Nxpy Documentation

Release 1.0.5

Nicola Musatti

Apr 16, 2023

Contents

Nxpy is an etherogeneous collection of libraries, dealing with diverse topics such as wrapping complex commands with API's, automation of backup files, support for writing your own file-like objects and many other things.

CHAPTER 1

`abstract` - Additions to the *abc* standard module

Helpers for the standard `abc` module.

class abstractstatic(*function*)

Decorator that combines `staticmethod` and `abc.abstractmethod`.

Copied from this answer to this StackOverflow question.

CHAPTER 2

`backup_file` - File objects with automated backup

Backup a file or directory to make editing reversible.

Implement the context manager protocol, so as to be suitable to be used with the *with* statement. When used in this fashion changes are discarded when an exception is thrown.

class BackupDir(*dir_*, *ext='.BAK'*, *mode=1*)
Move or copy a directory that needs to be recreated or modified.

__enter__()
When the controlling *with* statement is entered, create the backup directory.

__exit__(*exc_type*, *exc_val*, *exc_tb*)
When the controlling *with* statement is exited normally discard the backup directory, otherwise restore it to its original place.

__init__(*dir_*, *ext='.BAK'*, *mode=1*)
Prepare to backup the *dir_* directory.

The backup will be created in *dir_*'s parent directory, which must be writable, with extension *ext*. If *mode* is *MOVE*, the default, the original directory will be moved to the backup destination; if *mode* is *COPY* it will be copied there.

commit()
Discard the backup, i.e. keep the supposedly modified file.

rollback()
Replace the original file with the backup copy.

save()
Create a backup copy of the original directory.

class BackupFile(*file_*, *ext='.BAK'*, *dir='.'*, *mode=2*)
Implements a read only file object used to automatically back up a file that has to be modified.

__enter__()
When the controlling *with* statement is entered, create the backup file.

__exit__(*exc_type*, *exc_val*, *exc_tb*)
When the controlling *with* statement is exited normally discard the backup file, otherwise restore it to its original place.

__init__(*file_*, *ext='.BAK'*, *dir='.'*, *mode=2*)
Prepare to backup *file_*, either a file-like object or a path.

The backup file will be created in directory *dir* with extension *ext*. If *mode* is *COPY* the original file will be copied to the backup destination; if *mode* is *MOVE* it will be moved there.

close()
Close the backup file and release the corresponding reference.

The backup file may not be reopened.

commit()
Discard the backup, i.e. keep the supposedly modified file.

name
The name of the file to be backed up.

open(*mode=4*)
Open the backup file for reading. *mode* may be either *TEXT* or *BINARY*.

rollback()
Replace the original file with the backup copy.

save()
Create a backup copy of the original file.

Throw *SaveError* if it wasn't possible.

exception MissingBackupError
raised when a backup file or directory isn't found.

exception NotSavedError
Raised when commit or rollback is called on an inactive *BackUpFile* or *BackUpDirectory*.

exception RemovalError
Raised to signal errors in the removal of backup files or directories.

exception SaveError
Raised when a backup file or directory could not be created.

CHAPTER 3

command - Wrap complex commands in Python objects

Tools to wrap a Python API around interactive and non-interactive programs.

The *command.Command* and *interpreter.Interpreter* classes handle batch and interactive commands respectively. They can be provided with *option.Config* instances which describe the options available to the programs being wrapped. The *option.Parser* class can then be used to validate option sets and construct the corresponding command lines. See the *svn.svn* module for a concrete example.

3.1 command - Drive batch commands with function calls

Non interactive command driver.

class Command(*cmd*, *debug=False*)

Represents the command to be executed. Typically you would derive from this class and provide a different method for each alternative way of invoking the program. If the program you want to execute has many sub-commands you might provide a different method for each sub-command. You can use the *option.Config* class to declare the options supported by your command and then use the *option.Parser* class to validate your methods' arguments and generate the resulting command line. A debug mode is available in which commands are echoed rather than run. This can be enabled globally or separately for each invocation.

__init__(*cmd*, *debug=False*)

Takes as arguments the command name and a boolean value indicating whether debug mode should be activated for all executions of this command.

run(*parser*, *debug=False*)

Executes the command. Takes as arguments a command line parser (see the *option* module) and a boolean indicating whether debug mode should be used for this execution.

exception Error(*cmd*, *returncode*, *err*)

Raised when command execution fails.

__init__(*cmd*, *returncode*, *err*)

Takes the command line, the error code and the contents of the error stream.

