

# itools Packaging

Juan David Ibáñez Palomar\*

jdavid@itaapy.com

December 20, 2007

## Contents

|          |   |          |
|----------|---|----------|
| <b>1</b> | <b>Introduction</b>                                       | <b>2</b> |
| <b>2</b> | <b>Anatomy of an <code>itools</code> based package</b>    | <b>2</b> |
| <b>3</b> | <b>The configuration file</b>                             | <b>2</b> |
| 3.1      | Example: <code>mypkg</code> . . . . .                     | 2        |
| 3.2      | Options . . . . .   | 3        |
| <b>4</b> | <b>The version number</b>                                 | <b>4</b> |
| <b>5</b> | <b>Git</b>  | <b>4</b> |
| <b>6</b> | <b>The build, install and release processes</b>           | <b>5</b> |
| 6.1      | The <code>isetup-build.py</code> script . . . . .         | 5        |
| <b>7</b> | <b>Multilingual packages</b>                              | <b>6</b> |
| 7.1      | The source and target languages . . . . .                 | 7        |
| 7.2      | The <code>isetup-update-locale.py</code> script . . . . . | 7        |
| 7.3      | Building a multilanguage package . . . . .                | 7        |
| <b>8</b> | <b>Tell <i>Git</i> to ignore non-source files</b>         | <b>8</b> |
| <b>9</b> | <b>The release procedure</b>                              | <b>8</b> |

---

\*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>

## 1 Introduction

As most other Python packages, `itools` uses the standard *Python Distribution Utilities* (a.k.a. *distutils*). Above them `itools` adds a thin layer to simplify the work of making up a new Python package.

This document describes this thin layer, and two related scripts from the *isetup* family: `isetup-build.py` and `isetup-update-locale.py`.

## 2 Anatomy of an `itools` based package

An `itools` based package has this structure:

```
__init__.py
setup.py
setup.conf
locale/
scripts/
test/
```

The `locale`, `scripts` and `test` folders are optional. The `locale` folder is only required if the package is going to be multilingual (see Section 7 for the details). The `scripts` folder is where we will put the scripts, if there are any. The `test` folder is for the unit tests.

One difference with normal Python packages, is that `itools` based packages have a more normalized structure.

## 3 The configuration file

With *distutils* the `setup.py` module defines the package. We believe that a Python module is not the most appropriate file format to define a package. For this purpose `itools` uses a configuration file, what reduces the `setup.py` module to a few lines of boilerplate:

```
setup.py:
    # Import from itools
    from itools.utils import setup

    if __name__ == '__main__':
        setup(globals())
```

### 3.1 Example: `mypkg`

A minimal configuration file must at least define the package name. But it is recommended to add a few description fields:

```
setup.conf:
    # The name of the package
    name = mypkg

    # Recommended metadata
    title = "This package is a test."
    url = http://www.example.com/
    author_name = "J. David Ibáñez"
    author_email = j david@itaapy.com
    license = "GNU General Public License (GPL) "
```

## 3.2 Options

Here we list the options currently supported by the configuration file:

`name` The package name is the only mandatory option.

`title` A short summary (one line) describing the package.

`url` The URL of the package, the home Web Site.

`author_name` The full name of the main author.

`author_email` The email address of the main author.

`license` The name of the license.

`description` A multi-line description of the package.

`packages` The list of sub-packages, if any.

`scripts` The list of scripts, if any.

`source_language` The source language of the package, generally `en` for English.

`target_languages` The list of human languages the package is translated to, other than the source language.

### Limits

It is true that as of today the configuration file does not allow to do everything, for example there is no way to define extension modules (modules written in C). When a feature not supported by the configuration file is required, we will need a more elaborate `setup.py` module than the boilerplate seen before.

The `itools` package itself is an example of a more complex package that requires a more elaborate `setup.py` module.

## 4 The version number

Note that the version number is not an option of the configuration file. We prefer to store it in the `version.txt` file, for instance:

```
version.txt:
    1.0.2
```

The first advantage of this approach is the possibility to automatize the generation of the version number with the help of external tools. This is what we do with *Git*<sup>1</sup> (see Section 5) and the `isetup-build.py` script (see Section 6).

The second advantage is the possibility to export the version number with just two lines of boilerplate in the `init` module:

```
__init__.py:
    # Import from itools
    from itools.utils import get_version

    __version__ = get_version(globals())
```

This way we can easily know the version of an installed package:

```
>>> import mypkg
>>> print mypkg.__version__
1.0.2
```

## 5 Git

*Git* is a *Source Code Management* tool. Unlike the most widely used CVS<sup>2</sup>, *Git* belongs to the new generation of distributed *SCMs*, and is best known to be the tool used to manage the Linux<sup>3</sup> source code.

As of today the `itools` packaging system relies heavily on *Git*. This means that our package must be managed by *Git*, if we want to use the `itools` packaging facilities.

Following our example, so far we have three files with the content seen before:

```
mypkg/
    __init__.py
    setup.py
    setup.conf
```

At this point we are going to initialize the *Git* archive:

---

<sup>1</sup><http://git.or.cz>

<sup>2</sup>[http://en.wikipedia.org/wiki/Concurrent\\_Versions\\_System](http://en.wikipedia.org/wiki/Concurrent_Versions_System)

<sup>3</sup><http://www.kernel.org/>

```
$ git init
Initialized empty Git repository in .git/
$ git add __init__.py setup.conf setup.py
$ git commit -m "Initial commit."
Created initial commit 41a1f72: Initial commit.
2 files changed, 8 insertions(+), 0 deletions(-)
create mode 100644 __init__.py
create mode 100644 setup.conf
create mode 100644 setup.py
```

It is not the purpose of this document to explain *Git*, for that we recommend the *Introduction to Git* manual available from the `itools` web site<sup>4</sup>. For the scope of this document this is all you need to know about *Git*.

