itools

Juan David Ibáñez Palomar[1]
jdavid@itaapy.com

June 6, 2005

[1]Permission 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

Chapter 1

Introduction

This paper is the official documentation of `itools`.

1.1 What is `itools`?

`itools` is a Python[1] library. Originally intended to develop web applications, it has evolved into a large framework suitable for a wide range of applications.

It can be seen as an extension of the Python's *Standard Library*, as it provides a set of lightly coupled sub-packages which can be used independently; `itools` is split into the following sub-packages:

- `itools.uri` – an API to manage URIs, to identify and locate resources.
- `itools.types` – type marshalers for basic types (integer, date, etc.) and not so basic types (filenames, XML qualified names, etc.).
- `itools.resources` – an abstraction layer over resources that lets to manage them with a consistent API, independently of where they are stored.
- `itools.handlers` – resource handlers infrastructure (resource handlers are non persistent classes that add specific semantics to resources). This package also includes several handlers *out of the box*.
- `itools.gettext` – resource handlers for PO and MO files.
- `itools.xml` – includes an intuitive event driven XML parser, a handler for XML documents, and the **S**imple **T**emplate **L**anguage.
- `itools.xhtml` – resource handlers for XHTML documents.
- `itools.html` – resource handlers for HTML documents.
- `itools.i18n` – tools for language negotiation and text segmentation.
- `itools.workflow` – represent workflows as automatons, objects can move from one state to another through transitions, classes can add specific semantics to states and transitions.
- `itools.catalog` – An index and search engine.

[1]`http://www.python.org`

1.1.1 The resource-handler model

Of everything in `itools` I am probably most proud of the the three sub-packages `itools.uri`, `itools.resources` and `itools.handlers`, which make up what I call the *resource-handler* model.

There is a linear relationship of dependency between these modules, `uri` depends on nothing but Python, `resources` depends on `uri`, and `handlers` depends on `resources`.

This allows to use these modules on a flexible way. For example, you can just use `itools.uri` (ignoring the others) to benefit from a higher level API to work with URIs than that provided by the Standard Library (`urlparse`).

Or you may use `itools.resources` if you want to benefit from an abstraction layer over the storage. This is to say, if you want to be able to manipulate many resources stored in different systems and accessed through different protocols with the same, consistent, rich API.

You may even take advantage of `itools.handlers` without fully understanding the model behind. For example, you could use some of the handlers `itools` offers out-of-the-box for standard file formats like CSV, PO or XHTML, to simplify your live if you ever need to work with one of these formats.

But, in order to exploit everything `itools` has to offer to its limit, you may choose to base part or all of your application architecture on the *resource-handler* model, a model heavily influenced by the filesystem and the Web. In this situation the three packages (`uri`, `resources` and `handlers`) show themselves as three different components with a distinct role in the architecture:

- `itools.uri` identifies and locates resources, wherever they are. Altogether with `itools.resources` it effectively enables your information system to be distributed through an heterogeneous base.

- `itools.resources` provides persistency to your data, this is to say, resources are where the data is stored.

- and `itools.handlers` is where the logic lives. The essential characteristic of a resource handler (or handler for short) is that it is non-persistent, instead it is associated with a resource, whose content it is responsible to manage.

The first chapters of this document will cover these sub-packages, `itools.uri`, `itools.resources` and `itools.handlers` with detail.

1.1.2 eXtensible Markup Language

The `itools.xml` package depends on `itools.handlers`, this is its first advantage. It represents an XML document as an DOM like tree. But it also provides support for schemas, what allows to easily build higher level data structures.

The package also provides handlers for specific document types out-of-the-box, most notably XHTML and HTML (even if HTML is not XML, it shares a lot with XHTML).

1.1.3 The Simple Template Language

The Simple Template Language is included in the `itools.xml` package, but it is important enough to deserve its own chapter in the documentation.

Its design goals are:

- Truly separate logic and presentation. Even Python expressions are not allowed within the template.
- Really simple. It can be mastered in half a day.
- Damn fast (easy to achieve through simplicity).
- Secure. Even non-trusted users could write templates without risk, because code is not allowed within the template.

And the key idea behind, is to make it a *descriptive* language.

1.1.4 Workflow

The sub-package `itools.workflow` (once known as *flux*) is the oldest code in `itools`. It does not depend on anything but the Standard Library, and don't puts any restriction on the storage. This makes it very easy to combine with other frameworks.

There is a chapter exclusively dedicated to it.

1.1.5 Internationalization and Localization

The `itools.i18n` package provides a wide range of tools for internationalization and localization of both software and data. From message extraction to language negotiation, through text segmentation, fuzzy matching or algorithms to guess the language a text is written in.

A chapter is devoted to this topic.

1.1.6 Index and Search

The `itools.catalog` sub-package provides an index and search engine. Though still young, it already provides full text indexing, boolean queries and results sorted by weight.

1.2 Who is this document for?

This document is addressed to Python developers. Though it may be useful to software architects in general, as some ideas exposed here could be found interesting[2].

It is convenient to have some basic skills with the Python programming language to fully understand this document. Probably the best introduction to Python is the official tutorial:

`http://python.org/doc/2.3.5/tut/tut.html`.

[2]For example, `itools.workflow` inspired XXX to write a similar engine in Java, see `http://XXX`

This document touches many different technologies and standards, such as XML. References will be given in the relevant chapters.

1.3 Project status

Currently `itools` is in alpha stage of development (at the time of this writing the last version available is **0.8**). This means that the API is not yet stable, the documentation is incomplete, and there may be bugs.

However, we have been using it on a daily basis for a long time now. It fuels the *i*Kaaro[3] Content Management System, and most of our customer projects.

1.4 Installation

Before going further, be sure your system is correctly setup. First you will need Python 2.3.4 or later.

Then download the last `itools` version from `http://www.ikaaro.org`, unpack it somewhere, and install it; `itools` uses *distutils*, so just type:

```
$ python setup.py install
```

Be sure to have the right permissions.

1.5 About this document

The development keeps ahead of the documentation. I would say the docs cover around 60% of `itools`.

There are also two appendixes, one explaining the coding style `itools` is written in, another one introducing the use of *GNU arch*. Both are specially addressed to those that want to contribute back to the main tree.

[3] http://www.ikaaro.org

Chapter 2

Uniform Resource Identifiers

In the wild Internet, the first challenge we encounter is how to identify and locate the numerous resources that populate it. Well, that's what **U**niform **R**esource **I**dentifiers (or just URIs) are for.

The Python Standard Library (batteries included!) provides a module named `urlparse` which is able to split a generic URI into its main parts (scheme, authority, path, query and fragment), to rebuild it, and to resolve relative references.

But there are more things we would like to do with a URI. For example we could go further in the parsing process and split the path into its segments, then split each segment into the name and the parameters if any; we could get the user information, host address and port number from the authority; we could normalize URIs; we could implement other operations beyond just resolving relative references, etc.

This is the purpose of `itools.uri`, to provide a complete API to parse and work with URIs, following the standard as described by **RFC2396**.

The main function provided by `itools.uri` is `get_reference`, a factory to build references:

`get_reference(reference)`
- Parses the given string and returns a reference object.

Let's go right to the code:

```
>>> from itools import uri
>>> r1 = uri.get_reference('http://www.w3.org/TR/REC-xml/#sec-intro')
>>> r2 = uri.get_reference('mailto:jdavid@itaapy.com')
>>> r3 = uri.get_reference('http://www.ietf.org/rfc/rfc2616.txt')
>>> r4 = uri.get_reference('http://sf.net/cvs/?group_id=5470')
>>> r5 = uri.get_reference('news:comp.infosystems.www.servers.unix')
```

2.1 Syntax

Before going further, we will give an overview of the URI syntax, something required to understand the rest of this chapter.

A URI is divided in two parts. The first one is the scheme: `http`, `ftp`, `mailto`, 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 for the second part, these URIs are known as *generic URIs*.

2.1.1 Generic URIs

The syntax of a generic URI reference is:

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

Generic URIs are modeled by `uri.Reference`. Following the code at the beginning, we are going to inspect the `r1` object:

```
>>> r1
<itools.uri.Reference object at 0x403ebc4c>
>>> print r1
http://www.w3.org/TR/REC-xml/#sec-intro
>>> print r1.scheme
http
>>> print r1.authority
www.w3.org
>>> print r1.path
/TR/REC-xml/
>>> print r1.query

>>> print r1.fragment
sec-intro
```

Note that there is an attribute for every component: the scheme, the authority, the path, the query and the fragment. Now we are going to quickly describe each of these components:

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 The path identifies the resource within the scope of the scheme and authority.

It consists of a sequence of segments. A segment has two parts, the name and the parameters, though the parameters are optional. The syntax is:

```
absolute path = /<relative path>
relative path = <segment>[/<relative path>]
segment = <name>[;<parameters>]
```

Query The query is information to be interpreted by the resource. It does not have a pre-defined syntax.

Fragment The fragment is a reference within the resource.

Actually, the fragment does not belong to the URI, as it does not help to identify the resource, however we include it here because it does appears in URI references, what is what we work with.

As the query, the fragment does not have a pre-defined syntax.

2.1.2 Non Generic URIs

Other schemes do not follow the generic syntax. As an example, let's inspect the `r2` object seen before:

```
>>> r2 = uri.get_reference('mailto:jdavid@itaapy.com')
>>> print r2
mailto:jdavi@itaapy.com
>>> r2
<itools.uri.Mailto object at 0x403f45ec>
>>> print r2.scheme
mailto
>>> print r2.username
jdavid
>>> print r2.host
itaapy.com
```

As you see the `r2` is not an instance of `uri.Reference`, but an instance of `uri.Mailto`.

2.2 Relative references

So far the examples we have seen show absolute URIs, but there are relative URI references too. A relative reference is one that lacks, at least, the scheme. There are three types of of relative references: network paths, absolute paths, and relative paths:

Network paths Network paths only lack the scheme, they start by a double slash and the authority, followed by the absolute path. They are rarely used.

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

Absolute paths The absolute paths lack both the scheme and the authority. They start by a slash.

```
/rfc/rfc2396.txt
```

Relative paths Relative paths lack the first slash of absolute paths. They can start by the special segment ".", or by one or more "..". Examples are:

```
rfc/rfc2396.txt
./rfc/rfc2396.txt
../rfc2616.txt
```

2.2.1 Resolving references

The most common operation with relative references is to resolve them. That is to say, to obtain (with the help of a base reference) the absolute reference that identifies our resource. This is achieved with the `resolve` method:

```
>>> base = uri.get_reference('http://www.ietf.org/rfc/rfc2615.txt')
>>> print base.resolve('//www.ietf.org/rfc/rfc2396.txt')
http://www.ietf.org/rfc/rfc2396.txt
>>> print base.resolve('/rfc/rfc2396.txt')
http://www.ietf.org/rfc/rfc2396.txt
>>> print base.resolve('rfc2396.txt')
http://www.ietf.org/rfc/rfc2396.txt
```

2.3 Paths

One component that deserves special attention is the path. The path of a generic URI is an instance of the `uri.Path` class:

```
>>> ref = uri.get_reference('http://www.ietf.org/rfc/rfc2616.txt')
>>> ref.path
<itools.uri.Path at 0x403f50a4>
>>> print ref.path
/rfc/rfc2616.txt
```

Paths are iterable:

```
>>> for segment in ref.path:
...     print segment
...
rfc
rfc2616.txt
```

Each component of the path is called a segment. Segments are instances of the class `uri.Segment`. Each segment has two components, the name and the parameter. The code below illustrates this:

```
>>> path = uri.Path('/itaapy;lang=es/team')
>>> for segment in path:
...     print repr(segment)
...     print '  name:', segment.name
...     print '  param:', segment.param
...
<itools.uri.Segment object at 0x404c1acc>
  name: itaapy
  param: lang=es
```

```
<itools.uri.Segment object at 0x404c1a2c>
  name: team
  param: None
```

The `uri.Path` class also provides an API to manipulate paths:

`is_absolute()`
- Returns `True` if the path is absolute (i.e. if it starts by an slash), `False` otherwise.

