# `!urllib.parse` --- Parse URLs into components


- URL Parsing
- URL parsing security
- Parsing ASCII Encoded Bytes
- Structured Parse Results
- URL Quoting


urllib.parse

**Source code:** `Lib/urllib/parse.py`

single: WWW single: World Wide Web single: URL pair: URL; parsing pair: relative; URL

This module defines a standard interface to break Uniform Resource Locator (URL) strings up in components (addressing scheme, network location, path etc.), to combine the components back into a URL string, and to convert a "relative URL" to an absolute URL given a "base URL."

The module has been designed to match the internet RFC on Relative Uniform Resource Locators. It supports the following URL schemes: `file`, `ftp`, `gopher`, `hdl`, `http`, `https`, `imap`, `itms-services`, `mailto`, `mms`, `news`, `nntp`, `prospero`, `rsync`, `rtsp`, `rtsps`, `rtspu`, `sftp`, `shttp`, `sip`, `sips`, `snews`, `svn`, `svn+ssh`, `telnet`, `wais`, `ws`, `wss`.

The inclusion of the `itms-services` URL scheme can prevent an app from passing Apple's App Store review process for the macOS and iOS App Stores. Handling for the `itms-services` scheme is always removed on iOS; on macOS, it *may* be removed if CPython has been built with the `--with-app-store-compliance` option.

The `!urllib.parse` module defines functions that fall into two broad categories: URL parsing and URL quoting. These are covered in detail in the following sections.

This module's functions use the deprecated term `netloc` (or `net_loc`), which was introduced in `1808`. However, this term has been obsoleted by `3986`, which introduced the term `authority` as its

replacement. The use of `netloc` is continued for backward compatibility.

# URL Parsing

The URL parsing functions focus on splitting a URL string into its components, or on combining URL components into a URL string.

urlsplit(urlstring, scheme=None, allow_fragments=True, *, missing_as_none=False)

Parse a URL into five components, returning a 5-item `named tuple` `SplitResult` or `SplitResultBytes`. This corresponds to the general structure of a URL: `scheme://netloc/path?query#fragment`. Each tuple item is a string, possibly empty, or `None` if *missing_as_none* is true. Not defined component are represented an empty string (by default) or `None` if *missing_as_none* is true. The delimiters as shown above are not part of the result, except for a leading slash in the *path* component, which is retained if present.

Additionally, the netloc property is broken down into these additional attributes added to the returned object: username, password, hostname, and port.

Percent-encoded sequences are not decoded.

For example:

>>> from urllib.parse import urlsplit >>> urlsplit("scheme://netloc/path?query#fragment") SplitResult(scheme='scheme', netloc='netloc', path='/path', query='query', fragment='fragment') >>> o = urlsplit("http://docs.python.org:80/3/library/urllib.parse.html?" ... "highlight=params#url-parsing") >>> o SplitResult(scheme='http', netloc='docs.python.org:80', path='/3/library/urllib.parse.html', query='highlight=params', fragment='url-parsing') >>> o.scheme 'http' >>> o.netloc 'docs.python.org:80' >>> o.hostname 'docs.python.org' >>> o.port 80 >>> o._replace(fragment="").geturl() 'http://docs.python.org:80/3/library/urllib.parse.html?highlight=params' >>> urlsplit("http://docs.python.org?") SplitResult(scheme='http', netloc='docs.python.org', path='', query='', fragment='') >>> urlsplit("http://docs.python.org?", missing_as_none=True) SplitResult(scheme='http', netloc='docs.python.org', path='', query='', fragment=None)

Following the syntax specifications in `1808`, `!urlsplit` recognizes a netloc only if it is properly introduced by '//'. Otherwise the input is presumed to be a relative URL and thus to start with a path component.

>>> from urllib.parse import urlsplit >>> urlsplit('//www.cwi.nl:80/%7Eguido/Python.html') SplitResult(scheme='', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html', query='', fragment='') >>> urlsplit('www.cwi.nl/%7Eguido/Python.html') SplitResult(scheme='', netloc='', path='www.cwi.nl/%7Eguido/Python.html', query='', fragment='') >>> urlsplit('help/Python.html') SplitResult(scheme='', netloc='', path='help/Python.html', query='', fragment='') >>>

urlsplit('help/Python.html', missing_as_none=True) SplitResult(scheme=None, netloc=None, path='help/Python.html', query=None, fragment=None)

The *scheme* argument gives the default addressing scheme, to be used only if the URL does not specify one. It should be the same type (text or bytes) as *urlstring* or `None`, except that the `''` is always allowed, and is automatically converted to `b''` if appropriate.

