Module Reference

James Gardner

April 10, 2005

http://www.pythonweb.org
docs at pythonweb.org

Abstract

This manual provides a detailed reference of each of the modules which make up the Python Web Modules.

For an overview of the modules, their purpose and licence see *Overview of the Python Web Modules*

Warning: This version of the modules has undergone many changes and should be considered a development release, likely to contain some bugs.

CONTENTS

1 Web Modules **1**

1.1 `web` — Web modules . . . 1
1.2 `web.auth` — Easy to use authorisation, authentication and user management system . . . 2
1.3 `datetime` — Compatibility code providing date and time classes for Python 2.2 users . . . 19
1.4 `web.database` — SQL database layer . . . 22
1.5 `web.database.object` — An object relation mapper built on the `web.database` and `web.form` modules . . . 73
1.6 `web.error` — Enhanced error handling based on the `cgitb` module . . . 99
1.7 `web.environment` — Tools for seting up an environment . . . 107
1.8 `web.form` — Construction of persistant forms/wizards for HTML interfaces . . . 109
1.9 `web.image` — Create and manipulate graphics including JPG, PNG, PDF, PS using `PIL` . . . 120
1.10 `web.mail` — Simple function to send email using `email` . . . 122
1.11 `web.session` — Persistent storage of sessions and automatic cookie handling . . . 124
1.12 `web.template` — For the easy display of data as HTML/XML . . . 136
1.13 `web.util` — Useful utility functions that don't fit elsewhere . . . 141
1.14 `web.wsgi` — Web Server Gateway Interface tools . . . 145

A Reporting Bugs **167**

B History and License **169**

B.1 History of the software . . . 169

Module Index **171**

Index **173**

CHAPTER

ONE

Web Modules

The Web Modules are a series of useful librares for building web applications without the need to learn a framework.

1.1 `web` — Web modules

The `web` module provides some basic utility functions and objects which are used throughout the Web Modules.

Version Information

The `web` module has the following variables:

`web.version_info` A tuple similar to `sys.version_info` in the form `(major version, minor version, revision, release candidate, status)`

`web.version` The version as a string eg `'0.4.0rc1'`

`web.name` The name of the modules as a string

`web.date` The date of the release as a string in the format `'yyyy-mm-dd'`.

`web.status` The release status of the code. For example `'beta'`

Useful Objects

The `web` module provides the following objects:

`web.cgi` An object based on the cgi.FieldStorage() object.

The `web.cgi` object is used to access CGI environment variables such as information submitted from forms or appended to a URL or information about the user's browser. `web.cgi` provides a dictionary-like interface to all the sumbitted CGI variables.

Warning: Creating a `cgi.FieldStorage` object can destroy data that would be used in subsequent creating subsequent `cgi.FieldStorage` objects so you should only use the `web.cgi` object which will be created first in order to avoid this problem.

See Also:

`cgi` Module Documentation
(http://www.python.org/doc/current/lib/module-cgi.html)

The `cgi` module documentation distributed with Python has more information about `cgi.FieldStorage` objects and a full functional specification.

Useful Functions

header([*type='text/html'*])
Returns a Content-type HTTP header

typeThe content-type to output

encode(*html*[, *mode='url'*])
Encode a string for use in an HTML document

htmlThe string to encode

modeIf *mode* is `'url'` the *html* string is encoded for use in a URL. If *mode* is `'form'` *html* is encoded for use in a form field.

Warning: The HTTP protocol doesn't specify the maximum length of URLs but to be absolutely safe try not to let them be longer than 256 characters. Internet Explorer supports URLs of up to 2,083 characters. Any long strings are better off encoded to be put as hidden values in a form with `method="POST"` rather than encoded and embedded in URLs. Information sent using `POST` is sent in the HTTP header where there is no limit to the length.

Another reason not to encode larege amounts if information in URLs is that doing so may also result in strange behaviour in certain browsers.

1.2 `web.auth` — Easy to use authorisation, authentication and user management system

The `web.auth` module provides methods for allowing multiple users in different groups, multiple access levels to multiple applications with multiple roles and activity status using a single login. It offers a poweful, flexible and simple way to restrict or manage access to different parts of your code and is suitable for enterprise use.

1.2.1 Background Information

An auth system has to perform two main tasks:

Authentication Finding out if the user is who he claims to be

Authorisation Checking the authenticated user has sufficient access priviledges to perform the task

In order to authenticate a user a username and password is usually entered and if the password matches the username the user is authenticated. This process involves several steps:

- Displaying a sign in form
- Checking the username and password
- Displaying the form again if the details are not correct
- Storing the information that the user is signed in

Before this can happen a mechanism is required to be able to add users to the system and set their access priveledges and a mechainsm is required to store information about the signed in user so that they remain signed in.

The `web.auth` module provides the following classes to achieve all these tasks in a simple and yet flexible way.

admin objects (`AuthAdmin` class) This class is used to administer the auth environment, add users, set access levels etc.

session objects (`AuthSession` class) This class is used to store information about who is signed in, when they signed in and when thry should be signed out

manager objects (`AuthManager` class) Used to manage the auth functions, has all the functionality of the admin and session objects

user objects (`AuthUser` class) Contain all the information about a particular user and can be used to set simple user properties

handler objects Are used to automatically handle tasks such as user sign in

1.2.2 Creating a basic auth environment

The `web.auth` module is designed so that the data can be stored in ways using different drivers. Currently only a database storage driver exists allowing auth information to be stored in any relational database supported by the `web.database` module. The `web.database` module includes SnakeSQL, a pure Python database which works like a library, so you can use the `web.auth` module even if you do not have access to another relational database engine.

In this example we are using a database to store the auth and session information so we setup a database cursor named `cursor` as described in the documentation for the `web.database` module.

```
import web.database
connection = web.database.connect(adapter='snakesql', database='test', autoCreate=1)
cursor = connection.cursor()
```

Next we need to create the necessary tables which will store information about the users and their access rights. To do this we need an admin object:

```
admin = web.auth.admin(driver='database', cursor=cursor)
```

If we haven't already created the auth tables we can do so like this:

```
if not admin.completeAuthEnvironment():
    admin.removeAuthEnvironment(ignoreErrors=True)
    admin.createAuthEnvironment()
connection.commit()
```

If any of the tables are missing, this code removes all existing tables thereby destroying all the data they contain (ignoring errors produced because of missing tables) and re-creates all the tables. The `connection.commit()` saves the changes to the database.

Adding Applications

To add an application to the auth environment you use the `addApplication()` method of the admin object created above. To add an application named `app` we would use the following:

```
admin.addApp('app')
```

Likewise you can check if applications exist using `admin.appExists(`*app*`)` and remove applications using `admin.removeApp(`*app*`)`. You can get a list of applications using `admin.apps()`

Adding Users

To add a user you use the `admin.addUser()` method of the admin object.

The `addUser()` method takes the parameters *username*, *password*, *firstname*, *surname*, *email*, *active* and *group*. Only *username* and *password* are required.

Using the admin object created in the example above:

```
admin.addUser(username='john', password='bananas')
```

There are a also number of methods to manipulate details of users described later.

Access Levels and Roles

The `web.auth` module has two methods of setting access priviledges, the first is by setting an access level which is simply a positive integer. The higher the number the greater the access level. An access level of `0` or `None` means no access. You can then check that the signed in user has an access level which is high enough to grant them access to a particular piece of functionality.

The second method is using roles; these are best described using an example. In a content management system users may be allowed to add, edit and remove pages so you might create the roles `add`, `edit` and `remove`. An administrator might have all three roles and an editor might only be specified the role `edit`. You can then grant users access to different parts of your application based on their role.

For the timebeing we will describe how to use access levels since they are simpler.

You can set the access level for a particular user or applicaiton using the `admin.setLevel()` method of the admin object. Each user can have a different access level to each application so you must specify the username, app name and level to set an access level.

```
admin.setLevel(username='john', app='app', level=1)
```

The easy way

If you just want to get up and running quickly `web.auth.admin()` takes a parameter *autoCreate*. If you specify *autoCreate=1* all the steps performed so far with the exception of committing the database will be performed automatically and you will have the user `john` already set up ready to test the system.

```
admin = web.auth.admin(driver='database', cursor=cursor, autoCreate=1)
connection.commit()
```

1.2.3 Authentication and Authorisation

Once the auth environment is set up and the appropriate users and access priveledges have been set up you will want to authorise and authenticate users.

In order to authenticate a user the user needs to be able to sign in with their username and password. In order to remain signed in the user information needs to be stored somewhere so that the user isn't immediately signed out agian on the next HTTP request.

The `web.auth` module uses a `web.session` module session store to store the auth session information about the current signed in user. This means you need to setup a `web.seesion` store as shown below. See the `web.session` module for full details. The session store for the auth session information is normally called `auth` but you can use whichever session store you prefer. You should be sure that the variables set in the store are not going to be accidently over-written by other applications by choosing a sotre name that other applications do not have access to.

```
import web.session
session = web.session.manager(driver='database', cursor=cursor, autoCreate=1)
if not session.load():
    session.create()
store = session.store('auth')
```

In order to authenticate users you will need to use a manager object. This has all the functionality of the admin object already described but also has session functionality.

```
import web.auth
auth = web.auth.manager(
    store=store,
    driver='database',
    expire=100,
    idle=20,
    autoCreate=1,
    cursor=cursor
)
```

The manager object takes the parameters *store*, *idle* and *expire* in addition to all the parameters of the admin object. *store* is the session store to use for the auth session, *expire* is the maximum length of time a user can be signed in for. If *expire* is `0` it means the user can be signed in indefinately (although practically the session from the `web.session` itself will not last forever). *idle* is the maximum length of time a user can be signed in for without visiting the site. Again a value of `0` means there is no limit.

Checking Who Is Signed In

If the manager finds that a user is currently signed in and that the auth session has not idled or expired, then the attribute `auth.signedInUser` will contain a user object containing all the auth information about that user. This is set to `None` if no user is signed in.

Usernames are case insensitive but are always stored in the driver as lowercase.

You can directly set the firstname, surname, email, group and active status of the user like this `auth.signedInUser.firstname = 'John'`

If no user is signed in you will need to present a sign in form to allow the user to sign in.

```
if auth.signedInUser != None:
    print web.header('text/plain'), "Authorised"
else:
    print web.header()
    # display sign in form
```

You can use whatever methods you like to sign a user in, just use `auth.signIn(username)` once you have checked the user's password and want to sign them in. They will be added to the auth session store.

If you don't want to provide the sign in functionality yourself you can use a sign in handler.

The Sign In Handler

The sign in handler performs all the checks necessary and returns a dictionary of variables to display to the user if the sign in was unsuccessful. You can us it like this:

```
import web.auth.handler.signIn
print web.header()
signInHandler = web.auth.handler.signIn.SignInHandler(manager=auth)
form = signInHandler.handle()
if form: # form needs displaying
    print '<html><body><h1>Please Sign In</h1>%(form)s<p>%(message)s</p></body></html>'%for
else:
    # We have just signed in, but we have not authorised the user
    pass
```

Even though the user is authenticated and signed in, we have not yet authorised them.

Authenticating the Signed In User

Once we have checked a user is signed in using `auth.signedInUser != None` we can authorise the user. The `.signedInUser` attribute of the manager object will contain a user object for the signed in user. The user object has an `authorise()` method which can be used to check the user's access priveledges. The method returns `False` if the user does not meet all the authorisation criteria and `True` otherwise.

```
if auth.signedInUser.authorise(app='app', level=1):
    print "Signed in successfully and authorised"
else:
    print "Not authorised to use this application"
```

The `authorise()` method takes a number of parameters for more advanced authorisation functionality.

1.2.4 Simple Example

Putting together everything in the previous sections gives us this full (but not very useful) application:

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

"""Auth Example. Username=john and Password=bananas (Case sensitive)"""
```

```
# show python where the modules are and enable error displays
import sys; sys.path.append('../'); sys.path.append('../../../')
import web.error; web.error.enable()
import web, web.database

# Setup a database connection
connection = web.database.connect(
    adapter="snakesql",
    database="webserver-auth",
    autoCreate=1
)
cursor = connection.cursor()

# Obtain a session manager
import web.session
session = web.session.manager(
    driver='database',
    autoCreate=1,
    cursor=cursor
)
if not session.load():
    session.create()

# Obtain Auth objects
import web.auth
auth = web.auth.manager(
    session.store('auth'),
    'database',
    idle=20,
    autoCreate=1,
    encryption='md5',
    cursor=cursor
)

# Authentication and Authorisation code
if auth.signedInUser != None and auth.signedInUser.authorise(app='app', level=1):
    print web.header('text/plain'), "Authorised"
else:
    print web.header()
    # Sign in however you like.. but you could use this signIn handler
    import web.auth.handler.signIn
    signInHandler = web.auth.handler.signIn.SignInHandler(
        manager=auth,
        encryption='md5'
    )
    form = signInHandler.handle()
    if form: # ie there is a problem and the sign in form needs displaying
        print """<html><body><h1>Please Sign In</h1>
        %(form)s<p>%(message)s</p></body></html>"""%form
    else:
        # We have just signed in, but we have not authorised the user
        if auth.signedInUser.authorise(app='app', level=1):
            print "Signed in successfully"
        else:
            print "Not authorised to use this application"

connection.commit()
connection.close()
```

You can test this example by starting the test webserver in 'scripts/webserver.py' and visiting http://localhost:8080/doc/src/lib/webserver-web-auth.py on your local machine. The username is `john` and the password is `bananas`.

1.2.5 Advanced Authorisation Options

As well as access levels, `web.auth` also supports role based authorisation, groups of users, and disabling user accounts by setting the `active` property.

Using Roles

Role based authorisation is the reccommended way of using the `web.auth` module as it is more powerful and flexible than using access levels. Of course, you can if you wish combine access levels and roles by specifying both.

Before you can grant a user a particular role you must first add the role to the auth database using an admin or manager object as described earlier.

```
admin.addRole('add')
admin.addRole('edit')
```

Each user is granted different roles to different applications. If you have applications named `cms` and `news` then a particular user might be granted the role `add` and `edit` to `news` but only `edit` to `cms`.

```
admin.setRole(username='james', app='news', role='add')
admin.setRole(username='james', app='news', role='edit')
admin.setRole(username='james', app='cms', role='edit')
```

The user can then be authorised based on their roles:

```
>>> user = admin.user('james')
>>> user.authorise(app='cms', role='add')
0
>>> user.authorise(app='cms', role='edit')
1
>>> user.authorise(app='news', role='add')
1
>>> user.authorise(app='news', role='edit')
1
```

To obtain a user's roles you can do one of the following:

```
>>> user = admin.user('james')
>>> user.roles
{'cms':['edit'], 'news':['add','edit']}
>>> admin.roles(username='james', app='cms')
'edit'
```

There are also methods for `roleExists()`, `removeRole` and `unsetRole()`.

Using Groups

It is sometimes useful to consider groups of users, perhaps if people from different companies use your application but you want to keep their users separate. This can be achieved with groups.

Before you can use a group, it must be added to the database:

```
admin.addGroup('butcher')
admin.addGroup('fishmonger')
```

The when adding users you can specify the group:

```
admin.addUser(username='james', password='password', group='butcher')
admin.addUser(username='sally', password='password', group='butcher')
admin.addUser(username='anne',  password='password', group='fishmonger')
```

Alternatively you can specify or change the group of an already created user:

```
user = admin.user('james')
user.group = 'fishmonger'
```

You can obtain a list of groups using the `groups()` method:

```
>>> admin.groups()
('butcher', 'fishmonger')
```

You can then authorise a user based on their group.

```
>>> admin.user('anne').authorise(group='fishmonger')
1
>>> user = admin.user('james')
>>> user.authorise(group='butcher')
0
>>> user.authorise(group='fishmonger')
1
```

There are also methods `groupExists()` and `removeGroups()`.

Disabling Accounts using active

Occasionally it is useful to disable a user, say for example if they haven't renewed their subscription. You don't want to completely remove their accound since you would have to add all the information if they paid the fee, you just want to disable the account.

This can be achieved by setting a user's active property. Here are some ways of setting the active property:

```
admin.addUser(username='vicki', group='butcher', password='password', active=0)
admin.user('james').active = 0
```

The default value of the *active* parameter of the `authorise()` method if 1 so once active is set to 0, the user will not be authorised unless *active* is specified as 0 to mean only authorise disabled accounts ot `None` to mean authorise

all accounts.

```
>>> admin.user('james').authorise()
0
>>> admin.user('james').authorise(active=0)
1
```

Example

Putting together everything in the previous sections gives us this full authorisation example:

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

# show python where the modules are
import sys; sys.path.append('../'); sys.path.append('../../../')
import web, web.database

# Setup a database connection
connection = web.database.connect(
    adapter="snakesql",
    database="command-auth"
)
cursor = connection.cursor()

# Obtain Auth objects
import web.auth
from web.errors import AuthError
admin = web.auth.admin('database', cursor=cursor)

# Setup the environment (destroying the existing environment)
admin.removeAuthEnvironment(ignoreErrors=True)
admin.createAuthEnvironment()

# Setup the users and their access rights
admin.addApp('cms')
admin.addApp('news')

admin.addGroup('butcher')
admin.addGroup('fishmonger')

admin.addRole('add')
admin.addRole('edit')

admin.addUser(username='james', group='butcher', password='password')
admin.addUser(username='sally', group='butcher', password='password')
admin.addUser(username='vicki', group='butcher', password='password', active=0)

admin.addUser(username='anne',  group='fishmonger', password='password')
admin.addUser(
    username='john',
    group='fishmonger',
    password='password',
    firstname='John',
    surname='Smith',
    email='john@example.com'
)
```

```
admin.setLevel('anne', 'news', 2)
admin.setLevel('john', 'news', 1)
admin.setLevel('anne', 'cms', 1)
admin.setLevel('john', 'cms', 2)

admin.setRole(username='james', app='cms', role='add')
admin.setRole(username='sally', app='cms', role='edit')
admin.setRole(username='james', app='news', role='edit')

print 'Active Option'
print 'Sally: ', admin.user('sally').authorise()
print 'Sally: ', admin.user('sally').authorise(active=0)
print 'Sally: ', admin.user('sally').authorise(active=None)
print 'Vicki: ', admin.user('vicki').authorise()
print 'Vicki: ', admin.user('vicki').authorise(active=0)
print 'Vicki: ', admin.user('vicki').authorise(active=None)
print ''
print 'Group Option'
print 'Anne:  ', admin.user('anne').authorise(group='fishmonger')
print 'James: ', admin.user('james').authorise(group='fishmonger')
print ''
print 'Access Levels'
print 'Anne:  ', admin.user('anne').authorise(app='news', level=2)
print 'John:  ', admin.user('john').authorise(app='news', level=2)
print 'Anne:  ', admin.user('anne').authorise(app='cms',  level=2)
print 'John:  ', admin.user('john').authorise(app='cms',  level=2)
print ''
print 'Roles'
print 'James: ', admin.user('james').authorise(app='cms',  role='add')
print 'Sally: ', admin.user('sally').authorise(app='cms',  role='add')
print 'Sally: ', admin.user('sally').authorise(app='cms',  role='edit')
print 'James: ', admin.user('james').authorise(app='news', role='add')
print 'James: ', admin.user('james').authorise(app='news', role='edit')
print 'Sally: ', admin.user('sally').authorise(app='news', role='edit')
print admin.roles()
print admin.roleExists('edit')
print admin.roleExists('delete')
admin.addApp('test')
admin.addUser(username='test', password='password')
admin.addRole('test')
admin.setRole(username='test', app='test', role='test')
admin.setLevel('test', 'test', 1)
try:
    admin.removeRole('test')
except AuthError,e:
    print str(e)
else:
    raise Exception('Failed to catch remove role error')
admin.removeRole('test', force=1)
print admin.roles(username='test')
admin.addRole('test')
admin.setRole(username='test', app='test', role='test')
admin.setRole(username='test', app='cms', role='test')

print admin.roles()
print admin.roles(username='test')
print admin.roles(username='test', app='test')
print admin.user('test').roles
```

```
admin.unsetRole('test','test','test')
print admin.user('test').roles

print ''
print 'Groups'
print admin.groups()
print admin.groupExists('butcher')
print admin.groupExists('newsagents')
admin.addGroup('newsagents')
admin.user('test').group = 'newsagents'
print admin.user('test').group

print admin.groups()
try:
    admin.removeGroup('newsagents')
except AuthError,e:
    print str(e)
else:
    raise Exception('Failed to catch remove group error')
admin.removeGroup('newsagents', force=1)
print admin.groups()

print ''
print 'Users'
print admin.userExists('james')
print admin.users()
print admin.users(group='butcher')
print admin.users(group='butcher', active=0)
print admin.users(group='butcher', active=1)
print admin.users(app='cms', role='add')
print admin.users(group='butcher', app='cms', role='add', active=0)
print admin.users(group='butcher', app='cms', role='add', active=1)
print admin.users(group='fishmonger', app='cms', role='add')
vicki = admin.user('vicki')
print vicki.active
vicki.active = 1
print admin.user('vicki').active
print vicki.firstname
vicki.firstname = 'Victoria'
print admin.user('vicki').firstname
print ''
print 'Apps'
print admin.apps()
print admin.appExists('cms')
try:
    admin.removeApp('test')
except AuthError,e:
    print str(e)
else:
    raise Exception('Failed to catch app in use error')
print "App removed"
print admin.roles(username='test')
print admin.levels(username='test')
admin.removeUser('test')
print admin.userExists('test')
print ''
print 'Levels'
print admin.levels('anne')
admin.setLevel('anne', 'cms', None)
```

```
print admin.user('anne').levels
print
print 'Authorise'
print admin.user('james').authorise(group='butcher', app='cms', level=1)
print admin.user('john').authorise(group='butcher', app='cms', level=1)
print admin.user('john').authorise(group='fishmonger', app='cms', level=1)

connection.rollback()
connection.close()
```

Note: Since this example is very database intensive you may wish to change the first few lines to use a different database adapter rather than SnakeSQL which runs rather slowly.

You can test this example by running `python doc/src/lib/command-web-auth.py`

The ouput produced is as follows:

```
Active Option
Sally:  True
Sally:  False
Sally:  True
Vicki:  False
Vicki:  True
Vicki:  True

Group Option
Anne:   True
James:  False

Access Levels
Anne:   True
John:   False
Anne:   False
John:   True

Roles
James:  True
Sally:  False
Sally:  True
James:  False
James:  True
Sally:  False
('add', 'edit')
True
False
The role 'test' is still in use by the following users: test
{}
('add', 'edit', 'test')
{'test': 'test', 'cms': 'test'}
('test',)
{'test': 'test', 'cms': 'test'}
{'cms': 'test'}

Groups
('butcher', 'fishmonger')
True
False
newsagents
('butcher', 'fishmonger', 'newsagents')
The group 'newsagents' is still in use by the following users: test
('butcher', 'fishmonger')

Users
True
('james', 'sally', 'vicki', 'anne', 'john', 'test')
('james', 'sally', 'vicki')
('vicki',)
('james', 'sally')
('james',)
()
('james',)
()
0
1

Victoria
```



```
('cms',)
True
The app 'test' is still in use specifying access levels for the following users: test
App removed
```

1.2.6 Encryption

The password stored in the database can be encrypted for extra security. Encryption can be enabled by specifying *encryption='md5'* in the admin or manager objects.

```
auth = web.auth.manager(
    session.store('auth'),
    'database',
    autoCreate=1,
    cursor=cursor,
    encryption='md5',
)
```

The encryption method must also be specified in the sign in handler so that the handler knows to encrypt the password specified before comparing it with the encrypted version stored in the database.

```
signInHandler = web.auth.handler.signIn.SignInHandler(manager=auth, encryption='md5')
```

There are some drawbacks to using encryption, the main one being that the users password is not actually stored anywhere so if a user forgets their password you must reset it rather than reading it from the database.

Also the password attribute of a user object will return the encrypted password not the real password.

```
>>> auth.user('john').password
'5f4dcc3b5aa765d61d8327deb882cf99'
```

Finally, if you wish to change the type of encryption you are using after having added users to the datbase you will need to rest their passwords.

1.2.7 API Reference

AuthAdmin Object

The `AuthAdmin` object is aliased as `web.auth.admin` and should be used as `web.auth.admin`.

class `AuthAdmin`(*driver*, [*autoCreate=0*], [*encryption=None*], [***driverParams*])
Auth Manager for creating modifying and removing users and applications.

driver The type of driver being used. Currently only `'database'` is allowed

autoCreate If set to `True` the necessary tables will be created (removing any existing tables) if any of the tables are missing and a user named `john` with a password `bananas` will be set up with an access level of `1` to the application `app`. This is designed for easy testing of the module.

encryption The encryption method used to encrypt the password. Can be `None` or `'md5'`. `Warning you cannot change the encryption method once a user is added without resetting the password.`

*****driverParams*** Any parameters to be specified in the format *name=value* which are needed by the driver specified by *driver*

`autoCreated`
Will be `True` if the tables and user were autoCreated, `False` otherwise.

enycryption
The encryption method used

completeAuthEnvironment()
Returns `True` if the environment is correctly setup, `False` otherwise. In the case of the database driver this method simply checks that all the necessary tables exist.

createAuthEnvironment()
Creates the necessary environment. In the case of the database driver this method creates all the required tables. If any of the tables already exist an `AuthError` is raised.

removeAuthEnvironment([*ignoreErrors=False*])
Removes the environment. In the case of the database driver this method drops all the tables. If any of the tables are not present an `AuthError` is raised unless *ignoreErrors* is `True`

apps()
Return a list of application names.

appExists(*app*)
Return `True` if there is an application named *app*, `False` otherwise.

addApp(*app*)
Adds an application named *app*.

removeApp(*app,*[*force=0*])
Remove the application named *app*. If *force=1*, the application is removed even if access levels or roles are specified for users using the application.

user(*username*)
Return an `AuthUser` object for the user specified.

users([*group=[]*], [*active=None*], [*app=None*], [*role=None*])
Return a list of usernames.

userExists(*username*)
Returns `True` if there is a user with the username *username*, `False` otherwise.

addUser(*username, password*[*,firstname=''*][*,surname=''*][*email=''*], [*active=1*], [*group=None*])
Adds a user with the username *username* and password *password* to the system. You can optionally also specify the firstname, surname and email address of the user. You can choose a group for the user and whether or not the user is active. If encryption is used the password is encrypted.

removeUser(*username*)
Removes the user with the username *username*.

levels(*username,* [*app=None*])
Returns the access level of the user *username* for the application named *app*. If *app* is not specified or `None`, a dictionary of application name, access level pairs is returned.

setLevel(*username, app, level*)
Sets the access level of the user *username* for the application named *app* to *level*.

roles([*username=None*], [*app=None*])
Returns the roles based on the options specified. If *username* and *app* are not specified, the available roles are returned as a sequence. If *username* is specified, a dictionary of application namd role pairs are returned for that user, if *username* and *app* are both specified, the roles for the particular user and application are returned.

roleExists(*role*)
Returns `True` if there is a role named *role*, `False` otherwise.

addRole(*role*)
Adds the new role *role* to the database. If it already exists an `AuthError` is raised.

removeRole(*role,* [*force=0*])
Remove the role named *role*. If *force=1*, the role is removed even if it is being used by any users.

setRole(*username, app, role*)
Give the role *role* to the user *username* for the application *app*

setRole(*username, app, role*)
Remove the role *role* from the user *username* for the application *app*

groups()
Returns a sequence of available group names

groupExists(*group*)
Returns `True` if there is a group named *group*, `False` otherwise. `None` is a valid group since a user can have no group.

addGroup(*group*)
Adds the new group *group* to the database. If it already exists an `AuthError` is raised.

removeGroup(*group,* [*force=0*])
Remove the group named *group*. If *force=1*, the group is removed even if it is being used by any users.

AuthSession Object

The `AuthSession` object is aliased as `web.auth.session` and should be used as `web.auth.session`.

class AuthSession(*store,* [*expire=0*], [*idle=0*]) ***store***
A valid `web.session` `Store` object.

expireAn integer specifying the number of seconds before the user is signed out. A value of 0 disables the expire functionality and the user will be signed in until they sign out. **Note:** If the underlying session expires, the cookie is removed or the sign in idles before the expire time specified in *expire* the user will be signed out.

idleAn integer specifying the maximum number of seconds between requests before the user is automatically signed out. A value of 0 disables the idle functionality allowing an unlimited amount of time between user requests. **Note:** If the underlying session expires, the cookie is removed or the sign in expires before the idle time specified in *idle* the user will be signed out.

For managing the auth information stored in the session store.

Has the following attributes which should not be set.

store
The session store used to store the auth session information

expire
The expire time

store
The idle time

Has the following methods:

username()
Returns the username as a string if a user is signed in, otherwise returns an empty string `''`.

signIn(*username*)
Sign in the user with username *username*.

signOut()
Sign out the signed in user.

userInfo()
If a user is signed in, returns a dictionary with the following keys: `'username'`, `'started'`, `'accessed'`, `'expire'`, `'idle'`. If no user is signed in returns `None`.

AuthManager Object

The `AuthManager` object is aliased as `web.auth.manager` and should be used as `web.auth.manager`. It is the object most frequently used for auth management.

The `AuthManager` object is derived from the `AuthAdmin` and `AuthSession` objects. Conesequently it has all the functions and methods of the admin and session objects as well as the following functionality:

class `AuthManager`(*store, driver,* [*expire=0*], [*idle=0*], [*autoCreate=0*], [*encryption=None*], [***driverParams*)

`user`([*username*])

Return an `AuthUser` object for the user specified. If no user is specified an `AuthUser` object for the currently signed in user is returned. If no user is signed in, `None` is returned.

AuthUser Objects

The user object is retuned by `AuthAdmin` and `AuthManager` objects' `user()` method and should not be created directly.

The attributes `firstname`, `surname`, `email`, `password`, `group` and `active` can all be directly set and their values will be updated in the database.

The class has the following properties:

class `AuthUser`

`username`

The username of the user. Usernames are case insensitive but are always stored and returned as lowercase. This means that if you want to compare a username from a database with a value entered by a user, you should first convert the value entered by a user to lowercase like this: `username = username.lower()`

The username of a user cannot be changed.

`password`

The user's password, 1-255 characters.

`firstname`

The user's firstname, 1-255 characters. Optional

`surname`

The user's surname, 1-255 characters. Optional

`email`

The user's email address, max 255 characters. Optional

`group`

The user's group, max 255 characters or `None` if no group has been set. Optional

`active`

`True` or `False` depending on whether the user is considered active.

`levels`

The access levels for the applications the user has access to as a dictionary with application names as keys. Levels can only be set through the `setLevel()` method of `AuthManager` or `AuthAdmin` objects. **Warning:** Changing the value stored in `levels` will not update the database.

`roles`

The user's roles for each application as a dictionary with application names as keys. Roles can only be set through the `setRole()` method of `AuthManager` or `AuthAdmin` objects. **Warning:** Changing the value stored in `roles` will not update the database.

`authorise`([*app=None*], [*level=None*], [*role=None*], [*active=1*], [*group=[]*])

Return `True` if the user is authorised for the options specified, `False` otherwise.

If *active=0* only disabled accounts are authorised, if *active=None* both active and disabled accounts are authorised. If *group* is not specified all groups are authorised, if *group=None* only users not in a group are

authorised, otherwise only users in the group specified are authorised. If *level* or *role* are specified, *app* must be specified too.

Sign In Handler

class **`SignInHandler`**(*manager*, [*cgi=None*], [*message=”*], [*encryption=None*])
Used to automate the sign in process.

managerAn auth manager object

cgiAn alternative `cgi.FieldStorage()` object to be use instead of the default one.

messageThe default error message to display before a user tires to sign in.

encryptionThe encryption method being used to encrypt passwords in the database.

`handle`()
If the user has been signed in returns `None`. Otherwise, returns a dictionary with two keys, `'form'` containing the sign in form together with any error messages and `'message'` a message explaining why the user couldn't be signed in.

The dictionary can then be combined with a template using `%(form)s` dictionary substitution to display an HTML page to the user.

1.3 `datetime` — Compatibility code providing date and time classes for Python 2.2 users

The following classes provide a subset of the functionality of the Python 2.3 `date`, `time` and `datetime` Objects. If you want to do sophisticated date and time classes is it is reccommended that you use Python 2.3. These classes are designed only so that Python 2.2 users can still use date and time functionality in the `web.database` module.

Note: It should be noted that although the `time` and `datetime` classes have the ability to support microseconds, the `web.database` module only deals in whole seconds since some of the underlying databases do not support microseconds.

class **`date`**(*year, month, day*)
A `date` object represents a date (year, month and day) in an idealized calendar, the current Gregorian calendar indefinitely extended in both directions. January 1 of year 1 is called day number 1, January 2 of year 1 is called day number 2, and so on. This matches the definition of the ”proleptic Gregorian” calendar in Dershowitz and Reingold's book *Calendrical Calculations*, where it's the base calendar for all computations. See the book for algorithms for converting between proleptic Gregorian ordinals and many other calendar systems.

All arguments are required. Arguments may be ints or longs, in the following ranges:

- `MINYEAR <=` *year* `<= MAXYEAR`
- `1 <=` *month* `<= 12`
- `1 <=` *day* `<= number of days in the given month and year`

If an argument outside those ranges is given, `ValueError` is raised.

Instance Attributes:

`year`
Between `MINYEAR` and `MAXYEAR` inclusive.

`month`
Between 1 and 12 inclusive.

day
Between 1 and the number of days in the given month of the given year.

class time([*hour=0*][,*minute=0*][,*second=0*][,*microsecond=0*])
A `time` object represents a (local) time of day, independent of any particular day.

All arguments are required. Arguments may be ints or longs, in the following ranges:

- `0 <=` *hour* `< 24`
- `0 <=` *minute* `< 60`
- `0 <=` *second* `< 60`
- `0 <=` *microsecond* `< 1000000`

If an argument outside those ranges is given, `ValueError` is raised.

Instance Attributes:

hour
In `range(24)`.

minute
In `range(60)`.

second
In `range(60)`.

microsecond
In `range(1000000)`.

class datetime(*year, month, day*[,*hour=0*][,*minute=0*][,*second=0*][,*microsecond=0*])
A `datetime` object is a single object containing all the information from a `date` object and a `time` object. Like a `date` object, `datetime` assumes the current Gregorian calendar extended in both directions; like a `time` object, `datetime` assumes there are exactly 3600*24 seconds in every day.

All arguments are required. Arguments may be ints or longs, in the following ranges:

- `MINYEAR <=` *year* `<= MAXYEAR`
- `1 <=` *month* `<= 12`
- `1 <=` *day* `<= number of days in the given month and year`
- `0 <=` *hour* `< 24`
- `0 <=` *minute* `< 60`
- `0 <=` *second* `< 60`
- `0 <=` *microsecond* `< 1000000`

If an argument outside those ranges is given, `ValueError` is raised.

Instance Attributes:

year
Between `MINYEAR` and `MAXYEAR` inclusive.

month
Between 1 and 12 inclusive.

day
Between 1 and the number of days in the given month of the given year.

hour
In `range(24)`.

minute
In `range(60)`.

second
In `range(60)`.

microsecond
In `range(1000000)`.

All classes have the following class methods:

now()
Returns a `datetime.datetime` object representing the current date and time

strftime(*format*)
Format the date using standard time module string format strings:

```
%a Locale's abbreviated weekday name.
%A Locale's full weekday name.
%b Locale's abbreviated month name.
%B Locale's full month name.
%c Locale's appropriate date and time representation.
%d Day of the month as a decimal number [01,31].
%H Hour (24-hour clock) as a decimal number [00,23].
%I Hour (12-hour clock) as a decimal number [01,12].
%j Day of the year as a decimal number [001,366].
%m Month as a decimal number [01,12].
%M Minute as a decimal number [00,59].
%p Locale's equivalent of either AM or PM.
%S Second as a decimal number [00,61]. (1)
%U Week number of the year (Sunday as the first day of the week)
   as a decimal number [00,53]. All days in a new year preceding
   the first Sunday are considered to be in week 0.
%w Weekday as a decimal number [0(Sunday),6].
%W Week number of the year (Monday as the first day of the week)
   as a decimal number [00,53]. All days in a new year preceding
   the first Monday are considered to be in week 0.
%x Locale's appropriate date representation.
%X Locale's appropriate time representation.
%y Year without century as a decimal number [00,99].
%Y Year with century as a decimal number.
%Z Time zone name (no characters if no time zone exists).
%% A literal "%" character.
```

For example:

```
>>> datetime.datetime(2004,5,3,10,30,50).strftime('%x-%X')
'05/03/04-10:30:50'
```

timetuple()
Returns a basic time tuple for the date. **Warning:** The last 6 entries in the tuple returned from this function are obtained from `time.localtime()` and do not represent anything.

isoformat()
Return the date as a standard SQL string of the format. **Warning:** `microseconds` are ignored.

```
>>> import web, datetime
>>> datetime.date(2004,5,3).isoformat()
'2004-05-03'
>>> datetime.time(10,30,50,9).isoformat() # Microseconds are ignored
'10:30:50'
>>> datetime.datetime(2004,5,3,10,30,50).isoformat()
'2004-05-03 10:30:50'
```

1.3.1 Module-Level Functionality

The `datetime` module exports the following constants:

`MINYEAR` The smallest year number allowed in a date or datetime object. MINYEAR is 1.

`MAXYEAR` The largest year number allowed in a date or datetime object. MAXYEAR is 9999.

For Example:

```
>>> import web # Necessary to set up the paths so datetime can be imported
>>> import datetime
>>> datetime.MINYEAR
1
>>> datetime.MAXYEAR
9999
```

1.3.2 Compatibility with Python 2.3 and above

The `datetime` module is as combitible as possible with Python 2.3. It does not implement all the features of the Python 2.3 `datetime` module but it implements all the ones the modules themselves need. Most of the time this is all that is required. One important omission is that you cannot add or subtract date objects in this combatibility module. Instead convert them to times and then convert them back again.

In order to write code compatible with both Python 2.2 and 2.3 there is one particular point to note; `datetime` is not a type in Python 2.2, it is a class. This means that `datetime.datetime.now()` will not work because you can't call the `now()` of an uninitialised class. Instead use `datetime.datetime(2004,1,1).now()`. This will produce the same (correct) result in both versions regardless of the values chosen for the date.

1.4 `web.database` — SQL database layer

The `web.database` module is a simple SQL abstraction layer which sits on top of a DB-API 2.0 cursor to implement data type conversions, provide database independance and offer a more Python-like interface to the data returned from queries. This is achieved by implementing common field types, a portable SQL dialect and a standard API for all supported databases.

Here are the main features of the module:

- 100% compatible with the underlying DB-API 2.0 cursor. A `web.database` cursor provides access to the underlying DB-API 2.0 cursor.

