always run regardless of exceptions raised. The code in the `finally` block first checks if the response object exists with `is not None`, and then closes it.

That said, this is exactly what a context manager does, and the `with` syntax is generally preferred. Not only is the `with` syntax less verbose and more readable, but it also protects

you from pesky errors of omission. Put another way, it's a far better guard against accidentally forgetting to close the object:

Python

>>>

```
>>> from urllib.request import urlopen
>>> with urlopen("https://www.example.com") as response:
...     response.read(50)
...     response.read(50)
...
b'<!doctype html>\n<html>\n<head>\n    <title>Example D'
b'omain</title>\n\n    <meta charset="utf-8" />\n    <m'
```

In this example, you import `urlopen()` from the `urllib.request` module. You use the `with` keyword with `.urlopen()` to assign the `HTTPResponse` object to the variable `response`. Then, you read the first fifty bytes of the response and then read the following fifty bytes, all within the `with` block. Finally, you close the `with` block, which executes the request and runs the lines of code within its block.

With this code, you cause two sets of fifty bytes each to be displayed. The `HTTPResponse` object will close once you exit the `with` block scope, meaning that the `.read()` method will only return empty bytes objects:

Python

>>>

```
>>> import urllib.request
>>> with urllib.request.urlopen("https://www.example.com") as response:
...     response.read(50)
...
b'<!doctype html>\n<html>\n<head>\n    <title>Example D'
>>> response.read(50)
b''
```

In this example, the second call to read fifty bytes is outside the `with` scope. Being outside the `with` block means that `HTTPResponse` is closed, even though you can still access the

variable. If you try to read from `HTTPResponse` when it's closed, it'll return an empty bytes object.

Another point to note is that you can't reread a response once you've read all the way to the end:

Python

>>>

```
>>> import urllib.request
>>> with urllib.request.urlopen("https://www.example.com") as response:
...     first_read = response.read()
...     second_read = response.read()
...
>>> len(first_read)
1256
>>> len(second_read)
0
```

This example shows that once you've read a response, you can't read it again. If you've fully read the response, the subsequent attempt just returns an empty bytes object even though the response isn't closed. You'd have to make the request again.

In this regard, the response is different from a file object, because with a file object, you can read it multiple times by using the `.seek()` method, which `HTTPResponse` doesn't support. Even after closing a response, you can still access the headers and other metadata, though.

Exploring Text, Octets, and Bits

In most of the examples so far, you read the response body from `HTTPResponse`, displayed the resulting data immediately, and noted that it was displayed as a `bytes object`. This is because text information in computers isn't stored or transmitted as letters, but as bytes!

A raw HTTP message sent over the wire is broken up into a sequence of [bytes](#), sometimes referred to as [octets](#). Bytes are 8-bit chunks. For example, 01010101 is a byte. To learn more about binary, bits, and bytes, check out [Bitwise Operators in Python](#).

So how do you represent letters with bytes? A byte has 256 potential combinations, and you can assign a letter to each combination. You can assign 00000001 to A, 00000010 to B, and so on. [ASCII](#) character encoding, which is quite common, uses this type of system to encode 128 characters, which is enough for a language like English. This is particularly convenient because just one byte can represent all the characters, with space to spare.

All the standard English characters, including capitals, punctuation, and numerals, fit within ASCII. On the other hand, Japanese is thought to have around fifty thousand logographic characters, so 128 characters won't cut it! Even the 256 characters that are theoretically available within one byte wouldn't be nearly enough for Japanese. So, to accomodate all the world's languages there are many different systems to encode characters.

Even though there are many systems, one thing you can rely on is the fact that they'll always be broken up into **bytes**. Most servers, if they can't resolve the [MIME type](#) and character encoding, default to application/octet-stream, which literally means a stream of bytes. Then whoever receives the message can work out the character encoding.

Python Tricks The Book

A Buffet of Awesome Python Features

[Get Your Free Sample Chapter](#)



 [Remove ads](#)

Dealing With Character Encodings

Problems often arise because, as you may have guessed, there are many, many different potential character encodings. The dominant character encoding today is [UTF-8](#), which is an implementation of [Unicode](#). Luckily, [ninety-eight percent of web pages](#) today are encoded in

UTF-8!

UTF-8 is dominant because it can efficiently handle a mind-boggling number of characters. It handles all the 1,112,064 potential characters defined by Unicode, encompassing Chinese, Japanese, Arabic (with right-to-left scripts), Russian, and many more character sets, including [emojis](#)!

UTF-8 remains efficient because it uses a variable number of bytes to encode characters, which means that for many characters, it only requires one byte, while for others it can require up to four bytes.

Note: To learn more about encodings in Python, check out [Unicode & Character Encodings in Python: A Painless Guide](#).

