Ice Documentation

Release 0.0.1

Susam Pal

Sep 06, 2017

Contents

Ice is a Python module with a WSGI microframework meant for developing small web applications in Python. It is a single file Python module inspired by Bottle.

CHAPTER 1

Why Ice?

This microframework was born as a result of experimenting with WSGI framework. Since what started as a small experiment turned out to be several hundred lines of code, it made sense to share the source code on the web, just in case anyone else benefits from it.

This microframework has a very limited set of features currently. It may be used to develop small web applications. For large web applications, it may make more sense to use a more wholesome framework such as Flask or Django.

It is possible that you may find that this framework is missing a useful API that another major framework provides. In such a case, you have direct access to the WSGI internals to do what you want via the documented API.

If you believe that a missing feature or a bug fix would be useful to others, you may report an issue, or even better, fork this project on GitHub, develop the missing feature or the bug fix, and send a patch or a pull request. In fact, you are very welcome to do so, and turn this experimental project into a matured one by contributing your code and expertise.

CHAPTER 2

Requirements

This module should be used with Python 3.3 or any later version of Python interpreter.

This module depends only on the Python standard library. It does not depend on any third party libraries.

CHAPTER 3

Installation

You can install this module using pip3 using the following command.

```
pip3 install ice
```

You can install this module from source distribution. To do so, download the latest .tar.gz file from https://pypi.python.org/pypi/ice, extract it, then open command prompt or shell, and change your current directory to the directory where you extracted the source distribution, and then execute the following command.

```
python3 setup.py install
```

Note that on a Windows system, you may have to replace `python3` with the path to your Python 3 interpreter.

CHAPTER 4

Resources

Here is a list of useful links about this project.

- Documentation on Read The Docs
- Latest release on PyPI
- Source code on GitHub
- Issue tracker on GitHub
- Changelog on GitHub

CHAPTER 5

Support

To report bugs, suggest improvements, or ask questions, please create a new issue at http://github.com/susam/ice/issues.

CHAPTER 6

License

This is free software. You are permitted to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of it, under the terms of the MIT License. See LICENSE.rst for the complete license.

This software is provided WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See LICENSE.rst for the complete disclaimer.

CHAPTER 7

Tutorial & API

Tutorial

Ice is a Python module with a WSGI microframework meant for developing small web applications in Python. It is a single file Python module inspired by Bottle.

You can install this module using pip3 using the following command.

```
pip3 install ice
```

This module should be used with Python 3.3 or a later version of Python interpreter.

The source code of this module is available at https://github.com/susam/ice.

Getting Started

The simplest way to get started with an ice application is to write a minimal application that serves a default web page.

```
import ice
app = ice.cube()
if __name__ == '__main__':
    app.run()
```

Save the above code in a file and execute it with your Python interpreter. Then open your browser, visit http://localhost:8080/, and you should be able to see a web page that says, 'It works!'.

Routes

Once you are able to run a minimal ice application as mentioned in the previous section, you'll note that while visiting http://localhost:8080/ displays the default 'It works!' page, visiting any other URL, such as http://localhost:8080/foo displays the '404 Not Found' page. This happens because the application object returned by the `ice.cube` function

has a default route defined to invoke a function that returns the default page when the client requests / using the HTTP GET method. There is no such route defined by default for /foo or any request path other than /.

In this document, a request path is defined as the part of the URL after the domain name and before the query string. For example, in a request for http://localhost:8080/foo/bar?x=10, the request path is /foo/bar.

A route is used to map an HTTP request to a Python callable. This callable is also known as the route handler. A route consists of three objects:

1. HTTP request method, e.g. `'GET'`, `'POST'`.
2. Request path pattern, e.g. `'/foo'`, `'/post/<id>'`, `'/(.*)'`.
3. Route handler, a Python callable object, e.g. Python function

A route is said to match a request path when the request pattern of the route matches the request path. When a client makes a request to an ice application, if a route matches the request path, then the route's handler is invoked and the value returned by the route's handler is used to send a response to the client.

The request path pattern of a route can be specified in one of three ways:

1. Literal path, e.g. `'/'`, `'/contact/'`, `'/about/'`.
2. Pattern with wildcards, e.g. `'/blog/<id>'`, `'/order/<:int>'`.
3. Regular expression, e.g. `'/blog/\w+'`, `'/order/\d+'`.

These three types of routes are described in the subsections below.

Literal Routes

The following application overrides the default 'It works!' page for / with a custom page. Additionally, it sets up a route for /foo.

```
import ice
app = ice.cube()

@app.get('/')
def home():
    return ('<!DOCTYPE html>'
            '<html><head><title>Home</title></head>'
            '<body><p>Home</p></body></html>')

@app.get('/foo')
def foo():
    return ('<!DOCTYPE html>'
            '<html><head><title>Foo</title></head>'
            '<body><p>Foo</p></body></html>')

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

The routes defined in the above example are called literal routes because they match the request path exactly as specified in the argument to `app.get` decorator. Routes defined with the `app.get` decorator matches HTTP GET requests. Now, visiting http://localhost:8080/ displays a page with the following text.

> Home

Visiting http://localhost:8080/foo displays a page with the following text.

> Foo

However, visiting http://localhost:8080/foo/ or http://localhost:8080/foo/bar displays the '404 Not Found' page because the literal pattern `'/foo'` does not match the request path `'/foo/'` or `'/foo/bar'`.

Wildcard Routes

Anonymous Wildcards

The following code example is the simplest application demonstrating a wildcard route that matches request path of the form `/` followed by any string devoid of `/`, `<` and `>`. The characters `<>` is an anonymous wildcard because there is no name associated with this wildcard. The part of the request path matched by an anonymous wildcard is passed as a positional argument to the route's handler.

