21 Requests

The Requests library is the de facto standard for making HTTP requests in Python. It abstracts the complexities of making requests behind a beautiful, simple API so that you can focus on interacting with services and consuming data in your application.

Getting Started With Python's Requests Library

Because Requests is a third-party library, you need to install it before you can use it in your code.

```
python -m pip install requests
```

Note: Requests doesn't support asynchronous HTTP requests directly. If you need async support in your program, you should try out AIOHTTP or HTTPX. The latter library is broadly compatible with Requests' syntax.

The GET Request

HTTP methods, such as `GET` and `POST`, determine which action you're trying to perform when making an HTTP request. Besides `GET` and `POST`, there are several other common methods.

One of the most common HTTP methods is `GET`. The `GET` method indicates that you're trying to get or retrieve data from a specified resource. To make a `GET` request using Requests, you can invoke `requests.get()`

To test this out, you can make a `GET` request to GitHub's REST API by calling `get()` with the following URL

```
>>> import requests
>>> requests.get("https://api.github.com")
<Response [200]>
```

The Response

A `Response` is a powerful object for inspecting the results of the request. Make that same request again, but this time store the return value in a variable so that you can get a closer look at its attributes and behaviours:

```
>>> import requests
>>> response = requests.get("https://api.github.com")
```

You can now use `response` to see a lot of information about the results of your `GET` request.

Status Codes

The first bit of information that you can gather from `Response` is the status code. A status code informs you of the status of the request.

For example, a `200 OK` status means that your request was successful, whereas a `404 NOT FOUND` status means that the resource you were looking for wasn't found.

By accessing `.status_code`, you can see the status code that the server returned:

```
>>> response.status_code
200
```

Sometimes, you might want to use this information to make decisions in your code:

```
if response.status_code == 200:
    print("Success!")
elif response.status_code == 404:
    print("Not Found.")
```

Requests goes one step further in simplifying this process for you. If you use a `Response` instance in a conditional expression, then it'll evaluate to `True` if the status code was smaller than `400`, and `False` otherwise.

Therefore, you can simplify the last example by rewriting the `if` statement:

```
if response:
    print("Success!")
else:
    raise Exception(f"Non-success status code: {response.status_code}")
```

In the code snippet above, you implicitly check whether the `.status_code` of `response` is between `200` and `399`. If it's not, then you raise an exception that includes the non-success status code in an f-string.

Keep in mind that this method is *not* verifying that the status code is equal to `200`. The reason for this is that other status codes within the `200` to `399` range, such as `204 NO`

`CONTENT` and `304 NOT MODIFIED`, are also considered successful in the sense that they provide some workable response.

For example, the status code `204` tells you that the response was successful, but there's no content to return in the message body.

Let's say you don't want to check the response's status code in an `if` statement. Instead, you want to use Request's built-in capacities to raise an exception if the request was unsuccessful. You can do this using `.raise_for_status()`:

```
import requests
from requests.exceptions import HTTPError

URLS = ["https://api.github.com", "https://api.github.com/invalid"]

for url in URLS:
    try:
        response = requests.get(url)
        response.raise_for_status()
    except HTTPError as http_err:
        print(f"HTTP error occurred: {http_err}")
    except Exception as err:
        print(f"Other error occurred: {err}")
    else:
        print("Success!")
```

If you invoke `.raise_for_status()`, then Requests will raise an `HTTPError` for status codes between `400` and `600`. If the status code indicates a successful request, then the program will proceed without raising that exception.

Content

The response of a `GET` request often has some valuable information, known as a payload, in the message body. Using the attributes and methods of `Response`, you can view the payload in a variety of different formats.

To see the response's content in `bytes`, you use `.content`:

```
>>> import requests

>>> response = requests.get("https://api.github.com")
>>> response.content
b'{"current_user_url":"https://api.github.com/user", ...}'
```

```
>>> type(response.content)
<class 'bytes'>
```

While `.content` gives you access to the raw bytes of the response payload, you'll often want to convert them into a string using a character encoding such as UTF-8. `response` will do that for you when you access `.text`:

```
>>> response.text
'{"current_user_url":"https://api.github.com/user", ...}'

>>> type(response.text)
<class 'str'>
```

Because the decoding of `bytes` to a `str` requires an encoding scheme, Requests will try to guess the encoding based on the response's headers if you don't specify one. You can provide an explicit encoding by setting `.encoding` before accessing `.text`:

```
>>> response.encoding = "utf-8"  # Optional: Requests infers this.
>>> response.text
'{"current_user_url":"https://api.github.com/user", ...}'
```

If you take a look at the response, then you'll see that it's actually serialized JSON content.