3.2 `error` - The `command` package exception hierarchy

Exception classes for the nxpy.command package.

exception BadLogFormat
Raised if the requested formatting option is unknown.

exception Error
Package exceptions' base class.

exception ExpectError
Raised on invalid input from stdout or stderr.

exception TimeoutError
Raised when expect didn't satisfy a timing constraint.

exception TimerError
Raised on misuse of the Timer class.

3.3 `interpreter` - Wrap interactive programs in Python classes

Interactive program driver.

exception BadCommand(*cmd*, *err*)
Raised on a command execution failure

__init__(*cmd*, *err*)
Takes the failed command and the contents of the error stream.

class BaseInterpreter(*popen*)
Controls the execution of an interactive program in a sub-process. Provides means to send input to the controlled process and to check different conditions on its output and error streams.

__init__(*popen*)
Creates an interpreter instance. *popen* is a `Popen`-like object which must support non-blocking I/O.

expect(*cond=None*, *timeout=0*, *retries=0*, *interval=0.01*, *quantum=0.01*, *raise_on_error=True*, *log=None*)
Express expectations on the outcome of a command.

cond is a two argument callable which will be passed the command's standard output and standard error, and which should return *True* if the expectation is satisfied. For the other arguments see the documentation for the *Timer* class.

expect_any(***kwargs*)
Expect any output.

expect_lines(*count=1*, ***kwargs*)
Expect *count* lines of output.

expect_regexp(*regexp*, *where=0*, ***kwargs*)
Expect to find a match for the *regexp* regular expression within the *where* stream.

expect_string(*string*, *where=0*, ***kwargs*)
Expect a *string* in the *where* stream.

run(*cmd*, *log=None*, ***kwargs*)
Executes the command and waits for the expected outcome or an error.

send_cmd(*cmd*, *log=None*)
Write *cmd* to the interpreter's input, optianally logging it. If *log* is not *None*, override the global setting.

setLog(*log*)
If *log* is *True*, enable logging of command output and error, otherwise disable it.

class Interpreter(*cmd*)
The actual Interpreter class.

This implementation uses a `core.nonblocking_subprocess.NonblockingPopen` instance.

__init__(*cmd*)
Creates an interpreter instance. *popen* is a `Popen`-like object which must support non-blocking I/O.

class LineWaiter(*count*)
Wait for *count* lines of output.

__call__(*...) <==> x(...*)

__init__(*count*)
x.__init__(...) initializes x; see help(type(x)) for signature

class RegexpWaiter(*regexp*, *where*)
Wait for a match to a given *regexp*, passed either compiled or as a string.

__call__(*...) <==> x(...*)

__init__(*regexp*, *where*)
x.__init__(...) initializes x; see help(type(x)) for signature

class StringWaiter(*string*, *where*)
Wait for a specific *string* in the *where* stream.

__call__(*...) <==> x(...*)

__init__(*string*, *where*)
x.__init__(...) initializes x; see help(type(x)) for signature

class Timer(*timeout=0*, *retries=0*, *interval=0.1*, *quantum=0.01*)
A collaborative timer class. Support a polling mechanism by keeping track of the amount of time to wait before the next attempt, according to different policies.

__init__(*timeout=0*, *retries=0*, *interval=0.1*, *quantum=0.01*)
Specify an overall *timeout*, a number of *retries* and/or an *interval* between them. The next attempt will not take place before a *quantum* has passed. Timings are expressed in seconds. If a timeout is specified it will take precedence over the other arguments; in that case the number of retries will take precedence over the interval. If neither a timeout nor a number of retries are specified the overall timer will never expire.

expired()
Indicate whether the current timer expired. Use as polling loop control condition.

getInterval()
Return the next wait interval. Call after each attempt in order to know how long to wait for.

reset()
Reset the timer.

waitError(*out*, *err*)
Wait for any error.

waitOutput(*out*, *err*)
Wait for any output.

3.4 `option` - Describe complex command lines

Function argument to command line option conversion. Provides means to describe commands with complicated syntaxes, which often combine sub-commands, options and arguments. Typical examples include subversion and ftp.

class Config(*prefix='–'*, *separator=' '*, *bool_opts=()*, *value_opts=()*, *iterable_opts=()*, *format_opts={}*, *mapped_opts={}*, *opposite_opts={}*)

Command option definitions. Provides a single definition point for all the options supported by a command.