```
import ice
app = ice.cube()

@app.get('/<>')
def foo(a):
    return ('<!DOCTYPE html>'
            '<html><head><title>' + a + '</title></head>'
            '<body><p>' + a + '</p></body></html>')

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

Save the above code in a file and execute it with Python interpreter. Then open your browser, visit http://localhost:8080/foo, and you should be able to see a page with the following text.

> foo

If you visit http://localhost:8080/bar instead, you should see a page with the following text.

> bar

However, visiting http://localhost:8080/foo/ or http://localhost:8080/foo/bar displays the '404 Not Found' page because the wildcard based pattern `/<>` does not match `/foo/` or `/foo/bar`.

Named Wildcards

A wildcard with a valid Python identifier as its name is called a named wildcard. The part of the request path matched by a named wildcard is passed as a keyword argument, with the same name as that of the wildcard, to the route's handler.

```
import ice
app = ice.cube()

@app.get('/<a>')
def foo(a):
    return ('<!DOCTYPE html>'
            '<html><head><title>' + a + '</title></head>'
            '<body><p>' + a + '</p></body></html>')

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

The a, in <a>, is the name of the wildcard. The ice application in this example with a named wildcard behaves similar to the earlier one with an anonymous wildcard. The following example code clearly demonstrates how matches due to anonymous wildcards are passed differently from the matches due to named wildcards.

```
import ice
app = ice.cube()

@app.get('/foo/<>-<>/<a>-<b>/<>-<c>')
def foo(*args, **kwargs):
    return ('<!DOCTYPE html> '
            '<html><head><title>Example</title></head><body> '
            '<p>args: {}<br>kwargs: {}</p> '
            '</body></html>').format(args, kwargs)

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

After running this application, visiting http://localhost:8080/foo/hello-world/ice-cube/wsgi-rocks displays a page with the following text.

> args: ('hello', 'world', 'wsgi')
> kwargs: {'a': 'ice', 'b': 'cube', 'c': 'rocks'}

Here is a more typical example that demonstrates how anonymous wildcard and named wildcard may be used together.

```
import ice
app = ice.cube()

@app.get('/<user>/<category>/<>')
def page(page_id, user, category):
    return ('<!DOCTYPE html>'
            '<html><head><title>Example</title></head><body> '
            '<p>page_id: {}<br>user: {}<br>category: {}</p> '
            '</body></html>').format(page_id, user, category)

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

After running this application, visiting http://localhost:8080/snowman/articles/python displays a page with the following text.

> page_id: python
> user: snowman
> category: articles

Note: Since parts of the request path matched by anonymous wildcards are passed as positional arguments and parts of the request path matched by named wildcards are passed as keyword arguments to the route's handler, it is required by the Python language that all positional arguments must come before all keyword arguments in the function definition. However, the wildcards may appear in any order in the route's pattern.

Throwaway Wildcard

A wildcard with exclamation mark, !, as its name is a throwaway wildcard. The part of the request path matched by a throwaway wildcard is not passed to the route's handler. *They are thrown away!*

```
import ice
app = ice.cube()
```

```
@app.get('/<!>')
def foo(*args, **kwargs):
    return ('<!DOCTYPE html>'
            '<html><head><title>Example</title></head><body>'
            '<p>args: {}<br>kwargs: {}</p>'
            '</body></html>').format(args, kwargs)

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

After running this application, visiting http://localhost:8080/foo displays a page with the following text.

> args: ()
> kwargs: {}

The output confirms that no argument is passed to the `foo` function. Here is a more typical example that demonstrates how a throwaway wildcard may be used with other wildcards.

```
import ice
app = ice.cube()

@app.get('/<!>/<!>/<>')
def page(page_id):
    return ('<!DOCTYPE html>'
            '<html><head><title>Example</title></head><body>'
            '<p>page_id: ' + page_id + '</p>'
            '</body></html>')

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

After running this application, visiting http://localhost:8080/snowman/articles/python displays a page with the following text.

> page_id: python

There are three wildcards in the route's request path pattern but there is only one argument in the route's handler because two out of the three wildcards are throwaway wildcards.

Wildcard Specification

The complete syntax of a wildcard specification is: *<name:type>*.

The following rules describe how a wildcard is interpreted.

1. The delimiters < (less-than sign) and > (greater-than sign), are mandatory.
2. However, *name*, `:` (colon) and *type* are optional.
3. Either a valid Python identifier or the exclamation mark, `!`, must be specified as *name*.
4. If *name* is missing, the part of the request path matched by the wildcard is passed as a positional argument to the route's handler.
5. If *name* is present and it is a valid Python identifier, the part of the request path matched by the wildcard is passed as a keyword argument to the route's handler.
6. If *name* is present and it is `!`, the part of the request path matched by the wildcard is not passed to the route's handler.

7. If *name* is present but it is neither `!` nor a valid Python identifier, ice.RouteError is raised.
8. If *type* is present, it must be preceded by `:` (colon).
9. If *type* is present but it is not `str`, `path`, `int`, `+int` and `-int`, ice.RouteError is raised.
10. If *type* is missing, it is assumed to be `str`.
11. If *type* is `str`, it matches a string of one or more characters such that none of the characters is `/`. The path of the request path matched by the wildcard is passed as an `str` object to the route's handler.
12. If *type* is `path`, it matches a string of one or more characters that may contain `/`. The path of the request path matched by the wildcard is passed as an `str` object to the route's handler.
13. If *type* is `int`, `+int` or `-int`, the path of the request path matched by the wildcard is passed as an `int` object to the route's handler.
14. If *type* is `+int`, the wildcard matches a positive integer beginning with a non-zero digit.
15. If *type* is `int`, the wildcard matches `0` as well as everything that a wildcard of type `+int` matches.
16. If *type* is `-int`, the wildcard matches a negative integer that begins with the − sign followed by a non-zero digit as well as everything that a wildcard of type `int` matches.

Here is an example that demonstrates a typical route with `path` and `int` wildcards.