If the *allow_fragments* argument is false, fragment identifiers are not recognized. Instead, they are parsed as part of the path or query component, and `fragment` is set to `None` or the empty string (depending on the value of *missing_as_none*) in the return value.

The return value is a `named tuple`, which means that its items can be accessed by index or as named attributes, which are:

| Attribute | Index | Value | Value if not present |
|---|---|---|---|
| `scheme` | 0 | URL scheme specifier | *scheme* parameter or empty string[^1] |
| `netloc` | 1 | Network location part | `None` or empty string[^2] |
| `path` | 2 | Hierarchical path | empty string |
| `query` | 3 | Query component | `None` or empty string[^3] |
| `fragment` | 4 | Fragment identifier | `None` or empty string[^4] |
| `username` | | User name | `None` |
| `password` | | Password | `None` |
| `hostname` | | Host name (lower case) | `None` |
| `port` | | Port number as integer, if present | `None` |

Reading the `port` attribute will raise a `ValueError` if an invalid port is specified in the URL. See section `urlparse-result-object` for more information on the result object.

Unmatched square brackets in the `netloc` attribute will raise a `ValueError`.

Characters in the `netloc` attribute that decompose under NFKC normalization (as used by the IDNA encoding) into any of `/`, `?`, `#`, `@`, or `:` will raise a `ValueError`. If the URL is decomposed before parsing, no error will be raised.

Following some of the WHATWG spec that updates `3986`, leading C0 control and space characters are stripped from the URL. `\n`, `\r` and tab `\t` characters are removed from the URL at any position.

As is the case with all named tuples, the subclass has a few additional methods and attributes that are particularly useful. One such method is `_replace`. The `_replace` method will return a new `SplitResult` object replacing specified fields with new values.

>>> from urllib.parse import urlsplit >>> u = urlsplit('//www.cwi.nl:80/%7Eguido/Python.html') >>> u SplitResult(scheme='', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html', query='', fragment='') >>> u._replace(scheme='http') SplitResult(scheme='http', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html', query='', fragment='')

> [!WARNING] `urlsplit` does not perform validation. See `URL parsing security <url-parsing-security>` for details.

3.2 Added IPv6 URL parsing capabilities.

3.3 The fragment is now parsed for all URL schemes (unless *allow_fragments* is false), in accordance with `3986`. Previously, an allowlist of schemes that support fragments existed.

3.6 Out-of-range port numbers now raise `ValueError`, instead of returning `None`.

3.8 Characters that affect netloc parsing under NFKC normalization will now raise `ValueError`.

3.10 ASCII newline and tab characters are stripped from the URL.

3.12 Leading WHATWG C0 control and space characters are stripped from the URL.

3.15 Added the *missing_as_none* parameter.

parse_qs(qs, keep_blank_values=False, strict_parsing=False, encoding='utf-8', errors='replace', max_num_fields=None, separator='&')

Parse a query string given as a string argument (data of type `application/x-www-form-urlencoded`). Data are returned as a dictionary. The dictionary keys are the unique query variable names and the values are lists of values for each name.

The optional argument *keep_blank_values* is a flag indicating whether blank values in percent-encoded queries should be treated as blank strings. A true value indicates that blanks should be retained as blank strings. The default false value indicates that blank values are to be ignored and treated as if they were not included.

The optional argument *strict_parsing* is a flag indicating what to do with parsing errors. If false (the default), errors are silently ignored. If true, errors raise a `ValueError` exception.

The optional *encoding* and *errors* parameters specify how to decode percent-encoded sequences into Unicode characters, as accepted by the `bytes.decode` method.

The optional argument *max_num_fields* is the maximum number of fields to read. If set, then throws a `ValueError` if there are more than *max_num_fields* fields read.

The optional argument *separator* is the symbol to use for separating the query arguments. It defaults to `&`.

Use the `urllib.parse.urlencode` function (with the `doseq` parameter set to `True`) to convert such dictionaries into query strings.

3.2 Add *encoding* and *errors* parameters.

3.8 Added *max_num_fields* parameter.

3.10 Added *separator* parameter with the default value of `&`. Python versions earlier than Python 3.10 allowed using both `;` and `&` as query parameter separator. This has been changed to allow only a single separator key, with `&` as the default separator.

