Requests and Responses

Scrapy uses `Request` and `Response` objects for crawling web sites.

Typically, `Request` objects are generated in the spiders and pass across the system until they reach the Downloader, which executes the request and returns a `Response` object which travels back to the spider that issued the request.

Both `Request` and `Response` classes have subclasses which add functionality not required in the base classes. These are described below in Request subclasses and Response subclasses.

Request objects

class **scrapy.http.Request**(*url* [, *callback, method='GET', headers, body, cookies, meta, encoding='utf-8', priority=0, dont_filter=False, errback, flags*])

A `Request` object represents an HTTP request, which is usually generated in the Spider and executed by the Downloader, and thus generating a `Response`.

Parameters:

- **url** (*string*) – the URL of this request
- **callback** (*callable*) – the function that will be called with the response of this request (once its downloaded) as its first parameter. For more information see Passing additional data to callback functions below. If a Request doesn't specify a callback, the spider's `parse()` method will be used. Note that if exceptions are raised during processing, errback is called instead.
- **method** (*string*) – the HTTP method of this request. Defaults to `'GET'`.
- **meta** (*dict*) – the initial values for the `Request.meta` attribute. If given, the dict passed in this parameter will be shallow copied.
- **body** (*str or unicode*) – the request body. If a `unicode` is passed, then it's encoded to `str` using the *encoding* passed (which defaults to `utf-8`). If `body` is not given, an empty string is stored. Regardless of the type of this argument, the final value stored will be a `str` (never `unicode` or `None`).
- **headers** (*dict*) – the headers of this request. The dict values can be strings (for single valued headers) or lists (for multi-valued headers). If `None` is passed as value, the HTTP header will not be sent at all.
- **cookies** (*dict or list*) –
  the request cookies. These can be sent in two forms.

  1. Using a dict:

  ```
  request_with_cookies = Request(url="http://www.example.com",
                                 cookies={'currency': 'USD',
  'country': 'UY'})
  ```

  2. Using a list of dicts:

  ```
  request_with_cookies = Request(url="http://www.example.com",
                                 cookies=[{'name': 'currency',
                                          'value': 'USD',
                                          'domain': 'example.com',
                                          'path': '/currency'}])
  ```

  The latter form allows for customizing the `domain` and `path` attributes of the cookie. This is only useful if the cookies are saved for later requests.

  When some site returns cookies (in a response) those are stored in the cookies for that domain and will be sent again in future requests. That's the typical behaviour of any regular web browser. However, if, for some reason, you want to avoid merging with existing cookies you can instruct Scrapy to do so by setting the `dont_merge_cookies` key to True in the `Request.meta`.

  Example of request without merging cookies:

  ```
  request_with_cookies = Request(url="http://www.example.com",
                                 cookies={'currency': 'USD', 'country':
  'UY'},
                                 meta={'dont_merge_cookies': True})
  ```

  For more info see CookiesMiddleware.

- **encoding** (*string*) – the encoding of this request (defaults to `'utf-8'`). This encoding will be used to percent-encode the URL and to convert the body to `str` (if given as `unicode`).
- **priority** (*int*) – the priority of this request (defaults to `0`). The priority is used by the scheduler to define the order used to process requests. Requests with a higher priority value will execute earlier. Negative values are allowed in order to indicate relatively low-priority.
- **dont_filter** (*boolean*) – indicates that this request should not be filtered by the scheduler. This is used when you want to perform an identical request multiple times, to ignore the duplicates filter. Use it with care, or you will get into crawling loops. Default to `False`.
- **errback** (*callable*) – a function that will be called if any exception was raised while processing the request. This includes pages that failed with 404 HTTP errors and such. It receives a Twisted Failure instance as first parameter. For more information, see Using errbacks to catch exceptions in request processing below.
- **flags** (*list*) – Flags sent to the request, can be used for logging or similar purposes.

`url`

A string containing the URL of this request. Keep in mind that this attribute contains the escaped URL, so it can differ from the URL passed in the constructor.

This attribute is read-only. To change the URL of a Request use `replace()`.

`method`

A string representing the HTTP method in the request. This is guaranteed to be uppercase. Example: `"GET"`, `"POST"`, `"PUT"`, etc

`headers`

A dictionary-like object which contains the request headers.