`is_relative()`
- Returns `True` if the path is relative (i.e. if it does not start by a slash), `False` otherwise.

`get_prefix(path)`
- Returns the path that is common to `self` and to the given path.

`resolve(path)`
- Returns a new path from a base path (`self`) and the given path. Follows the RFC2396 standard, i.e. takes into account the trailing slash.

`resolve2(path)`
- Same as `resolve`, but it does not teke into account the trailing slash.

`get_pathto(path)`
- Returns the path needed to go from `self` to the given path (this complements the `resolve` method).

`get_pathtoroot()`
- Returns a relative path to the root (something like `../../..`).

Chapter 3

Type marshalers

The module `itools.types` provides deserialization and serialization code for several simple types, from built-in types like intergers or dates to slightly more complex types like URIs or XML qualified names.

All of them are implemented as the couple of class methods `decode` and `encode`, for example:

```
>>> from itools.types import DateTime
>>> datetime = DateTime.decode('2005-05-02 16:47')
>>> datetime
datetime.datetime(2005, 5, 2, 16, 47)
>>> DateTime.encode(datetime)
'2005-05-02 16:47'
```

This approach, to implement the serialization/deserialization code separate from the type itself, allows to avoid subclassig built-in types, what has a performance impact.

This also illustrates one of the software principles behind the itools coding, different programming aspects should be cleary distinct in the implementation and programming interface.

3.1 Built-in types

The Python built-in types supported, including those provided by the standard library, are integers, text strings (unicode), byte strings, booleans, dates and datetimes.

Integers (`itools.types.Integer`)

An integer number is serialized using ASCII characters. This means a call to `Integer.decode(x)` is equivalent to `int(x)`, and `Integer.encode(x)` does the same than `str(x)`.

Unicode strings (`itools.types.Unicode`)

Unicode strings are serialized using the UTF-8 encoding (by default).

Byte strings (`itools.types.String`)

A byte string does not needs to be serialized or deserialized, the output is always equal to the input.

Booleans (`itools.types.Boolean`)

Boolean values are encoded with the “0” character for the *false* value and with the “1” character for the *true* value.

Dates (`itools.types.Date`)

Dates are encoded following the ISO 8601 standard[1]: “YYYY-MM-DD”.

Datetimes (`itools.types.DateTime`)

Date and time is encoded with the pattern: “YYYY-MM-DD hh:mm”.

3.2 Other types

Type marshalers are provided for other three types: URIs, filenames and XML qualified names.

URIs (`itools.types.URI`)

The URI decoder will build and return one of the URI reference objects defined in the `itools.uri` package, usually it will be an instance of the class `itools.uri.generic.Reference`.

Filenames (`itools.types.FileName`)

Usually filenames include extensions to indicate the file type, and sometimes other information like the language. The filename decoder will parse a filename and return a tuple where the first element is the filename, the second element is the file type, and the last element is the language. For example:

```
>>> from itools.types import FileName
>>> FileName.decode('index.html.en')
('index', 'html', 'en')
>>> FileName.decode('index.html')
('index', 'html', None)
>>> FileName.decode('index')
('index', None, None)
```

XML qualified names (`itools.types.QName`)

An XML qualified name has two parts, the prefix and the local name, so our decoder will return a tuple with these two elements:

[1]http://www.iso.org/iso/en/prods-services/popstds/datesandtime.html

```
>>> from itools.types import QName
>>> QName.decode('dc:title')
('dc', 'title')
>>> QName.decode('href')
(None, 'href')
```

The encoder expects a two element tuple:

```
>>> QName.encode(('dc', 'title'))
'dc:title'
>>> QName.encode((None, 'href'))
'href'
```

Chapter 4

Resources

In the previous chapter we have learned how to work with URIs, objects that identify resources; but how to manipulate the resources themselves?

The problem we face now is the fact that the resources are dispersed, stored in different systems, and accessed through different protocols. In spite of that, it would be nice to be able to manipulate them with a uniform API.

That's the purpose of `itools.resources`, to provide an abstraction layer over resources: doesn't matters where the resources are stored (in the local file system, in a remote web server, etc.), they are handled with the same consistent API.

4.1 Retrieving resources

Of course, the first step, before working with a resource, is to retrieve it, this is done with the function `get_resource`:

`get_resource(reference)`
- From the given URI reference returns a resource object.

Let's see a few examples:

```
>>> from itools.resources import get_resource
>>> get_resource('examples/hello.txt')
<itools.resources.file.File instance at 0x402175ac>
>>> get_resource('http://example.com')
<itools.resources.http.File instance at 0x404aadec>
>>> get_resource('examples')
<itools.resources.file.Folder instance at 0x401e568c>
```

These examples show two fundamental aspects of `itools.resources` which we will explore on detail later. First is the support for multiple protocols, in particular the example illustrates the schemes `file` for the local file system, and `http` for the **H**yper**T**ext **T**ransfer **P**rotocol.

The second aspect is the support for two kinds of resources, files and folders. Before looking at each type of resource, files and folders, let's see the API they share:

`uri`
- The URI reference the resource was retrieved from.

`get_mimetype()`
- Tries to guess the mimetype of the resource and returns it as a string. Folders are always `application/x-not-regular-file`. Returns `None` if it fails.

`get_ctime()`
- Returns a `datetime` object with the time the resource was created.

`get_mtime()`
- Returns a `datetime` object with the last time the resource was modified.

`get_atime()`
- Returns a `datetime` object with the last time the resource was accessed.

It may happen that a specific scheme does not implements one or more of these methods.

4.2 File Resources

Besides the common methods for files and folders, each resource type has a specific API, the one for file resources is:

`read()`
- Returns the resource data as a byte string.

`write(data)`
- Replaces the resource content by the given data (a byte string).

`get_size()`
- Returns the length (the number of bytes) of the resource data.

`__getitem__(index)`
- Returns the byte at the given position.

`__getslice__(a, b)`
- Returns the byte string that goes from *a* to *b*.

`__setitem__(index, value)`
- Sets the given value to the given position (*index*). Usually value will be just a byte, but it may be a slice too.

`append(data)`
- Appends the given data to the resource.

As an example of the API let's exercise it with a web page:

```
>>> resource = get_resource('http://example.com')
>>> print resource
<itools.resources.http.File instance at 0x404aadec>
>>> print resource.get_mimetype()
```

```
text/html
>>> print resource.get_size()
438
>>> print resource.read()
<HTML>
<HEAD>
  <TITLE>Example Web Page</TITLE>
</HEAD>
<body>
<p>You have reached this web page by typing
...
```

Of course, in this example, we can not modify the resource as we don't have write access to the web server.

The methods `__getitem__`, `__setitem__`, etc. are specially useful to work with binary files, as they allow to read and write data at specific positions within a resource. These methods allow to access the resource with the same syntax used to access lists in Python:

```
>>> import struct
>>> n = struct.pack('>I', 47)
>>> resource[20:24] = n
```

This brief example shows how to write the number 47 at the position 20 of a given resource, encoded as an 32 bits unsigned integer.

For a real example you can look at `itools.catalog`, which makes extensive and systematic use of this approach.

4.3 Folder Resources

The specific API for folders is:

`get_resource(path)`
- Returns the resource in the given path (where path is either an instance of `uri.Path` or a string).

`get_resource_names(path='.')`
- Returns a list with the names of all the resources in the given path.

`get_resources(path='.')`
- Returns the resources in the given path (it is a generator).

`has_resource(path)`
- Returns `True` if there is a resource in the given path, `False` otherwise.

`set_resource(path, resource)`
- Adds the given resource to the given path.

`del_resource(path)`
- Removes the resource at the given path.

`del_resources(paths)`
- Removes the resources at the given paths (where paths is a list of paths).

`traverse()`
- This method allows to traverse the resource tree below this folder. It is a generator which returns a resource at a time, starting by this folder.

Let's exercise the API a little:

```
>>> examples = get_resource('examples')
>>> print examples.get_resource_names()
['hello.xhtml~', 'hello.xhtml', 'hello.txt', '.arch-ids']
>>> hello = examples.get_resource('hello.txt')
>>> print hello.read()
hello world
```

To copy a web page to the local file system:

```
>>> tmp = get_resource('/tmp')
>>> web_page = get_resource('http://example.com')
>>> tmp.set_resource('example.html', web_page)
>>> copy_of_web_page = tmp.get_resource('example.html')
>>> print web_page
<itools.resources.http.File instance at 0x405e8a2c>
>>> print copy_of_web_page
<itools.resources.file.File instance at 0x405fb64c>
```

To copy a whole tree:

```
>>> from pprint import pprint
>>> talks = get_resource('/home/jdavid/talks')
>>> tmp.set_resource('talks', talks)
>>> pprint(tmp.get_resource_names('talks/EuroPython2004'))
['itools-vfs',
 'Makefile',
 'itools-vfs.log',
 'itools-vfs.tex',
 'itools-vfs.aux',
 'itools-vfs.toc',
 'itools-vfs.dvi',
 'itools-vfs.ps',
 'itools-vfs.pdf',
 'itools-vfs.tex~']
```

This is more impressive when copying a big tree from one storage to another. For example we use it in the context of the **iKaaro** Content Management System to export and import web sites from the **Z**ope **O**bject **D**ata**B**ase to the file system, and back from the file system to the **ZODB**.

4.4 Supported schemes

At the time of this writing the number of supported schemes is pretty short: `itools.file` for the file system, `itools.http` for the HTTP protocol (though the implementation is minimal), and `itools.memory` to store resources in memory. The **iKaaro** CMS provides support to store resources in Zope 2.

4.4.1 The memory storage

The memory storage (`itools.memory`) deserves few lines as it is very handy. It is possible to build a resource from scratch directly calling its constructor:

```
>>> from itools.resources import memory
>>> resource = memory.File('hello world')
>>> resource
<itools.resources.memory.File instance at 0x405e458c>
>>> resource.read()
'hello world'
>>> tmp.set_resource('hello.txt', resource)
```

This technique is used internally by resource handlers to build handler instances without passing a resource. They are the object of our next section.

Chapter 5

File handlers

Resources provide persistence to our data, but they lack any knowledge about the structure and the meaning of the information they contain. For example, the resource layer ignores that an XML document has a tree structure, hence it does not provide an API to traverse it; nor it knows how to get the messages and translations from a PO file.

This is the purpose of the *resource handlers*, to add semantics to specific resources.

As with resources, there are two types of handlers, files and folders. This first chapter focuses on file handlers, the next one will analyze the folder handlers.

Several key concepts will be exposed in this chapter, the relationship between resources and handlers, the load (de-serialization) and save (serialization) operations, the handler skeleton, and the handler factory. We will learn to use the file handlers `itools` offers out of the box, and how to write custom handler classes.

5.1 Introduction

To get a feeling, let's start with an example:

1. First we load a resource as seen in the previous chapter:

```
>>> from itools.resources import get_resource
>>> resource = get_resource('http://example.com')
```

2. Now we build a handler for the resource:

```
>>> from itools.xml import HTML
>>> handler = HTML.Document(resource)
>>> print handler
<itools.html.HTML.Document object at 0x40647b4c>
>>> print handler.to_str()
<HTML>
<HEAD>
  <TITLE>Example Web Page</TITLE>
</HEAD>
<body>
```

```
    <p>You have reached this web page by typing
    ...
```

There are several things to highlight here. First the way we have built our handler instance, passing a resource to a handler class (`HTML.Document`), later we will see other ways to build handler instances.