```
import ice
app = ice.cube()

@app.get('/notes/<:path>/<:int>')
def note(note_path, note_id):
    return ('<!DOCTYPE html>'
            '<html><head><title>Example</title></head><body>'
            '<p>note_path: {}<br>note_id: {}</p>'
            '</body></html>').format(note_path, note_id)

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

After running this application, visiting http://localhost:8080/notes/tech/python/12 displays a page with the following text.

> note_path: tech/python
> note_id: 12

Visiting http://localhost:8080/notes/tech/python/0 displays a page with the following text.

> note_path: tech/python
> note_id: 0

However, visiting http://localhost:8080/notes/tech/python/+12 http://localhost:8080/notes/tech/python/+0 or http://localhost:8080/notes/tech/python/012, displays the '404 Not Found' page because `<:int>` does not match an integer with a leading `+` sign or with a leading `0`. It matches `0` and a positive integer beginning with a non-zero digit only.

Regular Expression Routes

The following code demonstrates a simple regular expression based route. The part of the request path matched by a non-symbolic capturing group is passed as a positional argument to the route's handler.

```
import ice
app = ice.cube()

@app.get('/(.*)')
def foo(a):
    return ('<!DOCTYPE html>'
            '<html><head><title>' + a + '</title></head>'
            '<body><p>' + a + '</p></body></html>')

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

After running this application, visiting http://localhost:8080/foo displays a page with the following text.

> foo

Visiting http://localhost:8080/foo/bar/ displays a page with the following text.

> foo/bar/

The part of the request path matched by a symbolic capturing group in the regular expression is passed as a keyword argument with the same name as that of the symbolic group.

```
import ice
app = ice.cube()

@app.get('/(?P<user>[^/]*)/(?P<category>[^/]*)/([^/]*)')
def page(page_id, user, category):
    return ('<!DOCTYPE html>'
            '<html><head><title>Example</title></head><body>'
            '<p>page_id: {}<br>user: {}<br>category: {}</p>'
            '</body></html>').format(page_id, user, category)

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

After running this application, visiting http://localhost:8080/snowman/articles/python displays a page with the following text.

> page_id: python
> user: snowman
> category: articles

Note: Since parts of the request path matched by non-symbolic capturing groups are passed as positional arguments and parts of the request path matched by symbolic capturing groups are passed as keyword arguments to the route's handler, it is required by the Python language that all positional arguments must come before all keyword arguments in the function definition. However, the capturing groups may appear in any order in the route's pattern.

Interpretation of Request Path Pattern

The request path pattern is interpreted according to the following rules. The rules are processed in the order specified and as soon as one of the rules succeeds in determining how the request path pattern should be interpreted, further rules are not processed.

1. If a route's request path pattern begins with `regex:` prefix, then it is interpreted as a regular expression route.
2. If a route's request path pattern begins with `wildcard:` prefix, then it is interpreted as a wildcard route.
3. If a route's request path pattern begins with `literal:` prefix, then it is interpreted as a literal route.

4. If a route's request path pattern contains what looks like a regular expression capturing group, i.e. it contains `(` before `)` somewhere in the pattern, then it is automatically interpreted as a regular expression route.
5. If a route's request path pattern contains what looks like a wildcard, i.e. it contains < before > somewhere in the pattern with no `/`, < and > in between them, then it is automatically interpreted as a wildcard route.
6. If none of the above rules succeed in determining how to interpret the request path, then it is interpreted as a literal route.

The next three sections clarify the above rules with some contrived examples.

Explicit Literal Routes

To define a literal route with the request path pattern as `/<foo>`, `literal:` prefix must be used. Without it, the `<foo>` in the pattern is interpreted as a wildcard and the route is defined as a wildcard route. With the `literal:` prefix, the pattern is explicitly defined as a literal pattern.

```
import ice
app = ice.cube()

@app.get('literal:/<foo>')
def foo():
    return ('<!DOCTYPE html>'
            '<html><head><title>Foo</title></head>'
            '<body><p>Foo</p></body></html>')

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

After running this application, visiting http://localhost:8080/%3Cfoo%3E displays a page containing the following text.

> Foo

A request path pattern that seems to contain a wildcard or a capturing group but needs to be treated as a literal pattern must be prefixed with the string `literal:`.

Explicit Wildcard Routes

To define a wildcard route with the request path pattern as `/(foo)/<>`, the `wildcard:` prefix must be used. Without it, the pattern is interpreted as a regular expression pattern because the `(foo)` in the pattern looks like a regular expression capturing group.

```
import ice
app = ice.cube()

@app.get('wildcard:/(foo)/<>')
def foo(a):
    return ('<!DOCTYPE html>'
            '<html><head><title>Foo</title></head>'
            '<body><p>a: ' + a + '</p></body></html>')

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

After running this application, visiting http://localhost:8080/(foo)/bar displays a page with the following text.

a: bar

A request path pattern that seems to contain a regular expression capturing group but needs to be treated as a wildcard pattern must be prefixed with the string `wildcard:`.

