

itools

Juan David Ibáñez Palomar¹

`jdavid@itaapy.com`

May 3, 2004

¹Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. There is a copy of the license at <http://www.gnu.org/copyleft/fdl.html>

Contents

Introduction	vii
Background	vii
Installation	viii
Run the demo	viii
1 Uniform Resource Identifiers	1
1.1 Syntax	2
1.2 References	2
2 Resources	5
2.1 Files and Folders	6
2.2 Other schemes	6
2.2.1 http	6
2.3 API	7
3 Resource Handlers	9
3.1 Files and Folders	10
3.2 Handler databases	10
4 eXtensible Markup Language	13
5 Simple Template Language	15
5.1 The namespace	15
5.2 Repeat	16
5.2.1 The Python side	16

List of Tables

3.1 Database API	11
----------------------------	----

Introduction

This document describes and explains the *itools* package, which enhances the Python standard library with modules as a workflow engine or just the simplest and fastest template language available.

What **itools** pretends is to redefine the state of the art in the development of web applications with the Python programming language, and this document wants to help developers to learn about it.

Key goals of **itools** include to make a very easy to learn framework, where new programmers can be added to the development team in a record time. Improved productivity, as a consequence of its simplicity; not only for beginners, but also for experienced programmers; with a focus on scalability: as the project gets bigger the framework stands. And high modularity, so you can use just the modules you like and combine them with other solutions; this also makes for an stepped learning curve.

The exposition will be driven by useful examples. We will see some simple scripts at the beginning; but soon after that we will start to build a web based addressbook, whose development will drive the rest of the exposition.

Background

A web application is more complex than it looks like at a first glance, as it involves many different technologies. There are always two sides, the server and the client, which communicate through the HTTP protocol. The client builds and sends a request to the server, then the server builds and sends back a response.

To read this book it is required to have a minimum understanding of the web, the HTTP protocol, and its languages. In the client side there is the HTML markup language, in the server side we will use the Python¹ programming language. The client will be a web browser. In the server side you need to be able to setup a server (e.g. Apache) with CGI (Common Gateway Interface) support.

Also, a minimum knowledge about XML is needed, for example it is important to know what is a well formed XML document.

¹To learn Python start with the tutorial, <http://www.python.org/.../html/tut/tut.html>.

Installation

Before going further, be sure your system is correctly setup. The `itools` package requires Python 2.3 or later, and for now it only works on Unix systems (we are working to support other platforms).

Download the last version from <http://sf.net/projects/1leu>, then unpack it somewhere. The `itools` software is a Python package, so it uses *distutils*, to install it just type:

```
python setup.py install
```

Be sure to have the right permissions.

Run the demo

To run the demo a web server with CGI support enabled is required. In this section we suppose the web server is Apache and the GNU/Linux distribution is Debian; this means that the CGI directory is `/usr/lib/cgi-bin`².

The demo must be accessible from the CGI directory, you have two options, either copy the whole demo:

```
cd /usr/lib/cgi-bin
sudo cp -r <itools source directory>/demo itools
```

Or create a symbolic link³:

```
cd /usr/lib/cgi-bin
sudo ln -s <itools source directory>/demo itools
```

Now go to the URL <http://localhost/cgi-bin/itools/hello.cgi>. If everything was right you will see the *hello world* message in your web browser. The `demo` directory contains many scripts of increasing complexity that we will study through this book.

²In Gentoo it is `/var/www/localhost/cgi-bin`

³In this case you must be sure Apache has the option `SymLinksIfOwnerMatch`.

Chapter 1

Uniform Resource Identifiers

Uniform **R**esource **I**dentifiers are a standard defined by the **RFC2396**¹. They provide means to identify and locate resources in the internet.

The problem

Some programming languages provide ways to manipulate URIs, for example Rebol has an `url` data type, Perl XXX. Unfortunately Python's support of URIs, provided by the `urlparse` module, is not as good.

The solution

This chapter presents the `itools.uri` module, which provides an Object Oriented and standards compliant API to manage URIs.