Second the `to_str` method (all the file handlers have this method), which serializes the handler, i.e. transforms the handler to a sequence of bytes. This is a key concept to which we will come back later.

3. Now we can start working with the handler:

```
>>> from itools.xml import XML
>>> for node in handler.traverse():
...     if isinstance(node, XML.Element) and node.name == 'title':
...         print node.children
Example Web Page
```

Unlike `to_str`, the method `traverse` is specific to XML documents. Every handler class provides its own API.

One resource, many handlers

The relationship between resources to handlers is *1* to *n*. While there may be several different handlers associated to the same resource (though it is not obvious how this is useful), a handler is only associated to one resource.

The resource associated to a handler is accessible through the `resource` attribute. Following the example above, the code below checks that the resource we loaded in the first place is exactly the same our handler is associated to:

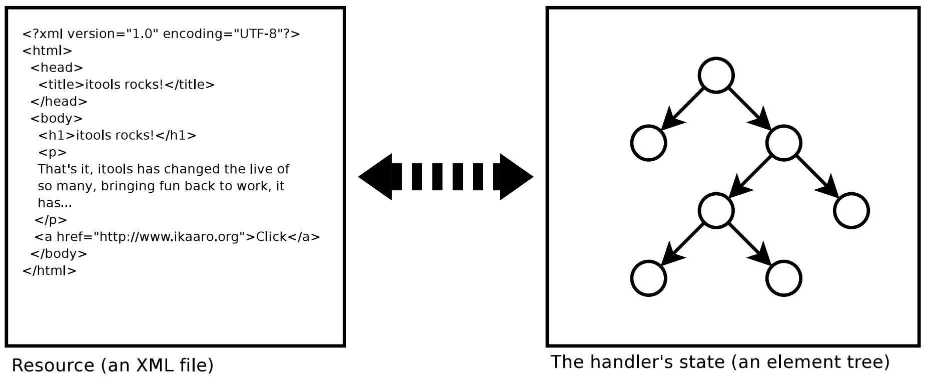
```
>>> resource
<itools.resources.http.File instance at 0x404aadec>
>>> handler.resource
<itools.resources.http.File instance at 0x404aadec>
>>> resource is handler.resource
True
```

5.2 Handler's state

A handler is a non-persitent object, it stores in volatile memory a data structure that represents the resource's content. For example, the Figure 5.1 shows at the left an XML file, and at the right the state of the handler as a tree of XML elements.

The state of a handler is stored within the instance variable `state`, what this variable contains depends on the handler class; e.g. an element tree for XML documents, a mapping from message to translation for PO files, or just a unicode string for plain text files. For example:

```
>>> from itools.xhtml import XHTML
>>> handler = XHTML.Document()
>>> print handler
```



```
<?xml version="1.0" encoding="UTF-8"?>
<html>
 <head>
  <title>itools rocks!</title>
 </head>
 <body>
  <h1>itools rocks!</h1>
  <p>
  That's it, itools has changed the live of
  so many, bringing fun back to work, it
  has...
  </p>
  <a href="http://www.ikaaro.org">Click</a>
 </body>
</html>
```

Figure 5.1: Handler's state

```
<itools.xhtml.XHTML.Document object at 0xb7cae8ac>
>>> print handler.state
<itools.handlers.Handler.State object at 0xb7cae92c>
>>>
>>> from pprint import pprint
>>> pprint(handler.state.__dict__.keys())
['xml_version',
 'source_encoding',
 'document_type',
 'root_element',
 'standalone']
```

The state variable should never be accessed directly, instead the API each handler provides should be used.

We call *load* to the process of building the handler state from the data stored in the resource. And viceversa, we call *save* to the process of updating the resource with the changes made to the handler state.

So, while the handler and the resource are synced inmediately after the handler is loaded, this is to say, they contain the same information, one or the other may change, hence becoming desynchronized. There are a couple of scenarios we want to deal with:

- Once the handler is loaded, the resource is modified by other means, for example you open the file with an editor and modify it.

  Then the handler is outdated, its state represents an older version of the resource.

- After loading the handler, its state is modified through the API it provides.

  Then the resource is outdated, as the handler contains a newer version of the information.

5.2.1 Load

It may happen that the resource (a web page in our example) gets updated, then the handler will keep a data structure on memory that does not correspond anymore to the content of the resource, in other words, the handler would be out-of-date. We can check this by comparing the last modification time of the resource with the timestamp every handler always keeps:

```
>>> print handler.resource.get_mtime()
2004-11-28 19:53:57
>>> print handler.timestamp
2004-12-29 19:31:39.055367
>>> handler.resource.get_mtime() > handler.timestamp
False
```

Ok, the example above shows everything is alright. But what to do if it was not? then we would need to reload the resource, this is done with the `load_state` method:

```
>>> print handler.timestamp
2004-12-29 19:31:39.055367
>>> handler.load_state()
>>> print handler.timestamp
2004-12-29 19:51:50.152499
```

You should try it yourself with a resource in the filesystem: start the Python interpreter, build a handler for a resource in the filesystem, modify the resource with another program, see how the resource modification time is more recent than the handler's timestamp, then reload the handler and verify the handler is up-to-date now.

The full prototype of the method `load_state` follows:

`load_state(resource=None)`
- Loads the data from the given resource into the handler, if no resource is given, the one the handler is attached to will be used.

As you see the `load_state` method accepts an optional parameter, a resource. If it is not given the handler will be reloaded from the resource it handles. But if other resource is passed it will be loaded from the given resource. Then we would reach a situation opposite to the one seen before: the resource would be outdated, as the handler would keep a newer version.

5.2.2 Save

Another way to get a handler more recent than the resource it is associated with, is to modify the handler through the API it offers, which depends on the handler class (e.g. the API of an XML handler is different from the API of an image handler).

To update the resource so both resource and handler are in sync again we use the `save_state` method:

`save_state()`
- Saves the handler into the resource.

So `load_state` and `save_state` are two sides of the same coin. The first one reloads the handler with the resource, the second one saves the handler state in the resource.

5.2.3 Serialization and de-serialization

A file resource represents a sequence of bytes. A handler keeps a data structure in memory, and offers an API to inspect and modify this data structure.

Two key concepts in computer science will help to clarify what we have seen up to now:

serialization is the process by which a sequence of bytes is transformed into a data structure.

de-serialization is the process by which a data structure is transformed to a sequence of bytes.

Now it becomes obvious that the `load_state` method what actually does is to de-serialize the resource and update the handler. And viceversa, the `save_state` method serializes the handler and updates the resource.

Remember the method `to_str` we saw at the beginning? it is the one responsible for the serialization process. The `save_state` method calls `to_str` first, and then updates the resource.

5.3 The skeleton

As we have seen the handler constructor expects one parameter: the resource it is meant to handle (and once the handler is built the resource is always accessible with `handler.resource`).

However, it is also possible to build a handler without passing it any parameter; in this case a memory resource will be built on the fly, let's see an example:

```
>>> from itools.html import HTML
>>> handler = HTML.Document()
>>> handler
<itools.handlers.HTML.Document object at 0x403ee12c>
>>> handler.resource
<itools.resources.memory.File instance at 0x40638e4c>
>>> print handler.to_str()
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
  "http://www.w3.org/TR/html4/loose.dtd">
<html>
  <head>
    <meta content="text/html; charset=UTF-8" http-equiv="Content-Type">
    <title></title>

  <body></body>
</head></html>
```

The default content of the resource depends on the handler, and it is called the *handler skeleton*. The constructor also accepts arbitrary keyword parameters:

```
>>> handler = HTML.Document(title='Hello World')
>>> print handler.to_str()
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
```

```
<html>
  <head>
    <meta content="text/html; charset=UTF-8" http-equiv="Content-Type">
    <title>Hello World</title>
  </head>
  <body></body>
</html>
```

The keyword parameters are used to initialize the skeleton; in the example above the `title` parameter defines the HTML document title, which by default is empty. The parameters accepted depend on the handler.

5.4 Text and binary handlers

File handlers split into text and binary handlers. The class `Text` is the base class for all the text handlers.

The key difference is that text handlers represent the resource's content in memory as a unicode string, instead of just a byte string.

When the resource is loaded the text handler tries to guess the character encoding of the resource's content. For some file formats this information is stored within the content itself; for example an XML document starts by an XML declaration that specifies the encoding, if the declaration is missing it is assumed the encoding is UTF-8. When this information is not explicitly indicated, the text handler tries to guess the encoding by brute force.

With the character encoding the text handler decodes and loads the resource's content.

The API of binary and text handlers also differs, compare the one offered by the `File` handler class:

`to_str()`
- Returns the resource data as a byte string (this is similar to the method `handler.resource.read()`).

`set_data(data)`
- Updates the handler state with the given data.

with the one offered by the `Text` handler class:

`to_str(encoding='UTF-8')`
- Returns the resource data as a byte string, using the given encoding (defaults to `UTF-8`). Note that this will be different than the string returned by `handler.resource.read()` if the resource data is not encoded in `UTF-8` in the source.

`to_unicode(encoding='UTF-8')`
- Returns the resource data as an unicode string.

Note the `to_str` method for text handlers accepts the encoding as an optional parameter. This not only will return a byte string in the given charater encoding, it will also specify the encoding in the byte string for file formats that should include it.

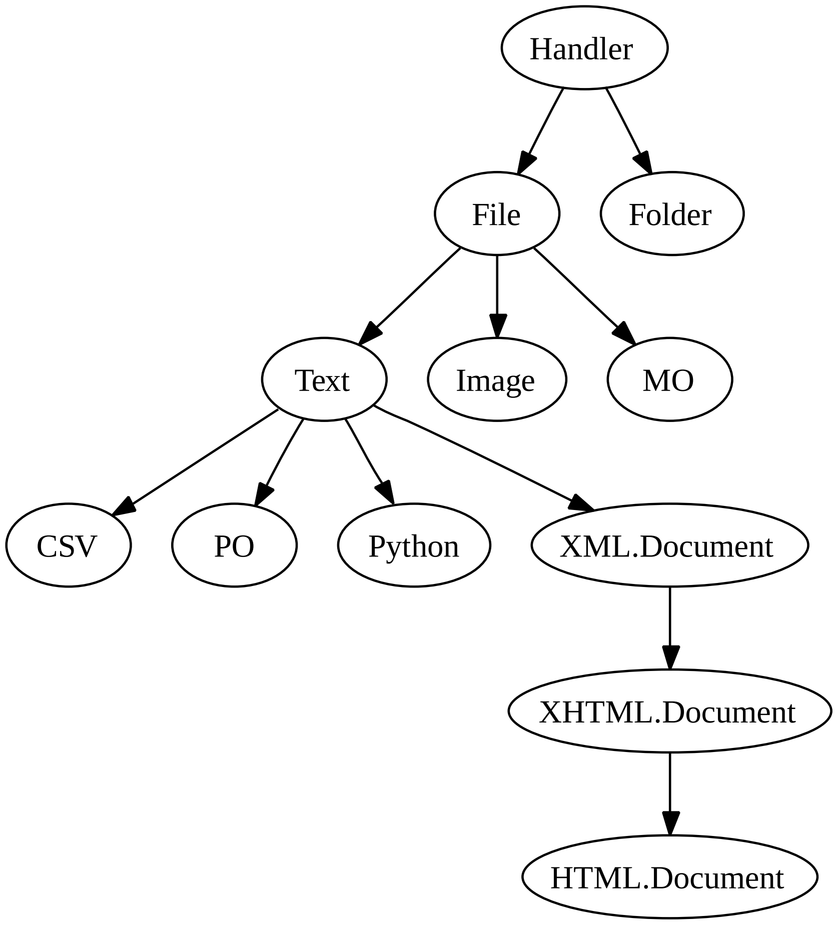


Figure 5.2: The handler tree

There is also the new method `to_unicode`, which returns the content as a unicode string. This method accepts the encoding parameter too. For file formats that do not include the encoding within the content, the result of `to_unicode` will be the same regardless of the given encoding. But for file formats like XML that should specify the encoding within the content, the returned unicode string will include it.

