Updater4Pyi Documentation

Release 0.7

Philippe Faist

July 22, 2015

Contents

Contents:

CHAPTER 1

Quick Start

Simple example: say you want to add an auto-update feature to your python/PyQt4 program, hosted on github. Add the following lines into your python program:

```
from updater4pyi import upd_source, upd_core
from updater4pyi.upd_iface_pyqt4 import UpdatePyQt4Interface

swu_source = upd_source.UpdateGithubReleasesSource('githubusername/githubproject')
swu_updater = upd_core.Updater(current_version=...,
                               update_source=swu_source)
swu_interface = UpdatePyQt4Interface(swu_updater,
                                     progname='Your Project Name',
                                     ask_before_checking=True,
                                     parent=QApplication.instance())
```

You should also add the hook *hook-updater4pyi.py* into your PyInstaller hooks path (presumably into your custom hooks). Then your PyInstaller-packaged program will ask the user if they agree to auto-update, then regularly check for updates and ask to install them. You may need to fiddle a bit with the naming patterns of your releases (specify a second argument to the *UpdateGithubReleasesSource* constructor).

The library is designed to be highly flexible. It should be easy to write update sources for any other type of sources, such as XML application feeds etc. Likewise, the library does *not* rely on PyQt4 (it just provides a convenient interface for those PyQt4 apps). It should be simple to write an interface in whater Gui toolkit you prefer–see `upd_iface.UpdateGenericGuiInterface`.

1.1 Installation & Usage

Updater4pyi is available on PyPI, so the recommended installation is through there.

You will also need to install one hook for pyinstaller. If you have a custom hook directory in your project, you can place the file named *hook-updater4pyi.py* it there; otherwise, locate your pyinstaller *hooks* directory and copy the file *hook-updater4pyi.py* in there.

To use updater4pyi in your programs, you need to:

- describe where to look for updates (the *sources*)
- instantiate the updater, giving it the corresponding sources
- create an interface, which will interact with the user.

For example, the [bibolamazi project](https://github.com/phfaist/bibolamazi) uses the updater4pyi framework, and the relevant lines in there are:

```
from updater4pyi import upd_source, upd_core
from updater4pyi.upd_iface_pyqt4 import UpdatePyQt4Interface

swu_source = upd_source.UpdateGithubReleasesSource('phfaist/bibolamazi')
swu_updater = upd_core.Updater(current_version=...,
                               update_source=swu_source)
swu_interface = UpdatePyQt4Interface(swu_updater,
                                     progname='Bibolamazi',
                                     ask_before_checking=True,
                                     parent=QApplication.instance())
```

Then, you need to make sure that pyinstaller can find updater4pyi, i.e. it has to be in your python path.

1.2 Sources

At the moment, there are only two source types

- github releases. See upd_source.UpdateGithubReleasesSource. You can specify naming patterns as regexp's to match the release files with corresponding platforms
- local directory source (used for debugging)

However, it is straightforward to write your own source. Look at *upd_source.py* to get an idea. If you do so, it would be great to contribute it to updater4pyi so that other people can profit!

1.3 Security

This library supports downloads through HTTPS with certificate verification, making the download and update process secure. The default root certificate is provided with this package, and can be changed to include your custom one if needed.

1.4 Interfaces

At the moment, there are the following interfaces for updates:

- A simple console interface, running at each program start, that prompts the user to check for updates. Not very useful, more meant for debugging.
- A full-featured abstract GUI generic interface. This interface does not depend on any GUI, and abstracts out tasks such as timing, saving settings and message boxes which need to be implemented by subclasses. This class is meant to be used as a base class to write updater interfaces for other GUI systems (wxWidgets, ...)
- A PyQt4 interface is provided based on the generic GUI interface mentioned in the previous point.

