

# latua documentation

## Usage

**Contact:**     [team@petunial.com](mailto:team@petunial.com)  
**Date:**        2007-11-23  
**Status:**     This is document is “mostly finished”  
**Revision:**   326  
**Copyright:**  BSD License

### Dedication

For python users and developers.

### Abstract

This document describes how to use latua library.

### Table of Contents

- 1 [About](#)
- 2 [latua classes](#)
  - 2.1 [Base class](#)
    - 2.1.1 [platform class](#)
    - 2.1.2 [i18n class](#)
    - 2.1.3 [logger class](#)
  - 2.2 [System class](#)
    - 2.2.1 [configuration class](#)
    - 2.2.2 [crypt class](#)
    - 2.2.3 [files class](#)
    - 2.2.4 [modules class](#)
  - 2.3 [Index class](#)
    - 2.3.1 [database class](#)
    - 2.3.2 [file class](#)
    - 2.3.3 [search class](#)
- 3 [latua scripts](#)
  - 3.1 [latua\\_documentation](#)
  - 3.2 [latua\\_index](#)
  - 3.3 [latua\\_propset](#)
  - 3.4 [latua\\_translation](#)

# 1 About

As lightweight library `latua` comes with just three main classes which could be used for inheritance or simple usage.

**Base** Contains: platform specific variables, `il8n` helpers and logging wrappers.

**System** Contains: crypt helpers, a configuration wrapper, modules helpers and various file functions.

**Index** Is a simple and fast full test indexer.

## 2 `latua` classes

`latua` classes could be used for inheritance or usage.

### 2.1 Base class

The `latua Base` class simplifies recurrent initialization tasks of applications. It stores initialized values in a singleton context to avoid initializing them multiple times.

#### 2.1.1 `platform` class

The `platform` class initialize platform dependend variables only once and makes them easily accessible for the application.

An example for the usage of the `platform` class is:

```
>>> import latua
>>> base = latua.Base()
>>> base.platform.admin
'root'
```

On unix platform admin will return *root* and on windows *Administrator*.

Available variables are:

**admin** Returns the admin user of the platform.

**application\_directory** Returns the installation path of running module.

**application\_name** Returns the name of the running application.

**command\_file** Returns a string which represents the way on starting commands on this platform.

**configuration\_directory** Returns the path to the directory for configuration files.

**data\_directory** Returns the path to the directory for data files.

**home\_directory** Returns the path to the home directory of the running user.

**languages** Returns a list of all languages.

**locales\_directory** Returns the path to the locales directory.

**logging\_directory** Returns the path to the logging directory.

**logging\_handler** Returns syslog logging handler.

**logging\_level** Returns default logging level.

**modules\_directory** Returns path to installed python modules.

**user** Returns the user which started the application.

**hidden(path)** Function returning `True` if given path contains hidden components. Raises an platform error if given path not exists.

### 2.1.2 i18n class

The `i18n` class simplifies the internationalization process of an application by providing an easy wrapper to the core modules `gettext` and `locale`.

An example for the usage of the `i18n` class is:

```
>>> import latua
>>> base = latua.Base()
>>> base.i18n.find_translation(domain="latua")
>>> base.i18n.languages
['en', 'de']
>>> base.i18n.install_translation("de")
```

This example found two languages for the application `latua` which are either installed in the system path or in the local application path and installs one of them.

Available functions and variables are:

**languages** Contains a list of already found languages.

**natives** Contains a dict with native names (unicode) for languages.

**find\_translation(domain=None)** Searches for installed translations (gettext based po/mo-files) in the system path and the application directory. The second is needed if the application is just started from directory without installation. Search results are stored in languages list.

**install\_translation(language="en", domain=None)** Installs translation for given language from given domain on the fly.

### 2.1.3 logger class

The `logger` class simplifies logging by providing an easy wrapper to the `logging` core module.

