werkzeug

Werkzeug is the Swiss Army knife of Python web development.

It provides useful classes and functions for any WSGI application to make the life of a python web developer much easier. All of the provided classes are independent from each other so you can mix it with any other library.

Functions

`append_slash_redirect`(environ[, code])	Redirects to the same URL but with a slash appended.
`bind_arguments`(func, args, kwargs)	Bind the arguments provided into a dict.
`check_password_hash`(pwhash, password)	check a password against a given salted and hashed password value.
`cookie_date`([expires])	Formats the time to ensure compatibility with Netscape’s cookie standard.
`create_environ`(*args, **kwargs)	Create a new WSGI environ dict based on the values passed.
`dump_cookie`(key[, value, max_age, expires, ...])	Creates a new Set-Cookie header without the `Set-Cookie` prefix The parameters are the same as in the cookie Morsel object in the Python standard library but it accepts unicode data, too.
`dump_header`(iterable[, allow_token])	Dump an HTTP header again.
`dump_options_header`(header, options)	The reverse function to `parse_options_header()`.
`escape`(s[, quote])	Replace special characters “&”, “<”, “>” and (”) to HTML-safe sequences.
`extract_path_info`(environ_or_baseurl, ...[, ...])	Extracts the path info from the given URL (or WSGI environment) and path.
`find_modules`(import_path[, ...])	Finds all the modules below a package.
`format_string`(string, context)	String-template format a string:
`generate_etag`(data)	Generate an etag for some data.
`generate_password_hash`(password[, method, ...])	Hash a password with the given method and salt with with a string of the given length.
`get_current_url`(environ[, root_only, ...])	A handy helper function that recreates the full URL as IRI for the current request or parts of it.
`get_host`(environ[, trusted_hosts])	Return the real host for the given WSGI environment.
`http_date`([timestamp])	Formats the time to match the RFC1123 date format.

`import_string`(import_name[, silent])	Imports an object based on a string.
`iri_to_uri`(iri[, charset, errors, ...])	Converts any unicode based IRI to an acceptable ASCII URI.
`is_entity_header`(header)	Check if a header is an entity header.
`is_hop_by_hop_header`(header)	Check if a header is an HTTP/1.1 “Hop-by-Hop” header.
`is_resource_modified`(environ[, etag, data, ...])	Convenience method for conditional requests.
`make_line_iter`(stream[, limit, buffer_size, ...])	Safely iterates line-based over an input stream.
`parse_accept_header`(value[, cls])	Parses an HTTP Accept-* header.
`parse_authorization_header`(value)	Parse an HTTP basic/digest authorization header transmitted by the web browser.
`parse_cache_control_header`(value[, ...])	Parse a cache control header.
`parse_cookie`(header[, charset, errors, cls])	Parse a cookie.
`parse_date`(value)	Parse one of the following date formats into a datetime object:
`parse_dict_header`(value[, cls])	Parse lists of key, value pairs as described by RFC 2068 Section 2 and
`parse_etags`(value)	Parse an etag header.
`parse_form_data`(environ[, stream_factory, ...])	Parse the form data in the environ and return it as tuple in the form `(stream, form, files)`.
`parse_list_header`(value)	Parse lists as described by RFC 2068 Section 2.
`parse_options_header`(value[, multiple])	Parse a `Content-Type` like header into a tuple with the content
`parse_set_header`(value[, on_update])	Parse a set-like header and return a
`parse_www_authenticate_header`(value[, on_update])	Parse an HTTP WWW-Authenticate header into a `WWWAuthenticate` object.
`peek_path_info`(environ[, charset, errors])	Returns the next segment on the *PATH_INFO* or *None* if there is none.
`pop_path_info`(environ[, charset, errors])	Removes and returns the next segment of *PATH_INFO*, pushing it onto *SCRIPT_NAME*.
`quote_etag`(etag[, weak])	Quote an etag.
`quote_header_value`(value[, extra_chars, ...])	Quote a header value if necessary.

`redirect`(location[, code, Response])	Returns a response object (a WSGI application) that, if called, redirects the client to the target location.
`release_local`(local)	Releases the contents of the local for the current context.
`remove_entity_headers`(headers[, allowed])	Remove all entity headers from a list or `Headers` object.
`remove_hop_by_hop_headers`(headers)	Remove all HTTP/1.1 “Hop-by-Hop” headers from a list or `Headers` object.
`responder`(f)	Marks a function as responder.
`run_simple`(hostname, port, application[, ...])	Start a WSGI application.
`run_wsgi_app`(app, environ[, buffered])	Return a tuple in the form (app_iter, status, headers) of the application output.
`secure_filename`(filename)	Pass it a filename and it will return a secure version of it.
`test_app`(environ, start_response)	Simple test application that dumps the environment.
`unescape`(s)	The reverse function of *escape*.
`unquote_etag`(etag)	Unquote a single etag:
`unquote_header_value`(value[, is_filename])	Unquotes a header value.
`uri_to_iri`(uri[, charset, errors])	Converts a URI in a given charset to a IRI.
`url_decode`(s[, charset, decode_keys, ...])	Parse a querystring and return it as `MultiDict`.
`url_encode`(obj[, charset, encode_keys, ...])	URL encode a dict/*MultiDict*.
`url_fix`(s[, charset])	Sometimes you get an URL by a user that just isn’t a real URL because it contains unsafe characters like ‘ ‘ and so on.
`url_quote`(string[, charset, errors, safe, ...])	URL encode a single string with a given encoding.
`url_quote_plus`(string[, charset, errors, safe])	URL encode a single string with the given encoding and convert whitespace to “+”.
`url_unquote`(string[, charset, errors, unsafe])	URL decode a single string with a given encoding.
`url_unquote_plus`(s[, charset, errors])	URL decode a single string with the given *charset* and decode “+” to whitespace.
`validate_arguments`(func, args, kwargs[, ...])	Checks if the function accepts the arguments and keyword arguments.
`wrap_file`(environ, file[, buffer_size])	Wraps a file.