CHAPTER 2

Python API - updater4pyi package

2.1 updater4pyi.upd_core module

class updater4pyi.upd_core.**FileToUpdate**(*fn*, *reltype*, *executable*)
Bases: tuple

executable
Alias for field number 2

fn
Alias for field number 0

reltype
Alias for field number 1

class updater4pyi.upd_core.**Updater**(*current_version*, *update_source*)
Bases: object

The main Updater object.

This class is responsible for actually checking for updates and performing the software update.

It does not take care of scheduling the checks, however. That's done with an UpdaterInterface.

This class needs to be specified a *source* for updates. See upd_source.UpdateSource.

SPECIAL_ZIP_FILES = ('_updater4pyi_metainf.json', '_METAINF')

check_for_updates()
Perform an update check.

Queries the source for possible updates, which matches our system. If a software update is found, then a upd_source.BinReleaseInfo object is returned, describing the software update. Otherwise, if no update is available, *None* is returned.

current_version()
Return the current version of the running program, as given to the constructor.

download_file(*theurl*, *fdst*)
Download the file given at location *theurl* to the destination file *fdst*.

You may reimplement this function to customize the download process. Check out *upd_downloader.url_opener* if you want to download stuff from an HTTPS url, it may be useful.

The default implementation downloads the file with the *upd_downloader* utility which provides secure downloads with certificate validation for HTTPS downloads.

This function should return nothing. If an error occurs, this function should raise an *IOError*.

file_to_update()
Return the file one should update. See *determine_file_to_update()*.

install_update(*rel_info*)
Install a given update. *rel_info* should be a `upd_source.BinReleaseInfo` returned by *check_for_updates()*.

The actual updates are downloaded by calling *download_file()*. You may overload that function if you need to customize the download process. You may also override *verify_download()* to implement some download integrity verification.

This function does not return anything. If an error occurred, `upd_defs.Updater4PyiError` is raised.

restart_app()
Utility to restart the application. This is meant for graphical applications which start in the background.

The application exit is done by calling `sys.exit(0)`.

update_source()
Return the source given to the constructor.

verify_download(*rel_info*, *tmpfile*)
Verify the integrity of the downloaded file. Return *True* if the download succeeded or *False* if not.

Arguments:

- *rel_info* is the release information as a `upd_source.BinReleaseInfo` instance, as given to *install_update()*.
- *tmpfile* is a python `tempfile.NamedTemporaryFile` instance where the file was downloaded. This function should in principle check the validity of the contents of this file.

You may reimplement this function to implement integrity check. The default implementation does nothing and returns *True*.

Don't raise arbitrary exceptions here because they might not be caught. You may raise `upd_defs.Updater4PyiError` for serious errors, though.

updater4pyi.upd_core.**determine_file_to_update**()
Inspects the program currently running, and determines the location of the file one should replace in the event of a software update.

Returns a named tuple *FileToUpdate(fn=.., reltype=.., executable=...)*. The values are:

- *fn*: the actual file we should update. This could be a directory in the case of a onedir PyInstaller package. For a MAC OS X bundle, it is the *XYZ.app* file.
- *reltype*: the release type we have. This may be one of `upd_defs.RELTYPE_EXE`, `upd_defs.RELTYPE_ARCHIVE`, `upd_defs.RELTYPE_BUNDLE_ARCHIVE`.
- *executable*: the actual executable file. This may be different from *fn*, for example in Mac OS X bundles, where *executable* is the actual file being executed within the bundle.

2.2 updater4pyi.upd_defs module

updater4pyi.upd_defs.**RELTYPE_ARCHIVE = 2**
An archive containing several files. The archive should be extracted into a directory.

updater4pyi.upd_defs.**RELTYPE_BUNDLE_ARCHIVE = 3**
An archive containing a Mac OS X application bundle.

updater4pyi.upd_defs.**RELTYPE_EXE = 1**
A single executable. For example, a MS Windows .exe file or a linux executable.

updater4pyi.upd_defs.**RELTYPE_UNKNOWN = 0**
Unknown release type.

exception updater4pyi.upd_defs.**Updater4PyiError**(*msg*)
Bases: exceptions.Exception

An exception class used to signify an error in the installation of a software update, for example.

