web.py Documentation

Release 0.37

Anand Chitipothu

January 12, 2017

Contents:

CHAPTER 1

Accessing User Input

While building web applications, one basic and important thing is to respond to the user input that is sent to the server.

Web.py makes it easy to access that whether it is parameters in the url (*GET* request) or the form data (*POST* or *PUT* request). The *web.input()* method returns a dictionary-like object (more specifically a *web.storage* object) that contains the user input, whatever the request method is.

To access the URL parameters (?key=value) from the *web.input* object, just use *web.input().key*.

1.1 GET

For a URL which looks like */page?id=1&action=edit*, you do

```
class Page(object):
    def GET(self):
        data = web.input()
        id = int(data.id)    # all the inputs are now strings. Cast it to int, to get integer.
        action = data.action
        ...
```

KeyError exception is thrown if *key* is not there in the URL parameters. Web.py makes it easier to handle that with default values to web.input().

```
class Page(object):
    def GET(self):
        data = web.input(id=1, action='read')
        id, action = int(data.id), data.action
        ...
```

1.2 POST

It works exactly the same way with POST method. If you have a form with *name* and *password* elements, you would do

```
class Login(object):
    def POST(self):
        data = web.input()
        name, password = data.name, data.password
        ...
```

1.3 Multiple inputs with same name

What if you have a URL which looks like */page?id=1&id=2&id=3* or you have a form with multiple selects? What would *web.input().id* give us? It simply swallows all but one value. But to let web.input() know that we're expecting more values with the same name is simple. Just pass *[]* as the default argument for that name.

```
class Page(object):
    def GET(self):
        data = web.input(id=[])
        ids = data.id         # now, `ids` is a list with all the `id`s.
        ...
```

1.4 File uploads

Uploading files is easy with web.py. *web.input()* takes care of that too. Just make sure that the upload form has an attribute enctype="multipart/form-data". The *input()* gives you *filename* and *value*, which are the uploaded file name and the contents of it, respectively. To make things simpler, it also gives you *file*, a file-like object if you pass *myfile={}* where *myfile* is the name of the input element in your form.

```
class Upload(object):
    def GET(self):
        return render.upload()

    def POST(self):
         data = web.input(myfile={})
         fp = data.myfile
         save(fp)   # fp.filename, fp.read() gives name and contents of the file
         ...
```

or

```
class Upload(object):
    ...

    def POST(self):
         data = web.input() # notice that `myfile={}` is missing here.
         fp = data.myfile
         save(fp.filename, fp.value)
         ...
```

CHAPTER 2

Accessing the database

Web.py provides a simple and uniform interface to the database that you want to work with, whether it is PostgreSQL, MySQL, SQLite or any other. It doesn't try to build layers between you and your database. Rather, it tries to make it easy to perform common tasks, and get out of your way when you need to do more advanced things.

2.1 Create database object

The first thing to work with databases from web.py is to create a create a database object with *web.database()*. It returns database object, which has convenient methods for you to use.

Make sure that you have appropriate database library installed (*psycopg2* for PostgreSQL, *MySQLdb* for MySQL, *sqlite3* for SQLite).

```
db = web.database(dbn='postgres', db='dbname', user='username', pw='password')
```

dbn for MySQL is *mysql* and *sqlite* for SQLite. SQLite doesn't take *user pw* parameters.

2.1.1 Multiple databases

Working with more databases is not at all difficult with web.py. Here's what you do.

```
db1 = web.database(dbn='postgres', db='dbname1', user='username1', pw='password2')
db2 = web.database(dbn='postgres', db='dbname2', user='username2', pw='password2')
```

And use *db1*, *db2* to access those databases respectively.

2.2 Operations

web.database() returns an object which provide you all the functionality to insert, select, update and delete data from your database. For each of the methods on *db* below, you can pass *_test=True* to see the SQL statement rather than executing it.

2.2.1 Inserting

```
# Insert an entry into table 'user'
userid = db.insert('user', firstname="Bob", lastname="Smith", joindate=web.SQLLiteral("NOW()"))
```

The first argument is the table name and the rest of them are set of named arguments which represent the fields in the table. If values are not given, the database may create default values or issue a warning.

For bulk insertion rather than inserting record by record, use *Multiple Inserts* rather.

2.2.2 Selecting

The *select* method is used for selecting rows from the database. It returns a *web.iterbetter* object, which can be looped through.

To select *all* the rows from the *user* table, you would simply do

```
users = db.select('user')
```

For the real world use cases, *select* method takes *vars*, *what*, *where*, *order*, *group*, *limit*, *offset*, and *_test* optional parameters.

```
users = db.select('users', where="id>100")
```

To prevent SQL injection attacks, you can use *$key* in where clause and pass the *vars* which has { 'key': value }.

```
vars = dict(name="Bob")
results = db.select('users', where="name = $name", vars=vars, _test=True)
>>> results
<sql: "SELECT * FROM users WHERE name = 'Bob'">
```

2.2.3 Updating

The *update* method accepts same kind of arguments as Select. It returns the number of rows updated.

```
num_updated  = db.update('users', where="id = 10", firstname = "Foo")
```

2.2.4 Deleting

The *delete* method returns the number of rows deleted. It also accepts "using" and "vars" parameters. See `Selecting` for more details on *vars*.

```
num_deleted = db.delete('users', where="id=10")
```

2.2.5 Multiple Inserts

The *multiple_insert* method on the *db* object allows you to do that. All that's needed is to prepare a list of dictionaries, one for each row to be inserted, each with the same set of keys and pass it to *multiple_insert* along with the table name. It returns the list of ids of the inserted rows.

The value of *db.supports_multiple_insert* tells you if your database supports multiple inserts.

```
values = [{"name": "foo", "email": "foo@example.com"}, {"name": "bar", "email": "bar@example.com"}]
db.multiple_insert('person', values=values)
```

2.2.6 Advanced querying

Many a times, there is more to do with the database, rather than the simple operations which can be done by *insert*, *select*, *delete* and *update* - Things like your favorite (or scary) joins, counts etc. All these are possible with *query* method, which also takes *vars*.

```
results = db.query("SELECT COUNT(*) AS total_users FROM users")
print results[0].total_users # prints number of entries in 'users' table
```

Joining tables

```
results = db.query("SELECT * FROM entries JOIN users WHERE entries.author_id = users.id")
```

2.2.7 Transactions

The database object has a method *transaction* which starts a new transaction and returns the transaction object. The transaction object can be used to commit or rollback that transaction. It is also possible to have nested transactions.

From Python 2.5 onwards, which support *with* statements, you would do

```
with db.transaction():
    userid = db.insert('users', name='foo')
    authorid = db.insert('authors', userid=userid)
```

For earlier versions of Python, you can do

```
t = db.transaction()
try:
    userid = db.insert('users', name='foo')
    authorid = db.insert('authors', userid=userid)
except:
    t.rollback()
    raise
else:
    t.commit()
```

CHAPTER 3

Templating

There are almost as many Python templating systems as there are web frameworks (and, indeed, it seems like many templating systems are adopting web framework-like features). The following are the goals of *templetor*, which is the (codename of) templating system of web.py.

1. The templating system has to *look* decent. No `<%#foo#%>` crud.
2. Reuse Python terms and semantics as much as possible.
3. Expressive enough to do real computation.
4. Usable for any text language, not just HTML and XML.

And requirements for the implementation as well:

4. Sandboxable so that you can let untrusted users write templates.
5. Simple and fast implementation.

So here it is.

3.1 Variable substitution

```
Look, a $string.
Hark, an ${arbitrary + expression}.
Gawk, a $dictionary[key].function('argument').
Cool, a $(limit)ing.

Stop, \$money isn't evaluated.
```

We use basically the same semantics as (rejected) PEP 215. Variables can go anywhere in a document.

3.2 Newline suppression

```
If you put a backslash \
at the end of a line \
(like these) \
then there will be no newline.
```

renders as all one line.

3.3 Expressions

```
Here are some expressions:

$for var in iterator: I like $var!

$if times > max:
    Stop! In the name of love.
$else:
    Keep on, you can do it.

That's all, folks.
```

All your old Python friends are here: `if`, `while`, `for`, `else`, `break`, `continue`, and `pass` also act as you'd expect. (Obviously, you can't have variables named any of these.) The Python code starts at the `$` and ends at the `:`. The `$` has to be at the beginning of the line, but that's not such a burden because of newline suppression (above).

Also, we're very careful about spacing – all the lines will render with no spaces at the beginning. (Open question: what if you want spaces at the beginning?) Also, a trailing space might break your code.

There are a couple changes from Python: `for` and `while` now take an `else` clause that gets called if the loop is never evaluated.

(Possible feature to add: Django-style for loop variables.)

3.4 Comments

```
$# Here's where we hoodwink the folks at home:

Please enter in your deets:

CC: [       ]  $#this is the important one
SSN: $#Social Security Number#$ [       ]
```

Comments start with `$#` and go to `#$` or the end of the line, whichever is first.

