werkzeug.Accept

class werkzeug.Accept(*values=()*)[source]

An Accept object is just a list subclass for lists of (value, quality) tuples. It is automatically sorted by quality.

All Accept objects work similar to a list but provide extra functionality for working with the data. Containment checks are normalized to the rules of that header:

```
>>>

>>> a = CharsetAccept([('ISO-8859-1', 1), ('utf-8', 0.7)])
>>> a.best
'ISO-8859-1'
>>> 'iso-8859-1' in a
True
>>> 'UTF8' in a
True
>>> 'utf7' in a
False
```

To get the quality for an item you can use normal item lookup:

```
>>>

>>> print a['utf-8']
0.7
>>> a['utf7']
0
```

Changed in version 0.5: Accept objects are forced immutable now.

Methods

__init__([values])	
append(item)	
best_match(matches[, default])	Returns the best match from a list of possible matches based on the quality of the client.
count(...)	
extend(iterable)	
find(key)	Get the position of an entry or return -1.
index(key)	Get the position of an entry or raise ValueError.
insert(pos, value)	

itervalues()	Iterate over all values.
pop([index])	
quality(key)	Returns the quality of the key.
remove(item)	
reverse()	
sort([cmp, key, reverse])	
to_header()	Convert the header set into an HTTP header string.
values(*a, **kw)	Like `itervalues()`, but returns a list.

Attributes

best	The best match as value.

werkzeug.AcceptMixin

class `werkzeug.AcceptMixin`[source]

A mixin for classes with an `environ` attribute to get all the HTTP accept headers as `Accept` objects (or subclasses thereof).

Attributes

accept_charsets	List of charsets this client supports as `CharsetAccept` object.
accept_encodings	List of encodings this client accepts.
accept_languages	List of languages this client accepts as `LanguageAccept` object.
accept_mimetypes	List of mimetypes this client supports as `MIMEAccept` object.

werkzeug.Authorization

class `werkzeug.Authorization`(*auth_type*, *data=None*)[source]

Represents an *Authorization* header sent by the client. You should not create this kind of object yourself but use it when it's returned by the *parse_authorization_header* function.

This object is a dict subclass and can be altered by setting dict items but it should be considered immutable as it's returned by the client and not meant for modifications.

Changed in version 0.5: This object became immutable.

Methods

`__init__`(auth_type[, data])	
`clear`()	
`copy`(() -> a shallow copy of D)	
`fromkeys`(keys[, value])	
`get`((k[,d]) -> D[k] if k in D, ...)	
`has_key`((k) -> True if D has a key k , else False)	
`items`(() -> list of D's (key, value) pa irs, ...)	
`iteritems`(() -> an iterator over the (key, ...)	
`iterkeys`(() -> an iterator over the k eys of D)	
`itervalues`(...)	
`keys`(() -> list of D's keys)	
`pop`(key[, default])	
`popitem`()	
`setdefault`(key[, default])	
`update`(*args, **kwargs)	
`values`(() -> list of D's values)	
`viewitems`(...)	
`viewkeys`(...)	
`viewvalues`(...)	

Attributes

`cnonce`	If the server sent a qop-header in the `WWW-Authenticate` header, the client has to provide this value for HTTP digest auth.
`nc`	The nonce count value transmitted by clients if a qop-header is also transmitted.
`nonce`	The nonce the server sent for digest auth, sent back by the client.
`opaque`	The opaque header from the server returned unchanged by the client.
`password`	When the authentication type is basic this is the password transmitted by the client, else *None*.

qop	Indicates what “quality of protection” the client has applied to the message for HTTP digest auth.
realm	This is the server realm sent back for HTTP digest auth.
response	A string of 32 hex digits computed as defined in RFC 2617, which proves that the user knows a password.
uri	The URI from Request-URI of the Request-Line; duplicated because proxies are allowed to change the Request-Line in transit.
username	The username transmitted.

werkzeug.BaseRequest

class `werkzeug.BaseRequest`(*environ*, *populate_request=True*, *shallow=False*)[source]

Very basic request object. This does not implement advanced stuff like entity tag parsing or cache controls. The request object is created with the WSGI environment as first argument and will add itself to the WSGI environment as `'werkzeug.request'` unless it's created with *populate_request* set to False.

There are a couple of mixins available that add additional functionality to the request object, there is also a class called *Request* which subclasses *BaseRequest* and all the important mixins.

It's a good idea to create a custom subclass of the `BaseRequest` and add missing functionality either via mixins or direct implementation. Here an example for such subclasses:

```
from werkzeug.wrappers import BaseRequest, ETagRequestMixin

class Request(BaseRequest, ETagRequestMixin):
    pass
```

Request objects are **read only**. As of 0.5 modifications are not allowed in any place. Unlike the lower level parsing functions the request object will use immutable objects everywhere possible.

Per default the request object will assume all the text data is *utf-8* encoded. Please refer to the unicode chapter for more details about customizing the behavior.

Per default the request object will be added to the WSGI environment as *werkzeug.request* to support the debugging system. If you don't want that, set *populate_request* to *False*.

If *shallow* is *True* the environment is initialized as shallow object around the environ. Every operation that would modify the environ in any way (such as consuming form data) raises an exception unless the *shallow* attribute is explicitly set to *False*. This is useful for middlewares where you don't want to consume the form data by accident. A shallow request is not populated to the WSGI environment.

Changed in version 0.5: read-only mode was enforced by using immutables classes for all data.

Methods

`__init__`(environ[, populate_request, shallow])	
`application`(f)	Decorate a function as responder that accepts the request as first argument.
`close`()	Closes associated resources of this request object.
`from_values`(*args, **kwargs)	Create a new request object based on the values provided.
`get_data`([cache, as_te	This reads the buffered incoming data from the client into one bytestring.

xt, parse_form_data])	
make_form_data_parser()	Creates the form data parser.

Attributes

access_route	If a forwarded header exists this is a list of all ip addresses from the client ip to the last proxy server.
args	The parsed URL parameters.
base_url	Like `url` but without the querystring See also: `trusted_hosts`.
charset	the charset for the request, defaults to utf-8
cookies	Read only access to the retrieved cookie values as dictionary.
data	
disable_data_descriptor	Indicates whether the data descriptor should be allowed to read and buffer up the input stream.
encoding_errors	the error handling procedure for errors, defaults to ‘replace’
files	`MultiDict` object containing
form	The form parameters.
full_path	Requested path as unicode, including the query string.
headers	The headers from the WSGI environ as immutable `EnvironHeaders`.
host	Just the host including the port if available.
host_url	Just the host with scheme as IRI.
input_stream	
is_multiprocess	boolean that is *True* if the application is served by
is_multithread	boolean that is *True* if the application is served by
is_run_once	boolean that is *True* if the application will be executed only
is_secure	*True* if the request is secure.
is_xhr	True if the request was triggered via a JavaScript XMLHttpRequest.
max_content_length	the maximum content length. This is forwarded to the form data
max_form_memory_size	the maximum form field size. This is forwarded to the form data
method	The transmission method.
path	Requested path as unicode.
query_string	The URL parameters as raw bytestring.
remote_addr	The remote address of the client.
remote_user	If the server supports user authentication, and the script is protected, this attribute contains the username the user has authenticated as.
scheme	URL scheme (http or https).

script_root	The root path of the script without the trailing slash.
stream	The stream to read incoming data from.
trusted_hosts	Optionally a list of hosts that is trusted by this request.
url	The reconstructed current URL as IRI.
url_charset	The charset that is assumed for URLs.
url_root	The full URL root (with hostname), this is the application root as IRI.
values	Combined multi dict for `args` and `form`.
want_form_data_parsed	Returns True if the request method carries content.

werkzeug.BaseResponse

class werkzeug.BaseResponse(*response=None*, *status=None*, *headers=None*, *mimetype=None*, *content_type=None*, *direct_passthrough=False*)[source]

Base response class. The most important fact about a response object is that it's a regular WSGI application. It's initialized with a couple of response parameters (headers, body, status code etc.) and will start a valid WSGI response when called with the environ and start response callable.

Because it's a WSGI application itself processing usually ends before the actual response is sent to the server. This helps debugging systems because they can catch all the exceptions before responses are started.

