
hyperlink Documentation

Release 18.0.0

Mahmoud Hashemi

May 02, 2018

Contents

1	Installation and Integration	3
2	Gaps	5
3	Table of Contents	7
3.1	Hyperlink Design	7
3.2	Hyperlink API	9
3.3	FAQ	16
	Python Module Index	19

Cool URLs that don't change.

Hyperlink provides a pure-Python implementation of immutable URLs. Based on [RFC 3986](#) and [RFC 3987](#), the Hyperlink URL balances simplicity and correctness for both *URIs and IRIs*.

Hyperlink is tested against Python 2.7, 3.4, 3.5, 3.6, and PyPy.

For an introduction to the hyperlink library, its background, and URLs in general, see [this talk from PyConWeb 2017](#) (and [the accompanying slides](#)).

Installation and Integration

Hyperlink is a pure-Python package and only depends on the standard library. The easiest way to install is with pip:

```
pip install hyperlink
```

Then, URLs are just an import away:

```
from hyperlink import URL

url = URL.from_text(u'http://github.com/python-hyper/hyperlink?utm_source=readthedocs
→')

better_url = url.replace(scheme=u'https', port=443)
org_url = better_url.click(u'.')

print(org_url.to_text())
# prints: https://github.com/python-hyper/

print(better_url.get(u'utm_source'))
# prints: readthedocs
```

See *the API docs* for more usage examples.

CHAPTER 2

Gaps

Found something missing in hyperlink? [Pull Requests](#) and [Issues](#) are welcome!

3.1 Hyperlink Design

The URL is a nuanced format with a long history. Suitably, a lot of work has gone into translating the standards, [RFC 3986](#) and [RFC 3987](#), into a Pythonic interface. Hyperlink's design strikes a unique balance of correctness and usability.

3.1.1 A Tale of Two Representations

The URL is a powerful construct, designed to be used by both humans and computers.

This dual purpose has resulted in two canonical representations: the URI and the IRI.

Even though the W3C themselves have [recognized the confusion](#) this can cause, Hyperlink's URL makes the distinction quite natural. Simply:

- **URI**: Fully-encoded, ASCII-only, suitable for network transfer
- **IRI**: Fully-decoded, Unicode-friendly, suitable for display (e.g., in a browser bar)

We can use Hyperlink to very easily demonstrate the difference:

```
>>> url = URL.from_text('http://example.com/café')
>>> url.to_uri().to_text()
u'http://example.com/café'
```

We construct a URL from text containing Unicode (é), then transform it using `to_uri()`. This results in ASCII-only percent-encoding familiar to all web developers, and a common characteristic of URIs.

Still, Hyperlink's distinction between URIs and IRIs is pragmatic, and only limited to output. Input can contain *any mix* of percent encoding and Unicode, without issue:

```
>>> url = URL.from_text("http://example.com/café au lait/s'il vous plaît!")
>>> print(url.to_iri().to_text())
```

(continues on next page)

(continued from previous page)

```
http://example.com/café au lait/s'il vous plaît!
>>> print(url.to_uri().to_text())
http://example.com/caf%C3%A9%20au%20lait/s'il%20vous%20pla%C3%AEt!
```

Note that even when a URI and IRI point to the same resource, they will often be different URLs:

```
>>> url.to_uri() == url.to_iri()
False
```

And with that caveat out of the way, you're qualified to correct other people (and their code) on the nuances of URI vs IRI.

3.1.2 Immutability

Hyperlink's URL is notable for being an *immutable* representation. Once constructed, instances are not changed. Methods like `click()`, `set()`, and `replace()`, all return new URL objects. This enables URLs to be used in sets, as well as dictionary keys.

3.1.3 Query parameters

One of the URL format's most useful features is the mapping formed by the query parameters, sometimes called “query arguments” or “GET parameters”. Regardless of what you call them, they are encoded in the query string portion of the URL, and they are very powerful.