To get a dictionary, you could take the `str` that you retrieved from `.text` and deserialize it using `json.loads()`. However, a simpler way to accomplish this task is to use `.json()`:

```
>>> response.json()
{'current_user_url': 'https://api.github.com/user', ...}

>>> type(response.json())
<class 'dict'>

>>> response_dict = response.json()
>>> response_dict["emojis_url"]
'https://api.github.com/emojis'
```

You can do a lot with status codes and message bodies. But, if you need more information, like metadata about the response itself, then you'll need to look at the response's headers.

Headers

The response headers can give you useful information, such as the content type of the response payload and a time limit on how long to cache the response. To view these headers, access `.headers`:

```
>>> import requests

>>> response = requests.get("https://api.github.com")
>>> response.headers
{'Server': 'GitHub.com',
...
'X-GitHub-Request-Id': 'AE83:3F40:2151C46:438A840:65C38178'}
```

`.headers` returns a dictionary-like object, allowing you to access header values by key. For example, to see the content type of the response payload, you can access `"Content-Type"`:

```
>>> response.headers["Content-Type"]
'application/json; charset=utf-8'
```

There's something special about this dictionary-like headers object, though. The HTTP specification defines headers as case-insensitive, which means that you're able to access these headers without worrying about their capitalization:

```
>>> response.headers["content-type"]
'application/json; charset=utf-8'
```

Query String Parameters

One common way to customize a `GET` request is to pass values through query string parameters in the URL. To do this using `get()`, you pass data to `params`. For example, you can use GitHub's repository search API to look for popular Python repositories:

```
# search_popular_repos.py
import requests

# Search GitHub's repositories for popular Python projects
response = requests.get(
    "https://api.github.com/search/repositories",
    params={"q": "language:python", "sort": "stars", "order": "desc"},
)

# Inspect some attributes of the first three repositories
json_response = response.json()
```

```
popular_repositories = json_response["items"]
for repo in popular_repositories[:3]:
    print(f"Name: {repo['name']}")
    print(f"Description: {repo['description']}")
    print(f"Stars: {repo['stargazers_count']}")
    print()
```

By passing a dictionary to the `params` parameter of `get()`, you're able to modify the results that come back from the search API.

You can pass `params` to `get()` in the form of a dictionary, as you've just done, or as a list of tuples:

```
>>> import requests

>>> requests.get(
...     "https://api.github.com/search/repositories",
...     [("q", "language:python"), ("sort", "stars"), ("order", "desc")],
... )
<Response [200]>
```

You can even pass the values as `bytes`:

```
>>> requests.get(
...     "https://api.github.com/search/repositories",
...     params=b"q=language:python&sort=stars&order=desc",
... )
<Response [200]>
```

Query strings are useful for parameterizing `GET` requests. Another way to customize your requests is by adding or modifying the headers that you send.

Request Headers

To customize headers, you pass a dictionary of HTTP headers to `get()` using the `headers` parameter. For example, you can change your previous search request to highlight matching search terms in the results by specifying the `text-match` media type in the `Accept` header:

```
# text_matches.py
import requests

response = requests.get(
```

```
    "https://api.github.com/search/repositories",
    params={"q": '"real python"'},
    headers={"Accept": "application/vnd.github.text-match+json"},
)

# View the new `text-matches` list which provides information
# about your search term within the results
json_response = response.json()
first_repository = json_response["items"][0]
print(first_repository["text_matches"][0]["matches"])
```

The `Accept` header tells the server what content types your application can handle. In this case, since you're expecting the matching search terms to be highlighted, you're using the header value `application/vnd.github.text-match+json`, which is a proprietary GitHub `Accept` header where the content is a special JSON format.