While UTF-8 is dominant, and you usually won't go wrong with assuming UTF-8 encodings, you'll still run into different encodings all the time. The good news is that you don't need to be an expert on encodings to handle them when using `urllib.request`.

Going From Bytes to Strings

When you use `urllib.request.urlopen()`, the body of the response is a bytes object. The first thing you may want to do is to convert the bytes object to a string. Perhaps you want to do some [web scraping](#). To do this, you need to **decode** the bytes. To decode the bytes with Python, all you need to find out is the **character encoding** used. Encoding, especially when referring to character encoding, is often referred to as a **character set**.

As mentioned, ninety-eight percent of the time, you'll probably be safe defaulting to UTF-8:

Python

>>>

```
>>> from urllib.request import urlopen
>>> with urlopen("https://www.example.com") as response:
...     body = response.read()
...
>>> decoded_body = body.decode("utf-8")
>>> print(decoded_body[:30])
<!doctype html>
<html>
<head>
```

In this example, you take the bytes object returned from `response.read()` and decode it with the bytes object's `.decode()` method, passing in `utf-8` as an argument. When you `print` `decoded_body`, you can see that it's now a string.

That said, leaving it up to chance is rarely a good strategy. Fortunately, headers are a great place to get character set information:

Python

>>>

```
>>> from urllib.request import urlopen
>>> with urlopen("https://www.example.com") as response:
...     body = response.read()
...
>>> character_set = response.headers.get_content_charset()
>>> character_set
'utf-8'
>>> decoded_body = body.decode(character_set)
>>> print(decoded_body[:30])
<!doctype html>
<html>
<head>
```

In this example, you call `.get_content_charset()` on the `.headers` object of response and use that to decode. This is a convenience method that parses the Content-Type header so that you can painlessly decode bytes into text.

Going From Bytes to File

If you want to decode bytes into text, now you're good to go. But what if you want to write the body of a response into a file? Well, you have two options:

1. Write the bytes directly to the file
2. Decode the bytes into a Python string, and then encode the string back into a file

The first method is the most straightforward, but the second method allows you to change the encoding if you want to. To learn about file manipulation in more detail, take a look at Real Python's [Reading and Writing Files in Python \(Guide\)](#).

To write the bytes directly to a file without having to decode, you'll need the built-in `open()` function, and you'll need to ensure that you use write binary mode:

Python

>>>

```
>>> from urllib.request import urlopen
>>> with urlopen("https://www.example.com") as response:
...     body = response.read()
...
>>> with open("example.html", mode="wb") as html_file:
...     html_file.write(body)
...
1256
```

Using `open()` in `wb` mode bypasses the need to decode or encode and dumps the bytes of the HTTP message body into the `example.html` file. The number that's output after the writing operation indicates the number of bytes that have been written. That's it! You've written the bytes directly to a file without encoding or decoding anything.

Now say you have a URL that doesn't use UTF-8, but you want to write the contents to a

file with UTF-8. For this, you'd first decode the bytes into a string and then encode the string into a file, specifying the character encoding.

Google's home page seems to use different encodings depending on your location. In much of Europe and the US, it uses the [ISO-8859-1](#) encoding:

Python

>>>

```
>>> from urllib.request import urlopen
>>> with urlopen("https://www.google.com") as response:
...     body = response.read()
...
>>> character_set = response.headers.get_content_charset()
>>> character_set
'ISO-8859-1'
>>> content = body.decode(character_set)
>>> with open("google.html", encoding="utf-8", mode="w") as file:
...     file.write(content)
...
14066
```

In this code, you got the response character set and used it to decode the bytes object into a string. Then you wrote the string to a file, encoding it using UTF-8.

Note: Interestingly, Google seems to have various layers of checks that are used to determine what language and encoding to serve the web page in. This means that you can specify an [Accept-Language header](#), which seems to override your IP location. Try it out with different [Locale Identifiers](#) to see what encodings you can get!

Once you've written to a file, you should be able to open the resulting file in your browser or text editor. Most modern text processors can detect the character encoding automatically.

If there are encoding errors and you're using Python to read a file, then you'll likely get an error:

Python

>>>

```
>>> with open("encoding-error.html", mode="r", encoding="utf-8") as file:
...     lines = file.readlines()
...
UnicodeDecodeError:
'utf-8' codec can't decode byte
```

Python explicitly stops the process and raises an exception, but in a program that displays text, such as the browser where you're viewing this page, you may find the infamous [replacement characters](#):



A Replacement Character

The black rhombus with a white question mark (◆), the square (□), and the rectangle (▭) are often used as replacements for characters which couldn't be decoded.

Sometimes, decoding seems to work but results in unintelligible sequences, such as æ-#å-åÆ-ã #', which also suggests the wrong character set was used. In Japan, they even have a word for text that's garbled due to character encoding issues, [Mojibake](#), because these issues plagued them at the start of the Internet age.