In the simplest case, these query parameters can be provided as a dictionary:

```
>>> url = URL.from_text('http://example.com/')
>>> url = url.replace(query={'a': 'b', 'c': 'd'})
>>> url.to_text()
u'http://example.com/?a=b&c=d'
```

Query parameters are actually a type of “multidict”, where a given key can have multiple values. This is why the `get()` method returns a list of strings. Keys can also have no value, which is conventionally interpreted as a truthy flag.

```
>>> url = URL.from_text('http://example.com/?a=b&c')
>>> url.get(u'a')
['b']
>>> url.get(u'c')
[None]
>>> url.get('missing') # returns None
[]
```

Values can be modified and added using `set()` and `add()`.

```
>>> url = url.add(u'x', u'x')
>>> url = url.add(u'x', u'y')
>>> url.to_text()
u'http://example.com/?a=b&c&x=x&x=y'
>>> url = url.set(u'x', u'z')
>>> url.to_text()
u'http://example.com/?a=b&c&x=z'
```

Values can be unset with `remove()`.


```
>>> from hyperlink import URL
>>> url = URL(scheme='https', host='example.com', path=[u'hello', u'world'])
>>> print(url.to_text())
https://example.com/hello/world
```

The constructor runs basic type checks. All strings are expected to be decoded (unicode in Python 2). All arguments are optional, defaulting to appropriately empty values. A full list of constructor arguments is below.

Parameters

- **scheme** (*unicode*) – The text name of the scheme.
- **host** (*unicode*) – The host portion of the network location
- **port** (*int*) – The port part of the network location. If *None* or no port is passed, the port will default to the default port of the scheme, if it is known. See the `SCHEME_PORT_MAP` and `register_default_port()` for more info.
- **path** (*tuple*) – A tuple of strings representing the slash-separated parts of the path.
- **query** (*tuple*) – The query parameters, as a dictionary or as an iterable of key-value pairs.
- **fragment** (*unicode*) – The fragment part of the URL.
- **rooted** (*bool*) – Whether or not the path begins with a slash.
- **userinfo** (*unicode*) – The username or colon-separated username:password pair.
- **uses_netloc** (*bool*) – Indicates whether two slashes appear between the scheme and the host (`http://eg.com` vs `mailto:e@g.com`). Set automatically based on scheme.

All of these parts are also exposed as read-only attributes of `URL` instances, along with several useful methods.

classmethod `URL.from_text(text)`

Whereas the `URL` constructor is useful for constructing URLs from parts, `from_text()` supports parsing whole URLs from their string form:

```
>>> URL.from_text(u'http://example.com')
URL.from_text(u'http://example.com')
>>> URL.from_text(u'?a=b&x=y')
URL.from_text(u'?a=b&x=y')
```

As you can see above, it's also used as the `repr()` of `URL` objects. The natural counterpart to `to_text()`. This method only accepts *text*, so be sure to decode those bytestrings.

Parameters *text* (*unicode*) – A valid URL string.

Returns The structured object version of the parsed string.

Return type `URL`

Note: Somewhat unexpectedly, URLs are a far more permissive format than most would assume. Many strings which don't look like URLs are still valid URLs. As a result, this method only raises `URLParseError` on invalid port and IPv6 values in the host portion of the URL.

3.2.2 Transformation

Once a URL is created, some of the most common tasks are to transform it into other URLs and text.

`URL.to_text` (*with_password=False*)

Render this URL to its textual representation.

By default, the URL text will *not* include a password, if one is set. RFC 3986 considers using URLs to represent such sensitive information as deprecated. Quoting from RFC 3986, *section 3.2.1*:

“Applications should not render as clear text any data after the first colon (“:”) character found within a userinfo subcomponent unless the data after the colon is the empty string (indicating no password).”

Parameters `with_password` (*bool*) – Whether or not to include the password in the URL text. Defaults to False.

Returns The serialized textual representation of this URL, such as `u"http://example.com/some/path?some=query"`.

Return type `str`

The natural counterpart to `URL.from_text()`.