Here a small example WSGI application that takes advantage of the response objects:

```
from werkzeug.wrappers import BaseResponse as Response

def index():
    return Response('Index page')

def application(environ, start_response):
    path = environ.get('PATH_INFO') or '/'
    if path == '/':
        response = index()
    else:
        response = Response('Not Found', status=404)
    return response(environ, start_response)
```

Like BaseRequest which object is lacking a lot of functionality implemented in mixins. This gives you a better control about the actual API of your response objects, so you can create subclasses and add custom functionality. A full featured response object is available as Response which implements a couple of useful mixins.

To enforce a new type of already existing responses you can use the force_type() method. This is useful if you're working with different subclasses of response objects and you want to post process them with a known interface.

Per default the request object will assume all the text data is *utf-8* encoded. Please refer to the unicode chapter for more details about customizing the behavior.

Response can be any kind of iterable or string. If it's a string it's considered being an iterable with one item which is the string passed. Headers can be a list of tuples or a Headers object.

Special note for *mimetype* and *content_type*: For most mime types *mimetype* and *content_type* work the same, the difference affects only 'text' mimetypes. If the mimetype passed with *mimetype* is a mimetype starting with *text/*, the charset parameter of the response object is appended to it. In contrast the *content_type* parameter is always added as header unmodified.

Changed in version 0.5: the *direct_passthrough* parameter was added.

Parameters:

- **response** – a string or response iterable.
- **status** – a string with a status or an integer with the status code.
- **headers** – a list of headers or a `Headers` object.
- **mimetype** – the mimetype for the request. See notice above.
- **content_type** – the content type for the request. See notice above.
- **direct_passthrough** – if set to *True* `iter_encoded()` is not called before iteration which makes it possible to pass special iterators through unchanged (see `wrap_file()` for more details.)

Methods

`__init__`([response, status, headers, ...])	
`calculate_content_length`()	Returns the content length if available or *None* otherwise.
`call_on_close`(func)	Adds a function to the internal list of functions that should be called as part of closing down the response.
`close`()	Close the wrapped response if possible.
`delete_cookie`(key[, path, domain])	Delete a cookie.
`force_type`(response[, environ])	Enforce that the WSGI response is a response object of the current type.
`freeze`()	Call this method if you want to make your response object ready for being pickled.
`from_app`(app, environ[, buffered])	Create a new response object from an application output.
`get_app_iter`(environ)	Returns the application iterator for the given environ.
`get_data`([as_text])	The string representation of the request body.
`get_wsgi_headers`(environ)	This is automatically called right before the response is started and returns headers modified for the given environment.
`get_wsgi_response`(environ)	Returns the final WSGI response as tuple.
`iter_encoded`()	Iter the response encoded with the encoding of the response.
`make_sequence`()	Converts the response iterator in a list.
`set_cookie`(key[, value, max_age, expires, ...])	Sets a cookie.
`set_data`(value)	Sets a new string as response.

Attributes

`autocorrect_lo`	Should this response object correct the location header to be RFC conformant?

cation_header	This is true by default.
automatically_set_content_length	Should this response object automatically set the content-length header if possible? This is true by default.
charset	the charset of the response.
data	A descriptor that calls `get_data()` and `set_data()`.
default_mimetype	the default mimetype if none is provided.
default_status	the default status if none is provided.
implicit_sequence_conversion	if set to *False* accessing properties on the response object will
is_sequence	If the iterator is buffered, this property will be *True*.
is_streamed	If the response is streamed (the response is not an iterable with a length information) this property is *True*.
status	The HTTP Status code
status_code	The HTTP Status code as number

werkzeug.BaseResponse.__init__

`BaseResponse.__init__`(*response=None*, *status=None*, *headers=None*, *mimetype=None*, *content_type=None*, *direct_passthrough=False*)[source]

werkzeug.BaseResponse.delete_cookie

`BaseResponse.delete_cookie`(*key*, *path='/'*, *domain=None*)[source]

Delete a cookie. Fails silently if key doesn't exist.

Parameters:

- **key** – the key (name) of the cookie to be deleted.
- **path** – if the cookie that should be deleted was limited to a path, the path has to be defined here.
- **domain** – if the cookie that should be deleted was limited to a domain, that domain has to be defined here.

---

werkzeug.BaseResponse.force_type

classmethod `BaseResponse.force_type`(*response*, *environ=None*)[source]

Enforce that the WSGI response is a response object of the current type. Werkzeug will use the `BaseResponse` internally in many situations like the exceptions. If you call `get_response()` on an exception you will get back a regular `BaseResponse` object, even if you are using a custom subclass.

This method can enforce a given response type, and it will also convert arbitrary WSGI callables into response objects if an environ is provided:

```
# convert a Werkzeug response object into an instance of the
# MyResponseClass subclass.
response = MyResponseClass.force_type(response)

# convert any WSGI application into a response object
response = MyResponseClass.force_type(response, environ)
```

This is especially useful if you want to post-process responses in the main dispatcher and use functionality provided by your subclass.

Keep in mind that this will modify response objects in place if possible!

Parameters:
- **response** – a response object or wsgi application.
- **environ** – a WSGI environment object.

Returns: a response object.

werkzeug.BaseResponse.from_app

classmethod `BaseResponse.from_app`(*app*, *environ*, *buffered=False*)[source]

Create a new response object from an application output. This works best if you pass it an application that returns a generator 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.

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

Returns: a response object.

werkzeug.BaseResponse.set_cookie

`BaseResponse.set_cookie`(*key*, *value=''*, *max_age=None*, *expires=None*, *path='/'*, *domain=None*, *secure=None*, *httponly=False*)[source]

Sets a cookie. The parameters are the same as in the cookie *Morsel* object in the Python standard library but it accepts unicode data, too.

Parameters:

- **key** – the key (name) of the cookie to be set.
- **value** – the value of the cookie.
- **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.
- **expires** – should be a *datetime* object or UNIX timestamp.
- **domain** – 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.
- **path** – limits the cookie to a given path, per default it will span the whole domain.

werkzeug.CallbackDict

class werkzeug.CallbackDict(*initial=None*, *on_update=None*)[source]

A dict that calls a function passed every time something is changed. The function is passed the dict instance.

Methods

__init__([initial, on_update])	
clear(*args, **kw)	
copy(() -> a shallow copy of D)	
fromkeys(...)	v defaults to None.
get((k[,d]) -> D[k] if k in D, ...)	
has_key((k) -> True if D has a key k, else False)	
items(() -> list of D's (key, value) pairs, ...)	
iteritems(() -> an iterator over the (key, ...)	
iterkeys(() -> an iterator over the keys of D)	
itervalues(...)	
keys(() -> list of D's keys)	
pop(key[, default])	
popitem(*args, **kw)	
setdefault(key[, default])	
update(*args, **kw)	
values(() -> list of D's values)	
viewitems(...)	
viewkeys(...)	
viewvalues(...)	

Attributes

on_update	

werkzeug.CharsetAccept

class werkzeug.CharsetAccept(*values=()*)[source]

Like Accept but with normalization for charsets.

Methods

__init__([values])	
append(item)	
best_match(matches[, default])	Returns the best match from a list of possible matches based on the quality of the client.
count(...)	
extend(iterable)	
find(key)	Get the position of an entry or return -1.
index(key)	Get the position of an entry or raise ValueError.
insert(pos, value)	
itervalues()	Iterate over all values.
pop([index])	
quality(key)	Returns the quality of the key.
remove(item)	
reverse()	
sort([cmp, key, reverse])	
to_header()	Convert the header set into an HTTP header string.
values(*a, **kw)	Like itervalues(), but returns a list.

Attributes

best	The best match as value.

werkzeug.Client

class werkzeug.Client(*application*, *response_wrapper=None*, *use_cookies=True*, *allow_subdomain_redirects=False*)[source]

This class allows to send requests to a wrapped application.

The response wrapper can be a class or factory function that takes three arguments: app_iter, status and headers. The default response wrapper just returns a tuple.

Example:

```
class ClientResponse(BaseResponse):
    ...

client = Client(MyApplication(), response_wrapper=ClientResponse)
```

The use_cookies parameter indicates whether cookies should be stored and sent for subsequent requests. This is True by default, but passing False will disable this behaviour.

If you want to request some subdomain of your application you may set *allow_subdomain_redirects* to *True* as if not no external redirects are allowed.

New in version 0.5: *use_cookies* is new in this version. Older versions did not provide builtin cookie support.