- Provides methods including `select()`, `insert()`, `update()`, `delete()`, `create()`, `alter()` and `drop()` which build and customise the SQL depending on the database being used providing database independance.
- Provides strong typing for the data being used. No need to deal with SQL strings, the module automatically encodes and decodes data for the approriate column.

This module has a number of different layers of increasing complexity and decreasing portability. It is important you understand which layer you wish to use for a particular task. If for example you are only going to work with one database you do not need to be concerned about portability and so might use the `cursor` object in direct mode. If you don't know any SQL you might choose to use the `web.database.object` module to treat the database as a Python dictionary and allow portable access.

Warning: The `web.database` module provides total database portability by converting SQL and data types to an appropriate form for a limited subset of functions and data types of the underlying database engine. There are two drawbacks to this approach. Firstly the `web.database` layer needs to know the structure of the database. It does this by mainting a special table which it hides. The second drawback is that if you access the database outside the `web.database` module it is possible that the changes you make will not be compatible with the `web.database` module.

Having said all that, if you only access your databases through the `web.database` module in protable mode (the default) these drawbacks will not be an issue.

If you are looking for a database abstraction module to get away from using DB-API methods, and to pick up features such as results that are returned by field name but are not so worried about the complete portability provided by portable mode, perhaps because you only intend to use one database, you could instead use these modules in direct mode.

See Also:

Python DB-SIG Pages
(http://www.python.org/topics/database/)

To find out more about the DB-API 2.0 or how to program using DB-API 2.0 methods, please visit http://www.python.org/topics/database/. The rest of this documentation will assume you are not interested in using the cursor as a DB-API 2.0 cursor and that you want to know the additional features available.

1.4.1 Background

Most database engines currently have many common features but their differences are such that Python code written for one database engine using the DB-API 2.0 is unlikely to work with another database engine without some degree of modification. To complicate matters further many DB-API 2.0 drivers are not actually fully DB-API 2.0 compliant.

Variation between database engines occurs in SQL syntax, choice of field types and choice of which Python object to use to represent field values.

The DB-API 2.0 specification was designed with these differences in mind so that module implementers could make full use of the features of their particular database engine. This module provides a simple, standardised and portable API and SQL dialect which also exposes the interface components of the underlying DB-API 2.0 cursor. In this way users can access a database in a simplified and portable fashion for simple operations whilst exposing the DB-API 2.0 interface for more complex operations.

The drawback of this approach is that some of the fields available in a particular database will not be available through this module. Also there is no support for complex SQL commands including indexes or views since not all databases support them. The approach is only to support what is available to all databases being used.

If a database-specific feature is needed for a specific call you can always use the underlying cursor object directly. By using the `web.database` module as much as possible you will still make your code more portable across databases should you ever need to change servers and by using the `web.database` module exclusively you can gain true

database portability.

One of the major advantages of using `web.database` is that it comes with a pure Python SQL engine named SnakeSQL which fully implements the specification (albeit slowly) so if you use `web.database` in your own code you can guarantee your users will be able to run your application even if they do not have access to a better known database engine.

Comments and questions about this specification may be directed to James Gardner at docs at pythonweb.org.

Example Code

Here is some example code to give you a flavour of the various ways the module operates.

The user connects to a database and obtain a cursor as follows:

```
import web.database
connection = web.database.connect(adapter='MySQLdb', database='testDatabase')
cursor = connection.cursor()
```

They interact with the database through a series of methods which form a database abstraction layer. Each method builds an SQL string in accordance with the syntax of the driver and executes it according to the options specified:

```
results = cursor.select(columns=['name'], tables=['testTable'], where=cursor.where("name = 'Jam
```

Results could instead be retrieved as follows if *fetch* was `False`:

```
results = cursor.fetchall(format='dict')
```

Results are automatically converted to the defined types and returned in the correct format.

`web.database` cursors also support the execute() statement and qmark style parameter substitutions:

```
cursor.execute("select name from testTable where name = ?", ['James'])
```

The SQL is parsed, the parameters converted to SQL and inserted in the correct places and the appropriate abstraction layer method is executed.

The `web.database` field types are stored in a table in the database so that the `web.database` driver knows the field types and names of the fields in the tables so that conversions can be made.

The `Connection` object has an attribute `.tables` which is a dictionary of `Table` objects describing everything `web.database` knows about the tables. Each `Table` object is made up of `Column` objects containing field information and converters for each field.

After a `SELECT` statement the cursor attribute `.info` contains a list of `Column` objects for the columns selected to provide all the information available about those columns as well as conversion methods.

If the user needs to access the underlying DB-API driver using its own SQL dialect instead of the portable `web.database` one he can do so easily in two ways:

```
cursor.execute("select name from testTable where name = 'James'", mode='direct')
cursor.baseCursor.execute("select name from testTable where name = 'James'")
```

All the methods and objects have similar functionality to allow the user access to the underlying driver.

The use of an object relational mapper in `web.database.object` means forms can be automatically generated to provide data access.

1.4.2 Introduction

Understanding Field Types

The information you send to the database and the information retrieved from the database will be automatically converted to the correct formats so that you can treat the values as normal Python objects.

Traditional SQL databases usually have support for a number of different fields. Date fields behave differently to integer fields for example. All of the fields are set using an SQL representation of the data in the form of a string and all of the queries from the database return strings.

The `web.database` module provides ten field types and rather than passing information to and from the database as specially SQL encoded strings, you can also pass it as a python data structure. For example to set an `Integer` field you could give the cursor an integer. To set a `Date` field you would give the cursor a `datetime.date` object. The `web.database` cursor would do all the conversion for you.

Furthermore when you retrieve information from the database the `cursor` will convert the strings recieved back into Python objects so that you never need to worry about the encodings.

This doesn't sound like too much of a big deal but because different databases handle different datatypes in slightly different ways your SQL could have different results on different databases. Programming with a `web.database` `cursor` removes these inconsistencies.

Here are the supported datatypes:

Type	Description
`Bool`	True or False
`Integer`	Any Python integer (not Python Long or Decimal)
`Long`	Any Python long integer between -9223372036854775808 and 9223372036854775807
`Float`	Any Python floating point number
`String`	A string of 255 characters or less (Not unicode?) [a]
`Text`	A 24-bit string [b]
`Binary`	A 24-bit binary string [b]]
`Date`	Any valid Python `datetime.date` object. Takes values in the form of python `datetime` objects. Only stores days
`Time`	Any valid Python `datetime.time` object. Takes values in the form of python `datetime` objects. Only stores hour
`Datetime`	Any valid Python `datetime.datetime` object. Takes values in the form of python `datetime` objects. Only stores

[a] Some databases make a distinction between short strings (often named VARCHAR) and long strings (often TEXT). Short string fields are normally faster and so a distinction is also made in this specification.

[b] Although Python supports strings of greater than 24 bit, a lot of databases do not and so in order to be compatible with those databases Binary and String objects should be no longer than 24 bit.

[c] Although Python ¡ 2.3 does not support datetime objects, pure Python compatible libraries exist for Python ¡ 2.3 and these can be used instead so it makes sense to use the standard Python types where possible.

The values you pass to the `cursor.execute()` method should be of the correct type for the field they are repre-

senting. The values returned by the `cursor.fetchall()` method will automatically be returned as the appropriate Python type.

For example, `Bool` fields should have the Python values `True` or `False`, `Long` fields should be a valid Python `long` etc.

There are some exceptions:

`String` fields should contain Python strings of 255 characters of less. `Text` fields should contain 24 bit strings less. For strings longer than this length you should consider saving the string in a file and saving the filename in the database instead.

`Date`, `Datetime` and `Time` fields take Python `datetime.date`, `datetime.datetime` and `datetime.time` objects respectively.

Unfortunately Python 2.2 and below do not support the `datetime` module. However `web.database` uses a compatibility module that behaves closely enough for most purposes. Simply import `web.database` and then you can import the datetime module automatically. This is what it looks like at the Python prompt:

```
Python 2.2.3 (#42, May 30 2003, 18:12:08) [MSC 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import \module{web.database}
>>> import datetime
>>> print datetime.date(2004,11,24)
2004-11-24
>>>
```

1.4.3 Connecting to a Database

Connecting to a database is really very easy. The code below will connect to a MySQL database named 'test'.

```
import web, web.database
connection = web.database.connect(adapter="mysql", database="test")
```

Below is a description of the full range of parameters the `connect()` function can take (Obviously not all of the database support all of the parameters):

`connect``(`*adapter*`,`[*database*`,`][*user*`,`][*password*`,`][*host*`,`][*port*`,`][*socket*`,`][`**`*params*`])`

Constructor for creating a connection to a database. Returns a `Connection` object. Not all databases will use all the parameters, but databases should use the parameters specified and not abbreviated versions. Any more complex parameters are passed directly to the underlying driver's `connect()` method.

adapter The type of database to connect to. Can currently be `'MySQL'`, `'PySQLite'` or `'web.database'` but it is hoped that most database drivers will eventually be supported.

database The database name to connect to.

user The username to connect with.

password The password to use.

prepend A string to be transparantly used in front of all database tables.

host The host to connect to if the database is running on a remote server.

port The port to connect to if the database is running on a remote server.

socket The socket to connect to if the database is running locally and requires a socket.

params Any other parameters to be passed to the driver

Here are some examples:

Connect to the unpassworded MySQL database `MyDatabase` on a local server connected through a socket '/tmp/mysqld.sock'. Another common socket file used is '/tmp/mysql.sock'.

```
connection = web.database.connect(
    adapter="mysql",
    database="MyDatabase",
    socket="/tmp/mysqld.sock",
)
```

Connect to a the database `MyDatabase` as `username` with password `password`. The MySQL server is runing remotely at `mysql.example.com` on port `3336`:

```
connection = web.database.connect(
    adapter="mysql",
    database="MyDatabase",
    host="mysql.example.com",
    port="3336",
    user="username",
    password="password",
)
```

Connect to the `web.database` database in the directory 'C:/TestDirectory'

```
connection = web.database.connect(
    adapter="SnakeSQL",
    database="C:/TestDirectory",
)
```

Note: Windows users may find it easier to use forward slahes in paths to avoid having to quote backslashes. Both work equally well.

1.4.4 Using a Table Prepend

The `web.database` `connection` object also supports using a table prepend which is a string prepended to every table the `web.database` module uses but is totally transparent to the programmer.

You can use a table prepend like this:

```
connection = web.database.connect(
    adapter="SnakeSQL",
    database="C:/TestDirectory",
    prepend = 'Test',
)
```

Every table created will have the word `Test` prepended to its name but you would access the database as if no prepend existed. For example if you created tables named `People` and `Houses` they would actually be created as

`TestPeople` and `TestHouses` but a call to `cursor.tables()` would return `('People', 'Houses')` so that you can treat the database as if no prepend exists.

This is very handy as it means that you could, for example, setup test, development and production environments all within the same database simply by modifying the table prepend and no other changes to your code need to be made. It also means you can run more than one copy of code which uses a database in the same database but with each connection having a different table prepend. This is useful in a shared hosting environment where the number of databases you have access to is restricted.

1.4.5 Cursor Options

Once you have connected to the database you will need a `Cursor` object with which to manipulate the database. Cursor stands for a "CURrent Set Of Results".

Once we have the connection to the database, `connection`, we can easily create a cursor by calling the `connection`'s `cursor()` method.

```
import web, web.database
connection = web.database.connect(adapter="mysql", database="test")
cursor = connection.cursor()
```

The next sections show you the different ways to use the `cursor`.

1.4.6 Executing SQL

The `execute()` method is used to retrieve information from a database and looks like this:

```
cursor.execute("SELECT * FROM Test")
```

or

```
cursor.execute("INSERT INTO Test (dateColumn, numberColumn) VALUES ('2004-11-8', 4)")
```

`web.database` uses `?` style parameter substitution. This means the `execute()` method can take a list of values to substitute for any unquoted `?` symbols in the SQL string.

```
values = [datetime.date(2004,11,8), 4]
cursor.execute("INSERT INTO Test (dateColumn, numberColumn) VALUES (?, ?)", values)
```

or

```
cursor.execute(
    sql="UPDATE Test SET dateColumn=?, numberColumn=? WHERE stringColumn=?",
    parameters=[datetime.date(2004,11,8), 4, "where string"]
)
```

At first sight the parameter substitution doesn't seem to offer much of an advantage but in fact it is extremely useful because `web.database` will automatically convert the values to SQL for you so that you don't need to convert them

yourself.

Note: Parameter substitution can be done for any value which needs conversion. This includes default values in `CREATE` statements and values in `INSERT` and `UPDATE` statements or `WHERE` clauses. Parameter substitutions are **not** available for strings which do not need conversions such as table names, column names etc.

The module also supports `executemany()`. This method does the same as `execute()` except it executes once for each sequence in the values parameter. For example:

```
cursor.executemany(
    sql="UPDATE Test SET dateColumn=?, numberColumn=? WHERE stringColumn=?",
    parameters=[
       [datetime.date(2004,11,8), 4, "string1"],
       [datetime.date(2004,11,8), 5, "string2"],
       [datetime.date(2004,11,8), 6, "string3"],
    ]
)
```

In `web.database` this is no more efficient than executing a number of normal `cursor.execute()` methods.

`web.database` also provides cursor abstraction methods which provide a functional interface to execute SQL. For example here we insert some values into a table.

```
cursor.insert(
    table   = 'testTable',
    columns = ['col1','col2'],
    values  = ['val1', 2],
)
```

Cursor abstraction methods exist for all the SQL commands supported by `web.database`. These are described later.

The `cursor()` method takes the following options and will return the appropriate cursor object:

`cursor`([*execute=True*], [*format='tuple'*], [*convert=True*], [*mode='portable'*])

The default values which the cursor abstraction methods will take for the values of *execute*, *format* and *convert* can be set using this method.

format This can be `'tuple'` to return the results as a tuples, `'text'` to return as text wrapped to 80 characters for display in a terminal, `'dict'` to return the results as dictionaries or `'object'` to return the results as result objects to be treated as dictionaries, tuples or via attribute access.

convert Convert the results to standard formats (should be `True` for most users)

execute Used in the cursor SQL methods. If `True` then rather than returning an SQL string, the methods execute the results

mode The default mode for the `execute()` method. Can be `'portable'` to use the SQL abstraction methods or `'direct'` to send the SQL directly to the underlying cursor.

1.4.7 Retrieving Results

Once you have executed a SELECT statement you will want to retrieve the results. This is done using the `cursor.fetchall()` method:

```
cursor.execute("SELECT * FROM Test")
results = cursor.fetchall()
```

The `results` variable will always contain a tuple of tuples of fields. If the query matched no rows, result will be `((),)`. If it matched one row it will be in the form `((col1, col2, col3, etc),)`. If it matched more than one it will be in the form `((col1, col2, col3, etc), (col1, col2, col3, etc), etc )`

You can print the results like this:

```
for row in cursor.fetchall():
    print "New Row"
    for field in row:
        print field
```

The `cursor.fetchall()` method will return the same results until another SQL query is executed using `cursor.execute()`.

1.4.8 Transactions, Rollbacks and Committing Changes

Most databases supported by `web.database` support basic transactions. This means that you can make a number of changes to the database but if your program crashes your changes will not be saved so that the database is not left in an unstable state where you have updated some tables but not others.

Changes are only saved (or committed) to the database when you call the `connection` object's `commit()` method:

```
connection.commit()
```

If you have made a mistake and want to lose all the changes you have made, you can rollback the database to its previous state using the `connection` object's `rollback()` method:

```
connection.rollback()
```

Finally, if you have finished using a connection you can close it using the `connection` object's `close()` method. This will also rollback the database to the time you last committed your changes so if you want to save your changes you should call `commit()` first.

```
connection.commit()
connection.close()
```

Note: Please note that making these changes to the `connection` object will automatically affect all `cursor` objects of that connection as well since they all share the same connection object.

Warning: The MySQL adapter does **not** support transactions. Results are automatically committed. If anyone can suggest an effective way around this please let me know!

1.4.9 Exporting and Importing SQL

For databases that are implemented entirely in portable mode you can export the SQL needed to entirely recreate the database using the `cursor.export()` method.

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

import sys; sys.path.append('../../../') # show python where the modules are

import web.database
connection = web.database.connect(
    adapter="snakesql",
    database="database-export",
    autoCreate = 1,
)
cursor = connection.cursor()
originalSQL = """CREATE TABLE People( LastName String PRIMARY KEY, FirstName String , Number In
CREATE TABLE Houses( House Integer, Owner String REQUIRED FOREIGN KEY=People)
INSERT INTO People ( LastName, FirstName, Number, DateOfBirth) VALUES ('Smith', 'John', 10, '19
INSERT INTO People ( LastName, FirstName, Number, DateOfBirth) VALUES ('Doe', 'James', 3, '1981
INSERT INTO Houses ( House, Owner ) VALUES (1, 'Smith')
INSERT INTO Houses ( House, Owner ) VALUES (2, 'Smith')
INSERT INTO Houses ( House, Owner ) VALUES (3, 'Doe')"""
for sql in sqls.split('\n'):
    cursor.execute(sql)
exportedSQL = cursor.export()
if exportedSQL == originalSQL:
    print "The exported SQL exaclty matches the original"
    print exportedSQL
else:
    print "The SQL is different"
    print exportedSQL

connection.close() # Close the connection without saving changes
```

Note: The SQL exported from the database may not exactly match the SQL which was used to create it since there is some redundancy in SQL syntax but the effect of the SQL will be the same.

Warning: This functionality is fairly new and hasn't yet had extensive testing so please be warned there may be issues.

To import the SQL generated by the `export()` method you can use the following code:

```
exportedSQL = cursor.export()
for sql in exportedSQL.split('\n'):
    cursor.execute(sql, mode='portable')
```

1.4.10 Using the Interactive Prompt

The `web.database` distribution comes with an Interactive Prompt which allows you to enter SQL queries directly into a database and see the results as a table of data. The 'sql.py' file is in the 'scripts' directory of the distribution.

Warning: Each time an SQL command is issued a new connection is made and the result of the SQL statement is committed. You cannot rollback changes made through the interactive prompt.

Starting the Prompt

To see a list of all the available options for starting an Interactive Prompt session, load a command prompt and type the following at the command line:

```
> python sql.py -h
```

This will display all the options available to you for using the interactive prompt. For example to connect to an SQLite database named 'test.db' you might use the following command:

```
> python sql.py -a sqlite -d test.db
```

This will run the interactive prompt. You should see something similar to this:

```
SQLite Interactive Prompt
Type SQL or "exit" to quit, "help", "copyright" or "license" for information.
sql>
```

It looks a bit like the Python prompt only allows SQL queries to be entered.

When you connect to a database using the SnakeSQL or SQLite adapters the database specified is automatically created if it doesn't already exist.

Options which are specific to the underlying database and which are not handled through any of the options listed by using `sql.py -h` can be entered using the `--more` switch and specifying a string containing a Python dictionary. For example:

```
> python sql.py -a sqlite -d test.db --more "{'customOption1':5, 'customOption2': 'value'}"
```

Using the Prompt

Try selecting some information form the table `testTable`:

```
sql> SELECT * FROM testTable
Error: Table 'testTable' not found.
sql>
```

Unsurprisingly, this gives an error message since we haven't yet created a table. All the supported commands, including creating a table will be demonstrated in the section SQL Reference.

If you are new to SQL you should read the SQL Reference and test the examples using the interactive prompt.

To exit the Interactive Prompt type `exit` and press Enter:

```
sql> exit

C:\Documents and Settings\James\Desktop\scripts>
```

You will be returned to the usual command prompt.

Some test SQL commands can be sound in the file 'web/external/PDBC/database/external/SnakeSQL/test/test.sql'. You can copy and paste the commands into the prompt and you should see output similar to that specified in the 'test.sql' file.

1.4.11 Special Characters

This section describes how to deal with special characters in Python and `web.database`.

In Python

Within a Python string, certain sequences have special meaning. Each of these sequences begins with a backslash `\`, known as the escape character. The values (and different escape methods) allowed in string literals are described in the Python documentation at http://www.python.org/doc/current/ref/strings.html. This is a brief summary.

Python recognizes the following escape sequences:

```
\\   Backslash (\)
\'  Single quote (')
\"  Double quote (")
\a  ASCII Bell (BEL)
\b  ASCII Backspace (BS)
\f  ASCII Formfeed (FF)
\n  ASCII Linefeed (LF)
\N{name}  Character named name in the Unicode database (Unicode only)
\r  ASCII Carriage Return (CR)
\t  ASCII Horizontal Tab (TAB)
\uxxxx  Character with 16-bit hex value xxxx (Unicode only)
\Uxxxxxxxx  Character with 32-bit hex value xxxxxxxx (Unicode only)
\v  ASCII Vertical Tab (VT)
\ooo  Character with octal value ooo
\xhh  Character with hex value hh
```

These sequences are case sensitive. For example, `\b` is interpreted as a backspace, but `\B` is not.

You can use these characters in SQL exactly the same way as you would in Python. For example `'end of one line\nstart of new line'` is a valid SQL string containing a line break in the middle and could be used like this:

```
cursor.execute("INSERT INTO table (columnOne) VALUES ('end of one line\nstart of new line')")
```

There is one important point to note about how Python (and hence `web.database`) deals with these escape characters. If a string contains a backslash `\` but the character after the backslash is not a character which can be escaped then the single backslash is treated as a single backslash. If the character can be used in an escape sequence then the backslash is treated as an escape character and the character is escaped.

Note: All examples in this section are from the Python prompt not the `web.database` one.

For example:

```
Python 2.4 (#60, Nov 30 2004, 11:49:19) [MSC v.1310 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print 'hello\%world'
hello\%world
>>> print 'hello\nworld'
hello
world
>>>
```

If a string contains both escaped and non-escaped characters Python guesses which are backslashes and which are escape characters:

```
>>> print 'hello\nworld\%again'
hello
world\%again
>>>
```

If a string contains a double backslash `\\` it is always treated as an escaped backslash character and printed as `\`.

```
>>> print '\\%'
\%
>>> print '\%'
\%
```

This means that the following expression is True:

```
>>> print '\\%' == '\%'
True
>>>
```

But the following is not:

```
>>> print '\\\\%' == '\\%'
False
>>>
```

When writing Python strings you have to be very careful how the backslash character is being used and then you will have no problems.

Interactive Prompt

The Interactive Prompt obeys the same special character rules as Python and SQL described above. One point which could cause confusion is the way the Interactive Prompt displays strings. If strings can be easily displayed they are. Otherwise the `repr()` function is used on them to explicitly display all their escape characters. This means all genuine backslashes appear double quoted.

```
Python 2.4 (#60, Nov 30 2004, 11:49:19) [MSC v.1310 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print repr('hello\nworld\%once\\more')
'hello\nworld\\%once\\more'
```

In SQL

In SQL all strings must be properly quoted using single quotes. To insert a string like `James` into the database, we use the SQL `'James'` but what if we want to insert the string `tail's`? Because it has a `'` character in it we can't simply do `'tail's'` as the SQL parser won't know which `'` ends the string. Instead we use `'tail''s'`. Double single quotes (`''`) in SQL mean a `'` character.

The single quote character `'` is the only character which needs special treatment in SQL all the others like `\n` behave exactly as they do in Python as described above.

For example:

```
cursor.execute("INSERT INTO table (columnOne) VALUES ('James''s')")
```

The Easy Way

If you are using the advanced cursor methods like `cursor.insert()` or `cursor.update()` (described later) or parameter substitution (described earlier), the easiest way to deal with special characters is to do nothing with them at all. The methods will automatically handle the conversions for you.

For example:

```
cursor.insert(
    table='table',
    columns=['columnOne'],
    values=["James's"],
)
```

or

```
cursor.execute("INSERT INTO table (columnOne) VALUES (?)", "James's")
```

If you want explicitly want to use the cursor methods like `cursor.insert()` or `cursor.update()` but with quoted SQL strings rather than having the conversions done automatically you can do so like this:

```
cursor.insert(
    table='table',
    columns=['columnOne'],
    __sqlValues=["'James''s'"],
)
```

1.4.12 SQL Reference

The SQL parser to parse `cursor.execute(sql, mode='portable')` statements has already been written and is available as a standalone module named `SQLParserTools`. The approach of parsing an SQL statement just to rebuild it again in an abstraction layer function might sound unnecessary but the advantage is that the SQL written in this manner is guaranteed to function in the same way across all `web.database` databases.

This specification implements what is considered the lowest possible useful SQL feature set which is commonly used and which all databases will support. A balance has had to be made between including useful features and excluding features which only some database engines support. Also no duplication of features has been included. For example `BETWEEN` can be implemented using `>` and `<` operators in the `WHERE` clause so has not been included but the `LIKE` operator has.

The specification includes:

SQL SELECT

SQL WHERE

SQL INSERT

SQL UPDATE

SQL DELETE

SQL ORDER BY

SQL AND & OR

Simple Joins

SQL CREATE

SQL DROP

NULL values

Database Tables A database most often contains one or more tables. Each table is identified by a name (e.g. `Customers` or `Orders`). Tables contain records (rows) with data.

Below is an example of a table called `Person`:

```
+----------+-----------+--------+-------------+
| LastName | FirstName | Number | DateOfBirth |
+----------+-----------+--------+-------------+
| Smith    | John      | 10     | 1980-01-01  |
+----------+-----------+--------+-------------+
| Doe      | John      | 3      | 1981-12-25  |
+----------+-----------+--------+-------------+
```

The table above contains two records (one for each person) and four columns (LastName, FirstName, Address, and DateOfBirth).

Queries With SQL, we can query a database and have a result set returned.

A query looks like this:

```
SELECT LastName FROM Person
```

Gives a result set like this:

```
+----------+
| LastName |
+----------+
| Smith    |
+----------+
| Doe      |
+----------+
```

Note: Some database systems require a semicolon at the end of the SQL statement. `web.database` does not.

The SELECT Statement

The SELECT statement is used to select data from a table. The tabular result is stored in a result table (called the result-set).

```
SELECT column_name(s) FROM table_name
```

Select Some Columns To select the columns named `LastName` and `FirstName`, use a SELECT statement like this:

SELECT LastName, FirstName FROM Person

Table Person:

```
+----------+-----------+--------+-------------+
| LastName | FirstName | Number | DateOfBirth |
+----------+-----------+--------+-------------+
| Smith    | John      | 10     | 1980-01-01  |
+----------+-----------+--------+-------------+
| Doe      | John      | 3      | 1981-12-25  |
+----------+-----------+--------+-------------+
```

Result Set:

```
+----------+-----------+
| LastName | FirstName |
+----------+-----------+
| Smith    | John      |
+----------+-----------+
| Doe      | John      |
+----------+-----------+
```

The order of the columns in the result is the same as the order of the columns in the query.

Select All Columns To select all columns from the `Person` table, use a `*` symbol instead of column names, like this:

```
SELECT * FROM Person
```

Result Set:

```
+----------+-----------+--------+-------------+
| LastName | FirstName | Number | DateOfBirth |
+----------+-----------+--------+-------------+
| Smith    | John      | 10     | 1980-01-01  |
+----------+-----------+--------+-------------+
| Doe      | John      | 3      | 1981-12-25  |
+----------+-----------+--------+-------------+
```

The WHERE Clause

The WHERE clause is used to specify a selection criterion.

The syntax of the where clause is:

```
SELECT column FROM table WHERE column operator value
```

With the WHERE clause, the following operators can be used:

```
Operator    Description
--------    -----------
=           Equal
<>          Not equal
>           Greater than
<           Less than
>=          Greater than or equal
<=          Less than or equal
LIKE        Pattern match (described later)
IS          Used for comparison to NULL
IS NOT      Used for comparison to NULL
```

In some versions of SQL the `<>` operator may be written as `!=` but not in `web.database`. Note that the equals operator in SQL is `=` not `==` as it is in Python.

The `=` and `<>` operators cannot be used to compare `NULL` values because a field cannot be equal to nothing. Instead the `IS` and `IS NOT` operators should be used.

Using the WHERE Clause To select only the people whose last name are `Smith`, we add a WHERE clause to the SELECT statement:

```
SELECT * FROM Person WHERE LastName='Smith'
```

Person table:

```
+----------+-----------+--------+-------------+
| LastName | FirstName | Number | DateOfBirth |
+----------+-----------+--------+-------------+
| 'Smith'  | 'John'    | 10     | 1980-01-01  |
| 'Doe'    | 'John'    | 3      | 1981-12-25  |
+----------+-----------+--------+-------------+
```

Result set:

```
+----------+-----------+--------+-------------+
| LastName | FirstName | Number | DateOfBirth |
+----------+-----------+--------+-------------+
| 'Smith'  | 'John'    | 10     | 1980-01-01  |
+----------+-----------+--------+-------------+
```

Using Quotes Note that we have used single quotes around the conditional values in the examples.

SQL uses single quotes around text values (some database systems will also accept double quotes, not `web.database`). Numeric values should not be enclosed in quotes.

For text values:

This is correct:

```
SELECT * FROM Person WHERE LastName='Smith'
```

This is wrong:

```
SELECT * FROM Person WHERE LastName=Smith
```

For numeric values:

This is correct:

```
SELECT * FROM Person WHERE Number>10
```

This is wrong:

```
SELECT * FROM Person WHERE Number>'10'
```

The LIKE Condition The `LIKE` condition is used to specify a search for a pattern in a column.

```
SELECT column FROM table WHERE column LIKE pattern
```

A % sign can be used to define wildcards (missing letters in the pattern).

The following SQL statement will return people with first names that start with an 'O':

```
SELECT * FROM Person WHERE FirstName LIKE 'O%'
```

The following SQL statement will return people with first names that end with an 'a':

```
SELECT column FROM table WHERE FirstName LIKE '%a'
```

The following SQL statement will return people with first names that contain the pattern 'la':

```
SELECT column FROM table WHERE FirstName LIKE '%la%'
```

You can use as many % characters as you need in the pattern to match zero or more characters. If you need to have an actual % characters in the pattern you will need to escape it like this
%.

The following SQL statement will return values that end with a % character.

```
SELECT column FROM table WHERE Percentage LIKE '%\%'
```

`web.database` does not support the `BETWEEN` condition since the same thing can be achieved using comparison operators.

The INSERT INTO Statement

The INSERT INTO statement is used to insert new rows into a table.

Syntax

```
INSERT INTO table_name (column1, column2,...) VALUES (value1, value2,....)
```

Insert a New Row This Person table:

```
+----------+-----------+--------+-------------+
| LastName | FirstName | Number | DateOfBirth |
+----------+-----------+--------+-------------+
| 'Smith'  | 'John'    | 10     | 1980-01-01  |
| 'Doe'    | 'John'    | 3      | 1981-12-25  |
+----------+-----------+--------+-------------+
```

And this SQL statement:

```
INSERT INTO Person (LastName, FirstName, Number, DateOfBirth)
VALUES ('Blair', 'Tony', 8, '1953-05-06')
```

Note: `web.database` expects the SQL to all be on one line. The line break here is for formatting

Will give this result:

```
+----------+-----------+--------+-------------+
| LastName | FirstName | Number | DateOfBirth |
+----------+-----------+--------+-------------+
| 'Smith'  | 'John'    | 10     | 1980-01-01  |
| 'Doe'    | 'John'    | 3      | 1981-12-25  |
| 'Blair'  | 'Tony'    | 8      | 1953-05-06  |
+----------+-----------+--------+-------------+
```

If you are extremely careful, the column names can be omitted as long as the values are specified in the same order as the columns when the table was created.

The SQL below would achieve the same result as the previous SQL statement:

```
INSERT INTO Person VALUES ('Blair', 'Tony', 8, '1953-05-06')
```

Warning: It is very easy to make a mistake with the shortened syntax so it is recommended you use the full version and specify the column names.

The UPDATE Statement

The UPDATE statement is used to modify the data in a table.

Syntax:

```
UPDATE table_name SET column_name = new_value WHERE column_name = some_value
```

Update one Column in a Row

Person table

```
+----------+-----------+--------+-------------+
| LastName | FirstName | Number | DateOfBirth |
+----------+-----------+--------+-------------+
| 'Smith'  | 'John'    | 10     | 1980-01-01  |
| 'Doe'    | 'John'    | 3      | 1981-12-25  |
| 'Blair'  | 'Tony'    | 8      | 1953-05-06  |
+----------+-----------+--------+-------------+
```

We want to add a change Tony Blair's first name to `James`:

```
UPDATE Person SET FirstName = 'James' WHERE LastName = 'Blair'
```

Person table

```
+----------+-----------+--------+-------------+
| LastName | FirstName | Number | DateOfBirth |
+----------+-----------+--------+-------------+
| 'Smith'  | 'John'    | 10     | 1980-01-01  |
| 'Doe'    | 'John'    | 3      | 1981-12-25  |
| 'Blair'  | 'James'   | 8      | 1953-05-06  |
+----------+-----------+--------+-------------+
```

Update several Columns in a Row We want to change the number of everyone with a FirstName `John` and make their DateOfBirth all `1980-01-01`:

```
UPDATE Person SET Number = 1, DateOfBirth = '1980-01-01' WHERE FirstName = 'John'
```

Result:

```
+----------+-----------+--------+-------------+
| LastName | FirstName | Number | DateOfBirth |
+----------+-----------+--------+-------------+
| 'Smith'  | 'John'    | 1      | 1980-01-01  |
| 'Doe'    | 'John'    | 1      | 1980-01-01  |
| 'Blair'  | 'James'   | 8      | 1953-05-06  |
+----------+-----------+--------+-------------+
```

The DELETE Statement

The DELETE statement is used to delete rows in a table.

Syntax

```
DELETE FROM table_name
WHERE column_name = some_value
```

Delete a Row Person:

```
+----------+-----------+--------+-------------+
| LastName | FirstName | Number | DateOfBirth |
+----------+-----------+--------+-------------+
| 'Smith'  | 'John'    | 1      | 1980-01-01  |
| 'Doe'    | 'John'    | 1      | 1980-01-01  |
| 'Blair'  | 'James'   | 8      | 1953-05-06  |
+----------+-----------+--------+-------------+
```

`John Doe` is going to be deleted:

```
DELETE FROM Person WHERE LastName = 'Doe'
```

Result

```
+----------+-----------+--------+-------------+
| LastName | FirstName | Number | DateOfBirth |
+----------+-----------+--------+-------------+
| 'Smith'  | 'John'    | 1      | 1980-01-01  |
| 'Blair'  | 'James'   | 8      | 1953-05-06  |
+----------+-----------+--------+-------------+
```

Delete All Rows It is possible to delete all rows in a table without deleting the table. This means that the table structure and attributes will be intact:

```
DELETE FROM table_name
```

Result

```
+----------+-----------+--------+-------------+
| LastName | FirstName | Number | DateOfBirth |
+----------+-----------+--------+-------------+
+----------+-----------+--------+-------------+
```

ORDER BY

The ORDER BY keyword is used to sort the result.

Sort the Rows The ORDER BY clause is used to sort the rows.

Orders:

```
+-------------+-------------+
| Company     | OrderNumber |
+-------------+-------------+
| 'Asda'      | 5678        |
| 'Morrisons' | 1234        |
| 'Tesco'     | 2345        |
| 'Morrisons' | 7654        |
+-------------+-------------+
```

To display the companies in alphabetical order:

```
SELECT Company, OrderNumber FROM Orders ORDER BY Company
```

Result:

```
+-------------+-------------+
| Company     | OrderNumber |
+-------------+-------------+
| 'Asda'      | 5678        |
| 'Morrisons' | 1234        |
| 'Morrisons' | 7654        |
| 'Tesco'     | 2345        |
+-------------+-------------+
```

Example

To display the companies in alphabetical order AND the order numbers in numerical order:

```
SELECT Company, OrderNumber FROM Orders ORDER BY Company, OrderNumber
```

Result:

```
+-------------+-------------+
| Company     | OrderNumber |
+-------------+-------------+
| 'Asda'      | 5678        |
| 'Morrisons' | 1234        |
| 'Morrisons' | 7654        |
| 'Tesco'     | 2345        |
+-------------+-------------+
```

Example

To display the companies in reverse alphabetical order:

```
SELECT Company, OrderNumber FROM Orders ORDER BY Company DESC
```

Result:

```
+-------------+-------------+
| Company     | OrderNumber |
+-------------+-------------+
| 'Tesco'     | 2345        |
| 'Morrisons' | 1234        |
| 'Morrisons' | 7654        |
| 'Asda'      | 5678        |
+-------------+-------------+
```

Example

To display the companies in alphabetical order AND the order numbers in reverse numerical order:

```
SELECT Company, OrderNumber FROM Orders ORDER BY Company ASC, OrderNumber DESC
```

Result:

```
+-------------+-------------+
| Company     | OrderNumber |
+-------------+-------------+
| 'Asda'      | 5678        |
| 'Morrisons' | 7654        |
| 'Morrisons' | 1234        |
| 'Tesco'     | 2345        |
+-------------+-------------+
```

AND & OR

AND and OR join two or more conditions in a WHERE clause.

The AND operator displays a row if ALL conditions listed are true. The OR operator displays a row if ANY of the conditions listed are true.

Original Table (used in the examples)

```
+----------+-----------+--------+-------------+
| LastName | FirstName | Number | DateOfBirth |
+----------+-----------+--------+-------------+
| 'Smith'  | 'John'    | 1      | 1980-01-01  |
| 'Doe'    | 'John'    | 1      | 1980-01-01  |
| 'Blair'  | 'James'   | 8      | 1953-05-06  |
+----------+-----------+--------+-------------+
```

Use AND to display each person with the first name equal to `John`, and the last name equal to `Smith`:

```
SELECT * FROM Person WHERE FirstName='John' AND LastName='Smith'
```

Result Set

```
+----------+-----------+--------+-------------+
| LastName | FirstName | Number | DateOfBirth |
+----------+-----------+--------+-------------+
| 'Smith'  | 'John'    | 1      | 1980-01-01  |
+----------+-----------+--------+-------------+
```

Use OR to display each person with the first name equal to `James`, or the last name equal to `Smith`:

```
SELECT * FROM Person WHERE FirstName='James' OR LastName='Smith'
```

Result Set

```
+----------+-----------+--------+-------------+
| LastName | FirstName | Number | DateOfBirth |
+----------+-----------+--------+-------------+
| 'Smith'  | 'John'    | 1      | 1980-01-01  |
| 'Blair'  | 'James'   | 8      | 1953-05-06  |
+----------+-----------+--------+-------------+
```

Example

You can also combine AND and OR use parentheses to form complex expressions:

```
SELECT * FROM Person WHERE (FirstName='James' AND LastName='Smith') OR LastName='Blair'
```

Result Set