Classes

`Aborter`([mapping, extra])	When passed a dict of code -> exception items it can be used as callable that raises exceptions.
`Accept`([values])	An `Accept` object is just a list subclass for lists of `(value, quality)` tuples.
`AcceptMixin`	A mixin for classes with an `environ` attribute to get all the HTTP accept headers as `Accept` objects (or subclasses thereof).
`Authorization`(auth_type[, data])	Represents an *Authorization* header sent by the client.
`AuthorizationMixin`	Adds an `authorization` property that represents the parsed value of the *Authorization* header as `Authorization` object.
`BaseRequest`(environ[, populate_request, shallow])	Very basic request object.
`BaseResponse`([response, status, headers, ...])	Base response class.
`CallbackDict`([initial, on_update])	A dict that calls a function passed every time something is changed.
`CharsetAccept`([values])	Like `Accept` but with normalization for charsets.
`Client`(application[, response_wrapper, ...])	This class allows to send requests to a wrapped application.
`ClosingIterator`(iterable[, callbacks])	The WSGI specification requires that all middlewares and gateways respect the *close* callback of an iterator.
`CombinedMultiDict`([dicts])	A read only `MultiDict` that you can pass multiple `MultiDict`
`CommonRequestDescriptorsMixin`	A mixin for `BaseRequest` subclasses.
`CommonResponseDescriptorsMixin`	A mixin for `BaseResponse` subclasses.
`DebuggedApplication`(app[, evalex, ...])	Enables debugging support for a given application:
`DispatcherMiddleware`(app[, mounts])	Allows one to mount middlewares or applications in a WSGI application.
`ETagRequestMixin`	Add entity tag and cache descriptors to a request object or object with a WSGI environment available as `environ`.
`ETagResponseMixin`	Adds extra functionality to a response object for etag and cache handling.
`ETags`([strong_etags, weak_etags, star_tag])	A set that can be used to check if one etag is present in a collection of etags.
`EnvironBuilder`([path, base_url, ...])	This class can be used to conveniently create a WSGI environment for testing purposes.
`EnvironHeaders`(environ)	Read only version of the headers from a WSGI environment.
`FileMultiDict`([mapping])	A special `MultiDict` that has convenience methods to

	add files to it.
FileStorage([stream, filename, name, ...])	The FileStorage class is a thin wrapper over incoming files.
FileWrapper(file[, buffer_size])	This class can be used to convert a `file`-like object into an iterable.
HTMLBuilder(dialect)	Helper object for HTML generation.
HeaderSet([headers, on_update])	Similar to the ETags class this implements a set-like structure.
Headers([defaults])	An object that stores some headers.
Href([base, charset, sort, key])	Implements a callable that constructs URLs with the given base.
ImmutableDict	An immutable `dict`.
ImmutableList	An immutable `list`.
ImmutableMultiDict([mapping])	An immutable MultiDict.
ImmutableOrderedMultiDict([mapping])	An immutable OrderedMultiDict.
ImmutableTypeConversionDict	Works like a TypeConversionDict but does not support modifications.
LanguageAccept([values])	Like Accept but with normalization for languages.
LimitedStream(stream, limit)	Wraps a stream so that it doesn’t read more than n bytes.
Local()	
LocalManager([locals, ident_func])	Local objects cannot manage themselves.
LocalProxy(local[, name])	Acts as a proxy for a werkzeug local.
LocalStack()	This class works similar to a Local but keeps a stack of objects instead.
MIMEAccept([values])	Like Accept but with special methods and behavior for mimetypes.
MultiDict([mapping])	A MultiDict is a dictionary subclass customized to deal with multiple values for the same key which is for example used by the parsing functions in the wrappers.
OrderedMultiDict([mapping])	Works like a regular MultiDict but preserves the order of the fields.
Request(environ[, populate_request, shallow])	Full featured request object implementing the following mixins:
RequestCacheControl([values, on_update])	A cache control for requests.
Response([response, status, headers, ...])	Full featured response object implementing the following mixins:
ResponseCacheControl([values, on_update])	A cache control for responses.
ResponseStreamMixin	Mixin for BaseRequest subclasses.

`SharedDataMiddleware`(app, exports[, ...])	A WSGI middleware that provides static content for development environments or simple server setups.
`TypeConversionDict`	Works like a regular dict but the `get()` method can perform type conversions.
`UserAgent`(environ_or_string)	Represents a user agent.
`UserAgentMixin`	Adds a *user_agent* attribute to the request object which contains the parsed user agent of the browser that triggered the request as a `UserAgent` object.
`WWWAuthenticate`([auth_type, values, on_update])	Provides simple access to *WWW-Authenticate* headers.
`WWWAuthenticateMixin`	Adds a `www_authenticate` property to a response object.
`cached_property`(func[, name, doc])	A decorator that converts a function into a lazy property.
`environ_property`(name[, default, load_func, ...])	Maps request attributes to environment variables.
`header_property`(name[, default, load_func, ...])	Like *environ_property* but for headers.

Exceptions

`ArgumentValidationError`([missing, extra, ...])	Raised if `validate_arguments()` fails to validate

werkzeug.append_slash_redirect

`werkzeug.append_slash_redirect`(*environ*, *code=301*)[source]

Redirects to the same URL but with a slash appended. The behavior of this function is undefined if the path ends with a slash already.

Parameters:
- **environ** – the WSGI environment for the request that triggers the redirect.
- **code** – the status code for the redirect.

werkzeug.bind_arguments

`werkzeug.bind_arguments`(*func*, *args*, *kwargs*)[source]

Bind the arguments provided into a dict. When passed a function, a tuple of arguments and a dict of keyword arguments *bind_arguments* returns a dict of names as the function would see it. This can be useful to implement a cache decorator that uses the function arguments to build the cache key based on the values of the arguments.

Parameters:
- **func** – the function the arguments should be bound for.
- **args** – tuple of positional arguments.
- **kwargs** – a dict of keyword arguments.

Returns: a `dict` of bound keyword arguments.

werkzeug.check_password_hash

`werkzeug.check_password_hash`(*pwhash*, *password*)[source]

check a password against a given salted and hashed password value. In order to support unsalted legacy passwords this method supports plain text passwords, md5 and sha1 hashes (both salted and unsalted).