With that, you should now be equipped to write files with the raw bytes returned from `urlopen()`. In the next section, you'll learn how to parse bytes into a Python dictionary with the `json` module.

Write Cleaner & More Pythonic Code

realpython.com



 Remove ads

Going From Bytes to Dictionary

For `application/json` responses, you'll often find that they don't include any encoding information:

Python

>>>

```
>>> from urllib.request import urlopen
>>> with urlopen("https://httpbin.org/json") as response:
...     body = response.read()
...
>>> character_set = response.headers.get_content_charset()
>>> print(character_set)
None
```

In this example, you use the `json` endpoint of [httpbin](https://httpbin.org/json), a service that allows you to experiment with different types of requests and responses. The `json` endpoint simulates a typical API that returns JSON data. Note that the `.get_content_charset()` method returns nothing in its response.

Even though there's no character encoding information, all is not lost. According to [RFC 4627](https://tools.ietf.org/html/rfc4627), the default encoding of UTF-8 is an *absolute requirement* of the `application/json` specification. That's not to say that every single server plays by the rules, but generally, you can assume that if JSON is being transmitted, it'll almost always be encoded using UTF-8.

Fortunately, `json.loads()` decodes byte objects under the hood and even has some leeway in terms of different [encodings](#) that it can deal with. So, `json.loads()` should be able to cope with most bytes objects that you throw at it, as long as they're valid JSON:

Python

>>>

```
>>> import json
>>> json.loads(body)
{'slideshow': {'author': 'Yours Truly', 'date': 'date of publication', 'slides': [{'title': 'Wake up to WonderWidgets!', 'type': 'all'}, {'items': ['Why <em>WonderWidgets</em> are great', 'Who <em>buys</em> WonderWidgets'], 'title': 'Overview', 'type': 'all'}], 'title': 'Sample Slide Show'}}
```

As you can see, the `json` module handles the decoding automatically and produces a Python dictionary. Almost all APIs return key-value information as JSON, although you might run into some older APIs that work with [XML](#). For that, you might want to look into the [Roadmap to XML Parsers in Python](#).

With that, you should know enough about bytes and encodings to be dangerous! In the next section, you'll learn how to troubleshoot and fix a couple of common errors that you might run into when using `urllib.request`.

Common `urllib.request` Troubles

There are many kinds of issues you can run into on the world *wild* web, whether you're using `urllib.request` or not. In this section, you'll learn how to deal with a couple of the most common errors when getting started out: **403 errors** and **TLS/SSL certificate errors**. Before looking at these specific errors, though, you'll first learn how to implement **error handling** more generally when using `urllib.request`.

Implementing Error Handling

Before you turn your attention to specific errors, boosting your code's ability to gracefully

Before you turn your attention to specific errors, boosting your code's ability to gracefully deal with assorted errors will pay off. Web development is plagued with errors, and you can

invest a lot of time in handling errors sensibly. Here, you'll learn to handle HTTP, URL, and timeout errors when using `urllib.request`.

[HTTP status codes](#) accompany every response in the status line. If you can read a status code in the response, then the request reached its target. While this is good, you can only consider the request a complete success if the response code starts with a 2. For example, 200 and 201 represent successful requests. If the status code is 404 or 500, for example, something went wrong, and `urllib.request` will raise an [HTTPError](#).

Sometimes mistakes happen, and the URL provided isn't correct, or a connection can't be made for another reason. In these cases, `urllib.request` will raise a [URLError](#).

Finally, sometimes servers just don't respond. Maybe your network connection is slow, the server is down, or the server is programmed to ignore specific requests. To deal with this, you can pass a `timeout` argument to `urlopen()` to raise a [TimeoutError](#) after a certain amount of time.

The first step in handling these exceptions is to catch them. You can catch errors produced within `urlopen()` with a `try ... except` block, making use of the `HTTPError`, `URLError`, and `TimeoutError` classes:

Python

```
# request.py

from urllib.error import HTTPError, URLError
from urllib.request import urlopen

def make_request(url):
    try:
        with urlopen(url, timeout=10) as response:
            print(response.status)
            return response.read(), response
    except HTTPError as error:
        print(error.status, error.reason)
    except URLError as error:
        print(error.reason)
    except TimeoutError:
        print("Request timed out")
```

The function `make_request()` takes a URL string as an argument, tries to get a response from that URL with `urllib.request`, and catches the `HTTPError` object that's raised if an error occurs. If the URL is bad, it'll catch a `URLError`. If it goes through without any errors, it'll just print the status and return a tuple containing the body and the response. The response will close after return.

The function also calls `urlopen()` with a `timeout` argument, which will cause a `TimeoutError` to be raised after the seconds specified. Ten seconds is generally a good amount of time to wait for a response, though as always, much depends on the server that you need to make the request to.