However, if you're not digging into the internals of the update interface, you probably won't even have to bother with catching these. See also Updater and UpdateInterface.

2.3 updater4pyi.upd_downloader module

Utilities to download files over secure HTTPS connections, with *server certificate verification*.

See Validate SSL certificates with Python and and this solution on Stack Overflow.

class updater4pyi.upd_downloader.**ValidHTTPSConnection**(**args*, ***kwargs*)
Bases: httplib.HTTPConnection

HTTPS connection based on httplib.HTTPConnection, with complete certificate validation based on known root certificates packaged with the program.

The root certificate file is given in the module-level variable CERT_FILE. Note you may use util.resource_path() to get a file in the pyinstaller bundle.

connect()
Connect to a host on a given (SSL) port.

default_port = 443

class updater4pyi.upd_downloader.**ValidHTTPSHandler**(*debuglevel=0*)
Bases: urllib2.HTTPSHandler

A HTTPS urllib2 handler using ValidHttpsConnection, i.e. with correct server certificate validation.

https_open(*req*)

updater4pyi.upd_downloader.**url_opener = <urllib2.OpenerDirector instance>**
The URL opener obtained with *urllib2.build_opener*, with valid HTTPS server certificate validation.

2.4 updater4pyi.upd_iface module

class updater4pyi.upd_iface.**UpdateConsoleInterface**(*updater*, *ask_before_checking=False*, ***kwargs*)
Bases: *updater4pyi.upd_iface.UpdateInterface*

A very simple *UpdateInterface* implementation that checks for updates each time the program is run. This is mostly meant for debugging purposes.

start()

class updater4pyi.upd_iface.**UpdateGenericGuiInterface**(*updater*, *ask_before_checking=True*, ***kwargs*)

Bases: *updater4pyi.upd_iface.UpdateInterface*

all_settings()
Utility to get all settings. Useful for subclasses; this doesn't need to be reimplemented.

ask_first_time()
Subclasses should prompt the user whether they want to regularly look for updates.

This is prompted to the user only if the main program set *ask_before_checking* to *True* in the constructor of this object.

Return TRUE if the program should regularly check for updates, or FALSE if not.

ask_to_restart()
Subclasses should prompt the user to restart the program after a successful update.

Return TRUE if the program should be restarted, or FALSE if not.

ask_to_update(*rel_info*)
Subclasses should prompt the user whether they want to install the update *rel_info* or not.

Note: Interfaces may also present additional buttons such as "Never check for updates", or "Skip this update", and set properties and/or settings accordingly with e.g. *setCheckForUpdatesEnabled()*.

Return TRUE if the program should be restarted, or FALSE if not.

checkForUpdatesEnabled()

checkInterval()

check_for_updates()
Perform a possible update check. You don't have to reimplement this function, the default implementation should be good enough and relies on your implementations of *ask_to_update()* and *ask_to_restart()*.

If the update check isn't due yet, this function does not do the update check. If you want to force an update check, call *do_check_for_updates()*.

do_check_for_updates()
Actually perform the udpate check. Call this function if you want to force an update check even though it's not yet due. If you want to periodically possibly check only if a check is due, then call *check_for_updates()* instead.

Returns:

- *None* if we asked the user for the first time if they want to check regularly for updates, and they refused.
- *False* if no new update is available
- **a tuple if a new update is available:**
  - *(True, rel_info)* if the user installed the update but did not restart the app;
  - *(False, rel_info)* if the user declined to install the update now
- the tuple *(False, None, error_str)* if an error occurred while checking for updates.

initCheckDelay()

is_check_now_due(*tolerance=datetime.timedelta(0, 10)*)

lastCheck()

load_settings(*keylist*)
Subclasses may reimplement this function to cusomize where and how the settings are stored, usually using a toolbox-specific utility, such as QSettings in PyQt4.

