`flask.Blueprint`

class `flask.Blueprint`(*name*, *import_name*, *static_folder=None*, *static_url_path=None*, *template_folder=None*, *url_prefix=None*, *subdomain=None*, *url_defaults=None*, *root_path=None*)
[source]

Represents a blueprint. A blueprint is an object that records functions that will be called with the `BlueprintSetupState` later to register functions or other things on the main application. See Modular Applications with Blueprints for more information.

New in version 0.7.

Methods

`__init__`(name, import_name[, static_folder, ...])	
`add_app_template_filter`(f[, name])	Register a custom template filter, available application wide.
`add_app_template_global`(f[, name])	Register a custom template global, available application wide.
`add_app_template_test`(f[, name])	Register a custom template test, available application wide.
`add_url_rule`(rule[, endpoint, view_func])	Like `Flask.add_url_rule()` but for a blueprint.
`after_app_request`(f)	Like `Flask.after_request()` but for a blueprint.
`after_request`(f)	Like `Flask.after_request()` but for a blueprint.
`app_context_processor`(f)	Like `Flask.context_processor()` but for a blueprint.
`app_errorhandler`(code)	Like `Flask.errorhandler()` but for a blueprint.
`app_template_filter`([name])	Register a custom template filter, available application wide.
`app_template_global`([name])	Register a custom template global, available application wide.
`app_template_test`([name])	Register a custom template test, available application wide.

app_url_defaults(f)	Same as url_defaults() but application wide.
app_url_value_preprocessor(f)	Same as url_value_preprocessor() but application wide.
before_app_first_request(f)	Like Flask.before_first_request().
before_app_request(f)	Like Flask.before_request().
before_request(f)	Like Flask.before_request() but for a blueprint.
context_processor(f)	Like Flask.context_processor() but for a blueprint.
endpoint(endpoint)	Like Flask.endpoint() but for a blueprint.
errorhandler(code_or_exception)	Registers an error handler that becomes active for this blueprint only.
get_send_file_max_age(filename)	Provides default cache_timeout for the send_file() functions.
make_setup_state(app, options[, ...])	Creates an instance of BlueprintSetupState() object that is later passed to the register callback functions.
open_resource(resource[, mode])	Opens a resource from the application's resource folder.
record(func)	Registers a function that is called when the blueprint is registered on the application.
record_once(func)	Works like record() but wraps the function in another function that will ensure the function is only called once.
register(app, options[, first_registration])	Called by Flask.register_blueprint() to register a blueprint on the application.
register_error_handler(code_or_exception, f)	Non-decorator version of the errorhandler() error attach function, akin to the register_error_handler() application-wide function of the Flask object but for error handlers limited to this blueprint.
route(rule, **options)	Like Flask.route() but for a blueprint.
send_static_file(filename)	Function used internally to send static files from the static folder to the browser.
teardown_app_request(f)	Like Flask.teardown_request() but for a blueprint.
teardown_request(f)	Like Flask.teardown_request() but for a blueprint.

`url_defaults`(f)	Callback function for URL defaults for this blueprint.
`url_value_preprocessor`(f)	Registers a function as URL value preprocessor for this blueprint.

Attributes

`has_static_folder`	This is `True` if the package bound object's container has a folder for static files.
`jinja_loader`	The Jinja loader for this package bound object.
`static_folder`	The absolute path to the configured static folder.
`static_url_path`	
`warn_on_modifications`	

flask.Blueprint.__init__

`Blueprint.__init__`(*name*, *import_name*, *static_folder=None*, *static_url_path=None*, *template_folder=None*, *url_prefix=None*, *subdomain=None*, *url_defaults=None*, *root_path=None*) [source]

flask.Blueprint.add_app_template_filter

`Blueprint.add_app_template_filter`(*f*, *name=None*)[source]

Register a custom template filter, available application wide. Like `Flask.add_template_filter()` but for a blueprint. Works exactly like the `app_template_filter()` decorator.

Parameters: **name** – the optional name of the filter, otherwise the function name will be used.

flask.Blueprint.add_app_template_global

`Blueprint.add_app_template_global`(*f*, *name=None*)[source]

Register a custom template global, available application wide. Like `Flask.add_template_global()` but for a blueprint. Works exactly like the `app_template_global()` decorator.

New in version 0.10.

Parameters: **name** – the optional name of the global, otherwise the function name will be used.

flask.Blueprint.add_app_template_test

`Blueprint.add_app_template_test`(*f*, *name=None*)[source]

Register a custom template test, available application wide. Like `Flask.add_template_test()` but for a blueprint. Works exactly like the `app_template_test()` decorator.

New in version 0.10.

Parameters: **name** – the optional name of the test, otherwise the function name will be used.

flask.Blueprint.add_url_rule

`Blueprint.add_url_rule`(*rule*, *endpoint=None*, *view_func=None*, ***options*)[source]

Like `Flask.add_url_rule()` but for a blueprint. The endpoint for the `url_for()` function is prefixed with the name of the blueprint.

flask.Blueprint.after_app_request

`Blueprint.after_app_request`(*f*)[source]

Like `Flask.after_request()` but for a blueprint. Such a function is executed after each request, even if outside of the blueprint.

flask.Blueprint.after_request

`Blueprint.after_request`(*f*)[source]

Like `Flask.after_request()` but for a blueprint. This function is only executed after each request that is handled by a function of that blueprint.

flask.Blueprint.app_context_processor

`Blueprint.app_context_processor`(*f*)[source]

Like `Flask.context_processor()` but for a blueprint. Such a function is executed each request, even if outside of the blueprint.

flask.Blueprint.app_errorhandler

`Blueprint.app_errorhandler`(*code*)[source]

Like `Flask.errorhandler()` but for a blueprint. This handler is used for all requests, even if outside of the blueprint.

---

flask.Blueprint.app_template_filter

`Blueprint.app_template_filter`(*name=None*)[source]

Register a custom template filter, available application wide. Like `Flask.template_filter()` but for a blueprint.

Parameters: **name** – the optional name of the filter, otherwise the function name will be used.

flask.Blueprint.app_template_global

`Blueprint.app_template_global`(*name=None*)[source]

Register a custom template global, available application wide. Like `Flask.template_global()` but for a blueprint.

New in version 0.10.

Parameters: **name** – the optional name of the global, otherwise the function name will be used.

flask.Blueprint.app_template_test

`Blueprint.app_template_test`(*name=None*)[source]

Register a custom template test, available application wide. Like `Flask.template_test()` but for a blueprint.

New in version 0.10.

Parameters: **name** – the optional name of the test, otherwise the function name will be used.

flask.Blueprint.app_url_defaults

`Blueprint.app_url_defaults`(*f*)[source]

Same as `url_defaults()` but application wide.

flask.Blueprint.app_url_value_preprocessor

`Blueprint.app_url_value_preprocessor`(*f*)[source]

Same as `url_value_preprocessor()` but application wide.

flask.Blueprint.before_app_first_request

`Blueprint.before_app_first_request`(*f*)[source]

Like `Flask.before_first_request()`. Such a function is executed before the first request to the application.

flask.Blueprint.before_app_request

`Blueprint.before_app_request`(*f*)[source]

Like `Flask.before_request()`. Such a function is executed before each request, even if outside of a blueprint.

flask.Blueprint.before_request

`Blueprint.before_request`(*f*)[source]

Like `Flask.before_request()` but for a blueprint. This function is only executed before each request that is handled by a function of that blueprint.

flask.Blueprint.context_processor

`Blueprint.context_processor`(*f*)[source]

Like `Flask.context_processor()` but for a blueprint. This function is only executed for requests handled by a blueprint.

flask.Blueprint.endpoint

`Blueprint.endpoint`(*endpoint*)[source]

Like `Flask.endpoint()` but for a blueprint. This does not prefix the endpoint with the blueprint name, this has to be done explicitly by the user of this method. If the endpoint is prefixed with a . it will be registered to the current blueprint, otherwise it's an application independent endpoint.

flask.Blueprint.errorhandler

`Blueprint.errorhandler`(*code_or_exception*)[source]

Registers an error handler that becomes active for this blueprint only. Please be aware that routing does not happen local to a blueprint so an error handler for 404 usually is not handled by a blueprint unless it is caused inside a view function. Another special case is the 500 internal server error which is always looked up from the application.

Otherwise works as the `errorhandler()` decorator of the `Flask` object.

flask.Blueprint.get_send_file_max_age

`Blueprint.get_send_file_max_age`(*filename*)

Provides default cache_timeout for the `send_file()` functions.

By default, this function returns `SEND_FILE_MAX_AGE_DEFAULT` from the configuration of `current_app`.

Static file functions such as `send_from_directory()` use this function, and `send_file()` calls this function on `current_app` when the given cache_timeout is `None`. If a cache_timeout is given in `send_file()`, that timeout is used; otherwise, this method is called.