Now you're set up to gracefully handle a variety of errors, including but not limited to the errors that you'll cover next.

 Remove ads

Dealing With 403 Errors

You'll now use the `make_request()` function to make some requests to httpstat.us, which is a mock server used for testing. This mock server will return responses that have the status code you request. If you make a request to `https://httpstat.us/200`, for example, you should expect a 200 response.

APIs like `httpstat.us` are used to ensure that your application can handle all the different status codes it might encounter. `httpbin` also has this functionality, but `httpstat.us` has a more comprehensive selection of status codes. It even has the [infamous and semi-official](#) 418 status code that returns the message *I'm a teapot*!

To interact with the `make_request()` function that you wrote in the previous section, run the script in interactive mode:

Shell

```
$ python3 -i request.py
```

With the `-i` flag, this command will run the script in [interactive mode](#). This means that it'll execute the script and then open the [Python REPL](#) afterward, so you can now call the function that you just defined:

Python

>>>

```
>>> make_request("https://httpstat.us/200")
200
(b'200 OK', <http.client.HTTPResponse object at 0x0000023D612660B0>)
>>> make_request("https://httpstat.us/403")
403 Forbidden
```

Here you tried the 200 and 403 endpoints of `httpstat.us`. The 200 endpoint goes through as anticipated and returns the body of the response and the response object. The 403 endpoint just printed the error message and didn't return anything, also as expected.

The `403` status means that the server understood the request but won't fulfill it. This is a common error that you can run into, especially while web scraping. In many cases, you can solve it by passing a `User-Agent` header.

Note: There are two closely related 4xx codes that sometimes cause confusion:

1. `401 Unauthorized`
2. `403 Forbidden`

Servers should return `401` if the user isn't identified or logged in and must do something to gain access, like log in or register.

The `403` status should be returned if the user is sufficiently identified but doesn't have access to the resource. For example, if you're logged in to a social media account and try to look at a person's private profile page, then you'll likely get a `403` status.

That said, don't place all your trust in status codes. Bugs exist and are common in complex distributed services. Some servers just aren't model citizens!

One of the primary ways that servers identify who or what is making the request is by examining the `User-Agent` header. The raw default request sent by `urllib.request` is the following:

Text

```
GET https://httpstat.us/403 HTTP/1.1
Accept-Encoding: identity
Host: httpstat.us
User-Agent: Python-urllib/3.10
Connection: close
```

Notice that User-Agent is listed as Python-urllib/3.10. You may find that some sites will try to block web scrapers, and this User-Agent is a dead giveaway. With that said, you can set your own User-Agent with `urllib.request`, though you'll need to modify your function a little:

File Changes (diff)

```
# request.py

from urllib.error import HTTPError, URLError
-from urllib.request import urlopen
+from urllib.request import urlopen, Request

-def make_request(url):
+def make_request(url, headers=None):
+    request = Request(url, headers=headers or {})
    try:
-        with urlopen(url, timeout=10) as response:
+        with urlopen(request, timeout=10) as response:
            print(response.status)
            return response.read(), response
    except HTTPError as error:
        print(error.status, error.reason)
    except URLError as error:
        print(error.reason)
    except TimeoutError:
        print("Request timed out")
```

To customize the headers that you send out with your request, you first have to instantiate a [Request](#) object with the URL. Additionally, you can pass in a [keyword argument](#) of `headers`, which accepts a standard dictionary representing any headers you wish to include. So, instead of passing the URL string directly into `urlopen()`, you pass this Request object which has been instantiated with the URL and headers.

Note: In the example above, when Request is instantiated, you need to pass it the headers if they've been defined. Otherwise, pass a blank object, like `{}`. You can't pass `None`, as this will cause an error.

To use this revamped function, restart the interactive session, then call `make_request()` with a dictionary representing the headers as an argument:

Python

>>>



```
>>> body, response = make_request(
...     "https://www.httpbin.org/user-agent",
...     {"User-Agent": "Real Python"}
... )
200
>>> body
b'{\n  "user-agent": "Real Python"\n}\n'
```


In this example, you make a request to `httpbin`. Here you use the `user-agent` endpoint to return the request's User-Agent value. Because you made the request with a custom user agent of `Real Python`, this is what gets returned.

Some servers are strict, though, and will only accept requests from specific browsers. Luckily, it's possible to find standard User-Agent strings on the web, including through a [user agent database](#). They're just strings, so all you need to do is copy the user agent string

or the browser that you want to impersonate and use it as the value of the User-Agent header.

