hyperlink Documentation

Release 20.0.1

Mahmoud Hashemi

January 08, 2021

Contents

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, 3.7, 3.8, 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).

CHAPTER 1

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:

```
import hyperlink

url = hyperlink.parse(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')[0])
# prints: readthedocs
```

See *the API docs* for more usage examples.

CHAPTER 2

Gaps

Found something missing in hyperlink? Pull Requests and Issues are welcome!

CHAPTER 3

Table of Contents

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/caf%C3%A9'
```

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/caf%C3%A9 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()`.

```
>>> url = url.remove(u'a')
>>> url = url.remove(u'c')
>>> url.to_text()
u'http://example.com/?x=z'
```

Note how all modifying methods return copies of the URL and do not mutate the URL in place, much like methods on strings.

3.1.4 Origins and backwards-compatibility

Hyperlink's URL is descended directly from twisted.python.url.URL, in all but the literal code-inheritance sense. While a lot of functionality has been incorporated from boltons.urlutils, extra care has been taken to maintain backwards-compatibility for legacy APIs, making Hyperlink's URL a drop-in replacement for Twisted's URL type.

If you are porting a Twisted project to use Hyperlink's URL, and encounter any sort of incompatibility, please do not hesitate to file an issue.

3.2 Hyperlink API

Hyperlink provides Pythonic URL parsing, construction, and rendering.

Usage is straightforward:

```
>>> import hyperlink
>>> url = hyperlink.parse(u'http://github.com/mahmoud/hyperlink?utm_source=docs')
>>> url.host
u'github.com'
>>> secure_url = url.replace(scheme=u'https')
>>> secure_url.get('utm_source')[0]
u'docs'
```

Hyperlink's API centers on the `DecodedURL` type, which wraps the lower-level `URL`, both of which can be returned by the `parse()` convenience function.

3.2.1 Creation

Before you can work with URLs, you must create URLs.

Parsing Text

If you already have a textual URL, the easiest way to get URL objects is with the `parse()` function:

`hyperlink.`**`parse`**(*url*, *decoded=True*, *lazy=False*)

Automatically turn text into a structured URL object.

```
>>> url = parse(u"https://github.com/python-hyper/hyperlink")
>>> print(url.to_text())
https://github.com/python-hyper/hyperlink
```

Parameters

- **`url`** – A text string representation of a URL.
- **`decoded`** – Whether or not to return a *DecodedURL*, which automatically handles all encoding/decoding/quoting/unquoting for all the various accessors of parts of the URL, or a *URL*, which has the same API, but requires handling of special characters for different parts of the URL.
- **`lazy`** – In the case of *decoded=True*, this controls whether the URL is decoded immediately or as accessed. The default, *lazy=False*, checks all encoded parts of the URL for decodability.

New in version 18.0.0.

By default, *parse()* returns an instance of `DecodedURL`, a URL type that handles all encoding for you, by wrapping the lower-level `URL`.

DecodedURL

`class` `hyperlink.`**`DecodedURL`**(*url=URL.from_text(u")*, *lazy=False*, *query_plus_is_space=None*)

DecodedURL is a type designed to act as a higher-level interface to *URL* and the recommended type for most operations. By analogy, *DecodedURL* is the `unicode` to URL's `bytes`.

DecodedURL automatically handles encoding and decoding all its components, such that all inputs and outputs are in a maximally-decoded state. Note that this means, for some special cases, a URL may not "roundtrip" character-for-character, but this is considered a good tradeoff for the safety of automatic encoding.

Otherwise, *DecodedURL* has almost exactly the same API as *URL*.

Where applicable, a UTF-8 encoding is presumed. Be advised that some interactions can raise `UnicodeEncodeErrors` and `UnicodeDecodeErrors`, just like when working with bytestrings. Examples of such interactions include handling query strings encoding binary data, and paths containing segments with special characters encoded with codecs other than UTF-8.

Parameters

- **`url`** – A *URL* object to wrap.
- **`lazy`** – Set to True to avoid pre-decode all parts of the URL to check for validity. Defaults to False.
- **`query_plus_is_space`** –

– characters in the query string should be treated

as spaces when decoding. If unspecified, the default is taken from the scheme.

Note: The *DecodedURL* initializer takes a *URL* object, not URL components, like *URL*. To programmatically construct a *DecodedURL*, you can use this pattern:

```
>>> print(DecodedURL().replace(scheme=u'https',
... host=u'pypi.org', path=(u'projects', u'hyperlink')).to_text())
https://pypi.org/projects/hyperlink
```

New in version 18.0.0.

classmethod DecodedURL.**from_text**(*text*, *lazy=False*, *query_plus_is_space=None*)
Make a *DecodedURL* instance from any text string containing a URL.

Parameters

- **text** – Text containing the URL
- **lazy** – Whether to pre-decode all parts of the URL to check for validity. Defaults to True.

The Encoded URL

The lower-level `URL` looks very similar to the `DecodedURL`, but does not handle all encoding cases for you. Use with caution.

Note: `URL` is also available as an alias, `hyperlink.EncodedURL` for more explicit usage.

---

class hyperlink.**URL**(*scheme=None*, *host=None*, *path=()*, *query=()*, *fragment=u''*, *port=None*, *rooted=None*, *userinfo=u''*, *uses_netloc=None*)
From blogs to billboards, URLs are so common, that it's easy to overlook their complexity and power. With hyperlink's *URL* type, working with URLs doesn't have to be hard.

URLs are made of many parts. Most of these parts are officially named in RFC 3986 and this diagram may prove handy in identifying them:

```
 foo://user:pass@example.com:8042/over/there?name=ferret#nose
 \_/   \_______/ \_________/ \__/\_________/ \_________/ \__/
  |        |           |       |      |           |        |
scheme  userinfo     host     port   path       query   fragment
```

While *from_text()* is used for parsing whole URLs, the *URL* constructor builds a URL from the individual components, like so:

```
>>> from hyperlink import URL
>>> url = URL(scheme=u'https', host=u'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 text (`str` in Python 3, `unicode` in Python 2). All arguments are optional, defaulting to appropriately empty values. A full list of constructor arguments is below.

