pynsist Documentation

Release 1.12

Thomas Kluyver

Aug 19, 2017

Contents

pynsist is a tool to build Windows installers for your Python applications. The installers bundle Python itself, so you can distribute your application to people who don't have Python installed.

At present, pynsist requires Python 3.3 or above, or Python 2.7.

CHAPTER 1

Quickstart

1. Get the tools. Install NSIS, and then install pynsist from PyPI by running `pip install pynsist`.
2. Write a config file `installer.cfg`, like this:

```
[Application]
name=My App
version=1.0
# How to launch the app - this calls the 'main' function from the 'myapp' package:
entry_point=myapp:main
icon=myapp.ico

[Python]
version=3.4.0

[Include]
# Importable packages that your application requires, one per line
packages = requests
     bs4
     html5lib

# Other files and folders that should be installed
files = LICENSE
    data_files/
```

   See *The Config File* for more details about this.

3. Run `pynsist installer.cfg` to generate your installer. If `pynsist` isn't found, you can use `python -m nsist installer.cfg` instead.

CHAPTER 2

Contents

The Config File

All paths in the config file are relative to the directory where the config file is located, unless noted otherwise.

Application section

name
: The user-readable name of your application. This will be used for various display purposes in the installer, and for shortcuts and the folder in 'Program Files'.

version
: The version number of your application.

publisher (optional)
: The publisher name that shows up in the *Add or Remove programs* control panel.

  New in version 1.10.

entry_point
: The function to launch your application, in the format `module:function`. Dots are allowed in the module part. pynsist will create a script like this, plus some boilerplate:

```
from module import function
function()
```

script (optional)
: Path to the Python script which launches your application, as an alternative to `entry_point`.

  Ensure that this boilerplate code is at the top of your script:

```
#!python3.3
import sys
sys.path.insert(0, 'pkgs')
```

The first line tells it which version of Python to run with. If you use binary packages, packages compiled for Python 3.3 won't work with Python 3.4. The other lines make sure it can find the packages installed along with your application.

target (optional)
parameters (optional)
Lower level definition of a shortcut, to create start menu entries for help pages or other non-Python entry points. You shouldn't normally use this for Python entry points.

Note: Either `entry_point`, `script` or `target` must be specified, but not more than one. Specifying `entry_point` is normally easiest and most reliable.

---

icon (optional)
Path to a `.ico` file to be used for shortcuts to your application. Pynsist has a default generic icon, but you probably want to replace it.

console (optional)
If `true`, shortcuts will be created using the `py` launcher, which opens a console for the process. If `false`, or not specified, they will use the `pyw` launcher, which doesn't create a console.

extra_preamble (optional)
Path to a file containing extra Python commands to be run before your code is launched, for example to set environment variables needed by pygtk. This is only valid if you use `entry_point` to specify how to launch your application.

If you use the Python API, this parameter can also be passed as a file-like object, such as `io.StringIO`.

Shortcut sections

One shortcut will always be generated for the application. You can add extra shortcuts by defining sections titled `Shortcut` *`Name`*. For example:

```
[Shortcut IPython Notebook]
entry_point=IPython.html.notebookapp:launch_new_instance
icon=scripts/ipython_nb.ico
console=true
```

entry_point
script (optional)
icon (optional)
console (optional)
target (optional)
parameters (optional)
extra_preamble (optional)
These options all work the same way as in the Application section.

Microsoft offers guidance on what shortcuts to include in the Start screen/menu. Most applications should only need one shortcut, and things like help and settings should be accessed inside the app rather than as separate shortcuts.

Command sections

New in version 1.7.

Your application can install commands to be run from the Windows command prompt. This is not standard practice for desktop applications on Windows, but if your application specifically provides a command line interface, you can define one or more sections titled `Command name`:

```
[Command guessnumber]
entry_point=guessnumber:main
```

If you use this, the installer will modify the system `PATH` environment variable.

entry_point
: As with shortcuts, this specifies the Python function to call, in the format `module:function`.

extra_preamble (optional)
: As for shortcuts, a file containing extra code to run before importing the module from `entry_point`. This should rarely be needed.

Python section

version
: The Python version to download and bundle with your application, e.g. `3.4.3`. Python 3.3 or later and 2.7 are supported.

bitness (optional)
: `32` or `64`, to use 32-bit (x86) or 64-bit (x64) Python. On Windows, this defaults to the version you're using, so that compiled modules will match. On other platforms, it defaults to 32-bit.

format (optional)

- `installer` includes a copy of the Python MSI installer in your application and runs it at install time, setting up Python systemwide. This is the default for Python up to 3.5.
- `bundled` includes an embeddable Python build, which will be installed as part of your application. This is available for Python 3.5 and above, and is the default for Python 3.6 and above.