Returns *True* if the password matched, *False* otherwise.

Parameters:
- **pwhash** – a hashed string like returned by `generate_password_hash()`.
- **password** – the plaintext password to compare against the hash.

werkzeug.cookie_date

`werkzeug.cookie_date`(*expires=None*)[source]

Formats the time to ensure compatibility with Netscape's cookie standard.

Accepts a floating point number expressed in seconds since the epoch in, a datetime object or a timetuple. All times in UTC. The `parse_date()` function can be used to parse such a date.

Outputs a string in the format `Wdy, DD-Mon-YYYY HH:MM:SS GMT`.

Parameters: **expires** – If provided that date is used, otherwise the current.

werkzeug.create_environ

`werkzeug.create_environ`(**args*, ***kwargs*)[source]

Create a new WSGI environ dict based on the values passed. The first parameter should be the path of the request which defaults to '/'. The second one can either be an absolute path (in that case the host is localhost:80) or a full path to the request with scheme, netloc port and the path to the script.

This accepts the same arguments as the `EnvironBuilder` constructor.

Changed in version 0.5: This function is now a thin wrapper over `EnvironBuilder` which was added in 0.5. The *headers*, *environ_base*, *environ_overrides* and *charset* parameters were added.

werkzeug.dump_cookie

`werkzeug.dump_cookie`(*key*, *value=''*, *max_age=None*, *expires=None*, *path='/'*, *domain=None*, *secure=False*, *httponly=False*, *charset='utf-8'*, *sync_expires=True*)[source]

Creates a new Set-Cookie header without the `Set-Cookie` prefix The parameters are the same as in the cookie Morsel object in the Python standard library but it accepts unicode data, too.

On Python 3 the return value of this function will be a unicode string, on Python 2 it will be a native string. In both cases the return value is usually restricted to ascii as the vast majority of values are properly escaped, but that is no guarantee. If a unicode string is returned it's tunneled through latin1 as required by PEP 3333.

The return value is not ASCII safe if the key contains unicode characters. This is technically against the specification but happens in the wild. It's strongly recommended to not use non-ASCII values for the keys.

Parameters:

- **max_age** – should be a number of seconds, or *None* (default) if the cookie should last only as long as the client's browser session. Additionally *timedelta* objects are accepted, too.
- **expires** – should be a *datetime* object or unix timestamp.
- **path** – limits the cookie to a given path, per default it will span the whole domain.
- **domain** – Use this if you want to set a cross-domain cookie. For example, `domain=".example.com"` will set a cookie that is readable by the domain `www.example.com`, `foo.example.com` etc. Otherwise, a cookie will only be readable by the domain that set it.
- **secure** – The cookie will only be available via HTTPS
- **httponly** – disallow JavaScript to access the cookie. This is an extension to the cookie standard and probably not supported by all browsers.
- **charset** – the encoding for unicode values.
- **sync_expires** – automatically set expires if max_age is defined but expires not.

werkzeug.dump_header

`werkzeug.dump_header`(*iterable*, *allow_token=True*)[source]

Dump an HTTP header again. This is the reversal of `parse_list_header()`, `parse_set_header()` and `parse_dict_header()`. This also quotes strings that include an equals sign unless you pass it as dict of key, value pairs.

```
>>>

>>> dump_header({'foo': 'bar baz'})
'foo="bar baz"'
>>> dump_header(('foo', 'bar baz'))
'foo, "bar baz"'
```

Parameters:
- **iterable** – the iterable or dict of values to quote.
- **allow_token** – if set to *False* tokens as values are disallowed. See `quote_header_value()` for more details.

werkzeug.dump_options_header

`werkzeug.dump_options_header`(*header*, *options*)[source]

The reverse function to `parse_options_header()`.

Parameters:
- **header** – the header to dump
- **options** – a dict of options to append.

werkzeug.escape

`werkzeug.escape`(*s*, *quote=None*)[source]

Replace special characters "&", "<", ">" and (") to HTML-safe sequences.

There is a special handling for *None* which escapes to an empty string.

Changed in version 0.9: *quote* is now implicitly on.

Parameters:
- **s** – the string to escape.
- **quote** – ignored.

werkzeug.extract_path_info

`werkzeug.extract_path_info`(*environ_or_baseurl*, *path_or_url*, *charset='utf-8'*, *errors='replace'*, *collapse_http_schemes=True*)[source]

Extracts the path info from the given URL (or WSGI environment) and path. The path info returned is a unicode string, not a bytestring suitable for a WSGI environment. The URLs might also be IRIs.

If the path info could not be determined, *None* is returned.

Some examples:

>>>

```
>>> extract_path_info('http://example.com/app', '/app/hello')
u'/hello'
>>> extract_path_info('http://example.com/app',
...                   'https://example.com/app/hello')
u'/hello'
>>> extract_path_info('http://example.com/app',
...                   'https://example.com/app/hello',
...                   collapse_http_schemes=False) is None
True
```

Instead of providing a base URL you can also pass a WSGI environment.

New in version 0.6.

Parameters:

- **environ_or_baseurl** – a WSGI environment dict, a base URL or base IRI. This is the root of the application.
- **path_or_url** – an absolute path from the server root, a relative path (in which case it's the path info) or a full URL. Also accepts IRIs and unicode parameters.
- **charset** – the charset for byte data in URLs
- **errors** – the error handling on decode
- **collapse_http_schemes** – if set to *False* the algorithm does not assume that http and https on the same server point to the same resource.

werkzeug.find_modules

`werkzeug.find_modules`(*import_path*, *include_packages=False*, *recursive=False*)[source]

Finds all the modules below a package. This can be useful to automatically import all views / controllers so that their metaclasses / function decorators have a chance to register themselves on the application.

Packages are not returned unless *include_packages* is *True*. This can also recursively list modules but in that case it will import all the packages to get the correct load path of that module.

Parameters:

- **import_name** – the dotted name for the package to find child modules.
- **include_packages** – set to *True* if packages should be returned, too.
- **recursive** – set to *True* if recursion should happen.