5.5 Overview of the available handlers

Out of the box `itools` comes with several handlers for different standard file formats. The Figure 5.2 shows an excerpt of the tree of the available handler classes, which express the inheritance relationship between them.

We are not going to see folder handlers here, as they will be studied in the next chapter.

We have already seen the difference between binary and text handlers. Now we are going to present some of the higher level file handler classes `itools` includes.

XML.Document This is the default handler for XML documents, which internally are represented as a tree. The API includes the methods already described for the `Text` handler; specific methods of `XML.Document` are:

`__cmp__(other)`
- Lets to compare two XML documents.

`get_root_element()`
- Returns the root element of this XML document.

`traverse()`
- This method allows to traverse the XML document, as it has a tree structure. It is a generator which returns a node at a time, starting by the document instance.

`traverse2()`
- As `traverse`, this method allows to traverse the XML document, though it is more powerful, and complex. It will be explained in the XML chapter.

XHTML.Document The handler for XHTML documents, it extends the API provided by `XML.Document` API with the methods:

`get_head()`
- Returns the head element.

`get_body()`
- Returns the body element.

`to_text()`
- Strips all the XML markup and returns the text content of the XHTML document. This is useful, for example, to index the document.

PO The handler to manage PO files, the message catalog of the GNU gettext utilities.

`get_msgids()`
- Returns the list of message ids stored in the catalog.

`get_messages()`
- Returns the list of messages stored in the catalog, where each message is represented as an instance of the class `PO.Message`.

`get_msgstr(msgid)`
- Returns the message string for the given message id.

`set_message(msgid, msgstr=[u''], comments=[u''], references={})`
- Adds a message (from the given parameters) to the catalog.

5.6 The handler factory

So far we have built a handler instance through a handler class:

```
>>> handler = HTML.Document(resource)
>>> handler
<itools.xml.HTML.Document object at 0x405740cc>
```

This is the standard pattern to build instances. However, if the resource is not an HTML document this procedure would fail. In other words, this procedure is only useful if you already know the kind of resource you are working with.

Another option is to let `itools` to choose which handler class to use. This can be done with the `build_handler` method, which provides the factory pattern.

`build_handler(resource)`
- A class method that identifies the given resource, chooses the available handler class that better matches it, and builds and returns a handler instance for it.

For example, from the `examples` directory type:

```
>>> from itools.resources import get_resource
>>> from itools.handlers.Handler import Handler
>>> from itools import xml
>>>
>>> here = get_resource('.')
>>> for name in here.get_resource_names():
...     resource = here.get_resource(name)
...     handler = Handler.build_handler(resource)
...     print name, handler
...
chapter6 <itools.handlers.Folder.Folder object at 0x40538d4c>
chapter7 <itools.handlers.Folder.Folder object at 0x4032c0cc>
hello.txt <itools.handlers.Text.Text object at 0x40536f4c>
hello.xhtml <itools.xml.XHTML.Document object at 0x4053b70c>
```

Note that `build_handler` is a class method. In the example above we have called it through the most abstract handler class: `Handler`, which is the root of the inheritance tree. But it is also possible to call it with another handler class:

```
>>> from itools.xml import XML
>>>
>>> resource = here.get_resource('hello.xhtml')
>>> XML.Document.build_handler(resource)
<itools.xml.XHTML.Document object at 0x405c6ecc>
```

There is an important difference between calling `build_handler` from one or another class: the set of possible handler classes to use is restricted to all the sub-classes of the choosen handler class. For example, if we pass a plain text file to `XML.Document.build_handler`, it will fail:

```
>>> resource = here.get_resource('hello.txt')
>>> XML.Document.build_handler(resource)
Traceback (most recent call last):
  [...]
xml.parsers.expat.ExpatError: syntax error: line 1, column 0
```

5.6.1 The `get_handler` shorthand

There is a short way to load a handler (instead of loading first the resource and then building the handler explicitly), the function `get_handler`:

`get_handler(uri)`
- Loads the resource at the given uri, tries to guess its mimetype by different meanings (name extension, etc.), searches for a suitable handler class in the registry, builds and returns the handler for the resource.

Compare the explicit way seen before:

```
>>> from itools.resources import get_resource
>>> from itools.handlers.Handler import Handler
>>>
>>> resource = get_resource('http://example.com')
>>> handler = Handler.build_handler(resource)
```

With the shorthand:

```
>>> from itools.handlers import get_handler
>>>
>>> handler = get_handler('http://example.com')
```

Chapter 6

Writing file handler classes

The chapter before we have learnt about file handlers and how to use them, now we are going to learn how to write our own handler classes, what by the way will help to solidify the concepts seen before.

The explanation will be driven by an example: we are going to write a task tracker. The code can be found in the directory `examples/chapter6`.

6.1 Functional scope

Lets start by defining the functional scope of our task tracker. It is going to be very simple, it will be a collection of tasks where every task will have three fields:

- *title*, a short sentence describing the task.
- *description*, a longer description detailing the task.
- *state*, it may be *open* (if the task has not been finished yet), or *closed* (if the task has been finished).

The task tracker will provide an API to manipulate the collection of tasks: create a new task, see either the open or the closed tasks, and close a task.

6.2 The file format

Now that we know what we want to do, we have to decide where and how the information will be stored.

We will keep the tasks in a single text file, with a format somewhat similar to the one used by the standards *vCard* and *iCal*, for example:

```
title:Re-write the chapter about writing handler classes.
description:A new chapter that explains how to write file handler
 classes must be written, it should go inmediately after the chapter
 that introduces file handlers.
state:closed

title:Finish the chapter about folder handlers.
```

```
description:The chapter about folder handlers needs much more work.
 For example the skeleton of folder handlers must be explained.
state:open
```

Each task is separated from the next one by a blank line. Every field starts by the field name follewed by the field value, both separated by a colon. If a field value is very long it can be written in multiple lines, where the second and next lines start by an space.

This very same file can be found in the examples directory with the name `itools.tt`. Using our own filename extension (`tt`) will prove useful, as we will see later.

6.3 De-serialization

The first draft of our handler class will be able to load (de-serialize) the resource into a data structure on memory.

```
from itools.handlers.Text import Text


class Task(object):
    def __init__(self, title, description, state='open'):
        self.title = title
        self.description = description
        self.state = state


class TaskTracker(Text):

    def _load_state(self, resource=None):
        # Load the resource as a unicode string
        Text._load_state(self, resource)
        # Split the raw data in lines.
        state = self.state
        lines = state.data.splitlines()
        # Append None to signal the end of the data.
        lines.append(None)
        # Free the un-needed data structure, 'state.data'
        del state.data

        # Initialize the internal data structure
        state.tasks = []
        # Parse and load the tasks
        fields = {}
        for line in lines:
            if line is None or line.strip() == '':
                if fields:
                    task = Task(fields['title'],
                                fields['description'],
                                fields['state'])
                    state.tasks.append(task)
                    fields = {}
            else:
                if line.startswith(' '):
                    fields[field_name] += line
                else:
                    field_name, field_value = line.split(':', 1)
                    fields[field_name] = field_value
```

First, our handler class `TaskTracker` inherits from the handler class `Text`, because it is intended to manage a text file.

The method `_load_state` is the one to implement, it is responsible to deserialize the resource and build a data structure on memory that represents it.

The first thing it does is to call the parent's `_load_state` method, which will read the resource's data and get a unicode string from it, which will be stored in the `self.state.data` variable. This is an intermediary representation, which will be discarded.

The rest of the method process the raw data and builds a list of instances of the class `Task` into the variable `self.state.tasks`.

Lets try it:

```
>>> from pprint import pprint
>>> import textwrap
>>> from itools.resources import get_resource
>>> from TaskTracker import TaskTracker
>>>
>>> resource = get_resource('itools.tt')
>>> task_tracker = TaskTracker(resource)
>>>
>>> pprint(task_tracker.state.tasks)
[<TaskTracker.Task object at 0xb7aebd4c>,
 <TaskTracker.Task object at 0xb7aebe6c>]
>>>
>>> task = task_tracker.state.tasks[0]
>>> print task.title
Re-write the chapter about writing handler classes.
>>> print textwrap.fill(task.description)
A new chapter that explains how to write file handler classes must be
written, it should go inmediately after the chapter that introduces
file handlers.
>>> print task.state
closed
```

6.4 Serialization

Now we are going to write the other half, the serialization process, just adding the `to_unicode` method to the `TaskTracker` class:

```
    def to_unicode(self, encoding='UTF-8'):
        lines = []
        for task in self.state.tasks:
            lines.append(u'title:%s' % task.title)
            description = u'description:%s' % task.description
            description = textwrap.wrap(description)
            lines.append(description[0])
            for line in description[1:]:
                lines.append(u' %s' % line)
            lines.append(u'state:%s' % task.state)
            lines.append('')
        return u'\n'.join(lines)
```

Note that we implement the `to_unicode` method instead of `to_str` because this is a text handler. The `encoding` parameter is not used in our example because the file format does not specifiy the encoding within its content, but it must be declared anyway.

Lets try our new code:

```
>>> print task_tracker.to_str()
title:Re-write the chapter about writing handler classes.
description:A new chapter that explains how to write file handler
 classes must be written, it should go inmediately after the chapter
 that introduces file handlers.
state:closed

title:Finish the chapter about folder handlers.
description:The chapter about folder handlers needs much more work.
 For example the skeleton of folder handlers must be explained.
state:open
```

We could have tried `to_unicode` to get a similar output, except that then the result would be a unicode string instead of a byte string.

6.5 The API

Now it is time to write the API to manage the tasks, here is an excerpt:

```
def add_task(self, title, description):
    task = Task(title, description)
    self.state.tasks.append(task)


def show_open_tasks(self):
    for id, task in enumerate(self.state.tasks):
        if task.state == 'open':
            print 'Task #%d: %s' % (id, task.title)
            print
            print textwrap.fill(task.description)
            print
            print


def close_task(self, id):
    task = self.state.tasks[id]
    task.state = u'closed'
```

The first method, `add_task` creates a new task, whose state will be *open*. The method `show_open_tasks` prints the list of open tasks with a human readable format (we could write a method that returns HTML instead, to use our task tracker on the web). Finally, the method `close_task` closes the task.

6.6 The skeleton

The skeleton for our task tracker should be empty, but to illustrate this feature we are going to implement an skeleton with one dummy task:

```
        def get_skeleton(self):
            return 'title:Read the docs!\n' \
                   'description:Read the itools documentation, it is\n' \
                   ' so gooood.\n' \
                   'state:open\n'
```

To exercise the whole thing we are going to create a new task tracker, we will close the first task, add a new one, and look what we have.

```
    >>> from TaskTracker import TaskTracker
    >>>
    >>> task_tracker = TaskTracker()
    >>> task_tracker.show_open_tasks()
    Task #0: Read the docs!

    Read the itools documentation, it is so gooood.


    >>> task_tracker.close_task(0)
    >>> task_tracker.add_task('Join itools!',
    ...   'Subscribe to the itools mailing list.')
    >>> task_tracker.show_open_tasks()
    Task #1: Join itools!

    Subscribe to the itools mailing list.
```

Now, don't forget to save the task tracker in the file system, so you can come back to it later:

```
    >>> from itools.handlers import get_handler
    >>>
    >>> tmp = get_handler('/tmp')
    >>> tmp.set_handler('most_important_things_in_live.tt', task_tracker)
    >>> tmp.save_state()
```

6.7 Register

However:

```
    >>> task_tracker = tmp.get_handler('most_important_things_in_live.tt')
    >>> print task_tracker
    <itools.handlers.File.File object at 0xb7c00f0c>
```

It would be nice if the code above worked. To achieve it we will associate the new mimetype `text/x-task-tracker` to the file extension `tt`, we will tell our handler class is able to manage that mimetype with the variable class `class_mimetypes`, and we will register our handler class to its parent:

```
import mimetypes
mimetypes.add_type('text/x-task-tracker', '.tt')


class TaskTracker(Text):

    class_mimetypes = ['text/x-task-tracker']

    [...]


Text.register_handler_class(TaskTracker)
```

And *voilà*:

```
>>> task_tracker = tmp.get_handler('most_important_things_in_live.tt')
>>> print task_tracker
<TaskTracker.TaskTracker object at 0xb7af084c>
```

The full code can be found in `examples/chapter6/TaskTracker.py`.