__init__(*prefix='–'*, *separator=' '*, *bool_opts=()*, *value_opts=()*, *iterable_opts=()*, *format_opts={}*, *mapped_opts={}*, *opposite_opts={}*)

Constructor. Its arguments are used to specify all the valid options. Each option is prefixed by *prefix*. When an option takes multiple arguments these are separated by a *separator*. *bool_opts* must be specified on the command line when they are *True*. *value_opts* take a single argument; *iterable_opts* take multiple arguments; *format_opts* have their syntax specified by means of a format string; *mapped_opts* require some form of translation, usually because they are not valid Python identifiers; *opposite_opts* must be specified on the command line when they are *False*.

exception InvalidOptionError

Raised when an option is not supported.

class Parser(*config*, *command*, *arguments*, *options*, ***defaults*)

Constructs a complex command line from the provided *command* and its *options* and *arguments*. Uses a `Config` instance, *config*, to provide means to check conditions on the supplied options. Other constraints on how options should be used may be expressed and verified by means of the *check* methods.

__init__(*config*, *command*, *arguments*, *options*, ***defaults*)

Takes an instance of `Config`, a *command* to execute, an iterable of *arguments* and a mapping of *options* and their actual values. The remaining keyword arguments indicate the options supported by *command* with their default values.

checkExactlyOneOption(**options*)

Checks that one and only one in a set of mutually exclusive options has been specified.

checkExclusiveOptions(**options*)

Checks that at most one in a set of mutually exclusive options has been specified.

checkMandatoryOptions(**options*)

Checks that all compulsory options have been specified.

checkNotBothOptsAndArgs(**options*)

Checks that options incompatible with arguments haven't been specified if any argument is present.

checkOneBetweenOptsAndArgs(**options*)

Checks that either at least one in a set of options or some arguments have been specified, but not both.

getCommandLine()

Returns the command line to be executed.

CHAPTER 4

file_object - Stubs for read-only and modifiable file-like objects

Helper classes for the implementation of read-only and writable file objects that forward calls to an actual file object variable.

class ReadOnlyFileObject (*file_=None*)
Implement the non modifying portion of the file object protocol by delegating to another file object.

Subclass and override as needed.

__init__ (*file_=None*)
Set the delegate file object.

setFile (*file_*)
Set the delegate file object.

class WritableFileObject (*file_=None*)
Implement the file object protocol by delegating to another file object.

Subclass and override as needed.

__init__ (*file_=None*)
Set the delegate file object.

CHAPTER 5

`file` - File related utilities

File related utilities.

compare(*file1*, *file2*, *ignore_eof=True*, *encoding=None*)
Compare two text files for equality. If *ignore_eof* is *True*, end of line characters are not considered. If not *None* *encoding* is used to open the files. On Python 2.x *encoding* is ignored.

open_(**args*, ***kwargs*)
Open a file removing invalid arguments on Python 2.x.

CHAPTER 6

maven - Tools to execute the Maven build tool and manipulate its configuration

Tools to drive the Maven build tool and to manipulate its configuration files.

6.1 artifact - Representation of a Maven artifact

6.2 assembly_descriptor - Representation of a Maven Assembly plugin's descriptor

6.3 mvn - Wrapper class for the mvn command line tool

Maven wrapper.

class Mvn(*debug=None*)

__init__(*debug=None*)
Takes as arguments the command name and a boolean value indicating whether debug mode should be activated for all executions of this command.

clean(*projects=None*, *debug=None*)

deploy(*projects=None*, *debug=None*)

package(*projects=None*, *debug=None*)

6.4 pom - Representation of a Maven POM file

CHAPTER 7

`memo` - Memoize objects according to a given key

Memoize class instances according to a given key.

By default the key only assumes the *True* value, thus implementing a singleton.

class Memo

Base class for classes that require memoization.

Subclasses should override the *_key(*args, **kwargs)* method to compute a key on the constructor's arguments.

Care should be taken to avoid calling *__init__()* again for entities already constructed.

static __new__(*cls*, **args*, ***kwargs*)

Return the instance corresponding to the given key, creating it if it doesn't exist.

CHAPTER 8

nonblocking_subprocess - Subprocesses with non-blocking I/O

Allow non-blocking interaction with a subprocess.

This module was taken from this recipe in the ActiveState Code Recipes website, with only minor modifications. This is the original description:

```
Title:        Module to allow Asynchronous subprocess use on Windows and Posix␣
↪platforms
Submitter:    Josiah Carlson (other recipes)
Last Updated: 2006/12/01
Version no:   1.9
Category:     System
```

On Windows pywin32 is required.

class NonblockingPopen(*cmd*, *encoding=None*, ***kwargs*)

An asynchronous variant to `subprocess.Popen`, which doesn't block on incomplete I/O operations.