If you run this code, then you'll get a result similar to the one shown below:

```
python text_matches.py
[{'text': 'Real Python', 'indices': [23, 34]}]
```

Other HTTP Methods

Aside from `GET`, other popular HTTP methods include `POST`, `PUT`, `DELETE`, `HEAD`, `PATCH`, and `OPTIONS`. For each of these HTTP methods, Requests provides a function, with a similar signature to `get()`.

Note: To try out these HTTP methods, you'll make requests to httpbin.org. The httpbin service is a great resource created by the original author of Requests, Kenneth Reitz. The service accepts test requests and responds with data about the requests.

You'll notice that Requests provides an intuitive interface to all the mentioned HTTP methods:

```
>>> import requests

>>> requests.get("https://httpbin.org/get")
<Response [200]>
>>> requests.post("https://httpbin.org/post", data={"key": "value"})
<Response [200]>
>>> requests.put("https://httpbin.org/put", data={"key": "value"})
<Response [200]>
>>> requests.delete("https://httpbin.org/delete")
<Response [200]>
>>> requests.head("https://httpbin.org/get")
```

```
<Response [200]>
>>> requests.patch("https://httpbin.org/patch", data={"key": "value"})
<Response [200]>
>>> requests.options("https://httpbin.org/get")
<Response [200]>
```

Note: All of these functions are high-level shortcuts to `requests.request()`, passing the name of the relevant HTTP method:

```
>>> requests.request("GET", "https://httpbin.org/get")
<Response [200]>
```

You can inspect the responses in the same way as you did before:

```
>>> response = requests.head("https://httpbin.org/get")
>>> response.headers["Content-Type"]
'application/json'

>>> response = requests.delete("https://httpbin.org/delete")
>>> json_response = response.json()
>>> json_response["args"]
{}
```

The Message Body

According to the HTTP specification, `POST`, `PUT`, and the less common `PATCH` requests pass their data through the message body rather than through parameters in the query string. Using Requests, you'll pass the payload to the corresponding function's `data` parameter. `data` takes a dictionary, a list of tuples, bytes, or a file-like object.

For example, if your request's content type is `application/x-www-form-urlencoded`, then you can send the form data as a dictionary:

```
>>> import requests

>>> requests.post("https://httpbin.org/post", data={"key": "value"})
<Response [200]>
```

You can also send that same data as a list of tuples:

```
>>> requests.post("https://httpbin.org/post", data=[("key", "value")])
<Response [200]>
```

If, however, you need to send JSON data, then you can use the `json` parameter. When you pass JSON data via `json`, Requests will serialize your data and add the correct `Content-Type` header for you.

```
>>> response = requests.post("https://httpbin.org/post", json={"key":
"value"})
>>> json_response = response.json()
>>> json_response["data"]
'{"key": "value"}'
>>> json_response["headers"]["Content-Type"]
'application/json'
```

Request Inspection

When you make a request, the Requests library prepares the request before actually sending it to the destination server. Request preparation includes things like validating headers and serializing JSON content.

You can view the `PreparedRequest` object by accessing `.request` on a `Response` object:

```
>>> import requests

>>> response = requests.post("https://httpbin.org/post", json=
{"key":"value"})

>>> response.request.headers["Content-Type"]
'application/json'
>>> response.request.url
'https://httpbin.org/post'
>>> response.request.body
b'{"key": "value"}'
```

Inspecting `PreparedRequest` gives you access to all kinds of information about the request being made, such as payload, URL, headers, authentication, and more.

Authentication

Authentication helps a service understand who you are. Typically, you provide your credentials to a server by passing data through the `Authorization` header or a custom header defined by the service. All the functions of Requests that you've seen to this point provide a parameter called `auth`, which allows you to pass your credentials:

```
>>> import requests

>>> response = requests.get(
...     "https://httpbin.org/basic-auth/user/passwd",
...     auth=("user", "passwd")
... )

>>> response.status_code
200
>>> response.request.headers["Authorization"]
'Basic dXNlcjpwYXNzd2Q='
```

The request succeeds if the credentials that you pass in the tuple to `auth` are valid.