Chapter 7

Folder handlers

A handler that deserves its particular chapter is the default handler for folders.

7.1 Folder's state

If the content of a file resource is a byte string, the content of a folder resource is a set of resources, each one identified by a name. Hence, the state of a folder handler is a mapping fron handler name to handler instance.

But imagine a folder that contains many big files. To load its state would mean to load all the files it contains, what is unacceptable from a performance point of view. This is the reason the folder handler implements lazy load, this is to say, it loads a handler only when it is needed.

The folder state is stored in the variable `cache`, because it behaves like a cache. See:

```
>>> examples = get_handler('examples')

[some operations later]

>>> pprint(examples.state.cache)
{'.arch-ids': None,
 'chapter8': <itools.handlers.Folder.Folder object at 0xb77b2fac>,
 'chapter9': None,
 'hello.txt': <itools.handlers.Text.Text object at 0xb77b2bcc>,
 'hello.xhtml': None,
 'task_tracker0': None,
 'task_tracker1': None,
 'task_tracker2': None}
```

As the code above shows, a handler that has not yet been loaded appears in the cache with the `None` value.

7.2 The API

The programming interface for folder handlers remembers the one of folder resources, where there we spelled `get_resource` here we will write `get_handler`. Details follow:

`get_handler(path)`
- Returns a handler for the resource at the given path. It will use the available handler class that better matches the resource mimetype.

`get_handler_names(path='.')`
- Returns a list with the names of all the handlers in the given path.

`get_handlers(path='.')`
- Returns the handlers in the given path (it is a generator).

`has_handler(path)`
- Returns `True` if there is a handler in the given path, `False` otherwise.

`set_handler(path, handler)`
- Adds the given handler to the given path. Actually what is added is the resource associated to the handler.

`del_handler(path)`
- Removes the handler at the given path (i.e. the associated resource).

7.3 Example

As the proverb says, *a code snippet is worth more than one thousand words*:

1. First we build a handler for the temporary directory:

```
>>> from itools.handlers import get_handler
>>>
>>> tmp = get_handler('/tmp')
>>> tmp
<itools.handlers.Folder.Folder object at 0x40652dec>
>>> tmp.resource
<itools.resources.file.Folder instance at 0x406acacc>
```

2. Second, we create a new HTML handler:

```
>>> from itools.xml import HTML
>>> hello = HTML.Document(title='Hello World')
>>> hello.resource
<itools.resources.memory.File instance at 0x405e49cc>
```

   Note that the associated resource is built on the fly and lives in memory.

3. Third, we set the HTML handler to the temporary folder:

```
>>> tmp.set_handler('hello.html', hello)
```

   What this actually does is to add the file `hello.resource` (which lives in memory) to the folder `tmp.resource` (which is on the file system); this is to say, it creates a new file in the file system at `/tmp/hello.html`.

4. Finally, we get the handler we just added:

   ```
   >>> hello = tmp.get_handler('hello.html')
   >>> hello.resource
   <itools.resources.file.File instance at 0x40638c6c>
   ```

   Note that the handler `hello` we have built in these last lines manages a resource that lives in the file system.

7.4 The handler tree

Folders allow to classify files, hence giving a tree structure to our data. Every handler has two attributes, `parent` and `name`, they tell us where the handler is in the handler tree:

```
>>> hello.parent
<itools.handlers.Folder.Folder object at 0x403ebb2c>
>>> hello.name
'hello.html'
>>> hello.parent is tmp
True
>>> print tmp.parent
None
>>> print tmp.name

>>>
```

Based on these two attributes handlers provide the following API:

`get_abspath()`
- Returns the absolute path from the tree root to the `self` handler.

`get_root()`
- Returns the handler for the root of the tree.

`get_pathtoroot()`
- Returns a relative path from `self` to the tree root (e.g. `../../..`).

`get_pathto(handler)`
- Returns a relative path from `self` to the given handler (which is supposed to be in the same tree), for example: `../../zoo/lion`.

`traverse()`
- This method allows to traverse the handler tree below this folder. It is a generator which returns a handler at a time, starting by this folder.

`acquire(name)`
- If the current handler is a folder and contains a resource with the given name, then return a handler for it; otherwise look at the parent folder, and recursively to the root tree. This method actually shows how to implement *acquisition*.

Chapter 8

eXtensible Markup Language

The purpose of this chapter is to explain the XML services provided by `itools`, which can be found in the sub-package `itools.xml`.

8.1 The parser

The first layer is the event driven parser implemented by `itools.xml.parser`, which is a wrapper around the `expat`[1] parser.

With `expat` you need a function for every event you want to manage, so for example, if you want to deal just with elements, comments and text nodes, you will need four functions (the start element, end element, comment and character data handlers), plus the main function that sets up the parser. State is typically shared across event handlers through instance variables. The `expat` approach makes it relatively hard to follow the program flow.

With `itools.xml.parser` all the code is within a single function, and state is stored in local variables. Let's see a dummy example:

```
from itools.xml import parser

for event, value, line_number in parser.parse(data):
    if event == parser.START_ELEMENT:
        namespace, prefix, local_name = value
        print 'START ELEMENT:', local_name
    elif event == parser.END_ELEMENT:
        namespace, prefix, local_name = value
        print 'END ELEMENT:', local_name
    elif event == parser.TEXT:
        print 'TEXT', value
```

The example above just prints a message to standard output each time the start of an element, the end of an element or a text node is found.

[1]http://expat.sourceforge.net/

The parser returns a list of events, where every event is a tuple of three valuese: the event type, the value (which depends on the event type) and the line number. The types of events implemented are:

Event	Value
`XML_DECLARATION`	(xml version, encoding, standalone)
`DOCUMENT_TYPE`	(name, system id, public id, has internal subset)
`START_ELEMENT`	(namespace uri, prefix, local name)
`END_ELEMENT`	(namespace uri, prefix, local name)
`ATTRIBUTE`	(namespace uri, prefix, local name, value)
`COMMENT`	value
`TEXT`	value

As you can appreciate only a subset of the XML standard is supported, though it is all what most users need.

Another impotant thing to note is that all values returned are normal strings, not unicode strings. It is not the job of the low level parser to deserialize the values.

8.2 Namespaces

From the table above you can see `itools.xml` provides support for XML namespaces. The event values for elements and attributes are tuples, where the first two components are the *namespace uri* and the *prefix*. If an element or attribute is not attached to a namespace, the *uri* and the *prefix* will be `None`.

The module `itools.xml.namespaces` provides a registry for namespace handlers, and an abstract class which defines the programming interface and provides a default behaviour for subclasses. The table below defines this programming interface:

`class_uri`
- The uri that uniquely identifies this namespace.

`class_prefix`
- The recommended prefix for the namespace. For example "`dc`" for Dublin Core. By default it is `None`.

`get_element_schema(name)`
- Returns the schema for the given element name.

`get_attribute_schema(name)`
- Returns the schema for the given attribute name.

The schema is a dictionary whose keys and values are somewhat arbitrary, they will depend on what you need. Though, it usually includes the type of the attribute or element, which is used to deserialize and serialize the values. For example, consider a link within an XHTML document, like:

```
<a href="http://www.example.com" title="Example" />
```

The `href` attribute should be loaded as a URI reference, and `title` should be a unicode string. The excerpt below will do the job:

```
from itools.xml import parser, namespaces
from itools import xhtml

for event, value, line_number in parser.parse(data):
    if event == parser.ATTRIBUTE:
        namespace_uri, prefix, local_name, value = value
        namespace = namespaces.get_namespace(namespace_uri)
        schema = namespace.get_attribute_schema(local_name)
        value = schema['type'].decode(value)
        print local_name, schema['type']
        print repr(value)
        print
```

The output when running this code is:

```
href <class 'itools.handlers.IO.URI'>
<itools.uri.generic.Reference object at 0xb7a8c8ac>

title <class 'itools.handlers.IO.Unicode'>
u'Example'
```

For examples about how to define your own namespace handlers see the Dublin Core (`itools.xml.DublinCore`) and XHTML (`itools.xhtml.XHTML`) implementations.

8.3 Documents

If `itools.xml.parser` provides an event driven parser, whose function is similar to SAX[2], we do have too an equivalent for DOM[3]. And of course it takes the form of a resource handler.

The handler class `itools.xml.XML.Document` loads the document into memory as a tree of nodes, with some global attributes. Let's inspect an example:

```
>>> from itools.handlers import get_handler
>>> import itools.xml
>>>
>>> doc = get_handler('examples/chapter8/hello.xml')
>>> doc
<itools.xml.XML.Document object at 0x4064466c>
>>> print doc.xml_version
1.0
>>> print doc.standalone
-1
>>> print doc.document_type
None
>>> print doc.root_element
<itools.xml.XML.Element object at 0xb7a3272c>
```

The code above shows the four attributes that keep the document's state:

[2]http://XXX
[3]http://XXX

`xml_version`
- The XML version of the document, usually it is `1.0`.

`standalone`
- Possible values are: `1` if the document was declared standalone, `0` if it was declared not to be standalone, or `-1` if the standalone clause was omitted.

`document_type`
- If the document lacks a document type declaration this attribute will be `None`. If the document type was specified this attribute will be a tuple with four values: the name, the system id, the public id and a boolean that tells wether the document contains an internal declaration subset.

`root_element`
- An instance of the `XML.Element` class, the root of the DOM-like tree that represents the XML data, and that we will study later with more detail.

The API for the documents is rather simple:

`get_root_element()`
- Returns the root element.

`traverse()`
- A generator that traverses the XML tree in pre-order, and returns each time a node. It is a shorthand for `root_element.traverse()`.

`traverse2()`
- A more powerful version of `traverse`. It is a shorthand for `root_element.traverse2()`.

8.3.1 Inspecting the tree

Coming back to the example, the `examples/hello.xml` file's content is:

```
<?xml version="1.0" encoding="UTF-8"?>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <title>Hello world</title>
    <!-- Changed by: , 02-Jun-2004 -->
  </head>
  <body>
  </body>
</html>
```

The method `traverse` lets us to easily inspect the tree nodes:

```
>>> for node in doc.traverse():
...     print repr(node)
...
<itools.xml.XML.Element object at 0xb7a3316c>
```

```
u'\n  '
<itools.xml.XML.Element object at 0xb7a331ac>
u'\n    '
<itools.xml.XML.Element object at 0xb7a332ac>
u'\n    '
<itools.xml.XML.Element object at 0xb7a331ec>
u'Hello world'
u'\n    '
<itools.xml.XML.Comment object at 0xb7a335cc>
u'\n  '
u'\n  '
<itools.xml.XML.Element object at 0xb798574c>
u'\n  '
u'\n'
```

Here we see the three kind of nodes currently supported: elements, comments and text nodes.

8.3.2 Elements

Of the three kinds of nodes, elements are the most important and complex. Element nodes give the tree structure, as they are the only ones that may have children. If you inspect an element you will see the following attributes:

`namespace`
- The XML namespace of the element (it will be None for a bare element).