Note that the terms input, output and error refer to the controlled program streams, so we receive from output or error and we send to input.

__init__(*cmd*, *encoding=None*, ***kwargs*)

Execute *cmd* in a subprocess, using *encoding* to convert to and from binary data written or read from/to the subprocess's input, output and error streams.

Additional keyword arguments are as specified by `subprocess.Popen.__init__()` method.

get_conn_maxsize(*which*, *maxsize*)

Return *which* output pipe (either stdout or stderr) and *maxsize* constrained to the [1, 1024] interval in a tuple.

recv(*maxsize=None*)

Receive at most *maxsize* bytes from the subprocess's standard output.

recv_err(*maxsize=None*)

Receive at most *maxsize* bytes from the subprocess's standard error.

send(*input_*)
Send *input_* to the subprocess's standard input.

send_recv(*input_=''*, *maxsize=None*)
Send *input_* to the subprocess's standard input and then receive at most *maxsize* bytes from both its standard output and standard error.

recv_some(*p*, *t=0.1*, *e=1*, *tr=5*, *stderr=0*)
Try and receive data from `NonblockingPopen` object *p*'s stdout in at most *tr* tries and with a timeout of *t*. If *stderr* is True receive from the subprocess's stderr instead.

send_all(*p*, *data*)
Send all of *data* to `NonblockingPopen` object *p*'s stdin.

CHAPTER 9

past - Python version support enforcement

Identification and enforcement of supported Python releases.

class Version(*version*)
Identifies a Python release in a way that is convenient for comparison and printing.

at_least()
Return *True* if the current Python version is equal or higher than *self*.

at_most()
Return *True* if the current Python version is equal or lower than *self*.

enforce_at_least(*version*)
Assert that the current Python version is equal or higher than *version*.

enforce_at_most(*version*)
Assert that the current Python version is equal or lower than *version*.

CHAPTER 10

path - File system related utilities

filesystem related utilities.

class CurrentDirectory(*path*)
A context manager that allows changing the current directory temporarily.

__init__(*path*)
Set the current directory to *path*.

current
Return the current directory.

blasttree(*dir_*)
Remove a directory more stubbornly than `shutil.rmtree()`.

Required on filesystems that do not allow removal of non-writable files

CHAPTER 11

`ply` - Add-ons for the PLY lexer & parser generator

Wrapper classes for the PLY parser generator.

11.1 `parser` - A class wrapper for PLY parsers

11.2 `scanner` - A class wrapper for PLY scanners

CHAPTER 12

`sequence` - Sequence related utilities

Utility functions that deal with non-string sequences.

`make_tuple`(*arg*)

An alternate way of creating tuples from a single argument.

A single string argument is turned into a single element tuple and a dictionary argument is turned into a tuple of its items. Otherwise it works like the standard tuple constructor.

CHAPTER 13

`sort` - Sorting functions

Sort functions.

`topological_sort` (*pairs*)

Provide a topological ordering of the supplied pair elements.

pairs is a sequence of two element sequences, in which the first element comes before the second according to the desired ordering criterium.

CHAPTER 14

svn - High level API for the Subversion version control tool

A Python API for the Subversion version control tool.

A lazy, ahem, agile person's answer to the official svn bindings.

14.1 svn - Wrapper for the svn client tool

Subversion client wrapper.

Only supports versions 1.6, 1.7 and 1.8, others might work but have not been tested. Requires at least Python 2.6.

class Info(*out*)
Represents the output of the `svn info` command in a structured way.

__init__(*out*)
x.__init__(...) initializes x; see help(type(x)) for signature

__str__() <==> *str(x)*

class Parser(*command*, *arguments*, *options*, ***defaults*)
Allows passing *nxpy.svn.url.Url* instances as arguments to *Svn*'s methods.

__init__(*command*, *arguments*, *options*, ***defaults*)
Takes an instance of *Config*, a *command* to execute, an iterable of *arguments* and a mapping of *options* and their actual values. The remaining keyword arguments indicate the options supported by *command* with their default values.

class Status(*line*)
Represents the output of one line of the `svn status` command in a structured way.

__init__(*line*)
x.__init__(...) initializes x; see help(type(x)) for signature

__str__() <==> *str(x)*

class Svn(*debug=False*)
The actual wrapper.