Explicit Regular Expression Routes

To define a regular expression route with the request path pattern as `^/foo\d*$`, the `regex:` prefix must be used. Without it, the pattern is interpreted as a literal pattern because there is no capturing group in the pattern.

```
import ice
app = ice.cube()

@app.get('regex:/foo\d*$')
def foo():
    return ('<!DOCTYPE html>'
            '<html><head><title>Foo</title></head>'
            '<body><p>Foo</p></body></html>')

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

After running this application, visiting http://localhost:8080/foo or http://localhost:8080/foo123 displays a page containing the following text.

Foo

A request path pattern that does not contain a regular expression capturing group but needs to be treated as a regular expression pattern must be prefixed with the string `regex:`.

Query Strings

The following example shows an application that can process a query string in a GET request.

```
import ice
app = ice.cube()

@app.get('/')
def home():
    return ('<!DOCTYPE html>'
            '<html><head><title>Foo</title></head>'
            '<body><p>name: {}</p></body>'
            '</html>').format(app.request.query['name'])

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

After running this application, visiting http://localhost:8080/?name=Humpty+Dumpty displays a page with the following text.

name: Humpty Dumpty

Note that the + sign in the query string has been properly URL decoded into a space.

The `app.request.query` object in the code is an `ice.MultiDict` object that can store multiple values for every key. However, when used like a dictionary, it returns the most recently added value for a key. Therefore, visiting http://localhost:8080/?name=Humpty&name=Santa displays a page with the following text.

name: Santa

Note that in this URL, there are two values passed for the `name` field in the query string, but accessing `app.request.query['name']` provides us only the value that is most recently added. To get all the values for a key in `app.request.query`, we can use the `ice.MultiDict.getall` method as shown below.

```
import ice
app = ice.cube()

@app.get('/')
def home():
    return ('<!DOCTYPE html>'
            '<html><head><title>Foo</title></head>'
            '<body><p>name: {}</p></body>'
            '</html>').format(app.request.query.getall('name'))

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

Now, visiting http://localhost:8080/?name=Humpty&name=Santa displays a page with the following text.

name: ['Humpty', 'Santa']

Note that the `ice.MultiDict.getall` method returns all the values belonging to the key as a `list` object.

Forms

The following example shows an application that can process forms submitted by a POST request.

```
import ice
app = ice.cube()

@app.get('/')
def show_form():
    return ('<!DOCTYPE html>'
            '<html><head><title>Foo</title></head>'
            '<body><form action="/result" method="post">'
            'First name: <input name="firstName"><br>'
            'Last name: <input name="lastName"><br>'
            '<input type="submit">'
            '</form></body></html>')

@app.post('/result')
def show_post():
    return ('<!DOCTYPE html>'
            '<html><head><title>Foo</title></head><body>'
            '<p>First name: {}<br>Last name: {}</p>'
            '</body></html>').format(app.request.form['firstName'],
                                     app.request.form['lastName'])

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

After running this application, visiting http://localhost:8080/, filling up the form and submitting it displays the form data.

The `app.request.form` object in this code, like the `app.request.query` object in the previous section, is a MultiDict object.

```
import ice
app = ice.cube()

@app.get('/')
def show_form():
    return ('<!DOCTYPE html>'
            '<html><head><title>Foo</title></head>'
            '<body><form action="/result" method="post">'
            'name1: <input name="name"><br>'
            'name2: <input name="name"><br>'
            '<input type="submit">'
            '</form></body></html>')

@app.post('/result')
def show_post():
    return ('<!DOCTYPE html>'
            '<html><head><title>Foo</title></head><body>'
            '<p>name (single): {}<br>name (multi): {}</p>'
            '</body></html>').format(app.request.form['name'],
                                     app.request.form.getall('name'))

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

After running this application, visiting http://localhost:8080/, filling up the form and submitting it displays the form data. While `app.request.form['name']` returns the string entered in the second input field, `app.request.form.getall('name')` returns strings entered in both input fields as a list object.

Cookies

The following example shows an application that can read and set cookies.

```
import ice
app = ice.cube()

@app.get('/')
def show_count():
    count = int(app.request.cookies.get('count', 0)) + 1
    app.response.set_cookie('count', str(count))
    return ('<!DOCTYPE html>'
            '<html><head><title>Foo</title></head><body>'
            '<p>Count: {}</p></body></html>'.format(count))

app.run()
```

The `app.request.cookies` object in this code, like the `app.request.query` object in a previous section, is a MultiDict object. Every cookie name and value sent by the client to the application found in the HTTP Cookie header is available in this object as key value pairs.

The `app.response.set_cookie` method is used to set cookies to be sent from the application to the client.

Error Pages

The application object returned by the `ice.cube` function contains a generic fallback error handler that returns a simple error page with the HTTP status line, a short description of the status and the version of the ice module.

This error handler may be overridden using the `error` decorator. This decorator accepts one optional integer argument that may be used to explicitly specify the HTTP status code of responses for which the handler should be invoked to generate an error page. If no argument is provided, the error handler is defined as a fallback error handler. A fallback error handler is invoked to generate an error page for any HTTP response representing an error when there is no error handler defined explicitly for the response status code of the HTTP response.

Here is an example.

```
import ice
app = ice.cube()

@app.error(404)
def error():
    return ('<!DOCTYPE html>'
            '<html><head><title>Page not found</title></head>'
            '<body><p>Page not found</p></body></html>')

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

After running this application, visiting http://localhost:8080/foo displays a page with the following text.