Parameters

- **scheme** – The text name of the scheme.
- **host** – The host portion of the network location
- **port** – 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** – A tuple of strings representing the slash-separated parts of the path, each percent-encoded.
- **query** – The query parameters, as a dictionary or as an sequence of percent-encoded key-value pairs.
- **fragment** – The fragment part of the URL.
- **rooted** – A rooted URL is one which indicates an absolute path. This is True on any URL that includes a host, or any relative URL that starts with a slash.
- **userinfo** – The username or colon-separated username:password pair.
- **uses_netloc** – Indicates whether `://` (the "netloc separator") will appear to separate the scheme from the *path* in cases where no host is present. Setting this to `True` is a non-spec-compliant affordance for the common practice of having URIs that are *not* URLs (cannot have a 'host' part) but nevertheless use the common `://` idiom that most people associate with URLs; e.g. `message:` URIs like `message://message-id` being equivalent to `message:message-id`. This may be inferred based on the scheme depending on whether *register_scheme()* has been used to register the scheme and should not be passed directly unless you know the scheme works like this and you know it has not been registered.

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** – 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)."

Args (bool):

with_password: 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 Text

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--mgba3a4fra.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--mgba3a4fra.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 the 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** – The text name of the scheme.
- **host** – The host portion of the network location.
- **path** – A tuple of strings representing the slash-separated parts of the path.
- **query** – The query parameters, as a dictionary or as an sequence of key-value pairs.
- **fragment** – The fragment part of the URL.
- **port** – The port part of the network location.
- **rooted** – Whether or not the path begins with a slash.
- **userinfo** – The username or colon-separated username:password pair.
- **uses_netloc** – Indicates whether `://` (the "netloc separator") will appear to separate the scheme from the *path* in cases where no host is present. Setting this to `True` is a non-spec-compliant affordance for the common practice of having URIs that are *not* URLs (cannot have a 'host' part) but nevertheless use the common `://` idiom that most people associate with URLs; e.g. `message:` URIs like `message://message-id` being equivalent to `message:message-id`. This may be inferred based on the scheme depending on whether *register_scheme()* has been used to register the scheme and should not be passed directly unless you know the scheme works like this and you know it has not been registered.

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** – Convert the scheme to lowercase

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

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

Args (Text): href: A string representing a clicked URL.

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

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** – 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(u'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** – Additional parts to be joined and added to the path, like

:param os.path.join(). Special characters in segments will be: :param 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** – 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[Optional[Text]]

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** – The name of the query parameter to add. The part before the =.
- **value** – 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 occurences, 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** – The name of the query parameter to set. The part before the =.
- **value** – 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*, *value=Sentinel('_UNSET')*, *limit=None*)

Make a new *URL* instance with occurrences of the query parameter *name* removed, or, if *value* is set, parameters matching *name* and *value*. No exception is raised if the parameter is not already set.

Parameters

- **name** – The name of the query parameter to remove.
- **value** – Optional value to additionally filter on. Setting this removes query parameters which match both name and value.
- **limit** – Optional maximum number of parameters 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(u'http://wikipedia.org/').absolute
True
>>> URL.from_text(u'?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(u'http://example.com/pa/th').port
80
>>> URL.from_text(u'foo://example.com/pa/th').port
>>> URL.from_text(u'foo://example.com:8042/pa/th').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*, *query_plus_is_space=True*)
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** – A string representation of the scheme. (the 'http' in 'http://hatnote.com')

- **uses_netloc** – Does the scheme support specifying a network host? For instance, "http" does, "mailto" does not. Defaults to True.
- **default_port** – The default port, if any, for netloc-using schemes.
- **query_plus_is_space** – If true, a "+" in the query string should be decoded as a space by DecodedURL.

hyperlink.**parse**(*url*, *decoded=True*, *lazy=False*)

Automatically turn text into a structured URL object.

```
>>> url = parse(u"https://github.com/python-hyper/hyperlink")
>>> print(url.to_text())
https://github.com/python-hyper/hyperlink
```

Parameters

- **url** – A text string representation of a URL.
- **decoded** – Whether or not to return a *DecodedURL*, which automatically handles all encoding/decoding/quoting/unquoting for all the various accessors of parts of the URL, or a *URL*, which has the same API, but requires handling of special characters for different parts of the URL.
- **lazy** – In the case of *decoded=True*, this controls whether the URL is decoded immediately or as accessed. The default, *lazy=False*, checks all encoded parts of the URL for decodability.

New in version 18.0.0.

3.3 FAQ

There were bound to be questions.

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 obsoletion 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.

Python Module Index

h

Index