Show me the code

What `itools.uri` works with are references, which can be absolute URIs, relative paths, etc. To get a reference from a string the factory `get_reference` is provided:

```
>>> from itools import uri
>>> ref = uri.get_reference('http://www.ietf.org/rfc/rfc2616.txt')
>>> ref
<itools.uri.URI object at 0x403e8f0c>
>>> print ref
http://www.ietf.org/rfc/rfc2616.txt
```

The reference can be inspected:

```
>>> print ref.scheme
http
>>> print ref.authority
```

¹<http://www.ietf.org/rfc/rfc2616.txt>

```
www.ietf.org
>>> print ref.path
/rfc/rfc2616.txt
>>> print ref.query

>>> print ref.fragment

>>>
```

1.1 Syntax

Below there is a short list of URIs:

```
http://www.ietf.org/rfc/rfc2396.txt
mailto:jdavid@itaapy.com
XXX
```

As the examples show URIs can be pretty different (compare the `http` and `mailto` URIs).

An URI is divided in two parts, the first one is the scheme (e.g. `http`, `ftp`, etc.). The syntax and semantics of the second part depends on the scheme:

```
uri = <scheme>:<scheme-specific-part>
```

However, many schemes share a similar syntax, these URIs are sometimes known as *generic URIs*:

```
<scheme>://<authority><absolute path>?<query>#<fragment>
```

Let's see what each component means.

Scheme

Defines the method or protocol to access the resource.

Authority

Defines the server address that hosts the resource. Its syntax is:

```
authority = [userinfo@]hostport
```

Absolute path

Query

Fragment

1.2 References

So far we have seen absolute URIs, but what we actually work with are URI references, which can be absolute or relative. Examples of relative references are:

```
//www.ietf.org/rfc/rfc2396.txt  
/rfc/rfc2396.txt  
../rfc2616.txt
```

A relative reference is one that lacks, at least, the scheme. The examples above show the three kinds of references that exist.

First, **network paths** include the authority, followed by the absolute path. They are rarely used.

Second, the **absolute paths**, which lack both the scheme and the authority.

Third and last, the **relative paths**.

Chapter 2

Resources

So far we have seen how `itools.uri` let us to identify and locate resources, and how we can work with resolve URI references.

The problem

But, how to retrieve and manipulate the resource an URI identifies? The standard library provides the modules `urllib` and `urllib2`, but XXX

The solution

To shortcut these constraints `itools.resources` has been developed. It provides an abstraction layer over resources: doesn't matters where the resources are stored (in the local filesystem, in a remote web server, etc.), they are handled with the same consistent API.

Show me the code

Lets load a resource first:

```
>>> from itools.resources import get_resource
>>> resource = get_resource('examples/hello.txt.en')
>>> print resource
<itools.resources.file.File instance at 0x402175ac>
```

This code creates a resource from a file in the local file system, now it is possible to perform some basic operations on it:

```
>>> print resource.get_data()
hello world

>>> print resource.get_size()
12
```

These are the most basic operations, the whole API is summarized in the Section 2.3.

2.1 Files and Folders

But in the file system there are more things than just files, there are folders too, so `itools.resources` provides folder resources:

```
>>> examples = get_resource('examples')
>>> print examples
<itools.resources.file.Folder instance at 0x401e568c>
```

That's it, just like files. Now we can look at the contents of the folder:

```
>>> print examples.get_resources()
['hello.txt.es', 'hello.txt.en', 'CVS']
```

And get a resource from within the folder:

```
>>> hello = examples.get_resource('hello.txt.en')
>>> print hello.get_data()
hello world
```

Resource tree

Thanks to folders the information is represented as a tree of resources. Actually, the methods we have seen so far expect paths as parameters.

2.2 Other schemes

Out of the box support for several schemes is provided.

