# HOWTO

***Release 2.6***

**Guido van Rossum**
**Fred L. Drake, Jr., editor**

October 02, 2008

**Python Software Foundation**
Email: docs@python.org

## Contents

**1 The low-level view** **2**
1.1 Common Gateway Interface . . . 2
Simple script for testing CGI . . . 3
Setting up CGI on your own server . . . 3
Common problems with CGI scripts . . . 3
1.2 mod_python . . . 4
1.3 FastCGI and SCGI . . . 5
Setting up FastCGI . . . 5
1.4 mod_wsgi . . . 6

**2 Step back: WSGI** **6**
2.1 WSGI Servers . . . 6
2.2 Case study: MoinMoin . . . 7

**3 Model-view-controller** **7**

**4 Ingredients for web sites** **7**
4.1 Templates . . . 8
4.2 Data persistence . . . 8

**5 Frameworks** **9**
5.1 Some notable frameworks . . . 9
Django . . . 9
TurboGears . . . 10
Other notable frameworks . . . 10

**Index** **11**

**Author** Marek Kubica

**Abstract**

This document shows how Python fits into the web. It presents some ways on how to integrate Python with the web server and general practices useful for developing web sites.

Programming for the Web has become a hot topic since the raise of the "Web 2.0", which focuses on user-generated content on web sites. It has always been possible to use Python for creating web sites, but it was a rather tedious task. Therefore, many so-called "frameworks" and helper tools were created to help developers creating sites faster and these sites being more robust. This HOWTO describes some of the methods used to combine Python with a web server to create dynamic content. It is not meant as a general introduction as this topic is far too broad to be covered in one single document. However, a short overview of the most popular libraries is provided.

**See Also:**

While this HOWTO tries to give an overview over Python in the Web, it cannot always be as up to date as desired. Web development in Python is moving forward rapidly, so the wiki page on Web Programming might be more in sync with recent development.

# 1 The low-level view

When a user enters a web site, his browser makes a connection to the site's webserver (this is called the *request*). The server looks up the file in the file system and sends it back to the user's browser, which displays it (this is the *response*). This is roughly how the unterlying protocol, HTTP works.

Now, dynamic web sites are not files in the file system, but rather programs which are run by the web server when a request comes in. They can do all sorts of useful things, like display the postings of a bulletin board, show your mails, configurate software or just display the current time. These programs can be written in about any programming language the server supports, so it is easy to use Python for creating dynamic web sites.

As most of HTTP servers are written in C or C++, they cannot execute Python code in a simple way – a bridge is needed between the server and the program. These bridges or rather interfaces define how programs interact with the server. In the past there have been numerous attempts to create the best possible interface, but there are only a few worth mentioning.

Not every web server supports every interface. Many web servers do support only old, now-obsolete interfaces. But they can often be extended using some third-party modules to support new interfaces.

## 1.1 Common Gateway Interface

This interface is the oldest one, supported by nearly every web server out of the box. Programs using CGI to communicate with their web server need to be started by the server for every request. So, every request starts a new Python interpreter – which takes some time to start up – thus making the whole interface only usable for low load situations.

The upside of CGI is that it is simple – writing a program which uses CGI is a matter of about three lines of code. But this simplicity comes at a price: it does very few things to help the developer.

Writing CGI programs, while still possible, is not recommended anymore. With WSGI (more on that later) it is possible to write programs that emulate CGI, so they can be run as CGI if no better option is available.

**See Also:**

The Python standard library includes some modules that are helpful for creating plain CGI programs:

- `cgi` – Handling of user input in CGI scripts

- `cgitb` – Displays nice tracebacks when errors happen in of CGI applications, instead of presenting a "500 Internal Server Error" message

The Python wiki features a page on CGI scripts with some additional information about CGI in Python.