Returns: generator

werkzeug.format_string

`werkzeug.format_string`(*string*, *context*)[source]

String-template format a string:

```
>>>

>>> format_string('$foo and ${foo}s', dict(foo=42))
'42 and 42s'
```

This does not do any attribute lookup etc. For more advanced string formattings have a look at the *werkzeug.template* module.

Parameters:
- **string** – the format string.
- **context** – a dict with the variables to insert.

werkzeug.generate_password_hash

`werkzeug.generate_password_hash`(*password*, *method='pbkdf2:sha1'*, *salt_length=8*) [source]

Hash a password with the given method and salt with with a string of the given length. The format of the string returned includes the method that was used so that `check_password_hash()` can check the hash.

The format for the hashed string looks like this:

```
method$salt$hash
```

This method can **not** generate unsalted passwords but it is possible to set the method to plain to enforce plaintext passwords. If a salt is used, hmac is used internally to salt the password.

If PBKDF2 is wanted it can be enabled by setting the method to `pbkdf2:method:iterations` where iterations is optional:

```
pbkdf2:sha1:2000$salt$hash
pbkdf2:sha1$salt$hash
```

Parameters:
- **password** – the password to hash.
- **method** – the hash method to use (one that hashlib supports). Can optionally be in the format `pbkdf2:<method>[:iterations]` to enable PBKDF2.
- **salt_length** – the length of the salt in letters.

werkzeug.get_current_url

`werkzeug.get_current_url`(*environ*, *root_only=False*, *strip_querystring=False*, *host_only=False*, *trusted_hosts=None*)[source]

A handy helper function that recreates the full URL as IRI for the current request or parts of it. Here an example:

>>>

```
>>> from werkzeug.test import create_environ
>>> env = create_environ("/?param=foo", "http://localhost/script")
>>> get_current_url(env)
'http://localhost/script/?param=foo'
>>> get_current_url(env, root_only=True)
'http://localhost/script/'
>>> get_current_url(env, host_only=True)
'http://localhost/'
>>> get_current_url(env, strip_querystring=True)
'http://localhost/script/'
```

This optionally it verifies that the host is in a list of trusted hosts. If the host is not in there it will raise a `SecurityError`.

Note that the string returned might contain unicode characters as the representation is an IRI not an URI. If you need an ASCII only representation you can use the `iri_to_uri()` function:

>>>

```
>>> from werkzeug.urls import iri_to_uri
>>> iri_to_uri(get_current_url(env))
'http://localhost/script/?param=foo'
```

Parameters:

- **environ** – the WSGI environment to get the current URL from.
- **root_only** – set *True* if you only want the root URL.
- **strip_querystring** – set to *True* if you don’t want the querystring.
- **host_only** – set to *True* if the host URL should be returned.
- **trusted_hosts** – a list of trusted hosts, see `host_is_trusted()` for more information.

werkzeug.get_host

`werkzeug.get_host`(*environ*, *trusted_hosts=None*)[source]

Return the real host for the given WSGI environment. This first checks the *X-Forwarded-Host* header, then the normal *Host* header, and finally the *SERVER_NAME* environment variable (using the first one it finds).

Optionally it verifies that the host is in a list of trusted hosts. If the host is not in there it will raise a `SecurityError`.

Parameters:
- **environ** – the WSGI environment to get the host of.
- **trusted_hosts** – a list of trusted hosts, see `host_is_trusted()` for more information.

werkzeug.http_date

`werkzeug.http_date`(*timestamp=None*)[source]

Formats the time to match the RFC1123 date format.

Accepts a floating point number expressed in seconds since the epoch in, a datetime object or a timetuple. All times in UTC. The `parse_date()` function can be used to parse such a date.

Outputs a string in the format `Wdy, DD Mon YYYY HH:MM:SS GMT`.

Parameters: **timestamp** – If provided that date is used, otherwise the current.

werkzeug.import_string

`werkzeug.import_string`(*import_name*, *silent=False*)[source]

Imports an object based on a string. This is useful if you want to use import paths as endpoints or something similar. An import path can be specified either in dotted notation (`xml.sax.saxutils.escape`) or with a colon as object delimiter (`xml.sax.saxutils:escape`).

If *silent* is True the return value will be *None* if the import fails.

Parameters:
- **import_name** – the dotted name for the object to import.
- **silent** – if set to *True* import errors are ignored and *None* is returned instead.

Returns: imported object

werkzeug.iri_to_uri

`werkzeug.iri_to_uri`(*iri*, *charset='utf-8'*, *errors='strict'*, *safe_conversion=False*)[source]

Converts any unicode based IRI to an acceptable ASCII URI. Werkzeug always uses utf-8 URLs internally because this is what browsers and HTTP do as well. In some places where it accepts an URL it also accepts a unicode IRI and converts it into a URI.

Examples for IRI versus URI:

>>>

```
>>> iri_to_uri(u'http://☃.net/')
'http://xn--n3h.net/'
>>> iri_to_uri(u'http://üser:pässword@☃.net/påth')
'http://%C3%BCser:p%C3%A4ssword@xn--n3h.net/p%C3%A5th'
```

There is a general problem with IRI and URI conversion with some protocols that appear in the wild that are in violation of the URI specification. In places where Werkzeug goes through a forced IRI to URI conversion it will set the *safe_conversion* flag which will not perform a conversion if the end result is already ASCII. This can mean that the return value is not an entirely correct URI but it will not destroy such invalid URLs in the process.

As an example consider the following two IRIs:

```
magnet:?xt=uri:whatever
itms-services://?action=download-manifest
```

The internal representation after parsing of those URLs is the same and there is no way to reconstruct the original one. If safe conversion is enabled however this function becomes a noop for both of those strings as they both can be considered URIs.

New in version 0.6.

Changed in version 0.9.6: The *safe_conversion* parameter was added.

Parameters:

- **iri** – The IRI to convert.
- **charset** – The charset for the URI.
- **safe_conversion** – indicates if a safe conversion should take place. For more information see the explanation above.

werkzeug.is_entity_header

`werkzeug.is_entity_header`(*header*)[source]

Check if a header is an entity header.

New in version 0.5.

Parameters: **header** – the header to test.