This allows subclasses to change the behavior when sending files based on the filename. For example, to set the cache timeout for .js files to 60 seconds:

```
class MyFlask(flask.Flask):
    def get_send_file_max_age(self, name):
        if name.lower().endswith('.js'):
            return 60
        return flask.Flask.get_send_file_max_age(self, name)
```

New in version 0.9.

flask.Blueprint.make_setup_state

`Blueprint.make_setup_state`(*app*, *options*, *first_registration=False*)[source]

Creates an instance of `BlueprintSetupState()` object that is later passed to the register callback functions. Subclasses can override this to return a subclass of the setup state.

flask.Blueprint.open_resource

`Blueprint.open_resource`(*resource*, *mode='rb'*)

Opens a resource from the application’s resource folder. To see how this works, consider the following folder structure:

```
/myapplication.py
/schema.sql
/static
    /style.css
/templates
    /layout.html
    /index.html
```

If you want to open the `schema.sql` file you would do the following:

```
with app.open_resource('schema.sql') as f:
    contents = f.read()
    do_something_with(contents)
```

Parameters:

- **resource** – the name of the resource. To access resources within subfolders use forward slashes as separator.
- **mode** – resource file opening mode, default is ‘r

flask.Blueprint.record

`Blueprint.record`(*func*)[source]

Registers a function that is called when the blueprint is registered on the application. This function is called with the state as argument as returned by the `make_setup_state()` method.

flask.Blueprint.record_once

`Blueprint.record_once`(*func*)[source]

Works like `record()` but wraps the function in another function that will ensure the function is only called once. If the blueprint is registered a second time on the application, the function passed is not called.

flask.Blueprint.register

`Blueprint.register`(*app*, *options*, *first_registration=False*)[source]

Called by `Flask.register_blueprint()` to register a blueprint on the application. This can be overridden to customize the register behavior. Keyword arguments from `register_blueprint()` are directly forwarded to this method in the *options* dictionary.

flask.Blueprint.register_error_handler

`Blueprint.register_error_handler`(*code_or_exception*, *f*)[source]

Non-decorator version of the `errorhandler()` error attach function, akin to the `register_error_handler()` application-wide function of the `Flask` object but for error handlers limited to this blueprint.

New in version 0.11.

flask.Blueprint.route

`Blueprint.route`(*rule*, ***options*)[source]

Like `Flask.route()` but for a blueprint. The endpoint for the `url_for()` function is prefixed with the name of the blueprint.

flask.Blueprint.send_static_file

`Blueprint.send_static_file`(*filename*)

Function used internally to send static files from the static folder to the browser.

New in version 0.5.

flask.Blueprint.teardown_app_request

`Blueprint.teardown_app_request`(*f*)[source]

Like `Flask.teardown_request()` but for a blueprint. Such a function is executed when tearing down each request, even if outside of the blueprint.

flask.Blueprint.teardown_request

`Blueprint.teardown_request`(*f*)[source]

Like `Flask.teardown_request()` but for a blueprint. This function is only executed when tearing down requests handled by a function of that blueprint. Teardown request functions are executed when the request context is popped, even when no actual request was performed.

flask.Blueprint.url_defaults

`Blueprint.url_defaults`(*f*)[source]

Callback function for URL defaults for this blueprint. It's called with the endpoint and values and should update the values passed in place.

flask.Blueprint.url_value_preprocessor

`Blueprint.url_value_preprocessor`(*f*)[source]

Registers a function as URL value preprocessor for this blueprint. It's called before the view functions are called and can modify the url values provided.

flask.Blueprint.has_static_folder

`Blueprint.has_static_folder`

This is `True` if the package bound object's container has a folder for static files.

New in version 0.5.

flask.Blueprint.jinja_loader

`Blueprint.jinja_loader`

The Jinja loader for this package bound object.

New in version 0.5.

flask.Blueprint.static_folder

`Blueprint.static_folder`

The absolute path to the configured static folder.

flask.Config

class `flask.Config`(*root_path*, *defaults=None*)[source]

Works exactly like a dict but provides ways to fill it from files or special dictionaries. There are two common patterns to populate the config.

Either you can fill the config from a config file:

```
app.config.from_pyfile('yourconfig.cfg')
```

Or alternatively you can define the configuration options in the module that calls `from_object()` or provide an import path to a module that should be loaded. It is also possible to tell it to use the same module and with that provide the configuration values just before the call:

```
DEBUG = True
SECRET_KEY = 'development key'
app.config.from_object(__name__)
```

In both cases (loading from any Python file or loading from modules), only uppercase keys are added to the config. This makes it possible to use lowercase values in the config file for temporary values that are not added to the config or to define the config keys in the same file that implements the application.

Probably the most interesting way to load configurations is from an environment variable pointing to a file:

```
app.config.from_envvar('YOURAPPLICATION_SETTINGS')
```

In this case before launching the application you have to set this environment variable to the file you want to use. On Linux and OS X use the export statement:

```
export YOURAPPLICATION_SETTINGS='/path/to/config/file'
```

On windows use *set* instead.

Parameters:
- **root_path** – path to which files are read relative from. When the config object is created by the application, this is the application’s `root_path`.
- **defaults** – an optional dictionary of default values

Methods

`__init__`(root_path[, defaults])	
`clear`(() -> None. Remove all items from D.)	
`copy`(() -> a shallow copy of D)	

`from_envvar`(variable_name[, silent])	Loads a configuration from an environment variable pointing to a configuration file.
`from_json`(filename[, silent])	Updates the values in the config from a JSON file.
`from_mapping`(*mapping, **kwargs)	Updates the config like `update()` ignoring items with non-upper keys.
`from_object`(obj)	Updates the values from the given object.
`from_pyfile`(filename[, silent])	Updates the values in the config from a Python file.
`fromkeys`(...)	v defaults to None.
`get`((k[,d]) -> D[k] if k in D, ...)	
`get_namespace`(namespace[, lowercase, ...])	Returns a dictionary containing a subset of configuration options that match the specified namespace/prefix.
`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`((k[,d]) -> v, ...)	If key is not found, d is returned if given, otherwise KeyError is raised
`popitem`(() -> (k, v), ...)	2-tuple; but raise KeyError if D is empty.
`setdefault`((k[,d]) -> D.get(k,d), ...)	
`update`(([E, ...)	If E present and has a .keys() method, does: for k in E: D[k] = E[k]
`values`(() -> list of D's values)	
`viewitems`(...)	
`viewkeys`(...)	
`viewvalues`(...)	

flask.Config.from_envvar

`Config.from_envvar`(*variable_name*, *silent=False*)[source]

Loads a configuration from an environment variable pointing to a configuration file. This is basically just a shortcut with nicer error messages for this line of code:

```
app.config.from_pyfile(os.environ['YOURAPPLICATION_SETTINGS'])
```

Parameters:
- **variable_name** – name of the environment variable
- **silent** – set to `True` if you want silent failure for missing files.

Returns: bool. `True` if able to load config, `False` otherwise.

flask.Config.from_json

`Config.from_json`(*filename*, *silent=False*)[source]

Updates the values in the config from a JSON file. This function behaves as if the JSON object was a dictionary and passed to the `from_mapping()` function.

Parameters:
- **filename** – the filename of the JSON file. This can either be an absolute filename or a filename relative to the root path.
- **silent** – set to `True` if you want silent failure for missing files.

flask.Config.from_mapping

`Config.from_mapping`(**mapping*, ***kwargs*)[source]

Updates the config like `update()` ignoring items with non-upper keys.

---

flask.Config.from_object

`Config.from_object`(*obj*)[source]

Updates the values from the given object. An object can be of one of the following two types:

- a string: in this case the object with that name will be imported
- an actual object reference: that object is used directly

Objects are usually either modules or classes.

Just the uppercase variables in that object are stored in the config. Example usage:

```
app.config.from_object('yourapplication.default_config')
from yourapplication import default_config
app.config.from_object(default_config)
```

You should not use this function to load the actual configuration but rather configuration defaults. The actual config should be loaded with `from_pyfile()` and ideally from a location not within the package because the package might be installed system wide.

Parameters: **obj** – an import name or object

flask.Config.from_pyfile

`Config.from_pyfile`(*filename*, *silent=False*)[source]

Updates the values in the config from a Python file. This function behaves as if the file was imported as module with the `from_object()` function.

Parameters:
- **filename** – the filename of the config. This can either be an absolute filename or a filename relative to the root path.
- **silent** – set to `True` if you want silent failure for missing files.

flask.Config.get_namespace

`Config.get_namespace`(*namespace*, *lowercase=True*, *trim_namespace=True*)[source]

Returns a dictionary containing a subset of configuration options that match the specified namespace/prefix. Example usage:

```
app.config['IMAGE_STORE_TYPE'] = 'fs'
app.config['IMAGE_STORE_PATH'] = '/var/app/images'
app.config['IMAGE_STORE_BASE_URL'] = 'http://img.website.com'
image_store_config = app.config.get_namespace('IMAGE_STORE_')
```

The resulting dictionary *image_store_config* would look like:

```
{
    'type': 'fs',
    'path': '/var/app/images',
    'base_url': 'http://img.website.com'
}
```

This is often useful when configuration options map directly to keyword arguments in functions or class constructors.

