Python Request Library Parameters

Simply these are the optional request parameters we can build and send in our request.

requests.**request**(*method*, *url*, ***kwargs*)

Constructs and sends a Request.

Parameters

- **method** – method for the new **Request** object: GET, OPTIONS, HEAD, POST, PUT, PATCH, or DELETE.
- **url** – URL for the new **Request** object.
- **params** – (optional) Dictionary, list of tuples or bytes to send in the query string for the **Request**.
- **data** – (optional) Dictionary, list of tuples, bytes, or file-like object to send in the body of the **Request**.
- **json** – (optional) A JSON serializable Python object to send in the body of the **Request**.
- **headers** – (optional) Dictionary of HTTP Headers to send with the **Request**.
- **cookies** – (optional) Dict or CookieJar object to send with the **Request**.
- **files** – (optional) Dictionary of 'name': file-like-objects (or {'name': file-tuple}) for multipart encoding upload. file-tuple can be a 2-tuple ('filename', fileobj), 3-tuple ('filename', fileobj, 'content_type') or a 4-tuple ('filename', fileobj, 'content_type', custom_headers), where 'content_type' is a string defining the content type of the given file and custom_headers a dict-like object containing additional headers to add for the file.
- **auth** – (optional) Auth tuple to enable Basic/Digest/Custom HTTP Auth.
- **timeout** (*float* *or* *tuple*) – (optional) How many seconds to wait for the server to send data before giving up, as a float, or a (connect timeout, read timeout) tuple.
- **allow_redirects** (*bool*) – (optional) Boolean. Enable/disable GET/OPTIONS/POST/PUT/PATCH/DELETE/HEAD redirection. Defaults to True.
- **proxies** – (optional) Dictionary mapping protocol to the URL of the proxy.
- **verify** – (optional) Either a boolean, in which case it controls whether we verify the server’s TLS certificate, or a string, in which case it must be a path to a CA bundle to use. Defaults to True.
- **stream** – (optional) if False, the response content will be immediately downloaded.
- **cert** – (optional) if String, path to ssl client cert file (.pem). If Tuple, (‘cert’, ‘key’) pair.

Returns

Response object

Return type

requests.Response

NOTE

- Python syntax you can use *args and **kwargs as arguments of a function when you are unsure about the number of arguments to pass in the functions. Further explanation here.
- OOP (Object Oriented Programming) is an important concept to understand when working with Python.

Python Request Library Exceptions

exception requests.**RequestException**(**args*, ***kwargs*)

- There was an ambiguous exception that occurred while handling your request.

exception requests.**ConnectionError**(**args*, ***kwargs*)

- A Connection error occurred.

exception requests.**HTTPError**(**args*, ***kwargs*)

- An HTTP error occurred.

exception requests.**URLRequired**(**args*, ***kwargs*)

- A valid URL is required to make a request.

exception requests.**TooManyRedirects(****args*, ***kwargs*)

- Too many redirects.

exception requests.**ConnectTimeout**(**args*, ***kwargs*)

- The request timed out while trying to connect to the remote server. Requests that produced this error are safe to retry.

exception requests.**ReadTimeout**(**args*, ***kwargs*)

- The server did not send any data in the allotted amount of time.

exception requests.**Timeout(****args*, ***kwargs*)[source]

- The request timed out. Catching this error will catch both **ConnectTimeout** and **ReadTimeout** errors.

exception requests.**JSONDecodeError**(**args*, ***kwargs*)[source]

- Couldn’t decode the text into json

Python Request Sessions

Connection pooling refers to reusage of existing pre-established connections to make HTTP requests, rather than creating a new connection for each service request, be it a connection of accessing remote REST API endpoint, or a backend database instance. Connection pooling can help to improve the performance of an application by reducing the overhead of establishing new connections, particularly for applications that make many HTTP requests concurrently.

Connection pools have implementation in various programming languages and frameworks, including Java/Spring, C#/.NET, JavaScript and Python etc.

First and foremost, connection pools should be used whenever and wherever the implementation is available in the framework you choose for your business applications. Our benchmarking exercise comparing load and stress testing of applications with and without connection pools shows that applications using connection pools gain tremendous performance optimization in service requests average response time. In certain cases, the transaction speed is 5 times faster, especially when applications are deployed in multi-cloud or hybrid cloud, or noisy environments.

Source: Microsoft

Python requests Session object allows one to persist certain parameters across requests. It also persists cookies across all requests made from the Session instance and will use urllib3's connection pooling. So, if several requests are being made to the same host, the underlying TCP connection will be reused, which can result in a significant performance increase. A session object all the methods as of requests.

class requests.**Session**[source]

A Requests session.

Provides cookie persistence, connection-pooling, and configuration.

max_redirects

Maximum number of redirects allowed. If the request exceeds this limit, a TooManyRedirects exception is raised. This defaults to requests.models.DEFAULT_REDIRECT_LIMIT, which is 30.

Python Requests Library Status Code Lookup

requests.**codes**

alias of <lookup 'status_codes'>

The codes object defines a mapping from common names for HTTP statuses to their numerical codes, accessible either as attributes or as dictionary items.

100: continue
101: switching_protocols
102: processing
103: checkpoint
122: uri_too_long, request_uri_too_long
200: ok, okay, all_ok, all_okay, all_good, \o/, ✓
201: created
202: accepted
203: non_authoritative_info, non_authoritative_information
204: no_content
205: reset_content, reset
206: partial_content, partial
207: multi_status, multiple_status, multi_stati, multiple_stati
208: already_reported
226: im_used
300: multiple_choices
301: moved_permanently, moved, \o-
302: found
303: see_other, other
304: not_modified
305: use_proxy
306: switch_proxy
307: temporary_redirect, temporary_moved, temporary
308: permanent_redirect, resume_incomplete, resume
400: bad_request, bad
401: unauthorized
402: payment_required, payment
403: forbidden
404: not_found, -o-
405: method_not_allowed, not_allowed
406: not_acceptable
407: proxy_authentication_required, proxy_auth, proxy_authentication
408: request_timeout, timeout
409: conflict
410: gone
411: length_required
412: precondition_failed, precondition
413: request_entity_too_large

414: request_uri_too_large
415: unsupported_media_type, unsupported_media, media_type
416: requested_range_not_satisfiable, requested_range, range_not_satisfiable
417: expectation_failed
418: im_a_teapot, teapot, i_am_a_teapot
421: misdirected_request
422: unprocessable_entity, unprocessable
423: locked
424: failed_dependency, dependency
425: unordered_collection, unordered, too_early
426: upgrade_required, upgrade
428: precondition_required, precondition
429: too_many_requests, too_many
431: header_fields_too_large, fields_too_large
444: no_response, none
449: retry_with, retry
450: blocked_by_windows_parental_controls, parental_controls
451: unavailable_for_legal_reasons, legal_reasons
499: client_closed_request
500: internal_server_error, server_error, /o\, ✗
501: not_implemented
502: bad_gateway
503: service_unavailable, unavailable
504: gateway_timeout
505: http_version_not_supported, http_version
506: variant_also_negotiates
507: insufficient_storage
509: bandwidth_limit_exceeded, bandwidth
510: not_extended
511: network_authentication_required, network_auth, network_authentication

References

Python Requests Documentation
GeeksforGeeks Python Requests Library