### Simple script for testing CGI

To test whether your web server works with CGI, you can use this short and simple CGI program:

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-

# enable debugging
import cgitb; cgitb.enable()

print "Content-Type: text/plain;charset=utf-8"
print

print "Hello World!"
```

You need to write this code into a file with a `.py` or `.cgi` extension, this depends on your web server configuration. Depending on your web server configuration, this file may also need to be in a `cgi-bin` folder, for security reasons.

You might wonder what the `cgitb` line is about. This line makes it possible to display a nice traceback instead of just crashing and displaying an "Internal Server Error" in the user's browser. This is useful for debugging, but it might risk exposing some confident data to the user. Don't use it when the script is ready for production use. Still, you should *always* catch exceptions, and display proper error pages – end-users don't like to see nondescript "Internal Server Errors" in their browsers.

### Setting up CGI on your own server

If you don't have your own web server, this does not apply to you. You can check whether if works as-is and if not you need to talk to the administrator of your web server anyway. If it is a big hoster, you can try filing a ticket asking for Python support.

If you're your own administrator or want to install it for testing purposes on your own computers, you have to configure it by yourself. There is no one and single way on how to configure CGI, as there are many web servers with different configuration options. The currently most widely used free web server is Apache HTTPd, Apache for short – this is the one that most people use, it can be easily installed on nearly every system using the systems' package management. But lighttpd has been gaining attention since some time and is said to have a better performance. On many systems this server can also be installed using the package management, so manually compiling the web server is never needed.

- On Apache you can take a look into the Dynamic Content with CGI tutorial, where everything is described. Most of the time it is enough just to set `+ExecCGI`. The tutorial also describes the most common gotchas that might arise.
- On lighttpd you need to use the CGI module which can be configured in a straightforward way. It boils down to setting `cgi.assign` properly.

### Common problems with CGI scripts

Trying to use CGI sometimes leads to small annoyances that one might experience while trying to get these scripts to run. Sometimes it happens that a seemingly correct script does not work as expected, which is caused by some small hidden reason that's difficult to spot.

Some of these reasons are:

- The Python script is not marked executable. When CGI scripts are not executable most of the web servers will let the user download it, instead of running it and sending the output to the user. For CGI scripts to run properly the `+x` bit needs to be set. Using `chmod a+x your_script.py` might already solve the problem.
- The line endings must be of Unix-type. This is important because the web server checks the first line of the script (called shebang) and tries to run the program specified there. It gets easily confused by Windows line endings (Carriage Return & Line Feed, also called CRLF), so you have to convert the file to Unix line endings (only Line Feed, LF). This can be done automatically by uploading the file via FTP in text mode instead of binary mode, but the preferred way is just telling your editor to save the files with Unix line endings. Most proper editors support this.
- Your web server must be able to read the file, you need to make sure the permissions are fine. Often the server runs as user and group `www-data`, so it might be worth a try to change the file ownership or making the file world readable by using `chmod a+r your_script.py`.
- The webserver must be able to know that the file you're trying to access is a CGI script. Check the configuration of your web server, maybe there is some mistake.
- The path to the interpreter in the shebang (`#!/usr/bin/env python`) must be currect. This line calls `/usr/bin/env` to find Python, but it'll fail if there is no `/usr/bin/env`. If you know where your Python is installed, you can also use that path. The commands `whereis python` and `type -p python` might also help to find where it is installed. Once this is known, the shebang line can be changed accordingly: `#!/usr/bin/python`.