Parameters:
- **namespace** – a configuration namespace
- **lowercase** – a flag indicating if the keys of the resulting dictionary should

be lowercase

- **trim_namespace –** a flag indicating if the keys of the resulting dictionary should not include the namespace

flask.Flask

class `flask.Flask`(*import_name*, *static_path=None*, *static_url_path=None*, *static_folder='static'*, *template_folder='templates'*, *instance_path=None*, *instance_relative_config=False*, *root_path=None*)
[source]

The flask object implements a WSGI application and acts as the central object. It is passed the name of the module or package of the application. Once it is created it will act as a central registry for the view functions, the URL rules, template configuration and much more.

The name of the package is used to resolve resources from inside the package or the folder the module is contained in depending on if the package parameter resolves to an actual python package (a folder with an `__init__.py` file inside) or a standard module (just a `.py` file).

For more information about resource loading, see `open_resource()`.

Usually you create a `Flask` instance in your main module or in the `__init__.py` file of your package like this:

```
from flask import Flask
app = Flask(__name__)
```

About the First Parameter

The idea of the first parameter is to give Flask an idea of what belongs to your application. This name is used to find resources on the filesystem, can be used by extensions to improve debugging information and a lot more.

So it’s important what you provide there. If you are using a single module, *__name__* is always the correct value. If you however are using a package, it’s usually recommended to hardcode the name of your package there.

For example if your application is defined in `yourapplication/app.py` you should create it with one of the two versions below:

```
app = Flask('yourapplication')
app = Flask(__name__.split('.')[0])
```

Why is that? The application will work even with *__name__*, thanks to how resources are looked up. However it will make debugging more painful. Certain extensions can make assumptions based on the import name of your application. For example the Flask-SQLAlchemy extension will look for the code in your application that triggered an SQL query in debug mode. If the import name is not properly set up, that debugging information is lost. (For example it would only pick up SQL queries in *yourapplication.app* and not *yourapplication.views.frontend*)

New in version 0.7: The *static_url_path*, *static_folder*, and *template_folder* parameters were added.

New in version 0.8: The *instance_path* and *instance_relative_config* parameters were added.

New in version 0.11: The *root_path* parameter was added.

Parameters:

- **import_name** – the name of the application package
- **static_url_path** – can be used to specify a different path for the static files on the web. Defaults to the name of the *static_folder* folder.
- **static_folder** – the folder with static files that should be served at *static_url_path*. Defaults to the `'static'` folder in the root path of the application.
- **template_folder** – the folder that contains the templates that should be used by the application. Defaults to `'templates'` folder in the root path of the application.
- **instance_path** – An alternative instance path for the application. By default the folder `'instance'` next to the package or module is assumed to be the instance path.
- **instance_relative_config** – if set to `True` relative filenames for loading the config are assumed to be relative to the instance path instead of the application root.
- **root_path** – Flask by default will automatically calculate the path to the root of the application. In certain situations this cannot be achieved (for instance if the package is a Python 3 namespace package) and needs to be manually defined.

Methods

`__init__`(import_name[, static_path, ...])	
`add_template_filter`(*args, **kwargs)	Register a custom template filter.
`add_template_global`(*args, **kwargs)	Register a custom template global function.
`add_template_test`(*args, **kwargs)	Register a custom template test.
`add_url_rule`(*args, **kwargs)	Connects a URL rule.
`after_request`(*args, **kwargs)	Register a function to be run after each request.
`app_context`()	Binds the application only.
`auto_find_instance_path`()	Tries to locate the instance path if it was not provided to the constructor of the application class.
`before_first_request`(*args, **kwar	Registers a function to be run before the first request to this instance of the application.

gs)	
before_request(*args, **kwargs)	Registers a function to run before each request.
context_processor(*args, **kwargs)	Registers a template context processor function.
create_global_jinja_loader()	Creates the loader for the Jinja2 environment.
create_jinja_environment()	Creates the Jinja2 environment based on `jinja_options` and `select_jinja_autoescape()`.
create_url_adapter(request)	Creates a URL adapter for the given request.
dispatch_request()	Does the request dispatching.
do_teardown_appcontext([exc])	Called when an application context is popped.
do_teardown_request([exc])	Called after the actual request dispatching and will call every as `teardown_request()` decorated function.
endpoint(*args, **kwargs)	A decorator to register a function as an endpoint.
errorhandler(*args, **kwargs)	A decorator that is used to register a function give a given error code.
full_dispatch_request()	Dispatches the request and on top of that performs request pre and postprocessing as well as HTTP exception catching and error handling.
get_send_file_max_age(filename)	Provides default cache_timeout for the `send_file()` functions.
handle_exception(e)	Default exception handling that kicks in when an exception occurs that is not caught.
handle_http_exception(e)	Handles an HTTP exception.
handle_url_build_error(error, endpoint, values)	Handle `BuildError` on `url_for()`.
handle_user_exception(e)	This method is called whenever an exception occurs that should be handled.
init_jinja_globals()	Deprecated.
inject_url_defaults(endpoint, values)	Injects the URL defaults for the given endpoint directly into the values dictionary passed.
iter_blueprints()	Iterates over all blueprints by the order they were registered.
log_exception(exc_info)	Logs an exception.

make_config([instance_relative])	Used to create the config attribute by the Flask constructor.
make_default_options_response()	This method is called to create the default `OPTIONS` response.
make_null_session()	Creates a new instance of a missing session.
make_response(rv)	Converts the return value from a view function to a real response object that is an instance of `response_class`.
make_shell_context()	Returns the shell context for an interactive shell for this application.
open_instance_resource(resource[, mode])	Opens a resource from the application's instance folder (`instance_path`).
open_resource(resource[, mode])	Opens a resource from the application's resource folder.
open_session(request)	Creates or opens a new session.
preprocess_request()	Called before the actual request dispatching and will call each `before_request()` decorated function, passing no arguments.
process_response(response)	Can be overridden in order to modify the response object before it's sent to the WSGI server.
raise_routing_exception(request)	Exceptions that are recording during routing are reraised with this method.
register_blueprint(*args, **kwargs)	Register a blueprint on the application.
register_error_handler(code_or_exception, f)	Alternative error attach function to the `errorhandler()` decorator that is more straightforward to use for non decorator usage.
request_context(environ)	Creates a `RequestContext` from the given environment and binds it to the current context.
route(rule, **options)	A decorator that is used to register a view function for a given URL rule.
run([host, port, debug])	Runs the application on a local development server.
save_session(session, response)	Saves the session if it needs updates.
select_jinja_autoescape(filename)	Returns `True` if autoescaping should be active for the given template name.
send_static_file(filename)	Function used internally to send static files from the static folder to the browser.
shell_context_processor(*args, **kwargs)	Registers a shell context processor function.

should_ignore_error(error)	This is called to figure out if an error should be ignored or not as far as the teardown system is concerned.
teardown_appcontext(*args, **kwargs)	Registers a function to be called when the application context ends.
teardown_request(*args, **kwargs)	Register a function to be run at the end of each request, regardless of whether there was an exception or not.
template_filter(*args, **kwargs)	A decorator that is used to register custom template filter.
template_global(*args, **kwargs)	A decorator that is used to register a custom template global function.
template_test(*args, **kwargs)	A decorator that is used to register custom template test.
test_client([use_cookies])	Creates a test client for this application.
test_request_context(*args, **kwargs)	Creates a WSGI environment from the given values (see `werkzeug.test.EnvironBuilder` for more information, this function accepts the same arguments).
trap_http_exception(e)	Checks if an HTTP exception should be trapped or not.
try_trigger_before_first_request_functions()	Called before each request and will ensure that it triggers the `before_first_request_funcs` and only exactly once per application instance (which means process usually).
update_template_context(context)	Update the template context with some commonly used variables.
url_defaults(*args, **kwargs)	Callback function for URL defaults for all view functions of the application.
url_value_preprocessor(*args, **kwargs)	Registers a function as URL value preprocessor for all view functions of the application.
wsgi_app(environ, start_response)	The actual WSGI application.