`URL.to_uri()`

Make a new `URL` instance with all non-ASCII characters appropriately percent-encoded. This is useful to do in preparation for sending a `URL` over a network protocol.

For example:

```
>>> URL.from_text(u'https://.com/foobar/').to_uri()
URL.from_text(u'https://xn--mgb3a4fra.com/foo%E2%87%A7bar/')
```

Returns A new instance with its path segments, query parameters, and hostname encoded, so that they are all in the standard US-ASCII range.

Return type `URL`

`URL.to_iri()`

Make a new `URL` instance with all but a few reserved characters decoded into human-readable format.

Percent-encoded Unicode and IDNA-encoded hostnames are decoded, like so:

```
>>> url = URL.from_text(u'https://xn--mgb3a4fra.example.com/foo%E2%87%A7bar/')
>>> print(url.to_iri().to_text())
https://.example.com/foobar/
```

Note: As a general Python issue, “narrow” (UCS-2) builds of Python may not be able to fully decode certain URLs, and in those cases, this method will return a best-effort, partially-decoded, URL which is still valid. This issue does not affect any Python builds 3.4+.

Returns A new instance with its path segments, query parameters, and hostname decoded for display purposes.

Return type `URL`

`URL.replace` (*scheme=Sentinel('_UNSET')*, *host=Sentinel('_UNSET')*, *path=Sentinel('_UNSET')*, *query=Sentinel('_UNSET')*, *fragment=Sentinel('_UNSET')*, *port=Sentinel('_UNSET')*, *rooted=Sentinel('_UNSET')*, *userinfo=Sentinel('_UNSET')*, *uses_netloc=Sentinel('_UNSET')*)

`URL` objects are immutable, which means that attributes are designed to be set only once, at construction. Instead of modifying an existing URL, one simply creates a copy with the desired changes.

If any of the following arguments is omitted, it defaults to the value on the current URL.

Parameters

- **scheme** (*unicode*) – The text name of the scheme.
- **host** (*unicode*) – The host portion of the network location
- **port** (*int*) – The port part of the network location.
- **path** (*tuple*) – A tuple of strings representing the slash-separated parts of the path.
- **query** (*tuple*) – The query parameters, as a tuple of key-value pairs.
- **query** – The query parameters, as a dictionary or as an iterable of key-value pairs.
- **fragment** (*unicode*) – The fragment part of the URL.
- **rooted** (*bool*) – Whether or not the path begins with a slash.
- **userinfo** (*unicode*) – The username or colon-separated username:password pair.
- **uses_netloc** (*bool*) – Indicates whether two slashes appear between the scheme and the host (`http://eg.com` vs `mailto:eg.com`)

Returns

a copy of the current [URL](#), with new values for parameters passed.

Return type [URL](#)

`URL.normalize` (*scheme=True, host=True, path=True, query=True, fragment=True, userinfo=True, percents=True*)

Return a new URL object with several standard normalizations applied:

- Decode unreserved characters ([RFC 3986 2.3](#))
- Uppercase remaining percent-encoded octets ([RFC 3986 2.1](#))
- Convert scheme and host casing to lowercase ([RFC 3986 3.2.2](#))
- Resolve any “.” and “..” references in the path ([RFC 3986 6.2.2.3](#))
- Ensure an ending slash on URLs with an empty path ([RFC 3986 6.2.3](#))
- Encode any stray percent signs (%) in percent-encoded fields (path, query, fragment, userinfo) ([RFC 3986 2.4](#))

All are applied by default, but normalizations can be disabled per-part by passing *False* for that part’s corresponding name.

Parameters

- **scheme** (*bool*) – Convert the scheme to lowercase
- **host** (*bool*) – Convert the host to lowercase
- **path** (*bool*) – Normalize the path (see above for details)
- **query** (*bool*) – Normalize the query string
- **fragment** (*bool*) – Normalize the fragment
- **userinfo** (*bool*) – Normalize the userinfo
- **percents** (*bool*) – Encode isolated percent signs for any percent-encoded fields which are being normalized (defaults to True).

```
>>> url = URL.from_text('Http://example.COM/a/./b/./c%2f?%61%')
>>> print(url.normalize().to_text())
http://example.com/b/c%2F?a%25
```

3.2.3 Navigation

Go places with URLs. Simulate browser behavior and perform semantic path operations.