__init__(*debug=False*)
Takes as arguments the command name and a boolean value indicating whether debug mode should be activated for all executions of this command.

cat(**targets*, ***options*)

checkout(*src*, *dest*, *debug=False*, ***options*)

commit(*src*, *debug=False*, ***options*)

copy(*src*, *dest*, *debug=False*, ***options*)

delete(**targets*, ***options*)

diff(**targets*, ***options*)

export(*src*, *dest*, ***options*)

getexternals(*d*)
Return *d*'s `svn:externals` property as a dictionary of directory - URL pairs.

Note that only a limited subset of the externals syntax is supported: either the pre-svn 1.5 one (directory - URL) or the same with inverted elements. Throw *nxpy.svn.url.BadUrlError* if an external URL is malformed.

getignore(*d*)

import_(*src*, *dest*, *debug=False*, ***options*)

info(**targets*)

list(**targets*)

log(*src*, ***options*)

mkdir(**targets*, ***options*)

move(*src*, *dest*, *debug=False*, ***options*)

propget(*name*, **targets*)

propset(*name*, **targets*, ***options*)

setexternals(*externals*, *d*, *username=''*, *password=''*, *debug=False*)

setignore(*ignore*, *d*, *username=''*, *password=''*, *debug=False*)

status(**targets*, ***options*)

update(**targets*, ***options*)

version()

14.2 `svnadmin` - Wrapper for the svnadmin administration tool

Subversion administration tool wrapper.

class SvnAdmin(*debug=None*)

__init__(*debug=None*)
Takes as arguments the command name and a boolean value indicating whether debug mode should be activated for all executions of this command.

create(*path*, *debug=None*)

14.3 `url` - Models a URL adhering to the trunk/tags/branches convention

Subversion URL manipulation.

exception BadUrlError
Indicates a malformed URL.

class Url(*path*)
A well-formed Subversion repository URL that follows standard svn conventions.

The URL must end in either 'trunk', 'tags/label' or 'branches/label'.

__eq__(*other*)
x.__eq__(y) <==> x==y

__init__(*path*)
x.__init__(...) initializes x; see help(type(x)) for signature

__ne__(*other*)
x.__ne__(y) <==> x!=y

__str__() <==> *str(x)*

getbranch(*branch*)

gettag(*tag*)

gettrunk()

isbranch(*branch=None*)

istag(*tag=None*)

istrunk()

14.4 `wcopy` - Models a working copy

Working copy manipulation.

exception ModifiedError
Raised when attempting to tag or branch a working copy that contains changes.

exception NotOnBranchError
Raised when attempting to delete a working copy that is not on the requested branch.

exception NotOnTagError
Raised when attempting to delete a working copy that is not on the requested tag.

class Wcopy(*dir_*, *url=None*, *username=''*, *password=''*)
A working copy obtained by checking out a *Url*.

__init__(*dir_*, *url=None*, *username=''*, *password=''*)
Initialize attributes.

If *url* is not None, perform a checkout, otherwise check that *dir_* points to a valid working copy.

__str__() <==> *str(x)*

branch(*label*)

commit()

delete_branch(*label*)

delete_path(*path*, *keep_local=False*)

delete_tag(*label*)

getexternals()

getignore()

setexternals(*ext*)

setignore(*ign*)

tag(*label*)

update(*ignore_externals=False*)

CHAPTER 15

temp_file - Temporary files that support the context protocol

Temporary files and directories.

Requires at least Python 2.6

class TempDir(**args*, ***kwargs*)
A temporary directory that implements the context manager protocol.

The directory is removed when the context is exited from. Uses tempfile.mkdtemp() to create the actual directory.

__init__(**args*, ***kwargs*)
Create a temporary directory with the given arguments.

name
Return the directory name.

class TempFile(**args*, ***kwargs*)
A temporary file that implements the context manager protocol.

Wrap a tempfile.NamedTemporaryFile() generated file-like object, to ensure it is not deleted on close, but rather when the underlying context is closed.

__init__(**args*, ***kwargs*)
Create a temporary file with the given arguments.

name
Return the actual file name.

CHAPTER 16

test - Test support utilities

Testing related utilities.

16.1 env - Access to the testing environment for the svn, maven and msvs packages

Environment configuration for tests that interact with the system.

class Data (*package*)

__init__ (*package*)
x.__init__(...) initializes x; see help(type(x)) for signature

class Env (*package*)

__init__ (*package*)
x.__init__(...) initializes x; see help(type(x)) for signature

class EnvBase (*elem*)

__init__ (*elem*)
x.__init__(...) initializes x; see help(type(x)) for signature

exception TestEnvNotSetError
Raised when the test environment hasn't been setup, i.e. NXPY_TEST_DIR is not set.

get_data (*test*, *package*)

get_env (*test*, *package*)

16.2 `log` - Log configuration for tests

Logging configuration for tests.