Find Your Dream Python Job
pythonjobshq.com



 Remove ads

Fixing the SSL CERTIFICATE_VERIFY_FAILED Error

Another common error is due to Python not being able to access the required security certificate. To simulate this error, you can use some mock sites that have known bad SSL certificates, provided by badssl.com. You can make a request to one of them, such as `superfish.badssl.com`, and experience the error firsthand:

```
Python >>>
>>> from urllib.request import urlopen
>>> urlopen("https://superfish.badssl.com/")
Traceback (most recent call last):
  (...)
ssl.SSLCertVerificationError: [SSL: CERTIFICATE_VERIFY_FAILED]
certificate verify failed: unable to get local issuer certificate (_ssl.c:997)

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  (...)
urllib.error.URLError: <urlopen error [SSL: CERTIFICATE_VERIFY_FAILED]
certificate verify failed: unable to get local issuer certificate (_ssl.c:997):
```

Here, making a request to an address with a known bad SSL certificate will result in `CERTIFICATE_VERIFY_FAILED` which is a type of `URLError`.

SSL stands for Secure Sockets Layer. This is something of a misnomer because SSL was deprecated in favor of TLS, [Transport Layer Security](#). Sometimes old terminology just

deprecated in favor of TLS, [Transport Layer Security](#). Sometimes old terminology just sticks! It's a way to encrypt network traffic so that a hypothetical listener can't eavesdrop on the information transmitted over the wire.

These days, most website addresses are preceded not by `http://` but by `https://`, with the `s` standing for *secure*. [HTTPS](#) connections must be encrypted through the TLS.

`urllib.request` can handle both HTTP and HTTPS connections.

The details of HTTPS are far beyond the scope of this tutorial, but you can think of an HTTPS connection as involving two stages, the [handshake](#) and the transfer of information. The handshake ensures that the connection is secure. For more information about Python and HTTPS, check out [Exploring HTTPS With Python](#).

To establish that a particular server is secure, programs that make requests rely on a store of trusted certificates. The server's certificate is verified during the handshake stage. Python uses the [operating system's store of certificates](#). If Python can't find the system's store of certificates, or if the store is out of date, then you'll run into this error.

Note: In previous versions of Python, the default behavior for `urllib.request` was **not** to verify certificates, which led [PEP 476](#) to enable certificate verification by default. The default changed in [Python 3.4.3](#).

Sometimes the store of certificates that Python can access is out of date, or Python can't reach it, for whatever reason. This is frustrating because you can sometimes visit the URL from your browser, which thinks that it's secure, yet `urllib.request` still raises this error.

You may be tempted to opt out of verifying the certificate, but this will render your connection *insecure* and is definitely *not recommended*:

Python

>>>

```
>>> import ssl
>>> from urllib.request import urlopen
>>> unverified_context = ssl._create_unverified_context()
>>> urlopen("https://superfish.hadssl.com/", context=unverified_context)
```

```
urlopen('https://superfish.badssl.com/', context=unverified_context)  
<http.client.HTTPResponse object at 0x00000209CBE8F220>
```

Here you import the `ssl` module, which allows you to create an `unverified context`. You can then pass this context to `urlopen()` and visit a known bad SSL certificate. The connection successfully goes through because the SSL certificate isn't checked.

Before resorting to these desperate measures, try updating your OS or updating your Python version. If that fails, then you can take a page from the `requests` library and install `certifi`:

 Windows

 Linux + macOS

Windows PowerShell

```
PS> python -m venv venv  
PS> .\venv\Scripts\activate  
(venv) PS> python -m pip install certifi
```

`certifi` is a collection of certificates that you can use instead of your system's collection. You do this by creating an SSL context with the `certifi` bundle of certificates instead of the OS's bundle:

Python

>>>

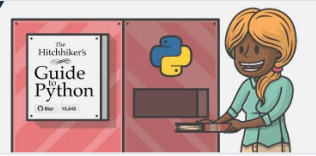
```
>>> import ssl  
>>> from urllib.request import urlopen  
>>> import certifi  
>>> certifi_context = ssl.create_default_context(cafile=certifi.where())  
>>> urlopen("https://sha384.badssl.com/", context=certifi_context)  
<http.client.HTTPResponse object at 0x000001C7407C3490>
```


In this example, you used `certifi` to act as your SSL certificate store, and you used it to successfully connect to a site with a known good SSL certificate. Note that instead of `._create_unverified_context()`, you use `.create_default_context()`.

This way, you can stay secure without too much trouble! In the next section, you'll be dipping your toes into the world of authentication.

Your Guide to the Python Programming Language and a Best Practices Handbook

python-guide.org



 Remove ads

Authenticated Requests

Authentication is a vast subject, and if you're dealing with authentication much more complicated than what's covered here, this might be a good jumping-off point into the `requests` package.

In this tutorial, you'll only cover one authentication method, which serves as an example of the type of adjustments that you have to make to authenticate your requests.

`urllib.request` does have a lot of other functionality that helps with authentication, but that won't be covered in this tutorial.