> Page not found

Status Codes

In all the examples above, the response message body is returned as a string from a route's handler. It is also possible to return the response status code as an integer. In other words, a route's handler must either return a string or an integer. When a string is returned, it is sent as response message body to the client. When an integer is returned and it is a valid HTTP status code, an HTTP response with this status code is sent to the client. If the value returned by a route's handler is neither a string nor an integer representing a valid HTTP status code, then an error is raised.

Therefore there are two ways to return an HTTP response from a route's handler.

1. Return message body and optionally set status code. This is the preferred way of returning content for normal HTTP responses (200 OK). If the status code is not set explicitly in a route's handler, then it has a default value of 200.
2. Return status code and optionally set message body. This is the preferred way of returning content for HTTP errors. If the message body is not set explicitly in a route's handler, then the error handler for the returned status code is invoked to return a message body.

Here is an example where status code is set to 403 and a custom error page is returned.

```
import ice

app = ice.cube()

@app.get('/foo')
def foo():
    app.response.status = 403
    return ('<!DOCTYPE html>'
            '<html><head><title>Access is forbidden</title></head>'
            '<body><p>Access is forbidden</p></body></html>')

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

After running this application, visiting http://localhost:8080/foo displays a page with the following text.

Access is forbidden

Here is another way of writing the above application. In this case, the message body is set and the status code is returned.

```
import ice

app = ice.cube()

@app.get('/foo')
def foo():
    app.response.body = ('<!DOCTYPE html>'
            '<html><head><title>Access is forbidden</title></head>'
            '<body><p>Access is forbidden</p></body></html>')
    return 403

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

Although the above way of setting message body works, using an error handler is the preferred way of defining the message body for an HTTP error. Here is an example that demonstrates this.

```
import ice

app = ice.cube()

@app.get('/foo')
def foo():
    return 403

@app.error(403)
def error403():
    return ('<!DOCTYPE html>'
            '<html><head><title>Access is forbidden</title></head>'
            '<body><p>Access is forbidden</p></body></html>')

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

For simple web applications, just returning the status code is sufficient. When neither a message body is defined nor an error handler is defined, a generic fallback error handler set in the application object returned by the `ice.cube` is used to return a simple error page with the HTTP status line, a short description of the status and the version of the ice module.

```
import ice

app = ice.cube()

@app.get('/foo')
def foo():
    return 403

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

After running this application, visiting http://localhost:8080/foo displays a page with the following text.

403 Forbidden
Request forbidden – authorization will not help

Redirects

Here is an example that demonstrates how to redirect a client to a different URL.

```
import ice
app = ice.cube()

@app.get('/foo')
def foo():
    return 303, '/bar'

@app.get('/bar')
def bar():
    return ('<!DOCTYPE html>'
            '<html><head><title>Bar</title></head>'
            '<body><p>Bar</p></body></html>')

app.run()
```

After running this application, visiting http://localhost:8080/foo with a browser redirects the browser to http://localhost:8080/bar and displays a page with the following text.

> Bar

To send a redirect, the route handler needs to return a tuple such that the first item in the tuple is an HTTP status code for redirection and the second item is the URL to which the client should be redirected to.

The behaviour of the above code is equivalent to the following code.

```
import ice
app = ice.cube()

@app.get('/foo')
def foo():
    app.response.add_header('Location', '/bar')
    return 303

@app.get('/bar')
def bar():
    return ('<!DOCTYPE html>'
            '<html><head><title>Bar</title></head>'
            '<body><p>Bar</p></body></html>')

app.run()
```

Much of the discussion in the *Status Codes* section applies to this section too, i.e. it is possible to set the status code in `app.response.status`, add a Location header and return a message body, or add a Location header, set the message body in `app.response.body` and return a status code. However, returning a tuple of redirection status code and URL, as shown in the first example in this section, is the simplest and preferred way to send a redirect.

Static Files

In a typical production environment, a web server may be configured to receive HTTP requests and forward it to a Python application via WSGI. In such a setup, it might make more sense to configure the web server to serve static files because web servers implement several standard file handling capabilities and response headers, e.g. 'Last-Modified', 'If-Modified-Since', etc. However, it is possible to serve static files from an ice application using *ice.Ice.static()* that provides a very rudimentary means of serving static files. This could be useful in a development

environment where one would want to test pages with static content such as style sheets, images, etc. served by an ice application without using a web server.

`Ice.`**`static`**(*root*, *path*, *media_type=None*, *charset='UTF-8'*)

Send content of a static file as response.

The path to the document root directory should be specified as the root argument. This is very important to prevent directory traversal attack. This method guarantees that only files within the document root directory are served and no files outside this directory can be accessed by a client.

The path to the actual file to be returned should be specified as the path argument. This path must be relative to the document directory.

The *media_type* and *charset* arguments are used to set the Content-Type header of the HTTP response. If *media_type* is not specified or specified as `None` (the default), then it is guessed from the filename of the file to be returned.

Parameters

- **`root`** (*`str`*) – Path to document root directory.
- **`path`** (*`str`*) – Path to file relative to document root directory.
- **`media_type`** (*`str, optional`*) – Media type of file.
- **`charset`** (*`str, optional`*) – Character set of file.

Returns Content of file to be returned in the HTTP response.