Methods

__init__(application[, response_wrapper, ...])	
delete(*args, **kw)	Like open but method is enforced to DELETE.
delete_cookie(server_name, key[, path, domain])	Deletes a cookie in the test client.
get(*args, **kw)	Like open but method is enforced to GET.
head(*args, **kw)	Like open but method is enforced to HEAD.
open(*args, **kwargs)	Takes the same arguments as the EnvironBuilder class with some additions: You can provide a EnvironBuilder or a WSGI environment as only argument instead of the EnvironBuilder arguments and two optional keyword arguments (*as_tuple*, *buffered*) that change the type of the return value or the way the application is executed.
options(*args, **kw)	Like open but method is enforced to OPTIONS.
patch(*args, **kw)	Like open but method is enforced to PATCH.

post(*args, **kw)	Like open but method is enforced to POST.
put(*args, **kw)	Like open but method is enforced to PUT.
resolve_redirect(response, new_location, environ)	Resolves a single redirect and triggers the request again directly on this redirect client.
run_wsgi_app(environ[, buffered])	Runs the wrapped WSGI app with the given environment.
set_cookie(server_name, key[, value, ...])	Sets a cookie in the client's cookie jar.
trace(*args, **kw)	Like open but method is enforced to TRACE.

---

werkzeug.ClosingIterator

class werkzeug.ClosingIterator(*iterable*, *callbacks=None*)[source]

The WSGI specification requires that all middlewares and gateways respect the *close* callback of an iterator. Because it is useful to add another close action to a returned iterator and adding a custom iterator is a boring task this class can be used for that:

```
return ClosingIterator(app(environ, start_response), [cleanup_session,
                                                      cleanup_locals])
```

If there is just one close function it can be passed instead of the list.

A closing iterator is not needed if the application uses response objects and finishes the processing if the response is started:

```
try:
    return response(environ, start_response)
finally:
    cleanup_session()
    cleanup_locals()
```

Methods

__init__(iterable[, callbacks])	
close()	
next()	

werkzeug.CombinedMultiDict

class werkzeug.CombinedMultiDict(*dicts=None*)[source]

A read only `MultiDict` that you can pass multiple `MultiDict` instances as sequence and it will combine the return values of all wrapped dicts:

```
>>>
```

```
>>> from werkzeug.datastructures import CombinedMultiDict, MultiDict
>>> post = MultiDict([('foo', 'bar')])
>>> get = MultiDict([('blub', 'blah')])
>>> combined = CombinedMultiDict([get, post])
>>> combined['foo']
'bar'
>>> combined['blub']
'blah'
```

This works for all read operations and will raise a *TypeError* for methods that usually change data which isn't possible.

From Werkzeug 0.3 onwards, the *KeyError* raised by this class is also a subclass of the `BadRequest` HTTP exception and will render a page for a `400 BAD REQUEST` if caught in a catch-all for HTTP exceptions.

Methods

`__init__`([dicts])	
`add`(key, value)	
`clear`()	
`copy`()	Return a shallow copy of this object.
`deepcopy`([memo])	Return a deep copy of this object.
`fromkeys`()	
`get`(key[, default, type])	
`getlist`(key[, type])	
`has_key`(key)	
`items`(*a, **kw)	Like `iteritems()`, but returns a list.
`iteritems`([multi])	
`iterkeys`()	
`iterlists`()	
`iterlistvalues`()	
`itervalues`()	
`keys`(*a, **kw)	Like `iterkeys()`, but returns a list.
`lists`(*a, **kw)	Like `iterlists()`, but returns a list.
`listvalues`(*a, **	Like `iterlistvalues()`, but returns a list.

kw)	
pop(key[, default])	
popitem()	
popitemlist()	
poplist(key)	
setdefault(key[, default])	
setlist(key, new_list)	
setlistdefault(key[, default_list])	
to_dict([flat])	Return the contents as regular dict.
update(*args, **kwargs)	
values(*a, **kw)	Like `itervalues()`, but returns a list.
viewitems(...)	
viewkeys(...)	
viewvalues(...)	

---

werkzeug.CommonRequestDescriptorsMixin

class `werkzeug.CommonRequestDescriptorsMixin`[source]

A mixin for `BaseRequest` subclasses. Request objects that mix this class in will automatically get descriptors for a couple of HTTP headers with automatic type conversion.

New in version 0.5.

Attributes

content_encoding	The Content-Encoding entity-header field is used as a modifier to the media-type.
content_length	The Content-Length entity-header field indicates the size of the entity-body in bytes or, in the case of the HEAD method, the size of the entity-body that would have been sent had the request been a GET.
content_md5	The Content-MD5 entity-header field, as defined in RFC 1864, is an MD5 digest of the entity-body for the purpose of providing an end-to-end message integrity check (MIC) of the entity-body.

content_type	The Content-Type entity-header field indicates the media type of the entity-body sent to the recipient or, in the case of the HEAD method, the media type that would have been sent had the request been a GET.
date	The Date general-header field represents the date and time at which the message was originated, having the same semantics as orig-date in RFC 822.
max_forwards	The Max-Forwards request-header field provides a mechanism with the TRACE and OPTIONS methods to limit the number of proxies or gateways that can forward the request to the next inbound server.
mimetype	Like `content_type`, but without parameters (eg, without charset, type etc.) and always lowercase.
mimetype_params	The mimetype parameters as dict.
pragma	The Pragma general-header field is used to include implementation-specific directives that might apply to any recipient along the request/response chain.
referrer	The Referer[sic] request-header field allows the client to specify, for the server's benefit, the address (URI) of the resource from which the Request-URI was obtained (the "referrer", although the header field is misspelled).

werkzeug.CommonResponseDescriptorsMixin

class `werkzeug.CommonResponseDescriptorsMixin`[source]

A mixin for `BaseResponse` subclasses. Response objects that mix this class in will automatically get descriptors for a couple of HTTP headers with automatic type conversion.

Attributes

age	The Age response-header field conveys the sender's estimate of the amount of time since the response (or its revalidation) was generated at the origin server.
allow	The Allow entity-header field lists the set of methods supported by the resource identified by the Request-URI.
content_encoding	The Content-Encoding entity-header field is used as a modifier to the media-type.
content_language	The Content-Language entity-header field describes the natural language(s) of the intended audience for the enclosed entity.
content_length	The Content-Length entity-header field indicates the size of the entity-body, in decimal number of OCTETs, sent to the recipient or, in the case of the HEAD method, the size of the entity-body that would have been sent had the request been a GET.
content_location	The Content-Location entity-header field MAY be used to supply the resource location for the entity enclosed in the message when that entity is accessible from a location separate from the requested resource's URI.

content_md5	The Content-MD5 entity-header field, as defined in RFC 1864, is an MD5 digest of the entity-body for the purpose of providing an end-to-end message integrity check (MIC) of the entity-body.
content_type	The Content-Type entity-header field indicates the media type of the entity-body sent to the recipient or, in the case of the HEAD method, the media type that would have been sent had the request been a GET.
date	The Date general-header field represents the date and time at which the message was originated, having the same semantics as orig-date in RFC 822.
expires	The Expires entity-header field gives the date/time after which the response is considered stale.
last_modified	The Last-Modified entity-header field indicates the date and time at which the origin server believes the variant was last modified.
location	The Location response-header field is used to redirect the recipient to a location other than the Request-URI for completion of the request or identification of a new resource.
mimetype	The mimetype (content type without charset etc.)
mimetype_params	The mimetype parameters as dict.
retry_after	The Retry-After response-header field can be used with a 503 (Service Unavailable) response to indicate how long the service is expected to be unavailable to the requesting client.
vary	The Vary field value indicates the set of request-header fields that fully determines, while the response is fresh, whether a cache is permitted to use the response to reply to a subsequent request without revalidation.

werkzeug.DebuggedApplication

class `werkzeug.DebuggedApplication`(*app*, *evalex=False*, *request_key='werkzeug.request'*, *console_path='/console'*, *console_init_func=None*, *show_hidden_frames=False*, *lodgeit_url=None*, *pin_security=True*, *pin_logging=True*)[source]

Enables debugging support for a given application:

```
from werkzeug.debug import DebuggedApplication
from myapp import app
app = DebuggedApplication(app, evalex=True)
```

The *evalex* keyword argument allows evaluating expressions in a traceback's frame context.

New in version 0.9: The *lodgeit_url* parameter was deprecated.