3.14 Accepting objects with false values (like `0` and `[]`) except empty strings and byte-like objects and `None` is now deprecated.

parse_qsl(qs, keep_blank_values=False, strict_parsing=False, encoding='utf-8', errors='replace', max_num_fields=None, separator='&')

Parse a query string given as a string argument (data of type `application/x-www-form-urlencoded`). Data are returned as a list of name, value pairs.

The optional argument *keep_blank_values* is a flag indicating whether blank values in percent-encoded queries should be treated as blank strings. A true value indicates that blanks should be retained as blank strings. The default false value indicates that blank values are to be ignored and treated as if they were not included.

The optional argument *strict_parsing* is a flag indicating what to do with parsing errors. If false (the default), errors are silently ignored. If true, errors raise a `ValueError` exception.

The optional *encoding* and *errors* parameters specify how to decode percent-encoded sequences into Unicode characters, as accepted by the `bytes.decode` method.

The optional argument *max_num_fields* is the maximum number of fields to read. If set, then throws a `ValueError` if there are more than *max_num_fields* fields read.

The optional argument *separator* is the symbol to use for separating the query arguments. It defaults to `&`.

Use the `urllib.parse.urlencode` function to convert such lists of pairs into query strings.

3.2 Add *encoding* and *errors* parameters.

3.8 Added *max_num_fields* parameter.

3.10 Added *separator* parameter with the default value of `&`. Python versions earlier than Python 3.10 allowed using both `;` and `&` as query parameter separator. This has been changed to allow only a single separator key, with `&` as the default separator.

urlunsplit(parts) urlunsplit(parts, *, keep_empty)

Construct a URL from a tuple as returned by `urlsplit`. The *parts* argument can be any five-item iterable.

This may result in a slightly different, but equivalent URL, if the URL that was parsed originally had unnecessary delimiters (for example, a `?` with an empty query; the RFC states that these are equivalent).

If *keep_empty* is true, empty strings are kept in the result (for example, a `?` for an empty query), only `None` components are omitted. This allows rebuilding a URL that was parsed with option `missing_as_none=True`. By default, *keep_empty* is true if *parts* is the result of the `urlsplit` call with `missing_as_none=True`.

3.15 Added the *keep_empty* parameter.

urlparse(urlstring, scheme=None, allow_fragments=True, *, missing_as_none=False)

This is similar to `urlsplit`, but additionally splits the *path* component on *path* and *params*. This function returns a 6-item `named tuple` `ParseResult` or `ParseResultBytes`. Its items are the same as for the `!urlsplit` result, except that *params* is inserted at index 3, between *path* and *query*.

This function is based on obsoleted `1738` and `1808`, which listed *params* as the main URL component. The more recent URL syntax allows parameters to be applied to each segment of the *path* portion of the URL (see `3986`). `urlsplit` should generally be used instead of `urlparse`. A separate function is needed to separate the path segments and parameters.

urlunparse(parts) urlunparse(parts, *, keep_empty)

Combine the elements of a tuple as returned by `urlparse` into a complete URL as a string. The *parts* argument can be any six-item iterable.

This may result in a slightly different, but equivalent URL, if the URL that was parsed originally had unnecessary delimiters (for example, a `?` with an empty query; the RFC states that these are equivalent).

If *keep_empty* is true, empty strings are kept in the result (for example, a `?` for an empty query), only `None` components are omitted. This allows rebuilding a URL that was parsed with option `missing_as_none=True`. By default, *keep_empty* is true if *parts* is the result of the `urlparse` call with `missing_as_none=True`.

3.15 Added the *keep_empty* parameter.

urljoin(base, url, allow_fragments=True)

Construct a full ("absolute") URL by combining a "base URL" (*base*) with another URL (*url*). Informally, this uses components of the base URL, in particular the addressing scheme, the network location and (part of) the path, to provide missing components in the relative URL. For example:

> >>> from urllib.parse import urljoin >>> urljoin('http://www.cwi.nl/%7Eguido/Python.html', 'FAQ.html') 'http://www.cwi.nl/%7Eguido/FAQ.html'

The *allow_fragments* argument has the same meaning and default as for `urlsplit`.