3.5 Code

NOTE: This feature has not been implemented in the current web.py implementation of templetor.

```
Sometimes you just need to break out the Python.

$ mapping = {
$   'cool': ['nice', 'sweet', 'hot'],
$   'suck': ['bad', 'evil', 'awful']
$ }

Isn't that $mapping[thought]?
That's$ del mapping $ fine with me.

$ complicatedfunc()

$ for x in bugs:
    $ if bug.level == 'severe':
        Ooh, this one is bad.
        $ continue
    And there's $x...
```

Body of loops have to be indented with exactly 4 spaces.

Code begins with a `$` and a space and goes until the next `$` or the end of the line, whichever comes first. Nothing ever gets output if the first character after the `$` is a space (so `complicatedfunc` above doesn't write anything to the screen like it might without the space).

3.6 Python integration

A template begins with a line like this:

```
$def with (name, title, company='BigCo')
```

which declares that the template takes those arguments. (The `with` keyword is special, like `def` or `if`.)

Don't forget to put spaces in the definition

The following *will not work*:

```
$def with (name,title,company='BigCo')
```

Inside Python, the template looks like a function that takes these arguments. It returns a storage object with the special property that evaluating it as a string returns the value of the body of the template. The elements in the storage object are the results of the `def`s and the `set`s.

Perhaps an example will make this clearer. Here's a template, "entry":

```
$def with (post)

$var title: $post.title

<p>$markdown(post.body)</p>

<p class="byline">by $post.author</p>
```

Here's another; "base":

```
$def with (self)
<html><head>
  <title>$self.title</title>
</head><body>
<h1>$self.title</h1>

$:self
</body></html>
```

Now let's say we compile both from within Python, the first as `entry`, the second as `base`. Here's how we might use them:

```
print base( entry( post ) )
```

`entry` takes the argument post and returns an object whose string value is a bit of HTML showing the post with its title in the property `title`. `base` takes this object and places the title in the appropriate place and displays the page itself in the body of the page. The Python code prints out the result.

Where did ‘‘markdown‘‘ come from? It wasn’t passed as an argument. You can pass a list of functions and variables to the template compiler to be made globally available to templates. *Why $:self?* See below

Here’s an example:

```
import template
render = template.render('templates/')
template.Template.globals['len'] = len

print render.base(render.message('Hello, world!'))
```

The first line imports templetor. The second says that our templates are in the directory `templates/`. The third give all our templates access to the `len` function. The fourth grabs the template `message.html`, passes it the argument `'Hello, world!'`, passes the result of rendering it to mcitp the template `base.html` and prints the result. (If your templates don’t end in `.html` or `.xml`, templetor will still find them, but it won’t do its automatic HTML-encoding.)

3.7 Turning Off Filter

By default `template.render` will use `web.websafe` filter to do HTML-encoding. To turn it off, put a : after the $ as in:

```
$:form.render()
```

Output from form.render() will be displayed as is.

```
$:fooBar    $# fooBar = <span>lorem ipsum</span>
```

Output from variable in template will be displayed as is.

3.8 Including / nesting templates

If you want to nest one template within another, you nest the `render()` calls, and then include the variable (unfiltered) in the page. In your handler:

```
print render.foo(render.bar())
```

or (to make things a little more clear):

```
barhtml = render.bar()
print render.foo(barhtml)
```

Then in the template `foo.html`:

```
$def with (bar)
html goes here
$:bar
more html
```

This replaces the `$:bar` with the output of the `render.bar()` call (which is why it must be `$:`/unfiltered, so that you get un-encoded HTML (unless you want something else of course)). You can pass variables in, in the same way:

```
print render.foo(render.bar(baz), qux)
```

In the template bar (`bar.html`):

```
$def with (baz)
bar stuff goes here + baz
```

In template foo (`foo.html`):

```
$def with (bar, qux)
html goes here
$:bar
Value of qux is $qux
```

3.9 Escaping

web.py automatically escapes any variables used in templates, so that if for some reason name is set to a value containing some HTML, it will get properly escaped and appear as plain text. If you want to turn this off, write $:name instead of $name.

CHAPTER 4

Deploying web.py applications

4.1 FastCGI

web.py uses flup library for supporting fastcgi. Make sure it is installed.

You just need to make sure you applicaiton file is executable. Make it so by adding the following line to tell the system to execute it using python:

```
#! /usr/bin/env python
```

and setting the exeutable flag on the file:

```
chmod +x /path/to/yourapp.py
```

4.1.1 Configuring lighttpd

Here is a sample lighttpd configuration file to expose a web.py app using fastcgi.

```
# Enable mod_fastcgi and mod_rewrite modules
server.modules   += ( "mod_fastcgi" )
server.modules   += ( "mod_rewrite" )

# configure the application
fastcgi.server = ( "/yourapp.py" =>
    ((
        # path to the socket file
        "socket" => "/tmp/yourapp-fastcgi.socket",

        # path to the application
        "bin-path" => "/path/to/yourapp.py",

        # number of fastcgi processes to start
        "max-procs" => 1,

        "bin-environment" => (
            "REAL_SCRIPT_NAME" => ""
        ),
        "check-local" => "disable"
    ))
)

 url.rewrite-once = (
    # favicon is usually placed in static/
    "^/favicon.ico$" => "/static/favicon.ico",

    # Let lighttpd serve resources from /static/.
    # The web.py dev server automatically servers /static/, but this is
    # required when deploying in production.
    "^/static/(.*)$" => "/static/$1",

    # everything else should go to the application, which is already configured above.
    "^/(.*)$" => "/yourapp.py/$1",
 )
```

With this configuration lighttpd takes care of starting the application. The webserver talks to your application using fastcgi via a unix domain socket. This means both the webserver and the application will run on the same machine.

4.2 nginx + Gunicorn

Gunicorn 'Green Unicorn' is a Python WSGI HTTP Server for UNIX. It's a pre-fork worker model ported from Ruby's Unicorn project.

To make a web.py application work with gunicorn, you'll need to get the wsgi app from web.py application object.

```
import web
...
app = web.application(urls, globals())

# get the wsgi app from web.py application object
wsgiapp = app.wsgifunc()
```

Once that change is made, gunicorn server be started using:

```
gunicorn -w 4 -b 127.0.0.1:4000 yourapp:wsgiapp
```

This starts gunicorn with 4 workers and listens at port 4000 on localhost.

It is best to use Gunicorn behind HTTP proxy server. The gunicorn team strongly advises to use nginx. Here is a sample nginx configuration which proxies to application running on *127.0.0.1:4000*.

```
server {
  listen 80;
  server_name example.org;
  access_log  /var/log/nginx/example.log;

  location / {
      proxy_pass http://127.0.0.1:4000;

      proxy_set_header Host $host;
      proxy_set_header X-Real-IP $remote_addr;
      proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
  }
}
```

CHAPTER 5

web.py API

5.1 web.application

Web application (from web.py)

class `web.application.`**`application`**(*mapping=(), fvars={}, autoreload=None*)
Application to delegate requests based on path.

```
>>> urls = ("/hello", "hello")
>>> app = application(urls, globals())
>>> class hello:
...     def GET(self): return "hello"
>>>
>>> app.request("/hello").data
b'hello'
```

`add_processor`(*processor*)
Adds a processor to the application.

```
>>> urls = ("/(.*)", "echo")
>>> app = application(urls, globals())
>>> class echo:
...     def GET(self, name): return name
...
>>>
>>> def hello(handler): return "hello, " +  handler()
...
>>> app.add_processor(hello)
>>> app.request("/web.py").data
b'hello, web.py'
```

`cgirun`(**middleware*)
Return a CGI handler. This is mostly useful with Google App Engine. There you can just do:

main = app.cgirun()

`gaerun`(**middleware*)
Starts the program in a way that will work with Google app engine, no matter which version you are using (2.5 / 2.7)

If it is 2.5, just normally start it with app.gaerun()

If it is 2.7, make sure to change the app.yaml handler to point to the global variable that contains the result of app.gaerun()

For example:

in app.yaml (where code.py is where the main code is located)

handlers: - url: /.*

script: code.app

Make sure that the app variable is globally accessible

internalerror()

Returns HTTPError with '500 internal error' message

load(*env*)

Initializes ctx using env.

notfound()

Returns HTTPError with '404 not found' message

request(*localpart='/'*, *method='GET'*, *data=None*, *host='0.0.0.0:8080'*, *headers=None*, *https=False*, ***kw*)

Makes request to this application for the specified path and method. Response will be a storage object with data, status and headers.

```
>>> urls = ("/hello", "hello")
>>> app = application(urls, globals())
>>> class hello:
...     def GET(self):
...         web.header('Content-Type', 'text/plain')
...         return "hello"
...
>>> response = app.request("/hello")
>>> response.data
b'hello'
>>> response.status
'200 OK'
>>> response.headers['Content-Type']
'text/plain'
```