Parameters:

- **app** – the WSGI application to run debugged.
- **evalex** – enable exception evaluation feature (interactive debugging). This requires a non-forking server.
- **request_key** – The key that points to the request object in ths environment. This parameter is ignored in current versions.

- **console_path** – the URL for a general purpose console.
- **console_init_func** – the function that is executed before starting the general purpose console. The return value is used as initial namespace.
- **show_hidden_frames** – by default hidden traceback frames are skipped. You can show them by setting this parameter to *True*.
- **pin_security** – can be used to disable the pin based security system.
- **pin_logging** – enables the logging of the pin system.

Methods

`__init__`(app[, evalex, request_key, ...])	
`check_pin_trust`(environ)	Checks if the request passed the pin test.
`debug_application`(environ, start_response)	Run the application and conserve the traceback frames.
`display_console`(request)	Display a standalone shell.
`execute_command`(request, command, frame)	Execute a command in a console.
`get_resource`(request, filename)	Return a static resource from the shared folder.
`log_pin_request`()	Log the pin if needed.
`paste_traceback`(request, traceback)	Paste the traceback and return a JSON response.
`pin_auth`(request)	Authenticates with the pin.

Attributes

`pin`	
`pin_cookie_name`	The name of the pin cookie.

werkzeug.DebuggedApplication.__init__

`DebuggedApplication.__init__`(*app*, *evalex=False*, *request_key='werkzeug.request'*, *console_path='/console'*, *console_init_func=None*, *show_hidden_frames=False*, *lodgeit_url=None*, *pin_security=True*, *pin_logging=True*)[source]

werkzeug.ETagRequestMixin

class `werkzeug.ETagRequestMixin`[source]

Add entity tag and cache descriptors to a request object or object with a WSGI environment available as `environ`. This not only provides access to etags but also to the cache control header.

Attributes

`cache_control`	A `RequestCacheControl` object for the incoming cache control headers.
`if_match`	An object containing all the etags in the *If-Match* header.
`if_modified_since`	The parsed *If-Modified-Since* header as datetime object.
`if_none_match`	An object containing all the etags in the *If-None-Match* header.
`if_range`	The parsed *If-Range* header.
`if_unmodified_since`	The parsed *If-Unmodified-Since* header as datetime object.
`range`	The parsed *Range* header.

werkzeug.ETagResponseMixin

class `werkzeug.ETagResponseMixin`[source]

Adds extra functionality to a response object for etag and cache handling. This mixin requires an object with at least a *headers* object that implements a dict like interface similar to `Headers`.

If you want the `freeze()` method to automatically add an etag, you have to mixin this method before the response base class. The default response class does not do that.

Methods

`add_etag`([overwrite, weak])	Add an etag for the current response if there is none yet.
`freeze`([no_etag])	Call this method if you want to make your response object ready for pickeling.
`get_etag`()	Return a tuple in the form `(etag, is_weak)`.
`make_conditional`(request_or_environ)	Make the response conditional to the request.
`set_etag`(etag[, weak])	Set the etag, and override the old one if there was one.

Attributes

accept_ranges	The *Accept-Ranges* header.
cache_control	The Cache-Control general-header field is used to specify directives that MUST be obeyed by all caching mechanisms along the request/response chain.
content_range	The *Content-Range* header as `ContentRange` object.

werkzeug.ETags

class `werkzeug.ETags`(*strong_etags=None*, *weak_etags=None*, *star_tag=False*)[source]

A set that can be used to check if one etag is present in a collection of etags.

Methods

__init__([strong_etags, weak_etags, star_tag])	
as_set([include_weak])	Convert the *ETags* object into a python set.
contains(etag)	Check if an etag is part of the set ignoring weak tags.
contains_raw(etag)	When passed a quoted tag it will check if this tag is part of the set.
contains_weak(etag)	Check if an etag is part of the set including weak and strong tags.
is_weak(etag)	Check if an etag is weak.
to_header()	Convert the etags set into a HTTP header string.

werkzeug.EnvironBuilder

class werkzeug.EnvironBuilder(*path='/'*, *base_url=None*, *query_string=None*, *method='GET'*, *input_stream=None*, *content_type=None*, *content_length=None*, *errors_stream=None*, *multithread=False*, *multiprocess=False*, *run_once=False*, *headers=None*, *data=None*, *environ_base=None*, *environ_overrides=None*, *charset='utf-8'*)[source]

This class can be used to conveniently create a WSGI environment for testing purposes. It can be used to quickly create WSGI environments or request objects from arbitrary data.

The signature of this class is also used in some other places as of Werkzeug 0.5 (create_environ(), BaseResponse.from_values(), Client.open()). Because of this most of the functionality is available through the constructor alone.

Files and regular form data can be manipulated independently of each other with the form and files attributes, but are passed with the same argument to the constructor: *data*.

data can be any of these values:

- a *str*: If it's a string it is converted into a input_stream, the content_length is set and you have to provide a content_type.
- a *dict*: If it's a dict the keys have to be strings and the values any of the following objects:
  - a file-like object. These are converted into FileStorage objects automatically.
  - a tuple. The add_file() method is called with the tuple items as positional arguments.

New in version 0.6: *path* and *base_url* can now be unicode strings that are encoded using the iri_to_uri() function.

Parameters:

- **path** – the path of the request. In the WSGI environment this will end up as *PATH_INFO*. If the *query_string* is not defined and there is a question mark in the *path* everything after it is used as query string.
- **base_url** – the base URL is a URL that is used to extract the WSGI URL scheme, host (server name + server port) and the script root (*SCRIPT_NAME*).
- **query_string** – an optional string or dict with URL parameters.
- **method** – the HTTP method to use, defaults to *GET*.
- **input_stream** – an optional input stream. Do not specify this and *data*. As soon as an input stream is set you can't modify args and files unless you set the input_stream to *None* again.
- **content_type** – The content type for the request. As of 0.5 you don't have to provide this when specifying files and form data via *data*.
- **content_length** – The content length for the request. You don't have to specify this when providing data via *data*.

- **errors_stream** – an optional error stream that is used for *wsgi.errors*. Defaults to `stderr`.
- **multithread** – controls *wsgi.multithread*. Defaults to *False*.
- **multiprocess** – controls *wsgi.multiprocess*. Defaults to *False*.
- **run_once** – controls *wsgi.run_once*. Defaults to *False*.
- **headers** – an optional list or `Headers` object of headers.
- **data** – a string or dict of form data. See explanation above.
- **environ_base** – an optional dict of environment defaults.
- **environ_overrides** – an optional dict of environment overrides.
- **charset** – the charset used to encode unicode data.

Methods

`__init__`([path, base_url, query_string, ...])	
`close`()	Closes all files.
`get_environ`()	Return the built environ.
`get_request`([cls])	Returns a request with the data.

Attributes

`args`	The URL arguments as `MultiDict`.
`base_url`	The base URL is a URL that is used to extract the WSGI URL scheme, host (server name + server port) and the script root (*SCRIPT_NAME*).
`content_length`	The content length as integer.
`content_type`	The content type for the request.
`files`	
`form`	
`input_stream`	An optional input stream.
`query_string`	The query string.
`server_name`	The server name (read-only, use `host` to set)
`server_port`	The server port as integer (read-only, use `host` to set)
`server_protocol`	the server protocol to use. defaults to HTTP/1.1
`wsgi_version`	the wsgi version to use. defaults to (1, 0)

werkzeug.EnvironBuilder.__init__

`EnvironBuilder.__init__`(*path='/', base_url=None, query_string=None, method='GET', input_stream=None, content_type=None, content_length=None, errors_stream=None,*

multithread=False, multiprocess=False, run_once=False, headers=None, data=None, environ_base=None, environ_overrides=None, charset='utf-8')[source]

werkzeug.EnvironHeaders

class `werkzeug.EnvironHeaders`(*environ*)[source]

Read only version of the headers from a WSGI environment. This provides the same interface as *Headers* and is constructed from a WSGI environment.

From Werkzeug 0.3 onwards, the *KeyError* raised by this class is also a subclass of the `BadRequest` HTTP exception and will render a page for a `400 BAD REQUEST` if caught in a catch-all for HTTP exceptions.