Attributes

debug	The debug flag.
default_config	Default configuration parameters.
error_handlers	
got_first_request	This attribute is set to `True` if the application started handling the first request.

has_static_folder	This is True if the package bound object's container has a folder for static files.
jinja_env	The Jinja2 environment used to load templates.
jinja_loader	The Jinja loader for this package bound object.
jinja_options	Options that are passed directly to the Jinja2 environment.
logger	A logging.Logger object for this application.
logger_name	The name of the logger to use.
name	The name of the application.
permanent_session_lifetime	A timedelta which is used to set the expiration date of a permanent session.
preserve_context_on_exception	Returns the value of the PRESERVE_CONTEXT_ON_EXCEPTION configuration value in case it's set, otherwise a sensible default is returned.
propagate_exceptions	Returns the value of the PROPAGATE_EXCEPTIONS configuration value in case it's set, otherwise a sensible default is returned.
request_globals_class	
secret_key	If a secret key is set, cryptographic components can use this to sign cookies and other things.
send_file_max_age_default	A timedelta which is used as default cache_timeout for the send_file() functions.
session_cookie_name	The secure cookie uses this for the name of the session cookie.
session_interface	the session interface to use. By default an instance of
static_folder	The absolute path to the configured static folder.
static_url_path	
test_client_class	the test client that is used with when *test_client* is used.
testing	The testing flag.
use_x_sendfile	Enable this if you want to use the X-Sendfile feature.

flask.Flask.__init__

`Flask.__init__`(*import_name*, *static_path=None*, *static_url_path=None*, *static_folder='static'*, *template_folder='templates'*, *instance_path=None*, *instance_relative_config=False*, *root_path=None*)
[source]

flask.Flask.add_template_filter

`Flask.add_template_filter`(**args*, ***kwargs*)[source]

Register a custom template filter. Works exactly like the `template_filter()` decorator.

Parameters: **name** – the optional name of the filter, otherwise the function name will be used.

flask.Flask.add_template_global

`Flask.add_template_global`(**args*, ***kwargs*)[source]

Register a custom template global function. Works exactly like the `template_global()` decorator.

New in version 0.10.

Parameters: **name** – the optional name of the global function, otherwise the function name will be used.

flask.Flask.add_template_test

`Flask.add_template_test`(**args*, ***kwargs*)[source]

Register a custom template test. Works exactly like the `template_test()` decorator.

New in version 0.10.

Parameters: **name** – the optional name of the test, otherwise the function name will be used.

flask.Flask.add_url_rule

`Flask.add_url_rule`(**args*, ***kwargs*)[source]

Connects a URL rule. Works exactly like the `route()` decorator. If a view_func is provided it will be registered with the endpoint.

Basically this example:

```
@app.route('/')
def index():
    pass
```

Is equivalent to the following:

```
def index():
    pass
app.add_url_rule('/', 'index', index)
```

If the view_func is not provided you will need to connect the endpoint to a view function like so:

```
app.view_functions['index'] = index
```

Internally `route()` invokes `add_url_rule()` so if you want to customize the behavior via subclassing you only need to change this method.

For more information refer to URL Route Registrations.

Changed in version 0.2: *view_func* parameter added.

Changed in version 0.6: `OPTIONS` is added automatically as method.

Parameters:

- **rule** – the URL rule as string
- **endpoint** – the endpoint for the registered URL rule. Flask itself assumes the name of the view function as endpoint
- **view_func** – the function to call when serving a request to the provided endpoint
- **options** – the options to be forwarded to the underlying `Rule` object. A change to Werkzeug is handling of method options. methods is a list of methods this rule should be limited to (`GET`, `POST` etc.). By default a rule just listens for `GET` (and implicitly `HEAD`). Starting with Flask 0.6, `OPTIONS` is implicitly added and handled by the standard request handling.

flask.Flask.after_request

`Flask.after_request`(**args*, ***kwargs*)[source]

Register a function to be run after each request.

Your function must take one parameter, an instance of `response_class` and return a new response object or the same (see `process_response()`).

As of Flask 0.7 this function might not be executed at the end of the request in case an unhandled exception occurred.

flask.Flask.app_context

`Flask.app_context()`[source]

Binds the application only. For as long as the application is bound to the current context the `flask.current_app` points to that application. An application context is automatically created when a request context is pushed if necessary.

Example usage:

```
with app.app_context():
    ...
```

flask.Flask.auto_find_instance_path

`Flask.auto_find_instance_path()`[source]

Tries to locate the instance path if it was not provided to the constructor of the application class. It will basically calculate the path to a folder named `instance` next to your main file or the package.

New in version 0.8.

flask.Flask.before_first_request

`Flask.before_first_request`(**args*, ***kwargs*)[source]

Registers a function to be run before the first request to this instance of the application.

The function will be called without any arguments and its return value is ignored.

New in version 0.8.

flask.Flask.before_request

`Flask.before_request`(**args*, ***kwargs*)[source]

Registers a function to run before each request.

The function will be called without any arguments. If the function returns a non-None value, it's handled as if it was the return value from the view and further request handling is stopped.

flask.Flask.context_processor

`Flask.context_processor`(**args*, ***kwargs*)[source]

Registers a template context processor function.

flask.Flask.create_global_jinja_loader

`Flask.create_global_jinja_loader`()[source]

Creates the loader for the Jinja2 environment. Can be used to override just the loader and keeping the rest unchanged. It's discouraged to override this function. Instead one should override the `jinja_loader()` function instead.

The global loader dispatches between the loaders of the application and the individual blueprints.

flask.Flask.create_jinja_environment

`Flask.create_jinja_environment`()[source]

Creates the Jinja2 environment based on `jinja_options` and `select_jinja_autoescape()`. Since 0.7 this also adds the Jinja2 globals and filters after initialization. Override this function to customize the behavior.

New in version 0.5.

Changed in version 0.11: `Environment.auto_reload` set in accordance with `TEMPLATES_AUTO_RELOAD` configuration option.

flask.Flask.create_url_adapter

`Flask.create_url_adapter`(*request*)[source]

Creates a URL adapter for the given request. The URL adapter is created at a point where the request context is not yet set up so the request is passed explicitly.

New in version 0.6.

Changed in version 0.9: This can now also be called without a request object when the URL adapter is created for the application context.

flask.Flask.dispatch_request

`Flask.dispatch_request`()[source]

Does the request dispatching. Matches the URL and returns the return value of the view or error handler. This does not have to be a response object. In order to convert the return value to a proper response object, call `make_response()`.

Changed in version 0.7: This no longer does the exception handling, this code was moved to the new `full_dispatch_request()`.

flask.Flask.do_teardown_appcontext

`Flask.do_teardown_appcontext`(*exc=<object object>*)[source]

Called when an application context is popped. This works pretty much the same as `do_teardown_request()` but for the application context.

flask.Flask.do_teardown_request

`Flask.do_teardown_request`(*exc=<object object>*)[source]

Called after the actual request dispatching and will call every as `teardown_request()` decorated function. This is not actually called by the `Flask` object itself but is always triggered when the request context is popped. That way we have a tighter control over certain resources under testing environments.

Changed in version 0.9: Added the *exc* argument. Previously this was always using the current exception information.

flask.Flask.endpoint

`Flask.endpoint`(**args*, ***kwargs*)[source]

A decorator to register a function as an endpoint. Example:

```
@app.endpoint('example.endpoint')
def example():
    return "example"
```

Parameters: **endpoint** – the name of the endpoint

flask.Flask.errorhandler

`Flask.errorhandler`(**args*, ***kwargs*)[source]

A decorator that is used to register a function give a given error code. Example:

```
@app.errorhandler(404)
def page_not_found(error):
    return 'This page does not exist', 404
```

You can also register handlers for arbitrary exceptions:

```
@app.errorhandler(DatabaseError)
def special_exception_handler(error):
    return 'Database connection failed', 500
```

You can also register a function as error handler without using the `errorhandler()` decorator. The following example is equivalent to the one above:

```
def page_not_found(error):
    return 'This page does not exist', 404
app.error_handler_spec[None][404] = page_not_found
```

Setting error handlers via assignments to `error_handler_spec` however is discouraged as it requires fiddling with nested dictionaries and the special case for arbitrary exception types.

The first `None` refers to the active blueprint. If the error handler should be application wide `None` shall be used.

New in version 0.7: Use `register_error_handler()` instead of modifying `error_handler_spec` directly, for application wide error handlers.

New in version 0.7: One can now additionally also register custom exception types that do not necessarily have to be a subclass of the `HTTPException` class.

Parameters: **code** – the code as integer for the handler

flask.Flask.full_dispatch_request

`Flask.full_dispatch_request`()[source]

Dispatches the request and on top of that performs request pre and postprocessing as well as HTTP exception catching and error handling.

flask.Flask.get_send_file_max_age

`Flask.get_send_file_max_age`(*filename*)

Provides default cache_timeout for the `send_file()` functions.

By default, this function returns `SEND_FILE_MAX_AGE_DEFAULT` from the configuration of `current_app`.

Static file functions such as `send_from_directory()` use this function, and `send_file()` calls this function on `current_app` when the given cache_timeout is `None`. If a cache_timeout is given in `send_file()`, that timeout is used; otherwise, this method is called.