Changed in version 1.9: The default switched to `bundled` for Python 3.6 and above.

include_msvcrt (optional)
: This option is only relevant with `format = bundled`. The default is `true`, which will include an app-local copy of the Microsoft Visual C++ Runtime, required for Python to run. The installer will only install this if it doesn't detect a system installation of the runtime.

  Setting this to `false` will not include the C++ Runtime. Your application may not run for all users until they install it manually (download from Microsoft). You may prefer to do this for security reasons: the separately installed runtime will get updates through Windows Update, but app-local copies will not.

  Users on Windows 10 should already have the runtime installed systemwide, so this does won't affect them. Users on Windows Vista, 7, 8 or 8.1 *may* already have it, depending on what else is installed.

  New in version 1.9.

Bundled Python

New in version 1.6: Support for bundling Python into the application.

Using `format = bundled`, an embeddable Python build will be downloaded at build time and packaged along with the application. When the installer runs, it will create a `Python` subfolder inside the install directory with the files Python needs to run.

This has the advantage of producing smaller, quicker installers (~7.5 MB for a trivial application), and more standalone installations. But it has a number of limitations:

- This option is only available for Python 3.5 and above. These versions of Python have dropped support for Windows XP, so your application will only work on Windows Vista and newer.
- Installing in Windows Vista to 8.1 (inclusive) may install an app-local copy of the Visual C++ runtime (see above). This isn't needed on Windows 10, which includes the necessary files.
- The embeddable Python builds don't include `tkinter`, to save space. Applications with a tkinter GUI can't easily use bundled Python. Workarounds may be found in the future.
- The user cannot easily install extra Python packages in the application's Python. If your application has plugins based on Python packages, this might require extra thought about how and where plugins are installed.

Include section

To write these lists, put each value on a new line, with more indentation than the line with the key:

```
key=value1
  value2
  value3
```

packages (optional)
A list of importable package and module names to include in the installer. Specify only top-level packages, i.e. without a `.` in the name.

pypi_wheels (optional)
A list of packages to download from PyPI, in the format `name==version`. These must be available as wheels; Pynsist will not try to download sdists or eggs.

New in version 1.7.

files (optional)
Extra files or directories to be installed with your application.

You can optionally add `> destination` after each file to install it somewhere other than the installation directory. The destination can be:

- An absolute path on the target system, e.g. `C:\\` (but this is not usually desirable).
- A path starting with `$INSTDIR`, the specified installation directory.
- A path starting with any of the constants NSIS provides, e.g. `$SYSDIR`.

The destination can also include `${PRODUCT_NAME}`, which will be expanded to the name of your application.

For instance, to put a data file in the (32 bit) common files directory:

```
[Include]
files=mydata.dat > $COMMONFILES
```

exclude (optional)
Files to be excluded from your installer. This can be used to include a Python library or extra directory only partially, for example to include large monolithic python packages without their samples and test suites to achieve a smaller installer file.

Please note:

- The parameter is expected to contain a list of files *relative to the build directory*. Therefore, to include files from a package, you have to start your pattern with `pkgs/<packagename>/`.

- You can use wildcard characters like `*` or `?`, similar to a Unix shell.
- If you want to exclude whole subfolders, do *not* put a path separator (e.g. `/`) at their end.
- The exclude patterns are only applied to packages and to directories specified using the `files` option. If your `exclude` option directly contradicts your `files` or `packages` option, the files in question will be included (you can not exclude a full package/extra directory or a single file listed in `files`).

Example:

```
[Include]
packages=PySide
files=data_dir
exclude=pkgs/PySide/examples
  data_dir/ignoredfile
```

Build section

`directory (optional)`
The build directory. Defaults to `build/nsis/`.

`installer_name (optional)`
The filename of the installer, relative to the build directory. The default is made from your application name and version.

`nsi_template (optional)`
The path of a template .nsi file to specify further details of the installer. The default template is part of pynsist.

This is an advanced option, and if you specify a custom template, you may well have to update it to work with future releases of Pynsist.

See the NSIS Scripting Reference for details of the NSIS language, and the Jinja2 Template Designer Docs for details of the template format. Pynsist uses templates with square brackets (`[]`) instead of Jinja's default curly braces (`{}`).