2.2.1 http

It is also possible to use web pages as file resources (this is an example of the storage abstraction):

```
>>> r = get_resource('http://example.com')
>>> print r
<itools.resources.http.File instance at 0x404aadc>
>>> print r.get_mimetype()
text/html
>>> print r.get_data()
<HTML>
<HEAD>
  <TITLE>Example Web Page</TITLE>
</HEAD>
<body>
<p>You have reached this web page by typing
...
```

2.3 API

Files

- `get_atime()`
Returns the last time the resource was accessed¹.
- `get_ctime()`
Returns the time the resource was created.
- `get_data()`
Returns the resource data as a byte string.
- `get_mimetype()`
Returns the mime type of the resource.
- `get_mtime()`
Returns the last time the resource was modified.
- `get_size()`
Returns the length of the resource data.

Folders

- `del_resource(path)`
Removes the resource at the given path².
- `del_resources(paths)`
Removes the resources at the given paths.
- `get_atime()`
Returns the last time the resource was accessed.
- `get_ctime()`
Returns the time the resource was created.
- `get_mimetype()`
Returns the mime type of the resource.
- `get_mtime()`
Returns the last time the resource was modified.
- `get_resource(path)`
Returns a list with the names of the resources that this resource contains.
- `get_resources(path='.')`
Returns a list with the names of all the resources in the given path.
- `has_resource(path)`
Returns true if the given resources exists, false otherwise.
- `set_resource(path, resource)`
Adds the given resource to the given path.

¹It is a `datetime.datetime` instance.

²The path parameter is always an string like 'a/b/c'.

Chapter 3

Resource Handlers

Ok, so we have learned to identify resources, to retrieve them and to get some information from them with a consistent API.

The problem

But the resource API is very simple, it lacks any knowledge about the content of the resource. No way to manipulate it: for example to transform XML documents, to ask a PO file for translations, etc..

The standard library from Python provides many different modules to parse and work with different file formats: there is the `xml` package, the `csv` module, `zipfile`, etc.. There are too third party modules that provide support for other file formats.

However, there is not consistence at all through the APIs of these modules, their “philosophy” is sometimes too much different. For example, the `HTMLParser` must be subclassed and its methods specialized, while with the `expat` parser the handlers are registered instead of being specialized.

The solution

In order to improve the state of the art, `itools.handlers` provides a consistent framework to develop resource handlers. There are two benefits from this approach: first, all handlers share a common “philosophy”, what makes learning and remembering how to use them something much more easy; second, as they are based on resources, they can handle the information wherever it is, as far as it is represented as a resource.

Show me the code

First load a resource, as seen in the previous chapter:

```
>>> from itools.resources import get_resource
>>> resource = get_resource('examples/hello.txt.en')
```

Now, build a handler for the resource:

```
>>> from itools.handlers.Text import Text
>>> handler = Text(resource)
```

```
>>> print repr(handler)
<itools.handlers.Text.Text object at 0x403eca2c>
```

Finally we can start to work with our handler:

```
>>> unicode(handler)
u'hello world\n'
>>> str(handler)
'hello world\n'
```

Of course only text files can be presented as unicode. `str(handler)` outputs the text file in UTF-8, to get it in a different encoding use `to_str`:

```
>>> handler.to_str(encoding='ascii')
'hello world\n'
```

3.1 Files and Folders

As there are two kinds of resources, there are two kinds of handlers too: files and folders. However, they share something in common. First, every handler manages one resource and only one resource:

```
>>> print handler.resource
<itools.resources.file.File instance at 0x406461cc>
```

Second, there are two ways to create a handler, with a resource or without. We have already seen how
XXX

3.2 Handler databases

The example above builds a text handler for the resource, but how do we know it is a text file?

Handler databases keep a mapping from *resource types* to *handler classes*:

```
>>> from itools.resources import get_resource
>>> from itools.handlers import database
>>> resource = get_resource('examples/hello.txt.en')
>>> handler = database.get_handler(resource)
>>> print handler
<itools.handlers.Text.Text object at 0x404a3fcc>
```

By default the resource type is its mime type, though it can be explicitly specified with a second optional argument:

```
>>> handler = database.get_handler(resource, 'text/plain')
```

The Table 3.1 shows the database API.

- `get_handler(resource, type=None)`
Returns a handler for the given resource.
- `set_file_handler(resource_type, handler, extensions=())`
Associates a file handler class to the resource type. The optional parameter `extensions` contains the file extensions to register to the given resource type.
- `set_folder_handler(resource_type, handler)`
Associates a folder handler class to the resource type.