Methods

`__init__`(environ)	
`add`(item)	
`add_header`(item)	
`clear`()	Clears all headers.
`copy`()	
`extend`(iterable)	
`get`(key[, default, type, as_bytes])	Return the default value if the requested data doesn't exist.
`get_all`(name)	Return a list of all the values for the named field.
`getlist`(key[, type, as_bytes])	Return the list of items for a given key.
`has_key`(key)	Check if a key is present.
`insert`(pos, value)	
`items`(*a, **kw)	Like `iteritems()`, but returns a list.
`iteritems`([lower])	
`iterkeys`([lower])	
`itervalues`()	
`keys`(*a, **kw)	Like `iterkeys()`, but returns a list.
`pop`([index])	
`popitem`()	
`remove`(item)	
`set`(key, value)	
`setdefault`(key, default)	
`to_list`([charset])	Convert the headers into a list suitable for WSGI.
`to_wsgi_list`()	Convert the headers into a list suitable for WSGI.
`values`(*a, **kw)	Like `itervalues()`, but returns a list.

werkzeug.FileMultiDict

class werkzeug.FileMultiDict(*mapping=None*)[source]

A special MultiDict that has convenience methods to add files to it. This is used for EnvironBuilder and generally useful for unittesting.

New in version 0.5.

Methods

__init__([mapping])	
add(key, value)	Adds a new value for the key.
add_file(name, file[, filename, content_type])	Adds a new file to the dict.
clear(() -> None. Remove all items from D.)	
copy()	Return a shallow copy of this object.
deepcopy([memo])	Return a deep copy of this object.
fromkeys(...)	v defaults to None.
get(key[, default, type])	Return the default value if the requested data doesn't exist.
getlist(key[, type])	Return the list of items for a given key.
has_key((k) -> True if D has a key k, else False)	
items(*a, **kw)	Like `iteritems()`, but returns a list.
iteritems([multi])	Return an iterator of `(key, value)` pairs.
iterkeys()	
iterlists()	Return a list of `(key, values)` pairs, where values is the list of all values associated with the key.
iterlistvalues()	Return an iterator of all values associated with a key.
itervalues()	Returns an iterator of the first value on every key's value list.
keys(*a, **kw)	Like `iterkeys()`, but returns a list.
lists(*a, **kw)	Like `iterlists()`, but returns a list.
listvalues(*a, **kw)	Like `iterlistvalues()`, but returns a list.
pop(key[, default])	Pop the first item for a list on the dict.
popitem()	Pop an item from the dict.
popitemlist()	Pop a `(key, list)` tuple from the dict.
poplist(key)	Pop the list for a key from the dict.
setdefault(key[, default])	Returns the value for the key if it is in the dict, otherwise it returns *default* and sets that value for *key*.

setlist(key, new_list)	Remove the old values for a key and add new ones.
setlistdefault(key[, default_list])	Like *setdefault* but sets multiple values.
to_dict([flat])	Return the contents as regular dict.
update(other_dict)	update() extends rather than replaces existing key lists:
values(*a, **kw)	Like `itervalues()`, but returns a list.
viewitems(...)	
viewkeys(...)	
viewvalues(...)	

werkzeug.FileStorage

class `werkzeug.FileStorage`(*stream=None*, *filename=None*, *name=None*, *content_type=None*, *content_length=None*, *headers=None*)[source]

> The `FileStorage` class is a thin wrapper over incoming files. It is used by the request object to represent uploaded files. All the attributes of the wrapper stream are proxied by the file storage so it's possible to do `storage.read()` instead of the long form `storage.stream.read()`.

Methods

__init__([stream, filename, name, ...])	
close()	Close the underlying file if possible.
save(dst[, buffer_size])	Save the file to a destination path or file object.

Attributes

content_length	The content-length sent in the header.
content_type	The content-type sent in the header.
mimetype	Like `content_type`, but without parameters (eg, without charset, type etc.) and always lowercase.
mimetype_params	The mimetype parameters as dict.

werkzeug.FileWrapper

class werkzeug.FileWrapper(*file*, *buffer_size=8192*)[source]

This class can be used to convert a file-like object into an iterable. It yields *buffer_size* blocks until the file is fully read.

You should not use this class directly but rather use the wrap_file() function that uses the WSGI server's file wrapper support if it's available.

New in version 0.5.

If you're using this object together with a BaseResponse you have to use the *direct_passthrough* mode.

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

Methods

__init__(file[, buffer_size])	
close()	
next()	

werkzeug.HTMLBuilder

class werkzeug.HTMLBuilder(*dialect*)[source]

Helper object for HTML generation.

Per default there are two instances of that class. The *html* one, and the *xhtml* one for those two dialects. The class uses keyword parameters and positional parameters to generate small snippets of HTML.

Keyword parameters are converted to XML/SGML attributes, positional arguments are used as children. Because Python accepts positional arguments before keyword arguments it's a good idea to use a list with the star-syntax for some children:

>>>

```
>>> html.p(class_='foo', *[html.a('foo', href='foo.html'), ' ',
...                        html.a('bar', href='bar.html')])
u'<p class="foo"><a href="foo.html">foo</a> <a href="bar.html">bar</a></p>'
```

This class works around some browser limitations and can not be used for arbitrary SGML/XML generation. For that purpose lxml and similar libraries exist.

Calling the builder escapes the string passed:

```
>>>
```

```
>>> html.p(html("<foo>"))
u'<p>&lt;foo&gt;</p>'
```

Methods

`__init__`(dialect)	

`werkzeug.HeaderSet`

class `werkzeug.HeaderSet`(*headers=None*, *on_update=None*)[source]

Similar to the `ETags` class this implements a set-like structure. Unlike `ETags` this is case insensitive and used for vary, allow, and content-language headers.

If not constructed using the `parse_set_header()` function the instantiation works like this:

```
>>>
```

```
>>> hs = HeaderSet(['foo', 'bar', 'baz'])
>>> hs
HeaderSet(['foo', 'bar', 'baz'])
```

Methods

`__init__`([headers, on_update])	
`add`(header)	Add a new header to the set.
`as_set`([preserve_casing])	Return the set as real python set type.
`clear`()	Clear the set.
`discard`(header)	Like `remove()` but ignores errors.
`find`(header)	Return the index of the header in the set or return -1 if not found.
`index`(header)	Return the index of the header in the set or raise an `IndexError`.

remove(header)	Remove a header from the set.
to_header()	Convert the header set into an HTTP header string.
update(iterable)	Add all the headers from the iterable to the set.

---

werkzeug.Headers

class werkzeug.Headers(*defaults=None*)[source]

An object that stores some headers. It has a dict-like interface but is ordered and can store the same keys multiple times.

This data structure is useful if you want a nicer way to handle WSGI headers which are stored as tuples in a list.

From Werkzeug 0.3 onwards, the KeyError raised by this class is also a subclass of the BadRequest HTTP exception and will render a page for a 400 BAD REQUEST if caught in a catch-all for HTTP exceptions.

Headers is mostly compatible with the Python wsgiref.headers.Headers class, with the exception of __getitem__. wsgiref will return *None* for headers['missing'], whereas Headers will raise a KeyError.

To create a new Headers object pass it a list or dict of headers which are used as default values. This does not reuse the list passed to the constructor for internal usage.

Parameters: **defaults** – The list of default values for the Headers.

Changed in version 0.9: This data structure now stores unicode values similar to how the multi dicts do it. The main difference is that bytes can be set as well which will automatically be latin1 decoded.

Changed in version 0.9: The linked() function was removed without replacement as it was an API that does not support the changes to the encoding model.

Methods

__init__([defaults])	
add(_key, _value, **kw)	Add a new header tuple to the list.
add_header(_key, _value, **_kw)	Add a new header tuple to the list.
clear()	Clears all headers.
copy()	

extend(iterable)	Extend the headers with a dict or an iterable yielding keys and values.
get(key[, default, type, as_bytes])	Return the default value if the requested data doesn't exist.
get_all(name)	Return a list of all the values for the named field.
getlist(key[, type, as_bytes])	Return the list of items for a given key.
has_key(key)	Check if a key is present.
items(*a, **kw)	Like `iteritems()`, but returns a list.
iteritems([lower])	
iterkeys([lower])	
itervalues()	
keys(*a, **kw)	Like `iterkeys()`, but returns a list.
pop([key, default])	Removes and returns a key or index.
popitem()	Removes a key or index and returns a (key, value) item.
remove(key)	Remove a key.
set(_key, _value, **kw)	Remove all header tuples for *key* and add a new one.
setdefault(key, value)	Returns the value for the key if it is in the dict, otherwise it returns *default* and sets that value for *key*.
to_list([charset])	Convert the headers into a list suitable for WSGI.
to_wsgi_list()	Convert the headers into a list suitable for WSGI.
values(*a, **kw)	Like `itervalues()`, but returns a list.