Installer details

The installers pynsist builds do a number of things:

1. Unpack and run the Python .msi installer for the version of Python you specified.
2. Unpack and run the .msi installer for the py launcher, if you're using Python 2 (in Python 3, this is installed as part of Python).
3. Install a number of files in the installation directory the user selects:
   - The launcher script(s) that start your application
   - The icon(s) for your application launchers
   - Python packages your application needs
   - Any other files you specified
4. Create a start menu shortcut for each launcher script. If there is only one launcher, it will go in the top level of the start menu. If there's more than one, the installer will make a folder named after the application.
5. Write an uninstaller, and the registry keys to put it in 'Add/remove programs'. The uninstaller only uninstalls your application (undoing steps 3-5); it leaves Python alone, because there might be other applications using Python.

The installer (and uninstaller) is produced using NSIS, with the Modern UI.

Uncaught exceptions

If there is an uncaught exception in your application - for instance if it fails to start because a package is missing - the traceback will be written to `%APPDATA%\scriptname.log`. On Windows 7, `APPDATA` defaults to `C:\Users\username\AppData\Roaming`. If users report crashes, details of the problem will probably be found there.

You can override this by setting `sys.excepthook()`.

This is only provided if you specify your application using `entry_point`.

You can also debug an installed application by using the installed Python to launch the application. This will show tracebacks in the Command Prompt. In the installation directory run:

```
C:\\Program Files\\Application>Python\\python.exe "Application.launch.pyw"
```

Working directory

If users start your application from the start menu shortcuts, the working directory will be set to their home directory (`%HOMEDRIVE%%HOMEPATH%`). If they double-click on the scripts in the installation directory, the working directory will be the installation directory. Your application shouldn't rely on having a particular working directory; if it does, use `os.chdir()` to set it first.

FAQs

Building on other platforms

You can use Pynsist to build Windows installers from a Linux or Mac system. You'll need to install NSIS so that the `makensis` command is available. Here's how to do that on some common platforms:

- Debian/Ubuntu: `sudo apt-get install nsis`
- Fedora: `sudo dnf install mingw32-nsis`
- Mac with Homebrew: `brew install makensis`

Installing Pynsist itself is the same on all platforms:

```
pip install pynsist
```

If your package relies on compiled extension modules, like PyQt4, lxml or numpy, you'll need to ensure that the installer is built with Windows versions of these packages. There are a few options for this:

- List them under `pypi_wheels` in the *Include section* of your config file. Pynsist will download Windows-compatible wheels from PyPI. This is the easiest option if the dependency publishes wheels.
- Get the importable packages/modules, either from a Windows installation, or by extracting them from an installer. Copy them into a folder called `pynsist_pkgs`, next to your `installer.cfg` file. Pynsist will copy everything in this folder to the build directory.
- Include exe/msi installers for those modules, and modify the `.nsi` template to extract and run these during installation. This can make your installer bigger and slower, and it may create unwanted start menu shortcuts (e.g. PyQt4 does), so it's a last resort. However, if the installer sets up other things on the system, you may need to do this.

When running on non-Windows systems, Pynsist will bundle a 32-bit version of Python by default, though you can override this *in the config file*. Whichever method you use, compiled libraries must have the same bit-ness as the version of Python that's installed.

Using data files

Applications often need data files along with their code. The easiest way to use data files with Pynsist is to store them in a Python package (a directory with a `__init__.py` file) you're creating for your application. They will be copied automatically, and modules in that package can locate them using `__file__` like this:

```
data_file_path = os.path.join(os.path.dirname(__file__), 'file.dat')
```

If you don't want to put data files inside a Python package, you will need to list them in the `files` key of the `[Include]` section of the config file. Your code can find them relative to the location of the launch script running your application (`sys.modules['__main__'].__file__`).

Note: The techniques above work for fixed data files which you ship with your application. For files which your app will *write*, you should use another location, because an app installed systemwide cannot write files in its install directory. Use the `APPDATA` or `LOCALAPPDATA` environment variables as locations to write hidden data files (what's the difference?):

```
writable_file = os.path.join(os.environ['LOCALAPPDATA'], 'MyApp', 'file.dat')
```

Code signing

People trying to use your installer will see an 'Unknown publisher' warning. To avoid this, you can sign it with a digital certificate. See Mozilla's instructions on signing executables using Mono.