To use https, use https=True.

```
>>> urls = ("/redirect", "redirect")
>>> app = application(urls, globals())
>>> class redirect:
...     def GET(self): raise web.seeother("/foo")
...
>>> response = app.request("/redirect")
>>> response.headers['Location']
'http://0.0.0.0:8080/foo'
>>> response = app.request("/redirect", https=True)
>>> response.headers['Location']
'https://0.0.0.0:8080/foo'
```

The headers argument specifies HTTP headers as a mapping object such as a dict.

```
>>> urls = ('/ua', 'uaprinter')
>>> class uaprinter:
...     def GET(self):
...         return 'your user-agent is ' + web.ctx.env['HTTP_USER_AGENT']
...
>>> app = application(urls, globals())
>>> app.request('/ua', headers = {
...       'User-Agent': 'a small jumping bean/1.0 (compatible)'
... }).data
b'your user-agent is a small jumping bean/1.0 (compatible)'
```

run (**middleware*)
: Starts handling requests. If called in a CGI or FastCGI context, it will follow that protocol. If called from the command line, it will start an HTTP server on the port named in the first command line argument, or, if there is no argument, on port 8080.

  middleware is a list of WSGI middleware which is applied to the resulting WSGI function.

stop ()
: Stops the http server started by run.

wsgifunc (**middleware*)
: Returns a WSGI-compatible function for this application.

class web.application.**auto_application**
: Application similar to *application* but urls are constructed automatically using metaclass.

```
>>> app = auto_application()
>>> class hello(app.page):
...     def GET(self): return "hello, world"
...
>>> class foo(app.page):
...     path = '/foo/.*'
...     def GET(self): return "foo"
>>> app.request("/hello").data
b'hello, world'
>>> app.request('/foo/bar').data
b'foo'
```

web.application.**subdir_application**
: alias of `application`

class web.application.**subdomain_application** (*mapping=()*, *fvars={}*, *autoreload=None*)
: Application to delegate requests based on the host.

```
>>> urls = ("/hello", "hello")
>>> app = application(urls, globals())
>>> class hello:
...     def GET(self): return "hello"
>>>
>>> mapping = (r"hello\.example\.com", app)
>>> app2 = subdomain_application(mapping)
>>> app2.request("/hello", host="hello.example.com").data
b'hello'
>>> response = app2.request("/hello", host="something.example.com")
>>> response.status
'404 Not Found'
>>> response.data
b'not found'
```

web.application.**loadhook** (*h*)
: Converts a load hook into an application processor.

```
>>> app = auto_application()
>>> def f(): "something done before handling request"
...
>>> app.add_processor(loadhook(f))
```

`web.application.`**`unloadhook`**`(`*h*`)`

Converts an unload hook into an application processor.

```
>>> app = auto_application()
>>> def f(): "something done after handling request"
...
>>> app.add_processor(unloadhook(f))
```

`web.application.`**`autodelegate`**`(`*prefix=''*`)`

Returns a method that takes one argument and calls the method named prefix+arg, calling *notfound()* if there isn't one. Example:

urls = ('/prefs/(.*)', 'prefs')

class prefs: GET = autodelegate('**GET_**') def GET_password(self): pass def GET_privacy(self):
pass

GET_password would get called for */prefs/password* while *GET_privacy* for *GET_privacy* gets called for */prefs/privacy*.

If a user visits */prefs/password/change* then *GET_password(self, '/change')* is called.

5.2 web.db

Database API (part of web.py)

exception `web.db.`**`UnknownParamstyle`**

raised for unsupported db paramstyles

(currently supported: qmark, numeric, format, pyformat)

exception `web.db.`**`UnknownDB`**

raised for unsupported dbms

`web.db.`**`sqllist`**`(`*lst*`)`

Converts the arguments for use in something like a WHERE clause.

```
>>> sqllist(['a', 'b'])
'a, b'
>>> sqllist('a')
'a'
```

`web.db.`**`sqlors`**`(`*left, lst*`)`

left is a SQL clause like 'tablename.arg = ' and 'lst is a list of values. Returns a reparam-style pair featuring the SQL that ORs together the clause for each item in the lst.

```
>>> sqlors('foo = ', [])
<sql: '1=2'>
>>> sqlors('foo = ', [1])
<sql: 'foo = 1'>
>>> sqlors('foo = ', 1)
<sql: 'foo = 1'>
>>> sqlors('foo = ', [1,2,3])
<sql: '(foo = 1 OR foo = 2 OR foo = 3 OR 1=2)'>
```

`web.db.`**`reparam`**`(`*string_, dictionary*`)`

Takes a string and a dictionary and interpolates the string using values from the dictionary. Returns an *SQLQuery* for the result.

```
>>> reparam("s = $s", dict(s=True))
<sql: "s = 't'">
>>> reparam("s IN $s", dict(s=[1, 2]))
<sql: 's IN (1, 2)'>
```

`web.db.`**`sqlquote`**`(`*a*`)`

Ensures *a* is quoted properly for use in a SQL query.

```
>>> 'WHERE x = ' + sqlquote(True) + ' AND y = ' + sqlquote(3)
<sql: "WHERE x = 't' AND y = 3">
>>> 'WHERE x = ' + sqlquote(True) + ' AND y IN ' + sqlquote([2, 3])
<sql: "WHERE x = 't' AND y IN (2, 3)">
```

class `web.db.`**`SQLQuery`**`(`*items=None*`)`

You can pass this sort of thing as a clause in any db function. Otherwise, you can pass a dictionary to the keyword argument *vars* and the function will call reparam for you.

Internally, consists of *items*, which is a list of strings and SQLParams, which get concatenated to produce the actual query.

static **`join`**`(`*items*, *sep=' '*, *prefix=None*, *suffix=None*, *target=None*`)`

Joins multiple queries.

```
>>> SQLQuery.join(['a', 'b'], ', ')
<sql: 'a, b'>
```

Optinally, prefix and suffix arguments can be provided.

```
>>> SQLQuery.join(['a', 'b'], ', ', prefix='(', suffix=')')
<sql: '(a, b)'>
```

If target argument is provided, the items are appended to target instead of creating a new SQLQuery.

`query``(`*paramstyle=None*`)`

Returns the query part of the sql query.

```
>>> q = SQLQuery(["SELECT * FROM test WHERE name=", SQLParam('joe')])
>>> q.query()
'SELECT * FROM test WHERE name=%s'
>>> q.query(paramstyle='qmark')
'SELECT * FROM test WHERE name=?'
```

`values``()`

Returns the values of the parameters used in the sql query.

```
>>> q = SQLQuery(["SELECT * FROM test WHERE name=", SQLParam('joe')])
>>> q.values()
['joe']
```

class `web.db.`**`SQLParam`**`(`*value*`)`

Parameter in SQLQuery.

```
>>> q = SQLQuery(["SELECT * FROM test WHERE name=", SQLParam("joe")])
>>> q
<sql: "SELECT * FROM test WHERE name='joe'">
>>> q.query()
'SELECT * FROM test WHERE name=%s'
```

```
>>> q.values()
['joe']
```

web.db.**sqlparam**
alias of *SQLParam*

class web.db.**SQLLiteral**(*v*)
Protects a string from *sqlquote*.

```
>>> sqlquote('NOW()')
<sql: "'NOW()'">
>>> sqlquote(SQLLiteral('NOW()'))
<sql: 'NOW()'>
```

web.db.**sqlliteral**
alias of *SQLLiteral*

web.db.**database**(*dburl=None*, ***params*)
Creates appropriate database using params.

Pooling will be enabled if DBUtils module is available. Pooling can be disabled by passing pooling=False in params.

class web.db.**DB**(*db_module*, *keywords*)
Database

delete(*table*, *where*, *using=None*, *vars=None*, *_test=False*)
Deletes from *table* with clauses *where* and *using*.

```
>>> db = DB(None, {})
>>> name = 'Joe'
>>> db.delete('foo', where='name = $name', vars=locals(), _test=True)
<sql: "DELETE FROM foo WHERE name = 'Joe'">
```

insert(*tablename*, *seqname=None*, *_test=False*, ***values*)
Inserts *values* into *tablename*. Returns current sequence ID. Set *seqname* to the ID if it's not the default, or to *False* if there isn't one.

```
>>> db = DB(None, {})
>>> q = db.insert('foo', name='bob', age=2, created=SQLLiteral('NOW()'), _test=True)
>>> q
<sql: "INSERT INTO foo (age, created, name) VALUES (2, NOW(), 'bob')">
>>> q.query()
'INSERT INTO foo (age, created, name) VALUES (%s, NOW(), %s)'
>>> q.values()
[2, 'bob']
```

multiple_insert(*tablename*, *values*, *seqname=None*, *_test=False*)
Inserts multiple rows into *tablename*. The *values* must be a list of dictioanries, one for each row to be inserted, each with the same set of keys. Returns the list of ids of the inserted rows. Set *seqname* to the ID if it's not the default, or to *False* if there isn't one.

```
>>> db = DB(None, {})
>>> db.supports_multiple_insert = True
>>> values = [{"name": "foo", "email": "foo@example.com"}, {"name": "bar", "email": "bar@exa
>>> db.multiple_insert('person', values=values, _test=True)
<sql: "INSERT INTO person (email, name) VALUES ('foo@example.com', 'foo'), ('bar@example.com
```

query(*sql_query*, *vars=None*, *processed=False*, *_test=False*)