Returns: *True* if it’s an entity header, *False* otherwise.

werkzeug.is_hop_by_hop_header

`werkzeug.is_hop_by_hop_header`(*header*)[source]

Check if a header is an HTTP/1.1 “Hop-by-Hop” header.

New in version 0.5.

Parameters: **header** – the header to test.

Returns: *True* if it’s an entity header, *False* otherwise.

werkzeug.is_resource_modified

`werkzeug.is_resource_modified`(*environ*, *etag=None*, *data=None*, *last_modified=None*)
[source]

Convenience method for conditional requests.

Parameters:
- **environ** – the WSGI environment of the request to be checked.
- **etag** – the etag for the response for comparison.
- **data** – or alternatively the data of the response to automatically generate an etag using `generate_etag()`.
- **last_modified** – an optional date of the last modification.

Returns: *True* if the resource was modified, otherwise *False*.

werkzeug.make_line_iter

`werkzeug.make_line_iter`(*stream*, *limit=None*, *buffer_size=10240*, *cap_at_buffer=False*)
[source]

Safely iterates line-based over an input stream. If the input stream is not a `LimitedStream` the *limit* parameter is mandatory.

This uses the stream’s `read()` method internally as opposite to the `readline()` method that is unsafe and can only be used in violation of the WSGI specification. The same problem applies to the *__iter__* function of the input stream which calls `readline()` without arguments.

If you need line-by-line processing it’s strongly recommended to iterate over the input stream using this helper function.

Changed in version 0.8: This function now ensures that the limit was reached.

New in version 0.9: added support for iterators as input stream.

Parameters:

- **stream** – the stream or iterate to iterate over.
- **limit** – the limit in bytes for the stream. (Usually content length. Not necessary if the *stream* is a `LimitedStream`.
- **buffer_size** – The optional buffer size.
- **cap_at_buffer** – if this is set chunks are split if they are longer than the buffer size. Internally this is implemented that the buffer size might be exhausted by a factor of two however.

werkzeug.parse_accept_header

`werkzeug.parse_accept_header`(*value*, *cls=None*)[source]

Parses an HTTP Accept-* header. This does not implement a complete valid algorithm but one that supports at least value and quality extraction.

Returns a new `Accept` object (basically a list of `(value, quality)` tuples sorted by the quality with some additional accessor methods).

The second parameter can be a subclass of `Accept` that is created with the parsed values and returned.

Parameters:

- **value** – the accept header string to be parsed.
- **cls** – the wrapper class for the return value (can be `Accept` or a subclass thereof)

Returns: an instance of *cls*.

werkzeug.parse_authorization_header

`werkzeug.parse_authorization_header`(*value*)[source]

Parse an HTTP basic/digest authorization header transmitted by the web browser. The return value is either *None* if the header was invalid or not given, otherwise an `Authorization` object.

Parameters: **value** – the authorization header to parse.

Returns: a `Authorization` object or *None*.

werkzeug.parse_cache_control_header

`werkzeug.parse_cache_control_header`(*value*, *on_update=None*, *cls=None*)[source]

Parse a cache control header. The RFC differs between response and request cache control, this method does not. It’s your responsibility to not use the wrong control statements.

New in version 0.5: The *cls* was added. If not specified an immutable `RequestCacheControl` is returned.

Parameters:

- **value** – a cache control header to be parsed.
- **on_update** – an optional callable that is called every time a value on the `CacheControl` object is changed.
- **cls** – the class for the returned object. By default `RequestCacheControl` is used.

Returns: a *cls* object.

werkzeug.parse_cookie

`werkzeug.parse_cookie`(*header*, *charset='utf-8'*, *errors='replace'*, *cls=None*)[source]

Parse a cookie. Either from a string or WSGI environ.

Per default encoding errors are ignored. If you want a different behavior you can set *errors* to `'replace'` or `'strict'`. In strict mode a `HTTPUnicodeError` is raised.

Changed in version 0.5: This function now returns a `TypeConversionDict` instead of a regular dict. The *cls* parameter was added.

Parameters:

- **header** – the header to be used to parse the cookie. Alternatively this can be a WSGI environment.
- **charset** – the charset for the cookie values.
- **errors** – the error behavior for the charset decoding.
- **cls** – an optional dict class to use. If this is not specified or *None* the default `TypeConversionDict` is used.

werkzeug.parse_date

`werkzeug.parse_date`(*value*)[source]

Parse one of the following date formats into a datetime object:

```
Sun, 06 Nov 1994 08:49:37 GMT  ; RFC 822, updated by RFC 1123
Sunday, 06-Nov-94 08:49:37 GMT ; RFC 850, obsoleted by RFC 1036
Sun Nov  6 08:49:37 1994       ; ANSI C's asctime() format
```

If parsing fails the return value is *None*.

Parameters: **value** – a string with a supported date format.

Returns: a `datetime.datetime` object.

werkzeug.parse_dict_header

`werkzeug.parse_dict_header`(*value*, *cls=<type 'dict'>*)[source]

Parse lists of key, value pairs as described by RFC 2068 Section 2 and convert them into a python dict (or any other mapping object created from the type with a dict like interface provided by the *cls* arugment):

```
>>>
```

```
>>> d = parse_dict_header('foo="is a fish", bar="as well"')
>>> type(d) is dict
True
>>> sorted(d.items())
[('bar', 'as well'), ('foo', 'is a fish')]
```

If there is no value for a key it will be *None*:

```
>>>
```

```
>>> parse_dict_header('key_without_value')
{'key_without_value': None}
```

To create a header from the `dict` again, use the `dump_header()` function.

Changed in version 0.9: Added support for *cls* argument.

Parameters:
- **value** – a string with a dict header.
- **cls** – callable to use for storage of parsed results.

Returns: an instance of *cls*

werkzeug.parse_form_data

`werkzeug.parse_form_data`(*environ*, *stream_factory=None*, *charset='utf-8'*, *errors='replace'*, *max_form_memory_size=None*, *max_content_length=None*, *cls=None*, *silent=True*)[source]

Parse the form data in the environ and return it as tuple in the form `(stream, form, files)`. You should only call this method if the transport method is *POST*, *PUT*, or *PATCH*.