Signing requires a certificate from a provider trusted by Microsoft. As of summer 2017, these are the cheapest options I can find:

- Certum's open source code signing certificate: €86 for a certificate with a smart card and reader, €28 for a new certificate if you have the hardware. Each certificate is valid for one year. This is only for open source software.
- Many companies resell Comodo code signing certificates at prices lower than Comodo themselves, especially if you pay for 3–4 years up front. CodeSignCert ($59–75 per year), K Software ($67–$84 per year) and Cheap SSL Security (UK, £54–£64 per year) are a few examples; a search will turn up many more like them.

I haven't used any of these companies, so I'm not making a recommendation. Please do your own research before buying from them.

If you find another good way to get a code signing certificate, please make a pull request to add it!

Alternatives

Other ways to distribute applications to users without Python installed include freeze tools, like cx_Freeze and PyInstaller, and Python compilers like Nuitka.

pynsist has some advantages:

- Python code often does things—like using `__file__` to find its location on disk, or `sys.executable` to launch Python processes—which don't work when it's run from a frozen exe. pynsist just installs Python files, so it avoids all these problems.

- It's quite easy to make Windows installers on other platforms, which is difficult with other tools.
- The tool itself is simpler to understand, and less likely to need updating for new Python versions.

And some disadvantages:

- Installers tend to be bigger because you're bundling the whole Python standard library.
- You don't get an exe for your application, just a start menu shortcut to launch it.
- pynsist only makes Windows installers.

Popular freeze tools also try to automatically detect what packages you're using. Pynsist could do the same thing, but in my experience, this detection is complex and often misses things, so for now it expects an explicit list of the packages your application needs.

Another alternative is conda constructor, which builds an installer out of conda packages. Conda packages are more flexible than PyPI packages, and many libraries are already packaged, but you have to make a conda package of your own code as well before using conda constructor to make an installer. Conda constructor can also make Linux and Mac installers, but unlike Pynsist, it can't make a Windows installer from Linux or Mac.

Release notes

Version 1.12

- Fix a bug with unpacking wheels on Python 2.7, by switching to `pathlib2` for the pathlib backport.

Version 1.11

- Lists in the config file, such as `packages` and `pypi_wheels` can now begin on the line after the key.
- Clearer error if the specified config file is not found.

Version 1.10

- New optional field `publisher`, to provide a publisher name in the uninstall list.
- The uninstall information in the registry now also includes `DisplayVersion`.
- The directory containing `python.exe` is now added to the `%PATH%` environment variable when your application runs. This fixes a DLL loading issue for PyQt5 if you use bundled Python.
- When installing a 64-bit application, the uninstall registry keys are now added to the 64-bit view of the registry.
- Fixed an error when using wheels which install files into the same package, such as `PyQt5` and `PyQtChart`.
- Issue a warning when we can't find the cache directory on Windows.

Version 1.9

- When building an installer with Python 3.6 or above, *bundled Python* is now the default. For Python up to 3.5, 'installer' remains the default format. You can override the default by specifying `format` in the *Python section* of the config file.

- The C Runtime needed for bundled Python is now installed 'app-local', rather than downloading and installing Windows Update packages at install time. This is considerably simpler, but the app-local runtime will not be updated by Windows Update. A new `include_msvcrt` config option allows the developer to exclude the app-local runtime - their applications will then depend on the runtime being installed systemwide.

Version 1.8

- New example applications using: - PyQt5 with QML - OpenCV and PyQt5 - Pywebview
- The code to pick an appropriate wheel now considers wheels with Python version specific ABI tags like `cp35m`, as well as the stable ABI tags like `abi3`.
- Fixed a bug with fetching a wheel when another version of the same package is already cached.
- Fixed a bug in extracting files from certain wheels.
- Installers using *bundled Python* may need a Windows update package for the Microsoft C runtime. They now download this from the RawGit CDN, rather than hitting GitHub directly.
- If the Windows update package fails to install, an error message will be displayed.

Version 1.7

- Support for downloading packages as wheels from PyPI, and new PyQt5 and Pyglet examples which use this feature.
- Applications can include commands to run at the Windows command prompt. See *Command sections*.

Version 1.6

- Experimental support for creating installers that *bundle Python with the application*.
- Support for Python 3.5 installers.
- The user agent is set when downloading Python builds, so downloads from Pynsist can be identified.
- New example applications using PyGI, numpy and matplotlib.
- Fixed a bug with different path separators in `exclude` patterns.