---

werkzeug.Href

class `werkzeug.Href`(*base='./', charset='utf-8', sort=False, key=None*)[source]

Implements a callable that constructs URLs with the given base. The function can be called with any number of positional and keyword arguments which than are used to assemble the URL. Works with URLs and posix paths.

Positional arguments are appended as individual segments to the path of the URL:

```
>>>
```

```
>>> href = Href('/foo')
>>> href('bar', 23)
'/foo/bar/23'
>>> href('foo', bar=23)
'/foo/foo?bar=23'
```

If any of the arguments (positional or keyword) evaluates to *None* it will be skipped. If no keyword arguments are given the last argument can be a `dict` or `MultiDict` (or any other dict subclass), otherwise the keyword arguments are used for the query parameters, cutting off the first trailing underscore of the parameter name:

```
>>>

>>> href(is_=42)
'/foo?is=42'
>>> href({'foo': 'bar'})
'/foo?foo=bar'
```

Combining of both methods is not allowed:

```
>>>

>>> href({'foo': 'bar'}, bar=42)
Traceback (most recent call last):
  ...
TypeError: keyword arguments and query-dicts can't be combined
```

Accessing attributes on the href object creates a new href object with the attribute name as prefix:

```
>>>

>>> bar_href = href.bar
>>> bar_href("blub")
'/foo/bar/blub'
```

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

```
>>>

>>> href = Href("/", sort=True)
>>> href(a=1, b=2, c=3)
'/?a=1&b=2&c=3'
```

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

Methods

`__init__`([base, charset, sort, key])	

werkzeug.ImmutableDict

class `werkzeug.ImmutableDict`[source]

An immutable `dict`.

New in version 0.5.

Methods

`clear`()	
`copy`()	Return a shallow mutable copy of this object.
`fromkeys`(keys[, value])	
`get`((k[,d]) -> D[k] if k in D, ...)	
`has_key`((k) -> True if D has a key k , else False)	
`items`(() -> list of D's (key, value) pa irs, ...)	
`iteritems`(() -> an iterator over the (key, ...)	
`iterkeys`(() -> an iterator over the k eys of D)	
`itervalues`(...)	
`keys`(() -> list of D's keys)	
`pop`(key[, default])	
`popitem`()	
`setdefault`(key[, default])	
`update`(*args, **kwargs)	
`values`(() -> list of D's values)	
`viewitems`(...)	
`viewkeys`(...)	
`viewvalues`(...)	

werkzeug.ImmutableList

class `werkzeug.ImmutableList`[source]

An immutable `list`.

New in version 0.5.

Private:

Methods

append(item)	
count(...)	
extend(iterable)	
index((value, [start, ...)	Raises ValueError if the value is not present.
insert(pos, value)	
pop([index])	
remove(item)	
reverse()	
sort([cmp, key, reverse])	

werkzeug.LanguageAccept

class werkzeug.LanguageAccept(*values=()*)[source]

Like Accept but with normalization for languages.

Methods

__init__([values])	
append(item)	
best_match(matches[, default])	Returns the best match from a list of possible matches based on the quality of the client.
count(...)	
extend(iterable)	
find(key)	Get the position of an entry or return -1.
index(key)	Get the position of an entry or raise ValueError.
insert(pos, value)	
itervalues()	Iterate over all values.
pop([index])	
quality(key)	Returns the quality of the key.
remove(item)	
reverse()	
sort([cmp, key, reverse])	

to_header()	Convert the header set into an HTTP header string.
values(*a, **kw)	Like itervalues(), but returns a list.

werkzeug.LimitedStream

class werkzeug.LimitedStream(*stream*, *limit*)[source]

Wraps a stream so that it doesn't read more than n bytes. If the stream is exhausted and the caller tries to get more bytes from it on_exhausted() is called which by default returns an empty string. The return value of that function is forwarded to the reader function. So if it returns an empty string read() will return an empty string as well.

The limit however must never be higher than what the stream can output. Otherwise readlines() will try to read past the limit.

Note on WSGI compliance

calls to readline() and readlines() are not WSGI compliant because it passes a size argument to the readline methods. Unfortunately the WSGI PEP is not safely implementable without a size argument to readline() because there is no EOF marker in the stream. As a result of that the use of readline() is discouraged.

For the same reason iterating over the LimitedStream is not portable. It internally calls readline().

We strongly suggest using read() only or using the make_line_iter() which safely iterates line-based over a WSGI input stream.

Parameters:
- **stream** – the stream to wrap.
- **limit** – the limit for the stream, must not be longer than what the string can provide if the stream does not end with *EOF* (like *wsgi.input*)

Methods

__init__(stream, limit)	
exhaust([chunk_size])	Exhaust the stream.
next()	
on_disconnect()	What should happen if a disconnect is detected? The return value of this function is returned from read functions in case the client went away.
on_exhausted()	This is called when the stream tries to read past the limit.
read([size])	Read *size* bytes or if size is not provided everything is read.
readline([size])	Reads one line from the stream.
readlines([size])	Reads a file into a list of strings.

tell()	Returns the position of the stream.

Attributes

is_exhausted	If the stream is exhausted this attribute is *True*.

werkzeug.MIMEAccept

class werkzeug.MIMEAccept(*values=()*)[source]

Like Accept but with special methods and behavior for mimetypes.

Methods

__init__([values])	
append(item)	
best_match(matches[, default])	Returns the best match from a list of possible matches based on the quality of the client.
count(...)	
extend(iterable)	
find(key)	Get the position of an entry or return -1.
index(key)	Get the position of an entry or raise ValueError.
insert(pos, value)	
itervalues()	Iterate over all values.
pop([index])	
quality(key)	Returns the quality of the key.
remove(item)	
reverse()	
sort([cmp, key, reverse])	
to_header()	Convert the header set into an HTTP header string.
values(*a, **kw)	Like itervalues(), but returns a list.

Attributes

accept_html	True if this object accepts HTML.
accept_json	True if this object accepts JSON.
accept_xhtml	True if this object accepts XHTML.

best	The best match as value.

---

werkzeug.Request

class werkzeug.Request(*environ, populate_request=True, shallow=False*)[source]

Full featured request object implementing the following mixins:

- AcceptMixin for accept header parsing
- ETagRequestMixin for etag and cache control handling
- UserAgentMixin for user agent introspection
- AuthorizationMixin for http auth handling
- CommonRequestDescriptorsMixin for common headers

Methods

__init__(environ[, populate_request, shallow])	
application(f)	Decorate a function as responder that accepts the request as first argument.
close()	Closes associated resources of this request object.
from_values(*args, **kwargs)	Create a new request object based on the values provided.
get_data([cache, as_text, parse_form_data])	This reads the buffered incoming data from the client into one bytestring.
make_form_data_parser()	Creates the form data parser.

Attributes

accept_charsets	List of charsets this client supports as CharsetAccept object.
accept_encodings	List of encodings this client accepts.
accept_languages	List of languages this client accepts as LanguageAccept object.
accept_mimetypes	List of mimetypes this client supports as MIMEAccept object.
access_route	If a forwarded header exists this is a list of all ip addresses from the client ip to the last proxy server.
args	The parsed URL parameters.
authorization	The *Authorization* object in parsed form.
base_url	Like url but without the querystring See also: trusted_hosts.