One of the most common authentication tools is the bearer token, specified by [RFC 6750](#). It's often used as part of [OAuth](#), but can also be used in isolation. It's also most common to see as a header, which you can use with your current `make_request()` function:

Python

>>>

```
>>> token = "abcdefghijklmnopqrstuvwxyz"
>>> headers = {
...     "Authorization": f"Bearer {token}"
... }
>>> make_request("https://httpbin.org/bearer", headers)
200
(b'{"authenticated": true, "token": "abcdefghijklmnopqrstuvwxyz"}\n',
<http.client.HTTPResponse object at 0x0000023D612642E0>)
```


In this example, you make a request to the `httpbin /bearer` endpoint, which simulates bearer authentication. It'll accept any string as a token. It only requires the proper format

specified by RFC 6750. The name *has* to be `Authorization`, or sometimes the lowercase `authorization`, and the value *has* to be `Bearer`, with a single space between that and the token.

Note: If you're using any form of tokens or secret information, be sure to protect these tokens appropriately. For example, don't commit them to a GitHub repository but instead store them as temporary [environment variables](#).

Congratulations, you've successfully authenticated, using a bearer token!

Another form of authentication is called **Basic Access Authentication**, which is a very simple method of authentication, only slightly better than sending a username and password in a header. It's very insecure!

One of the most common protocols in use today is **OAuth (Open Authorization)**. If you've ever used Google, GitHub, or Facebook to sign into another website, then you've used OAuth. The OAuth flow generally involves a few requests between the service that you want to interact with and an identity server, resulting in a short-lived bearer token. This bearer token can then be used for a period of time with bearer authentication.

Much of authentication comes down to understanding the specific protocol that the target server uses and reading the documentation closely to get it working.

POST Requests With `urllib.request`

You've made a lot of GET requests, but sometimes you want to *send* information. That's where POST requests come in. To make POST requests with `urllib.request`, you don't have to explicitly change the method. You can just pass a data object to a new `Request` object or directly to `urlopen()`. The data object must be in a special format, though. You'll

adapt your `make_request()` function slightly to support POST requests by adding the `data` parameter:

File Changes (diff)

```
# request.py

from urllib.error import HTTPError, URLError
from urllib.request import urlopen, Request

-def make_request(url, headers=None):
+def make_request(url, headers=None, data=None):

-    request = Request(url, headers=headers or {})
+    request = Request(url, headers=headers or {}, data=data)
    try:
        with urlopen(request, timeout=10) as response:
            print(response.status)
            return response.read(), response
    except HTTPError as error:
        print(error.status, error.reason)
    except URLError as error:
        print(error.reason)
    except TimeoutError:
        print("Request timed out")
```

Here you just modified the function to accept a `data` argument with a default value of `None`, and you passed that right into the `Request` instantiation. That's not all that needs to be done, though. You can use one of two different formats to execute a POST request:

1. **Form Data:** `application/x-www-form-urlencoded`
2. **JSON:** `application/json`

The first format is the oldest format for POST requests and involves encoding the data with [percent encoding](#), also known as URL encoding. You may have noticed key-value pairs URL encoded as a [query string](#). Keys are separated from values with an equal sign (=), key-value pairs are separated with an ampersand (&), and spaces are generally suppressed but

value pairs are separated with an ampersand (&), and spaces are generally suppressed but can be replaced with a plus sign (+).

If you're starting off with a Python dictionary, to use the form data format with your `make_request()` function, you'll need to encode twice:

1. Once to URL encode the dictionary
2. Then again to encode the resulting string into bytes

For the first stage of URL encoding, you'll use another `urllib` module, `urllib.parse`. Remember to start your script in interactive mode so that you can use the `make_request()` function and play with it on the REPL:

Python

>>>

```
>>> from urllib.parse import urlencode
>>> post_dict = {"Title": "Hello World", "Name": "Real Python"}
>>> url_encoded_data = urlencode(post_dict)
>>> url_encoded_data
'Title=Hello+World&Name=Real+Python'
>>> post_data = url_encoded_data.encode("utf-8")
>>> body, response = make_request(
...     "https://httpbin.org/anything", data=post_data
... )
200
>>> print(body.decode("utf-8"))
{
  "args": {},
  "data": "",
  "files": {},
  "form": {
    "Name": "Real Python",
    "Title": "Hello World"
  },
  "headers": {
    "Accept-Encoding": "identity",
    "Content-Length": "34",
    "Content-Type": "application/x-www-form-urlencoded",
    "Host": "httpbin.org",
    "User-Agent": "Python-urllib/3.10",
    "X-Amzn-Trace-Id": "Root=1-61f25a81-03d2d4377f0abae95ff34096"
  },
  "json": null,
  "method": "POST",
  "origin": "86.159.145.119",
  "url": "https://httpbin.org/anything"
}
```

In this example, you

in this example, you.