An example for the usage of the `logger` class is:

```
>>> import latua
>>> base = latua.Base()
>>> base.logger.error("example error log to console")
2007-11-19 21:17:45,865 python ERROR: __init__(): example error log to\
console (<stdin> at line 1)
```

The `logger` class of `latua` logs error messages by default to console.

Besides the functions and variables of the `logging.Logger` core class, there are various other functions and variables available:

**levels** Contains a list of supported log levels of the logger class. These are: `critical`, `error`, `warning`, `info` and `debug`.

**types** Contains a list of supported log types of the logger class. These are: `logfile`, `smtp` and `syslog`.

**flush()** Flush the logger.

**logfile(logfile=None, logfile\_size=None, logfile\_rotate=None)** Set logging to given logfile and rotates logfile after given size is reached as often as given.

**log\_level()** Set actual log level. This does not affect console logger which is set to error level by default.

**reset()** Reset actual logging to console only (default).

**smtp(mail\_host, from\_address, to\_address, subject)** Set logging to given smtp host with given addresses and subject.

**syslog()** Set logging to syslog (platform independent).

## 2.2 System class

The `System` class simplifies the process of achieving system specific tasks.

### 2.2.1 configuration class

The `configuration` class simplifies the reading and writing configuration files by providing an easy wrapper around the `ConfigParser` core module.

An example for the usage of the `configuration` class is:

```
>>> import latua
>>> system = latua.System()
>>> system.configuration.add_section("section")
>>> system.configuration.set("section", "option", "value")
>>> system.configuration.write_file("configurtaion/configuration.txt")
>>>
# cat configuration/configuration.txt
[section]
option = value
```

The configuration class of `latua` creates configuration sub-directories. Furthermore it supports case-sensitive options.

Besides the functions and variables of the `ConfigParser.SafeConfigParser` core class, there are various other functions available:

**read\_file(configuration\_file)** Reads given configuration file or raise an configuration error if file does not exists.

**write\_file(configuration\_file)** Creates configuration sub-directories if needed and and writes actual configuration to the given file. Exisiting configuration file will be overwritten. If something fails a configuration error is raised.

### 2.2.2 crypt class

The `crypt` class simplifies the handling of crypted strings e.g. passwords by providing some helper functions.

An exmaple for the usage of the `crypt` class is:

```
>>> import latua
>>> system = latua.System()
>>> system.crypt.generate()
'siGhLdrx'
```

The default length for generated strings is 8 characters.

Available functions are:

**check(text, crypted\_text)** Check given text against crypted text.

**encrypt(text, algorithm="md5")** Encrypt a given string with the given algorithm.

**generate(length=8, characters=None)** Generate a random string with given length containing a subset of the given characters.

**supported** Contains the supported crypt algorithmss, these are namely: `md5` and `sha1`.

### 2.2.3 files class

The `files` class simplifies working on files and directories by providing some helper functions.

An example for the usage of the `files` class is:

```
>>> import latua
>>> system = latua.System()
>>> system.files.permission(".")
('rwx', 'r-x', 'r-x')
```

The permissions are returned as tuple.

Available functions are:

**integrity(directory, files)** Check integrity of files in directory. Raise an file error if directory not exists.

**permission(path)** Return permission of the given path as tuple.

**search(directory, type="", extension="", absolute=True)** List all files or directories with given extension from given directory. Raise an files error if directory not exists.

### 2.2.4 modules class

The `modules` class simplifies working on modules by providing some helper functions.

An example for the usage of the `modules` class is:

```
>>> import latua
>>> system = latua.System()
>>> system.modules.filename_modulename("latua/system.py")
'latua.system'
```

Converting a filename to modulename is platform independent.

Available functions are:

**import(module\_path, variable=None)** Import a module from a path given as string and get given variable.

**append(target, path, value=None)** Create and append a given module to a given target module. Raises an module error on empty given path.

**filename\_modulename(filename)** Convert given filename to modulename.

**modulename\_filename(modulename)** Convert given modulename to filename.

## 2.3 Index class

The `Index` class helps on creating a full text index of various documents.