`body`

A str that contains the request body.

This attribute is read-only. To change the body of a Request use `replace()`.

`meta`

A dict that contains arbitrary metadata for this request. This dict is empty for new Requests, and is usually populated by different Scrapy components (extensions, middlewares, etc). So the data contained in this dict depends on the extensions you have enabled.

See Request.meta special keys for a list of special meta keys recognized by Scrapy.

This dict is shallow copied when the request is cloned using the `copy()` or `replace()` methods, and can also be accessed, in your spider, from the `response.meta` attribute.

> **copy()**
>
> Return a new Request which is a copy of this Request. See also: Passing additional data to callback functions.

> **replace([*url*, *method*, *headers*, *body*, *cookies*, *meta*, *encoding*, *dont_filter*, *callback*, *errback*])**
>
> Return a Request object with the same members, except for those members given new values by whichever keyword arguments are specified. The attribute `Request.meta` is copied by default (unless a new value is given in the `meta` argument). See also Passing additional data to callback functions.

Passing additional data to callback functions

The callback of a request is a function that will be called when the response of that request is downloaded. The callback function will be called with the downloaded `Response` object as its first argument.

Example:

```
def parse_page1(self, response):
    return scrapy.Request("http://www.example.com/some_page.html",
                          callback=self.parse_page2)

def parse_page2(self, response):
    # this would log http://www.example.com/some_page.html
    self.logger.info("Visited %s", response.url)
```

In some cases you may be interested in passing arguments to those callback functions so you can receive the arguments later, in the second callback. You can use the `Request.meta` attribute for that.

Here's an example of how to pass an item using this mechanism, to populate different fields from different pages:

```
def parse_page1(self, response):
    item = MyItem()
    item['main_url'] = response.url
    request = scrapy.Request("http://www.example.com/some_page.html",
                             callback=self.parse_page2)
    request.meta['item'] = item
    yield request

def parse_page2(self, response):
    item = response.meta['item']
    item['other_url'] = response.url
    yield item
```

Using errbacks to catch exceptions in request processing

The errback of a request is a function that will be called when an exception is raise while processing it.

It receives a Twisted Failure instance as first parameter and can be used to track connection establishment timeouts, DNS errors etc.

Here's an example spider logging all errors and catching some specific errors if needed:

```
import scrapy

from scrapy.spidermiddlewares.httperror import HttpError
from twisted.internet.error import DNSLookupError
from twisted.internet.error import TimeoutError, TCPTimedOutError

class ErrbackSpider(scrapy.Spider):
    name = "errback_example"
    start_urls = [
        "http://www.httpbin.org/",              # HTTP 200 expected
        "http://www.httpbin.org/status/404",    # Not found error
        "http://www.httpbin.org/status/500",    # server issue
        "http://www.httpbin.org:12345/",        # non-responding host, timeout expected
        "http://www.httphttpbinbin.org/",       # DNS error expected
    ]

    def start_requests(self):
        for u in self.start_urls:
            yield scrapy.Request(u, callback=self.parse_httpbin,
                                 errback=self.errback_httpbin,
                                 dont_filter=True)

    def parse_httpbin(self, response):
        self.logger.info('Got successful response from {}'.format(response.url))
        # do something useful here...

    def errback_httpbin(self, failure):
        # log all failures
        self.logger.error(repr(failure))

        # in case you want to do something special for some errors,
        # you may need the failure's type:

        if failure.check(HttpError):
            # these exceptions come from HttpError spider middleware
            # you can get the non-200 response
            response = failure.value.response
            self.logger.error('HttpError on %s', response.url)

        elif failure.check(DNSLookupError):
            # this is the original request
            request = failure.request
            self.logger.error('DNSLookupError on %s', request.url)

        elif failure.check(TimeoutError, TCPTimedOutError):
            request = failure.request
            self.logger.error('TimeoutError on %s', request.url)
```

Request.meta special keys

The `Request.meta` attribute can contain any arbitrary data, but there are some special keys recognized by Scrapy and its built-in extensions.