```
+----------+-----------+--------+-------------+
| LastName | FirstName | Number | DateOfBirth |
+----------+-----------+--------+-------------+
| 'Blair'  | 'James'   | 8      | 1953-05-06  |
+----------+-----------+--------+-------------+
```

NULL Values

An important feature of `web.database` is its ability to support `NULL` values. A field which contains a `NULL` value is simply a field where no value has been set or the value as been set to contain no value. This is quite different, for example, from a `String` field which has been set a value `''`, an empty string.

Original Table (used in the examples)

```
+----------+-----------+--------+-------------+
| LastName | FirstName | Number | DateOfBirth |
+----------+-----------+--------+-------------+
| 'Smith'  | 'John'    | 1      | 1980-01-01  |
| 'Doe'    | 'John'    | 1      | 1980-01-01  |
| 'Blair'  | 'James'   | 8      | 1953-05-06  |
+----------+-----------+--------+-------------+
```

Our query

```
UPDATE Person SET FirstName=NULL WHERE LastName='Doe'
```

Our table now looks like this:

```
+----------+-----------+--------+-------------+
| LastName | FirstName | Number | DateOfBirth |
+----------+-----------+--------+-------------+
| 'Smith'  | 'John'    | 1      | 1980-01-01  |
| 'Doe'    | NULL      | 1      | 1980-01-01  |
| 'Blair'  | 'James'   | 8      | 1953-05-06  |
+----------+-----------+--------+-------------+
```

This is quite different from this query which simply sets the FirstName to the string `'NULL'` not the value `NULL`:

```
UPDATE Person SET FirstName='NULL' WHERE FirstName IS NULL
```

Our table now looks like this:

```
+----------+-----------+--------+-------------+
| LastName | FirstName | Number | DateOfBirth |
+----------+-----------+--------+-------------+
| 'Smith'  | 'John'    | 1      | 1980-01-01  |
| 'Doe'    | 'NULL'    | 1      | 1980-01-01  |
| 'Blair'  | 'James'   | 8      | 1953-05-06  |
+----------+-----------+--------+-------------+
```

This is one of the reasons why it is important to use the correct quotations around values in you SQL.

Note: We use the `IS` operator rather than the `=` operator to compare fields to `NULL` values.

If you inserted a row into the table without specifying all the columns the columns you had not specified would contain the value `NULL` unless you had specified a `DEFAULT` value when you created the table.

CREATE

To create a table in a database:

Syntax

```
CREATE TABLE table_name
(
column_name1 data_type options,
column_name2 data_type options,
.......
)
```

Example

This example demonstrates how you can create a table named `Person`, with four columns. The column names will be `LastName`, `FirstName`, `Number`, and `DateOfBirth`:

```
CREATE TABLE Person (LastName String, FirstName String, Number String, DateOfBirth Date)
```

The data_type specifies what type of data the column can hold. The table below contains the data types supported by `web.database`:

Type	Description
`Bool`	True or False
`Integer`	Any Python integer (not Python Long or Decimal)
`Long`	Any Python long integer between -9223372036854775808 and 9223372036854775807
`Float`	Any Python floating point number
`String`	A string of 255 characters or less (Not unicode?) [a]
`Text`	A 24-bit string [b]
`Binary`	A 24-bit binary string [b]]
`Date`	Any valid Python `datetime.date` object. Takes values in the form of python `datetime` objects. Only stores days
`Time`	Any valid Python `datetime.time` object. Takes values in the form of python `datetime` objects. Only stores hour
`Datetime`	Any valid Python `datetime.datetime` object. Takes values in the form of python `datetime` objects. Only stores

[a] Some databases make a distinction between short strings (often named VARCHAR) and long strings (often TEXT). Short string fields are normally faster and so a distinction is also made in this specification.

[b] Although Python supports strings of greater than 24 bit, a lot of databases do not and so in order to be compatible with those databases Binary and String objects should be no longer than 24 bit.

[c] Although Python ¡ 2.3 does not support datetime objects, pure Python compatible libraries exist for Python ¡ 2.3 and these can be used instead so it makes sense to use the standard Python types where possible. The options can be used to further specify what values the field can take. They are described in the next sections.

REQUIRED In `web.database`, `REQUIRED` simply means that the field cannot contain a `NULL` value. If you insert a row into a table with a `REQUIRED` field, you must specify a value for the field unless you have also specified the field to have a `DEFAULT` value which is not `NULL` in which case the default value will be used. If you try to set the field to `NULL` an error will be raised.

To create a table with `LastName` and `FirstName` columns where `LastName` could not take a `NULL` value you would use:

```
CREATE TABLE Person (LastName String REQUIRED, FirstName String)
```

UNIQUE In `web.database`, a `UNIQUE` field is one in which all values in the table must be different. An error occurs if you try to add a new row with a value that matches an existing row. The exception to this is that if a column is not specified as `REQUIRED`, i.e. it is allowed to contain `NULL` values, it can contain multiple `NULL` values.

To create a table with `LastName` and `FirstName` columns where all the values of `LastName` had to be different or `NULL` you would use:

```
CREATE TABLE Person (LastName String UNIQUE, FirstName String)
```

If a field is specified as `UNIQUE`, `web.database` will not also let you specify a `DEFAULT` value.

Bool, Float, Text and Binary fields cannot be unique.

PRIMARY KEY `PRIMARY KEY` columns are unique and cannot take `NULL` values. Each table can only have one field specified as `PRIMARY KEY`.

Primary keys can sometimes be used by `web.database`'s drivers to speed up database queries. A `PRIMARY KEY` column is a column where the value is used to uniquely identify the row.

To create a table with `LastName` and `FirstName` columns where `LastName` is a primary key use:

```
CREATE TABLE Person (LastName String PRIMARY KEY, FirstName String)
```

Bool, Float, Text and Binary fields cannot be primary keys.

DEFAULT The `DEFAULT` option is used to specify a default value for a field to be used if a value is not specified when a new row is added to a table.

To create a table with `LastName` and `FirstName` columns where the default value for `LastName` is `'Smith'` we would use:

```
CREATE TABLE Person (LastName String DEFAULT='Smith', FirstName String)
```

You cannot specify a `DEFAULT` if the column is a `PRIMARY KEY` or `UNIQUE`.

If no `DEFAULT` is specified the `DEFAULT` is `NULL`.

Binary and Text fields cannot have default values.

FOREIGN KEY The final option is `FOREIGN KEY`. If a column is specified `FOREIGN KEY` it cannot have any other options. The table specified as providing the foreign key must have a primary key. It is the primary key value which is used as a foreign key in the other table.

For example:

```
CREATE TABLE Houses (House Integer, Owner String FOREIGN KEY=People)
```

Bool, Float, Text and Binary fields cannot be foreign key fields.

Foreign keys are described in more detail in the section on joins.

DROP Table

Delete a Table To delete a table (the table structure and attributes will also be deleted):

```
DROP TABLE table_name
```

Note: If you are using foreign key constraints you cannot drop a parent table if the child table still exists you should drop the child table first.

If you want to drop more than one table you can use this alternative syntax:

```
DROP TABLE table1, table2, table3
```

FOREIGN KEY and Joins

Sometimes we have to select data from two or more tables to make our result complete. We have to perform a join. Joins and the use of primary and foreign keys are inter-related.

FOREIGN KEY Tables in a database can be related to each other with keys. A primary key is a column with a unique value for each row. The purpose is to bind data together, across tables, without repeating all of the data in every table.

In the `People` table below, the `LastName` column is the primary key, meaning that no two rows can have the same `LastName`. The `LastName` distinguishes two persons even if they have the same name.

When you look at the example tables below, notice that:

- The `LastName` column is the primary key of the `People` table
- The `House` column is the primary key of the `Houses` table
- The `Owner` column in the `House` table is used to refer to the people in the `People` table. `Owner` is a foreign key field.

```
People
+----------+-----------+--------+-------------+
| LastName | FirstName | Number | DateOfBirth |
+----------+-----------+--------+-------------+
| Smith    | John      | 10     | 1980-01-01  |
+----------+-----------+--------+-------------+
| Doe      | James     | 3      | 1981-12-25  |
+----------+-----------+--------+-------------+

Houses
+-------+-------+
| House | Owner |
+-------+-------+
| 1     | Smith |
+-------+-------+
| 2     | Smith |
+-------+-------+
| 3     | Doe   |
+-------+-------+
```

People may own more than one house. In our example John Smith owns both House `1` and `2`. In order to keep the database consistent you would not want to remove `Smith` from the `People` table or drop the `People` table because the `Houses` table would still contain a reference to `Smith`. Similarly you wouldn't want to insert or update a value in the `Owner` column of the `Houses` table which didn't exist as a primary key for the `People` table.

By specifying the `Owner` column of the `Houses` table as a foregin key these constraints are enforced by `web.database`.

The SQL for the tables is below. **Note:** The line breaks in the first `CREATE` statement are for formatting; `web.database` doesn't support line breaks in SQL.

```
CREATE TABLE People (
    LastName String PRIMARY KEY, FirstName String,
    Number Integer, DateOfBirth Date
)
CREATE TABLE Houses (House Integer, Owner String FOREIGN KEY=People)
```

If a column is specified `FOREIGN KEY` it cannot have any other options. The table specified as providing the foreign key must have a primary key. It is the primary key value which is used as a foreign key in the other table.

Bool, Float, Text and Binary fields cannot be foreign key fields.

We can select data from two tables by referring to two tables, using the SQL below. **Note:** The line breaks are just for formatting; `web.database` doesn't support line breaks in SQL.

```
SELECT Houses.House, People.FirstName, Houses.Owner
FROM People, Houses
WHERE People.LastName=Houses.Owner
```

Here is the result

```
+--------------+------------------+--------------+
| Houses.House | People.FirstName | Houses.Owner |
+--------------+------------------+--------------+
| 1            | 'John'           | 'Smith'      |
| 2            | 'John'           | 'Smith'      |
| 3            | 'James'          | 'Doe'        |
+--------------+------------------+--------------+
```

and another example:

```
SELECT Houses.House, People.FirstName, Houses.Owner
FROM People, Houses
WHERE People.LastName=Houses.Owner  and People.DateOfBirth<'1981-01-01'
```

Here is the result

```
+--------------+------------------+--------------+
| Houses.House | People.FirstName | Houses.Owner |
+--------------+------------------+--------------+
| 1            | 'John'           | 'Smith'      |
| 2            | 'John'           | 'Smith'      |
+--------------+------------------+--------------+
```

1.4.13 Cursor Abstraction Methods

This section describes how to use the following SQL methods of the `cursor` object:

`select()`, `insert()`, `update()`, `delete()`, `create()`, `alter()`, `drop()`, `function()`

These functions are designed to reflect the SQL syntax you would use if you were writing the SQL directly. For example you might write:

```
SELECT fieldName FROM tableName
INSERT INTO tableName value1, value2
```

Accordingly the `select()` and `insert()` methods accept the *fields* and *table* parameters in a different order. It is reccomended however that you always specify parameters by name rather than relying on their order as future versions may have different parameters in different places.

See Also:

w3schools SQL Tutorial
(http://www.w3schools.com/sql/default.asp)
A good introduction to SQL commands can be found on the w3schools website at http://www.w3schools.com/sql/default.asp.

Selecting Data

`select`(*tables, columns,* [*values=[],*][*where=None,*][*order=None,*][*execute=None,*][*fetch=None,*][***params*])
Build an SQL string according to the options specified and optionally execute the SQL and return the results in the format specified. No error checking on field names if the SQL string is only being built. Strict error checking is only performed when executing the code.

tables A string containing the name of the table to select from or if selecting from multiple tables, a sequence of table names.

columns A sequence of column names to select. Can be a string if only one column is being selected. If selecting from multiple tables, all column names should be in the form `'tableName.columnName'`

values A list of values to substitute for `?` in the `WHERE` clause specified by *where*.

where The WHERE clause as a `web.database` list as returned by `cursor.where()`. If *where* is a string it is converted to the correct format.

order The ORDER BY clause as a `web.database` list as returned by `cursor.order()`. If *order* is a string it is converted to the correct format.

execute If `False` the method returns the SQL string needed to perform the desired operations. If `True` the SQL is executed and the results converted and returned in the appropriate form. If not specified takes the value specified in the cursor which by default is `True`

fetch Whether or not to fetch the results. If `True` and *execute* is not specified *execute* is set to `True`. If `True` and *execute* `False` an error is raised.

*****params*** The parameters to be passed to the `fetchall()` method if *fetch* is `True`

To select some information from a database using an SQL string you would use the following command:

```
SELECT column_name(s) FROM table_name
```

For example consider the table below:

```
# Table Person
+----------+-----------+---------+-------------+
| LastName | FirstName | Address | DateOfBirth |
+----------+-----------+---------+-------------+
| Smith    | John      | Bedford | 1980-01-01  |
+----------+-----------+---------+-------------+
| Doe      | John      | Oxford  | 1981-12-25  |
+----------+-----------+---------+-------------+
```

To retrieve a list of the surnames and dates of birth of all the people in the table you would use the following code:

```
rows = cursor.select(
    columns = ['LastName', 'DateOfBirth'],
    tables  = ['Person'],
    format  = 'object',
)
```

Note: If you have specified *fetch* as `False` in the cursor constructor you would need to specify *fetch* as `True` here to fetch the results, otherwise you would need to use `rows = cursor.fetchall()` to actually fetch the results.

Since we have specified *format* as `'object'`, the result from this call would be a tuple of TupleDescriptor objects which can be treated as a tuple or a dictionary:

```
>>> for record in rows:
...    print record[0], record[1]
...    print record['LastName'], record['DateOfBirth']
...
Smith 1980-01-01
Smith 1980-01-01
Doe 1981-12-25
Doe 1981-12-25
```

Using the `select()` method, information you select from a field is automatically converted to the correct Python type. Integer fields return Integers, Date fields return `datetime.date` objects.

The where Parameter The example above selected every `LastName` and `DateOfBirth` field from the table. To limit the information selected you need to specify the `where` parameter in the same way you would for any SQL query.

```
>>> rows=cursor.select(columns=['LastName'],tables=['Person'],where="LastName='Smith'")
>>> for record in rows:
...    print record['LastName'], record['DateOfBirth']
...
'Smith'
```

We had to specify the value `Smith` as properly encoded SQL since we specified the where clause as a string. Alternatively we could have used the `cursor.where()` method to help instead.

`where`(*where,* [*values=[]*])

Return a parsed `WHERE` clause suitable for use in the `select()`, `update()` and `delete()` methods of the `cursor` object.

whereA string containing the `WHERE` clause. Can include the `LIKE` operator which is used as follows:

```
WHERE columnName LIKE %s1%s2%s
```

Every % sign is matched against zero or more characters.

Note: *where* should not include the string `'WHERE'` at the beginning.

valuesA list of values to substitute for `?` parameters in the `WHERE` clause

More complex expressions can also be built into where clauses. See the SQL Reference section for full information.

The order Parameter You can specify the order in which the results are sorted using the `order` parameter. It is used as follows:

```
>>> for record in cursor.select('LastName', 'Person', order="'LastName'"):
...    print record['LastName']
...
'Doe'
'Smith'
>>> for record in cursor.select('LastName', 'Person', order="LastName DESC"):
...    print record['LastName']
...
'Smith'
'Doe'
```

Note that by placing the word `DESC` after the column to order by, the order is reversed.

You can place a number of Columns after each other. For example `order="LastName DESC DateOfBirth"` could be used to order the results in decending order by `LastName` and if any results have the same last name, order them by `DateOfBirth`.

Alternatively we could have used the `cursor.order()` method to help instead.

`order`(*order*)

Return a parsed `ORDER BY` clause suitable for use in the `select()` method of the `cursor` object.

order A string containing the `ORDER BY` clause. **Note:** *order* should not include the string `'ORDER BY'` at the beginning.

Disabling Execute If you do not want the SQL to actually be executed you can set the `execute` parameter of the `select()` method to `False`. You can then manually execute it using `cursor.execute()`.

```
>>> sql = cursor.select(columns=['LastName', 'DateOfBirth'], tables=['Person'], execute=False)
>>> sql
'SELECT LastName, DateOfBirth FROM Person'
>>> cursor.execute(sql)
>>> cursor.fetchall()
(('Smith','1980-01-01'),('Doe','1981-12-25'))
```

Using Joins The `select()` allows you to select information from multiple tables. In order to do this you must specify the tables you wish to select from as a list or tuple and use the fully qualified column name for each table you want to column you want to select from.

For example:

```
>>> rows = cursor.select(
...     columns = ['table1.LastName', 'table2.Surname'],
...     tables  = ['table1','table2'],
...     where   = "table1.Surname = table2.Surname",
...     format  = 'dict',
... )
>>> print rows[0]['table2.Surname']
'Smith'
```

Inserting Data

The insert method looks like this:

The `insert()` method of a `web.database` cursor looks like this:

`insert`(*table, columns, values, __sqlValues,* [*execute*])

Insert values into the columns in table. Either *values* or *__sqlValues* can be specified but not both.

tableThe name of the table to insert into

columnsA sequence of column names in the same order as the values which are going to be inserted into those columns. Can be a string if only one column is going to have values inserted

valuesA sequence of Python values to be inserted into the columns named in the *columns* variable. Can be the value rather than a list if there is only one value. If *values* is specified then *__sqlValues* must be either an empty sequence or contain a list of all quoted SQL strings for the columns specified in which case *values* contains the Python values of the SQL strings to be substituted for `?` parameters in the *__sqlValues* sequence.

__sqlValuesA sequence of quoted SQL strings to be inserted into the columns named in the *columns* variable. Can be the value rather than a list if there is only one value. If *__sqlValues* is specified and contains `?` parameters for substitution then *values* contains the values to be substituted. Otherwise *values* must be an empty sequence.

executeIf `False` the method returns the SQL string to perform the desired operations. If `True` the SQL is executed. If not specified takes the value specified in the cursor which by default is `True`

To insert data into a table using SQL you would use the following command:

```
INSERT INTO table_name (column1, column2,...)
VALUES (value1, value2,....)
```

For example consider the table used to demonstrate the `select()` method:

```
+----------+-----------+---------+-------------+
| LastName | FirstName | Address | DateOfBirth |
+----------+-----------+---------+-------------+
```

The SQL command to insert some information into the table might look like this:

```
INSERT INTO Person (LastName, FirstName, Address, Age)
VALUES ('Smith', 'John', '5 Friendly Place', '1980-01-01')
```

To insert the data using a `web.database` cursor we would do the following:

```
cursor.insert(
    table   = 'Person',
    columns = ['LastName', 'FirstName', 'Address', 'DateOfBirth'],
    values  = ['Smith', 'John', '5 Friendly Place', datetime.date(1980,1,1)],
)
```

Note: We specify the field values as real Python objects. The date was specified as a `date` object and was automatically converted. Python 2.2 users can also use `import datetime` if they have first used `import web` as the web modules come with a compatibility module.

The table now looks like this:

```
+----------+-----------+------------------+-------------+
| LastName | FirstName | Address          | DateOfBirth |
+----------+-----------+------------------+-------------+
| Smith    | John      | 5 Friendly Place | 1980-01-01  |
+----------+-----------+------------------+-------------+
```

Updating Data

For example consider the table we created earlier:

```
# table Person
+----------+-----------+------------------+-------------+
| LastName | FirstName | Address          | DateOfBirth |
+----------+-----------+------------------+-------------+
| Smith    | John      | 5 Friendly Place | 1980-01-01  |
+----------+-----------+------------------+-------------+
```

The SQL command to change every address in the table to '6 London Road' is:

```
UPDATE Person SET Address = '6 London Road'
```

To update the data using a `web.database` cursor we would do the following:

```
cursor.update(table='Person',columns=['Address'],values=['6 London Road'])
```

The table now looks like this:

```
+----------+-----------+---------------+-------------+
| LastName | FirstName | Address       | DateOfBirth |
+----------+-----------+---------------+-------------+
| Smith    | John      | 6 London Road | 1980-01-01  |
+----------+-----------+---------------+-------------+
```

The `update()` method of a `web.database` cursor looks like this:

`update`(*table, columns, values, __sqlValues* [, *where*] [, *execute*])

Update the columns in table with the values. Either *values* or *_sqlValues* can be specified but not both.

tableA string containing the name of the table to update

columnsA sequence of column names in the same order as the values which are going to be updated in those columns. Can be a string if only one column is going to have values inserted

valuesA sequence of Python values to be inserted into the columns named in the *columns* variable. Can be the value rather than a list if there is only one value. If *values* is specified then *_sqlValues* must be either an empty sequence or contain a list of all quoted SQL strings for the columns specified in which case *values* contains the Python values of the SQL strings to be substituted for ? parameters in the *_sqlValues*

sequence. If there are more values specified in *values* than *_sqlValues* the remaining values are used to substitute for `?` parameters in *where*.

_sqlValues A sequence of quoted SQL strings to be inserted into the columns named in the *columns* variable. Can be the value rather than a list if there is only one value. If *_sqlValues* is specified and contains `?` parameters for substitution then *values* contains the values to be substituted. Otherwise *values* must be an empty sequence.

where The WHERE clause as a `web.database` list as returned by `cursor.where()`. If *where* is a string it is converted to the correct format.

execute If `False` the method returns the SQL string to perform the desired operations. If `True` the SQL is executed. If not specified takes the value specified in the cursor which by default is `True`

Deleting Data

For example consider the table we created earlier:

```
# table Person
+----------+-----------+------------------+-------------+
| LastName | FirstName | Address          | DateOfBirth |
+----------+-----------+------------------+-------------+
| Smith    | John      | 5 Friendly Place | 1980-01-01  |
| Owen     | Jones     | 4 Great Corner   | 1990-01-01  |
+----------+-----------+------------------+-------------+
```

The SQL command to delete every address in the table is:

```
DELETE FROM Person
```

To delete all the data using a `web.database` cursor we would do the following:

```
cursor.delete(table="Person")
```

Note: This does not delete the table, it deletes all the data. To drop the table use the `drop()` method.

The table now looks like this:

```
+----------+-----------+---------+-------------+
| LastName | FirstName | Address | DateOfBirth |
+----------+-----------+---------+-------------+
+----------+-----------+---------+-------------+
```

To delete only some of the data you need to specify the *where* parameter. For example to delete all people with the first name `'Owen'` we would use the SQL:

```
DELETE FROM Person WHERE FirstName='Owen'
```

Similarly the function to use to execute this SQL command is:

```
cursor.delete(table="Person", where="FirstName='Owen'")
```

The table now looks like this:

```
+----------+-----------+------------------+-------------+
| LastName | FirstName | Address          | DateOfBirth |
+----------+-----------+------------------+-------------+
| Smith    | John      | 5 Friendly Place | 1980-01-01  |
+----------+-----------+------------------+-------------+
```

The `delete()` method of a `web.database` cursor looks like this:

`delete`(*table*, [*values=[]*][, *where*] [, *execute*])

Delete records from the table according to *where*.

tableA string containing the name of the table to select from or if selecting from multiple tables, a sequence of table names.

valuesA list of values to substitute for `?` in the `WHERE` clause specified by *where*.

whereThe WHERE clause as a `web.database` list as returned by `cursor.where()`. If *where* is a string it is converted to the correct format.

executeIf `False` the method returns the SQL string to perform the desired operations. If `True` the SQL is executed. If not specified takes the value specified in the cursor which by default is `True`

Creating Tables

To create a table in SQL you would use the following command:

```
CREATE TABLE table_name
(
    column_name1 data_type,
    column_name2 data_type,
    etc...
)
```

For example:

```
CREATE TABLE Person
(
    LastName varchar,
    FirstName varchar,
    Address varchar,
    Age int
)
```

To create the table above using a `web.database` cursor we would use the `cursor.column()` helper method:

`column`(*name*, *type*[, *required=0*][, *unique=0*][, *primaryKey=0*][, *foreignKey=None*][, *default=None*])

Return a column tuple suitable for use in the columns tuple used in the `create()` method.

`Bool`, `Float`, `Binary` and `Text` columns cannot be used as primary keys or unique columns. `Binary` and `Text` columns cannot have default values.

name The name of the field as a string.

type The field type. This can take one of the values: `'Bool'`, `'String'`, `'Text'`, `'Binary'`, `'Long'`, `'Integer'`, `'Float'`, `'Date'`, `'Time'`, `'Datetime'`

required Whether or not the field is required. Setting to `True` means the field cannot have `NULL` values.

unique Set to `True` if the value must be unique. Two fields in the column cannot have the same value unless that value is NULL

primaryKey The field is to be used as a primary key, the field has the same behaviour as being unique and required but no default value can be set

foreignKey The field is to be used as a foreign key, the value should be the name of the table for which this is a child table. **Note:** There is no need to specify the column name as tables can only have one primary key.

default The default value for the field to be set to. If not specified the default is NULL

For example:

```
cursor.create(
    table = 'Person',
    columns = [
        cursor.column(name='LastName',  type='String' ),
        cursor.column(name='FirstName', type='String' ),
        cursor.column(name='Address',   type='String' ),
        cursor.column(name='Age',       type='Integer'),
    ],
)
```

The `create()` method takes the table name as the first argument and then a sequence column dictionaries returned from the `cursor.column()` method as the second argument.

Here is a more complicated example:

```
cursor.create(
    table = 'Person',
    columns = [
        cursor.column(name='LastName',  type='String', required=True, unique=True),
        cursor.column(name='FirstName', type='String', default='Not Specified'),
        cursor.column(name='Address',   type='String' ),
        cursor.column(name='Age',       type='Integer'),
    ],
)
```

In this example we specified that the `LastName` must always be entered, does not have a default value and must be unique so that no two people in the database can have the same `LastName`. We have also specified that `FirstName` is not required and is not unique. If no value is entered for `FirstName` the field should be set to the string `Not Specified`.

In mysql This would create the following table:

```
mysql> describe Person;
+-------------+-------------+------+-----+---------------+-------+
| Field       | Type        | Null | Key | Default       | Extra |
+-------------+-------------+------+-----+---------------+-------+
| LastName    | varchar(255) |      | PRI |               |       |
| FirstName   | varchar(255) | YES  |     | Not Specified |       |
| Address     | varchar(255) | YES  |     | NULL          |       |
| DateOfBirth | date        | YES  |     | NULL          |       |
+-------------+-------------+------+-----+---------------+-------+
4 rows in set (0.00 sec)
```

The `create()` method of a `web.database` cursor looks like this:

`create``(` *table, columns* [, *values=[]*] [, *execute*] `)`

Create table with fields specified by fields. fields is a tuple of field tuples which can be obtained as follows:

```
columns = [
    cursor.column( field options... ),
    cursor.column( field options... ),
    cursor.column( field options... ),
    cursor.column( field options... ),
]
```

table The table name as a string.

columns A sequence of field tuples returned by `cursor.column()`

values A sequence of values to substitute for default values in the columns

execute If `False` the method returns the SQL string to perform the desired operations. If `True` the SQL is executed. If not specified takes the value specified in the cursor which by default is `True`

Dropping Tables

Warning: Dropping a table in SQL means removing the table from the database and therefore losing all the data it contained.

To drop (or remove) a table in SQL you would use the following command:

```
DROP TABLE table_name
```

For example:

```
DROP TABLE Person
```

To drop the table above using a `web.database` cursor we would use the following code:

```
cursor.drop('Person')
```

The `drop()` method of a `web.database` cursor looks like this:

`drop``(` *table*[, *execute*] `)`

Remove a table

tableA string containing the name of the table to drop.

executeIf `False` the method returns the SQL string to perform the desired operations. If `True` the SQL is executed. If not specified takes the value specified in the cursor which by default is `True`

Functions

The `cursor` objects currently support two function methods: `max()`, `min()` and `count()` as described below.

`max`(*table, column* [*,where=None*][*,values=[]*])
Returns the highest value of the column.

tableThe name of the table

columnThe name of the column

whereAn optional where clause

valuesValues to substitute for `?` parameters in the where clause.

`min`(*table, column* [*,where=None*][*,values=[]*])
Returns the lowest value of the column.

tableThe name of the table

columnThe name of the column

whereAn optional where clause

valuesValues to substitute for `?` parameters in the where clause.

`count`(*table, column* [*,where=None*][*,values=[]*])
Count the number of rows in the table matching *where*. If *where* is not specified, count all rows.

tableThe name of the table

columnThe name of the column

whereAn optional where clause

valuesValues to substitute for `?` parameters in the where clause.

For example consider the table below:

```
# Numbers
+--------+
| Number |
+--------+
| 1      |
+--------+
| 2      |
+--------+
| 3      |
+--------+
```

```
>>> cursor.max(table='Numbers', column='Number')
3
>>> cursor.min(table='Numbers', column='Number')
1
>>> cursor.max(table='Numbers', column='Number', where="Number<?", values=[3])
2
```

1.4.14 Supported Databases

The currently supported databases include:

SQLite Stores database in local text files. Full support.

SnakeSQL Pure Python SQL database. Used in the PythonWeb examples. Full support.

MySQL Supported through the MySQLdb module which is included with the web modules. Doesn't support transactions or foreign key constraint checks.

Other databases with varying levels of support:

PostgreSQL Support is planned but the authour has no access to a Postgres database so cannot yet write the wrapper.

ODBC Partially implemented, not yet available. All ODBC databases including MS Access are supported through the `mx.ODBC` driver available from http://www.egenix.com/. You will first need to indtall the `mx.BASE` package.

MySQL

Warning: The `MySQLdb` module on which the MySQL driver is based automatically commits any changes you have made to the database when the script exits, regardless of whether you have explicitly committed the changes in the code. This is different to the behaviour of the other databases and may catch you out so please be aware it is going on. (If anyone knows how to fix this please, please let the authour know!)

Also, MySQL doesn't explicitly check the foreign key constraints and so won't let you know you try an operation which would break those constriants.

SQLite

The SQLite implemenation appears robust and fully supports the entire specification.

Warning: The `Date`, `Time` and `DateTime` fields all use SQLite `Text` fields and not the corresponding `Date` fields so if you have an existing pysqlite database these fields my not be compatible. This may be changed in future releases of the modules.

ODBC

Implementation not finished. I'm having problems finding an SQL syntax guide to ODBC so that I can implement correct table create statements. Any ideas would be appreciated.

1.4.15 Example Code

Below is a script to test the database layer. It demonstrates the use of some of the commands:

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

import sys; sys.path.append('../../../') # show python where the modules are

import web.database
connection = web.database.connect(
    adapter="snakesql",
    database="database",
```

```
    autoCreate = 1,
)
cursor = connection.cursor()

import datetime

# Crete a table using the DB-API 2.0 interface and inset some information
cursor.execute('CREATE TABLE test( columnDate Date,  columnString String)')
cursor.execute(
    "INSERT INTO test ( columnDate, columnString) VALUES (?, 'This i\\s a str''ing with some aw
    datetime.date(2005,01,27)
)
# Retrieve the information
cursor.execute("SELECT * from test WHERE columnDate = '2005-01-27'")
print cursor.fetchall(format='dict')

# Update the row using the abstraction interface and retrieve the information
cursor.update(
    table = 'test',
    columns = ['columnString'],
    values = ["James's New String conta\\\\ining an apostrophe and awkward quoting"],
)
print cursor.select(columns='*', tables=['test'])

connection.close() # Close the connection without saving changes
```

1.4.16 API Reference

Warning: Developers using the `web.database` API should always specify values in methods by name and not rely on the position of parameters as the API may change in future versions.

Module Interface

Access to the database is made available through connection objects. The `web.database` module provides the following constructor for these:

`connect`(*adapter*,[*database*,][*user*,][*password*,][*host*,][*port*,][*socket*,][***params*])

Constructor for creating a connection to a database. Returns a `Connection` object. Not all databases will use all the parameters, but databases should use the parameters specified and not abbreviated versions. Any more complex parameters are passed directly to the driver's `connect()` method.

adapter The type of database to connect to. Can currently be `'MySQL'`, `'PySQLite'` or `'web.database'` but it is hoped that most database drivers will eventually be supported.

database The database name to connect to.

user The username to connect with.

password The password to use.

prepend A string to be transparantly used in front of all database tables.

host The host to connect to if the database is running on a remote server.

port The port to connect to if the database is running on a remote server.

socket The socket to connect to if the database is running locally and requires a socket.

****params*** Any other parameters to be passed to the driver

`web.database` implementers will usually override the method `makeConnection()` to provide this functionality as is clear from the source code.

These module globals are also be defined:

`version` String constant stating the supported DB API level.

`version_info` A tuple in the same format as `sys.version_info` for example something like `(2,4,0,rc1,'beta')`

Connection Objects

`Connection` objects respond to the following methods as defined in the DB-API 2.0 `close()`, `commit()` and `rollback()`. The `commit()` and `rollback()` methods should work as specified in the DB-API 2.0. Even if the database engine doesn't directly support transactions, these facilities should be emulated.

`Connection` objects also have a `cursor()` method.

`cursor`([*execute=True*], [*format='tuple'*], [*convert=True*], [*mode='portable'*])

The default values which the cursor abstraction methods will take for the values of *execute*, *format* and *convert* can be set using this method.

format This can be `'tuple'` to return the results as a tuples, `'text'` to return as text wrapped to 80 characters for display in a terminal, `'dict'` to return the results as dictionaries or `'object'` to return the results as result objects to be treated as dictionaries, tuples or via attribute access.

convert Convert the results to standard formats (should be `True` for most users)

execute Used in the cursor SQL methods. If `True` then rather than returning an SQL string, the methods execute the results

mode The default mode for the `execute()` method. Can be `'portable'` to use the SQL abstraction methods or `'direct'` to send the SQL directly to the underlying cursor.

Connection objects also have the following attributes:

`tables` A dictionary of `Table` objects with their names as the keys

`converters` A dictionary of field converter objects for all supported database types.

`baseConnection` The DB-API 2.0 `Connection` object

Cursor Objects

`close`()

Close the cursor now (rather than whenever __del__ is called). The cursor will be unusable from this point forward; an Error (or subclass) exception will be raised if any operation is attempted with the cursor.

`export`(*tables*, [*includeCreate=True*])

Export the tables specified by *tables* as portable SQL including statements to create the tables if *includeCreate* is `True`.

Importing the SQL is then simply a matter of executing the SQL. Here is an example:

```
backup = cursor.export(tables=['testTable'])
cursor.drop(table='testTable')
for sql in backup.split('\n'):
    cursor.execute(sql, mode='portable')
```

The `testTable` should be exactly the same as it was before the code was executed.

`Cursor` objects have the following attributes:

`connection`
This read-only attribute return a reference to the Connection object on which the cursor was created. The attribute simplifies writing polymorph code in multi-connection environments.

`info`
A list of `Column` objects for in the order of the fields from the last `SELECT` or `None` if the last SQL operation was not a `SELECT`. `Column` objects contain all the information about a particular field and provide conversion methods for that field.

`baseCursor`
The DB-API 2.0 `Cursor` object

`sql`
A list of tuples of parameters passed to the `execute()` methods

Execute SQL `web.database` compliant databases support qmark style parameters for substitutions as follows:

```
cursor.execute('SELECT * FROM Test WHERE columnName=?',['textEntry'])
```

`execute`(*sql*[, *parameters*][,*mode*])
Prepare and execute a database operation. Parameters are provided as a sequence and will be bound to `?` variables in the operation. *mode* can be `'direct'` to pass the parameters to the underlying DB-API 2.0 cursor or `'portable'` to execute the code in a portable fashion.

`executemany`(*sql*, *manyParameters*[,*mode*])
Similar to `execute()` but the operation is executed for each sequence in *manyParameters*.

Fetch Results All these methods take the parameters *format* and *convert*. If they are not specified the values set in the `cursor()` method of the `Connection` object is used.

`fetchone`([*format*],[*convert*])
Fetch the next row of a query result set, returning a single sequence, or None when no more data is available. [6]

An Error (or subclass) exception is raised if the previous call to executeXXX() did not produce any result set or no call was issued yet.

format The format of the results returned. Can be `'dict'` to return them as a tuple of dictionary objects, `'tuple'` to return them as a tuple of tuples, `'object'` to return them as a tuple of `dtuple` objects which can be treated as a tuple or a dictionary (or via attribute access for the majority of column names) or `'text'` to return tables designed to be displayed in a terminal 80 characters wide. If not specified takes the value specified in the cursor which by default is `'tuple'`

convert Can be `True` to convert the results to the correct types, `False` to leave the results as they are returned from the base cursor. If not specified takes the value specified in the cursor which by default is `True`

`fetchall`([*format*],[*convert*])
Fetch all (remaining) rows of a query result, returning them as a sequence of sequences (e.g. a list of tuples).

An Error (or subclass) exception is raised if the previous call to an `execute()` method did not produce any result set or no call was issued yet.

The values *format* and *convert* are as specified in `fetchone()`

Cursor Abstraction Methods It is assumed that if *execute* is `True` in the following methods then you wish to be executing the code in portable mode, otherwise it is unlikely you would be using abstraction methods.

If you did wish to execute code in direct mode (through the DB-API 2.0 cursor) you could do the following:

```
sql = cusror.select(columns=['*'], tables=['table'], execute=False)
cursor.execute(sql, mode='direct')
```

Warning: It is possible to get the cursor abstraction methods to perform operations they were not designed for. For example, in `cursor.select()` you could specify one of the columns as `'AVG(columnName)'`. This would produce an SQL statement which would return the mean value of the column `columnName` on some databases but certainly not on all and therefore breaks the specification which states that columns should be a list of column names. To ensure database portability please stick to the published API.

`select`(*tables, columns,* [*values=[],*][*where=None,*][*order=None,*][*execute=None,*][*fetch=None,*][***params*])

Build an SQL string according to the options specified and optionally execute the SQL and return the results in the format specified. No error checking on field names if the SQL string is only being built. Strict error checking is only performed when executing the code.

tables A string containing the name of the table to select from or if selecting from multiple tables, a sequence of table names.

columns A sequence of column names to select. Can be a string if only one column is being selected. If selecting from multiple tables, all column names should be in the form `'tableName.columnName'`

values A list of values to substitute for `?` in the `WHERE` clause specified by *where*.

where The WHERE clause as a `web.database` list as returned by `cursor.where()`. If *where* is a string it is converted to the correct format.

order The ORDER BY clause as a `web.database` list as returned by `cursor.order()`. If *order* is a string it is converted to the correct format.

execute If `False` the method returns the SQL string needed to perform the desired operations. If `True` the SQL is executed and the results converted and returned in the appropriate form. If not specified takes the value specified in the cursor which by default is `True`

fetch Whether or not to fetch the results. If `True` and *execute* is not specified *execute* is set to `True`. If `True` and *execute* `False` an error is raised.

*****params*** The parameters to be passed to the `fetchall()` method if *fetch* is `True`

`insert`(*table, columns, values, _sqlValues,* [*execute*])

Insert values into the columns in table. Either *values* or *_sqlValues* can be specified but not both.

table The name of the table to insert into

columns A sequence of column names in the same order as the values which are going to be inserted into those columns. Can be a string if only one column is going to have values inserted

values A sequence of Python values to be inserted into the columns named in the *columns* variable. Can be the value rather than a list if there is only one value. If *values* is specified then *_sqlValues* must be either an empty sequence or contain a list of all quoted SQL strings for the columns specified in which case *values* contains the Python values of the SQL strings to be substituted for `?` parameters in the *_sqlValues* sequence.