`URL.click(href=u"")`

Resolve the given URL relative to this URL.

The resulting URI should match what a web browser would generate if you visited the current URL and clicked on *href*.

```
>>> url = URL.from_text('http://blog.hatnote.com/')
>>> url.click(u'/post/155074058790').to_text()
u'http://blog.hatnote.com/post/155074058790'
>>> url = URL.from_text('http://localhost/a/b/c/')
>>> url.click(u'../d/./e').to_text()
u'http://localhost/a/b/d/e'
```

Parameters `href` (*unicode*) – A string representing a clicked URL.

Returns A copy of the current URL with navigation logic applied.

Return type *URL*

For more information, see [RFC 3986 section 5](#).

`URL.sibling(segment)`

Make a new *URL* with a single path segment that is a sibling of this URL path.

Parameters `segment` (*unicode*) – A single path segment.

Returns

A copy of the current URL with the last path segment replaced by *segment*. Special characters such as `/?#` will be percent encoded.

Return type *URL*

`URL.child(*segments)`

Make a new *URL* where the given path segments are a child of this URL, preserving other parts of the URL, including the query string and fragment.

For example:

```
>>> url = URL.from_text('http://localhost/a/b?x=y')
>>> child_url = url.child(u"c", u"d")
>>> child_url.to_text()
u'http://localhost/a/b/c/d?x=y'
```

Parameters `segments` (*unicode*) – Additional parts to be joined and added to the path, like `os.path.join()`. Special characters in segments will be percent encoded.

Returns A copy of the current URL with the extra path segments.

Return type *URL*

3.2.4 Query Parameters

CRUD operations on the query string multimap.

`URL.get` (*name*)

Get a list of values for the given query parameter, *name*:

```
>>> url = URL.from_text(u'?x=1&x=2')
>>> url.get('x')
[u'1', u'2']
>>> url.get('y')
[]
```

If the given *name* is not set, an empty list is returned. A list is always returned, and this method raises no exceptions.

Parameters *name* (*unicode*) – The name of the query parameter to get.

Returns

A list of all the values associated with the key, in string form.

Return type list

`URL.add` (*name*, *value=None*)

Make a new `URL` instance with a given query argument, *name*, added to it with the value *value*, like so:

```
>>> URL.from_text(u'https://example.com/?x=y').add(u'x')
URL.from_text(u'https://example.com/?x=y&x')
>>> URL.from_text(u'https://example.com/?x=y').add(u'x', u'z')
URL.from_text(u'https://example.com/?x=y&x=z')
```

Parameters

- **name** (*unicode*) – The name of the query parameter to add. The part before the =.
- **value** (*unicode*) – The value of the query parameter to add. The part after the =. Defaults to `None`, meaning no value.

Returns A new `URL` instance with the parameter added.

Return type `URL`

`URL.set` (*name*, *value=None*)

Make a new `URL` instance with the query parameter *name* set to *value*. All existing occurrences, if any are replaced by the single name-value pair.

```
>>> URL.from_text(u'https://example.com/?x=y').set(u'x')
URL.from_text(u'https://example.com/?x')
>>> URL.from_text(u'https://example.com/?x=y').set(u'x', u'z')
URL.from_text(u'https://example.com/?x=z')
```

Parameters

- **name** (*unicode*) – The name of the query parameter to set. The part before the =.
- **value** (*unicode*) – The value of the query parameter to set. The part after the =. Defaults to `None`, meaning no value.

Returns A new `URL` instance with the parameter set.

Return type `URL`

`URL.remove(name)`

Make a new `URL` instance with all occurrences of the query parameter `name` removed. No exception is raised if the parameter is not already set.

Parameters `name` (*unicode*) – The name of the query parameter to remove.