If the mimetype of the data transmitted is *multipart/form-data* the files multidict will be filled with *FileStorage* objects. If the mimetype is unknown the input stream is wrapped and returned as first argument, else the stream is empty.

This is a shortcut for the common usage of `FormDataParser`.

Have a look at dealing-with-request-data for more details.

New in version 0.5: The *max_form_memory_size*, *max_content_length* and *cls* parameters were added.

New in version 0.5.1: The optional *silent* flag was added.

Parameters:

- **environ** – the WSGI environment to be used for parsing.
- **stream_factory** – An optional callable that returns a new read and writeable file descriptor. This callable works the same as `_get_file_stream()`.
- **charset** – The character set for URL and url encoded form data.
- **errors** – The encoding error behavior.
- **max_form_memory_size** – the maximum number of bytes to be accepted for in-memory stored form data. If the data exceeds the value specified an `RequestEntityTooLarge` exception is raised.
- **max_content_length** – If this is provided and the transmitted data is longer than this value an `RequestEntityTooLarge` exception is raised.
- **cls** – an optional dict class to use. If this is not specified or *None* the default `MultiDict` is used.
- **silent** – If set to False parsing errors will not be caught.

Returns: A tuple in the form `(stream, form, files)`.

werkzeug.parse_list_header

`werkzeug.parse_list_header`(*value*)[source]

Parse lists as described by RFC 2068 Section 2.

In particular, parse comma-separated lists where the elements of the list may include quoted-strings. A quoted-string could contain a comma. A non-quoted string could have quotes in the middle. Quotes are removed automatically after parsing.

It basically works like `parse_set_header()` just that items may appear multiple times and case sensitivity is preserved.

The return value is a standard `list`:

>>>

```
>>> parse_list_header('token, "quoted value"')
['token', 'quoted value']
```

To create a header from the `list` again, use the `dump_header()` function.

Parameters: **value** – a string with a list header.

Returns: `list`

werkzeug.parse_options_header

`werkzeug.parse_options_header`(*value*, *multiple=False*)[source]

Parse a `Content-Type` like header into a tuple with the content type and the options:

```
>>>

>>> parse_options_header('text/html; charset=utf8')
('text/html', {'charset': 'utf8'})
```

This should not be used to parse `Cache-Control` like headers that use a slightly different format. For these headers use the `parse_dict_header()` function.

New in version 0.5.

Parameters:

- **value** – the header to parse.
- **multiple** – Whether try to parse and return multiple MIME types

Returns: (mimetype, options) or (mimetype, options, mimetype, options, …) if multiple=True

werkzeug.parse_set_header

`werkzeug.parse_set_header`(*value*, *on_update=None*)[source]

Parse a set-like header and return a `HeaderSet` object:

```
>>>

>>> hs = parse_set_header('token, "quoted value"')
```

The return value is an object that treats the items case-insensitively and keeps the order of the items:

```
>>>

>>> 'TOKEN' in hs
True
>>> hs.index('quoted value')
1
>>> hs
HeaderSet(['token', 'quoted value'])
```

To create a header from the `HeaderSet` again, use the `dump_header()` function.

Parameters:

- **value** – a set header to be parsed.

- **on_update** – an optional callable that is called every time a value on the `HeaderSet` object is changed.

Returns: a `HeaderSet`

werkzeug.parse_www_authenticate_header

`werkzeug.parse_www_authenticate_header`(*value*, *on_update=None*)[source]

Parse an HTTP WWW-Authenticate header into a `WWWAuthenticate` object.

Parameters:
- **value** – a WWW-Authenticate header to parse.
- **on_update** – an optional callable that is called every time a value on the `WWWAuthenticate` object is changed.

Returns: a `WWWAuthenticate` object.

---

werkzeug.pop_path_info

`werkzeug.pop_path_info`(*environ*, *charset='utf-8'*, *errors='replace'*)[source]

Removes and returns the next segment of *PATH_INFO*, pushing it onto *SCRIPT_NAME*. Returns *None* if there is nothing left on *PATH_INFO*.

If the *charset* is set to *None* a bytestring is returned.

If there are empty segments (`'/foo//bar`) these are ignored but properly pushed to the *SCRIPT_NAME*:

>>>

```
>>> env = {'SCRIPT_NAME': '/foo', 'PATH_INFO': '/a/b'}
>>> pop_path_info(env)
'a'
>>> env['SCRIPT_NAME']
'/foo/a'
>>> pop_path_info(env)
'b'
>>> env['SCRIPT_NAME']
'/foo/a/b'
```

New in version 0.5.

Changed in version 0.9: The path is now decoded and a charset and encoding parameter can be provided.

Parameters: **environ** – the WSGI environment that is modified.

werkzeug.redirect

`werkzeug.redirect`(*location*, *code=302*, *Response=None*)[source]

Returns a response object (a WSGI application) that, if called, redirects the client to the target location. Supported codes are 301, 302, 303, 305, and 307. 300 is not supported because it's not a real redirect and 304 because it's the answer for a request with a request with defined If-Modified-Since headers.

New in version 0.6: The location can now be a unicode string that is encoded using the `iri_to_uri()` function.

New in version 0.10: The class used for the Response object can now be passed in.

Parameters:

- **location** – the location the response should redirect to.
- **code** – the redirect status code. defaults to 302.
- **Response** (*class*) – a Response class to use when instantiating a response. The default is `werkzeug.wrappers.Response` if unspecified.

---

werkzeug.release_local

`werkzeug.release_local`(*local*)[source]

Releases the contents of the local for the current context. This makes it possible to use locals without a manager.

Example:

```
>>>
```

```
>>> loc = Local()
>>> loc.foo = 42
>>> release_local(loc)
>>> hasattr(loc, 'foo')
False
```

With this function one can release `Local` objects as well as `LocalStack` objects. However it is not possible to release data held by proxies that way, one always has to retain a reference to the underlying local object in order to be able to release it.

New in version 0.6.1.

werkzeug.remove_entity_headers

`werkzeug.remove_entity_headers`(*headers*, *allowed=('expires', 'content-location')*)[source]

Remove all entity headers from a list or `Headers` object. This operation works in-place. *Expires* and *Content-Location* headers are by default not removed. The reason for this is **RFC 2616** section 10.3.5 which specifies some entity headers that should be sent.