> [!NOTE] If *url* is an absolute URL (that is, it starts with `//` or `scheme://`), the *url*'s hostname and/or scheme will be present in the result. For example:
>
> >>> urljoin('http://www.cwi.nl/%7Eguido/Python.html', ... '//www.python.org/%7Eguido') 'http://www.python.org/%7Eguido'
>
> If you do not want that behavior, preprocess the *url* with `urlsplit` and `urlunsplit`, removing possible *scheme* and *netloc* parts.

> [!WARNING] Because an absolute URL may be passed as the `url` parameter, it is generally **not secure** to use `urljoin` with an attacker-controlled `url`. For example in, `urljoin("https://website.com/users/", username)`, if `username` can contain an absolute URL, the result of `urljoin` will be the absolute URL.

3.5

Behavior updated to match the semantics defined in `3986`.

urldefrag(url, *, missing_as_none=False)

If *url* contains a fragment identifier, return a modified version of *url* with no fragment identifier, and the fragment identifier as a separate string. If there is no fragment identifier in *url*, return *url* unmodified and an empty string (by default) or `None` if *missing_as_none* is true.

The return value is a `named tuple`, its items can be accessed by index or as named attributes:

| Attribute | Index | Value | Value if not present |
|---|---|---|---|

| | | | |
|---|---|---|---|
| `url` | 0 | URL with no fragment | empty string |
| `fragment` | 1 | Fragment identifier | `None` or empty string[^5] |

See section `urlparse-result-object` for more information on the result object.

3.2 Result is a structured object rather than a simple 2-tuple.

3.15 Added the *missing_as_none* parameter.

unwrap(url)

Extract the url from a wrapped URL (that is, a string formatted as `<URL:scheme://host/path>`, `<scheme://host/path>`, `URL:scheme://host/path` or `scheme://host/path`). If *url* is not a wrapped URL, it is returned without changes.

# URL parsing security

The `urlsplit` and `urlparse` APIs do not perform **validation** of inputs. They may not raise errors on inputs that other applications consider invalid. They may also succeed on some inputs that might not be considered URLs elsewhere. Their purpose is for practical functionality rather than purity.

Instead of raising an exception on unusual input, they may instead return some component parts as empty strings or `None` (depending on the value of the *missing_as_none* argument). Or components may contain more than perhaps they should.

We recommend that users of these APIs where the values may be used anywhere with security implications code defensively. Do some verification within your code before trusting a returned component part. Does that `scheme` make sense? Is that a sensible `path`? Is there anything strange about that `hostname`? etc.

What constitutes a URL is not universally well defined. Different applications have different needs and desired constraints. For instance the living WHATWG spec describes what user facing web clients such as a web browser require. While `3986` is more general. These functions incorporate some aspects of both, but cannot be claimed compliant with either. The APIs and existing user code with expectations on specific behaviors predate both standards leading us to be very cautious about making API behavior changes.

# Parsing ASCII Encoded Bytes

The URL parsing functions were originally designed to operate on character strings only. In practice, it is useful to be able to manipulate properly quoted and encoded URLs as sequences of

ASCII bytes. Accordingly, the URL parsing functions in this module all operate on `bytes` and `bytearray` objects in addition to `str` objects.

If `str` data is passed in, the result will also contain only `str` data. If `bytes` or `bytearray` data is passed in, the result will contain only `bytes` data.

Attempting to mix `str` data with `bytes` or `bytearray` in a single function call will result in a `TypeError` being raised, while attempting to pass in non-ASCII byte values will trigger `UnicodeDecodeError`.

To support easier conversion of result objects between `str` and `bytes`, all return values from URL parsing functions provide either an `encode` method (when the result contains `str` data) or a `decode` method (when the result contains `bytes` data). The signatures of these methods match those of the corresponding `str` and `bytes` methods (except that the default encoding is `'ascii'` rather than `'utf-8'`). Each produces a value of a corresponding type that contains either `bytes` data (for `encode` methods) or `str` data (for `decode` methods).

Applications that need to operate on potentially improperly quoted URLs that may contain non-ASCII data will need to do their own decoding from bytes to characters before invoking the URL parsing methods.

The behaviour described in this section applies only to the URL parsing functions. The URL quoting functions use their own rules when producing or consuming byte sequences as detailed in the documentation of the individual URL quoting functions.