16.3 `test` - Support functions for running tests

Unittest utility functions.

skipIfNotAtLeast(*version*)
Skip the current test if the current Python release is lower than *version*.

skipIfNotAtMost(*version*)
Skip the current test if the current Python release is higher than *version*.

testClasses(**classes*)
Runs all tests defined in the given *classes*.

testModules(**modules*)
Runs all tests defined in the given *modules*.

CHAPTER 17

`xml` - XML related utility classes

XML related utility classes.

17.1 `util` - Various utilities

CHAPTER 18

core - Common library infrastructure

18.1 error - nxpy's exception hierarchy

CHAPTER 19

Running the tests

Nxpy tests are based on the standard `unittest` module. As recent features are used the `unittest2` backport is required with Python 2.6. Tests reside in `_test` subdirectories of the library package directory. For each `module` module tests should be found in a `test_module` module.

Tests may be run for all supported Python versions installed on your system and present in your `PATH` environment variable with `tox` and `pytest`. Dependencies from libraries available from PyPI are automatically installed by tox. The following additional requirements should also be fulfilled:

- The `nxpy-maven` library requires that a recent version of Maven be present in your `PATH`.
- The `nxpy-svn` library requires that a recent version of Subversion be present in your `PATH`.

CHAPTER 20

Generating the documentation

Nxpy's documentation is written in reStructuredText and rendered with Sphinx.

CHAPTER 21

Creating new releases

Libraries should only be released when needed. Although a combined package is not released, its configuration should be updated to reflect library changes.

Assuming all changes to code and documentation have been pushed to the upstream repository, the basic steps for the creation of a new library release are:

- Run `tox` in the root directory of your development checkout. Ideally you should be developing against the most recent supported Python release on one of the supported platforms. As long as Python 2.7 is supported tests should be run against it too.
- Update any release related configuration file, e.g.:
  - `CHANGES.txt`
  - `README.rst`
  - `setup.py`
- Commit to `master` any remaining change and push upstream. This should trigger GitHub Actions tests against all the supported Python versions.
- Bump the library version number according to semantic versioning: increment the minor version element if the new release includes API breaking changes, the increment version element otherwise. Add `rc1` to the release number to mark the fact that this is a release candidate.
- Run `python setup.py sdist bdist_wheel` in the library's root directory. Check the contents of the resulting packages in the `dist` directory.
- Run `twine check dist/*`. Fix any resulting problem.
- Run `twine upload --repository-url https://test.pypi.org/legacy/ dist/*` to upload the library to Test PyPI. You will need a Test PyPI account for that.
- Create a new virtualenv and install the new library in it with `pip install --index-url https://test.pypi.org/simple/ --no-deps --pre <<Library>>`.
- Perform a minimal test.
- Remove the `build`, `dist` and `<<Library>>.egg-info` directories.

- Remove the `rc1` prefix from the version number in the `setup.py` file.
- Commit and push all outstanding changes.
- Run `python setup.py sdist bdist_wheel` again and check the contents of the resulting packages.
- Run `twine check dist/*`.
- Run `twine upload dist/*` to upload the library to PyPI. You will need a PyPI account.
- Create another virtualenv and install the new library in it with `pip install <<Library>>`.
- Perform a last test.

Supported Python versions
2.7
3.6
3.7
3.8
3.9
3.10

Supported platforms
Linux
MacOS
Windows 7 or later

The libraries are being developed with Python 3.10 so as to be compatible with Python 2.7. Tests are run and most modules work also with 3.6, 3.7, 3.8 and 3.9. Some should still work with versions as early as 3.2 and 2.5. There is no immediate plan to remove Python 2.x support, but in general earlier releases will only be supported as long as external tools, such as GitHub Actions or pip, keep supporting them.

Originally the libraries resided on SourceForge and were distributed as a single package. Starting from release 1.0.0 each library is being packaged separately even though they are all hosted within the same project on GitHub.

The Nxpy logo was drawn by Claudia Romano.

CHAPTER 22

Indices and tables

- genindex
- modindex
- search

Python Module Index

n

Index

Symbols

A

B

C

D

E

G

I

L

M

N

O

P

R

S

T

U

V

W