Changed in version 0.5: added *allowed* parameter.

Parameters:
- **headers** – a list or `Headers` object.
- **allowed** – a list of headers that should still be allowed even though they are entity headers.

werkzeug.remove_hop_by_hop_headers

`werkzeug.remove_hop_by_hop_headers`(*headers*)[source]

Remove all HTTP/1.1 “Hop-by-Hop” headers from a list or `Headers` object. This operation works in-place.

New in version 0.5.

Parameters: **headers** – a list or `Headers` object.

werkzeug.responder

`werkzeug.responder`(*f*)[source]

Marks a function as responder. Decorate a function with it and it will automatically call the return value as WSGI application.

Example:

```
@responder
def application(environ, start_response):
    return Response('Hello World!')
```

werkzeug.run_simple

`werkzeug.run_simple`(*hostname*, *port*, *application*, *use_reloader=False*, *use_debugger=False*, *use_evalex=True*, *extra_files=None*, *reloader_interval=1*, *reloader_type='auto'*, *threaded=False*, *processes=1*, *request_handler=None*, *static_files=None*, *passthrough_errors=False*, *ssl_context=None*)[source]

Start a WSGI application. Optional features include a reloader, multithreading and fork support.

This function has a command-line interface too:

```
python -m werkzeug.serving --help
```

New in version 0.5: *static_files* was added to simplify serving of static files as well as *passthrough_errors*.

New in version 0.6: support for SSL was added.

New in version 0.8: Added support for automatically loading a SSL context from certificate file and private key.

New in version 0.9: Added command-line interface.

New in version 0.10: Improved the reloader and added support for changing the backend through the *reloader_type* parameter. See reloader for more information.

Parameters:

- **hostname** – The host for the application. eg: `'localhost'`
- **port** – The port for the server. eg: `8080`
- **application** – the WSGI application to execute
- **use_reloader** – should the server automatically restart the python process if modules were changed?
- **use_debugger** – should the werkzeug debugging system be used?
- **use_evalex** – should the exception evaluation feature be enabled?
- **extra_files** – a list of files the reloader should watch additionally to the modules. For example configuration files.
- **reloader_interval** – the interval for the reloader in seconds.
- **reloader_type** – the type of reloader to use. The default is auto detection. Valid values are `'stat'` and `'watchdog'`. See reloader for more information.
- **threaded** – should the process handle each request in a separate thread?
- **processes** – if greater than 1 then handle each request in a new process up to this maximum number of concurrent processes.
- **request_handler** – optional parameter that can be used to replace the default one. You can use this to replace it with a different `BaseHTTPRequestHandler` subclass.
- **static_files** – a dict of paths for static files. This works exactly like `SharedDataMiddleware`, it’s actually just wrapping the application in that middleware before serving.
- **passthrough_errors** – set this to *True* to disable the error catching. This means that the

server will die on errors but it can be useful to hook debuggers in (pdb etc.)
- **ssl_context** – an SSL context for the connection. Either an `ssl.SSLContext`, a tuple in the form `(cert_file, pkey_file)`, the string `'adhoc'` if the server should automatically create one, or `None` to disable SSL (which is the default).

werkzeug.run_wsgi_app

`werkzeug.run_wsgi_app`(*app*, *environ*, *buffered=False*)[source]

Return a tuple in the form (app_iter, status, headers) of the application output. This works best if you pass it an application that returns an iterator all the time.

Sometimes applications may use the *write()* callable returned by the *start_response* function. This tries to resolve such edge cases automatically. But if you don't get the expected output you should set *buffered* to *True* which enforces buffering.

If passed an invalid WSGI application the behavior of this function is undefined. Never pass non-conforming WSGI applications to this function.

Parameters:
- **app** – the application to execute.
- **buffered** – set to *True* to enforce buffering.

Returns: tuple in the form `(app_iter, status, headers)`

werkzeug.secure_filename

`werkzeug.secure_filename`(*filename*)[source]

Pass it a filename and it will return a secure version of it. This filename can then safely be stored on a regular file system and passed to `os.path.join()`. The filename returned is an ASCII only string for maximum portability.

On windows systems the function also makes sure that the file is not named after one of the special device files.

>>>

```
>>> secure_filename("My cool movie.mov")
'My_cool_movie.mov'
>>> secure_filename("../../../etc/passwd")
'etc_passwd'
>>> secure_filename(u'i contain cool \xfcml\xe4uts.txt')
'i_contain_cool_umlauts.txt'
```

The function might return an empty filename. It's your responsibility to ensure that the filename is unique and that you generate random filename if the function returned an empty one.

New in version 0.5.

Parameters: **filename** – the filename to secure

werkzeug.test_app

`werkzeug.test_app`(*environ*, *start_response*)[source]

Simple test application that dumps the environment. You can use it to check if Werkzeug is working properly:

```
>>> from werkzeug.serving import run_simple
>>> from werkzeug.testapp import test_app
>>> run_simple('localhost', 3000, test_app)
 * Running on http://localhost:3000/
```

The application displays important information from the WSGI environment, the Python interpreter and the installed libraries.

werkzeug.unescape

`werkzeug.unescape`(*s*)[source]

The reverse function of *escape*. This unescapes all the HTML entities, not only the XML entities inserted by *escape*.

Parameters: **s** – the string to unescape.

werkzeug.uri_to_iri

`werkzeug.uri_to_iri`(*uri*, *charset='utf-8'*, *errors='replace'*)[source]

Converts a URI in a given charset to a IRI.

Examples for URI versus IRI:

>>>

```
>>> uri_to_iri(b'http://xn--n3h.net/')
u'http://\u2603.net/'
>>> uri_to_iri(b'http://%C3%BCser:p%C3%A4ssword@xn--n3h.net/p%C3%A5th')
u'http://\xfcser:p\xe4ssword@\u2603.net/p\xe5th'
```

Query strings are left unchanged:

>>>

```
>>> uri_to_iri('/?foo=24&x=%26%2f')
u'/?foo=24&x=%26%2f'
```

New in version 0.6.