Table 3.1: Database API

Chapter 4

eXtensible Markup Language

Chapter 5

Simple Template Language

This is a template language to process and transform XML files. It is aimed at presentation, for example to produce the web pages that make up the user interface of a web application.

Unlike other template languages in the Python world, STL does not mix Python code within the template. The STL statements only describe the transformations to be performed on the template. It is a descriptive language.

For example, look at the template below (see `demo/hello.stl`):

```
<html>
  <head></head>
  <body>
    <h1 stl:content="title" />
  </body>
</html>
```

When this template will be processed, the content of the `<h1>` tag will be replaced by the value of the variable `title`. But, which is its value?

5.1 The namespace

In order to process an STL template, you need to pass it a Python namespace. But first we have to load the template as a handler:

```
>>> from itools.handlers import get_handler
>>> handler = get_handler('demo/hello.stl')
>>> print handler
<itools.handlers.stl.Template object at 0x401ee1ec>
```

Now we build the namespace and process the template:

```
>>> namespace = {'title': 'hello world'}
>>> print handler(namespace)
<html>
  <head></head>
  <body>
    <h1>hello world</h1>
```

```

    </body>
</html>

```

As this example shows, the value of the variable `title` is looked at the namespace we pass to the template to process it.

5.2 Repeat

Let's see a more complex example. The template below shows a simple address-book (see `demo/addressbook.stl`):

```

<html>
  <head></head>
  <body>
    <h3>Addressbook</h3>
    <ul>
      <li stl:repeat="address addressbook">
        <stl:block content="address/lastname" />,
        <stl:block content="address/firstname" />:
        <stl:block content="address/telephone" />
      </li>
    </ul>
  </body>
</html>

```

The first new thing this example shows is the `repeat` statement. While the `content` statement expects a string as the value, the `repeat` statement expects a list. When this template is processed, the xml output will contain as many `<li>` elements as items are in the `addressbook` list. Within the element, in each iteration over the `addressbook` list, the variable `address` will be the respective item of the list.

The second new thing we see is the `<stl:block>` element. When the template is processed the `<stl:block>` elements are automatically removed.

Finally, look at the expression `address/lastname`, it shows the *slash* operator, which lets to traverse namespaces. So the variable `address` is expected to be a dictionary, and `lastname` a key in that dictionary, whose value is a string.

5.2.1 The Python side

Now let's see the Python code (in the script `demo/addressbook.cgi`), this time as a class which keeps the addressbook information, with a method that process the template to produce an HTML page that shows the addressbook.

```

# Import from itools
from itools import get_abspath
from itools.handlers import get_handler

class Address(object):
    def __init__(self, lastname, firstname, telephone):
        self.lastname = lastname

```

```

        self.firstname = firstname
        self.telephone = telephone

class Addressbook(object):
    def __init__(self):
        self.addressbook = []

    def add_address(self, lastname, firstname, telephone):
        address = Address(lastname, firstname, telephone)
        self.addressbook.append(address)

    def view(self):
        # Load the STL template
        path = get_abspath(globals(), 'addressbook.stl')
        template = get_handler(path)

        # Build the namespace
        namespace = {'addressbook': self.addressbook}

        # Process the template and return the output
        return template(namespace)

```

The end of the module creates an `addressbook` instance, fills it with a couple of entries and makes up the response:

```

if __name__ == '__main__':
    # Create the addressbook
    addressbook = Addressbook()
    addressbook.add_address('Jordan', 'Robert', '0606060606')
    addressbook.add_address('Buendia', 'Aureliano', '0612345678')

    # Output the HTTP headers
    print "Content-Type: text/html"
    print

    # Output the addressbook content
    print addressbook.view()

```

The result of running this program is:

```

Content-Type: text/html

<html>
  <head></head>
  <body>
    <h3>Addressbook</h3>
    <ul>
      <li>
        Jordan,
        Robert:
        0606060606

```

```
</li><li>
  Buendia,
  Aureliano:
  0612345678
</li>
</ul>
</body>
</html>
```