Returns A new `URL` instance with the parameter removed.

Return type `URL`

3.2.5 Attributes

URLs have many parts, and URL objects have many attributes to represent them.

`URL.absolute`

Whether or not the URL is “absolute”. Absolute URLs are complete enough to resolve to a network resource without being relative to a base URI.

```
>>> URL.from_text('http://wikipedia.org/').absolute
True
>>> URL.from_text('?a=b&c=d').absolute
False
```

Absolute URLs must have both a scheme and a host set.

`URL.scheme`

The scheme is a string, and the first part of an absolute URL, the part before the first colon, and the part which defines the semantics of the rest of the URL. Examples include “http”, “https”, “ssh”, “file”, “mailto”, and many others. See `register_scheme()` for more info.

`URL.host`

The host is a string, and the second standard part of an absolute URL. When present, a valid host must be a domain name, or an IP (v4 or v6). It occurs before the first slash, or the second colon, if a `port` is provided.

`URL.port`

The port is an integer that is commonly used in connecting to the `host`, and almost never appears without it.

When not present in the original URL, this attribute defaults to the scheme’s default port. If the scheme’s default port is not known, and the port is not provided, this attribute will be set to `None`.

```
>>> URL.from_text('http://example.com/path').port
80
>>> URL.from_text('foo://example.com/path').port
>>> URL.from_text('foo://example.com:8042/path').port
8042
```

Note: Per the standard, when the port is the same as the schemes default port, it will be omitted in the text URL.

`URL.path`

A tuple of strings, created by splitting the slash-separated hierarchical path. Started by the first slash after the host, terminated by a “?”, which indicates the start of the `query` string.

`URL.query`

Tuple of pairs, created by splitting the ampersand-separated mapping of keys and optional values representing non-hierarchical data used to identify the resource. Keys are always strings. Values are strings when present, or `None` when missing.

For more operations on the mapping, see `get()`, `add()`, `set()`, and `delete()`.

URL.fragment

A string, the last part of the URL, indicated by the first “#” after the `path` or `query`. Enables indirect identification of a secondary resource, like an anchor within an HTML page.

URL.userinfo

The colon-separated string forming the username-password combination.

URL.user

The user portion of `userinfo`.

URL.rooted

Whether or not the path starts with a forward slash (/).

This is taken from the terminology in the BNF grammar, specifically the “path-rootless”, rule, since “absolute path” and “absolute URI” are somewhat ambiguous. `path` does not contain the implicit prefixed “/” since that is somewhat awkward to work with.

3.2.6 Low-level functions

A couple of notable helpers used by the `URL` type.

class `hyperlink.URLParseError`

Exception inheriting from `ValueError`, raised when failing to parse a URL. Mostly raised on invalid ports and IPv6 addresses.

`hyperlink.register_scheme` (*text*, *uses_netloc=True*, *default_port=None*)

Registers new scheme information, resulting in correct port and slash behavior from the `URL` object. There are dozens of standard schemes preregistered, so this function is mostly meant for proprietary internal customizations or stopgaps on missing standards information. If a scheme seems to be missing, please [file an issue!](#)

Parameters

- **text** (*unicode*) – Text representing the scheme. (the ‘http’ in ‘http://hatnote.com’)
- **uses_netloc** (*bool*) – Does the scheme support specifying a network host? For instance, “http” does, “mailto” does not. Defaults to True.
- **default_port** (*int*) – The default port, if any, for netloc-using schemes.

3.3 FAQ

There were bound to be questions.

- *Why not just use text?*
- *How does Hyperlink compare to other libraries?*
- *Are URLs really a big deal in 201X?*

3.3.1 Why not just use text?

URLs were designed as a text format, so, apart from the principle of structuring structured data, why use `URL` objects?