_sqlValues A sequence of quoted SQL strings to be inserted into the columns named in the *columns* variable. Can be the value rather than a list if there is only one value. If *_sqlValues* is specified and contains `?` parameters for substitution then *values* contains the values to be substituted. Otherwise *values* must be an empty sequence.

execute If `False` the method returns the SQL string to perform the desired operations. If `True` the SQL is executed. If not specified takes the value specified in the cursor which by default is `True`

`insertMany`(*table, columns, values, _sqlValues,* [*execute*])
Same as `insert()` except that *values* or *_sqlValues* contain a sequence of sequences of values to be inserted.

`update`(*table, columns, values, _sqlValues* [, *where*] [, *execute*])
Update the columns in table with the values. Either *values* or *_sqlValues* can be specified but not both.

table A string containing the name of the table to update

columns A sequence of column names in the same order as the values which are going to be updated in those columns. Can be a string if only one column is going to have values inserted

values A sequence of Python values to be inserted into the columns named in the *columns* variable. Can be the value rather than a list if there is only one value. If *values* is specified then *_sqlValues* must be either an empty sequence or contain a list of all quoted SQL strings for the columns specified in which case *values* contains the Python values of the SQL strings to be substituted for `?` parameters in the *_sqlValues* sequence. If there are more values specified in *values* than *_sqlValues* the remaining values are used to substitute for `?` parameters in *where*.

_sqlValues A sequence of quoted SQL strings to be inserted into the columns named in the *columns* variable. Can be the value rather than a list if there is only one value. If *_sqlValues* is specified and contains `?` parameters for substitution then *values* contains the values to be substituted. Otherwise *values* must be an empty sequence.

where The WHERE clause as a `web.database` list as returned by `cursor.where()`. If *where* is a string it is converted to the correct format.

execute If `False` the method returns the SQL string to perform the desired operations. If `True` the SQL is executed. If not specified takes the value specified in the cursor which by default is `True`

`delete`(*table,* [*values=[]*][, *where*] [, *execute*])
Delete records from the table according to *where*.

table A string containing the name of the table to select from or if selecting from multiple tables, a sequence of table names.

values A list of values to substitute for `?` in the `WHERE` clause specified by *where*.

where The WHERE clause as a `web.database` list as returned by `cursor.where()`. If *where* is a string it is converted to the correct format.

execute If `False` the method returns the SQL string to perform the desired operations. If `True` the SQL is executed. If not specified takes the value specified in the cursor which by default is `True`

`create`(*table, columns* [, *values=[]*] [, *execute*])
Create table with fields specified by fields. fields is a tuple of field tuples which can be obtained as follows:

```
columns = [
    cursor.column( field options... ),
    cursor.column( field options... ),
    cursor.column( field options... ),
    cursor.column( field options... ),
]
```

table The table name as a string.

columns A sequence of field tuples returned by `cursor.column()`

values A sequence of values to substitute for default values in the columns

execute If `False` the method returns the SQL string to perform the desired operations. If `True` the SQL is executed. If not specified takes the value specified in the cursor which by default is `True`

drop(*table*[, *execute*])
Remove a table

tableA string containing the name of the table to drop.

executeIf `False` the method returns the SQL string to perform the desired operations. If `True` the SQL is executed. If not specified takes the value specified in the cursor which by default is `True`

_function(*function, table, column,* [*,where=None*][*, values=[]*])
Returns the result of applying the specified function to the field

functionThe function to be applied, can be `'max'`, `'min'`, `'sum'` or `'count'`

tableThe name of the table

columnThe name of the field

whereAn optional where clause

valuesA list of values to substitute for `?` parameters in the `WHERE` clause

max(*table, column* [*,where=None*][*,values=[]*])
Returns the highest value of the column.

tableThe name of the table

columnThe name of the column

whereAn optional where clause

valuesValues to substitute for `?` parameters in the where clause.

min(*table, column* [*,where=None*][*,values=[]*])
Returns the lowest value of the column.

tableThe name of the table

columnThe name of the column

whereAn optional where clause

valuesValues to substitute for `?` parameters in the where clause.

count(*table, column* [*,where=None*][*,values=[]*])
Count the number of rows in the table matching *where*. If *where* is not specified, count all rows.

tableThe name of the table

columnThe name of the column

whereAn optional where clause

valuesValues to substitute for `?` parameters in the where clause.

Helper Methods Helper methods build the data structures which should be passed to the `Cursor` abstraction methods.

column(*name, type*[*, required=0*][*, unique=0*][*, primaryKey=0*][*, foreignKey=None*][*, default=None*])
Return a column tuple suitable for use in the columns tuple used in the `create()` method.

`Bool`, `Float`, `Binary` and `Text` columns cannot be used as primary keys or unique columns. `Binary` and `Text` columns cannot have default values.

nameThe name of the field as a string.

typeThe field type. This can take one of the values: `'Bool'`, `'String'`, `'Text'`, `'Binary'`, `'Long'`, `'Integer'`, `'Float'`, `'Date'`, `'Time'`, `'Datetime'`

requiredWhether or not the field is required. Setting to `True` means the field cannot have `NULL` values.

unique Set to `True` if the value must be unique. Two fields in the column cannot have the same value unless that value is NULL

primaryKey The field is to be used as a primary key, the field has the same behaviour as being unique and required but no default value can be set

foreignKey The field is to be used as a foreign key, the value should be the name of the table for which this is a child table. **Note:** There is no need to specify the column name as tables can only have one primary key.

default The default value for the field to be set to. If not specified the default is NULL

`where`(*where,* [*values=[]*])

Return a parsed `WHERE` clause suitable for use in the `select()`, `update()` and `delete()` methods of the `cursor` object.

where A string containing the `WHERE` clause. Can include the `LIKE` operator which is used as follows:

```
WHERE columnName LIKE %s1%s2%s
```

Every % sign is matched against zero or more characters.

Note: *where* should not include the string `'WHERE'` at the beginning.

values A list of values to substitute for `?` parameters in the `WHERE` clause

`order`(*order*)

Return a parsed `ORDER BY` clause suitable for use in the `select()` method of the `cursor` object.

order A string containing the `ORDER BY` clause. **Note:** *order* should not include the string `'ORDER BY'` at the beginning.

Utility Methods

`export`()

Export the database as a series of SQL strings which can be executed to completely recreate precicely the original database structure. This is useful for database backups or for moving data from one database to another.

Warning: This is a new feature and has not had a lot of testing.

Table Objects

`Table` objects can be accessed through the `tables` attribute of the `Connection` object like this:

```
>>> print connection.tables['tableName'].name
tableName
>>> print connection.tables['tableName']['columnName'].name
columnName
```

class `Table`

`Table` objects store all the meta data there is to know about an SQL table. They are created by the `web.database` module and should not be created manually. They are simply structures to hold table information. The values should not be changed.

`Table` objects have the following attributes:

`name`

The name of the table correctly capitalised

`columns`

A list of `Column` objects describing each column of the table

primaryKey
The name of the primary key column of the table or `None` if no primary key is specified

parentTables
A list of the names of any tables for which the table has foreign key fields

childTables
A list of the names of any tables for which the table is a parent table

and the following methods:

has_key(*columnName*)

columnExists(*columnName*)
Returns `True` if *columnName* is the name of a column in the table

column(*columnName*)

__getitem__(*columnName*)
Returns the column object for *columnName*

`Table` objects can also contain any other useful methods which the module implementer feels are appropriate.

Column Objects

`Column` objects store all the information there is to know about a particular column. `Column` objects can be accessed through the `connection.tables` dictionary which contains all columns or through `cursor.info` which contains a tuple corresponding to the `Column` objects selected after a `SELECT` statement has been executed in portable mode (or `None` after any other SQL operation).

```
>>> cursor.select(columns=['columnName'], tables=['tableName'], execute=True)
>>> print cursor.info[0].name
columnName
>>> print cursor.info[0].table
tableName
```

class Column **name**
The name of the column

type
The capitalised string representing the column type

baseType
The capitalised string representing the column type of the base type

table
The table of which the column is a part

required
Can be `True` or `False` depending on whether or not the column value is required (i.e. cannot be NULL)

unique
`True` if the field should be unique, `False` otherwise

key
`True` if the field is a primary key, `False` otherwise

default
The default value of the field

converter
A reference to the `Converter` object for the field type

position
The position of the field in the tuple returned by `SELECT * FROM table`

Converter Objects

`Converter` objects contain methods to convert values between SQL and Python objects and to convert values returned by the database driver into the correct Python type. `Converter` objects are accessed through the `converter` attribute of the corresponding `Column` object.

Example: convert a list of values selected from a database to their SQL encoded equivalents

```
>>> cursor.select(columns=['table1.columnOne', 'table2.column2'], tables=['table1', 'table2'],
>>> results = cursor.fetchall()
>>> record = results[0]
>>> newRecord = []
>>> for i in range(len(record)):
...     newRecord.append(cursor.info[i].converter.valueToSQL(record[i]))
```

class `Converter` **`valueToSQL`**(*v*

Convert a Python object to an SQL string

`sqlToValue`(*value*)
Convert the an SQL string to a Python object

`databaseToValue`(*value*)
Convert the value stored in the database to a Python object

`valueToDatabase`(*value*)
Convert a Python object to the format needed to store it in the database

`type`
A string representing the column type

`sqlQuotes`
`True` if the SQL representation should be quoted, `False` otherwise

`Converter` objects are also available as a dictionary with column types as the keys as the `converters` attribute of the `Connection` object.

1.4.17 Developer's Guide

Implementing the Classes

Virtually all the functionality of the API has been implemented as base classes from which module implementers simply need to derive their own classes, over-riding methods to suit their particular database syntax as necessary.

In particular this requires writing custom converter methods to ensure that the database returns the correct values, overriding the `makeConnection()` method to convert `connect()` method parameters to the appropriate form for the driver, and overriding the cursor abstraction methods so that they build the correct SQL strings from the parameters.

`web.database`implementation comes with basic implementations for PySQLite, `web.database` and a partial implementation for MySQLdb (transaction support isn't implemented). These can all be used as examples.

If a particular database engine does not natively support part of the API it should be emulated in the derived classes even if it is difficult or slow to do so.

Creating the Dictionary

The implementation should contain a dictionary named `driver` in the sub-package of the main module named `pdbc`. So for example, `web.database` will have a module `web.database .pdbc` which will contain a dictionary

named `driver` similar to the following:

```
driver = {
    'converters':{
        'String':   base.BaseStringConverter(),
        'Text':     base.BaseTextConverter(),
        'Binary':   base.BaseBinaryConverter(),
        'Bool':     base.BaseBoolConverter(),
        'Integer':  base.BaseIntegerConverter(),
        'Long':     base.BaseLongConverter(),
        'Float':    base.BaseFloatConverter(),
        'Date':     base.BaseDateConverter(),
        'Datetime': base.BaseDatetimeConverter(),
        'Time':     base.BaseTimeConverter(),
    }
    'columnClass':base.BaseColumn,
    'tableClass':base.BaseTable,
    'cursorClass':Cursor,
    'connectionClass':Connection,
}
```

Where `Connection` and `Cursor` are classes derived from `base.Connection` and `base.Cursor` respectively.

1.4.18 Tools Under Development

This section describes certain tools based on `web.database` which are currently under development.

Web Based Admin

It also becomes possible to define HTML fields for each data type (and therefore each derivative data type) so that web-based editing of a `web.database` compliant database becomes very simple.

1.4.19 Future Additions

This section is just a list of currently excluded features which might be useful in the next version. They are in the order of importance:

- Autoincrement Integer fields
- Support for `UPDATE SET TOTAL = TOTAL + 100` syntax

This is a list of things currently not included in the module but which may be of use later on:

- Make sure max and min work in all cursors for all field types
- Check table aliases actually work!
- Caching of all values to save on SQL calls
- Checking type conversions for ODBC and also the mx.DateTime issues
- Fixing and testing of `alter()`

- Table copying code
- Executemany support

1.5 web.database.object — An object relation mapper built on the web.database and web.form modules

The web.database.object module is an object-relational mapper. It allows you to simply define complex database structures in Python code and then create the necessary tables automatically. It then allows you to manipulate the Python objects you have defined to transparently manipulate the underlying database including the facility to use multiple joins without knowing any SQL.

Furthermore the table column classes are derived from web.form.field objects which means you can transparently create HTML interfaces to edit the data structures through a web browser. This makes web.database.object module ideal as a middle layer for writing data-driver websites although it has broader uses as well.

A database object can in theory have any storage driver (text, XML, SQL Datbase, DBM) although currently only a driver for the web.database module has been written. This means that any storage system with a driver for web.database can be used with web.database.object. This currently includes MySQL, ODBC, SQLite and, to an extent, Gadlfy.

1.5.1 Introduction

Requirements

To use web.database.object you need Python 2.2 or above and the Web Modules of which web.database.object is a part and an SQL database supported by the web.database module and its associated Python driver. If you use MuSQL, a MySQL database is needed, alternatively use an ODBC database such as MS Access. The web.database driver is included with the Python Web Modules but you will need to download and install the ODBC driver from the http://www.eGenix.com site yourself as it comes with a non open source licence.

Compared To Other Database Wrappers

There are several object-relational mappers for Python and a series of basic database wrappers similar to web.database. The authour cannot comment deeply on these.

web.database.object is most similar to SQLObject available from http://www.sqlobject.org in that it creates objects that feel similar to normal Python objects and can be accessed and manipulated in a familiar way.

Modules like SQLObject and web.database.object differ from modules such as PyDO or the web.database module which simply provide more Python-like ways of executing SQL queries and then package up the database returns into more useable forms such as dictionaries. The web.database.object completely removes any need to know any SQL. You just manipulate the objects themselves and the rest is done for you. This makes SQL programming extremely simple and still gives you full control over the information stored in the database.

What separates web.database.object from some other software is the strong typing of the data. If you are accessing the property of a Date field the object will be a datetime.Date object. If you are setting an Email field, only strings in the format of an email address will be allowed. The module also direct support for one-to-many and many-to-many mappings which means you can build complex data structures.

Because the software interfaces the database through a web.database cursor (in principle it could interface to other drivers as well) the code written will be database independant and run on any database supported by the web.database module.

`web.database.object` uses classes derived from `Table`, `Database` and `web.form.field` classes to facilitate this integration. Some ORMs use code generation to create an interface, expressing the schema in a CSV or XML file (for example, MiddleKit, part of Webware). By using normal Python objects you are able to comfortably define your database in the Python source code. No code generation, no weird tools, no compilation step.

What truly separates `web.database.object` from any other ORM in any language (to the authour's knowledge - correct me please if I am wrong) is that on top of all the features mentioned above, the columns used to store the SQL data are also instances of `web.form.field.typed` and the tables have the ability to generate `web.form` `Form` objects. This means it is possible to create HTML interfaces to edit the database data automatically and in such a way that the user can only enter valid data otherwise the user will be asked to make corrections. This functionality makes building complex web databases much simpler.

1.5.2 Introductory Example

Below is about the simplest possible example where a database object named `MyDatabase` is created. The database object is connected to an SQLite database named `object-simple.db` but could equally well be a MySQL database or ODBC supported database like MS Access.

```
import web.database, web.database.object

connection = web.database.connect(type="sqlite",database="object-simple.db")
cursor = connection.cursor()

person = web.database.object.Table("Person")
person.addColumn(web.database.object.String(name="firstName"))
person.addColumn(web.database.object.String(name="surname"))

database = web.database.object.Database()
database.addTable(person)
database.init(cursor)
```

This first lines import the modules we need and make the `web.database` connection. We could have made any database connection supported by the `web.database` module. Below are some other examples for the 3rd line.

```
connection = web.database.connect(type="odbc", database="AccessDatabase")
connection = web.database.connect(type="mysql", host="pythonweb.org", user="james", password="h
```

The database will contain one table named `Person`. The `Person` table has two columns, both of which are String columns. One is named `firstName` and the other `surname`. All `web.database.object` column objects must take a *name* parameter and this is used as the column name.

Once we have finished defining our table we create a `web.database.object.Database()` which will be the object we use to manipulate the database. We add out table definition to the database definition using `database.addTable(person)` and then initilise the database to associate it with the live databaseusing `database.init(cursor)`.

Warning: Once a database object is initialised you cannot add any more tables or modify the database's structure in any way.

Now we have defined and initialised out database we can start using it. If the table does not already exist in the live database we need to create it as follows:

```
if not database.tablesExist():
    database.createTables()
    print "Created Table"
```

This command creates every table the database needs (in our case just the one). If you decide to change the structure of the tables at a later date after you have created the tables in the live database you will need to remove them all using `database.dropTables()` and recreate them from scratch. This means you would loose all the information so it is important to decide on the correct structure before creating the tables.

All information in the database can be accessed through a dictionary-like interface. For example the database object acts like a dictionary of tables and each table acts like a dictionary of rows. Each row acts like a dictionary of field values.

Now we have created the table we are free to add, edit and remove data. Following on from the previous example.

```
>>> john = database['Person'].insert(firstName="John", surname="Smith")
>>> print john['firstName']
John
>>> print john['surname']
Smith
>>> john['surname'] = 'Doe'
>>> print john['surname']
Doe
>>> print john['rowid']
1
```

In this way you can create and modify the table information. Take note of the line `>>> john['rowid']`. Each new object (which is equivalent to a row in the table) is given a unique integer number named the `rowid` by which it can be identified.

We can use this rowid to retrieve John Smith's information from the database at a later time. There are two ways to retrieve rows from the table using the rowid:

```
>>> row1 = database['Person'][1]
>>> row2 = database['Person'].row(1)
>>> print row1 == row2 == john
1
```

Once you have made changes to the database you will need to commit your changes using `connection.commit()` otherwise your changes may be lost. By the end of this session our database table looks like this:

```
# Tables in the 'test' database
+----------------+
| Tables_in_test |
+----------------+
| Person         |
+----------------+

# The Person table
+-------+-----------+---------+
| rowid | firstName | surname |
+-------+-----------+---------+
| 1     | John      | Doe     |
+-------+-----------+---------+
```

Thats about all there is to it!

Full Code Listing

Here is a complete code listing so that you can experiment:

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

import sys; sys.path.append('../../../') # show python where the modules are

import web.database, web.database.object
connection = web.database.connect(
    adapter="snakesql",
    database="database-object-simple",
    autoCreate = 1,
)
cursor = connection.cursor()

person = web.database.object.Table("Person")
person.addColumn(web.database.object.String(name="firstName"))
person.addColumn(web.database.object.String(name="surname"))

database = web.database.object.Database()
database.addTable(person)
database.init(cursor)

if not database.tablesExist():
    database.createTables()
    print "Created Table"

john = database['Person'].insert(firstName="John", surname="Smith")
print john['firstName']
print john['surname']

john['surname'] = 'Doe'
print john['surname']

print john['rowid']

row1 = database['Person'][1]
row2 = database['Person'].row(1)
print row1 == row2 == john
```

```
connection.close() # Close the connection without saving changes
```

The output is:

```
Created Table
John
Smith
Doe
1
1
```

Note: If you run the code more than once you will be adding lots of John Smiths to the test database and so the rowid value will be one larger each time you run the code. After the first time you run the code the line `Created Table` will not be output since the table will already be created.

Using Alternative Keys

In the example above we could access John Smith's information as follows:

```
>>> row1 = database['Person'][1]
>>> row2 = database['Person'].row(1)
```

We could have defined the `surname` column differently and added it like this instead:

```
person.addColumn(web.database.object.String(name="surname", unique=True, required=True, key=Tru
```

This defines the surname as a unique, required field. *unique* means that there cannot be two people with the same surname in the database. If you try to add two people with the same name an Exception will be raised. *required* means that you must always enter a surname, although in out example, because *required* is not specified for the firstName column, you would not have to enter a firstName.

Specifying *key* as `True` for the surname tells the table that you want to be able to retrieve data from the database based on the surname column rather than the rowid. We can now try the following:

```
>>> row1 = database['Person']['Smith']
>>> row2 = database['Person'].row(1)
>>> print row1 == row2
True
```

You can still access the information by rowid using the `row()` method.

Any column can be specified as a key but there can only be one column in each table specified as a key. Any column specified as a key must also be specified as unique and required.

Available Columns

There are a number of column types available for use with the `web.database.object` module. These include: `String`, `StringSelect`, `Text`, `Bool`, `Integer`, `IntegerSelect`, `Float`, `FloatSelect`, `Date`, `DateSelect`, `Time`, `TimeSelect`, `DateTime`, `DateTimeSelect`, `Email` and `URL`

Each `web.database.object` column is derived for the corresponding `web.form.field` field which means it behaves in exactly the same way. You can see the available options in the `web.form.field` documentation. Each `web.database.object` column has two more parameters in addition to those of its corresponding `web.form.field`. These are *unique* and *key* described in the previous example.

1.5.3 One-To-Many Mappings

One of the features that distinguishes this module from many others is its ability to deal with more complex datastructures than just simple tables. As an example of a one-to-many mapping we will consider an address book.

In our address book each person can have many addresses but each address is only associated with one person. The data structure looks like this:

```
            +--  Address 1
Person 1 ---|
            +--  Address 2
```

To create a database to describe this struture we need two tables, a Person table and an Address table.

```
import web.database, web.database.object
connection = web.database.connect(type="sqlite", database="object-multiple.db")
cursor = connection.cursor()

person = web.database.object.Table("Person")
person.addColumn(web.database.object.String(name="firstName"))
person.addColumn(web.database.object.String(name="surname"))
person.addMultiple(name="addresses", foreignTable="Address")

address = web.database.object.Table("Address")
address.addColumn(web.database.object.String(name="firstLine"))
address.addColumn(web.database.object.String(name="postcode"))
address.addSingle(name="person", foreignTable="Person")

database = web.database.object.Database()
database.addTable(person)
database.addTable(address)
database.init(cursor)
```

As in the introductory example we use the `addColumn()` method to add `Column` objects to the Address table. This time however we also use the `addSingle()` method to add a column named `person` to the table. We have also used `addMultiple()` method to add a multiple join called `addresses` from the Person foreign table to the Person table. The final change is that we have added the `Address` table to the database.

Note: We in the `addSingle()` and `addMultiple()` methods we refer to the `foreignTable` by the string representing its name and not the object itself.

When we access a person's `addresses` key, we will get back a list of all the Address objects associated with that person. Continuing the example above:

```
>>> john = database['Person'].insert(firstName='John', surname='Smith')
>>> print john['surname']
Smith
>>> print john['addresses']
{}
>>> database['Address'].insert(person=john, firstLine='12 Friendly Place', postcode='OX4 1AB')
>>> database['Address'].insert(person=john, firstLine='3a Crazy Gardens', postcode='OX1 2ZX')
>>> for address in john['addresses'].values:
...    print address['firstLine']
...
12 Friendly Place
3a Crazy Gardens
```

Note how we specify the person to add the address to using `person=john`. We could alternatively have specified the rowid of the person to add the address to. Just like the database, tables and rows, the value returned by john['addresses'] behaves like a dictionary. In this example we use the `values()` method to return a list of the `Row` objects.

It should be noted that you cannot set the values of multiple columns like the `'addresses'` column directly. Instead you should set the values of each object induvidually.

```
>>> john['addresses'] = something # XXX Doesn't work!
```

Again you muct use `cursor.commit()` to commit the changes to the database.

Just for interest here is how the tables look in the live database. You can see that the person column in the Address table contains the rowid in the Person table of the person to associate the address with.

```
# Tables in the 'test' database
+----------------+
| Tables_in_test |
+----------------+
| Address        |
| Person         |
+----------------+

# The Person table
+-------+-----------+---------+
| rowid | firstName | surname |
+-------+-----------+---------+
| 1     | John      | Smith   |
+-------+-----------+---------+

# The Address table
+-------+----------------------+----------+--------+
| rowid | firstLine            | postcode | person |
+-------+----------------------+----------+--------+
| 1     | 12 Friendly Place    | OX4 1AB  | 1      |
+-------+----------------------+----------+--------+
| 2     | 3a Crazy Gardens     | OX1 2ZX  | 1      |
+-------+----------------------+----------+--------+
```

Full Code Listing

Here is a complete code listing so that you can experiment:

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

import sys; sys.path.append('../../../') # show python where the modules are

import web.database, web.database.object
connection = web.database.connect(
    adapter="snakesql",
    database="database-object-multiple",
    autoCreate = 1,
)
cursor = connection.cursor()

person = web.database.object.Table("Person")
person.addColumn(web.database.object.String(name="firstName"))
person.addColumn(web.database.object.String(name="surname"))
person.addMultiple(name="addresses", foreignTable="Address")

address = web.database.object.Table("Address")
address.addColumn(web.database.object.String(name="firstLine"))
address.addColumn(web.database.object.String(name="postcode"))
address.addSingle(name="person", foreignTable="Person")

database = web.database.object.Database()
database.addTable(person)
database.addTable(address)
database.init(cursor)

if not database.tablesExist():
    database.createTables()
    print "Created Table"
else:
    raise Exception('Tables not created')
john = database['Person'].insert(firstName='John', surname='Smith')
print john['surname']
print john['addresses']

database['Address'].insert(person=john, firstLine='12 Friendly Place', postcode='OX4 1AB')
database['Address'].insert(person=john, firstLine='3a Crazy Gardens', postcode='OX1 2ZX')

for address in john['addresses'].values():
    print address['firstLine']

connection.close() # Close the connection without saving changes
```

The output is:

```
Created Table
Smith
{}
12 Friendly Place
3a Crazy Gardens
```

You will need to delete the database file 'object-multiple.db' each time you run the cose so that it can be recreated each

time.

1.5.4 Many-To-Many Mappings

In a real life more than one person might live at the same address and each person might have multiple addresses. The relationship is actually a many-to-many mapping. Have a look at the code below:

```
import web.database, web.database.object
connection = web.database.connect(type="sqlite", database="object-multiple.db")
cursor = connection.cursor()

person = web.database.object.Table("Person")
person.addColumn(web.database.object.String(name="firstName"))
person.addColumn(web.database.object.String(name="surname"))
person.addRelated(name="addresses", foreignTable="Address")

address = web.database.object.Table("Address")
address.addColumn(web.database.object.String(name="firstLine"))
address.addColumn(web.database.object.String(name="postcode"))
address.addRelated(name="person", foreignTable="Person")

database = web.database.object.Database()
database.addTable(person)
database.addTable(address)
database.init(cursor)
```

We have now related the two tables using the `addRelated()` method of each class instead of using `addMultiple()` and `addSingle()`.

Note: Because the two Classes use related joins the `database.createTables()` method actually creates an intermediate table to store the relationships. The modules hide this table so you don't need to worry about it to use `web.database.object`. If you are interested the table is named by taking the two tables in alphabetical order and joining thier names with an underscore. For example the table in the example above will create a table names `'Adrress_Person'`. This name can be customised by deriving a customised class from `web.database.object.Table` and overriding the `_relatedTableName()` method of both tables.

Here is an example:

```
>>> john = database['Person'].insert(firstName='John', surname='Smith')
>>> owen = database['Person'].insert(firstName='Owen', surname='Jones')
>>>
>>> friendlyPlace = database['Address'].insert(firstLine='12 Friendly Place', postcode='MK4 1AB
>>> crazyGardens  = database['Address'].insert(firstLine='3a Crazy Gardens', postcode='OX1 2ZX'
>>> greatRoad     = database['Address'].insert(firstLine='124 Great Road', postcode='JG6 3TR')
>>>
>>> john.relate(friendlyPlace)
>>> owen.relate(greatRoad)
>>> crazyGardens.relate(john)
>>>
>>> print john['addresses'].keys()
['MK4 1AB', 'OX1 2ZX']
>>> for address in john['addresses'].values():
...     print address['postcode']
...
MK4 1AB
OX1 2ZX
>>> print greatRoad['people'].keys()
['Owen']
>>> print owen['addresses']['JG6 3TR']['people'].keys()
['Owen']
>>> john['addresses']['MK4 1AB']['firstLine'] = 'The Cottage, 12 Friendly Place'
>>> print database['Person']['John']['addresses']['MK4 1AB']['firstLine']
The Cottage, 12 Friendly Place
```

The code should be fairly self-explainatory. We are inserting some different people and addresses into the table and the relating them to each other. Each row from each table can be related to as many other rows from the other table as you like. Or a row might not be related to another one at all.

It should be noted that you cannot set the values of multiple columns like the `'addresses'` column directly. Instead you should set the values of each object induvidually.

```
>>> john['addresses'] = something # XXX Doesn't work!
```

You can create fairly complex expressions as is demonstrated by the expression:

```
database['Person']['John']['addresses']['MK4 1AB']['firstLine']
```

Here we are selecting all the addresses from the row `'John'` from the `'Person'` table and then selecting the first line of the address with postcode `'MK4 1AB'`. It is actually possible to create circular references (although not very useful) as shown below.

```
>>> john == database['Person']['John'] == \
... database['Person']['John']['addresses']['MK4 1AB']['people']['John'] \
... == database['Person']['John']['addresses']['MK4 1AB']['people']['John'] \
... ['addresses']['MK4 1AB']['people']['John']
True
```

Just for interest here is how the tables look after running the example. You can see that the Address_Person table contains the rowids of the related people and addresses.

```
# Tables in the 'test' database
+----------------+
| Tables_in_test |
+----------------+
| Address        |
| Person         |
| Address_Person |
+----------------+

# The Person table
+-------+-----------+---------+
| rowid | firstName | surname |
+-------+-----------+---------+
| 1     | John      | Smith   |
+-------+-----------+---------+
| 2     | Owen      | Jones   |
+-------+-----------+---------+

# The Address table
+-------+-------------------------------------+----------+
| rowid | firstLine                           | postcode |
+-------+-------------------------------------+----------+
| 1     | The Cottage, 12 Friendly Place      | MK4 1AB  |
+-------+-------------------------------------+----------+
| 2     | 3a Crazy Gardens                    | OX1 2ZX  |
+-------+-------------------------------------+----------+
| 2     | 124 Great Road                      | JG6 3TR  |
+-------+-------------------------------------+----------+

# The Address_Person table

+--------+-----------+
| people | addresses |
+--------+-----------+
| 1      | 1         |
+--------+-----------+
| 2      | 2         |
+--------+-----------+
| 1      | 3         |
+--------+-----------+
```

It should be noted that each table can contain as many columns, multiple, related and single joins as you like.

Full Code Listing

Here is a complete code listing so that you can experiment:

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

import sys; sys.path.append('../../../') # show python where the modules are

import web.database, web.database.object
connection = web.database.connect(
    adapter="snakesql",
    database="database-object-related",
    autoCreate = 1,
```

```
)
cursor = connection.cursor()

person = web.database.object.Table("Person")
person.addColumn(web.database.object.String(name="firstName", unique=True, required=True, key=T
person.addColumn(web.database.object.String(name="surname"))
person.addRelated(name="addresses", foreignTable="Address")

address = web.database.object.Table("Address")
address.addColumn(web.database.object.String(name="firstLine"))
address.addColumn(web.database.object.String(name="postcode", unique=True, required=True, key=T
address.addRelated(name="people", foreignTable="Person")

database = web.database.object.Database()
database.addTable(person)
database.addTable(address)
database.init(cursor)

if not database.tablesExist():
    database.createTables()
    print "Created Table"

john = database['Person'].insert(firstName='John', surname='Smith')
owen = database['Person'].insert(firstName='Owen', surname='Jones')

friendlyPlace = database['Address'].insert(firstLine='12 Friendly Place', postcode='MK4 1AB')
crazyGardens  = database['Address'].insert(firstLine='3a Crazy Gardens', postcode='OX1 2ZX')
greatRoad     = database['Address'].insert(firstLine='124 Great Road', postcode='JG6 3TR')

john.relate(friendlyPlace)
owen.relate(greatRoad)
crazyGardens.relate(john)

print john['addresses'].keys()
for address in john['addresses'].values():
   print address['postcode']

print greatRoad['people'].keys()
print owen['addresses']['JG6 3TR']['people'].keys()

john['addresses']['MK4 1AB']['firstLine'] = 'The Cottage, 12 Friendly Place'
print database['Person']['John']['addresses']['MK4 1AB']['firstLine']

connection.close() # Close the connection without saving changes
```

The output is:

```
Created Table
['MK4 1AB', 'OX1 2ZX']
MK4 1AB
OX1 2ZX
['Owen']
['Owen']
The Cottage, 12 Friendly Place
```

You will need to delete the database file 'object-related.db' each time you run the cose so that it can be recreated each time.

1.5.5 Building Queries

You can build complex data structures because each table can contain as many columns, multiple, related and single joins as you like. This isn't a lot of use if you cannot then select the information you want. So far you know how to select data using a series of keys or rowids but the power of SQL is in being able to perform complex queries on that information. The `web.database.object` module has a facility for doing just that.

For this example we create two tables:

```
import web.database, web.database.object, datetime

connection = web.database.connect(type="sqlite",database="object-query.db")
cursor = connection.cursor()

person = web.database.object.Table("Person")
person.addColumn(web.database.object.String(name="firstName"))
person.addColumn(web.database.object.String(name="surname", unique=True, required=True, key=Tru

queryExample = web.database.object.Table('QueryExample')
queryExample.addColumn(web.database.object.Date(name="testDate"))
queryExample.addColumn(web.database.object.Integer(name="testInteger"))
queryExample.addColumn(web.database.object.Integer(name="testNumber"))
queryExample.addColumn(web.database.object.Email(name="email"))

database = web.database.object.Database()
database.addTable(person)
database.addTable(queryExample)
database.init(cursor)

database['Person'].insert(firstName="John", surname="Smith")
database['Person'].insert(firstName="Owen", surname="Jones")
database['QueryExample'].insert(
    testDate=datetime.date(2004,7,11),
    testInteger = 10,
    testNumber = 15,
    email = 'james@example.com'
)
```

To match any rows where the `firstName` is 'John' we make use of the `column` attribute of each table. The column attribute is a magic dictionary which allows you to compare columns to objects in natural Python code to produce a where clause string. It is best explained by an example:

```
>>> where = database['Person'].column['firstName'] == "John"
>>> print where
(Person.firstName = 'John')
>>> rows = database['Person'].select(where=where)
>>> print rows
{'Smith': <web.database.object.Row from Person table, rowid=1, firstName='John', surname='Smith
```

Here are some more examples.

```
>>> column = database['queryExample'].column
>>> column.date == datetime.date(2003,12,12)
"(QueryExample.testDate = '2003-12-12')"
>>> column.integer < 5
"(QueryExample.testInteger < 5)"
```

You can also do more complex queries using AND, OR or NOT. There are two ways of doing this. Both methods are equivalent so please use whichever one you prefer.

Using Methods `AND`, `OR` or `NOT` are methods of the `QueryBuilder` class.

```
>>> where = column.AND(column.email == 'james@jimmyg.org', column.integer < 5)
"(QueryExample.email = 'james@jimmyg.org') AND (QueryExample.testInteger < 5)"
>>> where = column.NOT(column.email == 'james@jimmyg.org')
"NOT (QueryExample.email = 'james@jimmyg.org')"
```

Using Operators The operators `&`, `|` or `~` are defined to mean AND, OR or NOT respectively. You can use them to achieve the same result as above like this:

```
>>> where = (column.email == 'james@jimmyg.org') & (column.integer < 5)
"((QueryExample.email = 'james@jimmyg.org') AND (QueryExample.testInteger < 5))"
>>> where = ~(column.email == 'james@jimmyg.org')
"(NOT (QueryExample.email = 'james@jimmyg.org'))"
```

Note: The brackets **are required** for queries using the `&`, `|` or `~` operators because the operators have the same precedence as other Python operators.

The QueryBuilder is not suitable for all queries. For example it does not currently support the multiple, single or related joins. If you try to access these columns you will get an error saying the key is not found.

However, all is not lost. Since this is an SQL database after all you can use an SQL `cursor.select()` method to get the rowids of the rows you are after and then convert them to objects using the `row()` method of the appropriate table object.

This situation may change with later versions of the module.

How It Works

Each `QueryBuilder` object returns a number of `Query` objects. These `Query` objects have most of there operators overloaded so that they return correctly encoded strings when compared to values or other`Query` objects. Unfortunately it is not possible to use `and`, `or` or `not` operators so instead the `Query` objects use `&`, `|` or `~` instead.

It is actually possible to write your where clauses as SQL if you are using an SQL driver. Changing the first line of our from `where = query.firstName == "John"` to `where = 'Person.firstName="John"'` we have:

```
>>> where = 'Person.firstName="John"'
>>> rows = database['Person'].select(where=where)
>>> print rows
{'Smith': <Row firstName="John", surname="Smith">}
```

and we get the same result. In fact the code `column.firstName == 'John'` from the first example actually returns the SQL encoded string (`'Person.firstName="John"'`) so the two approaches are the same.

There are two advantages of using the `QueryBuilder` approach rather than writing your own where clauses as strings:

1. The `QueryBuilder` automatiacally handles any data conversion. This is pretty trivial in the example above as the string `"John"` requires on conversion but if you are doing a query on a date it would be a little more complicated. Using the `QueryBuilder` takes care of it for you.
2. If a new driver was written for the `web.database.object` module it may require where clauses in a different format from SQL strings. If you write your code using a `QueryBuilder` you can avoid this complication.

Supported Operators

The `QueryBuilder` object supports the following operators:

The three tables below describe the overloaded operators which you can use with `QueryBuilder` objects.

Operator	Description
`<`	Less than.
`<=`	Less than or equal to.
`==`	Equal to.
`<>`	Not equal to.
`>`	Greater than.
`>=`	Greater than or equal to.

Other Operators

Operator	Description
`+`	Add
`-`	Subtract
`*`	Multiply
`/`	Divide
`abs`	Absolute value of
`**`	To the power of
`%`	Mod

Logical Operators

Operator	Description
`&`	AND
`|`	OR
`~`	NOT

Supported Functions

Function	Description
`AND`	Equivalent to using the & operator on a `Query` object.
`OR`	Equivalent to using the — operator on a `Query` object.
`NOT`	Equivalent to using the ~ operator on a `Query` object.

Full Code Listing

Here is a complete code listing so that you can experiment:

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

import sys; sys.path.append('../../../') # show python where the modules are

import web.database, web.database.object
connection = web.database.connect(
    adapter="snakesql",
    database="database-object-query",
    autoCreate = 1,
)
cursor = connection.cursor()

import datetime

person = web.database.object.Table("Person")
person.addColumn(web.database.object.String(name="firstName"))
person.addColumn(web.database.object.String(name="surname", unique=True, required=True, key=Tru

queryExample = web.database.object.Table('QueryExample')
queryExample.addColumn(web.database.object.Date(name="testDate"))
queryExample.addColumn(web.database.object.Integer(name="testInteger"))
queryExample.addColumn(web.database.object.Integer(name="testNumber"))
queryExample.addColumn(web.database.object.Email(name="email"))

database = web.database.object.Database()
database.addTable(person)
database.addTable(queryExample)
database.init(cursor)

if not database.tablesExist():
    database.createTables()
    print "Created Table"

database['Person'].insert(firstName="John", surname="Smith")
database['Person'].insert(firstName="Owen", surname="Jones")

database['QueryExample'].insert(
    testDate=datetime.date(2004,7,11),
    testInteger = 10,
    testNumber = 15,
    email = 'james@example.com'
)

where = database['Person'].column['firstName'] == "John"
print where

rows = database['Person'].select(where=where)
print rows

column = database['queryExample'].column

print column['testDate'] == datetime.date(2003,12,12)
print column['testInteger'] < 5

print column.AND(column['email'] == 'james@jimmyg.org', column['testInteger'] < 5)
```

```
print column.NOT(column['email'] == 'james@jimmyg.org')

print (column['email'] == 'james@jimmyg.org') & (column['testInteger'] < 5)
print ~(column['email'] == 'james@jimmyg.org')

connection.close() # Close the connection without saving changes
```

The output is:

```
Created Table
(Person.firstName = 'John')
{'Smith': <web.database.object.Row from Person table, rowid=1, firstName='John', surname='Smith
(QueryExample.testDate = '2003-12-12')
(QueryExample.testInteger < 5)
(QueryExample.email = 'james@jimmyg.org') AND (QueryExample.testInteger < 5)
NOT (QueryExample.email = 'james@jimmyg.org')
((QueryExample.email = 'james@jimmyg.org')  AND  (QueryExample.testInteger < 5))
( NOT  (QueryExample.email = 'james@jimmyg.org'))
```

You will need to delete the database file 'object-related.db' each time you run the cose so that it can be recreated each time.