- The file must not contain a BOM (Byte Order Mark). The BOM is meant for determining the byte order of UTF-16 encodings, but some editors write this also into UTF-8 files. The BOM interferes with the shebang line, so be sure to tell your editor not to write the BOM.
- *mod_python* might be making problems. mod_python is able to handle CGI scripts by itself, but it can also be a source for problems. Be sure you disable it.

## 1.2 mod_python

People coming from PHP often find it hard to grasp how to use Python in the web. Their first thought is mostly mod_python because they think that this is the equivalent to `mod_php`. Actually it is not really. It does embed the interpreter into the Apache process, thus speeding up requests by not having to start a Python interpreter every request. On the other hand, it is by far not "Python intermixed with HTML" as PHP often does. The Python equivalent of that is a template engine. mod_python itself is much more powerful and gives more access to Apache internals. It can emulate CGI, it can work an a "Python Server Pages" mode similar to JSP which is "HTML intermangled with Python" and it has a "Publisher" which destignates one file to accept all requests and decide on what to do then.

But mod_python has some problems. Unlike the PHP interpreter the Python interpreter uses caching when executing files, so when changing a file the whole web server needs to be re-started to update. Another problem ist the basic concept – Apache starts some child processes to handle the requests and unfortunately every child process needs to load the whole Python interpreter even if it does not use it. This makes the whole web server slower. Another problem is that as mod_python is linked against a specific version of `libpython`, it is not possible to switch from an older version to a newer (e.g. 2.4 to 2.5) without recompiling mod_python. mod_python is also bound to the Apache web server, so programs written for mod_python cannot easily run on other web servers.

These are the reasons why mod_python should be avoided when writing new programs. In some circumstances it might be still a good idea to use mod_python for deployment, but WSGI makes it possible to run WSGI programs under mod_python as well.

## 1.3 FastCGI and SCGI

FastCGI and SCGI try to solve the performance problem of CGI in another way. Instead of embedding the interpreter into the web server, they create long-running processes which run in the background. There still is some module in the web server which makes it possible for the web server to "speak" with the background process. As the background process is independent from the server, it can be written in any language of course also in Python. The language just needs to have a library which handles the communication with the web server.

The difference between FastCGI and SCGI is very small, as SCGI is essentially just a "simpler FastCGI". But as the web server support for SCGI is limited most people use FastCGI instead, which works the same way. Almost everything that applies to SCGI also applies to FastCGI as well, so we'll only write about the latter.

These days, FastCGI is never used directly. Just like `mod_python` it is only used for the deployment of WSGI applications.

**See Also:**

- FastCGI, SCGI, and Apache: Background and Future is a discussion on why the concept of FastCGI and SCGI is better that that of mod_python.

### Setting up FastCGI

Depending on the web server you need to have a special module.

- Apache has both mod_fastcgi and mod_fcgid. `mod_fastcgi` is the original one, but it has some licensing issues that's why it is sometimes considered non-free. `mod_fcgid` is a smaller, compatible alternative. One of these modules needs to be loaded by Apache.
- lighttpd ships its own FastCGI module as well as an SCGI module.
- nginx also supports FastCGI.

Once you have installed and configured the module, you can test it with the following WSGI-application:

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-

from cgi import escape
import sys, os
from flup.server.fcgi import WSGIServer

def app(environ, start_response):
    start_response('200 OK', [('Content-Type', 'text/html')])

    yield '<h1>FastCGI Environment</h1>'
    yield '<table>'
    for k, v in sorted(environ.items()):
        yield '<tr><th>%s</th><td>%s</td></tr>' % (escape(k), escape(v))
    yield '</table>'

WSGIServer(app).run()
```