When you pass your credentials in a tuple to the `auth` parameter, Requests applies the credentials using HTTP's Basic access authentication scheme under the hood.

The Basic Authentication Scheme
ou may wonder where the string `Basic dXNlcjpwYXNzd2Q=` that Requests set as the value for your `Authorization` header comes from. In short, it's a Base64-encoded string of the username and password with the prefix `"Basic "`:

1. First, Requests combines the username and password that you provided, putting a colon in between them. So for the username `"user"` and password `"passwd"`, this becomes `"user:passwd"`.
2. Then, Requests encodes this string in Base64 using `base64.b64encode()`. The encoding converts the `"user:passwd"` string to `"dXNlcjpwYXNzd2Q="`.
3. Finally, Requests adds `"Basic "` in front of this Base64 string.

You could make the same request by passing explicit Basic authentication credentials using `HTTPBasicAuth`:

```
>>> from requests.auth import HTTPBasicAuth
>>> requests.get(
...     "https://httpbin.org/basic-auth/user/passwd",
...     auth=HTTPBasicAuth("user", "passwd")
... )
<Response [200]>
```

Requests provides other methods of authentication out of the box, such as `HTTPDigestAuth` and `HTTPProxyAuth`.

A real-world example of an API that requires authentication is GitHub's authenticated user API. This endpoint provides information about the authenticated user's profile.

If you try to make a request without credentials, then you'll see that the status code is `401 Unauthorized`:

```
>>> requests.get("https://api.github.com/user")
<Response [401]>
```

To make a request to GitHub's authenticated user API, you first need to generate a personal access token with the *read:user* scope. Then you can pass this token as the second element in a tuple to `get()`:

```
>>> import requests

>>> token = "<YOUR_GITHUB_PA_TOKEN>"
>>> response = requests.get(
...     "https://api.github.com/user",
...     auth=("", token)
... )
>>> response.status_code
200

>>> response.request.headers["Authorization"]
'Basic OmdocF92dkd...WpremM0SGRuUGY='
```

This works, but it's not the right way to authenticate with a Bearer token and using an empty string input for the superfluous username is awkward.

With Requests, you can supply your own authentication mechanism to fix that. To try this out, create a subclass of `AuthBase` and implement `.__call__()`:

```
# custom_token_auth.py
from requests.auth import AuthBase

class TokenAuth(AuthBase):
    """Implements a token authentication scheme."""

    def __init__(self, token):
        self.token = token

    def __call__(self, request):
        """Attach an API token to the Authorization header."""
```

```
        request.headers["Authorization"] = f"Bearer {self.token}"
        return request
```

Here, your custom `TokenAuth` mechanism receives a token, then includes that token in the `Authorization` header of your request, also setting the recommended `"Bearer "` prefix to the string.

You can now use this custom token authentication to make your call to GitHub's authenticated user API:

```
>>> import requests
>>> from custom_token_auth import TokenAuth

>>> token = "<YOUR_GITHUB_PA_TOKEN>"
>>> response = requests.get(
...     "https://api.github.com/user",
...     auth=TokenAuth(token)
... )

>>> response.status_code
200
>>> response.request.headers["Authorization"]
'Bearer ghp_b...Tx'
```

Note: While you could construct the authentication string outside of a custom authentication class and pass it directly with `headers`, this appoach is discouraged because it can lead to unexpected behavior.

When you attempt to set your authentication credentials directly using `headers`, then Requests may internally overwrite your input. This can happen, for example, if you have a `.netrc` file that provides authentication credentials. Requests will attempt to get the credentials from the `.netrc` file if you don't provide an authentication method using `auth=`.

Bad authentication mechanisms can lead to security vulnerabilities. Unless a service requires a custom authentication mechanism for some reason, you'll always want to use a tried-and-true auth scheme like the built-in Basic authentication or OAuth, for example through Requests-OAuthlib.

SSL Certificate Verification

Anytime the data that you're trying to send or receive is sensitive, security is important. The way that you communicate with secure sites over HTTP is by establishing an encrypted connection using SSL, which means that verifying the target server's SSL certificate is critical.