### 2.3.1 database class

The `database` provides functions for managing the sqlite database which contains the index.

An example for the usage of the `database` class is:

```
>>> import latua
>>> index = latua.Index()
>>> index.database.maintenance()
```

The sqlite database will automatically initialized as `index.db`.

Available functions are:

**\_reset()** Resets and initialiazes the index database.  
**maintenance()** Runs various maintenance tasks on index database to improve performance.

### 2.3.2 file class

The **file** class provides functions for managing files in index.

An example for the usage of the **file** class is:

```
>>> import latua
>>> index = latua.Index()
>>> index.file.add("README")
```

Default file format handler is ascii.

Available functions are:

**add(filename, filetype="text")** Add file with given filename to index. Raises a file error if a problem occurs.

**meta(filename)** Return meta data for given file in index. Raises a file error if a problem occurs.

**remove(filename)** Remove given file from index. Raises a file error if a problem occurs.

**rename(old\_filename, new\_filename)** Rename given file in index to new given filename. Raises a file error if a problem occurs.

**update(filename)** Update given file in index, which means: index possible new lines after last known seek-point. Raises a file error if a problem occurs.

### 2.3.3 search class

The **search** class provides functions to search in index.

An example for the usage of the **search** class is:

```
>>> import latua
>>> index = latua.Index()
>>> index.search.word("latua")
[]
```

The search for words will try to match all similiar words.

Available functions are:

**words(query, maxresults=10)** Return a list of words which match given query. Raises a search error if a problem occurs.

**expression(expression, maxresults=10, regular\_expression=False)** Return index search results for given expressions. Raises a search error if a problem occurs.

## 3 latua scripts

The installation of latua installs various scripts on the system.

### 3.1 latua\_documentation

The `latua_documentation` script generates documentation files in various formats from ascii textfiles with the help of the `docutils` available at: [\[http://docutils.sourceforge.net/\]](http://docutils.sourceforge.net/). Furthermore it generates API documentation from source code fiels with the help of `epyoc` from [\[http://epydoc.sourceforge.net/\]](http://epydoc.sourceforge.net/).

An example for the usage of the `latua_documentation` script is:

```
# latua_documentation latua .
```

The usage options of `latua_documentation` are shown if `--help` or `-h` are provided on console.

The `latua_documentation` script expects the application name and the path to the application source as arguments.

### 3.2 latua\_index

The `latua_index` script is a wrapper around the `latua Index` class. It simplifies the process of using the `Index` class on console.

An example for the usage of the `latua_index` script is:

```
# latua_index add README
```

The usage options of `latua_index` are shown if `--help` or `-h` are provided on console.

The following various actions are recognized by the `latua_index` script on console:

- add** `<filename>` Adds a file to index.
- maintenance** Runs varoius maintenance tasks on index database.
- meta** `<filename>` Returns meta data for filename in index.
- remove** `<filename>` Removes a filename from index.
- rename** `<old_filename>` `<new_filename>` Renames a file in index.
- reset** Remove all files from index.
- search** `<expression>` Search for an expression in index.
- update** `<filename>` Index new lines of file.

### 3.3 latua\_propset

The `latua_propset` script simplifies the work the with a subversion repository. It removes temporary files and sets propset ignore and keywords on files and directories.

An example for the usage if the `latua_propset` script is:

```
# latua_propset .
```

The usage options of `latua_propset` are shown if `--help` or `-h` are provided on console.

The `latua_propset` script expects the path to the local copy of the subversion repository as argument.

### 3.4 latua\_translation

The `latua_translation` script generates and updates translation files for multiple languages of an application which could be used with `gettext`.

An example for the usage of the `latua_translation` script is:

```
# latua_documentation -l "en de" latua .
```

The usage options of `latua_translation` are shown if `--help` or `-h` are provided on console.

The `latua_translation` script expects the application name and the path to the application source as arguments. Furthermore the option `-l` should specified to set the languages which should be updated.