Return type bytes

Here is an example.

```
import ice
app = ice.cube()

@app.get('/code/<:path>')
def send_code(path):
    return app.static('/var/www/project/code', path)

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

If there is a file called /var/www/project/code/data/foo.txt, then visiting http://localhost:8080/code/data/foo.txt would return the content of this file as response.

However, visiting http://localhost:8080/code/%2e%2e/foo.txt would display a '403 Forbidden' page because this request attempts to access foo.txt in the parent directory of the document root directory (`%2e%2d` is the URL encoding of `..`). This is not allowed in order to prevent directory traversal attack.

In the above example, the 'Content-Type' header of the response is automatically set to 'text/plain; charset=UTF-8'. With only two arguments specified to this method, it uses the extension name of the file being returned to automatically guess the media type to be used in the 'Content-Type' header. For example, the media type of a .txt file is typically *guessed* to be 'text/plain'. But this may be different because system configuration files may be referred in order to guess the media type and such configuration files may map a .txt file to a different media type.

For example, on a Debian 8.0 system, /etc/mime.types maps a .c file to 'text/x-csrc'. This is one of the files that is referred to guess the media type. Therefore, the 'Content-Type' header for a request to http://localhost:8080/code/data/foo.c would be set to 'text/x-csrc; charset=UTF-8' on such a system.

To see the list of files that may be referred to guess media type, execute this command.

```
python3 -c "import mimetypes; print(mimetypes.knownfiles)"
```

The media type of static file being returned in a response can be set explicitly to a desired value using the `media_type` keyword argument.

The charset defaults to 'UTF-8' for any media type of type 'text' regardless of the subtype. This may be changed with the `charset` keyword argument.

```
import ice
app = ice.cube()

@app.get('/code/<:path>')
def send_code(path):
    return app.static('/var/www/project/code', path,
                      media_type='text/plain', charset='ISO-8859-1')

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

The above code guarantees that the 'Content-Type' header of a request to http://localhost:8080/code/data/foo.c is set to 'text/plain; charset=ISO-8859-1' regardless of how the media type of a .c file is defined in the system configuration files.

Downloads

The *ice.Ice.download()* method may be used to force a client, e.g. a browser, to prompt the user to save the returned content locally as a file.

Ice.**download**(*content*, *filename=None*, *media_type=None*, *charset='UTF-8'*)

Send content as attachment (downloadable file).

The *content* is sent after setting Content-Disposition header such that the client prompts the user to save the content locally as a file. An HTTP response status code may be specified as *content*. If the status code is not `200`, then this method does nothing and returns the status code.

The filename used for the download is determined according to the following rules. The rules are followed in the specified order.

1. If *filename* is specified, then the base name from this argument, i.e. `os.path.basename(filename)`, is used as the filename for the download.
2. If *filename* is not specified or specified as `None` (the default), then the base name from the file path specified to a previous `static()` call made while handling the current request is used.
3. If *filename* is not specified and there was no `static()` call made previously for the current request, then the base name from the current HTTP request path is used.
4. As a result of the above steps, if the resultant *filename* turns out to be empty, then *ice.LogicError* is raised.

The *media_type* and *charset* arguments are used in the same manner as they are used in `static()`.

Parameters

- **content** (*str, bytes or int*) – Content to be sent as download or HTTP status code of the response to be returned.
- **filename** (*str*) – Filename to use for saving the content
- **media_type** (*str, optional*) – Media type of file.

- **charset** (*str, optional*) – Character set of file.

Returns content, i.e. the first argument passed to this method.

Raises `LogicError` – When filename cannot be determined.

Here is an example.

```
import ice
app = ice.cube()

@app.get('/foo')
def foo():
    return app.download('hello, world', 'foo.txt')

@app.get('/bar')
def bar():
    return app.download('hello, world', 'bar',
                        media_type='text/plain', charset='ISO-8859-1')

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

The first argument to this method is the content to return, specified as a string or sequence of bytes. The second argument is the filename that the client should use to save the returned content.

The discussion about media type and character set described in the *Static Files* applies to this section too.

Visiting http://localhost:8080/foo with a standard browser displays a prompt to download and save a file called foo.txt. Visiting http://localhost:8080/bar displays a prompt to download and save a file called bar.

Since the first argument may be a sequence of bytes, it is quite simple to return a static file for download. The *ice.Ice.static()* method usually returns a sequence of bytes which can be passed directly to the *ice.Ice.download()* method. The `static()` method may return an HTTP status code, e.g. 403 or 404, which is handled gracefully by the `download()` method in order to return an error page as response.

```
import ice
app = ice.cube()

@app.get('/code/<:path>')
def send_download(path):
    return app.download(app.static('/var/www/project/code', path))

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

Note that in the above example, no filename argument is specified for the `download()` method. The path argument that was specified in the `static()` call is automatically used to obtain the filename for the `download()` call.

If there is a file called /var/www/project/code/data/foo.txt, then visiting http://localhost:8080/code/data/foo.txt with a standard browser displays a prompt to download and save a file called foo.txt.

Here are the complete set of rules that determine the filename that is used for the download. The rules are followed in the specified order.

1. If the *filename* argument is specified, the base name from this argument, i.e. `os.path.basename(filename)`, is used as the filename for the download.
2. If the *filename* argument is not specified, the base name from the file path specified to a previous *static()* method call made while handling the current request is used.

3. If the *filename* argument is not specified and there was no `static()` call made previously for the current request, then the base name from the current HTTP request path is used.
4. As a result of the above three steps, if the resultant *filename* turns out to be empty, then ice.LogicError is raised.

The first two points have been demonstrated in the previous two examples above. The last two points are demonstrated in the following example.

```
import ice
app = ice.cube()

@app.get('/<!:path>')
def send_download():
    return app.download('hello, world')

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

Visiting http://localhost:8080/foo.txt with a standard browser would download a file foo.txt. However, visiting http://localhost:8080/foo/ would display an error due to the unhandled ice.LogicError that is raised because no filename can be determined from the request path /foo/ which refers to a directory, not a file.

Request Environ

The following example shows how to access the `environ` dictionary defined in the WSGI specification.