Those are:

- `dont_redirect`
- `dont_retry`
- `handle_httpstatus_list`
- `handle_httpstatus_all`
- `dont_merge_cookies` (see `cookies` parameter of `Request` constructor)
- `cookiejar`
- `dont_cache`
- `redirect_urls`
- `bindaddress`
- `dont_obey_robotstxt`
- `download_timeout`
- `download_maxsize`
- `download_latency`
- `download_fail_on_dataloss`
- `proxy`
- `ftp_user` (See `FTP_USER` for more info)
- `ftp_password` (See `FTP_PASSWORD` for more info)
- `referrer_policy`
- `max_retry_times`

bindaddress

The IP of the outgoing IP address to use for the performing the request.

download_timeout

The amount of time (in secs) that the downloader will wait before timing out. See also: `DOWNLOAD_TIMEOUT`.

download_latency

The amount of time spent to fetch the response, since the request has been started, i.e. HTTP message sent over the network. This meta key only becomes available when the response has been downloaded. While most other meta keys are used to control Scrapy behavior, this one is supposed to be read-only.

download_fail_on_dataloss

Whether or not to fail on broken responses. See: `DOWNLOAD_FAIL_ON_DATALOSS`.

max_retry_times

The meta key is used set retry times per request. When initialized, the `max_retry_times` meta key

takes higher precedence over the `RETRY_TIMES` setting.

Request subclasses

Here is the list of built-in `Request` subclasses. You can also subclass it to implement your own custom functionality.

FormRequest objects

The FormRequest class extends the base `Request` with functionality for dealing with HTML forms. It uses lxml.html forms to pre-populate form fields with form data from `Response` objects.

class **scrapy.http.FormRequest**(*url* [, *formdata*, *...*])

The `FormRequest` class adds a new argument to the constructor. The remaining arguments are the same as for the `Request` class and are not documented here.

Parameters: **formdata** (*dict or iterable of tuples*) – is a dictionary (or iterable of (key, value) tuples) containing HTML Form data which will be url-encoded and assigned to the body of the request.

The `FormRequest` objects support the following class method in addition to the standard `Request` methods:

> *classmethod* **from_response**(*response* [, *formname=None*, *formid=None*, *formnumber=0*, *formdata=None*, *formxpath=None*, *formcss=None*, *clickdata=None*, *dont_click=False*, *...*])

Returns a new `FormRequest` object with its form field values pre-populated with those found in the HTML `<form>` element contained in the given response. For an example see Using FormRequest.from_response() to simulate a user login.

The policy is to automatically simulate a click, by default, on any form control that looks clickable, like a `<input type="submit">`. Even though this is quite convenient, and often the desired behaviour, sometimes it can cause problems which could be hard to debug. For example, when working with forms that are filled and/or submitted using javascript, the default `from_response()` behaviour may not be the most appropriate. To disable this behaviour you can set the `dont_click` argument to `True`. Also, if you want to change the control clicked (instead of disabling it) you can also use the `clickdata` argument.

Caution

Using this method with select elements which have leading or trailing whitespace in the option values will not work due to a bug in lxml, which should be fixed in lxml 3.8 and above.

Parameters:

- **response** (`Response` object) – the response containing a HTML form which will be used to pre-populate the form fields
- **formname** (*string*) – if given, the form with name attribute set to this value will be used.
- **formid** (*string*) – if given, the form with id attribute set to this value will be used.
- **formxpath** (*string*) – if given, the first form that matches the xpath will be used.
- **formcss** (*string*) – if given, the first form that matches the css selector will be used.
- **formnumber** (*integer*) – the number of form to use, when the response contains multiple forms. The first one (and also the default) is `0`.
- **formdata** (*dict*) – fields to override in the form data. If a field was already present in the response `<form>` element, its value is overridden by the one passed in this parameter. If a value passed in this parameter is `None`, the field will not be included in the request, even if it was present in the response `<form>` element.
- **clickdata** (*dict*) – attributes to lookup the control clicked. If it's not given, the form data will be submitted simulating a click on the first clickable element. In addition to html attributes, the control can be identified by its zero-based index relative to other submittable inputs inside the form, via the `nr` attribute.
- **dont_click** (*boolean*) – If True, the form data will be submitted without clicking in any element.

The other parameters of this class method are passed directly to the `FormRequest` constructor.