save_settings(*d=None*)
Save the given settings in the dictionary *d* to some local settings. If d is None, then all settings should be saved, effectively taking *d* to be the dictionary returned by *all_settings()*.

schedule_next_update_check()

setCheckForUpdatesEnabled(*enabled*, *save=True*, *schedule_check=True*)

setCheckInterval(*check_interval*, *save=True*)

setInitCheckDelay(*init_check_delay*, *save=True*)

setLastCheck(*last_check*, *save=True*)

set_timeout_check(*interval_timedelta*)
Subclasses should reimplement this function to call the function *check_for_updates()* after *interval_timedelta*. *interval_timedelta* is a *datetime.timedelta* object.

start()

timedelta_remaining_to_next_check()

class updater4pyi.upd_iface.**UpdateInterface**(*updater*, *progname=None*, ***kwargs*)
Bases: object

start(***kwargs*)
Start being aware of wanting to check for updates. It is up to the interface to decide when to check for updates, how often, etc. For example, a console interface would check right away, while a GUI might first load the application, and set a timer to check later, so that startup is not slowed down by the update check.

2.5 updater4pyi.upd_iface_pyqt4 module

2.6 updater4pyi.upd_log module

Set up a minimal logger. To integrate logging in your application, configure your Python logging as you wish. Updater4Pyi gets its logger by calling `logging.getLogger('updater4pyi')`, i.e. the Updater4Pyi's logger is called 'updater4pyi'.

updater4pyi.upd_log.**setup_logger**(*level=20*)
A utility function that you can call to set up a simple logging to the console. No hassles.

2.7 updater4pyi.upd_source module

This module defines how Updater4Pyi accesses *sources*, i.e. how information about the software updates are queried.

The base class is *UpdateSource*. Check out the *github.com releases* source UpdateGithubRelasesSource. For testing, you may want to try out *UpdateLocalDirectorySource*.

Information about individual releases are provided as *BinReleaseInfo* objects.

Some sources allow to determine information about releases from the file name. The class *ReleaseInfoFromNameStrategy* is provided for this purpose.

class updater4pyi.upd_source.**BinReleaseInfo**(*version=None*, *filename=None*, *url=None*, *reltype=0*, *platform=None*, ***kwargs*)

Bases: object

A description of a release. This includes the release type (executable, archive, archived Mac OS X bundle), the URL at which it can be downloaded, the platform, the version etc.

Update Sources (see *UpdateSource*) return *BinReleaseInfo* objects to describe available releases. You may even reimplement this class if you need specific needs for determining release information. Note that within Updater4Pyi internals, all standard fields (version, filename, url, reltype and platform) are always queried using the accessor functions (*get_version()*, *get_filename()*, *get_url()*, etc.), so you could even determine that information dynamically if you really wanted to do complicated things.

You may also want to check out *ReleaseInfoFromNameStrategy* for automatically determining release information from the file name. It's also highly customizable.

Arbitrary information about the release may be stored in this class, too.

get_filename()
Return the *filename* set in the constructor.

get_platform()
Return the *platform* set in the constructor.

get_reltype()
Return the *reltype* set in the constructor.

get_url()
Return the *url* set in the constructor.

get_version()
Return the *version* set in the constructor.

class updater4pyi.upd_source.**IgnoreArgument**

class updater4pyi.upd_source.**ReleaseInfoFromNameStrategy**(*patterns*, **args*, ***kwargs*)

Bases: object

Base class for a strategy to identify release details from a file name.

The information about a specific release (as a *BinReleaseInfo* object) can be obtained by calling *get_release_info()* with a filename and any additional information to include in the *BinReleaseInfo* object.

Some sources need such a stategy, such as *UpdateLocalDirectorySource* and UpdateGithubRelasesSource.

The *patterns* (see constructor) should be a list (or tuple) of patterns and rules constructed with *relpattern()*. The patterns are tested in the given order until a match is found. See *relpattern()* on information how to construct these rules.