```
import ice
app = ice.cube()

@app.get('/')
def foo():
    user_agent = app.request.environ.get('HTTP_USER_AGENT', None)
    return ('<!DOCTYPE html>'
            '<html><head><title>User Agent</title></head>'
            '<body><p>{}</p></body></html>'.format(user_agent))

app.run()
```

The `environ` dictionary specified in the WSGI specification is made available in `app.request.environ`. The above example retrieves the HTTP User-Agent header from this dictionary and displays it to the client.

More

Since this is a microframework with a very limited set of features, it is possible that you may find from time to time that this framework is missing a useful API that another major framework provides. In such a case, you have direct access to the WSGI internals to do what you want via the documented API (see *API Documentation*).

If you believe that the missing feature would be useful to all users of this framework, please feel free to send a patch or a pull request.

API Documentation

ice module

Ice - WSGI on the rocks.

Ice is a simple and tiny WSGI microframework meant for developing small Python web applications.

exception `ice.`**`Error`**
Bases: `Exception`

Base class for exceptions.

class `ice.`**`Ice`**
Bases: `object`

A single WSGI application.

Each instance of this class is a single, distinct callable object that functions as WSGI application.

`download`(*content*, *filename=None*, *media_type=None*, *charset='UTF-8'*)
Send content as attachment (downloadable file).

The *content* is sent after setting Content-Disposition header such that the client prompts the user to save the content locally as a file. An HTTP response status code may be specified as *content*. If the status code is not `200`, then this method does nothing and returns the status code.

The filename used for the download is determined according to the following rules. The rules are followed in the specified order.

1. If *filename* is specified, then the base name from this argument, i.e. `os.path.basename(filename)`, is used as the filename for the download.
2. If *filename* is not specified or specified as `None` (the default), then the base name from the file path specified to a previous *`static()`* call made while handling the current request is used.
3. If *filename* is not specified and there was no *`static()`* call made previously for the current request, then the base name from the current HTTP request path is used.
4. As a result of the above steps, if the resultant *filename* turns out to be empty, then *`ice.LogicError`* is raised.

The *media_type* and *charset* arguments are used in the same manner as they are used in *`static()`*.

Parameters

- **`content`** (*`str, bytes or int`*) – Content to be sent as download or HTTP status code of the response to be returned.
- **`filename`** (*`str`*) – Filename to use for saving the content
- **`media_type`** (*`str, optional`*) – Media type of file.
- **`charset`** (*`str, optional`*) – Character set of file.

Returns content, i.e. the first argument passed to this method.

Raises *`LogicError`* – When filename cannot be determined.

`error`(*status=None*)
Decorator to add a callback that generates error page.

The *status* parameter specifies the HTTP response status code for which the decorated callback should be invoked. If the *status* argument is not specified, then the decorated callable is considered to be a fallback callback.

A fallback callback, when defined, is invoked to generate the error page for any HTTP response representing an error when there is no error handler defined explicitly for the response code of the HTTP response.

Parameters **status** (*int, optional*) – HTTP response status code.

Returns Decorator function to add error handler.

Return type function

exit()
Stop the simple WSGI server running the appliation.

get(*pattern*)
Decorator to add route for an HTTP GET request.

Parameters **pattern** (*str*) – Routing pattern the path must match.

Returns Decorator to add route for HTTP GET request.

Return type function

post(*pattern*)
Decorator to add route for an HTTP POST request.

Parameters **pattern** (*str*) – Routing pattern the path must match.

Returns Decorator to add route for HTTP POST request.

Return type function

route(*method*, *pattern*)
Decorator to add route for a request with any HTTP method.

Parameters

- **method** (*str*) – HTTP method name, e.g. GET, POST, etc.
- **pattern** (*str*) – Routing pattern the path must match.

Returns Decorator function to add route.

Return type function

run(*host='127.0.0.1'*, *port=8080*)
Run the application using a simple WSGI server.

Parameters

- **host** (*str, optional*) – Host on which to listen.
- **port** (*int, optional*) – Port number on which to listen.

running()
Return `True` iff simple WSGI server is running.

Returns `True` if simple WSGI server associated with this application is running, `False` otherwise.

Return type bool

static(*root*, *path*, *media_type=None*, *charset='UTF-8'*)
Send content of a static file as response.

The path to the document root directory should be specified as the root argument. This is very important to prevent directory traversal attack. This method guarantees that only files within the document root directory are served and no files outside this directory can be accessed by a client.

The path to the actual file to be returned should be specified as the path argument. This path must be relative to the document directory.

The *media_type* and *charset* arguments are used to set the Content-Type header of the HTTP response. If *media_type* is not specified or specified as `None` (the default), then it is guessed from the filename of the file to be returned.

Parameters

- **root** (*str*) – Path to document root directory.
- **path** (*str*) – Path to file relative to document root directory.
- **media_type** (*str, optional*) – Media type of file.
- **charset** (*str, optional*) – Character set of file.

Returns Content of file to be returned in the HTTP response.

Return type bytes

exception ice.**LogicError**
Bases: *ice.Error*

Logical error that can be avoided by careful coding.

class ice.**MultiDict**(**args*, ***kwargs*)
Bases: collections.UserDict

Dictionary with multiple values for a key.