3.2 URL parsing functions now accept ASCII encoded byte sequences

# Structured Parse Results

The result objects from the `urlsplit`, `urlparse` and `urldefrag` functions are subclasses of the `tuple` type. These subclasses add the attributes listed in the documentation for those functions, the encoding and decoding support described in the previous section, as well as an additional method:

urllib.parse.SplitResult.geturl()

Return the re-combined version of the original URL as a string. This may differ from the original URL in that the scheme may be normalized to lower case and empty components may be dropped. Specifically, empty parameters, queries, and fragment identifiers will be removed unless the URL was parsed with `missing_as_none=True`.

For `urldefrag` results, only empty fragment identifiers will be removed. For `urlsplit` and `urlparse` results, all noted changes will be made to the URL returned by this method.

The result of this method remains unchanged if passed back through the original parsing function:

> >>> from urllib.parse import urlsplit >>> url = 'HTTP://www.Python.org/doc/#' >>> r1 = urlsplit(url) >>> r1.geturl() 'http://www.Python.org/doc/' >>> r2 = urlsplit(r1.geturl()) >>> r2.geturl() 'http://www.Python.org/doc/' >>> r3 = urlsplit(url, missing_as_none=True) >>> r3.geturl() 'http://www.Python.org/doc/#'

The following classes provide the implementations of the structured parse results when operating on `str` objects:

Concrete class for `urldefrag` results containing `str` data. The `encode` method returns a `DefragResultBytes` instance.

3.2

Concrete class for `urlparse` results containing `str` data. The `encode` method returns a `ParseResultBytes` instance.

Concrete class for `urlsplit` results containing `str` data. The `encode` method returns a `SplitResultBytes` instance.

The following classes provide the implementations of the parse results when operating on `bytes` or `bytearray` objects:

Concrete class for `urldefrag` results containing `bytes` data. The `decode` method returns a `DefragResult` instance.

3.2

Concrete class for `urlparse` results containing `bytes` data. The `decode` method returns a `ParseResult` instance.

3.2

Concrete class for `urlsplit` results containing `bytes` data. The `decode` method returns a `SplitResult` instance.

3.2

# URL Quoting

The URL quoting functions focus on taking program data and making it safe for use as URL components by quoting special characters and appropriately encoding non-ASCII text. They also support reversing these operations to recreate the original data from the contents of a URL component if that task isn't already covered by the URL parsing functions above.

quote(string, safe='/', encoding=None, errors=None)

Replace special characters in *string* using the `%{xx}` escape. Letters, digits, and the characters `'_.-~'` are never quoted. By default, this function is intended for quoting the path section of a URL. The optional *safe* parameter specifies additional ASCII characters that should not be quoted --- its default value is `'/'`.

*string* may be either a `str` or a `bytes` object.

3.7 Moved from `2396` to `3986` for quoting URL strings. "~" is now included in the set of unreserved characters.

The optional *encoding* and *errors* parameters specify how to deal with non-ASCII characters, as accepted by the `str.encode` method. Although these parameters default to `None` in the function signature, when processing `str` inputs, *encoding* effectively defaults to `'utf-8'` and *errors* to `'strict'`, meaning unsupported characters raise a `UnicodeEncodeError`. *encoding* and *errors* must not be supplied if *string* is a `bytes`, or a `TypeError` is raised.

Note that `quote(string, safe, encoding, errors)` is equivalent to `quote_from_bytes(string.encode(encoding, errors), safe)`.

Example: `quote('/El Niño/')` yields `'/El%20Ni%C3%B1o/'`.

quote_plus(string, safe='', encoding=None, errors=None)

Like `quote`, but also replace spaces with plus signs, as required for quoting HTML form values when building up a query string to go into a URL. Plus signs in the original string are escaped unless they are included in *safe*. It also does not have *safe* default to `'/'`.

Example: `quote_plus('/El Niño/')` yields `'%2FEl+Ni%C3%B1o%2F'`.

quote_from_bytes(bytes, safe='/')

Like `quote`, but accepts a `bytes` object rather than a `str`, and does not perform string-to-bytes encoding.

Example: `quote_from_bytes(b'a&\xef')` yields `'a%26%EF'`.

unquote(string, encoding='utf-8', errors='replace')

Replace `%{xx}` escapes with their single-character equivalent. The optional *encoding* and *errors* parameters specify how to decode percent-encoded sequences into Unicode characters, as accepted by the `bytes.decode` method.

*string* may be either a `str` or a `bytes` object.