Execute SQL query *sql_query* using dictionary *vars* to interpolate it. If *processed=True*, *vars* is a *reparam*-style list to use instead of interpolating.

```
>>> db = DB(None, {})
>>> db.query("SELECT * FROM foo", _test=True)
<sql: 'SELECT * FROM foo'>
>>> db.query("SELECT * FROM foo WHERE x = $x", vars=dict(x='f'), _test=True)
<sql: "SELECT * FROM foo WHERE x = 'f'">
>>> db.query("SELECT * FROM foo WHERE x = " + sqlquote('f'), _test=True)
<sql: "SELECT * FROM foo WHERE x = 'f'">
```

select (*tables*, *vars=None*, *what='*'*, *where=None*, *order=None*, *group=None*, *limit=None*, *offset=None*, *_test=False*)

Selects *what* from *tables* with clauses *where*, *order*, *group*, *limit*, and *offset*. Uses vars to interpolate. Otherwise, each clause can be a SQLQuery.

```
>>> db = DB(None, {})
>>> db.select('foo', _test=True)
<sql: 'SELECT * FROM foo'>
>>> db.select(['foo', 'bar'], where="foo.bar_id = bar.id", limit=5, _test=True)
<sql: 'SELECT * FROM foo, bar WHERE foo.bar_id = bar.id LIMIT 5'>
>>> db.select('foo', where={'id': 5}, _test=True)
<sql: 'SELECT * FROM foo WHERE id = 5'>
```

transaction ()

Start a transaction.

update (*tables*, *where*, *vars=None*, *_test=False*, ***values*)

Update *tables* with clause *where* (interpolated using *vars*) and setting *values*.

```
>>> db = DB(None, {})
>>> name = 'Joseph'
>>> q = db.update('foo', where='name = $name', name='bob', age=2,
...     created=SQLLiteral('NOW()'), vars=locals(), _test=True)
>>> q
<sql: "UPDATE foo SET age = 2, created = NOW(), name = 'bob' WHERE name = 'Joseph'">
>>> q.query()
'UPDATE foo SET age = %s, created = NOW(), name = %s WHERE name = %s'
>>> q.values()
[2, 'bob', 'Joseph']
```

where (*table*, *what='*'*, *order=None*, *group=None*, *limit=None*, *offset=None*, *_test=False*, ***kwargs*)

Selects from *table* where keys are equal to values in *kwargs*.

```
>>> db = DB(None, {})
>>> db.where('foo', bar_id=3, _test=True)
<sql: 'SELECT * FROM foo WHERE bar_id = 3'>
>>> db.where('foo', source=2, crust='dewey', _test=True)
<sql: "SELECT * FROM foo WHERE crust = 'dewey' AND source = 2">
>>> db.where('foo', _test=True)
<sql: 'SELECT * FROM foo'>
```

5.3 web.net

Network Utilities (from web.py)

`web.net.`**validipaddr** (*address*)

Returns True if *address* is a valid IPv4 address.

```
>>> validipaddr('192.168.1.1')
True
>>> validipaddr('192.168.1.800')
False
>>> validipaddr('192.168.1')
False
```

web.net.**validip6addr**(*address*)

Returns True if *address* is a valid IPv6 address.

```
>>> validip6addr('::')
True
>>> validip6addr('aaaa:bbbb:cccc:dddd::1')
True
>>> validip6addr('1:2:3:4:5:6:7:8:9:10')
False
>>> validip6addr('12:10')
False
```

web.net.**validipport**(*port*)

Returns True if *port* is a valid IPv4 port.

```
>>> validipport('9000')
True
>>> validipport('foo')
False
>>> validipport('1000000')
False
```

web.net.**validip**(*ip*, *defaultaddr='0.0.0.0'*, *defaultport=8080*)

Returns *(ip_address, port)* from string *ip_addr_port* >>> validip('1.2.3.4') ('1.2.3.4', 8080) >>> validip('80') ('0.0.0.0', 80) >>> validip('192.168.0.1:85') ('192.168.0.1', 85) >>> validip('::') ('::', 8080) >>> validip('[::]:88') ('::', 88) >>> validip('[::1]:80') ('::1', 80)

web.net.**validaddr**(*string_*)

Returns either (ip_address, port) or "/path/to/socket" from **string_**

```
>>> validaddr('/path/to/socket')
'/path/to/socket'
>>> validaddr('8000')
('0.0.0.0', 8000)
>>> validaddr('127.0.0.1')
('127.0.0.1', 8080)
>>> validaddr('127.0.0.1:8000')
('127.0.0.1', 8000)
>>> validip('[::1]:80')
('::1', 80)
>>> validaddr('fff')
Traceback (most recent call last):
    ...
ValueError: fff is not a valid IP address/port
```

web.net.**urlquote**(*val*)

Quotes a string for use in a URL.

```
>>> urlquote('://?f=1&j=1')
'%3A//%3Ff%3D1%26j%3D1'
>>> urlquote(None)
''
```

```
>>> urlquote(u'\u203d')
'%E2%80%BD'
```

web.net.**httpdate**(*date_obj*)
Formats a datetime object for use in HTTP headers.

```
>>> import datetime
>>> httpdate(datetime.datetime(1970, 1, 1, 1, 1, 1))
'Thu, 01 Jan 1970 01:01:01 GMT'
```

web.net.**parsehttpdate**(*string_*)
Parses an HTTP date into a datetime object.

```
>>> parsehttpdate('Thu, 01 Jan 1970 01:01:01 GMT')
datetime.datetime(1970, 1, 1, 1, 1, 1)
```

web.net.**htmlquote**(*text*)
Encodes *text* for raw use in HTML.

```
>>> htmlquote(u"<'&\">")
u'&lt;&#39;&amp;&quot;&gt;'
```

web.net.**htmlunquote**(*text*)
Decodes *text* that's HTML quoted.

```
>>> htmlunquote(u'&lt;&#39;&amp;&quot;&gt;')
u'<\'&">'
```

web.net.**websafe**(*val*)
Converts *val* so that it is safe for use in Unicode HTML.

```
>>> websafe("<'&\">")
u'&lt;&#39;&amp;&quot;&gt;'
>>> websafe(None)
u''
>>> websafe(u'\u203d') == u'\u203d'
True
```

5.4 web.form

HTML forms (part of web.py)

class web.form.**AttributeList**
List of atributes of input.

```
>>> a = AttributeList(type='text', name='x', value=20)
>>> a
<attrs: 'name="x" type="text" value="20"'>
```

class web.form.**Button**(*name*, **validators*, ***attrs*)
HTML Button.

```
>>> Button("save").render()
u'<button id="save" name="save">save</button>'
>>> Button("action", value="save", html="<b>Save Changes</b>").render()
u'<button id="action" name="action" value="save"><b>Save Changes</b></button>'
```

class `web.form.`**`Checkbox`**(*name*, **validators*, ***attrs*)
Checkbox input.

```
>>> Checkbox('foo', value='bar', checked=True).render()
u'<input checked="checked" id="foo_bar" name="foo" type="checkbox" value="bar"/>'
>>> Checkbox('foo', value='bar').render()
u'<input id="foo_bar" name="foo" type="checkbox" value="bar"/>'
>>> c = Checkbox('foo', value='bar')
>>> c.validate('on')
True
>>> c.render()
u'<input checked="checked" id="foo_bar" name="foo" type="checkbox" value="bar"/>'
```

class `web.form.`**`Dropdown`**(*name*, *args*, **validators*, ***attrs*)
Dropdown/select input.

```
>>> Dropdown(name='foo', args=['a', 'b', 'c'], value='b').render()
u'<select id="foo" name="foo">\n  <option value="a">a</option>\n  <option selected="selected" va
>>> Dropdown(name='foo', args=[('a', 'aa'), ('b', 'bb'), ('c', 'cc')], value='b').render()
u'<select id="foo" name="foo">\n  <option value="a">aa</option>\n  <option selected="selected" v
```

class `web.form.`**`File`**(*name*, **validators*, ***attrs*)
File input.

```
>>> File(name='f').render()
u'<input id="f" name="f" type="file"/>'
```

class `web.form.`**`Form`**(**inputs*, ***kw*)
HTML form.

```
>>> f = Form(Textbox("x"))
>>> f.render()
u'<table>\n    <tr><th><label for="x">x</label></th><td><input id="x" name="x" type="text"/></td
>>> f.fill(x="42")
True
>>> f.render()
u'<table>\n    <tr><th><label for="x">x</label></th><td><input id="x" name="x" type="text" value
```