Setting an existing key to a new value merely adds the value to the list of values for the key. Getting the value of an existing key returns the newest value set for the key.

getall(*key*, *default=[]*)
Return the list of all values for the specified key.

Parameters

- **key** (*object*) – Key
- **default** (*list*) – Default value to return if the key does not exist, defaults to `[]`, i.e. an empty list.

Returns List of all values for the specified key if the key exists, `default` otherwise.

Return type list

class ice.**RegexRoute**(*pattern*, *callback*)
Bases: object

A regular expression pattern.

static like(*pattern*)
Determine if a pattern looks like a regular expression.

Parameters **pattern** (*str*) – Any route pattern.

Returns `True` if the specified pattern looks like a regex, `False` otherwise.

match(*path*)
Return route handler with arguments if path matches this route.

Parameters **path** (*str*) – Request path

Returns

A tuple of three items:

1. Route handler (callable)
2. Positional arguments (list)
3. Keyword arguments (dict)

`None` if the route does not match the path.

Return type tuple or None

class `ice.`**Request**(*environ*)

Bases: `object`

Current request.

environ
dict – Dictionary of request environment variables.

method
str – Request method.

path
str – Request path.

query
MultiDict – Key-value pairs from query string.

form
MultiDict – Key-value pairs from form data in POST request.

cookies
MultiDict – Key-value pairs from cookie string.

class `ice.`**Response**(*start_response_callable*)

Bases: `object`

Current response.

start
callable – Callable that starts response.

status
int – HTTP response status code, defaults to 200.

media_type
str – Media type of HTTP response, defaults to 'text/html'. This together with *charset* determines the Content-Type response header.

charset
str – Character set of HTTP response, defaults to 'UTF-8'. This together with *media_type* determines the Content-Type response header.

body
str or bytes – HTTP response body.

add_header(*name*, *value*)

Add an HTTP header to response object.

Parameters

- **name** (*str*) – HTTP header field name
- **value** (*str*) – HTTP header field value

content_type
Return the value of Content-Type header field.

The value for the Content-Type header field is determined from the *media_type* and *charset* data attributes.

Returns Value of Content-Type header field

Return type str

response()
Return the HTTP response body.

Returns HTTP response body as a sequence of bytes

Return type bytes

set_cookie(*name*, *value*, *attrs={}*)
Add a Set-Cookie header to response object.

For a description about cookie attribute values, see https://docs.python.org/3/library/http.cookies.html#http.cookies.Morsel.

Parameters

- **name** (*str*) – Name of the cookie
- **value** (*str*) – Value of the cookie
- **attrs** (*dict*) – Dicitionary with cookie attribute keys and values.

status_detail
Return a description of the current HTTP response status.

Returns Response status description

Return type str

status_line
Return the HTTP response status line.

The status line is determined from *status* code. For example, if the status code is 200, then '200 OK' is returned.

Returns Status line

Return type str

exception ice.**RouteError**
Bases: *ice.Error*

Route related exception.

class ice.**Router**
Bases: object

Route management and resolution.

add(*method*, *pattern*, *callback*)
Add a route.

Parameters

- **method** (*str*) – HTTP method, e.g. GET, POST, etc.

- **pattern** (*str*) – Pattern that request paths must match.
- **callback** (*str*) – Route handler that is invoked when a request path matches the *pattern*.

contains_method(*method*)
Check if there is at least one handler for *method*.

Parameters **method** (*str*) – HTTP method name, e.g. GET, POST, etc.

Returns `True` if there is at least one route defined for *method*, `False` otherwise

resolve(*method*, *path*)
Resolve a request to a route handler.

Parameters

- **method** (*str*) – HTTP method, e.g. GET, POST, etc. (type: str)
- **path** (*str*) – Request path

Returns

A tuple of three items:

1. Route handler (callable)
2. Positional arguments (list)
3. Keyword arguments (dict)

`None` if no route matches the request.

Return type tuple or None

class ice.**Wildcard**(*spec*)
Bases: `object`

A single wildcard definition in a wildcard route pattern.

regex()
Convert the wildcard to a regular expression.

Returns A regular expression that matches strings that the wildcard is meant to match.

Return type str

value(*value*)
Convert specified value to a value of wildcard type.

This method does not check if the value matches the wildcard type. The caller of this method must ensure that the value passed to this method was obtained from a match by regular expression returned by the regex method of this class. Ensuring this guarantees that the value passed to the method matches the wildcard type.

Parameters **value** (*str*) – Value to convert.

Returns Converted value.

Return type str or int

class ice.**WildcardRoute**(*pattern*, *callback*)
Bases: `object`

Route containing wildcards to match request path.

static like(*pattern*)
Determine if a pattern looks like a wildcard pattern.

Parameters **pattern** (*str*) – Any route pattern.

Returns `True` if the specified pattern looks like a wildcard pattern, `False` otherwise.

match(*path*)
Return route handler with arguments if path matches this route.

Parameters **path** (*str*) – Request path

Returns

A tuple of three items:

1. Route handler (callable)
2. Positional arguments (list)
3. Keyword arguments (dict)

`None` if the route does not match the path.

Return type tuple or None

static tokens(*pattern*)
Return tokens in a pattern.

ice.**cube**()
Return an Ice application with a default home page.

Create *Ice* object, add a route to return the default page when a client requests the server root, i.e. /, using HTTP GET method, add an error handler to return HTTP error pages when an error occurs and return this object. The returned object can be used as a WSGI application.

Returns WSGI application.

Return type *Ice*

CHAPTER 8

Indices

- genindex
- modindex
- search

Python Module Index

i

Index

V

W