`prefix`
- The prefix used for the element's XML namespace.

`name`
- The name of the element.

`attributes`
- A dictionary mapping from the tuple XML namespace and local name to the attributes value.

`prefixes`
- A mapping from XML namespace to prefix (used to get the qualified name of an attribute).

`children`
- A list with the children of the element (elements, comments and text nodes).

And here is the API:

`get_qname()`
- Returns the qualified name of the element.

`copy()`
- Returns a clone of the element.

`set_attribute(namespace, local_name, value, prefix=None)`
- Sets the given attribute.

`get_attribute(namespace, local_name)`
- Returns the attribute value for the given XML namespace and local name.

`has_attribute(namespace, local_name)`
- Returns a boolean value, true if the element contains a value for the given XML namespace and local name, false otherwise.

`get_attributes()` - Is a generator that returns a three value tuple each time, the values are: XML namespace uri, local name, value.

`get_attribute_qname(namespace, local_name)`
- Returns the qualified name for the given XML namespace and local name.

`set_element(element)`
- Appends the given element to the list of children.

`set_comment(comment)`
- Appends the given comment to the list of children.

`set_text(text)`
- Appends the given text node (a unicode value) to the list of children.

`get_elements(name=None)`
- Returns a list with all the element nodes whose name is the given name; if the parameter `name` is not given, then return all the element nodes.

`traverse()`
- A generator that traverses the element's children in pre-order, and returns each time a node.

`traverse2()`
- A more powerful version of `traverse`.

8.4 XHTML

To finish the chapter we are going to give a glance to the implementation of XHTML provided by `itools.xhtml`.

To drive the explanation here is the document source we will use as example (see `examples/hello.xhtml`):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
```

```
    <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
      <title>Hello world</title>
      <!-- Changed by: , 02-Jun-2004 -->
    </head>
    <body>
    </body>
  </html>
```

The only two differences between this file and the example we saw at the beginning (`examples/chapter8/hello.xml`) are the *document type* declaration and the XML namespace declaration. The result is `get_handler` returns an XHTML document instead of an XML document:

```
>>> doc = get_handler('examples/hello.xhtml')
>>> doc
<itools.xhtml.XHTML.Document object at 0xb7a2ec4c>
>>>
>>> for node in doc.traverse():
...     print repr(node)
...
<itools.xhtml.XHTML.BlockElement object at 0xb79884ec>
u'\n  '
<itools.xhtml.XHTML.HeadElement object at 0xb798852c>
u'\n    \n    '
<itools.xhtml.XHTML.BlockElement object at 0xb79885ac>
u'Hello world'
u'\n    '
<itools.xml.XML.Comment object at 0xb798856c>
u'\n  '
u'\n  '
<itools.xhtml.XHTML.BlockElement object at 0xb798858c>
u'\n  '
u'\n'
```

Now, the element nodes are not any more instances of `XML.Element`, but instances of `XHTML.Element`, which extends the generic API with the methods:

`is_inline()` - Returns `True` if the element is an inline element, `False` otherwise.
`is_block()` - Returns `True` if the element is a block element, `False` otherwise.

Ok, not too much[4], but enough to give an idea of the power of `itools.xml`. In the next chapter we will see a much more compelling example of what can be done with `itools.xml`, the **S**imple **T**emplate **L**anguage.

[4]These two methods, `is_inline` and `is_block`, are actually really useful. They are used by the message extraction algorithm, a fundamental brick of the internationalization and localization services provided by `itools`.

Chapter 9

Simple Template Language

STL is a template language. It is implemented as an XML namespace handler, taking advantage of the underlying infrastructure provided by `itools.xml`.

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

9.1 A descriptive language

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.

This way the logic and the presentation are really separated.

There are always two sides: the *template*, which represents the presentation side; and the *namespace*, the logic side.

In five words: **STL** is a *descriptive* language.

9.1.1 The template

For example, look at the template below (`examples/chapter9/template.xml`):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html
     PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:stl="http://xml.itools.org/namespaces/stl">
  <head></head>
  <body>
    <h1 stl:content="title" />
  </body>
</html>
```

Note the declaration of the *stl* namespace, it is mandatory.

When this template will be processed, the content of the `<h1>` tag will be replaced by the value of the variable `title`. But, where will we find `title`? Solution: in the namespace.

9.1.2 The namespace

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

```
>>> from itools.handlers import get_handler
>>> from itools import xml
>>>
>>> template = get_handler('template.xml')
>>>
>>> template
<itools.xml.XML.Document object at 0x405ea38c>
```

Note that so far nothing delates the **STL** presence, but inspecting the `template` object shows that the *stl* namespace handler has been loaded:

```
>>> template.stl
<itools.xml.STL.STL object at 0x406516cc>
```

Ok, it is time to build the namespace and process the template:

```
>>> namespace = {'title': 'hello world'}
>>> print template.stl(namespace)
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html
     PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns:stl="http://xml.itools.org/namespaces/stl"
      xmlns="http://www.w3.org/1999/xhtml">
  <head></head>
  <body>
    <h1>hello world</h1>
  </body>
</html>
```

As this example shows, the value of the variable `title` is looked within the namespace passed as parameter to `template.stl`.

9.2 Example: Task Tracker

Now we are going to illustrate **STL** with a more complex example. Building up on the Task Tracker from the Chapter 6, we are going to write a method that produces an HTML page showing all the tasks.

First, the template (see `examples/chapter9/TaskTracker_view.xml`):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html
     PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:stl="http://xml.itools.org/namespaces/stl">
  <head></head>
  <body>
    <h2>Task Tracker</h2>
```

```
      <div stl:repeat="task tasks">
        <h4>
          #<stl:block content="task/id" />:
          <stl:block content="task/title" />
          (<em stl:content="task/state" />)
        </h4>
        <p stl:content="task/description" />
      </div>
    </body>
  </html>
```

The first new thing this example shows is the `repeat` statement. While `stl:content` expects a string as the value, `stl:repeat` expects a sequence. When this template is processed, the XML output will contain as many `<div>` elements as tasks are in the `tasks` variable. Within the div element, in each iteration over the `tasks` sequence, the variable `task` 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>` tags are automatically removed.

Finally, look at the expression `task/id` or `task/title`, it shows the *slash* operator, which lets to traverse namespaces. So the variable `task` is expected to be a mapping (e.g. a dictionary), and `id` a key in that mapping, whose value is a string.

The Python side

Now let's see the Python code (see `examples/chapter9/TaskTracker.py`). Basically we have added the method `view` to the class `TaskTracker`:

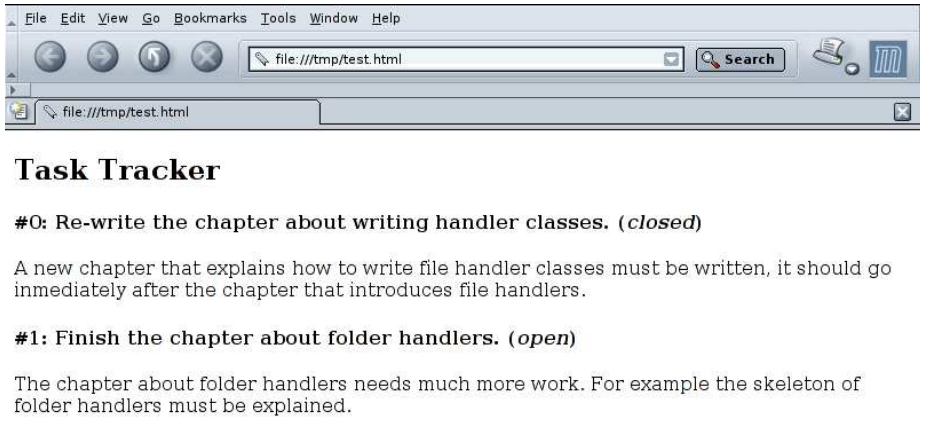
```
def view(self):
    # Load the STL template
    handler = get_handler('TaskTracker_view.xml')

    # Build the namespace
    namespace = {}
    namespace['tasks'] = []
    for i, task in enumerate(self.state.tasks):
        namespace['tasks'].append({'id': i,
                                   'title': task.title,
                                   'description': task.description,
                                   'state': task.state,
                                   'is_open': task.state == 'open'})

    # Process the template and return the output
    return handler.stl(namespace)
```

To try the code run the Python interpreter and type:

```
>>> from itools.handlers import get_handler
>>> import TaskTracker
>>>
>>> task_tracker = get_handler('itools.tt')
>>> print task_tracker.view()
```



Task Tracker

#0: Re-write the chapter about writing handler classes. (*closed*)

A new chapter that explains how to write file handler classes must be written, it should go inmediately after the chapter that introduces file handlers.

#1: Finish the chapter about folder handlers. (*open*)

The chapter about folder handlers needs much more work. For example the skeleton of folder handlers must be explained.

Figure 9.1: The task tracker view

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html
     PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns:stl="http://xml.itools.org/namespaces/stl"
      xmlns="http://www.w3.org/1999/xhtml">
  <head></head>
  <body>
    <h2>Task Tracker</h2>
    <div>
      <h4>
        #0:
        Re-write the chapter about writing handler classes.
        (<em>closed</em>)
      </h4>
      <p>A new chapter...
```

The Figure 9.1 shows how the HTML looks with a browser.

9.3 Language overview

So far we have seen `stl:content` and `stl:repeat`. Below is the summary with all the **STL** statements:

`content`=“*expression*”
- Replaces the element’s content by the result of evaluating the given **STL** expression.

`attributes`=“*name expression*[; *name expression*]*”
- For every pair “*name: expression*”, replace the the value of the attribute *name* by the result of evaluating *expression*.

`if`=“[not]*expression*”
- If the given expression evaluates to `True`, do nothing; if evaluates to `False`, remove the XML element.

`repeat`=“*name expression*”
- The given expression is expected to be a sequence or an iterator. For every item in *expression*, create an element and add the item to the namespace stack before processing the element.

9.3.1 Expressions

The **STL** expressions are pretty simple, their syntax is:

name[/name]*

That is, a sequence of names separated by slashes. The semantics is:

1. Look the first name in the namespace stack.
2. If there are more names left, the last value found must be a namespace, then look the next name in that namespace.

   Iterate until the last name is consumed.
3. Once the end of the sequence is reached, we will have a value. If the value is callable, then call it to get a new value.
4. Finally, we should have a value that is either a string, a boolean or a sequence, depending on which statement (`content`, `repeat`, etc.) the expression is being used with.

Chapter 10

Internationalization and Localization

The task tracker from the previous chapter provides a user interface in only one language. The purpose of this chapter is to explain how to build applications that provide a user interface in many languages. We will illustrate this building up on the task tracker example.

See the file layout of the already multilingual task tracker:

```
Makefile
TaskTracker.py
TaskTracker_view.xml.en
locale/
  locale.pot
  en.po
  es.po
```

There are two things to note. First the `locale` directory: it is a database which keeps the translations for the text messages of the user interface. The translations are stored in *PO*[1] files, one per language; in our example there is one for English (`en.po`) and another for Spanish (`es.po`). Each *PO* file keeps the source messages, usually written in English, and their translations (because the source language is English, the `en.po` file will be usually empty).

The second thing to note is the `Makefile`, it will help us to automatize the localization and build processes. The Figure 10.1 shows these two processes.