This allows subclasses to change the behavior when sending files based on the filename. For example, to set the cache timeout for .js files to 60 seconds:

```
class MyFlask(flask.Flask):
    def get_send_file_max_age(self, name):
        if name.lower().endswith('.js'):
            return 60
        return flask.Flask.get_send_file_max_age(self, name)
```

flask.Flask.handle_exception

`Flask.handle_exception`(*e*)[source]

Default exception handling that kicks in when an exception occurs that is not caught. In debug mode the exception will be re-raised immediately, otherwise it is logged and the handler for a 500 internal server error is used. If no such handler exists, a default 500 internal server error message is displayed.

flask.Flask.handle_http_exception

`Flask.handle_http_exception`(*e*)[source]

Handles an HTTP exception. By default this will invoke the registered error handlers and fall back to returning the exception as response.

flask.Flask.handle_url_build_error

`Flask.handle_url_build_error`(*error*, *endpoint*, *values*)[source]

Handle `BuildError` on `url_for()`.

flask.Flask.handle_user_exception

`Flask.handle_user_exception`(*e*)[source]

This method is called whenever an exception occurs that should be handled. A special case are `HTTPException`s which are forwarded by this function to the `handle_http_exception()` method. This function will either return a response value or reraise the exception with the same traceback.

flask.Flask.init_jinja_globals

`Flask.init_jinja_globals`()[source]

Deprecated. Used to initialize the Jinja2 globals.

New in version 0.5.

Changed in version 0.7: This method is deprecated with 0.7. Override `create_jinja_environment()` instead.

flask.Flask.inject_url_defaults

`Flask.inject_url_defaults`(*endpoint*, *values*)[source]

Injects the URL defaults for the given endpoint directly into the values dictionary passed. This is used internally and automatically called on URL building.

flask.Flask.iter_blueprints

`Flask.iter_blueprints`()[source]

Iterates over all blueprints by the order they were registered.

flask.Flask.log_exception

`Flask.log_exception`(*exc_info*)[source]

Logs an exception. This is called by `handle_exception()` if debugging is disabled and right before the handler is called. The default implementation logs the exception as error on the `logger`.

flask.Flask.make_config

`Flask.make_config`(*instance_relative=False*)[source]

Used to create the config attribute by the Flask constructor. The *instance_relative* parameter is passed in from the constructor of Flask (there named *instance_relative_config*) and indicates if the config should be relative to the instance path or the root path of the application.

flask.Flask.make_default_options_response

`Flask.make_default_options_response`()[source]

This method is called to create the default `OPTIONS` response. This can be changed through subclassing to change the default behavior of `OPTIONS` responses.

flask.Flask.make_null_session

`Flask.make_null_session`()[source]

Creates a new instance of a missing session. Instead of overriding this method we recommend replacing the `session_interface`.

flask.Flask.make_response

`Flask.make_response`(*rv*)[source]

Converts the return value from a view function to a real response object that is an instance of `response_class`.

The following types are allowed for *rv*:

`response_class`	the object is returned unchanged
`str`	a response object is created with the string as body
`unicode`	a response object is created with the string encoded to utf-8 as body
a WSGI function	the function is called as WSGI application and buffered as response object
`tuple`	A tuple in the form `(response, status, headers)` or `(response, headers)` where *response* is any of the types defined here, *status* is a string or an integer and *headers* is a list or a dictionary with header values.

Parameters: **rv** – the return value from the view function

flask.Flask.make_shell_context

`Flask.make_shell_context()`[source]

Returns the shell context for an interactive shell for this application. This runs all the registered shell context processors.

flask.Flask.open_instance_resource

`Flask.open_instance_resource`(*resource*, *mode='rb'*)[source]

Opens a resource from the application's instance folder (`instance_path`). Otherwise works like `open_resource()`. Instance resources can also be opened for writing.

Parameters:

- **resource** – the name of the resource. To access resources within subfolders use forward slashes as separator.
- **mode** – resource file opening mode, default is 'rb'.

flask.Flask.open_resource

`Flask.open_resource`(*resource*, *mode='rb'*)

Opens a resource from the application's resource folder. To see how this works, consider the following folder structure:

```
/myapplication.py
/schema.sql
/static
    /style.css
/templates
    /layout.html
    /index.html
```

If you want to open the `schema.sql` file you would do the following:

```
with app.open_resource('schema.sql') as f:
    contents = f.read()
    do_something_with(contents)
```

Parameters:

- **resource** – the name of the resource. To access resources within subfolders use forward slashes as separator.
- **mode** – resource file opening mode, default is 'rb'.

flask.Flask.open_session

`Flask.open_session`(*request*)[source]

Creates or opens a new session. Default implementation stores all session data in a signed cookie. This requires that the `secret_key` is set. Instead of overriding this method we recommend replacing the `session_interface`.

Parameters: **request** – an instance of `request_class`.

flask.Flask.preprocess_request

`Flask.preprocess_request`()[source]

Called before the actual request dispatching and will call each `before_request()` decorated function, passing no arguments. If any of these functions returns a value, it’s handled as if it was the return value from the view and further request handling is stopped.

This also triggers the `url_value_processor()` functions before the actual `before_request()` functions are called.

flask.Flask.process_response

`Flask.process_response`(*response*)[source]

Can be overridden in order to modify the response object before it’s sent to the WSGI server. By default this will call all the `after_request()` decorated functions.

Changed in version 0.5: As of Flask 0.5 the functions registered for after request execution are called in reverse order of registration.

Parameters: **response** – a `response_class` object.
Returns: a new response object or the same, has to be an instance of `response_class`.

flask.Flask.raise_routing_exception

`Flask.raise_routing_exception`(*request*)[source]

Exceptions that are recording during routing are reraised with this method. During debug we are not reraising redirect requests for non `GET`, `HEAD`, or `OPTIONS` requests and we’re raising a different error instead to help debug situations.

Internal:

flask.Flask.register_blueprint

`Flask.register_blueprint`(**args*, ***kwargs*)[source]

Register a blueprint on the application. For information about blueprints head over to Modular Applications with Blueprints.

The blueprint name is passed in as the first argument. Options are passed as additional keyword arguments and forwarded to *blueprints* in an “options” dictionary.

Parameters:

- **subdomain** – set a subdomain for the blueprint
- **url_prefix** – set the prefix for all URLs defined on the blueprint. `(url_prefix='/<lang code>')`
- **url_defaults** – a dictionary with URL defaults that is added to each and every URL defined with this blueprint
- **static_folder** – add a static folder to urls in this blueprint
- **static_url_path** – add a static url path to urls in this blueprint
- **template_folder** – set an alternate template folder
- **root_path** – set an alternate root path for this blueprint

flask.Flask.register_error_handler

`Flask.register_error_handler`(*code_or_exception*, *f*)[source]

Alternative error attach function to the `errorhandler()` decorator that is more straightforward to use for non decorator usage.

flask.Flask.request_context

`Flask.request_context`(*environ*)[source]

Creates a `RequestContext` from the given environment and binds it to the current context. This must be used in combination with the `with` statement because the request is only bound to the current context for the duration of the `with` block.

Example usage:

```
with app.request_context(environ):
    do_something_with(request)
```

The object returned can also be used without the `with` statement which is useful for working in the shell. The example above is doing exactly the same as this code:

```
ctx = app.request_context(environ)
ctx.push()
```

```
try:
    do_something_with(request)
finally:
    ctx.pop()
```

Changed in version 0.3: Added support for non-with statement usage and `with` statement is now passed the ctx object.

Parameters: **environ** – a WSGI environment

flask.Flask.route

`Flask.route`(*rule*, ***options*)[source]

A decorator that is used to register a view function for a given URL rule. This does the same thing as `add_url_rule()` but is intended for decorator usage:

```
@app.route('/')
def index():
    return 'Hello World'
```

For more information refer to URL Route Registrations.

Parameters:

- **rule** – the URL rule as string
- **endpoint** – the endpoint for the registered URL rule. Flask itself assumes the name of the view function as endpoint
- **options** – the options to be forwarded to the underlying `Rule` object. A change to Werkzeug is handling of method options. methods is a list of methods this rule should be limited to (`GET`, `POST` etc.). By default a rule just listens for `GET` (and implicitly `HEAD`). Starting with Flask 0.6, `OPTIONS` is implicitly added and handled by the standard request handling.

flask.Flask.run

`Flask.run`(*host=None*, *port=None*, *debug=None*, ***options*)[source]

Runs the application on a local development server.

Do not use `run()` in a production setting. It is not intended to meet security and performance requirements for a production server. Instead, see Application Deployment for WSGI server recommendations.

If the `debug` flag is set the server will automatically reload for code changes and show a debugger in case an exception happened.

If you want to run the application in debug mode, but disable the code execution on the interactive debugger, you can pass `use_evalex=False` as parameter. This will keep the debugger's traceback screen active, but disable code execution.

It is not recommended to use this function for development with automatic reloading as this is badly supported. Instead you should be using the **flask** command line script's `run` support.

Keep in Mind

Flask will suppress any server error with a generic error page unless it is in debug mode. As such to enable just the interactive debugger without the code reloading, you have to invoke `run()` with `debug=True` and `use_reloader=False`. Setting `use_debugger` to `True` without being in debug mode won't catch any exceptions because there won't be any to catch.

Changed in version 0.10: The default port is now picked from the `SERVER_NAME` variable.