## 6 The build, install and release processes

With `itools` the procedure to install a package from the source checkout, or to make a release are two lines.

### Build & Install

```
$ isetup-build.py
$ python setup.py install
```

### Make a source release

```
$ isetup-build.py
$ python setup.py sdist
```

### 6.1 The `isetup-build.py` script

The `isetup-build.py` script uses *Git* and the configuration file to automatize a few tasks. We can test it with our example:

```
$ isetup-build.py
* Version: master-200712081934
* Build MANIFEST file (list of files to install)
```

### The version number

First thing the `isetup-build.py` script does is to figure out the version number, which is made up of two parts:

```
<branch or tag name>-<timestamp>
```

---

<sup>4</sup><http://www.ikaaro.org/docs>

With *Git* the default branch name is *master*. The timestamp is the date and time of the last commit. This explains why the version number of the example above is `master-200712081934`.

But if we are in a branch named `1.0`, and we have a tag named `1.0.2`, the version number will be `1.0.2-<timestamp>`. If it happens to be that the tag points to the last commit, then the timestamp will be omitted, and the version number will just be `1.0.2`.

With this versioning scheme we will be able to produce releases numbered like this:

```
1.0.0
1.0.0-200712251143
1.0.0-200712271622
...
1.0.1
1.0.1-200712281203
...
1.0.2
```

As you may have guessed, this is the versioning scheme used by `itools` and `itools` based packages like `ikaaro`. The versions with a timestamp are development snapshots not released to the public. The versions without the timestamp are public releases.

### The MANIFEST file

The last thing the `isetup-build` script does is to build the MANIFEST file: the list of files that make up the package. This list is made up of:

- all files kept in the *Git* archive, this is to say, the source files;
- the automatically generated MANIFEST and `version.txt` files;
- the automatically generated files needed in a multilingual package (see Section 7).

## 7 Multilingual packages

Now, say you want to offer a multilingual user interface, and you choose to use `itools` to do the job (a wise decision).

The details on software internationalization and localization with `itools` are explained on the library documentation available from the `itools` web site, see in particular the chapter about the `itools.gettext` package.

Here we are going to explain the aspects related to packaging.

## 7.1 The source and target languages

The first thing to do is to define the source and target languages in the configuration file:

```
setup.conf:
# Languages
source_language = en
target_languages = es fr
```

In this example the source language is English, and there are two target languages, Spanish and French.

## 7.2 The `isetup-update-locale.py` script

Running the `isetup-update-locale.py` at this point will automatically create the `locale` folder, the POT template, and a PO file for each language:

```
$ isetup-update-locale.py
* Extract text strings from Python files..
* Update PO template
* Update PO files:
  en.po (new)
  es.po (new)
  fr.po (new)
$ tree locale
locale
|-- en.po
|-- es.po
|-- fr.po
`-- locale.pot
```

Since the PO files belong to the source, we should add them to the *Git* archive every time we run the `isetup-update-locale.py` script:

```
$ git add locale/locale.pot locale/*.po
$ git commit -m "Update PO files."
Created commit d79de97: Update PO files.
...
```

## 7.3 Building a multilingual package

At this point we must come back to the `isetup-build.py` script. If we run it again, once the package has been internationalized, we will find out it does a little more than before:

```
$ isetup-build.py
* Version: master-200712101700
* Compile message catalogs: en es fr
* Build MANIFEST file (list of files to install)
```

```

$ tree locale
locale
|-- en.mo
|-- en.po
|-- es.mo
|-- es.po
|-- fr.mo
|-- fr.po
`-- locale.pot

```

The `isetup-build.py` script has compiled the PO files to produce one binary MO file per language. These binary files will be used at run time by the internationalization logic to expose a multilingual interface to the user.

## 8 Tell *Git* to ignore non-source files

This may be a good time to make a break in the exposition and explain how to tell *Git* to ignore non-source files.

We have seen the `isetup-build.py` script produces a number of files that do not belong to the source code, but that are required to make a new release. These files must not be tracked by *Git*. To tell *Git* to ignore the non-source files we must create the `.gitignore` file:

```

.gitignore:
*.pyc
version.txt
MANIFEST
locale/*.mo

```

The example above shows that “compiled” Python files must be ignored, as well as the automatically generated `version.txt` and `MANIFEST` files, and the binary language files. Now we should commit `.gitignore`:

```

$ git add .gitignore
$ git commit -m "Tell Git to ignore non-source files."
Created commit 6790c7c: Tell Git to ignore non-source files.
1 files changed, 4 insertions(+), 0 deletions(-)
create mode 100644 .gitignore

```

## 9 The release procedure

To summarize up everything seen in this document, this is the procedure to make a public release of a multilingual package:

1. Once the strings in the user interface are frozen, we must update the translations. To do so we first extract the text strings from the source files with the help of the `isetup-update-locale.py` script, as seen before:

```
$ isetup-update-locale.py
* Extract text strings from Python files..
* Extract text strings from XHTML files
* Update PO template
* Update PO files:
  en.po . done.
  es.po . done.
  fr.po . done.
$ git add locale/locale.pot locale/*.po
$ git commit -m "Update PO files."
```

2. Now it is time for the human translators to update the translations for each target language.

3. Once this is done we can tell the source is ready, so we make a new tag. For example, if we are in the 1.0 branch, we may want to make the release number 1.0.2:

```
$ git tag 1.0.2
```

4. At last, we are ready to make the source release:

```
$ isetup-build.py
* Version: 1.0.2
* Compile message catalogs: en es fr
* Build MANIFEST file (list of files to install)
$ python setup.py sdist
...
```