New in version 0.10.3: The `formname` parameter.

New in version 0.17: The `formxpath` parameter.

New in version 1.1.0: The `formcss` parameter.

New in version 1.1.0: The `formid` parameter.

Request usage examples

Using FormRequest to send data via HTTP POST

If you want to simulate a HTML Form POST in your spider and send a couple of key-value fields, you can return a `FormRequest` object (from your spider) like this:

```
return [FormRequest(url="http://www.example.com/post/action",
                    formdata={'name': 'John Doe', 'age': '27'},
                    callback=self.after_post)]
```

Using FormRequest.from_response() to simulate a user login

It is usual for web sites to provide pre-populated form fields through `<input type="hidden">` elements, such as session related data or authentication tokens (for login pages). When scraping, you'll want these fields to be automatically pre-populated and only override a couple of them, such as the user name and password. You can use the `FormRequest.from_response()` method for this job. Here's an example spider which uses it:

```
import scrapy

class LoginSpider(scrapy.Spider):
    name = 'example.com'
    start_urls = ['http://www.example.com/users/login.php']

    def parse(self, response):
        return scrapy.FormRequest.from_response(
            response,
            formdata={'username': 'john', 'password': 'secret'},
            callback=self.after_login
        )

    def after_login(self, response):
        # check login succeed before going on
        if "authentication failed" in response.body:
            self.logger.error("Login failed")
            return

        # continue scraping with authenticated session...
```

Response objects

class **scrapy.http.Response**(*url* [, *status=200, headers=None, body=b'', flags=None, request=None*])

A `Response` object represents an HTTP response, which is usually downloaded (by the Downloader) and fed to the Spiders for processing.

Parameters:

- **url** (*string*) – the URL of this response
- **status** (*integer*) – the HTTP status of the response. Defaults to `200`.
- **headers** (*dict*) – the headers of this response. The dict values can be strings (for single valued headers) or lists (for multi-valued headers).
- **body** (*str*) – the response body. It must be str, not unicode, unless you're using a encoding-aware Response subclass, such as `TextResponse`.
- **flags** (*list*) – is a list containing the initial values for the `Response.flags` attribute. If given, the list will be shallow copied.
- **request** (`Request` object) – the initial value of the `Response.request` attribute. This represents the `Request` that generated this response.

`url`

A string containing the URL of the response.

This attribute is read-only. To change the URL of a Response use `replace()`.

`status`

An integer representing the HTTP status of the response. Example: `200`, `404`.

`headers`

A dictionary-like object which contains the response headers. Values can be accessed using `get()` to return the first header value with the specified name or `getlist()` to return all header values with the specified name. For example, this call will give you all cookies in the headers:

```
response.headers.getlist('Set-Cookie')
```

`body`

The body of this Response. Keep in mind that Response.body is always a bytes object. If you want the unicode version use `TextResponse.text` (only available in `TextResponse` and subclasses).

This attribute is read-only. To change the body of a Response use `replace()`.

`request`

The `Request` object that generated this response. This attribute is assigned in the Scrapy engine, after the response and the request have passed through all Downloader Middlewares. In particular, this means that:

- HTTP redirections will cause the original request (to the URL before redirection) to be assigned to the redirected response (with the final URL after redirection).
- Response.request.url doesn't always equal Response.url
- This attribute is only available in the spider code, and in the Spider Middlewares, but not in Downloader Middlewares (although you have the Request available there by other means) and handlers of the `response_downloaded` signal.

`meta`

A shortcut to the `Request.meta` attribute of the `Response.request` object (ie. `self.request.meta`).

Unlike the `Response.request` attribute, the `Response.meta` attribute is propagated along redirects and retries, so you will get the original `Request.meta` sent from your spider.

See also

`Request.meta` attribute

`flags`

A list that contains flags for this response. Flags are labels used for tagging Responses. For example: *'cached'*, *'redirected'*, etc. And they're shown on the string representation of the Response (*__str__* method) which is used by the engine for logging.

copy()

Returns a new Response which is a copy of this Response.

replace([*url*, *status*, *headers*, *body*, *request*, *flags*, *cls*])

Returns a Response object with the same members, except for those members given new values by whichever keyword arguments are specified. The attribute `Response.meta` is copied by default.

urljoin(*url*)

Constructs an absolute url by combining the Response's `url` with a possible relative url.