Parameters:

- **host** – the hostname to listen on. Set this to `'0.0.0.0'` to have the server available externally as well. Defaults to `'127.0.0.1'`.
- **port** – the port of the webserver. Defaults to `5000` or the port defined in the `SERVER_NAME` config variable if present.
- **debug** – if given, enable or disable debug mode. See `debug`.
- **options** – the options to be forwarded to the underlying Werkzeug server. See `werkzeug.serving.run_simple()` for more information.

flask.Flask.save_session

`Flask.save_session`(*session*, *response*)[source]

Saves the session if it needs updates. For the default implementation, check `open_session()`. Instead of overriding this method we recommend replacing the `session_interface`.

Parameters:

- **session** – the session to be saved (a `SecureCookie` object)
- **response** – an instance of `response_class`

flask.Flask.select_jinja_autoescape

`Flask.select_jinja_autoescape`(*filename*)[source]

Returns `True` if autoescaping should be active for the given template name. If no template name is given, returns *True*.

flask.Flask.send_static_file

`Flask.send_static_file`(*filename*)

Function used internally to send static files from the static folder to the browser.

flask.Flask.shell_context_processor

`Flask.shell_context_processor`(**args*, ***kwargs*)[source]

Registers a shell context processor function.

flask.Flask.should_ignore_error

`Flask.should_ignore_error`(*error*)[source]

This is called to figure out if an error should be ignored or not as far as the teardown system is concerned. If this function returns `True` then the teardown handlers will not be passed the error.

flask.Flask.teardown_appcontext

`Flask.teardown_appcontext`(**args*, ***kwargs*)[source]

Registers a function to be called when the application context ends. These functions are typically also called when the request context is popped.

Example:

```
ctx = app.app_context()
ctx.push()
...
ctx.pop()
```

When `ctx.pop()` is executed in the above example, the teardown functions are called just before the app context moves from the stack of active contexts. This becomes relevant if you are using such constructs in tests.

Since a request context typically also manages an application context it would also be called when you pop a request context.

When a teardown function was called because of an exception it will be passed an error object.

The return values of teardown functions are ignored.

flask.Flask.teardown_request

`Flask.teardown_request`(**args*, ***kwargs*)[source]

Register a function to be run at the end of each request, regardless of whether there was an exception or not. These functions are executed when the request context is popped, even if not an actual request was performed.

Example:

```
ctx = app.test_request_context()
ctx.push()
...
ctx.pop()
```

When `ctx.pop()` is executed in the above example, the teardown functions are called just before the request context moves from the stack of active contexts. This becomes relevant if you are using such constructs in tests.

Generally teardown functions must take every necessary step to avoid that they will fail. If they do execute code that might fail they will have to surround the execution of these code by try/except statements and log occurring errors.

When a teardown function was called because of a exception it will be passed an error object.

The return values of teardown functions are ignored.

Debug Note

In debug mode Flask will not tear down a request on an exception immediately. Instead it will keep it alive so that the interactive debugger can still access it. This behavior can be controlled by the `PRESERVE_CONTEXT_ON_EXCEPTION` configuration variable.

flask.Flask.template_filter

`Flask.template_filter`(**args*, ***kwargs*)[source]

A decorator that is used to register custom template filter. You can specify a name for the filter, otherwise the function name will be used. Example:

```
@app.template_filter()
def reverse(s):
    return s[::-1]
```

Parameters: **name** – the optional name of the filter, otherwise the function name will be used.

flask.Flask.template_global

`Flask.template_global`(**args*, ***kwargs*)[source]

A decorator that is used to register a custom template global function. You can specify a name for the global function, otherwise the function name will be used. Example:

```
@app.template_global()
def double(n):
    return 2 * n
```

New in version 0.10.

Parameters: **name** – the optional name of the global function, otherwise the function name will be used.

flask.Flask.template_test

`Flask.template_test`(**args*, ***kwargs*)[source]

A decorator that is used to register custom template test. You can specify a name for the test, otherwise the function name will be used. Example:

```
@app.template_test()
def is_prime(n):
    if n == 2:
        return True
    for i in range(2, int(math.ceil(math.sqrt(n))) + 1):
        if n % i == 0:
            return False
    return True
```

New in version 0.10.

Parameters: **name** – the optional name of the test, otherwise the function name will be used.

flask.Flask.test_client

`Flask.test_client`(*use_cookies=True*, ***kwargs*)[source]

Creates a test client for this application. For information about unit testing head over to 1 Testing Flask Applications.

Note that if you are testing for assertions or exceptions in your application code, you must set `app.testing = True` in order for the exceptions to propagate to the test client. Otherwise, the exception will be handled by the application (not visible to the test client) and the only indication of an AssertionError or other exception will be a 500 status code response to the test client. See the `testing` attribute. For example:

```
app.testing = True
client = app.test_client()
```

The test client can be used in a `with` block to defer the closing down of the context until the end of the `with` block. This is useful if you want to access the context locals for testing:

```
with app.test_client() as c:
    rv = c.get('/?vodka=42')
    assert request.args['vodka'] == '42'
```

Additionally, you may pass optional keyword arguments that will then be passed to the application's `test_client_class` constructor. For example:

```
from flask.testing import FlaskClient

class CustomClient(FlaskClient):
    def __init__(self, authentication=None, *args, **kwargs):
        FlaskClient.__init__(*args, **kwargs)
        self._authentication = authentication

app.test_client_class = CustomClient
client = app.test_client(authentication='Basic ....')
```

See `FlaskClient` for more information.

Changed in version 0.4: added support for `with` block usage for the client.

New in version 0.7: The *use_cookies* parameter was added as well as the ability to override the client to be used by setting the `test_client_class` attribute.

Changed in version 0.11: Added ***kwargs* to support passing additional keyword arguments to the constructor of `test_client_class`.

flask.Flask.test_request_context

`Flask.test_request_context`(**args*, ***kwargs*)[source]

Creates a WSGI environment from the given values (see `werkzeug.test.EnvironBuilder` for more information, this function accepts the same arguments).

flask.Flask.trap_http_exception

`Flask.trap_http_exception`(*e*)[source]

Checks if an HTTP exception should be trapped or not. By default this will return `False` for all exceptions except for a bad request key error if `TRAP_BAD_REQUEST_ERRORS` is set to `True`. It also returns `True` if `TRAP_HTTP_EXCEPTIONS` is set to `True`.

This is called for all HTTP exceptions raised by a view function. If it returns `True` for any exception the error handler for this exception is not called and it shows up as regular exception in the traceback. This is helpful for debugging implicitly raised HTTP exceptions.

flask.Flask.try_trigger_before_first_request_functions

`Flask.try_trigger_before_first_request_functions`()[source]

Called before each request and will ensure that it triggers the `before_first_request_funcs` and only exactly once per application instance (which means process usually).

Internal:

flask.Flask.update_template_context

`Flask.update_template_context`(*context*)[source]

Update the template context with some commonly used variables. This injects request, session, config and g into the template context as well as everything template context processors want to inject. Note that the as of Flask 0.6, the original values in the context will not be overridden if a context processor decides to return a value with the same key.

Parameters: **context** – the context as a dictionary that is updated in place to add extra variables.

flask.Flask.url_defaults

`Flask.url_defaults`(**args*, ***kwargs*)[source]

Callback function for URL defaults for all view functions of the application. It’s called with the endpoint and values and should update the values passed in place.

flask.Flask.url_value_preprocessor

`Flask.url_value_preprocessor`(**args*, ***kwargs*)[source]

Registers a function as URL value preprocessor for all view functions of the application. It's called before the view functions are called and can modify the url values provided.

flask.Flask.wsgi_app

`Flask.wsgi_app`(*environ*, *start_response*)[source]

The actual WSGI application. This is not implemented in *__call__* so that middlewares can be applied without losing a reference to the class. So instead of doing this:

```
app = MyMiddleware(app)
```

It's a better idea to do this instead:

```
app.wsgi_app = MyMiddleware(app.wsgi_app)
```

Then you still have the original application object around and can continue to call methods on it.

Changed in version 0.7: The behavior of the before and after request callbacks was changed under error conditions and a new callback was added that will always execute at the end of the request, independent on if an error occurred or not. See Callbacks and Errors.

Parameters:

- **environ** – a WSGI environment
- **start_response** – a callable accepting a status code, a list of headers and an optional exception context to start the response

flask.Flask.debug

`Flask.debug`

The debug flag. Set this to `True` to enable debugging of the application. In debug mode the debugger will kick in when an unhandled exception occurs and the integrated server will automatically reload the application if changes in the code are detected.

