

latua documentation

Usage

Contact: team@petunial.com
Date: 2008-04-05
Status: This is document is “mostly finished”.
Revision: 381
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 [file class](#)
 - 2.2.1 [archives class](#)
 - 2.3 [Pipe class](#)
 - 2.4 [Parser class](#)
 - 2.4.1 [ini class](#)
 - 2.4.2 [crypt class](#)
 - 2.4.3 [modules class](#)
 - 2.5 [Index class](#)
 - 2.5.1 [database class](#)
 - 2.5.2 [file class](#)
 - 2.5.3 [search class](#)
 - 2.6 [Singleton/Borg 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, i18n helpers and logging wrappers.
- File** Contains archives wrapper, modules helpers and various file functions.
- Pipe** Contains crypt helpers and validation class.
- Parser** Contains a parser for the ini file format. More parsers will probably follow in future.
- Index** Is a simple and fast full test indexer.

Furthermore there is a `Singleton/Borg` class available.

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.

The `latua Base` class accepts the following arguments:

- application_name** Sets the name of the running application in platform class.

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, automatically determined from script name or set from `Base` class arguments.
- 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.

configuration_file Returns the path to the configuration file of the application.

database_directory Returns the path to the directory for database files.

database_file Returns the path to the database file of the application.

home_directory Returns the path to the home directory of the running user.

installed_directory Returns the path to the directory for data files.

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_file Returns the path to the logging file of the application.

logging_handler Returns syslog logging handler.

logging_level Returns default logging level.

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("latua")
>>> base.i18n.find_translation()
>>> 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(directory=None) Searches for installed translations (gettext based po/mo-files) in the system path or the given 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", directory=None) Install translation for given language from given directory 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 `logging` 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 file class

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

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

```
>>> import latua
>>> file = latua.File()
>>> file.permission(".")
('rwx', 'r-x', 'r-x')
```

The permissions are returned as tuple.

The `latua File` class provides the following functions and variables:

types Returns supported file types for search.

fetch(url, filename, frequency=5) Fetch a given url into a given filename but not more often than with given frequency in minutes.

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.

2.2.1 archives class

The `archives` class simplifies working on different types of archives by providing some helper functions.

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

```
>>> import latua
>>> system = latua.System()
>>> system.archives.types
["bzip2", "gzip", "tarball", "zip"]
```

These archive types are supported by archives class.
Available functions and variables are:

types Returns all supported archive types.
extract(archive, path, type) Extract a given archive with given type to the given path.

2.3 Pipe class

system(item) Returns True if given item is a system item (starts with an underscore).

2.4 Parser class

2.4.1 ini class

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

An example for the usage of the ini 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. Existing configuration file will be overwritten. If something fails a configuration error is raised.

2.4.2 crypt class

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

An exmample 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 algorithms, these are namely: `md5` and `sha1`.

2.4.3 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.

2.5 Index class

The `latua` `Index` class helps on creating a full text index of various file types.

The `latua` `Index` class accepts the following arguments:

filename Sets the name of the sqlite database. Default is `index.db`.

2.5.1 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.

2.5.2 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.

2.5.3 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 similar 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.

2.6 Singleton/Borg class

The `Singleton` class provides an implementation of the singleton design pattern. Furthermore there is a `Borg` class as an alternative implementation.

An example for the usage of both is:

```
>>> import latua.singleton
>>> singleton_a = latua.singleton.Singleton()
>>> singleton_b = latua.singleton.Singleton()
>>> borg_a = latua.singleton.Borg()
>>> borg_b = latua.singleton.Borg()
>>> singleton_a.foo = 1
>>> singleton_b.foo
```

```

1
>>> borg_a.foo = 1
>>> borg_b.foo
1
>>> id(singleton_a)
2094153580
>>> id(singleton_b)
2094153580
>>> id(borg_a)
2094153484
>>> id(borg_b)
2094153516

```

Borgs are different objects sharing their states.

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/>.

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 with a subversion repository. It removes temporary files and sets propset ignore and keywords on files and directories.

An example for the usage of 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_translation -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 be specified to set the languages which should be updated.