1. Import `urlencode()` from the `urllib.parse` module
2. Initialize your POST data, starting with a dictionary
3. Use the `urlencode()` function to encode the dictionary
4. Encode the resulting string into bytes using UTF-8 encoding
5. Make a request to the anything endpoint of `httpbin.org`
6. Print the UTF-8 decoded response body

UTF-8 encoding is part of the [specification](#) for the `application/x-www-form-urlencoded` type. UTF-8 is used preemptively to decode the body because you already know that `httpbin.org` reliably uses UTF-8.

The anything endpoint from `httpbin` acts as a sort of echo, returning all the information it received so that you can inspect the details of the request you made. In this case, you can confirm that `method` is indeed `POST`, and you can see that the data you sent is listed under `form`.

To make the same request with JSON, you'll turn a Python dictionary into a JSON string with `json.dumps()`, encode it with UTF-8, pass it as the `data` argument, and finally add a special header to indicate that the data type is JSON:

Python

>>>

```
>>> post_dict = {"Title": "Hello World", "Name": "Real Python"}
>>> import json
>>> json_string = json.dumps(post_dict)
>>> json_string
'{"Title": "Hello World", "Name": "Real Python"}'
>>> post_data = json_string.encode("utf-8")
>>> body, response = make_request(
...     "https://httpbin.org/anything",
...     data=post_data,
...     headers={"Content-Type": "application/json"},
... )
200
>>> print(body.decode("utf-8"))
{
  "args": {},
  "data": "{\"Title\": \"Hello World\", \"Name\": \"Real Python\"}",
  "files": {},
  "form": {},
  "headers": {
    "Accept-Encoding": "identity",
    "Content-Length": "47",
    "Content-Type": "application/json",
    "Host": "httpbin.org",
    "User-Agent": "Python-urllib/3.10",
    "X-Amzn-Trace-Id": "Root=1-61f25a81-3e35d1c219c6b5944e2d8a52"
  },
  "json": {
    "Name": "Real Python",
    "Title": "Hello World"
  },
  "method": "POST",
  "origin": "86.159.145.119",
  "url": "https://httpbin.org/anything"
}
```

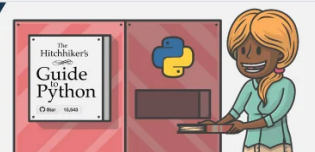
To [serialize](#) the dictionary this time around, you use `json.dumps()` instead of `urlencode()`. You also explicitly add the [Content-Type header](#) with a value of `application/json`. With this information, the `httpbin` server can deserialize the JSON on the receiving end. In its response, you can see the data listed under the `json` key.

Note: Sometimes it's necessary to send JSON data as plain text, in which case the steps are as above, except you set `Content-Type` as `text/plain; charset=UTF-8`. A lot of these necessities depend on the server or API that you're sending data to, so be sure to read the documentation and experiment!

With that, you can now start making POST requests. This tutorial won't go into more detail about the other request methods, such as [PUT](#). Suffice to say that you can also explicitly set the method by passing a `method` keyword argument to the instantiation of the [Request object](#).

A Python Best Practices Handbook

python-guide.org



 Remove ads

The Request Package Ecosystem

To round things out, this last section of the tutorial is dedicated to clarifying the package ecosystem around HTTP requests with Python. Because there are many packages, with no clear standard, it can be confusing. That said, there are use cases for each package, which just means more choice for you!

What Are `urllib2` and `urllib3`?

To answer this question, you need to go back to early Python, all the way back to version

1.2, when the original `urllib` was introduced. Around version 1.6, a revamped `urllib2` was added, which lived alongside the original `urllib`. When Python 3 came along, the original

`urllib` was deprecated, and `urllib2` dropped the 2, taking on the original `urllib` name. It also split into parts:

- `urllib.error`
- `urllib.parse`
- `urllib.request`
- `urllib.response`
- `urllib.robotparser`

So what about `urllib3`? That's a third-party library developed while `urllib2` was still around. It's not related to the standard library because it's an independently maintained library. Interestingly, the `requests` library actually uses `urllib3` under the hood, and so does `pip`!

When Should I Use `requests` Over `urllib.request`?

The main answer is ease of use and security. `urllib.request` is considered a low-level library, which exposes a lot of the detail about the workings of HTTP requests. The Python [documentation](#) for `urllib.request` makes no bones about recommending `requests` as a higher-level HTTP client interface.

If you interact with many different REST APIs, day in and day out, then `requests` is highly recommended. The `requests` library bills itself as “built for human beings” and has successfully created an intuitive, secure, and straightforward API around HTTP. It's usually considered the go-to library! If you want to know more about the `requests` library, check out the Real Python [guide to requests](#).

An example of how `requests` makes things easier is when it comes to character encoding. You'll remember that with `urllib.request`, you have to be aware of encodings and take a

few steps to ensure an error-free experience. The `requests` package abstracts that away

and will resolve the encoding by using `chardet`, a universal character encoding detector, just in case there's any funny business.