This is a wrapper over urlparse.urljoin, it's merely an alias for making this call:

```
urlparse.urljoin(response.url, url)
```

follow(*url*, *callback=None*, *method='GET'*, *headers=None*, *body=None*, *cookies=None*, *meta=None*, *encoding='utf-8'*, *priority=0*, *dont_filter=False*, *errback=None*)

Return a `Request` instance to follow a link `url`. It accepts the same arguments as `Request.__init__` method, but `url` can be a relative URL or a `scrapy.link.Link` object, not only an absolute URL.

`TextResponse` provides a `follow()` method which supports selectors in addition to absolute/relative URLs and Link objects.

Response subclasses

Here is the list of available built-in Response subclasses. You can also subclass the Response class to implement your own functionality.

TextResponse objects

class **scrapy.http.TextResponse**(*url* [, *encoding* [, ...]])

`TextResponse` objects adds encoding capabilities to the base `Response` class, which is meant to be used only for binary data, such as images, sounds or any media file.

`TextResponse` objects support a new constructor argument, in addition to the base `Response` objects. The remaining functionality is the same as for the `Response` class and is not documented here.

Parameters: **encoding** (*string*) – is a string which contains the encoding to use for this response. If you create a `TextResponse` object with a unicode body, it will be encoded using this encoding (remember the body attribute is always a string). If `encoding` is `None` (default value), the encoding will be looked up in the response headers and body instead.

`TextResponse` objects support the following attributes in addition to the standard `Response`

ones:

`text`

Response body, as unicode.

The same as `response.body.decode(response.encoding)`, but the result is cached after the first call, so you can access `response.text` multiple times without extra overhead.

Note

`unicode(response.body)` is not a correct way to convert response body to unicode: you would be using the system default encoding (typically *ascii*) instead of the response encoding.

`encoding`

A string with the encoding of this response. The encoding is resolved by trying the following mechanisms, in order:

1. the encoding passed in the constructor *encoding* argument
2. the encoding declared in the Content-Type HTTP header. If this encoding is not valid (ie. unknown), it is ignored and the next resolution mechanism is tried.
3. the encoding declared in the response body. The TextResponse class doesn't provide any special functionality for this. However, the `HtmlResponse` and `XmlResponse` classes do.
4. the encoding inferred by looking at the response body. This is the more fragile method but also the last one tried.

`selector`

A `Selector` instance using the response as target. The selector is lazily instantiated on first access.

`TextResponse` objects support the following methods in addition to the standard `Response` ones:

`xpath`(*query*)

A shortcut to `TextResponse.selector.xpath(query)`:

```
response.xpath('//p')
```

`css`(*query*)

A shortcut to `TextResponse.selector.css(query)`:

```
response.css('p')
```

> **follow**(*url, callback=None, method='GET', headers=None, body=None, cookies=None, meta=None, encoding=None, priority=0, dont_filter=False, errback=None*)
>
> Return a `Request` instance to follow a link `url`. It accepts the same arguments as `Request.__init__` method, but `url` can be not only an absolute URL, but also
>
> - a relative URL;
> - a scrapy.link.Link object (e.g. a link extractor result);
> - an attribute Selector (not SelectorList) - e.g. `response.css('a::attr(href)')[0]` or `response.xpath('//img/@src')[0]`.
> - a Selector for `<a>` or `<link>` element, e.g. `response.css('a.my_link')[0]`.
>
> See A shortcut for creating Requests for usage examples.

> **body_as_unicode**()
>
> The same as `text`, but available as a method. This method is kept for backwards compatibility; please prefer `response.text`.

HtmlResponse objects

class **scrapy.http.HtmlResponse**(*url* [, ...])

The `HtmlResponse` class is a subclass of `TextResponse` which adds encoding auto-discovering support by looking into the HTML meta http-equiv attribute. See `TextResponse.encoding`.

XmlResponse objects

class **scrapy.http.XmlResponse**(*url* [, ...])

The `XmlResponse` class is a subclass of `TextResponse` which adds encoding auto-discovering support by looking into the XML declaration line. See `TextResponse.encoding`.