Actually, each pattern (see return value of *relpattern()*) is a 2-element tuple *(regexpattern, callable)*. The *regexpattern* may be a precompiled regex object (i.e. with *re.compile*), or it may be a string, in which case it is compiled with the *re.IGNOREFLAGS* set. The *callable* is any python callable should have the signature *callable(m, filename, url, **kwargs)* accepting a regexp match object, the file name, the URL at which the release can be accessed, and any keyword arguments that should be passed to the *BinReleaseInfo* constructor. The callable should return a new *BinReleaseInfo* instance.

get_release_info(*filename*, *url*, ***kwargs*)
Return a *BinReleaseInfo* instance from information extracted from the filename (and possibly further information provided by url and keyword arguments).

Additional arguments are passed to the *BinReleaseInfo* constructor untouched. Patterns may refer to these additional fields to help identify the release information.

The list of patterns (see constructor) are tested in order until a match is found. At that point, the release information is compiled and returned.

If none of the patterns matched, then *None* is returned.

class updater4pyi.upd_source.**UpdateGithubReleasesSource**(*github_user_repo*, *naming_strategy=None*, **args*, ***kwargs*)

Bases: *updater4pyi.upd_source.UpdateSource*

Updates will be searched for in as releases of a github repo.

get_releases(*newer_than_version=None*, ***kwargs*)

Reimplemented from *UpdateSource.get_releases()*.

The information is retrieved using the github API, for example at https://api.github.com/repos/phfaist/bibolamazi/releases. This returns a JSON dictionary with information on the various releases. Each *release* has fields and a list of *assets*. (See Github API Documentation.)

Additional information such as the github release label is provided in each *BinReleaseInfo* instance:

```
rel_name             = the 'name' field of the release JSON dictionary
relfile_label        = the 'label' field of the release JSON dictionary
rel_description      = the 'body' field of the release JSON dictionary
rel_tag_name         = the 'tag_name' field of the release JSON dictionary
rel_html_url         = the 'html_url' field of the release JSON dictionary
relfile_content_type = the 'content_type' field of the asset JSON dictionary
```

class updater4pyi.upd_source.**UpdateLocalDirectorySource**(*source_directory*, *naming_strategy=None*, **args*, ***kwargs*)

Bases: *updater4pyi.upd_source.UpdateSource*

Updates will be searched for in a local directory. Useful for debugging.

Will check in the given *source_directory* directory for updates. Files should be organized in subdirectories which should be version names, e.g.:

```
1.0/
  binary-macosx[.zip]
  binary-linux[.zip]
  binary-win[.exe|.zip]
1.1/
  binary-macosx[.zip]
  binary-linux[.zip]
  binary-win[.exe|.zip]
...
```

This updater source is mostly for debugging purposes. There's no real-life utility I can see...

get_releases(*newer_than_version=None*, ***kwargs*)

class updater4pyi.upd_source.**UpdateSource**(**args*, ***kwargs*)

Bases: object

Base abstract class for an update source.

An update source takes care of accessing a e.g. repository or online server, and querying for available updates. It should be capable of returning information about available releases in the form of *BinReleaseInfo* objects.

Subclasses should reimplement the main function *get_releases()*.

add_release_filter(*filt*)
Adds a *release filter* to ignore some releases.

filt must be a callable which takes a positional argument, the release information object (`BinReleaseInfo` object). It should return *True* for keeping the release or *False* for ignoring it.

This could be, for example, to ignore beta releases.

It is the responsibility of the subclass to test release filters, for example using the `test_release_filters()` helper function.

get_releases(*newer_than_version=None*, ***kwargs*)
Should return a list of `BinReleaseInfo` describing available releases. If *newer_than_version* argument is provided, then this function should ignore releases older or equal to the given argument. (Check out `util.parse_version()` to parse and compare versions.)

Note that for filters to work, the subclass must explicitly test each candidate release with *test_release_filters()*, and ignore the release if that function returns *False*.