class `web.form.`**`GroupedDropdown`**(*name*, *args*, **validators*, ***attrs*)
Grouped Dropdown/select input.

```
>>> GroupedDropdown(name='car_type', args=(('Swedish Cars', ('Volvo', 'Saab')), ('German Cars',
u'<select id="car_type" name="car_type">\n  <optgroup label="Swedish Cars">\n    <option value="
>>> GroupedDropdown(name='car_type', args=(('Swedish Cars', (('v', 'Volvo'), ('s', 'Saab'))), ('
u'<select id="car_type" name="car_type">\n  <optgroup label="Swedish Cars">\n    <option value="
```

class `web.form.`**`Hidden`**(*name*, **validators*, ***attrs*)
Hidden Input.

```
>>> Hidden(name='foo', value='bar').render()
u'<input id="foo" name="foo" type="hidden" value="bar"/>'
```

class `web.form.`**`Password`**(*name*, **validators*, ***attrs*)
Password input.

```
>>> Password(name='password', value='secret').render()
u'<input id="password" name="password" type="password" value="secret"/>'
```

class `web.form.`**`Textarea`**(*name*, **validators*, ***attrs*)
Textarea input.

```
>>> Textarea(name='foo', value='bar').render()
u'<textarea id="foo" name="foo">bar</textarea>'
```

class `web.form.`**`Textbox`**(*name*, **validators*, ***attrs*)
Textbox input.

```
>>> Textbox(name='foo', value='bar').render()
u'<input id="foo" name="foo" type="text" value="bar"/>'
>>> Textbox(name='foo', value=0).render()
u'<input id="foo" name="foo" type="text" value="0"/>'
```

5.5 web.http

HTTP Utilities (from web.py)

`web.http.`**`expires`**(*delta*)
Outputs an *Expires* header for *delta* from now. *delta* is a *timedelta* object or a number of seconds.

`web.http.`**`lastmodified`**(*date_obj*)
Outputs a *Last-Modified* header for *datetime*.

`web.http.`**`prefixurl`**(*base=''*)
Sorry, this function is really difficult to explain. Maybe some other time.

`web.http.`**`modified`**(*date=None*, *etag=None*)
Checks to see if the page has been modified since the version in the requester's cache.

When you publish pages, you can include *Last-Modified* and *ETag* with the date the page was last modified and an opaque token for the particular version, respectively. When readers reload the page, the browser sends along the modification date and etag value for the version it has in its cache. If the page hasn't changed, the server can just return *304 Not Modified* and not have to send the whole page again.

This function takes the last-modified date *date* and the ETag *etag* and checks the headers to see if they match. If they do, it returns *True*, or otherwise it raises NotModified error. It also sets *Last-Modified* and *ETag* output headers.

`web.http.`**`changequery`**(*query=None*, ***kw*)
Imagine you're at */foo?a=1&b=2*. Then *changequery(a=3)* will return */foo?a=3&b=2* – the same URL but with the arguments you requested changed.

`web.http.`**`url`**(*path=None*, *doseq=False*, ***kw*)
Makes url by concatenating web.ctx.homepath and path and the query string created using the arguments.

`web.http.`**`profiler`**(*app*)
Outputs basic profiling information at the bottom of each response.

5.6 web.session

Session Management (from web.py)

class `web.session.`**`Session`**(*app*, *store*, *initializer=None*)
Session management for web.py

`expired`()
Called when an expired session is atime

kill()
Kill the session, make it no longer available

class web.session.**Store**
Base class for session stores

cleanup(*timeout*)
removes all the expired sessions

decode(*session_data*)
decodes the data to get back the session dict

encode(*session_dict*)
encodes session dict as a string

class web.session.**DiskStore**(*root*)
Store for saving a session on disk.

```
>>> import tempfile
>>> root = tempfile.mkdtemp()
>>> s = DiskStore(root)
>>> s['a'] = 'foo'
>>> s['a']
'foo'
>>> time.sleep(0.01)
>>> s.cleanup(0.01)
>>> s['a']
Traceback (most recent call last):
    ...
KeyError: 'a'
```

class web.session.**DBStore**(*db*, *table_name*)
Store for saving a session in database Needs a table with the following columns:

session_id CHAR(128) UNIQUE NOT NULL, atime DATETIME NOT NULL default current_timestamp, data TEXT

5.7 web.template

simple, elegant templating (part of web.py)

Template design:

Template string is split into tokens and the tokens are combined into nodes. Parse tree is a nodelist. TextNode and ExpressionNode are simple nodes and for-loop, if-loop etc are block nodes, which contain multiple child nodes.

Each node can emit some python string. python string emitted by the root node is validated for safeeval and executed using python in the given environment.

Enough care is taken to make sure the generated code and the template has line to line match, so that the error messages can point to exact line number in template. (It doesn't work in some cases still.)

Grammar:

template -> defwith sections defwith -> '$def with (' arguments ')' | '' sections -> section* section -> block | assignment | line

assignment -> '$ ' <assignment expression> line -> (text|expr)* text -> <any characters other than $> expr -> '$' pyexpr | '$(' pyexpr ')' | '${' pyexpr '}' pyexpr -> <python expression>

class `web.template.`**`Render`**(*loc='templates'*, *cache=None*, *base=None*, ***keywords*)

The most preferred way of using templates.

render = web.template.render('templates') print render.foo()

Optional parameter can be *base* can be used to pass output of every template through the base template.

render = web.template.render('templates', base='layout')

`web.template.`**`render`**

alias of `Render`

`web.template.`**`frender`**(*path*, ***keywords*)

Creates a template from the given file path.

exception `web.template.`**`SecurityError`**

The template seems to be trying to do something naughty.

`web.template.`**`test`**()

Doctest for testing template module.

Define a utility function to run template test.

```
>>> class TestResult:
...     def __init__(self, t): self.t = t
...     def __getattr__(self, name): return getattr(self.t, name)
...     def __repr__(self): return repr(unicode(self.t) if PY2 else str(self.t))
...
>>> def t(code, **keywords):
...     tmpl = Template(code, **keywords)
...     return lambda *a, **kw: TestResult(tmpl(*a, **kw))
...
```

Simple tests.

```
>>> t('1')()
u'1\n'
>>> t('$def with ()\n1')()
u'1\n'
>>> t('$def with (a)\n$a')(1)
u'1\n'
>>> t('$def with (a=0)\n$a')(1)
u'1\n'
>>> t('$def with (a=0)\n$a')(a=1)
u'1\n'
```

Test complicated expressions.

```
>>> t('$def with (x)\n$x.upper()')('hello')
u'HELLO\n'
>>> t('$(2 * 3 + 4 * 5)')()
u'26\n'
>>> t('${2 * 3 + 4 * 5}')()
u'26\n'
>>> t('$def with (limit)\nkeep $(limit)ing.')('go')
u'keep going.\n'
>>> t('$def with (a)\n$a.b[0]')(storage(b=[1]))
u'1\n'
```

Test html escaping.

```
>>> t('$def with (x)\n$x', filename='a.html')('<html>')
u'&lt;html&gt;\n'
>>> t('$def with (x)\n$x', filename='a.txt')('<html>')
u'<html>\n'
```

Test if, for and while.

```
>>> t('$if 1: 1')()
u'1\n'
>>> t('$if 1:\n    1')()
u'1\n'
>>> t('$if 1:\n    1\\')()
u'1'
>>> t('$if 0: 0\n$elif 1: 1')()
u'1\n'
>>> t('$if 0: 0\n$elif None: 0\n$else: 1')()
u'1\n'
>>> t('$if 0 < 1 and 1 < 2: 1')()
u'1\n'
>>> t('$for x in [1, 2, 3]: $x')()
u'1\n2\n3\n'
>>> t('$def with (d)\n$for k, v in d.items(): $k')({1: 1})
u'1\n'
>>> t('$for x in [1, 2, 3]:\n\t$x')()
u'    1\n    2\n    3\n'
>>> t('$def with (a)\n$while a and a.pop():1')([1, 2, 3])
u'1\n1\n1\n'
```

The space after : must be ignored.

```
>>> t('$if True: foo')()
u'foo\n'
```

Test loop.xxx.

```
>>> t("$for i in range(5):$loop.index, $loop.parity")()
u'1, odd\n2, even\n3, odd\n4, even\n5, odd\n'
>>> t("$for i in range(2):\n    $for j in range(2):$loop.parent.parity $loop.parity")()
u'odd odd\nodd even\neven odd\neven even\n'
```

Test assignment.

```
>>> t('$ a = 1\n$a')()
u'1\n'
>>> t('$ a = [1]\n$a[0]')()
u'1\n'
>>> t('$ a = {1: 1}\n$list(a.keys())[0]')()
u'1\n'
>>> t('$ a = []\n$if not a: 1')()
u'1\n'
>>> t('$ a = {}\n$if not a: 1')()
u'1\n'
>>> t('$ a = -1\n$a')()
u'-1\n'
>>> t('$ a = "1"\n$a')()
u'1\n'
```