1.5.6 Creating Forms/Tables

Lets go back to a simple example:

```
import web.error; web.error.enable()
import web, web.database, web.database.object, os

connection = web.database.connect(type="sqlite", database="object-form.db")
cursor = connection.cursor()

person = web.database.object.Table("Person")
person.addColumn(web.database.object.String(name="firstName", required=True))
person.addColumn(web.database.object.String(name="surname", required=True))

database = web.database.object.Database()
database.addTable(person)
database.init(cursor)
```

If we wanted to create a form to display as HTML to add a new person to the table we could use the following code:

```
>>> form = database['Person'].form()
>>> print form.html()
<form id="Person" class="pythonweb" action="" method="post" enctype="multipart/for
m-data">
<input type="hidden" name="table" value="Person">
<input type="hidden" name="mode" value="submitAdd">
<table border="0">
<tr><td><table border="0" cellpadding="3" cellspacing="0">
<tr>
  <td valign="top">Firstname </td>
  <td>   </td>
  <td valign="top"><input type="text" name="Person.firstName" size="40" maxlengt
h="255" value=""></td>
  <td valign="top"></td>
</tr>
<tr>
  <td valign="top">Surname </td>
  <td>   </td>
  <td valign="top"><input type="text" name="Person.surname" size="40" maxlength=
"255" value=""></td>
  <td valign="top"></td>
</tr>
</table>
</td></tr>
<tr><td> </td></tr>
<tr><td><input type="submit" value="Submit" name="action"></td></tr>
</table>
</form>
```

The form object generated by `form = database['Person'].form()` is a normal `web.form.Form` object and can be used exactly as any `Form` object can. See the documentation for the `web.form` module for more information.

Now we need to get the information the user enters into the database. As with all form objects we follow the following routine once we have a `form` object:

```
form = database['Person'].form() # Continuing from the previous example.
import web
print web.header()               # Print the content-type information
if len(web.cgi) > 1:             # Assume form submitted
    form.populate(web.cgi)
    if form.valid():
        entry = database['Person'].insert(all=form.dict())
        print '<html>%s<p><a href="%s">Go Back</a></html>'%(
            '<h1>Entry Added</h1>' + form.frozen(),
            os.environ['SCRIPT_NAME']
        )
    else:
        '<html><h1>Error</h1>%s</html>'%(
            """<p>There were some invalid fields.
            Please correct them.</p>""" + form.html()
        )
else:
    entries = ''
    for row in database['Person'].values():
        entries += '%s %s<br />'%(row['firstName'] ,row['surname'])
    print "<html>%s<h4>Entries</h4><p>%s</p></html>"%(
        '<h1>Enter Data</h1>'+form.html(),
        entries
    )
```

And that's about it. We populate the form and check it is valid exactly as we would with any form object. The dictionary returned by `form.dict()` can be used in the `database['Person'].insert()` function by specifying it as the *all* parameter.

A handy point to note is that if you don't want the user to be able to add information to all of the form fields you can use the `remove()` method of the form to remove a field from the form by name before creating the HTML version of the form. For example:

```
>>> form = database['Person'].form()
>>> form.remove('surname')
>>> print form.html()
<form id="Person" class="pythonweb" action="" method="post" enctype="multipart/for
m-data">
<table border="0">
<tr><td><table border="0" cellpadding="3" cellspacing="0">
<tr>
  <td valign="top">Firstname </td>
  <td>   </td>
  <td valign="top"><input type="text" name="Person.firstName" size="40" maxlengt
h="255" value=""></td>
  <td valign="top"></td>
</tr>
</table>
</td></tr>
<tr><td> </td></tr>
<tr><td><input type="submit" value="Submit" name="action"></td></tr>
</table>
</form>
```

Full Code Listing

Here is a complete code listing so that you can experiment:

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

# show python where the modules are
import sys; sys.path.append('../'); sys.path.append('../../../')

import web.error; web.error.enable()
import web, web.database, web.database.object, os

connection = web.database.connect(
    adapter="snakesql",
    database="database-object-form",
    autoCreate = 1,
)
cursor = connection.cursor()

person = web.database.object.Table("Person")
person.add(column="String", name='firstName', required=True)
person.addColumn(web.database.object.String(name="surname"))
person.addColumn(
    web.database.object.StringSelect(
        name="profession",
        options=[None, 'Developer', 'Web Developer'],
        displayNoneAs='Not Specified'
    )
)
person.add(column="Bool", name='sex', displayTrueAs='Male', displayFalseAs='Female')
database = web.database.object.Database()
database.addTable(person)
database.init(cursor)

if not database.tablesExist():
    database.createTables()

form = database['Person'].form()
print web.header()               # Print the content-type information
if len(web.cgi) > 1:             # Assume form submitted
    form.populate(web.cgi)
    if form.valid():
        entry = database['Person'].insert(all=form.dict())
        print '<html>%s<p><a href="%s">Go Back</a></html>'%(
            '<h1>Entry Added</h1>' + form.frozen(),
            os.environ['SCRIPT_NAME']
        )
    else:
        print """<html><h1>Error</h1><p>There were some invalid fields.
            Please correct them.</p>%s</html>"""%(form.html())
else:
    entries = '<table border="0"><tr><td>Firstname</td>'
    entries += '<td>Surname</td><td>Profession</td><td>Sex</td></tr>'
    for row in database['Person'].values():
        entries += '<tr><td>%s</td><td>%s</td><td>%s</td><td>%s</td></tr>'%(
            row['firstName'],
            row['surname'],
            row['profession'],
            row['sex']
```

```
            )
        entries += '</table>'
        print "<html>%s<h4>Entries</h4><p>%s</p></html>"%(
            '<h1>Enter Data</h1>'+form.html(),
            entries
        )

    connection.commit() # Save the changes
    connection.close()  # Close the connection
```

You can test this example by starting the test webserver in 'scripts/webserver.py' and visiting http://localhost:8080/doc/src/lib/webserver-web-database-object-form.py on your local machine.

1.5.7 Creating Tables by Defining Classes

As well as defining your table by adding columns to a `web.database.object.Table` object you can define your own class derived from a `web.database.object.Table` object instead. Here is the same database defined above but created using classes instead:

```
import web, web.database, web.database.object

connection = web.database.connect(type="mysql", database="MyDatabase")
cursor = connection.cursor()

class Person(web.database.object.Table):
    def setup(self):
        self.addColumn(web.database.object.String(name="firstName"))
        self.addColumn(web.database.object.String(name="surname"))
        self.addMultiple(name="addresses", foreignTable="Address")

class MyDatabase(web.database.object.Database):
    def setup(self):
        self.addTable(Person())

myDatabase = MyDatabase()
myDatabase.init(cursor)
```

Whilst this may look more complicated it is a more object oriented solution and allows you to build complex table objects with increased functionaility by defining your own objects. For example you could override the `_relatedTableName()` method of both tables to have your own table name created for multiple join tables.

1.5.8 Other Useful Features

This example below demonstrates some other useful methods.

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

import sys; sys.path.append('../../../') # show python where the modules are

import web.database, web.database.object
connection = web.database.connect(
    adapter="snakesql",
    database="database-object-others",
    autoCreate = 1,
```

```
)
cursor = connection.cursor()

person = web.database.object.Table("Person")
person.addColumn(web.database.object.String(name="firstName"))
person.addColumn(web.database.object.String(name="surname"))

database = web.database.object.Database()
database.addTable(person)
database.init(cursor)

if not database.tablesExist():
    database.createTables()
    print "Created Table"

john = database['Person'].insert(firstName="John", surname="Smith")
owen = database['Person'].insert(firstName="Owen", surname="Jones")

print database['Person'].max('rowid')
print database['Person'].max('firstName')
print database['Person'].min('surname')

print database.output()

connection.close() # Close the connection without saving changes
```

The output is:

```
Created Table
2
Owen
Jones
+---------------------+
| Database 'Database' |
+---------------------+
| Person              |
+---------------------+
```

1.5.9 Class Reference

The following sections describe the full Class reference of the three main classes used in the `web.database.object` module.

The Database Object

The `Database` object is used primarily as a container for `Table` objects. The function reference is shown below:

class `Database`(`[`*,name=None*`]`)

name is an arbitrary name for the database used by the `str()` and `repr()` funcitons. If not specified *name* is set to the class name for the database.

`addTable`(*table*)

Adds the table object *table* to the database

init(*cursor*)
Initialise the database by associating it with the `web.database` cursor specified bt *cursor*. Once the database is initialised you can't add or change the table definitions.

createTables()
Create all the necessary tables

dropTables([*ignoreErrors=False*])
Remove all tables defined in the database. If *ignoreErrors* is `True` don't raise an Exception if the table doesn't already exist.

tablesExist()
Return `True` if all the tables exist, `False` otherwise.

table(*name*)
Return the table object for the table named *name*

__getitem__(*name*)
Return the table object for the table named *name*

keys()
Return a tuple containing the names of the tables in the database

values()
Return a tuple containing the `web.database.object.Table` objects for each of the tables in the database

items()
Return a tuple containing 2-tuples of `(`*key*`, `*value*`)` pairs where the *key* is the table name and the *value* is the `web.database.object.Table` object.

dict([*tables=False*], [*rows=False*])
Return all the tables a dictionary indexed by the table names. If *tables* is `True` then each `table` object in the dictionary if also made into a dictionary of key: `Row` pairs. If *rows* is `True` then each `Row` object of each table is made into a dictionary of column name : value pairs, except for single, multiple and related joins columns, since this could result in circular references.

has_key(*key*)
Returns `True` if the database has a table `table`, `False` otherwise

output([*width=80*])
Return a string representation of the database and tables in the form of a table. If *width* is `0` then no wrapping is done. Otherwise the table is wrapped to *width* characters. See the `web.util.table()` documentation for more information.

cursor
The underlying `web.database` cursor.

name
The name of the database specified by the *name* parameter of the constructor. Used by the `str()` and `repr()` funcitons.

`Table` objects can be obtained from a `Database` object by treating the `Database` object as a dictionary of `Table` objects referenced by their names.

For example, if a `Database` object named `database` has tables named `Person` and `Address` you would access the `Person` table with `database['Person']` and the `Address` table with `database['Address']`.

```
>>> database['Person']
<web.database.object.Table 'Person'>
```

The `Database` object also provides a `setup()` method which can be used to setup fields if you want to create your own custom `Database` object.

class **`Database`**(`[`*ignoreCreateAndDrop=False*`]`)

If *ignoreCreateAndDrop* is `True` then the table is not created or dropped when the database methods `createTables()` or `dropTables()` are called.

`addColumn`(*column*)
Add a `web.database.object` column object to the table.

`addMultiple`(*name, foreignTable*)
Add a column named by the string *name* to the table. The column will be used to reference multiple rows from the table named by the string *foreignTable*. The foreign table will have a corresponding addSingle() entry for this table.

`addSingle`(*name, foreignTable*)
Add a column named by the string *name* to the table. The column will contain a reference to a row in the foreign table named by the string *foreignTable*. The foreign table will have a corresponding addMultiple() entry for this table.

`addRelated`(*name, foreignTable*)
Add a column named by the string *name* to the table. The column will contain a reference to any number of rows in the foreign table named by the string *foreignTable*. The foreign table will have a corresponding addRelated() entry for this table and will contain a reference to any number of rows from this table.

`columns`()
Return a tuple of the column names of the table.

`keys`()
Return a tuple containing the keys of the rows in the table.

`values`()
Return a tuple containing the `web.database.object.Row` objects in the table.

`items`()
Return a tuple containing 2-tuples of `(str(`*key*`),` *value*`)` pairs where the *key* is the `web.database.object.Row` key and the *value* is the `web.database.object.Row` object.

`has_key`(*key*)
Returns `True` if the table has a row with a key *key*, `False` otherwise

`dict`(`[`*rows=False*`]`)
Return the rows in the table as a dictionary indexed by string representations of their keys. If *rows* is `True` then each `Row` object is made into a dictionary of column name : value pairs, except for single, multiple and related joins columns, since this could result in circular references.

`create`()
Create the table. **Note:** Usually this is done automatically through the `createTables()` method of the `Database` class.

`drop`()
Drop the table. **Note:** Usually this is done automatically through the `dropTables()` method of the `Database` class.

`exists`()
Return `True` if the table exists in the database, `False` otherwise.

`rowExists`(*rowid*)
Return `True` if the row specified by the integer *rowid* exists in the table, `False` otherwise.

`columnExists`(*name*)
Return `True` if the column *name* exists in the table, `False` otherwise.

`insert`(`[`*all=None*`]`, `[`***params*`]`)
Insert a new row to the table. Either specify the values as a dictionary as the *all* parameter with the column names as keys and the values as the column values **or** specify each column value pair in the form

`colName=value,`. You must use one of the two methods. **Note:** `all` is a reserved word so there should be no confusion between using the two notations.

`delete`(*rowid*)
Delete a row by specifying the `rowid` of the row with the *rowid* parameter. **Warning:** This method does not delete corresponding rows in foreign tables. If you delete a row there will still be references to it in other tables if it contains any colums added by `addMultiple` or `addSingle()` for example. These should be deleted manually. XXX is this a bug or a useful feature?

`row`(*rowid*)
Return the `Row` with the `rowid` specified by the *rowid* parameter.

`__getitem__`(*key*)
Return the `Row` with the `key` specified by the *key* parameter. **Note:** Certain objects such as class objects cannot be used as dictionary keys. All keys are converted to strings using the `str()` function so any object to be used as a key must return a unique value when its `__str__()` is called. This also means that

`select`(*where*[,*order=None*][,*rowids=False*])
Select the `Row` objects specified by the *where* parameter in the oreder specified by the *order* parameter. If *rowids* is `True` then a list of `rowid`s is returned rather than a dictionary of `Row` objects.

`form`([*action=''*][, *method='post'*][, *stickyData={}*][, *enctype='multipart/form-data'*][, *submit='Submit'*][, *modeDict={'mode':'mode', 'table':'table', 'submode':'submode'}*][, *submode='add'*])
Return an empty `web.form` `Form` object to allow data to be added to the table.

`max`(*column*[, *rows='post'*])
Returns the highest value of *column* in the current table. If *rows* is `True` returns a list of rows which have the maximum value of *column*.

`min`(*column*[, *rows='post'*])
Returns the lowest value of *column* in the current table. If *rows* is `True` returns a list of rows which have the minimum value of *column*.

`column`
Magic attribute which allows you to build SQL where clauses in natural Python language. For example:

```
>>> print database['table'].column['column1'] == 23 \
... && database['table'].column['column2'] < datetime.date(2004,12,04)
column1=23 AND column2<'2004-12-24'
```

See the "Building Queries" section for more information.

Table rows can be accessed using the `row()` method or by using the `__getitem__()` method as follows. To return the row with where the `key` is `surname` and you want the row with surname `'Smith'` from the `'Person'` table of the database wrapped by `database` you would do this:

```
>>> database['Person']['Smith']
<web.database.object.Row from 'Person' Table, rowid=1, firstName='John', surname='Smith'>
```

The Row Object

You don't need to create `Row` objects directly. Instead they should be created by using the appropriate methods of the `Table` class.

`Row` objects support the standard comparison operators `<`, `<=`, `>`, `>=`, `==`, `<>` as well as the `len()` function.

class `Row`() **`form`**([*action=''*][, *modeDict=*

Return a `web.form` `Form` object populated with the information from the `Row`

relate(*row*)
Relate this Row to another Row object specified by *row*. Both Rows must be from tables related with addRelated() columns and must not alread be related.

unrelate(*row*)
Unrelate this Row from another Row object specified by *row*. Both Rows must be from tables related with addRelated() columns and must already be related.

isRelated(*row*)
Returns True if the Rows are already related, otherwise returns False.

update([*all=None*],[***params*])
Set multiple values of this row in one go. This currently not optimised so it makes an SQL call for each column set. Set either *all* as a dictionary of column:values pairs or set ***params* by using column=value pairs.

keys()
Return a tuple containing the column names of the fields.

values()
Return a tuple containing values of each field for the current row.

items()
Return a tuple containing 2-tuples of (*key*, *value*) pairs where the *key* is the column name and the *value* is the value of each field for the current row.

has_key(*column*)
Returns True if the row has a column named *column*, False otherwise

dict()
Return the row as a dictionary of column name : value pairs, except for single, multiple and related joins columns, since this could result in circular references.

rowid
The rowid of the row

Each column from the Row can be accessed through a dictionary-like interface. For example to print the value of the column named 'firstName' from the Row with rowid 1 from the 'Person' table in the database database you would use:

```
>>> print database['Person'][1]['firstName']
John
```

1.5.10 Future

This is a list of things currently not included in the module but which may be of use later on:

- Support for functions such as LIKE, BETWEEN, NOW etc.
- Specify different columns for automatic RSS generation
- Build many-to-many support into the query builder
- Deal with '' being interpreted as None

1.6 web.error — Enhanced error handling based on the cgitb module

The web.error module provides enhanced functionality similar to the cgitb module distributed with Python. If an exception is raised the web.error module can catch the error and produce a customised display of the error, the surrounding code and the values of variables in the line which caused the error. It also provides the ability or log errors to a file in various formats.

Using the module you can also provide your own error handling. The example at the end shows you how to create a custom error handler to email error reports to a devloper.

See Also:

cgitb Module Documentation
(http://www.python.org/doc/current/lib/module-cgitb.html)
Find out more about the cgitb module on which this module is based.

1.6.1 Basic Usage

The easiest way of catching and handling errors in Python is to use a try:.. except:.. block around all your code as shown below:

```
try:
    raise Exception('This error will be caught')
except:
    print "An error occured"
```

If you want to produce more detailed error reports you can do something like this:

```
try:
    raise Exception('This error will be caught and nicely displayed')
except:
    import web.error
    print web.error.info(output='traceback', format='text')
```

This will produce a text format output of the traceback information.

If no parameters are specified in the web.error.info() function the result returned is a full HTML debug representation of the error similar to that produced by the cgitb module.

Often a more convenient way to catch errors is by using the web.error.handle() method. If an error is raised it will be automatically handled. The default behaviour is to print a Content-type header followed by HTML information about the error suitable for display in a web browser. This can be done as follows:

```
import web.error
web.error.handle()

raise Exception('This error will be caught and nicely displayed for a web browser')
```

This will produce a full HTML page giving the debug traceback of the error.

Python allows you to put both lines of code on one line to make things look neater if you use a ; so in some of the following samples the error handling initialising will look like this:

```
import web.error; web.error.handle()
```

Agian a full HTML page giving the tracback of the error is displayed together with the HTTP header for display in a browser.

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

# show python where the web modules are
import sys; sys.path.append('../'); sys.path.append('../../../')

import web.error; web.error.handle()
raise Exception('This is a test exception')
```

You can test this example by starting the test webserver in 'scripts/webserver.py' and visiting http://localhost:8080/doc/src/lib/webserver-web-error.py on your local machine.

You can specify the information displayed by the `web.error.handle()` function by passing any parameters that can be passed to the `web.error.info()` function, but if you do this you should also specify the handler you wish to use. The example below prints a text representation of the code which caused the error to a web browser:

```
import web.error
web.error.handle(
    handler  = 'browser',
    output   = 'code',
    format   = 'text',
)
```

Finally, you may wish to use a different error handler, for example you may wish to log the error to a file rather than displaying it. You can specify the *handler* parameter as a string representing the name of the handler you wish to use. Any extra parameters the handler takes can also be specified in the `handle()` function. In this example *filename* is a parameter used by the `file` handler and *output* and *format* are used by the `web.error.info()` function to create a representation of the error:

```
import web.error
web.error.handle(
    handler  = 'file',
    filename = 'test.html',
    output   = 'traceback',
    format   = 'text',
)

raise Exception('This error will be caught appended to the test.html file as a text format trac
```

This time the error will be logged to the file 'test.html' and no output will be printed.

The next sections describe the options for the `error()` and `info()` functions and the various error handlers you can use with the `handle()` function provided in the `web.error.handler` module. The final section describes how you can create custom error handlers for even more advanced error handling.

There is a section in the documentation for the `web.wsgi` module describing how error handling could be performed in a Web Server Gateway Interface application.

1.6.2 Using The info() Function

The `web.error.info()` function returns a representation of the error raised according to the options specified. If no options are specified an HTML debug representation is returned.

The parameters used in the `web.error.info()` can also be used in the `web.error.handle()` function to describe how the handled error should be displayed.

Below is the API reference for the `web.error.info()`.

`web.error.info`([*error*], [*context=5*])

Return a string representing the error according to parameters specified.

output='debug' The output format for the exception. Can be `'traceback'` for a traceback, `'code'` for a code listing or `'debug'` for code and traceback listing suitable for script debugging. If not specified `info()` returns a `ErrorInformation` object.

format='html' The default output format. Can currently be `'text'` or `'html'`.

error An exception tuple as retured by `sys.exc_info()`. If not specified `sys.exc_info()` (which contains the current traceback information) is used.

context The default number of lines of code to display in traceback information. The default is `5`.

1.6.3 Using The handler() Function

If you want more control over the format of the error messages you can use one of the handlers in `web.error.handler`.

The `web.error.handle()` function has the following parameters:

`handle`([*handler*], [***params*])

handler should be a string representing the name of a default handler to use or a custom handler function. The parameters specified by *params* are a combination of parameters used by the handler function chosen and any of the parameters *output*, *format* and *context* used to specify how the error information is displayed.

For example:

```
web.error.handle(
    handler  = 'file',
    filename = 'test.html',
    output   = 'traceback',
    format   = 'text'
)
```

This would append a text format traceback of the error to the 'test.html' file.

The default value for *handler* is `'browser'` and the default display options produce a full HTML debug report so most of the time the following code is sufficient to add at the top of a CGI script:

```
import web.error; web.error.handle()
```

In the example below we specify *format* as `'text'` handler to ouput a text representation of the error:

```
import web.error; web.error.handle(handler='browser', output='debug', format='text')
# This is line 2
# This is line 3
# This is line 4
# This is line 5
raise Exception('This error will be caught and nicely displayed')
# This is line 7
# This is line 8
# This is line 9
# This is line 10
```

This produces the output:

```
Content-type: text/plain

exceptions.Exception
Python 2.2.3 : C:\WINDOWS\Python22\pythonw.exe
Tue Jan 18 20:43:21 2005

    A problem occurred in a Python script.  Here is the sequence of
    function calls leading up to the error, in the order they occurred.

 C:\Work\PythonWeb.org\CVS Branches\Web Modules 0.5\test.py
    4 # This is line 4
    5 # This is line 5
    6 raise Exception('This error will be caught and nicely displayed')
    7 # This is line 7
    8 # This is line 8
Exception undefined
exceptions.Exception: This error will be caught and nicely displayed
    args = ('This error will be caught and nicely displayed',)

    The above is a description of an error in a Python program.  Here is
    the original traceback:

    Traceback (most recent call last):
  File "test.py", line 6, in ?
    raise Exception('This error will be caught and nicely displayed')
Exception: This error will be caught and nicely displayed
```

Note that the handler printed `Content-type` HTTP header. This is so that the output could be displayed in a web browser. If this header wasn't displayed you would see an `Internal Server Error 500` message in the browser.

If you are not writing a web application you might choose to use the `'print'` handler instead of the `'browser'` handler so that the `Content-type` HTTP header is not displayed.

If you want to control the number of lines of code displayed in the error output you can set the *context* parameter. This is the number of lines to be displayed around each line of the traceback. In the example below we set `context=3` to reduce the amount of output:

```
import web.error; web.error.handle(handler='print', output='debug', format='text', context=3)
```

The output is:

```
exceptions.Exception
Python 2.2.3 : C:\WINDOWS\Python22\pythonw.exe
Tue Jan 18 20:45:02 2005

    A problem occurred in a Python script.  Here is the sequence of
    function calls leading up to the error, in the order they occurred.

 C:\Work\PythonWeb.org\CVS Branches\Web Modules 0.5\test.py
    5 # This is line 5
    6 raise Exception('This error will be caught and nicely displayed')
    7 # This is line 7
Exception undefined
exceptions.Exception: This error will be caught and nicely displayed
    args = ('This error will be caught and nicely displayed',)

    The above is a description of an error in a Python program.  Here is
    the original traceback:

    Traceback (most recent call last):
  File "test.py", line 6, in ?
    raise Exception('This error will be caught and nicely displayed')
Exception: This error will be caught and nicely displayed
```

Note that there are fewer lines of code in the code display of the traceback than before.

If *info* is not specified in the `handler()` function, information can be produced in any of the formats supported by `info()` simply by passing the `handler()` function the parameters you would normally pass to `info()` and not specifying the *info* parameter. The exception to this rule it that `handler()` does not accept the *output='class'* option as this does not produce text output.

There are three built-in handlers each of which handle the error information generated in different ways.

`'print'` (`web.error.handler.send()`) Simply prints the error information to the standard output.

`'browser'` (`web.error.handler.browser()`) Sends the error information to the standard output after first sending an HTTP `Content-type` header for display in a web browser. You can over-ride the default header to be sent by specifying *header*. For example `header='text/plain'` would send a `Content-type: text/plain` HTTP header.

`'file'` (`web.error.handler.file()`) Writes the error information to the file specified by *filename*.

If no *filename* is specified, the error information is written to a file in the format `2005-01-18.log`. If *append* is specified `False` the file is overwritten, the default is `True` meaning that error information is appended to the file. If *dir* is specified, files are logged to that directory, the default is to log to the script directory. **Warning:** It is good practice, but not enforced, to specify *dir* otherwise it is possible a logfile will overwrite a file of the same name.

If *message* is specified that message is sent to the standard output. Usually you should set *message* to be something like `web.header('text/plain')+'An error occured and has been logged.'`. Obviously you would not need to specify `web.header('text/plain')` if you are not outputting the error message to a web browser.

All of the handlers are used in the same way.

1.6.4 Using The error() Function

Alternatively you can create an `ErrorInformation` object to display the error information:

```
try:
    raise Exception('This error will be caught and nicely displayed')
except:
    import web.error
    errorInfo = web.error.error()
    print error.textException()
```

This would aproduce the same output described in the previous example.

The `web.error.error()` function returns an `ErrorInformation` object which can be used to format exception tuples in a variety of useful ways. Below is the API reference for the `web.error.error()` function and the `Information` objects returned.

`web.error.error`([*error=sys.exc_info()*], [*context=5*])
Return an `ErrorInformation` object representing the error.

error The traceback tuple you wish to display information for. If not specified the last exception is used.

context The default number of lines of code to display in traceback information. The default is `5`.

class `ErrorInformation`
Error `Information` objects have the following attributes:

`error`
The error tuple specified in the `info()` function or `sys.exc_info()` if no error was specified.

`format`
The default output format of the methods. Can currently be `'text'` or `'html'`.

`pythonVersion`
A string representing the version of Python being used.

`errorType`
The Exception raised

`errorValue`
The error message.

`date`
A string representing the date and time the `Information` object was created. **Note:** This may not be the time the error occured.

`context`
The number of lines of code to display in error information.

Error `Information` objects have the following methods for displaying error information **Note:** Python 2.1 and below do not have the `cgitb` module and so have slightly different implementations of the `html()` and `text()` methods so the output of those methods may be different to the output generated using Python 2.2 and above.

`ouput`(*output*, [*format*], [*error*], [*context*])
output can be `'traceback'` for a traceback, `'code'` for a code listing or `'debug'` for code and traceback listing suitable for script debugging. The method returns the result of calling the respective method below.

`traceback`([*format*], [*error*])
Returns the traceback of the error in the format specified by *format* which can currently be `'text'` or `'html'`. If not specified *format* takes the value of `format`. *error* should be an error tuple as returned by `sys.exc_info()`. If not specified `error` is used.

code([*format*], [*error*], [*context*])
Returns relevant lines of code and variables from the traceback in the format specified by *format* which can currently be `'text'` or `'html'`. If not specified *format* takes the value of `format`. *context* is the number of lines of code to display at each stage in the traceback information. If not specified `context` is used. *error* should be an error tuple as returned by `sys.exc_info()`. If not specified `error` is used.

debug([*format*], [*error*], [*context*])
Returns the traceback of the error in the format specified by *format* together with relevant lines of code and variables. *format* can currently be `'text'` or `'html'`. If not specified *format* takes the value of `format`. *context* is the number of lines of code to display at each stage in the traceback information. If not specified `context` is used. *error* should be an error tuple as returned by `sys.exc_info()`. If not specified `error` is used.

1.6.5 Creating Custom Handlers

If the built-in handlers don't provide the level of cutomisation you require you can create a custom handler.

Handlers are simply callables which take the info string to output as the first parameter and any parameters passed to the `handle()` function as subsequent parameters.

For example:

```
>>> def myHandler(info, message):
...     print message
>>>
>>> import web.error; web.error.handle(myHandler, message="An error occured")
>>> raise Exception('This is an error')
An error occured
```

This example isn't too useful as it always displays the same output. To make it more useful

```
>>> def myHandler(info, message):
...     print message
...     print info
>>>
>>> import web.error
>>> web.error.handle(
...     myHandler,
...     format='text',
...     output='traceback',
...     message='An error occured',
... )
>>> raise Exception('This is an error')
An error occured
exceptions.Exception: This is an error
    args = ('This is an error',)
```

output is used to obtain the error information from the `info()` function which is then sent as the first parameter to the `myHandler` function. *message* is also sent to the `myHandler` function which prints the error message.

This structure allows building very powerful handlers.

1.6.6 Example

Take a look at the example below demonstrating a handler which emails information to a developer:

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

# show python where the web modules are
import sys; sys.path.append('../'); sys.path.append('../../../')

# set your own email here
email = 'james@example.com'

# define our custom handler
def mail(info, email, message, reply):
    import web, web.mail
    web.mail.send(
        msg=info,
        to=email,
        reply=reply,
        subject='Error in website',
        sendmail='usr/bin/sendmail',
        smtp='smtp.ntlworld.com',
        method='smtp',# could use method='sendmail' to send using sendmail.
        type='html',
    )
    print web.header()
    print message

# setup our handler
import web.error
web.error.handle(
    handler = mail,
    output  = 'debug',
    email   = email,
    message = """  <html>
                   <head><title>An Error Occured</title></head>
                   <body><h1>Error Caught</h1>
                   <p>An HTML debug view of the error was sucessfully emailed to %s</p></body>
                   </html>"""%email,
    reply   = 'Developer <%s>'%email
)

# rasie a test exception and wait for the email to arrive
raise Exception('This is a test exception')
```

You can test this example by starting the test webserver in 'scripts/webserver.py' and visiting http://localhost:8080/doc/src/lib/webserver-web-error-mail.py on your local machine.

Warning: If you run this example please make sure you replace the email addresses with your own email address in. You may need to change the path of sendmail or use an SMTP server instead. See the `web.mail` module documentation for help with this.

Note: If an exception occurs in your custom error handling function it may be difficult to track down. You can put your code inside a `try` `except` block and make sure some sensible output is returned in the event of an Exception being raised.

1.6.7 Debugging Code

If you are using the `web.error` module from a command line or supporting webserver such as 'scripts/webserver.py' in the source directory you can raise a `web.error.Breakpoint` Exception and it will be caught and provide a prompt from which you can debug your code.

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

# show python where the modules are
import sys; sys.path.append('../'); sys.path.append('../../../')

import web.error; web.error.handle('debug')

print "Setting value to 5"
value = 5
print "Raising a Breakpoint so you can inspect value"
raise web.error.Breakpoint
print "The program has exited so this will not be printed"
```

This code provides a prompt that can be used as follows:

```
> python "command-web-error-debug.py"
Setting value to 5
Raising a Breakpoint so you can inspect value
> y:\doc\src\lib\command-web-error-debug.py(11)?()
-> raise web.error.Breakpoint
(Pdb) value
5
(Pdb) exit
```

The prompt uses the `pdb` module. To exit the debugger type `exit` and press Return. Code after the `web.error.Breakpoint` Exception was raised is not executed.

See Also:

The Python Debugger To find out more about the Python Debugger see the documentation for the `pdb` module distributed with Python.

1.7 `web.environment` — Tools for seting up an environment

The `web.environment` module provides a single function named `driver()` used to obtain an environment driver to setup or remove an environment.

In the context of a PythonWeb application the environment describes the structures in place in the storage medium and mainly relates to the `web.auth` and `web.session` modules.

Environments are best explained by an example. If you are using a database environment it means that you will be storing session and user information in a series of database tables. Before you can start using these tables they need to be created. The `web.environment` module provides tools to setup the database tables needed. If you were using a file environment, you may need to create the necessary directory structure.

Within an environment, applications can share session and user tables and access each other's information. For example if you had two applications named `guestbook` and `news`, you might want a user named `james` to be able to access both of them without having to sign in to both applications. If the `guestbook` and `news` applications are both in the same environment this is easy since they both use the same session ID and user information.

Each environment has a name. In the context of a database environment the environment name is simply a string which is used to prepend all the environment tables so that multiple environments (with different names) can exist in the same database. This means that you can run all the PythonWeb environments you want to from the same database which is handy if your shared web hosting agreement only gives you access to one database. In the context of a file environment, the environment name might be the name of the directory holding the data files.

1.7.1 Example

In order to use the `web.session` and `web.auth` modules the environment must be setup correctly. You can create the necessary environment using the `web.environment` module's `driver()` function as shown below:

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

# show python where the modules are
import sys; sys.path.append('../'); sys.path.append('../../../')

# Setup a database connection
import web.database
connection = web.database.connect(
    adapter="snakesql",
    database="environment",
    autoCreate = 1,
)
cursor = connection.cursor()

import web.environment
driver = web.environment.driver(
    name='testEnv',
    storage='database',
    cursor=cursor,
)
if not driver.completeEnvironment():
    driver.removeEnvironment(ignoreErrors=True)
    driver.createEnvironment()
    print "Environment created"
else:
    print "Environment already complete"

connection.commit() # Save changes
connection.close()  # Close the connection
```

If none or only some of the tables are present we drop all the existing tables (ignoring errors produced because of missing tables) losing any information they contain and recreate all the tables. We also need to commit our changes to the database so that they are saved using `connection.commit()`.

1.7.2 API Reference

The `EnvironmentDriver` object is used to manipulate the environment. It is obtained from the `driver()` method of the `web.environment` module.

`driver`(*storage*, [*name=''*], [***params*])

Used to return an `EnvironmentDriver` object.

storage The storage driver to be used in the environment. Currently can only be `'database'`.

name The name of the environment (used to prepend database tables if *storage* is `'database'`)

params Any other parameters needed by the `EnvironmentDriver` object. For example if *storage* is `'database'` then *cursor* should also be specified as a valid cursor to the database in which the environment exists.

class `EnvironmentDriver`

`EnvironmentDriver` objects have the following methods:

`completeEnvironment``()`

Returns `True` if all auth and session tables exist, `False` otherwise. XXX Does not check the structure of the tables.

`createEnvironment``()`

Creates all the auth and session tables, raising an error if any already exist.

`removeEnvironment``([`*ignoreErrors=False*`])`

Removes all the auth and session tables, raising an error if any don't exist unless *ignoreErrors* is `True`.

1.8 `web.form` — Construction of persistant forms/wizards for HTML interfaces

The `web.form` module a series of classes and functions for generating and managing persistant HTML forms. As well as basic fields such as `input` or `select` fields, the module provides fields for dates, email addresses, URLs and more. It also supports fields which return Python types, for example the Integer Select field or the Date field.

The `web.form` module also provides a mechanism for automatically handling invalid data and requesting more information from the user.

1.8.1 Introduction

The `web.form` module has three modules containg different types of fields. `web.form.field.basic` provides the standard HTML fields such as `input` boxes or `CheckBoxGroup`s. `web.form.field.typed` provides fields which return typed data such as Dates and `web.form.field.extra` provides fields such as email and URL.

The code below will create an `Integer` field:

```
>>> import web.form, web.form.field.basic as field
>>> input = field.Input(name='box', default='Default Text',
...     description='Input Box:', size=14, maxlength=25)
>>> print input.html()
<input type="text" name="box" size="14" maxlength="25" value="Default Text">
```

This on its own doesn't seem overly useful but when combined with a `web.form.Form` it becomes much more useful. Following on from the previous example:

```
>>> exampleForm = web.form.Form(name='form', action='forms.py', method='get')
>>> exampleForm.addField(input)
>>> exampleForm.addAction('Submit')
>>> print exampleForm.html()
<form name="form" class="lemon" action="forms.py" method="get" enctype="">
<table border="0">
<tr><td><table border="0" cellpadding="3" cellspacing="0">
<tr>
  <td valign="top">Input Box: </td>
  <td>   </td>
  <td valign="top"><input type="text" name="input" size="14" maxlength="25" valu
e="Default Text"></td>
  <td valign="top"></td>
</tr>
</table>
</td></tr>

</table>
</form>
```

In this case a properly formatted form is produced with labels for the fields.