This is a simple WSGI application, but you need to install flup first, as flup handles the low level FastCGI access.

**See Also:**

There is some documentation on setting up Django with FastCGI, most of which can be reused for other WSGI-compliant frameworks and libraries. Only the `manage.py` part has to be changed, the example used here can be used instead. Django does more or less the exact same thing.

### 1.4 mod_wsgi

mod_wsgi is an attempt to get rid of the low level gateways. As FastCGI, SCGI, mod_python are mostly used to deploy WSGI applications anyway, mod_wsgi was started to directly embed WSGI aplications into the Apache web server. The benefit from this approach is that WSGI applications can be deployed much easier as is is specially designed to host WSGI applications – unlike the other low level methods which have glue code to host WSGI applications (like flup which was mentioned before). The downside is that mod_wsgi is limited to the Apache web server, other servers would need their own implementations of mod_wsgi.

It supports two modes: the embedded mode in which it integrates with the Apache process and the daemon mode which is more FastCGI-like. Contrary to FastCGI, mod_wsgi handles the worker-processes by itself which makes administration easier.

## 2 Step back: WSGI

WSGI was already mentioned several times so it has to be something important. In fact it really is, so now it's time to explain.

The *Web Server Gateway Interface*, **PEP 333** or WSGI for short is currently the best possible way to Python web programming. While it is great for programmers writing frameworks, the normal person does not need to get in direct contact with it. But when choosing a framework for web development it is a good idea to take one which supports WSGI.

The big profit from WSGI is the unification. When your program is compatible with WSGI – that means that your framework has support for WSGI, your program can be deployed on every web server interface for which there are WSGI wrappers. So you do not need to care about whether the user uses mod_python or FastCGI – with WSGI it just works on any gateway interface. The Python standard library contains its own WSGI server `wsgiref`, which is a small web server that can be used for testing.

A really great WSGI feature are the middlewares. Middlewares are layers around your program which can add various functionality to it. There is a number of middlewares already available. For example, instead of writing your own session management (to identify a user in subsequent requests, as HTTP does not maintain state, so it does now know that the requests belong to the same user) you can just take one middleware, plug it in and you can rely an already existing functionality. The same thing is compression – say you want to compress your HTML using gzip, to save your server's bandwidth. So you only need to plug-in a middleware and you're done. Authentication is also a problem easily solved using a middleware.

So, generally – although WSGI may seem complex, the initial phase of learning can be very rewarding as WSGI does already have solutions to many problems that might arise while writing web sites.

### 2.1 WSGI Servers

The code that is used to connect to various low level gateways like CGI or mod_python is called *WSGI server*. One of these servers is `flup` which was already mentioned and supports FastCGI, SCGI as well as AJP. Some of these servers are written in Python as `flup` is, but there also exist others which are written in C and can be used as drop-in replacements.

There are quite a lot of servers already available, so a Python web application can be deployed nearly everywhere. This is one big advantage that Python has compared with other web techniques.

**See Also:**

A good overview of all WSGI-related code can be found in the WSGI wiki, which contains an extensive list of WSGI servers, which can be used by *every* application supporting WSGI.

You might be interested in some WSGI-supporting modules already contained in the standard library, namely:

- `wsgiref` – some tiny utilities and servers for WSGI

### 2.2 Case study: MoinMoin

What does WSGI give the web application developer? Let's take a look on one long existing web application written in Python without using WSGI.

One of the most widely used wiki software is MoinMoin. It was created in 2000, so it predates WSGI by about three years. While it now includes support for WSGI, older versions needed separate code to run on CGI, mod_python, FastCGI and standalone. Now, this all is possible by using WSGI and the already-written gateways. For running with on FastCGI `flup` can be used, for running a standalone server `wsgiref` is the way to go.

## 3 Model-view-controller

The term *MVC* is often heard in statements like "framework *foo* supports MVC". While MVC is not really something technical but rather organisational, many web frameworks use this model to help the developer to bring structure into his program. Bigger web applications can have lots of code so it is a good idea to have structure in the program right from the beginnings. That way, even users of other frameworks (or even languages, as MVC is nothing Python-specific) can understand the existing code easier, as they are already familiar with the structure.

MVC stands for three components:

- The *model*. This is the data that is meant to modify. In Python frameworks this component is often represented by the classes used by the object-relational mapper. So, all declarations go here.
- The *view*. This component's job is to display the data of the model to the user. Typically this component is represented by the templates.
- The *controller*. This is the layer between the user and the model. The controller reacts on user actions (like opening some specific URL) and tells the model to modify the data if neccessary.

While one might think that MVC is a complex design pattern, in fact it is not. It is used in Python because it has turned out to be useful for creating clean, maintainable web sites.

**Note:** While not all Python frameworks explicitly support MVC, it is often trivial to create a web site which uses the MVC pattern by seperating the data logic (the model) from the user interaction logic (the controller) and the templates (the view). That's why it is important not to write unneccessary Python code in the templates – it is against MVC and creates more chaos.