cache_control	A `RequestCacheControl` object for the incoming cache control headers.
charset	
content_encoding	The Content-Encoding entity-header field is used as a modifier to the media-type.
content_length	The Content-Length entity-header field indicates the size of the entity-body in bytes or, in the case of the HEAD method, the size of the entity-body that would have been sent had the request been a GET.
content_md5	The Content-MD5 entity-header field, as defined in RFC 1864, is an MD5 digest of the entity-body for the purpose of providing an end-to-end message integrity check (MIC) of the entity-body.
content_type	The Content-Type entity-header field indicates the media type of the entity-body sent to the recipient or, in the case of the HEAD method, the media type that would have been sent had the request been a GET.
cookies	Read only access to the retrieved cookie values as dictionary.
data	
date	The Date general-header field represents the date and time at which the message was originated, having the same semantics as orig-date in RFC 822.
disable_data_descriptor	
encoding_errors	
files	`MultiDict` object containing
form	The form parameters.
full_path	Requested path as unicode, including the query string.
headers	The headers from the WSGI environ as immutable `EnvironHeaders`.
host	Just the host including the port if available.
host_url	Just the host with scheme as IRI.
if_match	An object containing all the etags in the *If-Match* header.
if_modified_since	The parsed *If-Modified-Since* header as datetime object.
if_none_match	An object containing all the etags in the *If-None-Match* header.
if_range	The parsed *If-Range* header.
if_unmodified_since	The parsed *If-Unmodified-Since* header as datetime object.
input_stream	
is_multiprocess	boolean that is *True* if the application is served by
is_multithread	boolean that is *True* if the application is served by
is_run_once	boolean that is *True* if the application will be executed only
is_secure	*True* if the request is secure.
is_xhr	True if the request was triggered via a JavaScript XMLHttpRequest.
max_content_length	
max_form_memory_size	

max_forwards	The Max-Forwards request-header field provides a mechanism with the TRACE and OPTIONS methods to limit the number of proxies or gateways that can forward the request to the next inbound server.
method	The transmission method.
mimetype	Like `content_type`, but without parameters (eg, without charset, type etc.) and always lowercase.
mimetype_params	The mimetype parameters as dict.
path	Requested path as unicode.
pragma	The Pragma general-header field is used to include implementation-specific directives that might apply to any recipient along the request/response chain.
query_string	The URL parameters as raw bytestring.
range	The parsed *Range* header.
referrer	The Referer[sic] request-header field allows the client to specify, for the server's benefit, the address (URI) of the resource from which the Request-URI was obtained (the "referrer", although the header field is misspelled).
remote_addr	The remote address of the client.
remote_user	If the server supports user authentication, and the script is protected, this attribute contains the username the user has authenticated as.
scheme	URL scheme (http or https).
script_root	The root path of the script without the trailing slash.
stream	The stream to read incoming data from.
trusted_hosts	
url	The reconstructed current URL as IRI.
url_charset	The charset that is assumed for URLs.
url_root	The full URL root (with hostname), this is the application root as IRI.
user_agent	The current user agent.
values	Combined multi dict for `args` and `form`.
want_form_data_parsed	Returns True if the request method carries content.

werkzeug.RequestCacheControl

class `werkzeug.RequestCacheControl`(*values=()*, *on_update=None*)[source]

A cache control for requests. This is immutable and gives access to all the request-relevant cache control headers.

To get a header of the `RequestCacheControl` object again you can convert the object into a string or call the `to_header()` method. If you plan to subclass it and add your own items have a look at the sourcecode for that class.

New in version 0.5: In previous versions a *CacheControl* class existed that was used both for request and response.

Methods

`__init__`([values, on_update])	
`cache_property`(key, empty, type)	Return a new property object for a cache header.
`clear`()	
`copy`(() -> a shallow copy of D)	
`fromkeys`(keys[, value])	
`get`((k[,d]) -> D[k] if k in D, ...)	
`has_key`((k) -> True if D has a key k, else Fa lse)	
`items`(() -> list of D's (key, value) pairs, ...)	
`iteritems`(() -> an iterator over the (key, ...)	
`iterkeys`(() -> an iterator over the keys of D)	
`itervalues`(...)	
`keys`(() -> list of D's keys)	
`pop`(key[, default])	
`popitem`()	
`setdefault`(key[, default])	
`to_header`()	Convert the stored values into a cache control header.
`update`(*args, **kwargs)	
`values`(() -> list of D's values)	
`viewitems`(...)	
`viewkeys`(...)	
`viewvalues`(...)	

Attributes

`max_age`	accessor for 'max-age'
`max_stale`	accessor for 'max-stale'
`min_fresh`	accessor for 'min-fresh'
`no_cache`	accessor for 'no-cache'
`no_store`	accessor for 'no-store'
`no_transform`	accessor for 'no-transform'

on_update	
only_if_cached	accessor for ‘only-if-cached’

werkzeug.Response

class werkzeug.Response(*response=None*, *status=None*, *headers=None*, *mimetype=None*, *content_type=None*, *direct_passthrough=False*)[source]

Full featured response object implementing the following mixins:

- ETagResponseMixin for etag and cache control handling
- ResponseStreamMixin to add support for the *stream* property
- CommonResponseDescriptorsMixin for various HTTP descriptors
- WWWAuthenticateMixin for HTTP authentication support

Methods

__init__([response, status, headers, ...])	
add_etag([overwrite, weak])	Add an etag for the current response if there is none yet.
calculate_content_length()	Returns the content length if available or *None* otherwise.
call_on_close(func)	Adds a function to the internal list of functions that should be called as part of closing down the response.
close()	Close the wrapped response if possible.
delete_cookie(key[, path, domain])	Delete a cookie.
force_type(response[, environ])	Enforce that the WSGI response is a response object of the current type.
freeze()	Call this method if you want to make your response object ready for being pickled.
from_app(app, environ[, buffered])	Create a new response object from an application output.
get_app_iter(environ)	Returns the application iterator for the given environ.
get_data([as_text])	The string representation of the request body.
get_etag()	Return a tuple in the form `(etag, is_weak)`.
get_wsgi_headers(environ)	This is automatically called right before the response is started and returns headers modified for the given environment.
get_wsgi_response(environ)	Returns the final WSGI response as tuple.
iter_encoded()	Iter the response encoded with the encoding of the response.
make_conditional(request_or_environ)	Make the response conditional to the request.

make_sequence()	Converts the response iterator in a list.
set_cookie(key[, value, max_age, expires, ...])	Sets a cookie.
set_data(value)	Sets a new string as response.
set_etag(etag[, weak])	Set the etag, and override the old one if there was one.

Attributes

accept_ranges	The *Accept-Ranges* header.
age	The Age response-header field conveys the sender's estimate of the amount of time since the response (or its revalidation) was generated at the origin server.
allow	The Allow entity-header field lists the set of methods supported by the resource identified by the Request-URI.
autocorrect_location_header	
automatically_set_content_length	
cache_control	The Cache-Control general-header field is used to specify directives that MUST be obeyed by all caching mechanisms along the request/response chain.
charset	
content_encoding	The Content-Encoding entity-header field is used as a modifier to the media-type.
content_language	The Content-Language entity-header field describes the natural language(s) of the intended audience for the enclosed entity.
content_length	The Content-Length entity-header field indicates the size of the entity-body, in decimal number of OCTETs, sent to the recipient or, in the case of the HEAD method, the size of the entity-body that would have been sent had the request been a GET.
content_location	The Content-Location entity-header field MAY be used to supply the resource location for the entity enclosed in the message when that entity is accessible from a location separate from the requested resource's URI.
content_md5	The Content-MD5 entity-header field, as defined in RFC 1864, is an MD5 digest of the entity-body for the purpose of providing an end-to-end message integrity check (MIC) of the entity-body.
content_range	The *Content-Range* header as `ContentRange` object.
content_type	The Content-Type entity-header field indicates the media type of the entity-body sent to the recipient or, in the case of the HEAD method, the media type that would have been sent had the request been a GET.
data	A descriptor that calls `get_data()` and `set_data()`.
date	The Date general-header field represents the date and time at which the message was originated, having the same semantics as orig-date in RFC 822.

default_mimetype	
default_status	
expires	The Expires entity-header field gives the date/time after which the response is considered stale.
implicit_sequence_conversion	
is_sequence	If the iterator is buffered, this property will be *True*.
is_streamed	If the response is streamed (the response is not an iterable with a length information) this property is *True*.
last_modified	The Last-Modified entity-header field indicates the date and time at which the origin server believes the variant was last modified.
location	The Location response-header field is used to redirect the recipient to a location other than the Request-URI for completion of the request or identification of a new resource.
mimetype	The mimetype (content type without charset etc.)
mimetype_params	The mimetype parameters as dict.
retry_after	The Retry-After response-header field can be used with a 503 (Service Unavailable) response to indicate how long the service is expected to be unavailable to the requesting client.
status	The HTTP Status code
status_code	The HTTP Status code as number
stream	The response iterable as write-only stream.
vary	The Vary field value indicates the set of request-header fields that fully determines, while the response is fresh, whether a cache is permitted to use the response to reply to a subsequent request without revalidation.
www_authenticate	The *WWW-Authenticate* header in a parsed form.

werkzeug.ResponseCacheControl

class werkzeug.ResponseCacheControl(*values=()*, *on_update=None*)[source]

A cache control for responses. Unlike RequestCacheControl this is mutable and gives access to response-relevant cache control headers.