If your goal is to learn more about standard Python and the details of how it deals with HTTP requests, then `urllib.request` is a great way to get into that. You could even go further and use the very low-level `http modules`. On the other hand, you may just want to keep dependencies to a minimum, which `urllib.request` is more than capable of.

Why Is `requests` Not Part of the Standard Library?

Maybe you're wondering why `requests` isn't part of core Python by this point.

This is a complex issue, and there's no hard and fast answer to it. There are many speculations as to why, but two reasons seem to stand out:

1. `requests` has other third-party dependencies that would need to be integrated too.
2. `requests` needs to stay agile and can do this better outside the standard library.

The `requests` library has third-party dependencies. Integrating `requests` into the standard library would mean also integrating `chardet`, `certifi`, and `urllib3`, among others. The alternative would be to fundamentally change `requests` to use only Python's existing standard library. This is no trivial task!

Integrating `requests` would also mean that the existing team that develops this library would have to relinquish total control over the design and implementation, giving way to the [PEP](#) decision-making process.

HTTP specifications and recommendations change all the time, and a high-level library has to be agile enough to keep up. If there's a security exploit to be patched, or a new workflow to add, the `requests` team can build and release far more quickly than they could as part of

the Python release process. There have supposedly been times when they've released a security fix twelve hours after a vulnerability was discovered!

For an interesting overview of these issues and more, check out [Adding Requests to The Standard Library](#), which summarizes a discussion at the Python Language Summit with [Kenneth Reitz](#), the creator and maintainer of Requests.


Because this agility is so necessary to `requests` and its underlying `urllib3`, the paradoxical statement that `requests` is too important for the standard library is often used. This is because so much of the Python community depends on `requests` and its agility that integrating it into core Python would probably damage it and the Python community.

On the GitHub repository issues board for `requests`, an issue was posted, asking for the inclusion of [requests in the standard library](#). The developers of `requests` and `urllib3` chimed in, mainly saying they would likely lose interest in maintaining it themselves. Some even said they would fork the repositories and continue developing them for their own use cases.

With that said, note that the `requests` library GitHub repository is hosted under the Python Software Foundation's account. Just because something isn't part of the Python standard library doesn't mean that it's not an integral part of the ecosystem!

It seems that the current situation works for both the Python core team and the maintainers of `requests`. While it may be slightly confusing for newcomers, the existing structure gives the most stable experience for HTTP requests.

It's also important to note that HTTP requests are inherently complex. `urllib.request` doesn't try to sugarcoat that too much. It exposes a lot of the inner workings of HTTP requests, which is why it's billed as a low-level module. Your choice of `requests` versus `urllib.request` really depends on your particular use case, security concerns, and preference.

 Remove ads

Conclusion

You're now equipped to use `urllib.request` to make HTTP requests. Now you can use this built-in module in your projects, keeping them dependency-free for longer. You've also gained the in-depth understanding of HTTP that comes from using a lower-level module, such as `urllib.request`.

In this tutorial, you've:

- Learned how to make basic **HTTP requests** with `urllib.request`
- Explored the nuts and bolts of an **HTTP message** and studied how it's represented by `urllib.request`
- Figured out how to deal with character **encodings** of HTTP messages
- Explored some **common errors** when using `urllib.request` and learned how to resolve them
- Dipped your toes into the world of **authenticated requests** with `urllib.request`
- Understood why both `urllib` and the `requests` library exist and **when to use one or the other**

You're now in a position to make basic HTTP requests with `urllib.request`, and you also have the tools to dive deeper into low-level HTTP terrain with the standard library. Finally, you can choose whether to use `requests` or `urllib.request`, depending on what you want or need. Have fun exploring the Web!

Learn More: Click here to join 290,000+ Python developers on the Real Python Newsletter and get new Python tutorials and news that will make you a more

Mark as Completed



Python Tricks

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}
```

Send Me Python Tricks »

About Ian Currie



Ian is a Python nerd who uses it for everything from tinkering to helping people and companies manage their day-to-day and develop their businesses.

» [More about Ian](#)

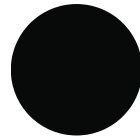
Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:



Aldren



Bartosz



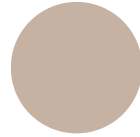
Geir Arne



Kate



Martin



Sadie

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: [api](#) [intermediate](#) [web-dev](#) [web-scraping](#)





 [Remove ads](#)

© 2012–2023 Real Python · [Newsletter](#) · [Podcast](#) · [YouTube](#) · [Twitter](#) · [Facebook](#) · [Instagram](#) ·
[Python Tutorials](#) · [Search](#) · [Privacy Policy](#) · [Energy Policy](#) · [Advertise](#) · [Contact](#)

 Happy Pythoning!