Test comments.

```
>>> t('$# 0')()
u'\n'
>>> t('hello$#comment1\nhello$#comment2')()
u'hello\nhello\n'
>>> t('$#comment0\nhello$#comment1\nhello$#comment2')()
u'\nhello\nhello\n'
```

Test unicode.

```
>>> t('$def with (a)\n$a')(u'\u203d')
u'\u203d\n'
>>> t(u'$def with (a)\n$a $:a')(u'\u203d')
u'\u203d \u203d\n'
>>> t(u'$def with ()\nfoo')()
u'foo\n'
>>> def f(x): return x
...
>>> t(u'$def with (f)\n$:f("x")')(f)
u'x\n'
>>> t('$def with (f)\n$:f("x")')(f)
u'x\n'
```

Test dollar escaping.

```
>>> t("Stop, $$money isn't evaluated.")()
u"Stop, $money isn't evaluated.\n"
>>> t("Stop, \$money isn't evaluated.")()
u"Stop, $money isn't evaluated.\n"
```

Test space sensitivity.

```
>>> t('$def with (x)\n$x')(1)
u'1\n'
>>> t('$def with(x ,y)\n$x')(1, 1)
u'1\n'
>>> t('$(1 + 2*3 + 4)')()
u'11\n'
```

Make sure globals are working.

```
>>> t('$x')()
Traceback (most recent call last):
    ...
NameError: global name 'x' is not defined
>>> t('$x', globals={'x': 1})()
u'1\n'
```

Can't change globals.

```
>>> t('$ x = 2\n$x', globals={'x': 1})()
u'2\n'
>>> t('$ x = x + 1\n$x', globals={'x': 1})()
Traceback (most recent call last):
    ...
UnboundLocalError: local variable 'x' referenced before assignment
```

Make sure builtins are customizable.

```
>>> t('$min(1, 2)')()
u'1\n'
```

```
>>> t('$min(1, 2)', builtins={})()
Traceback (most recent call last):
    ...
NameError: global name 'min' is not defined
```

Test vars.

```
>>> x = t('$var x: 1')()
>>> x.x
u'1'
>>> x = t('$var x = 1')()
>>> x.x
1
>>> x = t('$var x:  \n    foo\n    bar')()
>>> x.x
u'foo\nbar\n'
```

Test BOM chars.

```
>>> t('\xef\xbb\xbf$def with(x)\n$x')('foo')
u'foo\n'
```

Test for with weird cases.

```
>>> t('$for i in range(10)[1:5]:\n    $i')()
u'1\n2\n3\n4\n'
>>> t("$for k, v in sorted({'a': 1, 'b': 2}.items()):\n    $k $v", globals={'sorted':sorted})()
u'a 1\nb 2\n'
```

Test for syntax error.

```
>>> try:
...     t("$for k, v in ({'a': 1, 'b': 2}.items():\n    $k $v")()
... except SyntaxError:
...     print("OK")
... else:
...     print("Expected SyntaxError")
...
OK
```

Test datetime.

```
>>> import datetime
>>> t("$def with (date)\n$date.strftime('%m %Y')")(datetime.datetime(2009, 1, 1))
u'01 2009\n'
```

5.8 web.utils

General Utilities (part of web.py)

class `web.utils.`**`Storage`**

A Storage object is like a dictionary except *obj.foo* can be used in addition to *obj['foo']*.

```
>>> o = storage(a=1)
>>> o.a
1
>>> o['a']
1
```

```
>>> o.a = 2
>>> o['a']
2
>>> del o.a
>>> o.a
Traceback (most recent call last):
    ...
AttributeError: 'a'
```

web.utils.**storage**
alias of *Storage*

web.utils.**storify**(*mapping*, **requireds*, ***defaults*)
Creates a *storage* object from dictionary *mapping*, raising *KeyError* if d doesn't have all of the keys in *requireds* and using the default values for keys found in *defaults*.

For example, *storify({'a':1, 'c':3}, b=2, c=0)* will return the equivalent of *storage({'a':1, 'b':2, 'c':3})*.

If a *storify* value is a list (e.g. multiple values in a form submission), *storify* returns the last element of the list, unless the key appears in *defaults* as a list. Thus:

```
>>> storify({'a':[1, 2]}).a
2
>>> storify({'a':[1, 2]}, a=[]).a
[1, 2]
>>> storify({'a':1}, a=[]).a
[1]
>>> storify({}, a=[]).a
[]
```

Similarly, if the value has a *value* attribute, *storify will return _its_ value, unless the key appears in 'defaults* as a dictionary.

```
>>> storify({'a':storage(value=1)}).a
1
>>> storify({'a':storage(value=1)}, a={}).a
<Storage {'value': 1}>
>>> storify({}, a={}).a
{}
```

class web.utils.**Counter**
Keeps count of how many times something is added.

```
>>> c = counter()
>>> c.add('x')
>>> c.add('x')
>>> c.add('x')
>>> c.add('x')
>>> c.add('x')
>>> c.add('y')
>>> c['y']
1
>>> c['x']
5
>>> c.most()
['x']
```

least()
Returns the keys with mininum count.

most()
Returns the keys with maximum count.

percent(*key*)
Returns what percentage a certain key is of all entries.

```
>>> c = counter()
>>> c.add('x')
>>> c.add('x')
>>> c.add('x')
>>> c.add('y')
>>> c.percent('x')
0.75
>>> c.percent('y')
0.25
```

sorted_items()
Returns items sorted by value.

```
>>> c = counter()
>>> c.add('x')
>>> c.add('x')
>>> c.add('y')
>>> c.sorted_items()
[('x', 2), ('y', 1)]
```

sorted_keys()
Returns keys sorted by value.

```
>>> c = counter()
>>> c.add('x')
>>> c.add('x')
>>> c.add('y')
>>> c.sorted_keys()
['x', 'y']
```

sorted_values()
Returns values sorted by value.

```
>>> c = counter()
>>> c.add('x')
>>> c.add('x')
>>> c.add('y')
>>> c.sorted_values()
[2, 1]
```

web.utils.**counter**
alias of *Counter*

web.utils.**rstrips**(*text*, *remove*)
removes the string *remove* from the right of *text*

```
>>> rstrips("foobar", "bar")
'foo'
```

web.utils.**lstrips**(*text*, *remove*)
removes the string *remove* from the left of *text*

```
>>> lstrips("foobar", "foo")
'bar'
>>> lstrips('http://foo.org/', ['http://', 'https://'])
```

```
'foo.org/'
>>> lstrips('FOOBARBAZ', ['FOO', 'BAR'])
'BAZ'
>>> lstrips('FOOBARBAZ', ['BAR', 'FOO'])
'BARBAZ'
```

web.utils.**strips**(*text*, *remove*)
removes the string *remove* from the both sides of *text*

```
>>> strips("foobarfoo", "foo")
'bar'
```

web.utils.**safeunicode**(*obj*, *encoding='utf-8'*)
Converts any given object to unicode string.

```
>>> safeunicode('hello')
u'hello'
>>> safeunicode(2)
u'2'
>>> safeunicode('\xe1\x88\xb4')
u'\u1234'
```

web.utils.**safestr**(*obj*, *encoding='utf-8'*)
Converts any given object to utf-8 encoded string.

```
>>> safestr('hello')
'hello'
>>> safestr(2)
'2'
```

web.utils.**timelimit**(*timeout*)
A decorator to limit a function to *timeout* seconds, raising *TimeoutError* if it takes longer.

```
>>> import time
>>> def meaningoflife():
...     time.sleep(.2)
...     return 42
>>>
>>> timelimit(.1)(meaningoflife)()
Traceback (most recent call last):
    ...
RuntimeError: took too long
>>> timelimit(1)(meaningoflife)()
42
```

Caveat: The function isn't stopped after *timeout* seconds but continues executing in a separate thread. (There seems to be no way to kill a thread.)

inspired by <http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/473878>

class web.utils.**Memoize**(*func*, *expires=None*, *background=True*)
'Memoizes' a function, caching its return values for each input. If *expires* is specified, values are recalculated after *expires* seconds. If *background* is specified, values are recalculated in a separate thread.

```
>>> calls = 0
>>> def howmanytimeshaveibeencalled():
...     global calls
...     calls += 1
...     return calls
>>> fastcalls = memoize(howmanytimeshaveibeencalled)
>>> howmanytimeshaveibeencalled()
1
>>> howmanytimeshaveibeencalled()
2
>>> fastcalls()
3
>>> fastcalls()
3
>>> import time
>>> fastcalls = memoize(howmanytimeshaveibeencalled, .1, background=False)
>>> fastcalls()
4
>>> fastcalls()
4
>>> time.sleep(.2)
>>> fastcalls()
5
>>> def slowfunc():
...     time.sleep(.1)
...     return howmanytimeshaveibeencalled()
>>> fastcalls = memoize(slowfunc, .2, background=True)
>>> fastcalls()
6
>>> timelimit(.05)(fastcalls)()
6
>>> time.sleep(.2)
>>> timelimit(.05)(fastcalls)()
6
>>> timelimit(.05)(fastcalls)()
6
>>> time.sleep(.2)
>>> timelimit(.05)(fastcalls)()
7
>>> fastcalls = memoize(slowfunc, None, background=True)
>>> threading.Thread(target=fastcalls).start()
>>> time.sleep(.01)
>>> fastcalls()
9
```

web.utils.**memoize**
alias of *Memoize*

web.utils.**re_subm**(*pat*, *repl*, *string*)
Like re.sub, but returns the replacement _and_ the match object.

```
>>> t, m = re_subm('g(oo+)fball', r'f\1lish', 'gooooooofball')
>>> t
'foooooolish'
>>> m.groups()
('oooooo',)
```

web.utils.**group**(*seq*, *size*)
Returns an iterator over a series of lists of length size from iterable.

```
>>> list(group([1,2,3,4], 2))
[[1, 2], [3, 4]]
>>> list(group([1,2,3,4,5], 2))
[[1, 2], [3, 4], [5]]
```