There are two major advantages of using *URL* over representing URLs as strings. The first is that it's really easy to evaluate a relative hyperlink, for example, when crawling documents, to figure out what is linked:

```
>>> URL.from_text(u'https://example.com/base/uri/').click(u"/absolute")
URL.from_text(u'https://example.com/absolute')
>>> URL.from_text(u'https://example.com/base/uri/').click(u"rel/path")
URL.from_text(u'https://example.com/base/uri/rel/path')
```

The other is that URLs have two normalizations. One representation is suitable for humans to read, because it can represent data from many character sets - this is the Internationalized, or IRI, normalization. The other is the older, US-ASCII-only representation, which is necessary for most contexts where you would need to put a URI. You can convert *between* these representations according to certain rules. *URL* exposes these conversions as methods:

```
>>> URL.from_text(u"https://→example.com/foobar/").to_uri()
URL.from_text(u'https://xn--example-dk9c.com/foo%E2%87%A7bar/')
>>> URL.from_text(u'https://xn--example-dk9c.com/foo%E2%87%A7bar/').to_iri()
URL.from_text(u'https://\u2192example.com/foo\u21e7bar/')
```

For more info, see *A Tale of Two Representations*, above.

3.3.2 How does Hyperlink compare to other libraries?

Hyperlink certainly isn't the first library to provide a Python model for URLs. It just happens to be among the best.

urlparse: Built-in to the standard library (merged into *urllib* for Python 3). No URL type, requires user to juggle a bunch of strings. Overly simple approach makes it easy to make mistakes.

boltons.urlutils: Shares some underlying implementation. Two key differences. First, the *boltons* URL is mutable, intended to work like a string factory for URL text. Second, the *boltons* URL has advanced query parameter mapping type. Complete implementation in a single file.

furl: Not a single URL type, but types for many parts of the URL. Similar approach to *boltons* for query parameters. Poor netloc handling (support for non-network schemes like *mailto*). Unlicensed.

purl: Another immutable implementation. Method-heavy API.

rfc3986: Very heavily focused on various types of validation. Large for a URL library, if that matters to you. Exclusively supports URIs, *lacking IRI support* at the time of writing.

In reality, any of the third-party libraries above do a better job than the standard library, and much of the hastily thrown together code in a corner of a *util.py* deep in a project. URLs are easy to mess up, make sure you use a tested implementation.

3.3.3 Are URLs really a big deal in 201X?

Hyperlink's first release, in 2017, comes somewhere between 23 and 30 years after URLs were already in use. Is the URL really still that big of a deal?

Look, buddy, I don't know how you got this document, but I'm pretty sure you (and your computer) used one if not many URLs to get here. URLs are only getting more relevant. Buy stock in URLs.

And if you're worried that URLs are just another technology with an obsolescence date planned in advance, I'll direct your attention to the *IPvFuture* rule in the *BNF grammar*. If it has plans to outlast IPv6, the URL will probably outlast you and me, too.

h

hyperlink._url, 9

A

absolute (hyperlink.URL attribute), 15
add() (hyperlink.URL method), 14

C

child() (hyperlink.URL method), 13
click() (hyperlink.URL method), 13

F

fragment (hyperlink.URL attribute), 16
from_text() (hyperlink.URL class method), 10

G

get() (hyperlink.URL method), 14

H

host (hyperlink.URL attribute), 15
hyperlink._url (module), 9

N

normalize() (hyperlink.URL method), 12

P

path (hyperlink.URL attribute), 15
port (hyperlink.URL attribute), 15

Q

query (hyperlink.URL attribute), 15

R

register_scheme() (in module hyperlink), 16
remove() (hyperlink.URL method), 15
replace() (hyperlink.URL method), 11
rooted (hyperlink.URL attribute), 16

S

scheme (hyperlink.URL attribute), 15
set() (hyperlink.URL method), 14

sibling() (hyperlink.URL method), 13

T

to_iri() (hyperlink.URL method), 11
to_text() (hyperlink.URL method), 10
to_uri() (hyperlink.URL method), 11

U

URL (class in hyperlink), 9
URLParseError (class in hyperlink), 16
user (hyperlink.URL attribute), 16
userinfo (hyperlink.URL attribute), 16