Now in order for this to be useful a mechanism is needed for displaying the form data to the user, validating it, re-displaying it with an error message if it is invalid and then finally accessing the data.

To populate the form with data we use the `web.cgi` object which acts like a dictionary of submitted CGI variables. If the form is submitted then at least one cgi variable will be avaible so if `len(web.cgi)>0` then we know someone is trying to submit form data.

```
>>> if len(web.cgi) > 0:
...    exampleForm.populate(web.cgi)
```

The form will now be populated with the information from the `web.cgi` object. The values submitted to each field may not be of the appropriate types so in order to make sure the information is valid we call the `valid()` method of the form to validate each field.

Again, following on from the previous example:

```
>>> if exampleForm.valid():
...     print "It validated"
...
>>> else:
...     print exampleForm.html()
...
```

If the information entered into the form is not valid `exampleForm.html()` will return a form with the error marked on so that the user can change the field and resubmit the form. Once every field in the form is valid then we can go ahead and access the fields varaibles by their names like this:

```
>>> exampleForm['box']
<Input Class. Name='box'>
>>> exampleForm['box'].value
'Default Text'
```

If a valid value had been submitted then `exampleForm['box'].value` would have returned that value rather than the default.

1.8.2 Form Objects

class **`Form`**(*[name='form'][, action=''][, method='get'][, stickyData=][, enctype=''][, populate=None][, includeName=None]*)

`Form` objects have the following class structure and methods:

`valid`()
: tries to vaildate each field. If any of them contain invalid values returns `False` otherwise returns `True`

`populate`(*form*)
: Populates each field from the value of *form*. *form* should be a `web.cgi` object.

`addField`(*field*)
: Add the field object *field* to the form.

`addAction`(*name*)
: Add a Submit button named *name* to the form. XXX May remove this function in future versions.

`field`(*name*)
: Returns the field object named *name*

`__getitem__`(*name*)
: Returns the field object named *name*

`remove`(*name*)
: Remove the field named *name* from the form

`has_key`(*name*)
: Returns `True` if the form has a field named *name*, `False` otherwise

`values`()
: Return a tuple containing the values of the form fields in the order they were added. The values of the field can be accessed from the `value` attribute of each item in the tuple.

`keys`()
: Return a tuple containing the names of the form fields in the order they were added

`dict`()
: Return a dictionary containing the names and values of the fields as `key:value` pairs

`items`()
: Return a tuple containing 2-tuples of `(`*key*`,` *value*`)` pairs where the *key* is the field name and the *value* is the field object.

`html`()
: Return an HTML representation of the form

`hidden`()
: Return the form as hidden fields

`frozen`(*[action=None]*)
: Return the form as HTML with the values displayed as text and hidden fields instead of the fields. If *action* is specified a Submit button with the value specified by *action* is added to the form

templateDict()

Return the form as a dictionary suitable for use in a template.

The keys include: 'name','action','method','enctype','fields','actions' and 'stickyData'. 'fields' is the key to an array dictionarys containg field information with the keys: 'name','error','description','value' and 'html'. 'stickyData' is the stickyData as hidden fields.

1.8.3 Creating Custom Forms

Rather than creating a web.form.Form object and adding fields, it is also possible to define a custom form object. This has the advantage that you can easily override the default behaviour of the web.form.Form object so that your form will display information in a different way. More information on customising web.form.Form objects is given later on. The code below creates exactly the same form object as we created in the example above.

```
>>> class ExampleForm(web.form.Form):
...     def setup(self):
...         self.addField(
...             field.Input(
...                 name='box',
...                 default='Default Text',
...                 description='Input Box:',
...                 size=14, maxlength=25
...             )
...         )
...         self.addAction('Submit')
...
>>> exampleForm = ExampleForm(name='form', action='forms.py', method='get')
```

1.8.4 Fields

This section provides the full class reference for the web.form module field classes.

The fields in the web.form.field.basic are all designed to provide a functional interface to manipulate standard HTML form fields. Fields in the web.form.field.typed are used to return a typed object such as an Integer or a Date. Fields in the web.form.field.extra provide extra functionality. For example the Email field checks that the string entered could be a valid email address.

All the fields have the parameters, methods and attributes specified in the Field class as well as the parameters, methods and attributes documented in their own section. The Field should not be used in code. It is simply designed to be a base class for all the other classes to be derived from.

web.form.field.basic — Various fields for use with web.form

class Field(*name*,[*default=''*],[*description=''*][,*error=''*][,*required=False*][,*requiredError='Please enter a value'*])

basic.Field is an abstract class from which other classes are derived.

name The name of the field.

default The default value of the field

description A description of the field for use as part of a form

error The error message to initialise the field with

required If `True` a value must be entered. A string `''` is not a valid value. If *required=`True` default* cannot be `''`.

requiredError A string containing the error to display if no value is entered.

`populate`(*values*)
Populates the field from a `web.cgi` object.

values The `web.cgi` object to use.

`valid`(*[value=None]*)
Populates the field from a `web.cgi` object.

value The value to validate. If *value*=`None` then the current value of the field if validated instead. Returns `True` or `False`.

`html`()
Returns the object as an HTML string

`frozen`()
Returns a string representation of the field

`hidden`()
Returns the field as a hidden field

`error`()
Returns the contents of the error string

`setError`(*error*)
Set the error of the field to *error*

`description`()
Returns the contents of the description string

`name`()
Returns the name of the field

`value`
The value of the field

class `Input`(*name[,default=''][,description=''][,error=''][,required=False][,requiredError='Please enter a value'],[,size=40][,maxlength=None]*)
Size of the filed. The number of characters that are displayed

maxlength The maximum number of characters which can be entered into the field. `None` means there is no limit.

class `Password`(*name[,default=''][,description=''][,error=''][,required=False][,requiredError='Please enter a value'],[,size=40][,maxlength=None]*)
Size of the filed. The number of characters that are displayed

maxlength The maximum number of characters which can be entered into the field. `None` means there is no limit.

class `Hidden`(*name,[default=''],[description=''][,error=''][,required=False][,requiredError='Please enter a value']*)
Note: Although you can, it makes little sense to set or read an error on a hidden field.

class `Checkbox`(*name[,default=''][,description=''][,error=''][,required=False][,requiredError='Please enter a value']*)
The default can only be `'on'` or `''`

class `Submit`(*name[,default=''][,description=''][,error=''][,required=False][,requiredError='Please enter a value']*)
Creates a submit button. Same methods and attributes as `basic.Field`

class `Reset`(*name[,default=''][,description=''][,error=''][,required=False][,requiredError='Please enter a value']*)
Creates a reset button. Same methods and attributes as `basic.Field`

class `TextArea`(*name*[,*default=”*][,*description=”*][,*error=”*][,*required=False*][,*requiredError=’Please enter a value’*][,*cols=None*][,*rows=None*])

colsThe number of columns in the field. (The number of characters that are displayed in each row). `None` means not set.

rowsThe number of rows of text on display before the box has to scroll. `None` means not set.

class `File`(*name*[,*default=”*],[*description=”*][,*error=”*][,*required=False*][,*requiredError=’Please enter a value’*])

For file uploads.

Note: If a `web.form.Form` object has a `web.form.field.basic.File` field, the *method* parameter should be set to `'POST'` and the *enctype* should be set to `'multipart/form-data'` for file uploads to work.

class `Select`(*name,options*[,*default=”*][,*description=”*][,*error=”*][,*required=False*][,*requiredError=’Please choose an option’*])

optionsShould be a list or tuple of `[value, label]` pairs. Each `value` or `label` should be a string. Each `value` should be unique.

defaultA string equal to the `value` of the default option.

class `RadioGroup`(*name,options*[,*default=”*][,*description=”*][,*error=”*][,*required=False*][,*requiredError=’Please choose an option’*][,*align=’horiz’*][,*cols=4*])

optionsShould be a list or tuple of `[value, label]` pairs. Each `value` or `label` should be a string. Each `value` should be unique.

defaultA string equal to the `value` of the default option.

alignCan be `'horiz'`, `'vert'` or `'table'`

class `Menu`(*name,options*[,*default=[]*][,*description=”*][,*error=”*][,*required=False*][,*requiredError=’Please choose at least one option’*])

optionsShould be a list or tuple of `[value, label]` pairs. Each `value` or `label` should be a string. Each `value` should be unique.

defaultA list or tuple of strings for all the default values to be selected.

`value`

The value of the field returned as a List.

class `CheckBoxGroup`(*name,options*[,*default=[]*][,*description=”*][,*error=”*][,*required=False*][,*requiredError=’Please choose at least one option’*][,*align=’vert’*][,*cols=4*])

optionsShould be a list or tuple of `[value, label]` pairs. Each `value` or `label` should be a string. Each `value` should be unique.

defaultA list or tuple of strings for all the default values to be selected.

alignCan be `'horiz'`, `'vert'` or `'table'`

colsIf *align=’table’*, *cols* should be an integer specifying the number of columns in the table.

`value`

The value of the field returned as a List.

`web.form.field.typed` — Typed fields for use with `web.form` and `web.database.object`

This module provides fields to support the following data types:

Type	Description
`Char`	A character field taking strings of length 1
`String`	A string field taking strings of up to 255 characters
`Text`	A text field for storing large amounts of text (up to 16k characters)
`Integer`	An integer field taking any integer that is a valid Python integer (but **not** long)
`Float`	A float field taking Python float values
`Date`	A date field. Takes values in the form of python `datetime` objects. Only stores days, months and years, any other inf
`Time`	A time field. Takes values in the form of python `datetime` objects. Only stores hours, minutes and seconds, any othe
`DateTime`	A datetime field. Takes values in the form of python `datetime` objects. Only stores days, months, years, hours, minu

Note: These `Field` objects correspond to the fields used by the `web.database` module. There is a reason for this; the `Column` objects in the `web.database.object` module are each derived from a `web.form.field.typed Field`. This means that columns from a `web.database.object.Row` are also valid form fields. This is used in the `web.database.object` classes to automatically generate and validate forms which can be used seamlessly and easily submit and edit data in a database.

Each of the data types listed below has three types of field specified:

`Free` Allows the user to specify any value

`Select` Allows the user to choose one value from a number of values specified

`CheckBoxGroup` Allows the user to choose more than one option

The typed field classes have the same interface as their basic equivalents except that:

1. The Select and CheckBoxGroup classes take lists of their respective data types rather than `value, label` pairs.

2. The fields return their repsective Python object (or list of objects) when their `.value` attribute is called.

All the fields can either take their respective data type or the value `None` as possible values for the field. The only complications are the `web.form.field.typed.String`, `web.form.field.typed.Text` and class-web.form.field.typed.Char objects.

If someone enters no information into a String field there is a choice of whether to treat this as a null string `''` or a NULL value `None`. To specify which behaviour you would like the `web.form.field.typed.String` object accepts the parameter *treatNullStringAsNone* which takes a default value of `True`. The `web.form.field.typed.Char` and `web.form.field.typed.Text` fields also accept the *treatNullStringAsNone* parameter.

The `web.form.field.typed.Integer` field also takes the parameters *min* and *max* to specify the minimum and maximum values and the parameters *minError* and *maxError* to specify the errors to display if the values are outside the specified minimum and maximum.

One more complication is how to display `None` values in the `web.form.field.typed.StringSelect` and classweb.form.field.typed.CharSelect objects. If you choose the string `'None'` to display it how do you distinguish `None` from `'None'`? Any value you choose could be confused with another string. The solution is to set a string value to dispaly `None` that isn't another value in the *options*. You can set this using the *displayNoneAs* parameter. `None` values for the other Select fields are just displayed as `''`.

`web.form.field.extra` — Extra fields for use with `web.form`

This module provides two classes: `URL` and `Email`. Both these classes behave exactly the same as the `web.form.field.typed.String` class except that they only accept as values strings that are URLs or Emails respectively.

For example:

```
>>> import web.form.field.extra as field
>>> email = field.Email(name='emailField')
>>> print email.html()
<input type="text" name="emailField" value=""> <small>eg. james@example.com</small>
>>> email.value = 'this is not an email address'
>>> email.valid()
0
>>> print email.error()
Please enter a valid email address. eg. james@example.com
>>> email.setError('')
>>> email.value = 'james@example.com'
>>> email.valid()
1
```

1.8.5 Basic Fields Example

As an example showing the internal workings of the the form module.

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

"Forms example."

import sys, re, os
sys.path.append('../')
sys.path.append('../../')

import web.error; web.error.handle()
import web, web.form, web.form.field.basic, web.util

class ExampleForm(web.form.Form):

    def setup(self):
        self.addField(web.form.field.basic.Input('input', 'Default Text', 'Input Box:', size=14
        self.addField(web.form.field.basic.Password('password', 'Default Text', 'Password Field
        self.addField(web.form.field.basic.Hidden('hiddenfield', 'Default Text','Hidden Field')
        self.addField(web.form.field.basic.CheckBox('checkbox', 'DefaultValue', 'Checkbox:'))
        self.addField(web.form.field.basic.Button('button', 'Button Label', 'Button:'))
        self.addField(web.form.field.basic.TextArea('textarea', 'Text Area\n-----\nText', 'Text
        self.addField(web.form.field.basic.RadioGroup('radiogroup', [('1','one'),('2','two'),('
        self.addField(web.form.field.basic.Menu('menu', [('1','one'),('2','two'),('3','three')]
        self.addField(web.form.field.basic.Select('select', [('1','one'),('2','two'),('3','thre
        self.addField(web.form.field.basic.CheckBoxGroup('checkboxgroup', [('1','one'),('2','tw
        self.addField(web.form.field.basic.Reset('reset', 'Reset', 'Reset Button:'))
        self.addField(web.form.field.basic.Submit('submit', 'Submit', 'Submit Button (normally

        # The preffered way of adding submit buttons is as actions so Submit buttons are normal
        self.addAction('Validate This Form')

    def valid(self):
        if web.form.Form.valid(self):
            validates = True
            if self.get('input').value == 'Default Text':
                self.get('input').setError("ERROR: You must change the text in the input box.")
                validates = False
            return validates
```

```
        else:
            return False

# Print the HTTP Header
print web.header('text/html')


# Create a form

exampleForm = ExampleForm('form', os.environ['SCRIPT_NAME'], 'get')

if len(web.cgi) > 0: # Form submitted
    # Populate form with the values from get.

    # Prepare form values:
    values = {}

    for k in web.cgi.keys():
        values[k] = [k,str(web.cgi[k])]
        if exampleForm.has_key(k):
            values[k].append(exampleForm[k].value)
            values[k].append(exampleForm[k].error())
    exampleForm.populate(web.cgi)

    for k in web.cgi.keys():
        if exampleForm.has_key(k):
            values[k].append(exampleForm[k].value)
            values[k].append(exampleForm[k].error())

    if exampleForm.valid():
        for k in web.cgi.keys():
            if exampleForm.has_key(k):
                values[k].append(exampleForm[k].value)
                values[k].append(exampleForm[k].error())
        valueText = ''
        for k in exampleForm.keys():
            if web.cgi.has_key(k):
                valueText += '<strong>%s</strong><br>'%values[k][0]
                valueText += '<table border="0">'
                valueText += '<tr><td>Create</td><td>%s</td></tr>'%web.encode(repr(values[k][2]
                valueText += '<tr><td>Error</td><td>%s</td></tr>'%web.encode(repr(values[k][3])
                valueText += '<tr><td>Populate</td><td>%s</td></tr>'%web.encode(repr(values[k][
                valueText += '<tr><td>Error</td><td>%s</td></tr>'%web.encode(repr(values[k][5])
                valueText += '<tr><td>Validate</td><td>%s</td></tr>'%web.encode(repr(values[k][
                valueText += '<tr><td>Error</td><td>%s</td></tr>'%web.encode(repr(values[k][7])
                valueText += '</table><br><br>'
        print "<html><head><title>Form Test - Validated</title></head><body>\n<h1>It Validated!
    else:
        for k in web.cgi.keys():
            if exampleForm.has_key(k):
                values[k].append(exampleForm[k].value)
                values[k].append(exampleForm[k].error())
        valueText = ''
        for k in exampleForm.keys():
            if web.cgi.has_key(k):
                valueText += '<strong>%s</strong><br>'%values[k][0]
                valueText += '<table border="0">'
                valueText += '<tr><td>Create</td><td>%s</td></tr>'%web.encode(repr(values[k][2]
                valueText += '<tr><td>Error</td><td>%s</td></tr>'%web.encode(repr(values[k][3])
```

```
                valueText += '<tr><td>Populate</td><td>%s</td></tr>'%web.encode(repr(values[k][
                valueText += '<tr><td>Error</td><td>%s</td></tr>'%web.encode(repr(values[k][5])
                valueText += '<tr><td>Validate</td><td>%s</td></tr>'%web.encode(repr(values[k][
                valueText += '<tr><td>Error</td><td>%s</td></tr>'%web.encode(repr(values[k][7])
                valueText += '</table><br><br>'
            print "<html><head><title>Form Test - Errors</title></head><body>\n<h1>Please Check Ent
    else:
        print "<html><head><title>Form Test</title></head><body>\n<h1>Welcome Please Fill In The Fo
```

You can test this example by starting the test webserver in 'scripts/webserver.py' and visiting http://localhost:8080/doc/src/lib/webserver-web-form.py on your local machine.

1.8.6 Typed Fields Example

As an example showing the how to use the typed fields, not the use of `None` values.

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

"Forms example."

import sys, re, os
sys.path.append('../')
sys.path.append('../../')
import web.error; web.error.handle(handler='browser', output='debug', format='html')
import web, web.form, web.form.field.basic, web.util
import web.form.field.typed

class ExampleForm(web.form.Form):

    def setup(self):
        self.add(
            field='input',
            name='input',
            default='Default Text',
            description='Input Box:',
            size=14,
            maxlength=25
        )
        self.addField(web.form.field.typed.String('string', default=None, treatNullStringAsNone
        self.addField(web.form.field.typed.String('string None', default=None, treatNullStringA
        self.addField(web.form.field.typed.Bool('bool', required=True))
        self.addField(web.form.field.typed.Bool('bool None', default=None))
        self.addField(web.form.field.typed.Text('text', default=None, treatNullStringAsNone=Fal
        self.addField(web.form.field.typed.Text('text None', default=None, treatNullStringAsNon
        self.addField(web.form.field.typed.Integer('integer', default=None, required=True))
        self.addField(web.form.field.typed.DateTime('datetime', default=None, required=True))
        self.addField(web.form.field.typed.StringSelect('stringselect', options=[None, 'String'
        self.addField(web.form.field.typed.FloatSelect('floatselect', options=[None, 1]))
        self.addField(web.form.field.typed.FloatCheckBoxGroup('floatcheckboxgroup', options=[1,
        # The preffered way of adding submit buttons is as actions so Submit buttons are normal
        self.addAction('Validate This Form')

    def valid(self):
        if web.form.Form.valid(self):
            validates = True
            if self.get('input').value == 'Default Text':
                self.get('input').setError("ERROR: You must change the text in the input box.")
```

```
                validates = False
            return validates
        else:
            return False

# Print the HTTP Header
print web.header('text/html')

# Create a form
exampleForm = ExampleForm('form', 'webserver-web-form-typed.py', 'get')

if len(web.cgi) > 0: # Form submitted
    # Populate form with the values from get.

    # Prepare form values:
    values = {}

    for k in web.cgi.keys():
        values[k] = [k,str(web.cgi[k])]
        if exampleForm.has_key(k):
            values[k].append(exampleForm[k].value)
            values[k].append(exampleForm[k].error())
    exampleForm.populate(web.cgi)

    for k in web.cgi.keys():
        if exampleForm.has_key(k):
            values[k].append(exampleForm[k].value)
            values[k].append(exampleForm[k].error())

    if exampleForm.valid():
        for k in web.cgi.keys():
            if exampleForm.has_key(k):
                values[k].append(exampleForm[k].value)
                values[k].append(exampleForm[k].error())
        valueText = ''
        for k in exampleForm.keys():
            if web.cgi.has_key(k):
                valueText += '<strong>%s</strong><br>'%values[k][0]
                valueText += '<table border="0">'
                valueText += '<tr><td>Create</td><td>%s</td></tr>'%web.encode(repr(values[k][2]
                valueText += '<tr><td>Error</td><td>%s</td></tr>'%web.encode(repr(values[k][3])
                valueText += '<tr><td>Populate</td><td>%s</td></tr>'%web.encode(repr(values[k][
                valueText += '<tr><td>Error</td><td>%s</td></tr>'%web.encode(repr(values[k][5])
                valueText += '<tr><td>Validate</td><td>%s</td></tr>'%web.encode(repr(values[k][
                valueText += '<tr><td>Error</td><td>%s</td></tr>'%web.encode(repr(values[k][7])
                valueText += '</table><br><br>'
        print """<html><head><title>Form Test - Validated</title></head><body>\n<h1>It Validate
        %s\n<hr>\n<h2>Values</h2>%s</body></html>"""%(exampleForm.frozen(), valueText)
    else:
        for k in web.cgi.keys():
            if exampleForm.has_key(k):
                values[k].append(exampleForm[k].value)
                values[k].append(exampleForm[k].error())
        valueText = ''
        for k in exampleForm.keys():
            if web.cgi.has_key(k):
                valueText += '<strong>%s</strong><br>'%values[k][0]
                valueText += '<table border="0">'
                valueText += '<tr><td>Create</td><td>%s</td></tr>'%web.encode(repr(values[k][2]
```

```
                valueText += '<tr><td>Error</td><td>%s</td></tr>'%web.encode(repr(values[k][3]))
                valueText += '<tr><td>Populate</td><td>%s</td></tr>'%web.encode(repr(values[k][
                valueText += '<tr><td>Error</td><td>%s</td></tr>'%web.encode(repr(values[k][5]))
                valueText += '<tr><td>Validate</td><td>%s</td></tr>'%web.encode(repr(values[k][
                valueText += '<tr><td>Error</td><td>%s</td></tr>'%web.encode(repr(values[k][7]))
                valueText += '</table><br><br>'
            print """<html><head><title>Form Test - Errors</title></head><body>\n
            <h1>Please Check Entries</h1>%s\n<hr>\n<h2>Values</h2>
            %s</body></html>"""%(exampleForm.html(), valueText)
    else:

        print """<html><head><title>Form Test</title></head><body>\n
        <h1>Welcome Please Fill In The Form</h1>%s\n<hr>
        </body></html>"""%(exampleForm.html())
```

You can test this example by starting the test webserver in 'scripts/webserver.py' and visiting http://localhost:8080/doc/src/lib/webserver-web-form-typed.py on your local machine.

1.9 `web.image` — Create and manipulate graphics including JPG, PNG, PDF, PS using `PIL`

The `web.image` currently contains one sub package for creating simple graphs from data.

1.9.1 `web.image.graph` — Create graphs

`html2tuple`(*htmlColorCode*)

Returns a colour tuple of `(R, G, B)` in hex from an HTML colour code such as `#ffffff`. The return value from this function can be used to specify colours in the `graph` module.

htmlColorCode The html colour code to convert.

The `web.image.graph` module is used to create PNG or similar graphs for use on web pages.

1.9.2 Command Line Example

Currently the module only works with positive values for the axes and requires the presence of the 'Arial.ttf' font by default. This modules should be considered an early implementation. You should ensure the values you choose produce a nice looking graph because there is very little error checking and the values you choose may not result in the graph displaying correctly.

Here as an example showing the useage of the three main classes:

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

import sys; sys.path.append('../../../') # show python where the modules are
import web.image.graph

graph = web.image.graph.ScatterGraph(
    xAxis={'max':200, 'unit':20, 'label':'Value 1 /cm^2'},
    yAxis={'max':200, 'unit':20, 'label':'Value 2 /cm^2'},
    points=[(0,0),(13,68),(200,200)],
    size=(500, 300),
    bgColor=(240,240,240),
    title='Test Graph'
```

```
)
graph.save('scatter.ps')

graph = web.image.graph.BarGraph(
    xAxis={'max':200, 'unit':20, 'label':'Value 1 /cm^2'},
    yAxis={'max':200, 'unit':20, 'label':'Value 2 /cm^2'},
    points=[10,20,40,50,200,89, 30, 60, 70, 60],
    size=(500, 300),
    bgColor=(240,240,240),
    title='Test Graph'
)
graph.save('bar.png')

graph = web.image.graph.PieChart(
    points={
        'food':10,
        'numbers':20,
        'numbers2':30,
    },
    size=(500, 300),
    bgColor=(240,240,240),
    title='Test Graph'
)
graph.save('pie.jpg')

graph = web.image.graph.BarGraph(
    xAxis={'max':200, 'unit':20, 'label':'Value 1 /cm^2'},
    yAxis={'max':200, 'unit':20, 'label':'Value 2 /cm^2'},
    points=[10,20,40,50,200,89, 30, 60, 70, 60],
    size=(500, 300),
    bgColor=web.image.html2tuple('#F0F0F0'),
    title='Test Graph'
)
fp = open('string.pdf','wb')
fp.write(graph.toString('pdf'))
fp.close()
```

Note: The format of the image saved depends on the extension used. Currently supported are `'.png'`, `'.jpg'`, `'.ps'`. JPEG is a lossy compression method and so the graphics produced as JPEGs may not be as good quality as the others. The receommended format to use is `'.png'`. ust save your files with a `.png` extension to have PNG output.

1.9.3 Webserver Example

It is useful to be able to produce graphs in a script and then return them. The example below generates a graph. It can be used in an HTML tage like this `<img src="webserver-web-image-graph.py" alt="Graph" />`.

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

"""Graph Generation Example.
<img src="webserver-web-image-graph-web.py" alt="Graph" />"""

# show python where the web modules are
import sys, os
sys.path.append('../')
sys.path.append('../../../')
```

```
import web.error; web.error.handle()
import web.image, web.image.graph

graph = web.image.graph.BarGraph(
    xAxis={'max':10, 'unit':1, 'label':'Days Since Send'},
    yAxis={'max':10, 'unit':1, 'label':'Number of Page Views'},
    points=[1,5,7,8,4,3,6,8,0,1],
    size=(500, 300),
    bgColor=web.image.html2tuple('#ffffff'),
    barColor=web.image.html2tuple('#000080'),
    title='Page View Rate For Newsletter',
)
print web.header('image/png'), graph.toString('png')
```

You can test this example by starting the test webserver in 'scripts/webserver.py' and visiting http://localhost:8080/doc/src/lib/webserver-web-image-graph.py on your local machine. You will need the `Arial.ttf` font somewhere on your system where Python can find it.

1.10 `web.mail` — Simple function to send email using `email`

The mail module provides a simple function `send()` which can be used to send emails as shown in the example below:

```
import web.mail
web.mail.send(
    msg        = "Hello James!",
    to         = 'james@example.com',
    replyName  = 'James Gardner',
    replyEmail = 'james@example.com',
    subject    = 'Test Email',
    sendmail   = '/usr/bin/sendmail',
    method     = 'sendmail'
)
```

To send the same email via SMTP instead of using 'Sendmail' you would use:

```
import web.mail
web.mail.send(
    msg        = "Hello James!",
    to         = 'james@example.com',
    replyName  = 'James Gardner',
    replyEmail = 'james@example.com',
    subject    = 'Test Email',
    smtp       = 'smtp.ntlworld.com',
    method     = 'smtp'
)
```

If you get an error like `socket.error: (10060, 'Operation timed out')` it is likely that the SMTP address you specified either doesn't exist or will not give you access.

Function Definition:

`mail`(*msg, to,* [*subject=''*], [*method*], [*smtp*], [*sendmail*], [*blind*], [*reply*], [*replyName*], [*replyEmail*], [*type**msg***])

Text of the message

to A list of recipient addresses in the form: addr@addr.com separated by commas

subject Email subject line

method Describes which method to use to send the email. Can be `'smtp'` or `'sendmail'`. Only needs to be specified if both `smtp` and `sendmail` are specified otherwise the method that is defined is used.

smtp SMTP server address

sendmail Sendmail path

blind `True` if recipients are to be blocked from seeing who else the email was sent to.

replyName The name of the person sending the email.

replyEmail The address of the person sending the email.

reply The name and address of the person sending the email in the form: `"sender name <addr@example.com>"`. Should only be specified if *replyName* and *replyEmail* are not specified.

The module also provides a method `buildReply()` which can be used to put the name and email address into the format required for the *reply* parameter of the `send()` method:

```
>>> import web.mail
>>> web.mail.buildReply('James Gardner, 'james@example.com')
James Gardner <james@example.com>
```

type The second part of the content-type, eg `'plain'` for a plain text email, `'html'` for an HTML email.

1.10.1 Example

Below is an example demonstrating some of the features which you can use to test the module:

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

"Test program to send mail to recipients."
import sys; sys.path.append('../../../') # show python where the modules are

import web.mail
testAddr = raw_input('Email address 1 to recieve tests (will receive 6 emails): ')
testAddr2 = raw_input('Email address 2 to recieve tests (will receive 2 emails): ')
if raw_input('Run the 6 SMTP tests:  [y/n] ') == 'y':
    smtp = raw_input('SMTP server address: ')
    print "Running SMTP Test...."
    counter = 1
    for blind in [True, False]:
        for to in [testAddr, [testAddr], [testAddr, testAddr2]]:
            web.mail.send(
                msg="Hello User!\n\nBlind: " + str(blind),
                to=to,
                reply=web.mail.buildReply('web.mail Test',testAddr),
                subject="SMTP Test "+ str(counter),
                smtp=smtp,
                blind=blind,
                method='smtp'
            )
            print "Sent message %s."%counter
            counter += 1
    print "Done... check your mail!\n"
```

```
if raw_input('Run the 6 sendmail tests: [y/n] ') == 'y':
    sendmail = raw_input("Sendmail Path (usually /usr/lib/sendmail): ")
    print "Running Sendmail Test...."
    counter = 1
    for blind in [True, False]:
        for to in [testAddr, [testAddr], [testAddr, testAddr2]]:
            web.mail.send(
                msg="Hello User!\n\nBlind: " + str(blind),
                to=to,
                reply=web.mail.buildReply('web.mail Test',testAddr),
                subject="Sendmail Test "+ str(counter),
                sendmail=sendmail,
                blind=blind,
                method='sendmail'
            )
            print "Sent message %s."%counter
            counter += 1
    print "Done... check your mail!"
```

See Also:

`email` Module Documentation
(http://www.python.org/doc/current/lib/module-email.html)

The `email` module distributed with Python has a much broader API for constructing emails and should be consulted if you plan to anything complicated such as emailing attachements.

1.11 `web.session` — Persistent storage of sessions and automatic cookie handling

The session module is designed to provide the ability to manage sessions to allow data to persist between HTTP requests. It is not designed to any authorisation features the `web.auth` is for that purpose.

1.11.1 Background Information

Note: This section is meant as a guide for beginners and can be safely skipped if you already understand the principles of session handling in a multi-application environment.

The HTTP Protocol is Stateless

When discussing sessions the comment ”The HTTP protocol is a stateless protocol, and the Internet is a stateless development environment” is often used. This simply means that the HyperText Transfer Protocol that is the backbone of the Web is unable to retain a memory of the identity of each client that connects to a Web site and therefore treats each request for a Web page as a unique and independent connection, with no relationship whatsoever to the connections that preceded it.

For viewing statically generated pages the stateless nature of the HTTP protocol is not usually a problem because the page you view will be the same no matter what previous operations you had performed. However for applications such as shopping carts which accumulate information as you shop it is extremely important to know what has happened previously, for example what you have in your basket. What is needed for these applications is a way to ”maintain state” allowing connections to be tracked so that the application can respond to a request based on what has previously taken place.

Session IDs

There are two main ways in which applications can recognise a user, both of which involve identifying the connection using a short string known as a session ID.

In the first method every URL on a web page if modified with the session ID on the end so that whenever a user clicks on a link the application is aware of which user is requesting a page. One drawback of this approach is that the session ID can easily be read as it will appear in the address bar of your browser so that a malicious onlooker could read the session ID and type the URL into another computer. The application would think that both users were the same person because both would be using the same session ID.

The second method involves cookies. A cookie is a simple text file stored by your browser which contains `key:value` pairs of text. When you request a web page, if your browser has a cookie registered for that domain it sends the information to the web server before retrieving the page. The web browser can then react to the information in the cookie before returning the page. If a session ID is stored in a cookie then the application can read the session ID and therefore keep track of your connection history. Using cookies in this way is more secure that appending a session ID to a URL because only your web browser knows the cookie information and it cannot be read from your address bar.

Information Storage

The next step is to use a session ID to store information. One option is to put information into hidden fields in forms and append the information to URLs. This becomes difficult for large amounts of information. A much better way is to store the information in a server based on which session ID is accessing the website which is what session handling modules help with.

Multiple Applications

In a real world situation there might be many different applications storing information in a session store. If they weren't all carefully planned it would be easy for one application to over-write another's information. One solution might be to setup different session stores for each application but this would require tracking multiple session IDs. A better approach is for the session application to provide a session store to each application but handle the creation and expiry of the sessions collectively. This is exactly what the `web.session` does.

The HTTP Protocol and Cookie Handling

One issue which can cause problems with applications is the way session modules send cookies. When writing a normal CGI application which simply prints information to the client's web browser you must send the HTTP header information to the web browser before the main body of the web page. Once the browser receives two carriage return characters `\n\n` it knows that the information that follows is a web page and not more HTTP headers. This is why you always print `Content-type:  text/html\n\n` before printing `<html> etc`.

The session handling module also prints HTTP headers to set cookie information and so it is important that the session handling code appears before you send the `\n\n` characters to your browser otherwise the page may not display correctly. This is often hard to spot in application environments like mod_python or the WSGI where header information is separated from page content. If you have problems with the session code because pages are not displaying correctly check the headers are being sent correctly.

Of course the `web.session` module allows you to disable this automatic cookie header printing and handle the cookie headers in the way your application wants. This is described in the section `Custom Cookie Handling` later on in the documentation.

1.11.2 Session Module Overview

The `web.session` module provides two different objects to help users manage sessions. These are:

`SessionManager` These objects are used to handle creation, expiry, loading, validity checks and cleanup of sessions, the handling of cookies and the creation of store objects.

`SessionStore` These are the objects used to set and retrieve the values being stored for the particular application.

To begin using the session store for your application you must perform the following steps:

1. Create a manager object and load an existing session or create a new session
2. Obtain an application store object

1.11.3 Creating a basic session environment

The `web.session` module is designed so that the data can be stored in ways using different drivers. Currently only a database storage driver and file driver exist allowing auth information to be stored in any relational database supported by the `web.database` module or in a sirectory structure.

In this example we are using a database to store the auth and session information so we setup a database cursor named `cursor` as described in the documentation for the `web.database` module.

```
import web.database
connection = web.database.connect(adapter='snakesql', database='test', autoCreate=1)
cursor = connection.cursor()
```

Next we need to create the necessary tables which will store the session information. To do this we use the `SessionManager` object.

```
manager = web.session.manager(driver='database', cursor=cursor)
```

If we haven't already created the session tables we can do so like this:

```
if not manager.completeSessionEnvironment():
    manager.removeSessionEnvironment(ignoreErrors=True)
    manager.createSessionEnvironment()
```

Alternatively the session manager can also take the *autoCreate=1* parameter to automatically create the necessary tables in exactly the way described above automatically.

If any of the tables are missing, this code removes all existing tables thereby destroying all the data they contain (ignoring errors produced because of missing tables) and re-creates all the tables.

```
connection.commit()
```

The `connection.commit()` saves the changes to the database.

`web.session.manager()` also takes a range of parameters such as *expire* to set the length of time in seconds the session is valid for or *cookie* to set the cookie options. The full list of options is listed in the API reference section but the default values are usually adequate.

1.11.4 Loading a Session

If we are using cookies to store session IDs we use the code manager object to read the session ID of the current user from the cookie using the manager object's `cookieSessionID()` method otherwise we obtain the session ID in whichever way is appropriate for our application.

```
sessionID = manager.cookieSessionID()
```

Once a session ID is obtained we can load the session. The manager object's `load()` method will attempt to load a session from a session ID. If *sessionID* is not specified it will be obtained from a cookie. If the session is not valid or does not exist the method returns `False` and sets the error to the manager object's `error` attribute.

If the session does not exist or has expired we need to create a new session using `create()`. This will also automatically send cookie headers to set the session ID unless `session.create(sendCookieHeaders=False)` is used, in which case you can still print the headers manually using `sendCookieHeaders()`.