This attribute can also be configured from the config with the `DEBUG` configuration key. Defaults to `False`.

flask.Flask.default_config

`Flask.default_config` = *ImmutableDict({'JSON_AS_ASCII': True, 'USE_X_SENDFILE': False, 'SESSION_COOKIE_PATH': None, 'SESSION_COOKIE_DOMAIN': None, 'SESSION_COOKIE_NAME': 'session', 'DEBUG': False, 'LOGGER_HANDLER_POLICY': 'always', 'LOGGER_NAME': None, 'SESSION_COOKIE_SECURE': False, 'SECRET_KEY': None, 'EXPLAIN_TEMPLATE_LOADING': False, 'MAX_CONTENT_LENGTH': None,* *'PROPAGATE_EXCEPTIONS': None, 'APPLICATION_ROOT': None, 'SERVER_NAME': None, 'PREFERRED_URL_SCHEME': 'http', 'JSONIFY_PRETTYPRINT_REGULAR': True, 'TESTING': False, 'PERMANENT_SESSION_LIFETIME': datetime.timedelta(31), 'TEMPLATES_AUTO_RELOAD': None, 'TRAP_BAD_REQUEST_ERRORS': False, 'JSON_SORT_KEYS': True, 'JSONIFY_MIMETYPE': 'application/json', 'SESSION_COOKIE_HTTPONLY': True, 'SEND_FILE_MAX_AGE_DEFAULT': datetime.timedelta(0, 43200), 'PRESERVE_CONTEXT_ON_EXCEPTION': None, 'SESSION_REFRESH_EACH_REQUEST': True, 'TRAP_HTTP_EXCEPTIONS': False})*

Default configuration parameters.

`Flask.permanent_session_lifetime`

A `timedelta` which is used to set the expiration date of a permanent session. The default is 31 days which makes a permanent session survive for roughly one month.

This attribute can also be configured from the config with the `PERMANENT_SESSION_LIFETIME` configuration key. Defaults to `timedelta(days=31)`

flask.Flask.preserve_context_on_exception

`Flask.preserve_context_on_exception`

Returns the value of the `PRESERVE_CONTEXT_ON_EXCEPTION` configuration value in case it’s set, otherwise a sensible default is returned.

flask.Flask.secret_key

`Flask.secret_key`

If a secret key is set, cryptographic components can use this to sign cookies and other things. Set this to a complex random value when you want to use the secure cookie for instance.

This attribute can also be configured from the config with the `SECRET_KEY` configuration key. Defaults to `None`.

flask.Flask.send_file_max_age_default

`Flask.send_file_max_age_default`

A `timedelta` which is used as default cache_timeout for the `send_file()` functions. The default is 12 hours.

This attribute can also be configured from the config with the `SEND_FILE_MAX_AGE_DEFAULT` configuration key. This configuration variable can also be set with an integer value used as seconds. Defaults to `timedelta(hours=12)`

flask.Flask.session_cookie_name

`Flask.session_cookie_name`

The secure cookie uses this for the name of the session cookie.

This attribute can also be configured from the config with the `SESSION_COOKIE_NAME` configuration key. Defaults to `'session'`

flask.Flask.session_interface

`Flask.session_interface` = *<flask.sessions.SecureCookieSessionInterface object>*

the session interface to use. By default an instance of `SecureCookieSessionInterface` is used here.

flask.Flask.static_folder

`Flask.static_folder`

The absolute path to the configured static folder.

flask.Flask.testing

`Flask.testing`

The testing flag. Set this to `True` to enable the test mode of Flask extensions (and in the future probably also Flask itself). For example this might activate unittest helpers that have an additional runtime cost which should not be enabled by default.

If this is enabled and PROPAGATE_EXCEPTIONS is not changed from the default it’s implicitly enabled.

This attribute can also be configured from the config with the `TESTING` configuration key. Defaults to `False`.

flask.Flask.use_x_sendfile

`Flask.use_x_sendfile`

Enable this if you want to use the X-Sendfile feature. Keep in mind that the server has to support this. This only affects files sent with the `send_file()` method.

New in version 0.2.

This attribute can also be configured from the config with the `USE_X_SENDFILE` configuration key. Defaults to `False`.

flask.Markup

class `flask.Markup`[source]

Marks a string as being safe for inclusion in HTML/XML output without needing to be escaped. This implements the *__html__* interface a couple of frameworks and web applications use. `Markup` is a direct subclass of *unicode* and provides all the methods of *unicode* just that it escapes arguments passed and always returns *Markup*.

The *escape* function returns markup objects so that double escaping can't happen.

The constructor of the `Markup` class can be used for three different things: When passed an unicode object it's assumed to be safe, when passed an object with an HTML representation (has an *__html__* method) that representation is used, otherwise the object passed is converted into a unicode string and then assumed to be safe:

>>>

```
>>> Markup("Hello <em>World</em>!")
Markup(u'Hello <em>World</em>!')
>>> class Foo(object):
...  def __html__(self):
...   return '<a href="#">foo</a>'
...
>>> Markup(Foo())
Markup(u'<a href="#">foo</a>')
```

If you want object passed being always treated as unsafe you can use the `escape()` classmethod to create a `Markup` object:

>>>

```
>>> Markup.escape("Hello <em>World</em>!")
Markup(u'Hello &lt;em&gt;World&lt;/em&gt;!')
```

Operations on a markup string are markup aware which means that all arguments are passed through the `escape()` function:

>>>

```
>>> em = Markup("<em>%s</em>")
>>> em % "foo & bar"
Markup(u'<em>foo &amp; bar</em>')
>>> strong = Markup("<strong>%(text)s</strong>")
>>> strong % {'text': '<blink>hacker here</blink>'}
Markup(u'<strong>&lt;blink&gt;hacker here&lt;/blink&gt;</strong>')
>>> Markup("<em>Hello</em> ") + "<foo>"
Markup(u'<em>Hello</em> &lt;foo&gt;')
```

Methods

`capitalize`(() -> unicode)	Return a capitalized version of S, i.e.
`center`((width[, fillchar]) -> unicode)	Return S centered in a Unicode string of length width.
`count`((sub[, start[, end]]) -> int)	Return the number of non-overlapping occurrences of substring sub in Unicode string S[start:end].
`decode`(...)	Decodes S using the codec registered for encoding.
`encode`(...)	Encodes S using the codec registered for encoding.
`endswith`((suffix[, start[, end]]) -> bool)	Return True if S ends with the specified suffix, False otherwise.
`escape`(s)	Escape the string.
`expandtabs`(([tabsize]) -> unicode)	Return a copy of S where all tab characters are expanded using spaces.
`find`((sub [,start [,end]]) -> int)	Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end].
`format`(*args, **kwargs)	
`index`((sub [,start [,end]]) -> int)	Like S.find() but raise ValueError when the substring is not found.
`isalnum`(() -> bool)	Return True if all characters in S are alphanumeric and there is at least one character in S, False otherwise.
`isalpha`(() -> bool)	Return True if all characters in S are alphabetic and there is at least one character in S, False otherwise.
`isdecimal`(() -> bool)	Return True if there are only decimal characters in S, False otherwise.
`isdigit`(() -> bool)	Return True if all characters in S are digits and there is at least one character in S, False otherwise.
`islower`(() -> bool)	Return True if all cased characters in S are lowercase and there is at least one cased character in S, False otherwise.
`isnumeric`(() -> bool)	Return True if there are only numeric characters in S, False otherwise.
`isspace`(() -> bool)	Return True if all characters in S are whitespace and there is at least one character in S, False otherwise.
`istitle`(() -> bool)	Return True if S is a titlecased string and there is at least one character in S, i.e.
`isupper`(() -> bool)	Return True if all cased characters in S are uppercase and there is at least one cased character in S, False otherwise.
`join`((iterable) -> unicode)	Return a string which is the concatenation of the strings in the iterable.
`ljust`((width[, fillchar]) -> int)	Return S left-justified in a Unicode string of length width.
`lower`(() -> unicode)	Return a copy of the string S converted to lowercase.

lstrip(([chars]) -> unicode)	Return a copy of the string S with leading whitespace removed.
partition(sep)	
replace((old, new[, count]) -> unicode)	Return a copy of S with all occurrences of substring old replaced by new.
rfind((sub [,start [,end]]) -> int)	Return the highest index in S where substring sub is found, such that sub is contained within S[start:end].
rindex((sub [,start [, end]]) -> int)	Like S.rfind() but raise ValueError when the substring is not found.
rjust((width[, fillchar]) -> unicode)	Return S right-justified in a Unicode string of length width.
rpartition(sep)	
rsplit(([sep [,maxsplit]]) -> list of strings)	Return a list of the words in S, using sep as the delimiter string, starting at the end of the string and working to the front.
rstrip(([chars]) -> unicode)	Return a copy of the string S with trailing whitespace removed.
split(([sep [,maxsplit]]) -> list of strings)	Return a list of the words in S, using sep as the delimiter string.
splitlines((keepends=False) -> list of strings)	Return a list of the lines in S, breaking at line boundaries.
startswith((prefix[, start[, end]]) -> bool)	Return True if S starts with the specified prefix, False otherwise.
strip(([chars]) -> unicode)	Return a copy of the string S with leading and trailing whitespace removed.
striptags()	Unescape markup into an text_type string and strip all tags.
swapcase(() -> unicode)	Return a copy of S with uppercase characters converted to lowercase and vice versa.
title(() -> unicode)	Return a titlecased version of S, i.e.
translate((table) -> unicode)	Return a copy of the string S, where all characters have been mapped through the given translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, Unicode strings or None.
unescape()	Unescape markup again into an text_type string.
upper(() -> unicode)	Return a copy of S converted to uppercase.
zfill((width) -> unicode)	Pad a numeric string S with zeros on the left, to fill a field of the specified width.

flask.Request

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

The request object used by default in Flask. Remembers the matched endpoint and view arguments.