**See Also:**

The english Wikipedia has an article about the Model-View-Controller pattern, which includes a long list of web frameworks for different programming languages.

## 4 Ingredients for web sites

Web sites are complex constructs, so tools were created to help the web site developer to make his work maintainable. None of these tools are in any way Python specific, they also exist for other programming languages as well. Of course, developers are not forced to use these tools and often there is no "best" tool, but it is worth informing yourself before choosing something because of the big number of helpers that the developer can use.

**See Also:**

People have written far more components that can be combined than these presented here. The Python wiki has a page about these components, called Web Components.

## 4.1 Templates

Mixing of HTML and Python code is possible with some libraries. While convenient at first, it leads to horribly unmaintainable code. That's why templates exist. Templates are, in the simplest case, just HTML files with placeholders. The HTML is sent to the user's browser after filling out the placeholders.

Python already includes such simple templates:

```
# a simple template
template = "<html><body><h1>Hello %s!</h1></body></html>"
print template % "Reader"
```

The Python standard library also includes some more advanced templates usable through `string.Template`, but in HTML templates it is needed to use conditional and looping contructs like Python's *for* and *if*. So, some *template engine* is needed.

Now, Python has a lot of template engines which can be used with or without a framework. Some of these are using a plain-text programming language which is very easy to learn as it is quite limited while others use XML so the template output is always guaranteed to be valid XML. Some frameworks ship their own template engine or recommend one particular. If one is not yet sure, using these is a good idea.

**Note:** While Python has quite a lot of different template engines it usually does not make sense to use a homebrewed template system. The time needed to evaluate all templating systems is not really worth it, better invest the time in looking through the most popular ones. Some frameworks have their own template engine or have a recommentation for one. It's wise to use these.

Popular template engines include:

- Mako
- Genshi
- Jinja

**See Also:**

Lots of different template engines divide the attention between themselves because it's easy to create them in Python. The page Templating in the wiki lists a big, ever-growing number of these.

## 4.2 Data persistence

*Data persistence*, while sounding very complicated is just about storing data. This data might be the text of blog entries, the postings of a bulletin board or the text of a wiki page. As always, there are different ways to store informations on a web server.

Often relational database engines like MySQL or PostgreSQL are used due to their good performance handling very large databases consisting of up to millions of entries. These are *queried* using a language called SQL. Python programmers in general do not like SQL too much, they prefer to work with objects. It is possible to save Python objects into a database using a technology called ORM. ORM translates all object-oriented access into SQL code under the hood, the user does not need to think about it. Most frameworks use ORMs and it works quite well.

A second possibility is using files that are saved on the hard disk (sometimes called flatfiles). This is very easy, but is not too fast. There is even a small database engine called SQLite which is bundled with Python in the `sqlite` module and uses only one file. This database can be used to store objects via an ORM and has no other dependencies. For smaller sites SQLite is just enough. But it is not the only way in which data can be saved into the file systems. Sometimes normal, plain text files are enough.

The third and least used possibility are so-called object oriented databases. These databases store the *actual objects* instead of the relations that OR-mapping creates between rows in a database. This has the advantage that nearly all

objects can be saven in a straightforward way, unlike in relational databases where some objects are very hard to represent with ORMs.

Frameworks often give the users hints on which method to choose, it is usually a good idea to stick to these unless there are some special requirements which require to use the one method and not the other.

**See Also:**

- Persistence Tools lists possibilities on how to save data in the file system, some of these modules are part of the standard library
- Database Programming helps on choosing a method on how to save the data
- SQLAlchemy, the most powerful OR-Mapper for Python and Elixir which makes it easier to use
- SQLObject, another popular OR-Mapper
- ZODB and Durus, two object oriented databases

# 5 Frameworks

As web sites can easily become quite large, there are so-called frameworks which were created to help the developer with making these sites. Although the most well-known framework is Ruby on Rails, Python does also have its own frameworks which are partly inspired by Rails or which were existing a long time before Rails.