*encoding* defaults to `'utf-8'`. *errors* defaults to `'replace'`, meaning invalid sequences are replaced by a placeholder character.

Example: `unquote('/El%20Ni%C3%B1o/')` yields `'/El Niño/'`.

3.9 *string* parameter supports bytes and str objects (previously only str).

unquote_plus(string, encoding='utf-8', errors='replace')

Like `unquote`, but also replace plus signs with spaces, as required for unquoting HTML form values.

*string* must be a `str`.

Example: `unquote_plus('/El+Ni%C3%B1o/')` yields `'/El Niño/'`.

unquote_to_bytes(string)

Replace `%{xx}` escapes with their single-octet equivalent, and return a `bytes` object.

*string* may be either a `str` or a `bytes` object.

If it is a `str`, unescaped non-ASCII characters in *string* are encoded into UTF-8 bytes.

Example: `unquote_to_bytes('a%26%EF')` yields `b'a&\xef'`.

urlencode(query, doseq=False, safe='', encoding=None, errors=None, quote_via=quote_plus)

Convert a mapping object or a sequence of two-element tuples, which may contain `str` or `bytes` objects, to a percent-encoded ASCII text string. If the resultant string is to be used as a *data* for POST operation with the `~urllib.request.urlopen` function, then it should be encoded to bytes, otherwise it would result in a `TypeError`.

The resulting string is a series of `key=value` pairs separated by `'&'` characters, where both *key* and *value* are quoted using the *quote_via* function. By default, `quote_plus` is used to quote the values, which means spaces are quoted as a `'+'` character and '/' characters are encoded as `%2F`, which follows the standard for GET requests (`application/x-www-form-urlencoded`). An alternate function that can be passed as *quote_via* is `quote`, which will encode spaces as `%20` and not encode '/' characters. For maximum control of what is quoted, use `quote` and specify a value for *safe*.

When a sequence of two-element tuples is used as the *query* argument, the first element of each tuple is a key and the second is a value. The value element in itself can be a sequence and in that case, if the optional parameter *doseq* evaluates to `True`, individual `key=value` pairs separated by `'&'` are generated for each element of the value sequence for the key. The order of parameters in the encoded string will match the order of parameter tuples in the sequence.

The *safe*, *encoding*, and *errors* parameters are passed down to *quote_via* (the *encoding* and *errors* parameters are only passed when a query element is a `str`).

To reverse this encoding process, `parse_qs` and `parse_qsl` are provided in this module to parse query strings into Python data structures.

Refer to `urllib examples <urllib-examples>` to find out how the `urllib.parse.urlencode` method can be used for generating the query string of a URL or data for a POST request.

3.2 *query* supports bytes and string objects.

3.5 Added the *quote_via* parameter.

3.14 Accepting objects with false values (like `0` and `[]`) except empty strings and byte-like objects and `None` is now deprecated.

WHATWG - URL Living standard
Working Group for the URL Standard that defines URLs, domains, IP addresses, the application/x-www-form-urlencoded format, and their API.

`3986` - Uniform Resource Identifiers
This is the current standard (STD66). Any changes to urllib.parse module should conform to this. Certain deviations could be observed, which are mostly for backward compatibility purposes and for certain de-facto parsing requirements as commonly observed in major browsers.

`2732` - Format for Literal IPv6 Addresses in URL's.
This specifies the parsing requirements of IPv6 URLs.

`2396` - Uniform Resource Identifiers (URI): Generic Syntax
Document describing the generic syntactic requirements for both Uniform Resource Names (URNs) and Uniform Resource Locators (URLs).

`2368` - The mailto URL scheme.
Parsing requirements for mailto URL schemes.

`1808` - Relative Uniform Resource Locators
This Request For Comments includes the rules for joining an absolute and a relative URL, including a fair number of "Abnormal Examples" which govern the treatment of border cases.

`1738` - Uniform Resource Locators (URL)
This specifies the formal syntax and semantics of absolute URLs.

[^1]: Depending on the value of the *missing_as_none* argument.

[^2]: Depending on the value of the *missing_as_none* argument.

[^3]: Depending on the value of the *missing_as_none* argument.

[^4]: Depending on the value of the *missing_as_none* argument.

[^5]: Depending on the value of the *missing_as_none* argument.

---

□□□ #2
□ □□□ □□□ 2026-10-31 17:23:27 UTC
□ □□□ □□□ 2026-10-31 17:23:30 UTC