The good news is that Requests does this for you by default. However, there are some cases where you might want to change this behavior.

If you want to disable SSL certificate verification, then you pass `False` to the `verify` parameter of the request function:

```
>>> import requests

>>> requests.get("https://api.github.com", verify=False)
InsecureRequestWarning: Unverified HTTPS request is being made to host
↪ 'api.github.com'. Adding certificate verification is strongly advised.
↪ See: https://urllib3.readthedocs.io/en/latest/advanced-usage.html#tls-
warnings
↪  warnings.warn(
<Response [200]>
```

Note: Requests uses a package called `certifi` to provide certificate authorities. This lets Requests know which authorities it can trust. Therefore, you should update `certifi` frequently to keep your connections as secure as possible.

Performance

When using Requests, especially in a production application environment, it's important to consider performance implications. Features like timeout control, sessions, and retry limits can help you keep your application running smoothly.

Timeouts

When you make an inline request to an external service, your system will need to wait for the response before moving on. If your application waits too long for that response, requests to your service could back up, your user experience could suffer, or your background jobs could hang.

By default, Requests will wait indefinitely on the response, so you should almost always specify a timeout duration to prevent these issues from happening.

```
>>> requests.get("https://api.github.com", timeout=1)
<Response [200]>
>>> requests.get("https://api.github.com", timeout=3.05)
<Response [200]>
```

In the first request, the request will time out after 1 second. In the second request, the request will time out after 3.05 seconds.

You can also pass a tuple to `timeout` with the following two elements:

1. **Connect timeout**: The time it allows for the client to establish a connection to the server
2. **Read timeout**: The time it'll wait on a response once your client has established a connection
   Both of these elements should be numbers, and can be of type `int` or `float`:

```
>>> requests.get("https://api.github.com", timeout=(3.05, 5))
<Response [200]>
```

If the request times out, then the function will raise a `Timeout` exception:

```
import requests
from requests.exceptions import Timeout

try:
    response = requests.get("https://api.github.com", timeout=(3.05, 5))
except Timeout:
    print("The request timed out")
else:
    print("The request did not time out")
```

The Session Object

Until now, you've been dealing with high-level `requests` APIs such as `get()` and `post()`. These functions are abstractions of what's going on when you make your requests.

Underneath those abstractions is a class called `Session`. If you need to fine-tune your control over how requests are being made or improve the performance of your requests, you may need to use a `Session` instance directly.

Sessions are used to persist parameters across requests. For example, if you want to use the same authentication across multiple requests, then you can use a session:

```
import requests
from custom_token_auth import TokenAuth

TOKEN = "<YOUR_GITHUB_PA_TOKEN>"

with requests.Session() as session:
    session.auth = TokenAuth(TOKEN)

    first_response = session.get("https://api.github.com/user")
```

```
    second_response = session.get("https://api.github.com/user")

print(first_response.headers)
print(second_response.json())
```

The primary performance optimization of sessions comes in the form of persistent connections. When your app makes a connection to a server using a `Session`, it keeps that connection around in a connection pool. When your app wants to connect to the same server again, it'll reuse a connection from the pool rather than establishing a new one.

Max Retries

When a request fails, you may want your application to retry the same request. However, Requests won't do this for you by default. To apply this functionality, you need to implement a custom transport adapter.

Transport adapters let you define a set of configurations for each service that you're interacting with. For example, say you want all requests to `https://api.github.com` to retry two times before finally raising a `RetryError`. You'd build a transport adapter, set its `max_retries` parameter, and mount it to an existing `Session`:

```
import requests
from requests.adapters import HTTPAdapter
from requests.exceptions import RetryError

github_adapter = HTTPAdapter(max_retries=2)

session = requests.Session()

session.mount("https://api.github.com", github_adapter)

try:
    response = session.get("https://api.github.com/")
except RetryError as err:
    print(f"Error: {err}")
finally:
    session.close()
```

When you mount the `HTTPAdapter` in this case, `github_adapter` to `session`, then `session` will adhere to its configuration for each request to `https://api.github.com`.

Note: While the implementation shown above works, you won’t see any effect of the retry behavior unless there’s something wrong with your network connection or GitHub’s servers.