```
if not manager.load(sessionID):
    newSessionID = manager.create(sendCookieHeaders=False)
    manager.sendCookieHeaders()
```

If you are using a CGI environment all this code can be simplified to just the following:

```
if not manager.load():
    newSessionID = manager.create()
```

The `load()` method obtains a session ID automatically if not present and `create()` automatically sends the headers.

1.11.5 Multiple Applications and Stores

Once the session is successfully loaded we can create a store object.

The `web.session` module supports using multiple applications within an environment. Each application has its own session store and can only access values in its own store to avoid the risk of over-writing another application's data. This has the benefit of allowing applications to share the same session ID and cookie.

Application names can be a string made up of the characters `a-z`, `A-Z`, `0-9` and `-_.`. The application name must be between 1 and 255 characters in length. The application names do not have to be the same as application names used by the `web.auth` module, although these are the most appropriate choices.

It is important you choose a name for your application which is unique in the environment you are using. For example if you are also using the `web.auth` module you should not use the application name `'auth'` since the `web.auth` module used the application name `'auth'` to store its values.

To access a store using the `store()` method of the manager object you must specify an application name, for example:

```
store = manager.store('testApp')
```

1.11.6 Using Stores

We can now use our store variable to set and retrieve values from our `testApp` application's session store. Below is a demonstration of the functional interface:

```
store = session.store('testApp')
>>> store.set(key='first',value='This is the fist key to be set!')
>>> print store.get(key='first')
This is the fist key to be set!
>>> print store.keys()
['first']
>>> store.delete(key='first')
>>> store.has_key(key='first')
0
```

Alternatively we can treat the store object as a dictionary:

```
>>> store['first'] = 'This is the fist key to be set!'
>>> print store['first']
This is the fist key to be set!
>>> print store.keys()
['first']
>>> del store['first']
>>> store.has_key('first')
0
```

Both versions behave in exactly the same way and any Python value that can be pickled by the `pickle` module can be set and retrieved from the store so you can store strings, numbers and even classes and all the information will be available for each request until you remove it or the session expires.

One other useful method of the store object is the `empty()` method. This is used to remove all information from an application's session store. This is a better way of removing information than using the manager's `destroy()` method since `destroy()` will also remove all the information from other application's stores which might cause those applications to crash if the store is currently being accessed.

1.11.7 Managing Sessions

The following sections describe more about the manager object and how it can be used to manage sessions.

Checking Session Existence or Validity

If for any reason you have an application which has run for a long time, it is possible that the session has expired since the session was originally created or loaded.

To check if a session is still valid use `manager.valid()`. The `valid()` method returns `True` if the session is valid, `False` otherwise and raises a `SessionError` if the session no longer exists.

It is also conceivably possible that the session has been cleaned up and no longer exists. To check if a session exists use `manager.exists()`. The `exists()` method returns `True` if the session exists, `False` otherwise but makes no comment on whether or not it is still valid or has expired.

Destroying Sessions

Once a session has expired the data cannot be accessed by the session module. If a user tries to access an expired session, the session is destroyed immediately.

You can also manually destroy the session using the `destroy()` method. However it is highly recommended that you do not destroy sessions in this way as other applications may be using the session and my crash if during the course of program execution the session information is removed. Instead you can use the `empty()` method of the store instance to remove all store information for your application whilst leaving the session and other application's information safe:

```
store.empty()
```

If you do wish to destroy a session and understand the risks you can use:

```
manager.destroy(ignoreWarning=True)
```

failing to specify *ignoreWarning* as `True` will result in a `SessionWarning` being raised to inform you of the potential dangers.

Cleaning Up Expired Sessions

Every time a session is loaded or created there is a certain probability (specified by the *cleanupProbabilty* parameter of the `web.session.manager()` function) that the session module will look through all sessions to see which ones have expired, removing session information and expired sessions as necessary. This means sessions are not necessarily get destroyed when they expire.

Setting the cleanup parameter too high means unnecessary work is done checking expired session more than is needed. Too low and data may persist for a long time meaning that it takes a long time to cleanup the sessions once the cleanup process is finally begun.

System administrators can manually cleanup sessions using the manager instance's `cleanup()` method. Using the method without parameters removes all expired sessions. The method also accepts *min* and *max* to specify the range of expiry times to cleanup. You can also cleanup sessions which have not yet expired but this is dangerous for the same reasons destroying current sessions is and will raise a `SessionWarning`. To ignore the warning set the parameter *ignoreWarning* to `True`.

Changing the Expire Time of a Session

You can change the expire time of a session using `manager.setExpire()`. The method takes *expireTime* which is the time you want the session to expire in seconds since the epoch (00:00:00 UTC, January 1, 1970) This is the format returned by `time.time()`. *expireTime* is **not** the extra number of seconds to allow the session to exist for.

1.11.8 Custom Cookie Handling

To understand how cookies work you may want to first read `The HTTP Protocol and Cookie Handling` sub section of the `Background Information` section of this documentation.

If you don't want to have headers sent automatically when using the `create()` and `destroy()` methods you can set the *sendCookieHeaders* parameter to `False`. In this case the header is instead appended to the `response_headers` attribute in the form of a tuple `(type, info)` where `type` is the header type eg `Set-Cookie` and info is the header information.

To send the headers you can use `sendCookieHeaders()` to send all the headers. Once the headers are sent they are appended to the `response_headers` attribute for debugging purposes.

Alternatively you can retrieve the last header and turn it back into a usual HTTP header using this code:

```
cookieHeader = "%s: %s"%manager.response_headers[-1]
```

If you want to build your own cookie headers you can use `_setCookieString()` and `_deleteCookieString()` which return HTTP headers as strings suitable for printing directly.

Finally, cookies are read from the `HTTP_COOKIE` environmental variable. If you wish to provide your own environment dictionary instead of the default (if for example you are using a WSGI application) you can read a cookie like this:

```
sessionID = manager.cookieSessionID(environ=environ)
```

See the API documentation for more information.

1.11.9 Web Server Gateway Interface Middleware

A much more modular way of using the `web.session` module functions and classes is to use them as Web Server Gateway Interface Middleware. This is described in the `web.wsgi.session` module documentation which also includes an example.

1.11.10 Implementing a new Driver

To implement a new driver you need to create a new module in 'web/session/drivers/' with the name of the driver as the file name and '.py' as the extension.

The file should define two classes named in a similar way to the database driver classes, one of which implements the checking, creation and removal if the driver environment and the other implements the `web.session` module API and inherits from the first class.

The 'web/session/drivers/database.py' can be used as an example. If you implement all the methods in the same manner as the database driver and each method returns variables of the same type in the same order and raises the same extensions you will have a valid driver.

Please forward any such drivers to the developers who may wish to include your driver if it is of a sufficiently high standard and does not require any API changes to any of the other web modules.

1.11.11 Example

Here is a full example showing the creation of all the necessary objects and giving you full control over the session:

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

# show python where the modules are
import sys; sys.path.append('../'); sys.path.append('../../../')
import web.error; web.error.enable()
import os, time
import web.database

# Setup a database connection
connection = web.database.connect(
    adapter="snakesql",
    database="webserver-session",
    autoCreate = 1,
)
cursor = connection.cursor()


# Obtain a session manager the full way
import web.session
manager = web.session.manager(driver='database', cursor=cursor, autoCreate=1)
sessionID = manager.cookieSessionID()
if not manager.load(sessionID):
    manager.create(sendCookieHeaders=False)
    manager.sendCookieHeaders()
store = manager.store('testApp')

def printPage(title, url, link, url2, link2, data):
    print """
    <html>
    <h1>%s</h1>
    <p><a href="%s">%s</a></p>
    <p><a href="%s">%s</a></p>
    <p>%s</p>
    </html>"""%(title, url, link, url2, link2, data)

# Write a simple application the full way
if not manager.created:
    if web.cgi.has_key('destroy') and web.cgi['destroy'].value == 'True':
        manager.destroy(ignoreWarning=True, sendCookieHeaders=False)
        manager.sendCookieHeaders()
        print web.header('text/html')
        printPage(
            'Session Destroyed',
            os.environ['SCRIPT_NAME'],
            'Start Again', '','',''
        )
    else:
        manager.setExpire(manager.expireTime+5, sendCookieHeaders=1)
        print web.header('text/html')
        data = []
        data.append('SessionID:  ' +manager.sessionID)
        data.append('Store Keys: '+str(store.keys()))
        data.append('Store App:  '+store.app)
        data.append('Variable1:  '+str(store['Variable1']))
        data.append('ExpireTime: '+str(manager.expireTime))
        printPage(
            'Welcome back',
            os.environ['SCRIPT_NAME'],
            'Visit Again',
```

```
                os.environ['SCRIPT_NAME']+'?destroy=True',
                'Destroy Session',
                '<p>Every time you visit this page the expiry \
                time increases 5 seconds</p>'+'</p><p>'.join(data)
            )
    else:
        print web.header('text/html')
        store['Variable1'] = 'Python Rules!'
        printPage(
            'New Session Started',
            os.environ['SCRIPT_NAME'],
            'Visit Again', '', '',
            "Set variable1 to 'Python Rules!'"
        )

    connection.commit() # Save changes
    connection.close()  # Close the database connection
```

You can test this example by running the webserver 'scripts/webserver.py' and visiting http://localhost:8080/doc/src/lib/webserver-web-session.py

1.11.12 API Reference

Manager Objects

The `SessionManager` object is aliased as `web.session.manager` and should be used as `web.session.manager`.

class `SessionManager`(*driver*, [*expire=86400*], [*cookie*], [*autoCreate=0*], [*_seed*], [*_cleanupProbability*], [***driverParams*])

Used to return a session manager object.

Manager Parameters:

driver* and ***driverParamsIf *driver* is a string, any extra parameters passed to the `web.session.manager()` function are passed onto the `web.session.driver()` function to create a driver. Alternatively `driver` can be a `Driver` object as returned by `web.session.driver()` and no ***driverParams* need to be specified.

expireThe number of seconds a newly created session should be valid for. If not specified the default is 86400 seconds which is 24 hours.

cookieA dictionary specifying any parameters which you would like the session cookie to have. The defaults used are specified in `web.session.cookieDefaults`.

In particular you may wish to modify the following parameters:

`path`The path of the domain specified by *domain* for which the cookie is valid. If not specified the default is `'/'` which means the whole website. XXX is this correct?

`domain`The domain for which the cookie is valid. If not specified the default is `''` which means any domain. XXX is this correct?

`comment`An optional comment for your cookie to explain what it does or who set it

`max-age`The length of time in seconds the cookie should be valid for. If set to `0` the cookie will expire immediately. If not present the cookie will take the expire time of the session. If set to `None` the cookie will last until the web browser is closed.

By default the `max-age` of the cookie is set to be the same as the expire time set by the *expire* parameter.

autoCreateIf set to `True` the necessary tables will be created (removing any existing tables) if any of the tables are missing. This is designed for easy testing of the module.

_seed When generating session IDs it is important a hacker cannot guess what the next session ID will be otherwise they could make a cookie so that the application thinks they are someone else. You can specify a *_seed* which is simply a string to make the generation of session IDs even more random. The default is `'PythonWeb'`.

_cleanupProbability Every so often expired sessions and their corresponding data need to be removed from the session store. There is a probability specified by *_cleanupProbability* that this cleanup will occur when a manager object is created. If *_cleanupProbability* is `1` cleanup is done every time a manager is created. If *_cleanupProbability* is `0` no automatic cleanup is done and cleanup is left to the administrator. The default is `0.05` which means old session information is removed roughly every 20 times a manager object is created.

All session manager objects have the **read only** member variables which you should not set:

`sessionID`
The session ID for the current session. This is a unique 32 character string set after a session is created or loaded. It is `None` before that time.

`expire`
The expire length in seconds (the minimum length of the session)

`_seed`
The default seed used to generate the session ID

`cookie`
A dictionary containing the cookie parameters.

`error`
If an error occurred loading a session, for example the session ID did not exist or had expired, the error is available through this attribute. If no error occurred the value is `None`.

`_cleanupProbability`
The probability of checking for, and removing expired sessions

`response_headers`
A list of cookie headers in the WSGI format `(type, value)`

`sent_headers`
A list of the cookie headers printed after `sendCookieHeaders()` has been called. Useful for debugging

`completeSessionEnvironment``()`
Returns `True` if the environment is correctly setup, `False` otherwise. In the case of the database driver this method simply checks that all the necessary tables exist.

`createSessionEnvironment``()`
Creates the necessary environment. In the case of the database driver this method creates all the required tables. If any of the tables already exist a `SessionError` is raised.

`removeSessionEnvironment``([`*ignoreErrors=False*`])`
Removes the environment. In the case of the database driver this method drops all the tables. If any of the tables are not present a `SessionError` is raised unless *ignoreErrors* is `True`.

`load``([`*sessionID=None*`])`
Attempt to load the session with the session ID *sessionID*. If *sessionID* is not specified the session ID is obtained from a cookie using `os.environ`. If your environment doesn't support loading of a cookie in this way *seesionID* should be specified. If the session exists and is valid it is loaded and the method returns `True` otherwise it returns `False` and you should create a new session using `create()`. The reason the session could not be loaded is set to the `error` attribute.

`create``([`*sendCookieHeaders=True*`], [`*expire*`])`
Generate a new session ID and start a new session with it. If after 100 attempts no new session ID has been created because the IDs generated already exist, a `SessionError` is raised. `self._seed` is specified it is used to make the generation of session ID more random.

If *sendCookieHeaders* is `True` a `Set-Cookie` HTTP header is immediately printed. If `False` a WSGI `(type, info)` header is appended to `response_headers` so the application can handle the header itself. If *expire* is the number of seconds the session should be valid for. If not specified the value of the `expire` attribute is used. Returns the new session ID.

`store`(*app*)

Return a session store object for manipulating values in the application's store. *app* is the application name as a string made up of the characters `a-z`, `A-Z`, `0-9` and `-_.`. The application name must be between 1 and 255 characters in length. The application names do not have to be the same as application names used by the `web.auth` module, although these are the most appropriate choices. If you are not using multiple applications you should still give your application a name, perhaps `'default'` for example.

`destroy`([*sessionID*], [*sendCookieHeaders=True*], [*ignoreWarning=False*])

Remove all session information for the session. If *sessionID* is specified all session information for *sessionID* is removed. If *sendCookieHeaders* is `True` a `Set-Cookie` HTTP header is immediately printed. If `False` a WSGI `(type, info)` header is appended to `response_headers` so the application can handle the header itself. If *ignoreWarning* is not set to `True` a `SessionWarning` is raised explaining why destroying sessions is not a good idea.

Warning: Destroying sessions is strongly not recommended since any other application currently using the session store may crash as the session information will have been removed. If you wish to remove all data from the session store it would be better to use the store object's `empty()` method, emptying the store but leaving the session intact. If you must remove a session use `setExpire(time.time())` to make the session expire immediately or send a cookie built with `_deleteCookieString()`. Any applications using the session will still be able to access the information if they have already loaded the session but will not be able to load the session again.

`cookieSessionID`([*environ=None*], [*noSessionID=''*])

Obtain a session ID from the `HTTP_COOKIE` environmental variable. The default `environ` dictionary is `os.environ`. If you wish to provide your own environment dictionary (for example you are using a WSGI application) you can specify *environ*. If the session ID cannot be loaded *noSessionID* is returned which by default is an empty string.

`cleanup`([*min*], [*max*], [*ignoreWarning=False*])

Remove and information related to sessions which have expired between the times *min* and *max*. All times are in seconds since the epoch (00:00:00 UTC, January 1, 1970). If *min* is not specified it is assumed to be `0` (the beginning of the epoch), if *max* is not specified it is assumed to be the current time. If you specify a value *max* greater than the current time returned bt `time.time()` a `SessionWarning` is raised. To ignore the warning set *ignoreWarning* to `True`.

Warning: You should not set a value of *max* greater than the current time unless you understand the risk since doing so will remove sessions which haven't yet expired. If an application is using the session store and its session is cleaned up, that application may crash.

`setExpire`(*expireTime*, [*sessionID*])

The method is used to change the time an existing session will expire or to set a new expiry date if it has already expired. *expireTime* is the time you want the session to expire in seconds since the epoch (00:00:00 UTC, January 1, 1970). *expireTime* is NOT the extra number of seconds to allow the session to exist for. If *sessionID* is specified the expire time of the session with ID *sessionID* is updated, otherwise the current session expire time is modified.

`valid`([*sessionID*])

If *sessionID* is specified the validity of the session with ID *sessionID* is checked. Otherwise the current session is checked. Returns `True` if the session is valid, `False` if the session has expired. A `SessionError` is raised if the session does not exist. Whether or not a session exists can be checked with the `exists()` method.

`exists`([*sessionID*])

If *sessionID* is specified the session with ID *sessionID* is checked. Otherwise the current session is checked. Returns `True` if the session is exists, `False` if the session does not exist. No comment is made on whether or not the session is still valid, instead this can be checked with the `valid()` method.

sendCookieHeaders()
Uses Python's `print` statement to send any headers in the `response_headers` attribute to the standard output, appending the exact strings printed to the `send_headers` attribute for debugging purposes. Used by the `create()` and `destroy()` methods to send cookie headers so could be over-ridden in derived classes to change cookie handling behaviour.

setCookie([*sendCookieHeaders=False*], [*_cookieString=None*])
Get a cookie string from `_setCookieString()` to set a new cookie, parse the string into a WSGI `(type, info)` pair and append it to the `response_headers` attribute. If *sendCookieHeaders* is `True`, `sendCookieHeaders()` is called to send the cookie header. *_cookieString* can be used to specify the cookie to set, if `None` the cookie string is automatically generated.

_setCookieString([*maxAge*])
Returns an HTTP header to set a cookie using the default cookie values set in the class constructor. *maxage* is the length of time in seconds the cookie should last.

deleteCookie([*sendCookieHeaders=False*])
Get a cookie string from `_deleteCookieString()` to set the expire time of the cookie to one second, parse it into a WSGI `(type, info)` pair and append it to the `response_headers` attribute. If *sendCookieHeaders* is `True`, `sendCookieHeaders()` is called to send the cookie header.

_deleteCookieString()
Returns an HTTP header to set the expire time of the session cookie to 1 second, effectively forcing it to expire.

Store Objects

The store object is obtained from the `store()` method of the manager object. It is used to manipulate the session store of the application specified in the `store()` method.

class Store
store objects have the following attribute:

app
This is the name of the application whose store we are manipulating. `app` can be set to another application's name in order to manipulate a different session store. Application names are strings made up of the characters `a-z`, `A-Z`, `0-9` and `-_.` and are between 1 and 255 characters in length.

store objects have the following methods:

set(*key, value*)
Set the value of *key* to the value *value* in the session store. *value* can be any Python object capable of being pickled. See Python's `pickle` module for more information.

get(*key*)
Get the value of *key* from the session store.

delete(*key*)
Remove *key* and its associated value from the session store.

empty()
Empty this application's session store of all information removing all keys and values but leaving the session itself and other application's stores intact.

has_key(*key*)
Returns `True` if *key* exists on the session store otherwise `False`.

keys()
Returns a sequence of store keys. The order of the keys is not defined. Keys obtained from this method cannot be set directly. Instead the `set()` method should be used.

items()
Returns a tuple of `(key, value)` pairs for each key in the store. The order of the values is not defined.

Values and keys obtained from this method cannot be set directly. Instead the `set()` method should be used.

`values``()`
Returns a sequence of store values. The order of the values is not defined. Values obtained from this method cannot be set directly. Instead the `set()` method should be used.

store objects also implement the following methods: `__getitem__(key)`, `__setitem__(key, value)` and `__delitem__(key)` which map directly to the `get(key)`, `set(key)` and `delete(key)` methods respectively and allow the store object to be treated similarly to a dictionary as demonstrated earlier in the documentation.

1.12 `web.template` — For the easy display of data as HTML/XML

The `web.template` module currently only provides one function, `pasrse()`, used to parse a template.

`parse`(*type='python', dict=None* [*,file=None*][*,template=None*][*,useCompiled='auto'*][*,swapTemplatePaths=None*])
Simple wrapper method to load and parse a template from the options given.

type The type of template to parse. Can be `'python'`, `'cheetah'`, `'xyaptu'` or `'dreamweaverMX'`. A `'python'` template is a string using the dictionary filling format.

dict A dictionary of values used to fill the template

file The file containing the template. If not specified or `None`, *template* must be specified.

template The template as a string. If not specified or `None`, *file* must be specified.

useCompiled Only used for Cheetah. Specifies whether or not a compiled version of the template should be used.

swapTemplatePaths Only used for DreamweaverMX. If `None` nothing is done. Otherwise can be set to `(oldPath, newPath)` to swap paths in the template itself before the parsing is done.

Simple example:

```
>>> import web.template
>>> print web.template.parse(dict={'w':'World!'}, template="Hello %(w)s")
Hello World!
```

This is the same as doing this in Python:

```
>>> print "Hello %(w)s"%{'w':'World!'}
Hello World!
```

1.12.1 Cheetah Template

Cheetah is a powerful, stable and well documented templating system. It works by parsing the template into a Python script and then executing that script with the dictionary to produce output. The performance of Cheetah can be improved by writing this script to a file and executing it each time Cheetah is run rather than re-generating it every time.

The *useCompiled* parameter of the `parse()` function can be used to determine the behaviour of this compilation. If *useCompiled* is `False` the template is parsed every time. This is the slowest but simplest option. If *useCompiled* is `True` the compiled template is used even if the original template has changed. This is the fastest option but you must manually tell Cheetah to recompile the template if it changes. If *useCompiled* is `'auto'` then Cheetah will use the compiled file as long as the template has not been modified. If it has it will automatically recompile the template.

Warning: This is the best comprimise. If *useCompiled* is `True` or `'auto'` then Cheetah must have write access to the directory containing the templates. If it doesn't you may get Internal Server Errors, particularly if you are using `web.error` with Cheetah templates to catch errors as an error will be thrown in the error catching code and this will lead to an error that is hard to track down.

You can also use Cheetah directly by importing it as follows:

```
import web
import Cheetah
```

Here is an example Cheetah template:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>$title</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
</head>
<body>
<h1>$title</h1>
$welcomeMessage
#if $testVar == True
The test variable is True
#else
The test variable is not True
#end if
</body>
</html>
```

Here is a program to manipulate it:

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

import sys; sys.path.append('../../../') # show python where the web modules are
import web.template

dict =  {
    'welcomeMessage':'Welcome to the test page!',
    'testVar':True,
    'title':'Cheetah Example',
}

print web.template.parse(
    type='cheetah',
    file='file-web-template-cheetah.tmpl',
    dict=dict
)
```

And here is the output produced:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Cheetah Example</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
</head>
<body>
<h1>Cheetah Example</h1>
Welcome to the test page!
The test variable is True
</body>
</html>
```

See Also:

Cheetah Template Homepage
(http://cheetahtemplate.org/)

The Cheetah homepage has full documentation for using Cheeath and explains the full syntax available and the range of options that can be used.

1.12.2 XYAPTU Templating

XYAPTU is an ASPN recipie based on YAPTU. Both modules are included with the web modules and can be imported directly:

```
import web
import xyaptu, yaptu
```

Here is an example xyaptu template:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>$title</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
</head>

<body>
<p>$welcomeMessage</p>
    <py-open code="if testVar:" />
       The variable is: True
    <py-clause code="else:" />
       The variable is: False
    <py-close/>

</body>
</html>
```

Here is a program to manipulate it:

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

```
import sys; sys.path.append('../../../') # show python where the web modules are
import web.template

dict =  {
    'welcomeMessage':'Welcome to the test page!',
    'testVar':True,
    'title':'XYAPTU Example',
}

print web.template.parse(
    type='xyaptu',
    file='file-web-template-xyaptu.tmpl',
    dict=dict
)
```

And here is the output produced:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>XYAPTU Example</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
</head>

<body>
<p>Welcome to the test page!</p>


       The variable is: True



</body>
</html>
```

See Also:

XYAPTU Information on ASPN
(http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/162292)
This page is where the recipie first appeared and is where the most complete documentation can be found.

1.12.3 Dreamweaver MX

The web modules can also parse Dreamweaver MX templates as long as they only use standard Editable Regions and the regions are empty so that the tags look like this:

```
<!-- TemplateBeginEditable name="content" --><!-- TemplateEndEditable -->
```

DreamweaverMX templates are passed just like the others except you set *type* to `'dreamweaverMX'`.

Warning: If you set the `doctitle` editable region please remembe to include `<title>` and `</title>` tags around the title you set as the template doesn't include these for you.

Here is an example Dreamweaver MX template:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<!-- TemplateBeginEditable name="doctitle" -->
<title>PythonWeb.org - Dreamweaver MX Example</title>
<!-- TemplateEndEditable -->
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
<!-- TemplateBeginEditable name="head" --><!-- TemplateEndEditable -->
</head>

<body>

<h1><!-- TemplateBeginEditable name="Title" -->Web Modules<!-- TemplateEndEditable --></h1>
            <!-- TemplateBeginEditable name="Content" -->
            <p> </p>
            <!-- TemplateEndEditable -->

</body>
</html>
```

Here is a program to manipulate it:

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

import sys; sys.path.append('../../../') # show python where the web modules are
import web.template

dict =  {
    'Content':'Welcome to the test page!',
    'doctitle':'Dreamweaver MX Example',
    'Title':'Dreamweaver MX Example',
}

print web.template.parse(
    type='dreamweaverMX',
    file='file-web-template-dreamweaverMX.dwt',
    dict=dict
)
```

And here is the output produced:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<!-- InstanceBeginEditable name="doctitle" -->Dreamweaver MX Example<!-- InstanceEndEditable --
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
<!-- InstanceBeginEditable name="head" --><!-- InstanceEndEditable -->
</head>

<body>

<h1><!-- InstanceBeginEditable name="Title" -->Dreamweaver MX Example<!-- InstanceEndEditable -
            <!-- InstanceBeginEditable name="Content" -->Welcome to the test page!<!-- Instance

</body>
</html>
```

1.13 `web.util` — Useful utility functions that don't fit elsewhere

This module provides a number of functions which come in handy when programming web applications but that don't fit in elsewhere. It is a catch all module for useful leftovers.

`wrap`(*text, width*)

A word-wrap function that preserves existing line breaks and most spaces in the text. Expects that existing line breaks are posix newlines. (ie backslash n)

textThe text to wrap

widthThe maximum number of characters in a line

`strip`(*html,* [*validTags=[]*])

Strip illegal HTML tags from string

htmlThe HTML which needs some tags stripping

validTagsA list or tuple of tags to leave in place

`runWebServer`([*root='../', cgi='/cgi-bin',*])

Run a simple webserver on port 8080 on localhost. The root of the website corresponds to the directory *root*. *cgi* is URL of the the cgi-bin where files can be executed. Once this command is run code execution stops as the webserver listens for requests so there is no point in writing code after this command as it will not be run.

Warning: NOT SUITABLE FOR COMMERCIAL USE.

dirThe only directory where scripts are allowed to run. Directory names should be the full URL path from the root of the webserver and therefore should begin with `/`

1.13.1 Creating Tables

`table`(*columns, values,* [*width=80*], [*mode*])

Pretty print a table of data for display in a terminal of width *width*

Warning: This function has changed radically from version 0.4.0

columnsThe names of the columns in the order they are displayed in each row of values.

values The data to be displayed in the format:

```
(
    ('column1value1', 'column2value1', 'column3value1', 'column4value1'),
    ('column1value2', 'column2value2', 'column3value2', 'column4value2'),
    ('column1value3', 'column2value3', 'column3value3', 'column4value3'),
)
```

The values and column headings can be any object which can be converted to a string using `str()`.

width The wrap width of the string produced. The default is `80` which means the table will we wrapped to the width of a standard terminla or command line prompt. If *width* is set to `0` no wrapping is produced.

display If *display* is set to `'terminal'` the line ending at the wrap width (specified by *width*) will not be added since the line will wrap around to the next line anyway. Adding the linebreak would result in blank lines appearing.

mode If *mode* is set to `'sql'` the values are encoded in a way to represent `None` as `NULL` and use `repr()` when `str()` would be ambiguous.

For example:

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

import sys; sys.path.append('../../../') # show python where the web modules are

import web.util
columns = [
    'column1Heading',
    'column2Heading',
    'column3Heading',
    'column4Heading'
]
values = [
    ['column1value1', 'column2value1', 'column3value1', 'column4value1'],
    ['column1value2', 'column2value2', 'column3value2', 'column4value2'],
    ['column1value3', 'column2value3', 'column3value3', 'column4value3'],
]
print "Printing the table with wrap width=0...\n"
print web.util.table(columns, values, width=0)
print "Printing the table with wrap width=60...\n"
print web.util.table(columns, values, width=60)
```

```
The output produced is:

Printing the table with wrap width=0...

+----------------+----------------+----------------+----------------+
| column4Heading | column3Heading | column2Heading | column1Heading |
+----------------+----------------+----------------+----------------+
| column4value1  | column3value1  | column2value1  | column1value1  |
| column4value2  | column3value2  | column2value2  | column1value2  |
| column4value3  | column3value3  | column2value3  | column1value3  |
+----------------+----------------+----------------+----------------+

Printing the table with wrap width=60...

+----------------+----------------+----------------+--------
| column4Heading | column3Heading | column2Heading | column1
+----------------+----------------+----------------+--------
| column4value1  | column3value1  | column2value1  | column1
| column4value2  | column3value2  | column2value2  | column1
| column4value3  | column3value3  | column2value3  | column1
+----------------+----------------+----------------+--------

--------+
Heading |
--------+
value1  |
value2  |
value3  |
--------+
```

Warning: If you don't set the wrap width and your table is wider than the terminal then the terminal will wrap the table output itself. If this happens it will wrap each induvidual line of text rather than the whole table producing output that looks more like this:

```
+----------------+----------------+--------------
--+----------------+
| column4Heading | column3Heading | column2Headin
g | column1Heading |
+----------------+----------------+--------------
--+----------------+
| column4value1  | column3value1  | column2value1
| column1value1  |
| column4value2  | column3value2  | column2value2
| column1value2  |
| column4value3  | column3value3  | column2value3
| column1value3  |
+----------------+----------------+--------------
--+----------------+
```

1.13.2 Calendar Tools

The `web.util.calendarTools` module provides two functions `month()` to display an HTML calendar for the specified month and `calendar()` to display a range of months in HTML.

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

# show python where the modules are
import sys; sys.path.append('../'); sys.path.append('../../../')
import web.error; web.error.enable()
import web.util.calendarTools

print web.header()

calendars = [
'web.util.calendarTools.month(year=2004,month=12)',
"""web.util.calendarTools.month(
    year=2004,
    month=12,
    dayNameLength=0,
    firstDay=5
)""",
"""web.util.calendarTools.month(
    year=2004,
    month=12,
    dayNameLength=3,
    monthURL = 'month.py?month=%(month)s&amp;year=%(year)s',
    previousURL = 'previous.py?month=%(month)s&amp;year=%(year)s',
    nextURL = 'next.py?month=%(month)s&amp;year=%(year)s',
    previousHTML = 'prev',
    nextHTML = 'next',
)""",
"""web.util.calendarTools.month(
    year=2004,
    month=12,
    daysURL = 'day.py?day=%(day)s&amp;month=%(month)s&amp;year=%(year)s',
    days = {
        12:['day.py?day=12', 'twelve', 'style="background: #eee"']
    }
)""",
"""web.util.calendarTools.month(
    year=2004,
    month=12,
    tableColor = '#eeeecc',
    barColor = '#eeeeee',
    cellPadding = 16,
)""",

]

fullCal="""web.util.calendarTools.calendar(
    startYear=2004,
    startMonth=9,
    months=12,
    cols=3,
    dayNameLength=1,
    barColor="#eeeeee"
)
"""
output = ''
for cal in calendars:
    output += """<hr><table border="0" cellPadding=10 width="100%%">
    <tr><td width="1%%">%s</td><td width="99%%"><pre>%s</pre></td>
    </tr></table>"""%(eval(cal), web.encode(cal, mode='form'))
```

```
print """
<html>
<style>
.calendarTools-month-header{
    font-family: sans-serif;
    font-weight: bold;
}
</style>
<body>
<h1>Example Calendars</h1>
<p>This page demonstrates the parameters used to generate HTML
calendars. They can also be styled using CSS stylesheets. For
example the following is uses in this HTML page to make all the
calendar headings a sans serif font:</p>
<pre>
&lt;style&gt;
.calendarTools-month-header{
    font-family: sans-serif;
    font-weight: bold;
}
&lt;/style&gt;
</pre>

%s

<hr>
<p>The next calendar is generated using the following code:
<pre>
%s
</pre>

%s
</body>
</html>"""%(output, fullCal, eval(fullCal))
```

You can test this example by starting the test webserver in 'scripts/webserver.py' and visiting http://localhost:8080/doc/src/lib/webserver-web-util-calendar.py on your local machine.

XXX Full function reference

1.14 `web.wsgi` — Web Server Gateway Interface tools

The WSGI interface is a specification designed by Phillip J. Eby with contributions from the Python Web-SIG mailing list which defines a proposed standard interface between web servers and Python web applications or frameworks, to promote web application portability across a variety of web servers.

The `web.wsgi` module implements the WSGI interface for the Web Modules.

Note: The web server interface and tools proposed for previous versions of the modules have now been dropped in favour of supporting the WSGI in their place. All components which were implemented have now been moved into WSGI middleware components instead.

See Also:

PEP 333 - Python Web Server Gateway Interface
(http://www.python.org/peps/pep-0333.html)

This document specifies the Web Server Gateway Interface and defines some simple objects demonstrating the approach. **PEP 333 should be read before reading this documentation**

It should also be noted that the web modules WSGI implementation is based heavily on Phillip J. Eby's `wsgiref` implementation

Note: The WSGI specification is farily new and the author of this document is learning it as he goes along! Consequently there may be important omissions or even errors. I would very much appreciate any comments or corrections so please feel free to contact docs at pythonweb.org if you have any.

1.14.1 Introduction

What is a WSGI application?

The WSGI PEP can be quite confusing if all you want to do is write applications quickly and easily. The best way to explain the WSGI is to work through an example demonstrating how an application written as a CGI script has to be modified to work as a WSGI application.

Consider the CGI script code below:

```
#!/usr/bin/env python
print 'Content-type: text/plain\n\n'
print 'Hello world!'
```

This does nothing more than print the words `'Hello world!'` to a web browser in plain text. What we have done is sent an HTTP header `Content-type: text/plain\n\n` and then a text string to the browser. The webserver may also have sent a `'200 OK'` response if the application completed successfully.

To create the same result using a WSGI application we would use this code:

```
def simplestApp(environ, start_response):
    start_response('200 OK',[('Content-type','text/plain')])
    return ['Hello world!']

application = simplestApp
```

This is the most basic WSGI application. It is a function names `application` which a WSGI server will call and pass two parameters. The first is a dictionary named *environ* containing environmental variables and the second is a function named *start_response* which must be called before the application returns a value.

You may not be happy with the function `start_response` being passed as a parameter to our `application` callable. Whilst it is not possible to pass a function as a parameter in some languages it is allowed in Python. This ability to pass callables as function parameters is crucial to understanding how the WSGI works.

Here is an example to consider:

```
def b(text):
    print text

def a(print_response):
    print_response("Hello World!")
    return "It worked!"

print a(b)
```

In this case we are passing the function `b` to the `a` as the parameter *print_response*. We are then printing the value returned from `a`. What do you think the result will be?

The answer is this:

```
Hello World!
It worked!
```

Make sure you understand this example before you read on.

A WSGI application must do two things, these are:

1. Call the `start_response` function (passed to our `application` callable) with the parameters *status* and *headers* in the correct order. This will set the status of the application and send the HTTP headers. In our example the status is `'200 OK'` meaning everything has gone according to plan and we only send one header, the `Content-type` header with the value `text/plain`.
2. Return an iterable containing nothing but strings. In this example the iterable is simply a list containing one string. The return value could equally well have been `['Hello', ' ', 'world!']` but there was no need to make things more complicated.

There are some big advantages in rewriting our code as a WSGI application:

- Once a server has loaded our application it can execute it many times without having to reload it on each request. This makes for huge performance gains over a traditional CGI approach.
- By using callables in this standard way it is possible to chain together applications called middleware components to provide applications with extra functionality passed in the *environ* dictionary with very little programming effort.
- The application has control over its status. For example if the application encountered an error it could send an `'500 Error'` status message and the WSGI server would display its appropriate error page.
- All HTTP headers are sent at the same time before the main content avoiding the possibility of sending HTTP headers at the wrong time.

What Are Middleware Components?

Consider the slightly more complicated example below using the imaginary session handling module `superSession`:

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

import superSession
session = superSession.session()
print "Content-type: text/plain\n\n"
if session.has_key('visited'):
    print "You have already visited!"
else:
    session['visited'] = 1
    print "This is your first visit."
```

We create a session object and display a different string depending on whether or not the user has visited the site before. We could follow the approach above and create the following WSGI application to do the same thing:

```
def application(environ, start_response):
    import superSession
    session = superSession.session()
    if session.has_key('visited'):
        text = "You have already visited!"
    else:
        session['visited'] = 1
        text = "This is your first visit."
    start_response('200 OK', [('Content-type','text/plain')])
    return [text]
```

This would be perfectly good and work perfectly well. We could now refactor the code again:

```
def exampleApplication(environ, start_response):
    if environ['superSession'].has_key('visited'):
        text = "You have already visited!"
    else:
        environ['superSession']['visited'] = 1
        text = "This is your first visit."
    start_response('200 OK', [('Content-type','text/plain')])
    return [text]

def session(application):
    def app(environ, start_response):
        if "superSession" not in environ:
            import superSession
            environ["superSession"] = superSession.session() # Options would obviously need spe
        return application(environ, start_response)
    return app

application = session(exampleApplication)
```

We have separated out the session code into a different function and added a key to the `environ` dictionary called `"session"` which contains the session object. Our `exampleApplication` then accesses the session object through the `environ` dictionary. Note how we have renamed our `application` function to `exampleApplication` and mapped the name `application` to the `session(exampleApplication)` object. The WSGI server will still be able to find a callable named `application` and so will still be able to run our application.

The `session` function is now what we call a middleware component as it sits in between the server and the application. It gives the application new functionality but the result of calling `session(exampleApplication)` is also just a WSGI application (because the combined object still conforms to the rules listed earlier) and so the server can still run the code.

The huge advantage of refactoring code in this way is that the session functionality can now easily be added to any WSGI application using our `session` function. By chaining together these middleware components (which do not even have to be based on the Web Modules) WSGI applications can gain an enormous amount of functionality for very little programming effort by using existing middleware components. This helps make code easy to maintain and offers a very flexible programming methodology.

Callables, Classes or Functions?

I have been quite careful all the way through the introduction to describe the application and middleware as callables and not just as functions (which is what they have happened to be so far). We could re-write the session middleware component described in the previous section as follows:

```
class Session:
    def __init__(self, application):
        self.application = application

    def __call__(self, environ, start_response):
        if "superSession" not in environ:
            import superSession
            environ["superSession"] = superSession.session() # Options would obviously need spe
        return self.application(environ,start_response)

application = Session(exampleApplication)
```

If you think carefully about what is happening here you will realise that our `Session` class behaves in exactly the same way as the function `session` did in the previous example.

The advantage of using a class rather than a function for a middleware component is that you can derive another middleware component from an existing one that provides similar functionality without re-writing the entire component.

The `web.wsgi` module contains middleware classes for all of the web modules functionality which you can use on their own or as base classes for your own middleware components including session functionality. The middleware components are all described later on in this documentation.

You can also specify your application object as a class. Consider this example

```
def myApp(environ, start_response):
    start_response('200 OK', [('Content-type','text/plain')])
    return ['Hello World!']

application = myApp
```

The following code performs exactly the same task.

```
class MyApp(web.wsgi.base.BaseApplication):
    def __call__(self, environ, start_response):
        start_response('200 OK', [('Content-type','text/plain')])
        return ['Hello World!']

application = MyApp()
```

It can also be useful to specify applications as classes so that functionality can be derived from other applications.

Difficulties with WSGI Applications

One of the most important differences between using WSGI applications and ordinary CGI applications is that WSGI applications code is not loaded on each request. Instead it is loaded once and then repeatedly executed. This means that you cannot put information which is likely to change on each request in the global namespace because it will not be updated.

A good example to illustrate this problem is `web.cgi`. The `web.cgi` variable is loaded once and contains any CGI variables to be passed to a CGI script. Since each CGI request completely reloads and executes all code in the script, `web.cgi` will always contain the correct CGI variables when used in a CGI script.

However, consider this example application:

```
import web

def application(environ, start_response):
    start_response('200 OK', [('Content-type','text/plain')])
    return ['Here are the CGI variables: %s'%('\n'.join(web.cgi.keys()))]
```

The first time the application is run, the correct results will be displayed, the second time it is run by the WSGI server, `web` will already have been imported and will not be imported again. This means the `web.cgi` variable will be out of date.

The solution to this is to put everything which needs to be reloaded on each request into the main body of the application, or consider placing it as a middleware component.