`web.utils.`**`uniq`**`(`*seq*, *key=None*`)`

Removes duplicate elements from a list while preserving the order of the rest.

```
>>> uniq([9,0,2,1,0])
[9, 0, 2, 1]
```

The value of the optional *key* parameter should be a function that takes a single argument and returns a key to test the uniqueness.

```
>>> uniq(["Foo", "foo", "bar"], key=lambda s: s.lower())
['Foo', 'bar']
```

`web.utils.`**`iterview`**`(`*x*`)`

Takes an iterable *x* and returns an iterator over it which prints its progress to stderr as it iterates through.

class `web.utils.`**`IterBetter`**`(`*iterator*`)`

Returns an object that can be used as an iterator but can also be used via __getitem__ (although it cannot go backwards – that is, you cannot request *iterbetter[0]* after requesting *iterbetter[1]*).

```
>>> import itertools
>>> c = iterbetter(itertools.count())
>>> c[1]
1
>>> c[5]
5
>>> c[3]
Traceback (most recent call last):
    ...
IndexError: already passed 3
```

It is also possible to get the first value of the iterator or None.

```
>>> c = iterbetter(iter([3, 4, 5]))
>>> print(c.first())
3
>>> c = iterbetter(iter([]))
>>> print(c.first())
None
```

For boolean test, IterBetter peeps at first value in the itertor without effecting the iteration.

```
>>> c = iterbetter(iter(range(5)))
>>> bool(c)
True
>>> list(c)
[0, 1, 2, 3, 4]
>>> c = iterbetter(iter([]))
>>> bool(c)
False
>>> list(c)
[]
```

`first``(`*default=None*`)`

Returns the first element of the iterator or None when there are no elements.

If the optional argument default is specified, that is returned instead of None when there are no elements.

`web.utils.`**`iterbetter`**

alias of *`IterBetter`*

`web.utils.`**`safeiter`**(*it*, *cleanup=None*, *ignore_errors=True*)
Makes an iterator safe by ignoring the exceptions occured during the iteration.

`web.utils.`**`safewrite`**(*filename*, *content*)
Writes the content to a temp file and then moves the temp file to given filename to avoid overwriting the existing file in case of errors.

`web.utils.`**`dictreverse`**(*mapping*)
Returns a new dictionary with keys and values swapped.

```
>>> dictreverse({1: 2, 3: 4})
{2: 1, 4: 3}
```

`web.utils.`**`dictfind`**(*dictionary*, *element*)
Returns a key whose value in *dictionary* is *element* or, if none exists, None.

```
>>> d = {1:2, 3:4}
>>> dictfind(d, 4)
3
>>> dictfind(d, 5)
```

`web.utils.`**`dictfindall`**(*dictionary*, *element*)
Returns the keys whose values in *dictionary* are *element* or, if none exists, [].

```
>>> d = {1:4, 3:4}
>>> dictfindall(d, 4)
[1, 3]
>>> dictfindall(d, 5)
[]
```

`web.utils.`**`dictincr`**(*dictionary*, *element*)
Increments *element* in *dictionary*, setting it to one if it doesn't exist.

```
>>> d = {1:2, 3:4}
>>> dictincr(d, 1)
3
>>> d[1]
3
>>> dictincr(d, 5)
1
>>> d[5]
1
```

`web.utils.`**`dictadd`**(**dicts*)
Returns a dictionary consisting of the keys in the argument dictionaries. If they share a key, the value from the last argument is used.

```
>>> dictadd({1: 0, 2: 0}, {2: 1, 3: 1})
{1: 0, 2: 1, 3: 1}
```

`web.utils.`**`requeue`**(*queue*, *index=-1*)
Returns the element at index after moving it to the beginning of the queue.

```
>>> x = [1, 2, 3, 4]
>>> requeue(x)
4
>>> x
[4, 1, 2, 3]
```

`web.utils.`**`restack`**(*stack*, *index=0*)
Returns the element at index after moving it to the top of stack.

```
>>> x = [1, 2, 3, 4]
>>> restack(x)
1
>>> x
[2, 3, 4, 1]
```

`web.utils.`**`listget`**(*lst*, *ind*, *default=None*)
Returns *lst[ind]* if it exists, *default* otherwise.

```
>>> listget(['a'], 0)
'a'
>>> listget(['a'], 1)
>>> listget(['a'], 1, 'b')
'b'
```

`web.utils.`**`intget`**(*integer*, *default=None*)
Returns *integer* as an int or *default* if it can’t.

```
>>> intget('3')
3
>>> intget('3a')
>>> intget('3a', 0)
0
```

`web.utils.`**`datestr`**(*then*, *now=None*)
Converts a (UTC) datetime object to a nice string representation.

```
>>> from datetime import datetime, timedelta
>>> d = datetime(1970, 5, 1)
>>> datestr(d, now=d)
'0 microseconds ago'
>>> for t, v in iteritems({
...   timedelta(microseconds=1): '1 microsecond ago',
...   timedelta(microseconds=2): '2 microseconds ago',
...   -timedelta(microseconds=1): '1 microsecond from now',
...   -timedelta(microseconds=2): '2 microseconds from now',
...   timedelta(microseconds=2000): '2 milliseconds ago',
...   timedelta(seconds=2): '2 seconds ago',
...   timedelta(seconds=2*60): '2 minutes ago',
...   timedelta(seconds=2*60*60): '2 hours ago',
...   timedelta(days=2): '2 days ago',
... }):
...     assert datestr(d, now=d+t) == v
>>> datestr(datetime(1970, 1, 1), now=d)
'January  1'
>>> datestr(datetime(1969, 1, 1), now=d)
'January  1, 1969'
>>> datestr(datetime(1970, 6, 1), now=d)
'June  1, 1970'
>>> datestr(None)
''
```

`web.utils.`**`numify`**(*string*)
Removes all non-digit characters from *string*.

```
>>> numify('800-555-1212')
'8005551212'
>>> numify('800.555.1212')
'8005551212'
```

`web.utils.`**`denumify`**`(`*string*, *pattern*`)`

Formats *string* according to *pattern*, where the letter X gets replaced by characters from *string*.

```
>>> denumify("8005551212", "(XXX) XXX-XXXX")
'(800) 555-1212'
```

`web.utils.`**`commify`**`(`*n*`)`

Add commas to an integer *n*.

```
>>> commify(1)
'1'
>>> commify(123)
'123'
>>> commify(1234)
'1,234'
>>> commify(1234567890)
'1,234,567,890'
>>> commify(123.0)
'123.0'
>>> commify(1234.5)
'1,234.5'
>>> commify(1234.56789)
'1,234.56789'
>>> commify('%.2f' % 1234.5)
'1,234.50'
>>> commify(None)
>>>
```

`web.utils.`**`dateify`**`(`*datestring*`)`

Formats a numified *datestring* properly.

`web.utils.`**`nthstr`**`(`*n*`)`

Formats an ordinal. Doesn't handle negative numbers.

```
>>> nthstr(1)
'1st'
>>> nthstr(0)
'0th'
>>> [nthstr(x) for x in [2, 3, 4, 5, 10, 11, 12, 13, 14, 15]]
['2nd', '3rd', '4th', '5th', '10th', '11th', '12th', '13th', '14th', '15th']
>>> [nthstr(x) for x in [91, 92, 93, 94, 99, 100, 101, 102]]
['91st', '92nd', '93rd', '94th', '99th', '100th', '101st', '102nd']
>>> [nthstr(x) for x in [111, 112, 113, 114, 115]]
['111th', '112th', '113th', '114th', '115th']
```

`web.utils.`**`cond`**`(`*predicate*, *consequence*, *alternative=None*`)`

Function replacement for if-else to use in expressions.

```
>>> x = 2
>>> cond(x % 2 == 0, "even", "odd")
'even'
>>> cond(x % 2 == 0, "even", "odd") + '_row'
'even_row'
```

class `web.utils.`**`CaptureStdout`**`(`*func*`)`

Captures everything *func* prints to stdout and returns it instead.

```
>>> def idiot():
...     print("foo")
```

```
>>> capturestdout(idiot)()
'foo\n'
```

WARNING: Not threadsafe!