10.1 Internationalization

Internationalization:

> *the operation by which a program, or a set of programs turned into a package, is made aware of and able to support multiple languages.*

This is to say, if you have a monolingual product like the task tracker from the previous chapter, you will need to make some changes to the code, so it becomes

[1] XXX

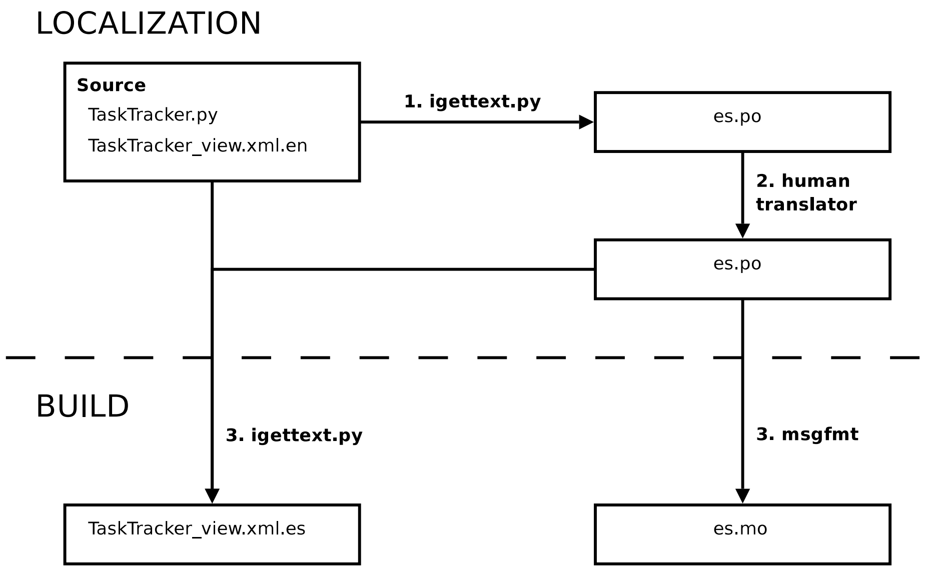


Figure 10.1: The localization process

able to deal with multiple languages. This process is called *internationalization*. Of course, you may also write international software since the beginning.

The text messages to translate are stored in the XML templates and the Python code, so these are the two things to internationalize.

10.1.1 Python code

To start, our task tracker must be *domain aware*:

```
from itools.gettext.domains import DomainAware

class TaskTracker(Text, DomainAware):

    class_mimetypes = ['text/x-task-tracker']
    class_domain = 'task tracker'
```

We use here the word *domain* because it is standard in the internationalization jargon. Here by a *domain* we understand a database that keeps message translations for one or more languages. A *domain aware* class is one that is implicitly associated to a domain and has an API to access that domain.

Every application and library will have a unique domain name, this will allow us to explicitly refer to a given domain to retrieve translations from it. The domain name is specified with the class variable `class_domain`.

By the way, we need to register the domain:

```
from itools import get_abspath
from itools.gettext.domains import register_domain
```

```
domain_path = get_abspath(globals(), '../locale')
register_domain(TaskTracker.class_domain, domain_path)
```

The class `DomainAware` provides the application programming interface we will use within the Python code:

`get_languages()`
- Returns a list with the language codes of our application. By default it is not implemented, so it must be overriden by subclasses.

`select_language(languages=None)`
- Chooses and returns a language from the given list of languages. If the languages are not given, the method `get_languages` will be called to get them. The default implementation uses the environment variable `LANGUAGE`.

`gettext(message, language=None, domain=None)`
- Returns the translation for the given message in the given language, from the given domain. If there is not a translation the input message is returned. If no language is given the method `select_language` will be called to choose one from the languages available in the domain. By default the domain used is the one specified in the class variable `class_domain`.

Now we just need to replace every text string in the source code by a call to `gettext`, e.g `u'hello'` becomes `self.gettext(u'hello')`. In our example it would be:

```
def show_open_tasks(self):
    for id, task in enumerate(self.state.tasks):
        if task.state == 'open':
            print self.gettext(u'Task #%(id)d: %(title)s') \
                  % {'id': id, 'title': task.title}
    ...
```

Note that we have used named arguments for the formatted string instead of positional arguments. This is because the position of the arguments may be different in another language.

10.1.2 Templates

The technique we use with `itools` requires to have one template per language, being English the master language. Each template must have a distinct name, we will achieve it by appending the language code to the end of the file:

```
TaskTracker_view.xml.en
TaskTracker_view.xml.es
```

Note that only the template in English belongs to the source, the others will be automatically built with the help of `itools`.

Now we are going to make some changes to the `TaskTracker` class. First we must override the method `get_languages`:

```
def get_languages(self):
    return ['en', 'es']
```

These are the languages of our task tracker, English and Spanish. Now we must modify the `view` method to choose the right template:

```
def view(self):
    # Load the STL template
    language = self.select_language()
    handler = get_handler('TaskTracker_view.xml.%s' % language)
    ...
```

10.2 Localization

Localization:

> *the operation by which, in a set of programs already internationalized, one gives the program all needed information so that it can adapt itself to handle its input and output in a fashion which is correct for some native language and cultural habits.*

The source is ready, now starts the localization process, it consists of two steps:

1. Message extraction.
2. Human translation.

10.2.1 Message extraction

This step consists on the parsing of the source code, the extraction of the translatable messages they contain, and the building of a *PO* file which will be the input for the human translator.

This is done with the `GNU gettext`[2] tools, which are the standard in the Free Software world for software internationalization and localization; and the script `igettext.py` provided by `itools`.

But the process is automatized by the `Makefile` so the only thing we need to do is to type:

```
$ make po
```

This will parse the Python and XHTML source and update the PO files within the `locale` directory. See below an excerpt:

```
#: TaskTracker.py:117
#, python-format
msgid "Task #%(id)d: %(title)s"
msgstr ""

#: TaskTracker_view.xml.en:0
msgid "Task Tracker"
msgstr ""
```

[2]http://www.gnu.org/software/gettext/

Marking messages

The message extraction script will pick all text strings[3] from the Python source, and all text nodes from the source templates. Then it will split the messages into sentences, a process known as segmentation, it will be these sentences which will feed the message catalog.

This means that no explicit markup is required to signal a text string as translatable. This way we reduce the burden on the developer, who just needs to remember to use Python unicode strings for text, i.e. to type:

```
u'Hello world.'
```

instead of:

```
'Hello world.'
```

For the templates, the only rule is to write them properly. In particular, to use block elements as block elements, and inline elements as inline elements.

10.2.2 Human translation

The human translator receives the PO file, and must return the same PO file, but including the translations for every message. In our example the output should be:

```
#: TaskTracker.py:117
#, python-format
msgid "Task #%(id)d: %(title)s"
msgstr "Tarea #%(id)d: %(title)s"

#: TaskTracker_view.xml.en:0
msgid "Task Tracker"
msgstr "Gestor de tareas"
```

To do this work the translator don't needs to understand the file format, there are graphical tools available to do the job like:

- KBabel, `http://www.kde.org`
- poedit, `http://poedit.sourceforge.net`

10.3 Build

To close the process, we must generate the translated templates (`TaskTracker_view.xml.es`) and the *MO* files (`en.mo`, `es.mo`) that will be used in runtime by the `gettext` method.

To build the templates from the master template (`TaskTracker_view.xml.en`) and from the translated PO file (`es.po`), is done with the script `igettext.py`:

```
$ igettext.py --xhtml addressbook_view.xml.en es.po \
      > addressbook_view.xml.es
```

[3]Text strings are most often known as unicode strings in Python.

The MO file is a binary version of the PO file, and is built with the `msgfmt` tool.

But, again, the process is automatized by the `Makefile`, so the only thing we need to do is to type:

```
$ make bin
igettext.py --output=TaskTracker_view.xml.es \
  --xhtml TaskTracker_view.xml.en locale/es.po
msgfmt locale/en.po -o locale/en.mo
msgfmt locale/es.po -o locale/es.mo
touch bin
```

And *voilà*.

Chapter 11

Addendum

This documentation is work in progress. It has covered only about half the packages provided by `itools`. It remains to look at the workflow engine from `itools.workflow` and the index and search possibilities offered by `itools.catalog`.

And the topics we have talked about have only been exposed on the surface. The API provided by the different classes is actually bigger than what this document shows. The examples are minimal, and leave most possibilities to the reader's imagination. We haven't seen all the available handlers in detail.

Anyway, I hope you have found these tools interesting enough to get a closer look.

If you didn't, I have still a few words to add.

11.1 In the real world

In spite of the youth of `itools` (version `0.5` at the time of this writing), it is already being used in real production projects. Specially, it is the foundation of the **iKaaro**[1] Content Management System.

Some of the advantages it provides include:

- The whole web site data is stored as a tree of human readable files, instead of opaque persistent Python objects. It is possible to export a web site to the file system, to inspect and manipulate the tree, to make a tarball, to import it back to the ZODB.
- This approach encourages the use of standards, every time we need a new feature we look first if there is already a standard file format. The result is a highly portable, standards compliant application, easy to interact with other applications.
- It is possible to seamlessly integrate content from foreign sources and to make it appear as if they were in our web site. We are just at the surface of the possibilities.

But the most important, it provides a simple, high-productive framework to build our applications on, to reduce the development time to the minimum.

[1]http://www.ikaaro.org

11.2 Future works

Some of the stuff waiting on the pipe...

- Schemas. Not only for XML documents (XML Schema[2] support is on the works), but also other handlers, specially folders.
- Many handlers for standard file formats: iCal, vCard, TMX, XLIFF, etc.
- Make `itools.uri` 100% compliant with the RFC2396, maybe write a **P**ython **E**xtension **P**roposal to get it in the Standard Library.
- Support for new schemes, protocols and storages for `itools.resources`.
- Add the last bricks to `itools.i18n` to get an engine able to power high end translation memory systems.

11.3 Free Software

And, last but not least, `itools` is free software, released to the world under the terms and conditions of the GNU Lesser General Public License[3], developers own the copyright of the code they write, contributions are very welcomed.

[2] http://www.w3.org/XML/Schema
[3] http://www.gnu.org/copyleft/lesser.html

Appendix A

Coding style guide

This chapter describes the coding conventions used to write `itools`. If you ever contribute a patch to `itools`, be sure your code adheres to these rules.

Sometimes this guide contradicts the *PEP 8*[1] recommendations; in these cases this guide applies, in the context of `itools`.

A.1 Language and encoding

Code must be written in english. The preferred encoding is *UTF-8*, though others encodings are allowed.

A.2 Module structure

Each module is splitted in six sections:

1. encoding statement;
2. copyright notice;
3. license reference;
4. the module documentation string;
5. import statements;
6. the code itself.

Here we are going to describe the module's header, made up of the first five sections. The style for the code itself will described in the rest of this chapter.

The encoding

It is only requiered if it is any other encoding than ASCII. Example:

```
# -*- coding: UTF-8 -*-
```

[1]http://www.python.org/peps/pep-0008.html

The Copyright

After the encoding comes the copyright, whose structure is:

```
# Copyright (C) <years> <author name> <email>
                <years> <author name> <email>
                ...
```

The License

Right after the copyright statement comes a reference to the license. For `itools` it is the *LGPL*.

The module's documentation string

There should be a documentation string explaining what the module does (though it is better to have none than a dummy one).

The explanation about how to write a documentation string is out of the scope of this chapter, it is covered by the *PEP 257*[2].

Imports

The imports statements are at the top of the file (just after the legal statements and the module docstrings), though there may be imports within a function or method to avoid circular references.