This function should return *None* if no release information could be obtained (e.g. not connected to the internet). This function should return an empty list if no new updates are available.

test_release_filters(*relinfo*)
Returns *True* if *relinfo* should be included in the releases given the installed filters, otherwise *False*. Note that the platform and the version selection are not implemented by filters. Filters are meant to choose between different editions, or to filter out/include beta unstable releases.

It is the responsibility of the subclass to test release filters for example with this function.

class `updater4pyi.upd_source.`**UpdateSourceDevelopmentReleasesFilter**(*include_devel_releases=False*, *regexname=None*)

Bases: `object`

Simple filter for including/not including developemnt releases.

You can specify a class instance to `UpdateSource.add_release_filter()`.

includeDevelReleases()

setIncludeDevelReleases(*include*)

`updater4pyi.upd_source.`**relpattern**(*re_pattern*, *reltype=0*, *platform=None*, ***kwargs*)
Construct a rule to set release information depending on filename and further attributes.

The rule applies to all releases whose *filename* matches the given regex pattern *re_pattern*. The latter should be either a precompiled regexp pattern (with *re.compile*) or given as a string.

All further arguments specify which rules to apply to set attributes for this release information.

The release information attribute rules are applied as follows:

1. the rules given as additional keyword arguments to *relpattern* are processed;
2. the values for *filename* and *url* given to `get_release_info()` are set;
3. the rules for *platform* and *reltype* given to this function are processed;
4. the values given as additional keyword arguments to `get_release_info()` are set.

(Not sure why I can justify this order here, but I'm afraid of changing it.)

A *rule* for setting an attribute may be one of the following:

- a fixed value: the fixed value is set to that attribute

- a python callable: the callable is called (no, you don't say?) and its return value is used as the value of the attribute. If the callable returned *IgnoreArgument*, then the rule is ignored (no one would have guessed). The callable may accept any combination of the following keyword arguments:
  - 'm' is the regex match object form the regex that matched the filename, and may be used to extract groups for example;
  - 'd' is a dictionary of values passed as additional keyword arguments to *get_release_info()*;
  - 'x' is the dictionary of attributes constructed so far (by the given values and rules being processed).

The following pattern will test for a filename of the form 'filename-VERSION-PLATFORM.EXTENSION', 'filename-VERSION.EXTENSION', 'filename-PLATFORM.EXTENSION' or 'filename.EXTENSION'. The information corresponding to VERSION and PLATFORM are set if they were found in the filename, otherwise we assume they'll be figured out by some other means. The release type (*reltype*) is guessed depending on the extension of the filename and the platform. The example is:

```
    pattern1 = relpattern(
        r'(-(?P<version>\d+[\w.]+))?(-(?P<platform>macosx|linux|win))?\.(?P<ext>[a-zA-Z]+)$',
        version=lambda m: m.group('version') if m.group('version') else IgnoreArgument,
        platform=lambda m: m.group('platform') if m.group('platform') else IgnoreArgument,
        reltype=lambda m, x: guess_reltype(m, x)
        )

    ...

    def guess_reltype(m, x):
        ext = m.group('ext').lower()
        if ext == 'zip' and x.get('platform', '') == 'macosx':
            return RELTYPE_BUNDLE_ARCHIVE
        if ext in ('exe', 'bin', 'run'):
            return RELTYPE_EXE
        if ext in ('zip', 'tgz'):
            return RELTYPE_ARCHIVE
        return IgnoreArgument
```

2.8 updater4pyi.upd_version module

updater4pyi.upd_version.**version_str = '0.7'**

The current version string.

2.9 updater4pyi.util module

A collection of various utilities.

updater4pyi.util.**applescript_quote**(*x*)

updater4pyi.util.**bash_quote**(*x*)

updater4pyi.util.**dirIsWritable**(*directory*)

updater4pyi.util.**ensure_datetime**(*x*)

updater4pyi.util.**ensure_timedelta**(*x*)

updater4pyi.util.**fileIsWritable**(*fn*)

updater4pyi.util.**getbool**(*x*)
Utility to parse a string representing a boolean value.