Two possible approaches to web frameworks exist: the minimalistic approach and the all-inclusive approach (somtimes called *full-stack*). Frameworks which are all-inclusive give you everything you need to start working, like a template engine, some way to save and access data in databases and many features more. Most users are best off using these as they are widely used by lots of other users and well documented in form of books and tutorials. Other web frameworks go the minimalistic approach trying to be as flexible as possible leaving the user the freedom to choose what's best for him.

The majority of users is best off with all-inclusive framewors. They bring everything along so a user can just jump in and start to code. While they do have some limitations they can fullfill 80% of what one will ever want to perfectly. They consist of various components which are designed to work together as good as possible.

The multitude of web frameworks written in Python demonstrates that it is really easy to write one. One of the most well-known web applications written in Python is Zope which can be regarded as some kind of big framework. But Zope was not the only framework, there were some others which are by now nearly forgotten. These do not need to be mentioned anymore, because most people that used them moved on to newer ones.

## 5.1 Some notable frameworks

There is an incredible number of frameworks, so there is no way to describe them all. It is not even neccessary, as most of these frameworks are nothing special and everything that can be done with these can also be done with one of the popular ones.

### Django

Django is a framework consisting of several tightly coupled elements which were written from scratch and work together very well. It includes an ORM which is quite powerful while being simple to use and has a great online administration interface which makes it possible to edit the data in the database with a browser. The template engine is text-based and is designed to be usable for page designers who cannot write Python. It supports so-called template inheritance and filters (which work like Unix pipes). Django has many handy features bundled, like creation of RSS feeds or generic views which make it possible to write web sites nearly without any Python code.

It has a big, international community which has created many sites using Django. There are also quite a lot of add-on projects which extend Django's normal functionality. This is partly due to Django's well written online documentation and the Django book.

**Note:** Although Django is an MVC-style framework, it calls the components differently, which is described in the Django FAQ.

### TurboGears

The other popular web framework in Python is TurboGears. It takes the approach of using already existing components and combining them with glue code to create a seamless experience. TurboGears gives the user more flexibility on which components to choose, the ORM can be switched between some easy to use but limited and complex but very powerful. Same goes for the template engine. One strong point about TurboGears is that the components that it consists of can be used easily in other projects without depending on TurboGears, for example the underlying web server CherryPy.

The documentation can be found in the TurboGears wiki, where links to screencasts can be found. TurboGears has also an active user community which can respond to most related questions. There is also a TurboGears book published, which is a good starting point.

The plan for the next major version of TurboGears, version 2.0 is to switch to a more flexible base provided by another very flexible web framework called Pylons.

### Other notable frameworks

These two are of course not the only frameworks that are available, there are also some less-popular frameworks worth mentioning.

One of these is the already mentioned Zope, which has been around for quite a long time. With Zope 2.x having been known as rather un-pythonic, the newer Zope 3.x tries to change that and therefore gets more acceptance from Python programmers. These efforts already showed results, there is a project which connects Zope with WSGI called Repoze and another project called Grok which makes it possible for "normal" Python programmers use the very mature Zope components.

Another framework that's already been mentioned is Pylons. Pylons is much like TurboGears with ab even stronger emphasis on flexibility, which is bought at the cost of being more difficult to use. Nearly every component can be exchanged, which makes it neccessary to use the documentation of every single component, because there are so many Pylons combinations possible that can satisfy every requirement. Pylons builds upon Paste, an extensive set of tools which are handy for WSGI.

And that's still not everything. The most up-to-date information can always be found in the Python wiki.

**See Also:**

The Python wiki contains an extensive list of web frameworks.

Most frameworks also have their own mailing lists and IRC channels, look out for these on the projects' websites. There is also a general "Python in the Web" IRC channel on freenode called #python.web.

# Index

## P

Python Enhancement Proposals
    PEP 333, 6