Imports should be grouped, with the order being:

1. standard library imports
2. `itools` imports
3. other package package imports

Every group should be preceded by an introductory comment, like:

```
# Import from the Standard Library
...

# Import from itools
...
```

A.2.1 Example

For example, at the time of this writting the module `itools.handlers.Handler` starts by:

```
# -*- coding: ISO-8859-1 -*-
# Copyright (C) 2003-2004 Juan David Ibáñez Palomar <jdavid@itaapy.com>
#
# This library is free software; you can redistribute it and/or
# modify it under the terms of the GNU Lesser General Public
# License as published by the Free Software Foundation; either
# version 2.1 of the License, or (at your option) any later version.
```

[2]http://www.python.org/peps/pep-0257.html

```
#
# This library is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
# Lesser General Public License for more details.
#
# You should have received a copy of the GNU Lesser General Public
# License along with this library; if not, write to the Free Software
# Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307  USA

"""
This module provides the abstract class which is the root in the
handler class hierarchy.
"""

# Import from the Standard Library
import datetime

# Import from itools
from itools import uri
from itools.resources import base
```

A.3 Format rules

Indentation: 4 spaces

Each indentation level must have four (4) spaces. Never use tabs.

Maximum line length: 80 characters

Lines should be 80 characters wide at most.

The preferred way of wrapping long lines is by using Python's implied line continuation inside parentheses, brackets and braces. If necessary, you can add an extra pair of parentheses around an expression, but usually using a backslash looks better. Make sure to indent the continued line appropriately.

One line, one statement

Don't put more than one statement on the same line:

	Good	Bad

1	`if x is True:`
	`    do_something()`
		`if x is True:  do_something()`

2	`do_one()`
	`do_two()`
	`do_three()`
		`do_one(); do_two(); do_three()`

3	`def get_area(x, y):`
	`    return x * y`
		`def get_area(x, y):  return x * y`

Blank lines

Separate classes with three blank lines. Separate methods and functions with two blank lines. There is also a blank line between the class definition and the first method definition.

Use blank lines in functions, sparingly, to indicate logical sections.

Whitespace in expressions and statements

Surround operators with one white space. Arithmetics operators may be the exception, for them it is possible to use either one space or none, whatever reads better. But never use more than one space.

Well, there is another exception, the sign "=" used in keyword arguments should not be surrounded by spaces. Type `Document(title="hello")` instead of `Document(title = "hello")`.

Never add spaces neither before nor after parentheses, brackets or braces. The only exception is for list comprehensions, where it is allowed to add a space after the opening bracket, and another space before the closing bracket.

The comma and colon must be followed by a space (or a new line), but never put a space before. The only exception is for one element tuples, where the comma must be immediately followed by the closing parentheses. The semicolon should never be used.

A.4 Comments

Comments must describe the code that follows them, and must be indented to the same level of that code. Inline comments are not allowed; this is to say, a comment always starts a new line.

A comment starts by a single `#` character followed by a space.

Comments must be written in good english (as good as the developer can write it). This means, for example, that the first letter must be capitalized.

A.5 Naming conventions

The names of *variables*, *classes*, *functions*, *methods* and *constants* are written with one or more english words. Most of the words used are *nouns*, *verbs*, and *adjectives*.

Abbreviations may be used, but in general it is preferred the complete word, for example, `language` instead of `lang`. When an abbreviation is not obvious, its meaning should be explained with a comment.

The allowed naming conventions are three:

lower_case_with_underscores All words are in lowercase and separated by an underscore. This convention is used for *variables*, *functions* and *methods*.

UPPER_CASE_WITH_UNDERSCORES All words are in uppercase and separated by an underscore. Used only for *constants*.

CapitalizedWords All words start by an uppercase, with the rest of the word in lowercase. Words are not separated by any character, the uppercase letters serve to visually distinguish when a new word starts. Used only for *classes.*

A.5.1 Class names

Class names are written in capitalized words. Typically they are made of nouns and/or adjectives.

A.5.2 Functions and methods

Functions and methods are written in lowercase with underscores.

They must start by a verb, and they should be followed by a complenent that clearifies what the funciont does. For example, it is better to spell `set_object` than just `set`.

A.5.3 Variables

Variables are written in lowercase with underscores. Most of the time they are nouns with or without adjectives.

One letter variables may be used in mathematical expressions, for sequence indexes, or in comprehensive lists:

```
>>> public_handlers = [x for x in handlers if x.state == 'public']
```

A.5.4 Constants

Constants are written in uppercase with underscores.

Appendix B

GNU arch

A control version system is a tool that keeps track of changes made in the code, when a change was made, by whom. It helps multiple developers to work on the same project, to merge the different changes they made in the same code base.

GNU arch[1] (also known as *tla*) is a modern and advanced control version system. It is the one we use to manage the `itools` source code.

There are many ways to work with *tla*, this appendix explains the one we use for `itools`.

The exposition is organized in three sections that detail:

1. How to keep track of the development of `itools`.
2. How to maintain private changes.
3. How to contribute your changes back to the main development tree.

B.1 Keeping track of `itools`

You may want to have the last bleeding edge features from `itools` in your system as soon as they are written, or to track how the development is going on. Then this section is for you.

B.1.1 Browsing the sources

To browse the `itools` archive tree through the web, just go the url below:

```
http://in-girum.net/cgi-bin/archzoom.cgi/jdavid@itaapy.com--public
```

B.1.2 Check out

To check out `itools` from the archive you need to install *tla*. Most distributions include it, for example, if you use Gentoo[2] just type:

```
$ sudo emerge tla
```

Once *tla* is installed, follow the steps described below.

[1] `http://www.gnu.org/software/gnu-arch/`
[2] http://www.gentoo.org

Set your id

```
$ tla my-id "Toto Bonaparte <toto@example.com>"
$ tla my-id
Toto Bonaparte <toto@example.com>
```

Register the official `itools` archive

```
$ tla register-archive jdavid@itaapy.com--public \
      http://in-girum.net/~jdavid/archives/public
$ tla archives
jdavid@itaapy.com--public
    http://in-girum.net/~jdavid/archives/public
```

Check out `itools`

```
$ cd ~/sandboxes
$ tla get jdavid@itaapy.com--public/itools--main--0.6 itools-0.6
$ cd itools-0.6
$ tla tree-version
jdavid@itaapy.com--public/itools--main--0.6
```

B.1.3 A session with *tla* and `itools`

Now, whenever you want to see if something has changed in `itools`, just type:

```
$ cd ~/sandboxes/itools-0.6
$ tla missing --summary
patch-80
    use Python's documentation to profile the catalog
patch-81
    fix XML error handling (hence better STL message errors)
```

The output shows the new patches available (if your code is up-to-date the output will be empty). Say you want to apply the patches, type:

```
$ tla update
[...]
```

B.1.4 Help

The Table B.1 summarizes the *tla* commands seen in this section. To learn about other commands use `tla help`, and for details about a command type:

```
$ tla <command> --help
```

B.2 Maintaining private changes

Now maybe you want to make some changes to `itools`. The wisest to do in this situation is to create a branch of `itools`, this will let you to easily update to the last version while keeping your changes.

The first step is to setup an archive (if you have already one you can skip to the next subsection).

`tla my-id`	- Print or change your id.
`tla register-archive`	- Change an archive location registration.
`tla archives`	- Report registered archives and their locations.
`tla get`	- Construct a project tree for a revision.
`tla tree-version`	- Print the default version for a project tree.
`tla missing`	- Print patches missing from a project tree.
`tla update`	- Update a project tree to reflect recent archived changes.
`tla help`	- Provide help with arch.

Table B.1: Basic *tla* commands

B.2.1 Create an archive

```
$ mkdir ~/{archives}
$ mkdir ~/{archives}/public
$ tla make-archive toto@example.com--public ~/{archives}/public
$ tla archives
jdavid@itaapy.com--public
    http://in-girum.net/~jdavid/archives/public
toto@example.com--public
    /home/toto/{archives}/public
```

Make it your default archive:

```
$ tla my-default-archive toto@example.com--public
$ tla my-default-archive
toto@example.com--public
```

B.2.2 Create a branch

With your own archive, it is time to create a branch of `itools`:

```
$ tla tag -S jdavid@itaapy.com--public/itools--main--0.6 itools--toto--0.6
  * creating category toto@example.com--public/itools
  * creating branch toto@example.com--public/itools--toto
  * creating version toto@example.com--public/itools--toto--0.6
  * Archive caching revision
```

So now you can replace the check-out from the main tree with a one from your own branch:

```
$ cd ~/sandboxes
$ rm -rf itools-0.6
$ tla get toto@example.com--public/itools--toto--0.6 itools-0.6
$ cd itools-0.6
$ tla tree-version
toto@example.com--public/itools--toto--0.6
```

B.2.3 Working with your branch

So, now you modify `itools` to add a new feature. Every change made to a file will be automatically detected by *tla*, but if your work includes new files or directories, or you have removed, renamed or moved a file or a directory, then you must tell *tla* about these changes, to do so use the commands below:

```
$ tla add <filename>
[...]
$ tla delete <filename>
[...]
$ tla move <old filename> <new filename>
[...]
```

Maybe you forgot to add a file, before committing is a very good idea to verify it with the command `tree-lint`:

```
$ tla tree-lint
[...]
```

This command looks at your project tree and tells you about files suspected to be source code that have non been added yet.

Ok, so you have finished working on this new cool feature and are willing to check it in your branch of `itools`. First, verify what you have changed:

```
$ tla changes
[...]
```

This command shows which files (and folders) have been modified, removed, added or moved. For a more detailed description, try:

```
$ tla changes --diffs | less
```

Take your time to examine the changes, maybe you forgot to remove a print statement? a close look at the output of `tla changes --diffs` will tell you.

Once you are sure everything is alright, it came the time to commit. First you have to write a log message:

```
$ vi ‘tla make-log‘
```

Within the editor, you should introduce a title that describes the changes you have done, and optionally a longer description. Once you are done, left the editor and type:

```
$ tla commit
$ tla revisions
[...]
patch-1
    add feature XXX
```

B.2.4 Merging from the main branch

Ok, so now the upstream version of `itools` is modified, how to merge the changes in your tree? easy:

```
$ cd ~/sandboxes/itools-0.6
$ tla star-merge -t jdavid@itaapy.com--public/itools--main--0.6
[...]
```

Beware, there may be conflicts that you must resolve.

Now, your project tree contains the changes from the upstream archive, you must commit them in your own archive. The log is written automatically by typing:

```
$ tla log-for-merge >> `tla make-log`
$ vi `tla make-log`
```

Within the editor there will be a description detailing the patchs that have been applied. So you just have to add the subject, something like "merging from the main tree". Once this is done just commit as usual:

```
$ tla commit
```

Summary

See the Table B.2 for a summary of the *tla* commands seen in this section.

B.3 Contributing your work to the main tree

To contribute your changes back to the main development branch you must make your branch available through internet. We assume the archive you have set-up is your local computer, so you have to create a mirror of your archive from your local computer to an internet server:

```
$ tla make-archive --listing --mirror toto@example.com--public \
      sftp://example.com/home/toto/{archives}/public
```

You must be sure your mirror can be accessed through an internet protocol, like HTTP or FTP. So other developers will be able to merge from your branch.

Now, whenever you make and commit a change, to share that change with the community, you have to synchronize your mirror:

```
$ tla archive-mirror
[...]
```

See the Table B.3 for a summary of the commands seen in this section.

`tla make-archive`
- Create a new archive directory.

`tla my-default-archive`
- Print or change your default archive.

`tla tag`
- Create a continuation revision (aka tag or branch).

`tla add`
- Add an explicit inventory id.

`tla delete`
- Remove an explicit inventory id.

`tla move`
- Move an explicit inventory id.

`tla tree-lint`
- Audit a source tree.

`tla changes`
- Report about local changes in a project tree.

`tla make-log`
- Initialize a new log file entry.

`tla commit`
- Archive a changeset-based revision.

`tla revisions`
- List the revisions in an archive version.

`tla star-merge`
- Merge mutually merged branches.

`tla log-for-merge`
- Generate a log entry body for a merge.

Table B.2: Maintaining private changes: summary

`tla archive-mirror`
- Update an archive mirror.

Table B.3: Maintaining private changes: summary

Index