To get a header of the ResponseCacheControl object again you can convert the object into a string or call the to_header() method. If you plan to subclass it and add your own items have a look at the sourcecode for that class.

New in version 0.5: In previous versions a *CacheControl* class existed that was used both for request and response.

Methods

`__init__`([values, on_update])	
`cache_property`(key, empty, type)	Return a new property object for a cache header.
`clear`(*args, **kw)	
`copy`(() -> a shallow copy of D)	
`fromkeys`(...)	v defaults to None.
`get`((k[,d]) -> D[k] if k in D, ...)	
`has_key`((k) -> True if D has a key k, else False)	
`items`(() -> list of D's (key, value) pairs, ...)	
`iteritems`(() -> an iterator over the (key, ...)	
`iterkeys`(() -> an iterator over the keys of D)	
`itervalues`(...)	
`keys`(() -> list of D's keys)	
`pop`(key[, default])	
`popitem`(*args, **kw)	
`setdefault`(key[, default])	
`to_header`()	Convert the stored values into a cache control header.
`update`(*args, **kw)	
`values`(() -> list of D's values)	
`viewitems`(...)	
`viewkeys`(...)	
`viewvalues`(...)	

Attributes

`max_age`	accessor for 'max-age'
`must_revalidate`	accessor for 'must-revalidate'
`no_cache`	accessor for 'no-cache'
`no_store`	accessor for 'no-store'
`no_transform`	accessor for 'no-transform'
`on_update`	
`private`	accessor for 'private'
`proxy_revalidate`	accessor for 'proxy-revalidate'
`public`	accessor for 'public'

s_maxage	accessor for ‘s-maxage’

werkzeug.SharedDataMiddleware

class werkzeug.SharedDataMiddleware(*app*, *exports*, *disallow=None*, *cache=True*, *cache_timeout=43200*, *fallback_mimetype='text/plain'*)[source]

A WSGI middleware that provides static content for development environments or simple server setups. Usage is quite simple:

```
import os
from werkzeug.wsgi import SharedDataMiddleware

app = SharedDataMiddleware(app, {
    '/shared': os.path.join(os.path.dirname(__file__), 'shared')
})
```

The contents of the folder `./shared` will now be available on `http://example.com/shared/`. This is pretty useful during development because a standalone media server is not required. One can also mount files on the root folder and still continue to use the application because the shared data middleware forwards all unhandled requests to the application, even if the requests are below one of the shared folders.

If *pkg_resources* is available you can also tell the middleware to serve files from package data:

```
app = SharedDataMiddleware(app, {
    '/shared': ('myapplication', 'shared_files')
})
```

This will then serve the `shared_files` folder in the *myapplication* Python package.

The optional *disallow* parameter can be a list of `fnmatch()` rules for files that are not accessible from the web. If *cache* is set to *False* no caching headers are sent.

Currently the middleware does not support non ASCII filenames. If the encoding on the file system happens to be the encoding of the URI it may work but this could also be by accident. We strongly suggest using ASCII only file names for static files.

The middleware will guess the mimetype using the Python *mimetype* module. If it’s unable to figure out the charset it will fall back to *fallback_mimetype*.

Changed in version 0.5: The cache timeout is configurable now.

New in version 0.6: The *fallback_mimetype* parameter was added.

Parameters:

- **app** – the application to wrap. If you don’t want to wrap an application you can pass it `NotFound`.

- **exports** – a dict of exported files and folders.
- **disallow** – a list of `fnmatch()` rules.
- **fallback_mimetype** – the fallback mimetype for unknown files.
- **cache** – enable or disable caching headers.
- **cache_timeout** – the cache timeout in seconds for the headers.

Methods

`__init__`(app, exports[, disallow, cache, ...])	
`generate_etag`(mtime, file_size, real_filename)	
`get_directory_loader`(directory)	
`get_file_loader`(filename)	
`get_package_loader`(package, package_path)	
`is_allowed`(filename)	

`werkzeug.UserAgent`

class `werkzeug.UserAgent`(*environ_or_string*)[source]

Represents a user agent. Pass it a WSGI environment or a user agent string and you can inspect some of the details from the user agent string via the attributes. The following attributes exist:

`string`

the raw user agent string

`platform`

the browser platform. The following platforms are currently recognized:

- *aix*
- *amiga*
- *android*
- *bsd*
- *chromeos*
- *hpux*
- *iphone*
- *ipad*
- *irix*
- *linux*
- *macos*
- *sco*

- *solaris*
- *wii*
- *windows*

`browser`

the name of the browser. The following browsers are currently recognized:

- *aol **
- *ask **
- *camino*
- *chrome*
- *firefox*
- *galeon*
- *google **
- *kmeleon*
- *konqueror*
- *links*
- *lynx*
- *msie*
- *msn*
- *netscape*
- *opera*
- *safari*
- *seamonkey*
- *webkit*
- *yahoo **

(Browsers maked with a star (`*`) are crawlers.)

`version`

the version of the browser

`language`

the language of the browser

Methods

`__init__`(environ_or_string)	
`to_header`()	

`werkzeug.WWWAuthenticate`

class `werkzeug.WWWAuthenticate`(*auth_type=None*, *values=None*, *on_update=None*)[source]

Provides simple access to *WWW-Authenticate* headers.

Methods

`__init__`([auth_type, values, on_update])	
`auth_property`(name[, doc])	A static helper function for subclasses to add extra authentication
`clear`(*args, **kw)	
`copy`(() -> a shallow copy of D)	
`fromkeys`(...)	v defaults to None.
`get`((k[,d]) -> D[k] if k in D, ...)	
`has_key`((k) -> True if D has a key k, else False)	
`items`(() -> list of D's (key, value) pairs, ...)	
`iteritems`(() -> an iterator over the (key, ...)	
`iterkeys`(() -> an iterator over the keys of D)	
`itervalues`(...)	
`keys`(() -> list of D's keys)	
`pop`(key[, default])	
`popitem`(*args, **kw)	
`set_basic`([realm])	Clear the auth info and enable basic auth.
`set_digest`(realm, nonce[, qop, opaque, ...])	Clear the auth info and enable digest auth.
`setdefault`(key[, default])	
`to_header`()	Convert the stored values into a WWW-Authenticate header.
`update`(*args, **kw)	
`values`(() -> list of D's values)	
`viewitems`(...)	
`viewkeys`(...)	
`viewvalues`(...)	

Attributes

`algorithm`	A string indicating a pair oftyp algorithms used to produce the digest and a checksum.
`domain`	A list of URIs that define the protection space.

nonce	A server-specified data string which should be uniquely generated each time a 401 response is made.
on_update	
opaque	A string of data, specified by the server, which should be returned by the client unchanged in the Authorization header of subsequent requests with URIs in the same protection space.
qop	A set of quality-of-privacy directives such as auth and auth-int.
realm	A string to be displayed to users so they know which username and password to use.
stale	A flag, indicating that the previous request from the client was rejected because the nonce value was stale.
type	The type of the auth mechanism.

---

werkzeug.environ_property

class werkzeug.environ_property(*name*, *default=None*, *load_func=None*, *dump_func=None*, *read_only=None*, *doc=None*)[source]

Maps request attributes to environment variables. This works not only for the Werzeug request object, but also any other class with an environ attribute:

```
>>>
```

```
>>> class Test(object):
...     environ = {'key': 'value'}
...     test = environ_property('key')
>>> var = Test()
>>> var.test
'value'
```

If you pass it a second value it's used as default if the key does not exist, the third one can be a converter that takes a value and converts it. If it raises `ValueError` or `TypeError` the default value is used. If no default value is provided *None* is used.

Per default the property is read only. You have to explicitly enable it by passing `read_only=False` to the constructor.

Methods

__init__(name[, default, load_func, ...])	
lookup(obj)	

Attributes

read_only	

werkzeug.header_property

class werkzeug.header_property(*name*, *default=None*, *load_func=None*, *dump_func=None*, *read_only=None*, *doc=None*)[source]

Like *environ_property* but for headers.

Methods

__init__(name[, default, load_func, ...])	
lookup(obj)	

Attributes

read_only	