`web.utils.`**`capturestdout`**
alias of `CaptureStdout`

class `web.utils.`**`Profile`**(*func*)
Profiles *func* and returns a tuple containing its output and a string with human-readable profiling information.

```
>>> import time
>>> out, inf = profile(time.sleep)(.001)
>>> out
>>> inf[:10].strip()
'took 0.0'
```

`web.utils.`**`profile`**
alias of `Profile`

`web.utils.`**`tryall`**(*context*, *prefix=None*)
Tries a series of functions and prints their results. *context* is a dictionary mapping names to values; the value will only be tried if it's callable.

```
>>> tryall(dict(j=lambda: True))
j: True
----------------------------------------
results:
   True: 1
```

For example, you might have a file *test/stuff.py* with a series of functions testing various things in it. At the bottom, have a line:

> if __name__ == "__main__": tryall(globals())

Then you can run *python test/stuff.py* and get the results of all the tests.

class `web.utils.`**`ThreadedDict`**
Thread local storage.

```
>>> d = ThreadedDict()
>>> d.x = 1
>>> d.x
1
>>> import threading
>>> def f(): d.x = 2
...
>>> t = threading.Thread(target=f)
>>> t.start()
>>> t.join()
>>> d.x
1
```

static **`clear_all`**()
Clears all ThreadedDict instances.

`web.utils.`**`threadeddict`**
alias of `ThreadedDict`

`web.utils.`**`autoassign`**(*self*, *locals*)
Automatically assigns local variables to *self*.

```
>>> self = storage()
>>> autoassign(self, dict(a=1, b=2))
>>> self.a
1
>>> self.b
2
```

Generally used in *__init__* methods, as in:

def __init__(self, foo, bar, baz=1): autoassign(self, locals())

`web.utils.`**`to36`**`(`*q*`)`

Converts an integer to base 36 (a useful scheme for human-sayable IDs).

```
>>> to36(35)
'z'
>>> to36(119292)
'2k1o'
>>> int(to36(939387374), 36)
939387374
>>> to36(0)
'0'
>>> to36(-393)
Traceback (most recent call last):
    ...
ValueError: must supply a positive integer
```

`web.utils.`**`safemarkdown`**`(`*text*`)`

Converts text to HTML following the rules of Markdown, but blocking any outside HTML input, so that only the things supported by Markdown can be used. Also converts raw URLs to links.

(requires [markdown.py](http://webpy.org/markdown.py))

`web.utils.`**`sendmail`**`(`*from_address*, *to_address*, *subject*, *message*, *headers=None*, ***kw*`)`

Sends the email message *message* with mail and envelope headers for from *from_address_* to *to_address* with *subject*. Additional email headers can be specified with the dictionary ‘headers.

Optionally cc, bcc and attachments can be specified as keyword arguments. Attachments must be an iterable and each attachment can be either a filename or a file object or a dictionary with filename, content and optionally content_type keys.

If *web.config.smtp_server* is set, it will send the message to that SMTP server. Otherwise it will look for */usr/sbin/sendmail*, the typical location for the sendmail-style binary. To use sendmail from a different path, set *web.config.sendmail_path*.

5.9 web.webapi

Web API (wrapper around WSGI) (from web.py)

`web.webapi.`**`header`**`(`*hdr*, *value*, *unique=False*`)`

Adds the header *hdr: value* with the response.

If *unique* is True and a header with that name already exists, it doesn’t add a new one.

`web.webapi.`**`debug`**`(`**args*`)`

Prints a prettyprinted version of *args* to stderr.

`web.webapi.`**`input`**`(`**requireds*, ***defaults*`)`

Returns a *storage* object with the GET and POST arguments. See *storify* for how *requireds* and *defaults* work.

`web.webapi.`**`data`**`()`
Returns the data sent with the request.

`web.webapi.`**`setcookie`**`(`*name*, *value*, *expires=''*, *domain=None*, *secure=False*, *httponly=False*, *path=None*`)`
Sets a cookie.

`web.webapi.`**`cookies`**`(`**requireds*, ***defaults*`)`
Returns a *storage* object with all the request cookies in it.

See *storify* for how *requireds* and *defaults* work.

This is forgiving on bad HTTP_COOKIE input, it tries to parse at least the cookies it can.

The values are converted to unicode if _unicode=True is passed.

exception `web.webapi.`**`OK`**`(`*data=''*, *headers={}*`)`
200 OK status

exception `web.webapi.`**`Created`**`(`*data='Created'*, *headers={}*`)`
201 Created status

exception `web.webapi.`**`Accepted`**`(`*data='Accepted'*, *headers={}*`)`
202 Accepted status

exception `web.webapi.`**`NoContent`**`(`*data='No Content'*, *headers={}*`)`
204 No Content status

`web.webapi.`**`ok`**
alias of `OK`

`web.webapi.`**`created`**
alias of `Created`

`web.webapi.`**`accepted`**
alias of `Accepted`

`web.webapi.`**`nocontent`**
alias of `NoContent`

exception `web.webapi.`**`Redirect`**`(`*url*, *status='301 Moved Permanently'*, *absolute=False*`)`
A *301 Moved Permanently* redirect.

exception `web.webapi.`**`Found`**`(`*url*, *absolute=False*`)`
A *302 Found* redirect.

exception `web.webapi.`**`SeeOther`**`(`*url*, *absolute=False*`)`
A *303 See Other* redirect.

exception `web.webapi.`**`NotModified`**
A *304 Not Modified* status.

exception `web.webapi.`**`TempRedirect`**`(`*url*, *absolute=False*`)`
A *307 Temporary Redirect* redirect.

`web.webapi.`**`redirect`**
alias of `Redirect`

`web.webapi.`**`found`**
alias of `Found`

`web.webapi.`**`seeother`**
alias of `SeeOther`

`web.webapi.`**`notmodified`**
  alias of *`NotModified`*

`web.webapi.`**`tempredirect`**
  alias of *`TempRedirect`*

exception `web.webapi.`**`BadRequest`**`(`*message=None*`)`
  400 Bad Request error.

exception `web.webapi.`**`Unauthorized`**`(`*message=None*`)`
  401 Unauthorized error.

exception `web.webapi.`**`Forbidden`**`(`*message=None*`)`
  403 Forbidden error.

`web.webapi.`**`NotFound`**`(`*message=None*`)`
  Returns HTTPError with '404 Not Found' error from the active application.

exception `web.webapi.`**`NoMethod`**`(`*cls=None*`)`
  A *405 Method Not Allowed* error.

exception `web.webapi.`**`NotAcceptable`**`(`*message=None*`)`
  406 Not Acceptable error.

exception `web.webapi.`**`Conflict`**`(`*message=None*`)`
  409 Conflict error.

exception `web.webapi.`**`Gone`**`(`*message=None*`)`
  410 Gone error.

exception `web.webapi.`**`PreconditionFailed`**`(`*message=None*`)`
  412 Precondition Failed error.

exception `web.webapi.`**`UnsupportedMediaType`**`(`*message=None*`)`
  415 Unsupported Media Type error.

`web.webapi.`**`UnavailableForLegalReasons`**`(`*message=None*`)`
  Returns HTTPError with '415 Unavailable For Legal Reasons' error from the active application.

`web.webapi.`**`badrequest`**
  alias of *`BadRequest`*

`web.webapi.`**`unauthorized`**
  alias of *`Unauthorized`*

`web.webapi.`**`forbidden`**
  alias of *`Forbidden`*

`web.webapi.`**`notfound`**`(`*message=None*`)`
  Returns HTTPError with '404 Not Found' error from the active application.

`web.webapi.`**`nomethod`**
  alias of *`NoMethod`*

`web.webapi.`**`notacceptable`**
  alias of *`NotAcceptable`*

`web.webapi.`**`conflict`**
  alias of *`Conflict`*

`web.webapi.`**`gone`**
  alias of *`Gone`*

`web.webapi.`**`preconditionfailed`**
alias of *`PreconditionFailed`*

`web.webapi.`**`unsupportedmediatype`**
alias of *`UnsupportedMediaType`*

`web.webapi.`**`unavailableforlegalreasons`**(*message=None*)
Returns HTTPError with '415 Unavailable For Legal Reasons' error from the active application.

`web.webapi.`**`InternalError`**(*message=None*)
Returns HTTPError with '500 internal error' error from the active application.

`web.webapi.`**`internalerror`**(*message=None*)
Returns HTTPError with '500 internal error' error from the active application.

Getting Started

Building webapps with web.py is easy. To get started, save the following code as say, *hello.py* and run it with *python hello.py*. Now point your browser to *http://localhost:8080/* which responds you with 'Hello, world!'. Hey, you're done with your first program with with web.py - with just 8 lines of code!

```
import web

urls = ("/.*", "hello")
app = web.application(urls, globals())

class hello:
    def GET(self):
        return 'Hello, world!'

if __name__ == "__main__":
    app.run()
```

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Index