It is what ends up as `request`. If you want to replace the request object used you can subclass this and set `request_class` to your subclass.

The request object is a `Request` subclass and provides all of the attributes Werkzeug defines plus a few Flask specific ones.

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.
`get_json`([force, silent, cache])	Parses the incoming JSON request data and returns it.
`make_form_dat`	Creates the

a_parser()	form data parser.
on_json_loading_failed(e)	Called if decoding of the JSON data failed.

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`.
blueprint	The name of the current blueprint
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_erro	

rs	
endpoint	The endpoint that matched the request.
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_json	Indicates if this request is JSON or not.
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.
json	If the mimetype is *application/json* this will contain the parsed JSON data.
max_content_length	Read-only view of the `MAX_CONTENT_LENGTH` config key.
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.
module	The name of the current module if the request was dispatched to an actual module.
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.
routing_exception	If matching the URL failed, this is the exception that will be raised / was raised as part of the request handling.
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.
url_rule	The internal URL rule that matched the request.
user_agent	The current user agent.
values	Combined multi dict for `args` and `form`.
view_args	A dict of view arguments that matched the request.
want_form_data_parsed	Returns True if the request method carries content.

flask.Request.application

`Request.application`(*f*)

Decorate a function as responder that accepts the request as first argument. This works like the `responder()` decorator but the function is passed the request object as first argument and the request object will be closed automatically:

```
@Request.application
def my_wsgi_app(request):
    return Response('Hello World!')
```

Parameters:	**f** – the WSGI callable to decorate
Returns:	a new WSGI callable

flask.Request.from_values

`Request.from_values`(**args*, ***kwargs*)

Create a new request object based on the values provided. If environ is given missing values are filled from there. This method is useful for small scripts when you need to simulate a request from an URL. Do not use this method for unittesting, there is a full featured client object (`Client`) that allows to create multipart requests, support for cookies etc.

This accepts the same options as the `EnvironBuilder`.

Changed in version 0.5: This method now accepts the same arguments as `EnvironBuilder`. Because of this the *environ* parameter is now called *environ_overrides*.

Returns: request object

flask.Request.get_data

`Request.get_data`(*cache=True*, *as_text=False*, *parse_form_data=False*)

This reads the buffered incoming data from the client into one bytestring. By default this is cached but that behavior can be changed by setting *cache* to *False*.

Usually it’s a bad idea to call this method without checking the content length first as a client could send dozens of megabytes or more to cause memory problems on the server.

Note that if the form data was already parsed this method will not return anything as form data parsing does not cache the data like this method does. To implicitly invoke form data parsing function set *parse_form_data* to *True*. When this is done the return value of this method will be an empty string if the form parser handles the data. This generally is not necessary as if the whole data is cached (which is the default) the form parser will used the cached data to parse the form data. Please be generally aware of checking the content length first in any case before calling this method to avoid exhausting server memory.

If *as_text* is set to *True* the return value will be a decoded unicode string.

flask.Request.get_json

`Request.get_json`(*force=False*, *silent=False*, *cache=True*)[source]

Parses the incoming JSON request data and returns it. By default this function will return `None` if the mimetype is not *application/json* but this can be overridden by the `force` parameter. If parsing fails the `on_json_loading_failed()` method on the request object will be invoked.

Parameters:	• **force** – if set to `True` the mimetype is ignored. • **silent** – if set to `True` this method will fail silently and return `None`. • **cache** – if set to `True` the parsed JSON data is remembered on the request.

flask.Request.on_json_loading_failed

`Request.on_json_loading_failed`(*e*)[source]

Called if decoding of the JSON data failed. The return value of this method is used by `get_json()` when an error occurred. The default implementation just raises a `BadRequest` exception.

Changed in version 0.10: Removed buggy previous behavior of generating a random JSON response. If you want that behavior back you can trivially add it by subclassing.

flask.Request.access_route

`Request.access_route`

If a forwarded header exists this is a list of all ip addresses from the client ip to the last proxy server.

flask.Request.args

`Request.args`

The parsed URL parameters. By default an `ImmutableMultiDict` is returned from this function. This can be changed by setting `parameter_storage_class` to a different type. This might be necessary if the order of the form data is important.

flask.Request.files

`Request.files`

`MultiDict` object containing all uploaded files. Each key in `files` is the name from the `<input type="file" name="">`. Each value in `files` is a Werkzeug `FileStorage` object.

Note that `files` will only contain data if the request method was POST, PUT or PATCH and the `<form>` that posted to the request had `enctype="multipart/form-data"`. It will be empty otherwise.

See the `MultiDict` / `FileStorage` documentation for more details about the used data structure.

flask.Request.form

`Request.form`

The form parameters. By default an `ImmutableMultiDict` is returned from this function. This can be changed by setting `parameter_storage_class` to a different type. This might be necessary if the order of the form data is important.

`flask.Response`

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

The response object that is used by default in Flask. Works like the response object from Werkzeug but is set to have an HTML mimetype by default. Quite often you don't have to create this object yourself because `make_response()` will take care of that for you.

If you want to replace the response object used you can subclass this and set `response_class` to your subclass.

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, ma	Sets a cookie.

x_age, expires, ...])	
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.

flask.Response.__init__

`Response.__init__`(*response=None*, *status=None*, *headers=None*, *mimetype=None*, *content_type=None*, *direct_passthrough=False*)

flask.Response.call_on_close

`Response.call_on_close`(*func*)

Adds a function to the internal list of functions that should be called as part of closing down the response. Since 0.7 this function also returns the function that was passed so that this can be used as a decorator.

flask.Response.set_cookie

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

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.

flask.Response.delete_cookie

`Response.delete_cookie`(*key*, *path='/'*, *domain=None*)

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.

flask.Response.set_data

`Response.set_data`(*value*)

Sets a new string as response. The value set must either by a unicode or bytestring. If a unicode string is set it’s encoded automatically to the charset of the response (utf-8 by default).

flask.Response.force_type

`Response.force_type`(*response, environ=None*)

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.

flask.Response.freeze

`Response.freeze`()

Call this method if you want to make your response object ready for being pickled. This buffers the generator if there is one. It will also set the *Content-Length* header to the length of the body.

flask.Response.content_encoding

`Response.content_encoding`

The Content-Encoding entity-header field is used as a modifier to the media-type. When present, its value indicates what additional content codings have been applied to the entity-body, and thus what decoding mechanisms must be applied in order to obtain the media-type referenced by the Content-Type header field.

flask.Response.get_app_iter

`Response.get_app_iter`(*environ*)

Returns the application iterator for the given environ. Depending on the request method and the current status code the return value might be an empty response rather than the one from the response.

If the request method is *HEAD* or the status code is in a range where the HTTP specification requires an empty response, an empty iterable is returned.

New in version 0.6.

Parameters:	**environ** – the WSGI environment of the request.
Returns:	a response iterable.

flask.Response.get_wsgi_headers

`Response.get_wsgi_headers`(*environ*)

This is automatically called right before the response is started and returns headers modified for the given environment. It returns a copy of the headers from the response with some modifications applied if necessary.

For example the location header (if present) is joined with the root URL of the environment. Also the content length is automatically set to zero here for certain status codes.

Changed in version 0.6: Previously that function was called *fix_headers* and modified the response object in place. Also since 0.6, IRIs in location and content-location headers are handled properly.

Also starting with 0.6, Werkzeug will attempt to set the content length if it is able to figure it out on its own. This is the case if all the strings in the response iterable are already encoded and the iterable is buffered.

Parameters:	**environ** – the WSGI environment of the request.
Returns:	returns a new `Headers` object.

flask.Response.make_conditional

`Response.make_conditional`(*request_or_environ*)

Make the response conditional to the request. This method works best if an etag was defined for the response already. The *add_etag* method can be used to do that. If called without etag just the date header is set.

This does nothing if the request method in the request or environ is anything but GET or HEAD.

It does not remove the body of the response because that's something the `__call__()` function does for us automatically.

Returns self so that you can do `return resp.make_conditional(req)` but modifies the object in-place.

Parameters: **request_or_environ** – a request object or WSGI environment to be used to make the response conditional against.

flask.Session

`flask.Session`

alias of `SecureCookieSession`

Methods

`__init__`([initial])	
`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) 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`(*args, **kw)	
`setdefault`(key[, default])	
`update`(*args, **kw)	
`values`(() -> list of D's values)	
`viewitems`(...)	
`viewkeys`(...)	
`viewvalues`(...)	

Attributes

`modified`	
`new`	
`on_update`	
`permanent`	