```
import cgi

def application(environ, start_response):
    start_response('200 OK', [('Content-type','text/plain')])
    return ['Here are the CGI variables: %s'%('\n'.join(cgi.FieldStorage().keys()))]
```

XXX Probably need to explain this better.

The PythonWeb WSGI Server

A WSGI server has to be able to convert a URL to a path on a drive, find the application named `application` within the file specified and call it, passing the application a dictionary of environmental variables and a `start_response` function to set the status of the application and send the HTTP headers.

Note: As we have seen the object named `application` may not be an application at all, it may in fact be a chain of middleware components and an application, but the WSGI server treats it in the same way because, as we have already seen, applications with middleware stacks behave in exactly the same way as an application on its own.

The Python Web Modules come with just such a WSGI server named 'WSGIServer.py' and available in the 'scripts' directory of the Web Modules distribution.

To use the WSGI server simply run the 'WSGIServer.py' file from the command line by executing the following:

```
> python WSGIServer.py
```

A sample WSGI application should be available by http://localhost:8000/simple with a web browser.

'WSGIServer.py' also takes a series of arguments to customise its behaviour. These can be viewed by running `python WSGIServer.py -h` at the command line.

Warning: Since the WSGIServer loads all the application code when it starts, if you make changes to the samples the server will need to be restarted before the changes will take effect.

The WSGI Server also supports a special debug mode. If your application raises a `web.error.Breakpoint` exception, the server will not handle the request but instead will give a debug prompt so that you can debug the

variables at the breakpoint.

```
import web.wsgi

class simpleApp(web.wsgi.base.BaseApplication):
    def start(self):
        import web.error
        value = 5
        raise web.error.Breakpoint('Test Exception')

application = simpleApp()
```

With the WSGI Server running test this example by visiting http://localhost:8000/debug with a web browser. You will see a `Handling Breakpoint` message at the server prompt (nothing will be displayed in the browser though). Press `Enter`.

You can debug the code as follows and then type `exit` to exit the prompt.

```
debug> value
5
debug> exit
```

`web.wsgi` Functions

`runCGI`(*application*)

Wrapper function to enable WSGI applications to run in a CGI environment. *application* is the WSGI application or middleware to run in a CGI environment.

You may not be in a situation where you have access to a WSGI server. The Python web modules also come with a code to allow WSGI applications and middleware to be run in a CGI environment such as Apache.

If you want to run a WSGI as a CGI application you need to turn it back into one. This can be done very simply by using the middleware component `web.wsgi.runCGI` as shown below:

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

# show python where the web modules are
import sys; sys.path.append('../'); sys.path.append('../../../')

def simpleApp(environ, start_response):
    status = '200 OK'
    headers = [('Content-type','text/plain')]
    start_response(status, headers)
    return ['Hello world from simple_application!\n']

import web.wsgi
web.wsgi.runCGI(simpleApp)
```

The application can then be run in a normal CGI webserver.

To test this approach run 'webserver.py' using `python webserver.py` in the 'scripts' directory and visit http://localhost:8080/doc/src/lib/webserver-web-wsgi-simple-cgi.py to see a sample CGI WSGI application running.

Note: It much faster to execute WSGI applications through a dedicated WSGI server than to run them as CGI scripts. When a CGI script is executed all the Python libraries and modules the script uses need to be loaded into memory and

then removed once the script exists. This has to happen for every request so there is an unecessary delay before the WSGI application is even executed. When using a WSGI server the libraries and modules only need to be loaded once and are then available for any subsequent requests so simple web requests can be handled perhaps 10 times faster.

`currentURL`(*environ*)
Return the current URL of the WSGI application from the environ dictionary *environ*

1.14.2 Understanding Middleware

As we learned in the introduction, WSGI middleware components can be chained together since each middleware, application pair is also a valid WSGI application.

In the example given, the `Session` class changes the *environ* dictionary to provide the application with more functionality. It could also have been chained with an `Auth` middleware component to provide auth functionality as shown below:

```
def exampleApplication(environ, start_response):
    if not environ.has_key('imaginaryAuth'):
        raise Exception{'No auth module found')
    if environ['superSession'].has_key('visited'):
        text = "You have already visited!"
    else:
        environ['superSession']['visited'] = 1
        text = "This is your first visit."
    start_response('200 OK', [('Content-type','text/plain')])
    return [text]

class Session:
    def __init__(self, application):
        self.application = application

    def __call__(self, environ, start_response):
        if "superSession" not in environ:
            import superSession
            environ["superSession"] = superSession.session()
        return self.application(environ, start_response)

class Auth:
    def __init__(self, application):
        self.application = application

    def __call__(self, environ, start_response):
        if "imaginaryAuth" not in environ:
            import imaginaryAuth
            environ["imaginaryAuth"] = imaginaryAuth.auth()
        return self.application(environ, start_response)

application = Auth(Session(exampleApplication))
```

Middleware classes usually do one of four things or a combination of them:

- Change the `environ` dictionary
- Change the application's `status`
- Change the HTTP `headers`

- Change the return value of the application

The most common use is to alter the *environ* dictionary in order to provide more functionality but here are some other ways in which they can be used.

Error Handling Error handling middleware might catch an error raised, format it for display as HTML, change any HTTP headers and status set and return the correct settings for an error page.

User Sign In User sign in middleware might wait for a `'403 Forbidden'` status and instead display a sign in page, setting a new status of '200 OK', new headers and of course a different result containg the HTML of the sign in page.

1.14.3 The PythonWeb Middleware Components

The `web.wsgi` module contains middleware components to make use of all the functionality of the Python Web Modules.

All PythonWeb WSGI middleware components are classes which take another WSGI middleware component or an application as the first argument. The subsequent arguments configure how the middleware behaves.

`web.wsgi.cgi` – CGI Variable Access

The `web.wsgi.cgi` module provides one class `CGI` which adds the key `'web.cgi'` to the *environ* dictionary. Middleware or applications further down the chain can access CGI variables usually accessed through the `web.cgi` object by using `environ['web.cgi']`. The class takes no arguments.

class `CGI`(*application*) ***application***

A WSGI application or middleware component

Entries added to *environ*:

`environ['web.cgi']` CGI information in the format of `web.cgi`

For example:

```
import web.wsgi.cgi

def myApp(environ, start_response):
    start_response('200 OK', [('Content-type','text/plain')])
    return ['Here are the CGI variables: %s'%('\n'.join(environ['web.cgi'].keys()))]

application = web.wsgi.cgi.CGI(myApp)
```

`web.wsgi.error` – Error Handling

Error handling middleware is designed to catch any exception which happened lower down the middleware chain and handle the exception in an appropriate way. The WSGI server or `runCGI` application will handle any exception left uncaught, usually by displaying an HTML page with a message such as "Server Error 500" so error handling middleware is not essential but can be useful for debugging or informational purposes.

The `web.wsgi.error` module provides one class `Error` which does not alter the `environ` dictionary but does catch any exception and print an HTML display of the traceback information. It can also be used to send an email containing a debug output of the error.

```
import web.wsgi

class simpleApp(web.wsgi.base.BaseApplication):
    def start(self):
        raise Exception('Test Exception')

application = web.wsgi.error.Error(
    simpleApp(),
    emailTo=['james@example.com'], # Enter your email address
    replyName='WSGI Error Example',
    replyEmail='none@example.com',
    subject='Error Report',
    sendmail = '/usr/bin/sendmail',  # Specify your sendmail path
    smtp = 'smtp.ntlworld.com',      # or specify an SMTP server and change method to 'smtp'
    method = 'smtp',
)
```

The `error` method should return the values `status`, `headers`, `iterable`.

You can test this example by running the WSGI server 'scripts/WSGIServer.py' and visiting http://localhost:8000/error
Note: Please remember to modify the sample with your own email address and settings. You will need to restart the WSGIServer after making a change.

You can also create your own error handling class by deriving a middleware class from `web.wsgi.error.Error`. In this example a text traceback is displayed instead:

```
import web.wsgi, web.error

def simpleApp(environ, start_response):
    raise Exception('Test Exception')

class myError(web.wsgi.error.Error):
    def error(self):
        "Generate an error report"
        return (
            '200 Error Handled',
            [('Content-type','text/plain')],
            [web.error.info(format='text')]
        )

application = myError(
    simpleApp,
)
```

The `error` method should return the values `status`, `headers`, `iterable`.

You can test this example by running the WSGI server 'scripts/WSGIServer.py' and visiting http://localhost:8000/errorCustom

Note: We do not need the `#!/usr/bin/env python` line or modifications to `sys.path` for WSGI applications since the relevant objects are imported from the files, the files are not executed as scripts.

Errors along the lines of the one shown below may be due to incorrectly formed headers with tuples of the wrong length and can be hard to track down.

```
ValueError: unpack list of wrong size
      args = ('unpack list of wrong size',)
```

`web.wsgi.error.documents` – Error Documents

An error document is simply a web page which is used to let the user know that an error occured.

Note: You will often see "Internal Server Error" error documents if you are using Apache and a CGI script does not have the correct permissions for example. This is an error document. In Apache you can change the error page displayed using a .htaccess file. In WSGI you can use the error document middleware to provide custom error documents.

If a WSGI application or calls `start_response()` with a `status` which is not `200` and is not handled by any middleware, the server will need to display an error message or handle the error in appropriate way, usually by displaying an error document in a similar way to the way Apache would.

The error document middleware lets you intercept these status messages before they are sent to the server to display a custom error document.

The `Documents` class lets you specify error documents in three ways, as files, text, or by calling a function. Each of the three methods involves specifying the status code as an integer key to the dictionary and the object as the item.

For example, to specify a file 'error/500.html' to be displayed if a `500` server error occurs you could specify `files={500:'error/500.html'}`. To specify a the error document as a text you could use the following: `text={500:'<html><h1>Internal Server Error</h1></html>'}`.

Here is a full example:

```
import web.wsgi.error.documents

def simpleApp(environ, start_response):
    start_response('500 There was a server error', [('Content-type','text/html')])
    return []

application = web.wsgi.error.documents.Documents(
    simpleApp,
    text = {
        500:"<html><body><h1>A server error occured</h1></body></html>"
    }
)
```

You can test this example by running the WSGI server 'scripts/WSGIServer.py' and visiting http://localhost:8000/documents

You can also provide more advanced error document handling by specifying a function to handle the error. For example:

```
def serverError(environ):
    import wsgi
    return """<html><h1>Internal Server Error</h1>
    <p>The page %s caused an error<\p></html>"""%wsgi.currentURL(environ)
```

You could then specify `functions={500:serverError}`. The *environ* paramter is the full WSGI environ dictionary passed to all WSGI applications, and as such can be used for advanced dynamic error document generation.

Note: You cannot specify different types of document for the same error. For example if you specify a text server error document for error `500`, you cannot also specify a function server error document for error `500`.

class `Documents`(*application*, [*files*={}], [*text*={}], [*functions*={}]) ***application***

A WSGI application or middleware component

***files*=**{}Any errors which should trigger the display of an error document from a file.

***text*=**{}Any errors which should trigger the display of an error document from a specified Python string.

***functions*=**{}Any errors which should trigger the display of an error document from a function.

`web.wsgi.database` – Database Access

The `web.wsgi.database` module provides one class `Database` which adds the keys `'web.database.connection'` and `'web.database.cursor'` to the `environ` dictionary based on the parameters specified in the class constructor.

class `Database`(*application*, [***params*]) ***application***

A WSGI application or middleware component

****params***Any parameter supported by the `web.database.connect()` function

Entries added to *environ*:

`environ['web.database.connection']`contains the `connection` object

`environ['web.database.cursor']`contains the `cursor` object

Middleware or applications further down the chain can access the database through these objects as follows:

```
def myApp(environ, start_response):
    result = []
    result.append('<html>')
    self.environ['web.database.cursor'].execute('SELECT * FROM test')
    rows = self.environ['web.database.cursor'].fetchall()
    for row in rows:
        result.append('<p>%s</p>'%row)
    result.append('</html>')
    start_response('200 OK', [('Content-Type','text/html')])
    return result

application = web.wsgi.database.Database(
    myApp,
    type='MySQLdb',
    database='test',
)
```

`web.wsgi.session` – Session Handling

The `web.wsgi.session` module provides one class `Session`

class `Session`(*application*, [***params*]) ***application***

A WSGI application or middleware component

****params***Any parameters which can be passed to `web.session.manager()`. If *cursor* is not specified it is assumed that you are using `Database` middleware and the cursor is obtained from `environ['web.database.cursor']`

Entries added to *environ*:

environ['web.session'] A `SessionManager` object as returned by web.session.manager(). You can obtain a session store using `store = self.environ['web.session'].store('testApp')` replacing `'testApp'` with the name of your application's store.

The `web.wsgi.session.Session` middleware requires the presence of the `Database` middleware and can be used as shown in the example below:

```
from web.wsgi import *

class simpleApp(base.BaseApplication):

    def printPage(self, title, url, link, url2, link2, data):
        self.output("""
            <html>
            <h1>%s</h1>
            <p><a href="%s">%s</a></p>
            <p><a href="%s">%s</a></p>
            <p>%s</p>
            </html>"""%(title, url, link, url2, link2, data)
        )
    def start(self):
        # Write a simple application
        store = self.environ['web.session'].store('testApp')

        if not self.environ['web.session'].created:
            if self.environ['web.cgi'].has_key('destroy') and self.environ['web.cgi']['destroy'
                self.environ['web.session'].destroy(ignoreWarning=True, sendCookieHeaders=False
                self.headers.append(self.environ['web.session'].response_headers[-1])
                self.printPage(
                    'Session Destroyed',
                    self.environ['SCRIPT_NAME'],
                    'Start Again', '','',''
                )
            else:
                self.environ['web.session'].setExpire(self.environ['web.session'].expireTime+5)
                self.environ['web.session'].setCookie()
                self.headers.append(self.environ['web.session'].response_headers[-1])
                data = []
                data.append('SessionID:  '+self.environ['web.session'].sessionID)
                data.append('Store Keys: '+str(store.keys()))
                data.append('Store App:  '+store.app)
                data.append('Variable1:  '+str(store['Variable1']))
                data.append('ExpireTime: '+str(self.environ['web.session'].expireTime))
                self.printPage(
                    'Welcome back',
                    self.environ['SCRIPT_NAME'],
                    'Visit Again',
                    self.environ['SCRIPT_NAME']+'?destroy=True',
                    'Destroy Session', '<p>Every time you visit this page the expiry time incre
                    '</p><p>'.join(data)
                )
        else:
            store['Variable1'] = 'Python Rules!'
            self.printPage(
                'New Session Started',
                self.environ['SCRIPT_NAME'],
                'Visit Again', '', '',
                "Set variable1 to 'Python Rules!'"
```

```
            )
        # Save changes
        self.environ['web.database.connection'].commit()

application = error.Error(
    database.Database(
        session.Session(
            cgi.CGI(
                simpleApp(),
            ),
            expire = 10,
            autoCreate = 1,
            driver = 'database',
        ),
        adapter = 'snakesql',
        database = 'wsgi-session',
        autoCreate = 1
    ),
)
```

You can test this example by running the WSGI server 'scripts/WSGIServer.py' and visiting http://localhost:8000/session

`web.wsgi.auth` – User Permission Handling

Auth handling middleware. If an application returns a `'403 Forbidden'` status message, the middleware intercepts it and instead provides a sign in form and sign in functionality.

Once a user is signed in, the user's information is added to the *environ* dictionary as `environ['web.auth.user']` for authorisation.

class `Session`(*application*, *driver*, [*store=None*], [*expire=0*], [*idle=0*], [*autoCreate=0*], [*app='auth'***application**], [*template='¡html¿¡body¿¡h1¿Please Sign In¡/h1¿%(form)s¡p¿%(message)s¡/p¿¡/body¿¡/html¿'], [*redirectMethod='http'*], [***driverParams*])

A WSGI application or middleware component

driver The type of driver being used. Currently only `'database'` is allowed

****driverParams*** Any parameters to be specified in the format *name=value* which are needed by the driver specified by *driver*

autoCreate If set to `True` the necessary tables will be created (removing any existing tables) if any of the tables are missing and a user named `john` with a password `bananas` will be set up with an access level of `1` to the application `app`. This is designed for easy testing of the module.

encryption The encryption method used to encrypt the password. Can be `None` or `'md5'`. `Warning you cannot change the encryption method once a user is added without resetting the password.`

store* or *app Store should be a valid `web.session` `Store` object for storing the auth session information. If not specified, a store can be obtained from the `environ['web.session']` object if the name of the store to used is specified by *app*.

expire An integer specifying the number of seconds before the user is signed out. A value of 0 disables the expire functionality and the user will be signed in until they sign out. **Note:** If the underlying session expires, the cookie is removed or the sign in idles before the expire time specified in *expire* the user will be signed out.

idle An integer specifying the maximum number of seconds between requests before the user is automatically signed out. A value of 0 disables the idle functionality allowing an unlimited amount of time between user requests. **Note:** If the underlying session expires, the cookie is removed or the sign in expires before the idle time specified in *idle* the user will be signed out.

template A string containing `%(form)s` and `%(message)s` for dictionary replacement of the sign in form and error message respectively.

redirectMethod Determines how the application should redirect back to the original code once a user is signed in. The default is HTTP redirection specified with *redirectMethod='http'* but alternatively a META refresh can be used, *redirectMethod='metaRefresh'* **Warning:** There currently appears to be a bug in the WSGI Server preventing HTTP redirection from working so META refresh redirection should be used.

Entries added to *environ*:

`environ['web.auth']` An `AuthManager` object as returned by web.auth.manager()

`environ['web.auth.user']` A user object for the current signed in user

`environ['REMOTE_USER']` The username of the signed in user

The example below demonstrates how to check if a user is signed in and if they are not signed in, provide them with a sign in form and handle the submissions until they are signed in.

```
import sys; sys.path.append('..//')
from web.wsgi import *

def simpleApp(environ, start_response):
    if not environ.has_key('web.auth.user'): # No user signed in
        start_response('403 User not signed in', [])
        return []
    elif not environ['web.auth.user'].authorise(app='app', level=1):
        start_response('403 The user does not have permission to access this application', [])
        return []
    else:
        start_response('200 OK', [('Content-type','text/html')])
        if environ['web.cgi'].has_key('mode') and environ['web.cgi']['mode'].value == 'signOut'
            environ['web.auth'].signOut()
            return ["""<html>
                <head><title>Auth Example</title></head>
                <body bgcolor="#ffffcc"><h1>Signed Out</h1><p><a href="auth">Sign in</a></p></b
                </html>"""]
        else:
            return ["""<html>
                <head><title>Auth Example</title></head>
                <body bgcolor="#ffffcc"><h1>Congratulations!</h1>
                <p>Signed in!</p>
                <p><a href="auth?mode=signOut">Sign out</a>, <a href="auth">Visit again</a></p>
                </body></html>"""]

# Middleware Setup
application = error.Error(
    database.Database(
        session.Session(
            cgi.CGI(
                auth.Auth(
                    simpleApp,
                    driver='database',
                    autoCreate=1,
                    expire=0,
                    idle=10,
                    template = """
                        <html>
                        <head><title>Auth Example</title></head>
                        <body bgcolor="#ffffcc">
```

```
                        <h1>Sign In</h1>
                        %(form)s
                        <p>%(message)s</p>
                        </body>
                        </html>
                    """,
                    redirectMethod='metaRefresh'
                ),
            ),
            expire = 1000,
            autoCreate = 1,
            driver='database',
        ),
        adapter = 'snakesql',
        database = 'wsgi-auth',
        autoCreate = 1
    ),
)
```

The message displayed under the sign in box is whatever you specify as the message after `403` in the `status` of `start_response()`.

You can test this example by running the WSGI server 'scripts/WSGIServer.py' and visiting http://localhost:8000/auth

1.14.4 Writing Applications

Below is a full example with a lot of functionality. It can be used as a base for your own applications.

```
import sys; sys.path.append('../')
from web.wsgi import *
import web.database.object, os

links = """<p><a href="example?mode=signOut">Sign out</a> |
<a href="example?mode=view">View</a> |
<a href="example?mode=add">Add</a></p>"""

def simpleApp(environ, start_response):

    person = web.database.object.Table("Person")
    person.add(column="String", name='firstName', required=True)
    person.addColumn(web.database.object.String(name="surname"))
    person.addColumn(
        web.database.object.StringSelect(
            name="profession",
            options=[None, 'Developer', 'Web Developer'],
            displayNoneAs='Not Specified'
        )
    )
    person.add(column="Bool", name='sex', displayTrueAs='Male', displayFalseAs='Female')
    database = web.database.object.Database()
    database.addTable(person)

    # Initialise the database
    database.init(environ['web.database.cursor'])

    if not database.tablesExist():
        database.createTables()
```

```
    mode = 'view'
    if environ['web.cgi'].has_key('mode'):
        mode = environ['web.cgi']['mode'].value

    # signIn mode needed to allow for sign in handling
    # you will be redirected to the correct place eventually
    if mode == 'signIn':
        start_response('403 User not signed in', [])
        environ['web.database.connection'].commit()
        return []
    elif mode == 'signOut':
        start_response('403 User not signed in', [])
        environ['web.database.connection'].commit()
        return []
    elif mode == 'add':
        if not environ.has_key('web.auth.user'): # No user signed in
            start_response('403 User not signed in', [])
            environ['web.database.connection'].commit()
            return []
        else:
            result = []
            form = database['Person'].form(stickyData={'mode':'add'})
            if len(environ['web.cgi']) > 1:             # Assume form submitted
                form.populate(environ['web.cgi'])
                if form.valid():
                    entry = database['Person'].insert(all=form.dict())
                    result.append('''<html><h1>Entry Added</h1>%s
                        <p><a href="example">Go Back</a></html>'''%(form.frozen())
                    )
                else:
                    result.append( """<html>%s<h1>Error</h1><p>There were some invalid fields.
                        Please correct them.</p>%s</html>"""%(links, form.html())
                    )
            else:
                result.append("""<html>%s<h1>Add Entry</h1>%s</html>"""%(links, form.html()))
            start_response('200 OK', [('Content-type','text/html')])
            environ['web.database.connection'].commit()
            return result
    else:
        entries = '<table border="1"><tr style="font-weight: bold;"><td>Firstname</td>'
        entries += '<td>Surname</td><td>Profession</td><td>Sex</td></tr>'
        for row in database['Person'].values():
            entries += '<tr><td>%s</td><td>%s</td><td>%s</td><td>%s</td></tr>'%(
                row['firstName'],
                row['surname'],
                row['profession'],
                row['sex']
            )
        entries += '</table>'
        info = """The table above shows people entries. To add an entry, click the add link abo
        but you will need to sign in using the username <tt>john</tt> and the password <tt>bana
        If you don't visit a page you will be signed out after 20 seconds and have to sign in a
        start_response('200 OK', [('Content-type','text/html')])
        environ['web.database.connection'].commit()
        return ["<html>%s<h1>Entries</h1><p>%s</p><p>%s</p></html>"%(links, entries,info)]


# Middleware Setup
application = error.Error(
```

```
        database.Database(
            session.Session(
                cgi.CGI(
                    auth.Auth(
                        simpleApp,
                        driver='database',
                        autoCreate=1,
                        expire=0,
                        idle=20,
                        template = """
                            <html>
                            <head><title>Sign In</title></head>
                            <body>
                            %s
                            <h1>Sign In</h1>
                            %%(form)s
                            <p>%%(message)s</p>
                            </body>
                            </html>
                        """%links,
                        redirectMethod='metaRefresh'
                    ),
                ),
                expire = 1000,
                autoCreate = 1,
                driver='database',
            ),
            adapter = 'snakesql',
            database = 'wsgi-example',
            autoCreate = 1
        ),
    )
```

You can test this example by running the WSGI server 'scripts/WSGIServer.py' and visiting http://localhost:8000/example

Note: The authour of the web modules is currently building a framework called Bricks which will automate many of the tasks involved in manually writing an application such as the one above.

1.14.5 Writing Your Own Middleware

See first the WSGI Middleware Introduction earlier in this document.

Eariler in this document we saw some simple middleware components and learned that for an object to be valid WSGI middleare it must take a WSGI application object as parameter and behave exaclty like a WSGI application itself.

With long middleware chains and functions being passed as parameters down the chain it can get a bit confusing to keep track of program flow.

Program flow is actually very straightfoward. The first piece of middleware is run first, any changes to the `environ` dictionary are passed on to the next piece of middleware and so on down the chain. Once the `start_response` function is called by the application at the end of the chain, the `status`, `headers` and application output are sent back up the chain to the server where they are sent to the web browser.

Here is a test application demonstrating middleware and program flow (the headers are not valid HTTP headers obviously):

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

```
import sys; sys.path.append('../../../')
import web.wsgi.base, time

class Application(web.wsgi.base.BaseApplication):
    def start(self):
        self.output('Environ Order:\n')
        self.environ['Application'] = time.time()
        time.sleep(1)
        self.headers.append(('Appliction',str(time.time())))
        self.output('Middleware1 ',self.environ['Middleware1'])
        self.output('\n')
        self.output('Middleware2 ',self.environ['Middleware2'])
        self.output('\n')
        self.output('Application ', self.environ['Application'])
        self.output('\n')

class Middleware1(web.wsgi.base.BaseMiddleware):
    def environ(self, environ):
        time.sleep(1)
        environ['Middleware1'] = time.time()
        return environ

    def headers(self, headers):
        time.sleep(1)
        headers.append(('Middleware1',str(time.time())))
        return headers

    def transform(self, output):
        return output + ['Middleware1\n']


class Middleware2(web.wsgi.base.BaseMiddleware):
    def environ(self, environ):
        time.sleep(1)
        environ['Middleware2'] = time.time()
        return environ

    def headers(self, headers):
        time.sleep(1)
        headers.append(('Middleware2',str(time.time())))
        return headers

    def transform(self, output):
        return output + ['Middleware2\n']

print "Running test..."
application = web.wsgi.runCGI(Middleware1(Middleware2(Application())))
```

The program will not run from a WSGI server because of the incorrect HTTP headers but you can run it from the command line. The output should look something like this:

```
Status: 200 OK
Content-type: text/html
Appliction: 1105847968.69
Middleware2: 1105847969.69
Middleware1: 1105847970.69

Environ Order:
Middleware1 1105847966.68
Middleware2 1105847967.69
Application 1105847967.69

Transform Order:
Middleware2
Middleware1
```

You can see that `environ` is modified by `Middleware1` then `Middleware2` then `Application`. Headers and return transforms are made in exactly the opposite order.

At each stage of the application and middleware chain the component can either return an list of strings in one go or return an iterable.

We also learned earlier that WSGI middleware can be implemented as a class and usually performs one of the following actions or a combination of them.

- Change the `environ` dictionary
- Change the application's `status`
- Change the HTTP `headers`
- Change the return value of the application

The `web.wsgi.base` module provides a base Middleware class with methods to accomplish these tasks so that you don't need to worry quite so much about program flow or how to implement your middleware.

class `BaseMiddleware`(*application*)

application should always be the first parameter to a derived middleware class, but you may also wish to have other parameters in derived classes to allow the middleware to be configured.

Warning: It is important you carefully read the documentation for the `__init__()` and `setup()` methods to understand where to configure variables.

The class defines the following attributes:

`application`

The WSGI application (or middleware stack) to which this middleware should be added.

The class defines the following methods:

`__init__`(*application*)

You can override the `__init__()` method but the first parameter should always be for the application object. Parameters used to configure the class at load time can be specified in the `__init__()` method but any variables which need to be reset every time the middleware is used should be specified in the `setup()` method. This is because a WSGI server only loads the middleware once but runs it lots of times so if a variable is specified in the `__init__()` method it would only be set once and on subsequent calls would retain the value from the previous call.

`setup`()

The `setup()` method is used to configure any class attributes which need to be configured every time the middleare is run and not just when the middleare is loaded. See the documentation for the `__init__()` method too.

__call__(*environ, start_response*)

You should not need to modify this method but is documented here for a complete understanding as it provides the functionality which makes derived classes WSGI middleware.

The fitst task of this method is to call `setup()` to re-initialise any variables which need to be set every time the class is run. It then intercepts the `environ` dictionary as well as the `headers` and `status` parameters sent by the WSGI server to the `start_response()` function. It then sends the `environ` dictionary to the `environ()` method for modification. The `status`, `headers` and `exc_info` parameters are sent to the `response()` method which controls the order in which the different parameters are modified. The `response()` method sends the parameters to the `status()`, `headers` and `exc_info()` methods for modification. The new values are then returned to the `__call__` where a modified application object is returned.

response(*status, headers,* [*exc_info=None*])

Calls the `status()`, `headers` and `exc_info()` methods to modify the respective parameters then returns the modified values in the order *status, headers, exc_info* to the `__call__()` method. Can be over-ridden to change the order in which the parameters are modified.

environ(*environ*)

Provides the dictionary *environ* for modification. Must return the `environ` dictionary to be passed on down the middleware chain.

status(*status*)

Provides the *status* string for modification. Must return the `status` string to be passed on down the middleware chain.

headers(*headers*)

Provides the *headers* list for modification. Must return the `headers` list to be passed on down the middleware chain.

exc_info(*exc_info*)

Provides the *exc_info* tuple object generated by a previous error (if one exists) for modification. Must return the `exc_info` tuple to be passed on down the middleware chain.

result(*result*)

Used to transform the body of output returned from the previous item in the middleware stack.

Be aware that you may need to have checked content-type headers and change the content length header if it is set if you intend to change the length of the returned information.

result is an iterable and an iterable should be returned from the output.

To produce your own middleware class, simply over-ride the appropriate methods in your class derived from the `BaseMiddleware` class. If you wish to pass information between the various methods, you should set member variables of the class which can then be read by all the methods. You can change the order in which some of the methods are called by overriding `response()` and calling the methods in the order you wish.

For some examples of how to write middleware comonents using this class look at the source code of the `web.wsgi` middleware classes.

APPENDIX

A

Reporting Bugs

Please email bugs at pythonweb.org

APPENDIX

B

History and License

B.1 History of the software

Python Web Modules are released under the GNU LGPL

MODULE INDEX

D

datetime, 19

W

web, 1
web.auth, 2
web.database, 22
web.database.object, 73
web.environment, 107
web.error, 99
web.form, 109
web.form.field.basic, 112
web.form.field.extra, 115
web.form.field.typed, 114
web.image, 120
web.mail, 122
web.session, 124
web.template, 136, 141
web.wsgi, 145

INDEX

Symbols

__call__() (callable method), 165
__getitem__() (method), 95, 97, 111
__getitem__() (bool method), 70
__init__() (callable method), 164
_cleanupProbability (Float attribute), 133
_deleteCookieString() (method), 135
_function() (method), 68
_seed (String attribute), 133
_setCookieString() (method), 135

A

active (String attribute), 18
addAction() (method), 111
addApp() (method), 16
addColumn() (method), 96
addField() (method), 111
addGroup() (method), 17
addMultiple() (method), 96
addRelated() (method), 96
addRole() (method), 16
addSingle() (method), 96
addTable() (method), 94
addUser() (method), 16
app (String attribute), 135
appExists() (method), 16
application (dictionary attribute), 164
apps() (method), 16
AuthAdmin (class in web.auth), 15
AuthManager (class in web.auth), 18
authorise() (method), 18
AuthSession (class in web.auth), 17
AuthUser (class in web.auth), 18
autoCreated (Bool attribute), 15

B

baseCursor (attribute), 65
BaseMiddleware (class in web.wsgi), 164
baseType (attribute), 70

C

CGI (class in web.wsgi), 153
Checkbox (class in web.form.field.basic), 113
CheckBoxGroup (class in web.form.field.basic), 114
childTables (list attribute), 70
cleanup() (method), 134
close() (method), 64
code() (String method), 105
Column (class in web.database), 70
column()
    method, 58, 68
    bool method, 70
column (QueryBuilder attribute), 97
columnExists()
    method, 96
    bool method, 70
columns() (method), 96
columns (list attribute), 69
completeAuthEnvironment() (Bool method), 16
completeEnvironment() (method), 109
completeSessionEnvironment() (Bool method), 133
connect() (in module web.database), 26, 63
connection (attribute), 65
context (Integer attribute), 104
Converter (class in web.database), 71
converter (attribute), 70
cookie (Dict attribute), 133
cookieSessionID() (String method), 134
count() (method), 61, 68
create()
    method, 96
    cursor method, 60, 67
    String method, 133
createAuthEnvironment() (method), 16
createEnvironment() (method), 109
createSessionEnvironment() (method), 133
createTables() (method), 95
currentURL() (in module web.wsgi), 152

cursor() (in module web.database), 29, 64
cursor (cursor attribute), 95

D

Database
  class in web.database.object, 94, 96
  class in web.wsgi, 156
databaseToValue() (method), 71
date
  class in datetime, 19
  String attribute, 104
datetime
  class in datetime, 20
  extension module, **19**
  module, 19
day (Integer attribute), 20
debug() (String method), 105
default (attribute), 70
delete()
  method, 97, 135
  cursor method, 58, 67
deleteCookie() (method), 135
description() (String method), 113
destroy() (String method), 134
dict() (method), 95, 96, 98, 111
Documents (class in web.wsgi), 156
driver() (in module web.environment), 108
drop()
  method, 96
  cursor method, 60, 68
dropTables() (method), 95

E

email (String attribute), 18
empty() (method), 135
encode() (in module web), 2
environ() (dictionary method), 165
EnvironmentDriver (class in web.environment), 109
enycryption (Bool attribute), 16
error() (String method), 113
error
  Error tuple attribute, 104
  String attribute, 133
ErrorInformation (class in web.error), 104
errorType (Error attribute), 104
errorValue (String attribute), 104
exc_info() (Exception method), 165
execute() (method), 65
executemany() (method), 65
exists() (method), 96, 134
expire
  attribute, 17
  Integer attribute, 133
export() (method), 64, 69

F

fetchall() (method), 65
fetchone() (method), 65
Field (class in web.form.field.basic), 112
field() (method), 111
File (class in web.form.field.basic), 114
firstname (String attribute), 18
Form (class in web.form), 111
form()
  method, 97
  String method, 97
format (String attribute), 104
frozen()
  method, 111
  String method, 113

G

get() (method), 135
group (String attribute), 18
groupExists() (method), 17
groups() (method), 17

H

handle()
  method, 19
  in module web.error, 101
has_key()
  method, 95, 96, 98, 111, 135
  bool method, 70
header() (in module web), 2
headers() (list method), 165
Hidden (class in web.form.field.basic), 113
hidden()
  method, 111
  String method, 113
hour (Integer attribute), 20
html()
  method, 111
  String method, 113
html2tuple() (in module web.image), 120

I

info (attribute), 65
init() (method), 94
Input (class in web.form.field.basic), 113
insert()
  method, 96
  cursor method, 55, 66
insertMany() (cursor method), 67
isoformat() (method), 21
isRelated() (method), 98

items() (method), 95, 96, 98, 111, 135

K

key (attribute), 70
keys() (method), 95, 96, 98, 111, 135

L

levels() (method), 16
levels (Dict attribute), 18
load() (Bool method), 133

M

mail() (in module web.mail), 122
max() (method), 61, 68, 97
Menu (class in web.form.field.basic), 114
microsecond (Integer attribute), 20, 21
min() (method), 61, 68, 97
minute (Integer attribute), 20, 21
month (Integer attribute), 19, 20

N

name() (String method), 113
name
    attribute, 70
    String attribute, 95
    string attribute, 69
now() (datetime method), 21

O

order() (method), 54, 69
ouput() (String method), 104
output() (method), 95

P

parentTables (list attribute), 70
parse() (in module web.template), 136
Password (class in web.form.field.basic), 113
password (String attribute), 18
populate()
    method, 111
    None method, 113
position (attribute), 70
primaryKey (string attribute), 70
pythonVersion (String attribute), 104

R

RadioGroup (class in web.form.field.basic), 114
relate() (method), 98
remove() (method), 111
removeApp() (method), 16
removeAuthEnvironment() (method), 16
removeEnvironment() (method), 109
removeGroup() (method), 17
removeRole() (method), 16
removeSessionEnvironment() (method), 133
removeUser() (method), 16
required (attribute), 70
Reset (class in web.form.field.basic), 113
response() (sequence method), 165
response_headers (List attribute), 133
result() (Iterable method), 165
roleExists() (method), 16
roles() (method), 16
roles (Dict attribute), 18
Row (class in web.database.object), 97
row() (method), 97
rowExists() (method), 96
rowid (Integer attribute), 98
runCGI() (in module web.wsgi), 151
runWebServer() (in module web.template), 141

S

second (Integer attribute), 20, 21
Select (class in web.form.field.basic), 114
select() (method), 52, 66, 97
sendCookieHeaders() (method), 135
sent_headers (List attribute), 133
Session (class in web.wsgi), 156, 158
sessionID (String attribute), 133
SessionManager (class in web.session), 132
set() (method), 135
setCookie() (method), 135
setError() (String method), 113
setExpire() (method), 134
setLevel() (method), 16
setRole() (method), 17
setup() (callable method), 164
signIn() (method), 17
SignInHandler (class in web.auth), 19
signOut() (method), 17
sql (attribute), 65
sqlQuotes (string attribute), 71
sqlToValue() (method), 71
status() (string method), 165
Store (class in web.session), 135
store() (Store Object method), 134
store (attribute), 17
strftime() (method), 21
strip() (in module web.template), 141
Submit (class in web.form.field.basic), 113
surname (String attribute), 18

T

Table (class in web.database), 69
table()
    method, 95
    in module web.template, 141

table (attribute), 70
tablesExist() (method), 95
templateDict() (method), 112
TextArea (class in web.form.field.basic), 114
time (class in datetime), 20
timetuple() (method), 21
traceback() (String method), 104
type
    attribute, 70
    string attribute, 71

U

unique (attribute), 70
unrelate() (method), 98
update()
    method, 98
    cursor method, 56, 67
user() (method), 16, 18
userExists() (method), 16
userInfo() (method), 17
username() (method), 17
username (String attribute), 18
users() (method), 16

V

valid()
    method, 134
    Bool method, 111
    True or False method, 113
value
    List attribute, 114
    String attribute, 113
values() (method), 95, 96, 98, 111, 136
valueToDatabase() (method), 71
valueToSQL() (method), 71

W

web (extension module), **1**
web.auth
    extension module, **2**
    module, 2
web.database (extension module), **22**
web.database.object (extension module), **73**
web.environment
    extension module, **107**
    module, 107
web.error
    extension module, **99**
    module, 99
web.error.error() (in module web.error), 104
web.error.info() (in module web.error), 101
web.form (extension module), **109**
web.form.field.basic (extension module), **112**
web.form.field.extra (extension module), **115**
web.form.field.typed (extension module), **114**
web.image (extension module), **120**
web.mail (extension module), **122**
web.session (extension module), **124**
web.template (extension module), **136**, **141**
web.wsgi (extension module), **145**
where() (method), 53, 69
wrap() (in module web.template), 141

Y

year (Integer attribute), 19, 20