Parameters:

- **uri** – The URI to convert.
- **charset** – The charset of the URI.
- **errors** – The error handling on decode.

werkzeug.url_decode

`werkzeug.url_decode`(*s*, *charset='utf-8'*, *decode_keys=False*, *include_empty=True*, *errors='replace'*, *separator='&'*, *cls=None*)[source]

Parse a querystring and return it as `MultiDict`. There is a difference in key decoding on different Python versions. On Python 3 keys will always be fully decoded whereas on Python 2, keys will remain bytestrings if they fit into ASCII. On 2.x keys can be forced to be unicode by setting *decode_keys* to *True*.

If the charset is set to *None* no unicode decoding will happen and raw bytes will be returned.

Per default a missing value for a key will default to an empty key. If you don't want that behavior you can set *include_empty* to *False*.

Per default encoding errors are ignored. If you want a different behavior you can set *errors* to `'replace'` or `'strict'`. In strict mode a *HTTPUnicodeError* is raised.

Changed in version 0.5: In previous versions ";" and "&" could be used for url decoding. This changed in 0.5 where only "&" is supported. If you want to use ";" instead a different *separator* can be provided.

The *cls* parameter was added.

Parameters:

- **s** – a string with the query string to decode.
- **charset** – the charset of the query string. If set to *None* no unicode decoding will take place.
- **decode_keys** – Used on Python 2.x to control whether keys should be forced to be unicode objects. If set to *True* then keys will be unicode in all cases. Otherwise, they remain *str* if they fit into ASCII.
- **include_empty** – Set to *False* if you don't want empty values to appear in the dict.
- **errors** – the decoding error behavior.
- **separator** – the pair separator to be used, defaults to &
- **cls** – an optional dict class to use. If this is not specified or *None* the default `MultiDict` is used.

werkzeug.url_encode

`werkzeug.url_encode`(*obj*, *charset='utf-8'*, *encode_keys=False*, *sort=False*, *key=None*, *separator='&'*)[source]

URL encode a dict/*MultiDict*. If a value is *None* it will not appear in the result string. Per default only values are encoded into the target charset strings. If *encode_keys* is set to `True` unicode keys are supported too.

If *sort* is set to *True* the items are sorted by *key* or the default sorting algorithm.

New in version 0.5: *sort*, *key*, and *separator* were added.

Parameters:

- **obj** – the object to encode into a query string.
- **charset** – the charset of the query string.
- **encode_keys** – set to *True* if you have unicode keys. (Ignored on Python 3.x)
- **sort** – set to *True* if you want parameters to be sorted by *key*.
- **separator** – the separator to be used for the pairs.
- **key** – an optional function to be used for sorting. For more details check out the `sorted()` documentation.

`werkzeug.url_fix`(*s*, *charset='utf-8'*)[source]

Sometimes you get an URL by a user that just isn’t a real URL because it contains unsafe characters like ‘ ‘ and so on. This function can fix some of the problems in a similar way browsers handle data entered by the user:

>>>

```
>>> url_fix(u'http://de.wikipedia.org/wiki/Elf (Begriffskl\xe4rung)')
'http://de.wikipedia.org/wiki/Elf%20(Begriffskl%C3%A4rung)'
```

Parameters:

- **s** – the string with the URL to fix.
- **charset** – The target charset for the URL if the url was given as unicode string.

werkzeug.url_quote

`werkzeug.url_quote`(*string*, *charset='utf-8'*, *errors='strict'*, *safe='/:'*, *unsafe=''*)[source]

URL encode a single string with a given encoding.

Parameters:

- **s** – the string to quote.
- **charset** – the charset to be used.
- **safe** – an optional sequence of safe characters.
- **unsafe** – an optional sequence of unsafe characters.

werkzeug.validate_arguments

`werkzeug.validate_arguments`(*func*, *args*, *kwargs*, *drop_extra=True*)[source]

Checks if the function accepts the arguments and keyword arguments. Returns a new `(args, kwargs)` tuple that can safely be passed to the function without causing a *TypeError* because the function signature is incompatible. If *drop_extra* is set to *True* (which is the default) any extra positional or keyword arguments are dropped automatically.

The exception raised provides three attributes:

missing
: A set of argument names that the function expected but where missing.

extra
: A dict of keyword arguments that the function can not handle but where provided.

extra_positional
: A list of values that where given by positional argument but the function cannot accept.

This can be useful for decorators that forward user submitted data to a view function:

```
from werkzeug.utils import ArgumentValidationError, validate_arguments

def sanitize(f):
    def proxy(request):
        data = request.values.to_dict()
        try:
            args, kwargs = validate_arguments(f, (request,), data)
        except ArgumentValidationError:
            raise BadRequest('The browser failed to transmit all '
                             'the data expected.')
        return f(*args, **kwargs)
    return proxy
```

Parameters:

- **func** – the function the validation is performed against.
- **args** – a tuple of positional arguments.
- **kwargs** – a dict of keyword arguments.
- **drop_extra** – set to *False* if you don’t want extra arguments to be silently dropped.

Returns: tuple in the form `(args, kwargs)`.

werkzeug.wrap_file

`werkzeug.wrap_file`(*environ*, *file*, *buffer_size=8192*)[source]

Wraps a file. This uses the WSGI server's file wrapper if available or otherwise the generic `FileWrapper`.

New in version 0.5.

If the file wrapper from the WSGI server is used it's important to not iterate over it from inside the application but to pass it through unchanged. If you want to pass out a file wrapper inside a response object you have to set `direct_passthrough` to *True*.

More information about file wrappers are available in **PEP 333**.

Parameters:

- **file** – a `file`-like object with a `read()` method.
- **buffer_size** – number of bytes for one iteration.

werkzeug.Aborter

class `werkzeug.Aborter`(*mapping=None*, *extra=None*)[source]

When passed a dict of code -> exception items it can be used as callable that raises exceptions. If the first argument to the callable is an integer it will be looked up in the mapping, if it's a WSGI application it will be raised in a proxy exception.

The rest of the arguments are forwarded to the exception constructor.

Methods

`__init__`([mapping, extra])	