If *x* is already of integer or boolean type (actually, anything castable to an integer), then the corresponding boolean convertion is returned. If it is a string-like type, then it is matched against something that looks like 't(rue)?', '1', 'y(es)?' or 'on' (ignoring case), or against something that looks like 'f(alse)?', '0', 'n(o)?' or 'off' (also ignoring case). Leading or trailing whitespace is ignored. If the string cannot be parsed, a `ValueError` is raised.

updater4pyi.util.**ignore_exc**(*f*, *exc=(<type 'exceptions.Exception'>,)*, *value_if_exc=None*)

updater4pyi.util.**is_linux**()

updater4pyi.util.**is_macosx**()

updater4pyi.util.**is_win**()

updater4pyi.util.**locationIsWritable**(*path*)

updater4pyi.util.**parse_version**(*s*)
Convert a version string to a chronologically-sortable key

This function is based on code from setuptools. (I didn't find any license text to copy from that project, but on PyPI it states that the license is 'PSF or ZPL'.)

This is a rough cross between distutils' StrictVersion and LooseVersion; if you give it versions that would work with StrictVersion, then it behaves the same; otherwise it acts like a slightly-smarter LooseVersion. It is *possible* to create pathological version coding schemes that will fool this parser, but they should be very rare in practice.

The returned value will be a tuple of strings. Numeric portions of the version are padded to 8 digits so they will compare numerically, but without relying on how numbers compare relative to strings. Dots are dropped, but dashes are retained. Trailing zeros between alpha segments or dashes are suppressed, so that e.g. "2.4.0" is considered the same as "2.4". Alphanumeric parts are lower-cased.

The algorithm assumes that strings like "-" and any alpha string that alphabetically follows "final" represents a "patch level". So, "2.4-1" is assumed to be a branch or patch of "2.4", and therefore "2.4.1" is considered newer than "2.4-1", which in turn is newer than "2.4".

Strings like "a", "b", "c", "alpha", "beta", "candidate" and so on (that come before "final" alphabetically) are assumed to be pre-release versions, so that the version "2.4" is considered newer than "2.4a1".

Finally, to handle miscellaneous cases, the strings "pre", "preview", and "rc" are treated as if they were "c", i.e. as though they were release candidates, and therefore are not as new as a version string that does not contain them, and "dev" is replaced with an '@' so that it sorts lower than than any other pre-release tag.

updater4pyi.util.**path2url**(*p*)

updater4pyi.util.**resource_path**(*relative_path*)
Get absolute path to resource, works for dev and for PyInstaller

updater4pyi.util.**run_as_admin**(*argv*)

updater4pyi.util.**run_win**(*argv*, *needs_sudo=False*, *wait=True*, *cwd=None*)
Run a process on windows.

Returns: the exit code of the process if *wait* is *True*, or the PID of the running process if *wait* is *False*.

updater4pyi.util.**simple_platform**()

updater4pyi.util.**which**(*name*, *flags=1*, *usecache=True*, *firstresult=True*)
Search PATH for executable files with the given name.

This function is based on code from twisted (see copyright notice in source code of this function).

On newer versions of MS-Windows, the PATHEXT environment variable will be set to the list of file extensions for files considered executable. This will normally include things like ".EXE". This fuction will also find files with the given name ending with any of these extensions.

On MS-Windows the only flag that has any meaning is os.F_OK. Any other flags will be ignored.

@type name: C{str} @param name: The name for which to search.

@type flags: C{int} @param flags: Arguments to L{os.access}.

@rtype: C{list} @param: A list of the full paths to files found, in the order in which they were found.

updater4pyi.util.**which_clear_cache**()

updater4pyi.util.**winshell_quote**(*x*)

CHAPTER 3

Credits and Copyright

I will try to maintain this project as much as possible, but note that it is not my priority. Contributions are welcome.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

U

V

W