Version 1.5

- New `exclude` option to cut unnecessary files out of directories and packages that are copied into the installer.
- The `installer.nsi` script is now built using Jinja templates instead of a custom templating system. If you have specify a custom `nsi_template` file, you will need to update it to use Jinja syntax.
- GUI applications (running under **`pythonw`**) have stdout and stderr written to a log file in `%APPDATA%`. This should catch all `print`, warnings, uncaught errors, and avoid the program freezing if it tries to print.
- Applications run in a console (under **`python`**) now show the traceback for an uncaught error in the console as well as writing it to the log file.
- Install **`pynsist`** command on Windows.
- Fixed an error message caused by unnecessarily rerunning the installer for the PEP 397 `py` launcher, bundled with Python 2 applications.

- **pynsist** now takes a `--no-makensis` option, which stops it before running **makensis** for debugging.

Version 1.0

- New `extra_preamble` option to specify a snippet of Python code to run before your main application.
- Packages used in the specified entry points no longer need to be listed under the Include section; they are automatically included.
- Write the crash log to a file in `%APPDATA%`, not in the installation directory - on modern Windows, the application can't normally write to its install directory.
- Added an example application using pygtk.
- *Installer details* documentation added.
- Install Python into `Program Files\Common Files` or `Program Files (x86)\Common Files`, so that if both 32- and 64-bit Pythons of the same version are installed, neither replaces the other.
- When using 64-bit Python, the application files now go in `Program Files` by default instead of `Program Files (x86)`.
- Fixed a bug in finding the NSIS install directory on 64-bit Windows.
- Fixed a bug that prevented using multiprocessing in installed applications.
- Fixed a bug where the `py.exe` launcher was not included if you built a Python 2 installer using Python 3.
- Better error messages for some invalid input.

Version 0.3

- Extra files can now be installed into locations other than the installation directory.
- Shortcuts can have non-Python commands, e.g. to create a start menu shortcut to a help file.
- The Python API has been cleaned up, and there is some *documentation* for it.
- Better support for modern versions of Windows:
  - Uninstall shortcuts correctly on Windows Vista and above.
  - Byte compile Python modules at installation, because the `.pyc` files can't be written when the application runs.
- The Python installers are now downloaded over HTTPS instead of using GPG to validate them.
- Shortcuts now launch the application with the working directory set to the user's home directory, not the application location.

Version 0.2

- Python 2 support, thanks to Johannes Baiter.
- Ability to define multiple shortcuts for one application.
- Validate config files to produce more helpful errors, thanks to Tom Wallroth.
- Errors starting the application, such as missing libraries, are now written to a log file in the application directory, so you can work out what happened.

Python API

Building installers

class nsist.**InstallerBuilder**(*appname*, *version*, *shortcuts*, *publisher=None*, *icon='/home/docs/checkouts/readthedocs.org/user_builds/pynsist/checkouts/latest/nsist/glossyo*
packages=None, *extra_files=None*, *py_version='3.6.1'*, *py_bitness=32*, *py_format=None*, *inc_msvcrt=True*, *build_dir='build/nsis'*, *installer_name=None*, *nsi_template=None*, *exclude=None*, *pypi_wheel_reqs=None*, *commands=None*)

Controls building an installer. This includes three main steps:

1. Arranging the necessary files in the build directory.
2. Filling out the template NSI file to control NSIS.
3. Running `makensis` to build the installer.

Parameters

- **appname** (*str*) – Application name
- **version** (*str*) – Application version
- **shortcuts** (*dict*) – Dictionary keyed by shortcut name, containing dictionaries whose keys match the fields of *Shortcut sections* in the config file
- **publisher** (*str*) – Publisher name
- **icon** (*str*) – Path to an icon for the application
- **packages** (*list*) – List of strings for importable packages to include
- **commands** (*dict*) – Dictionary keyed by command name, containing dicts defining the commands, as in the config file.
- **pypi_wheel_reqs** (*list*) – Package specifications to fetch from PyPI as wheels
- **extra_files** (*list*) – List of 2-tuples (file, destination) of files to include
- **exclude** (*list*) – Paths of files to exclude that would otherwise be included
- **py_version** (*str*) – Full version of Python to bundle
- **py_bitness** (*int*) – Bitness of bundled Python (32 or 64)
- **py_format** (*str*) – 'installer' or 'bundled'. Default 'bundled' for Python >= 3.6, 'installer' for older versions.
- **inc_msvcrt** (*bool*) – True to include the Microsoft C runtime with 'bundled' Python. Ignored when py_format='installer'.
- **build_dir** (*str*) – Directory to run the build in
- **installer_name** (*str*) – Filename of the installer to produce
- **nsi_template** (*str*) – Path to a template NSI file to use

run(*makensis=True*)

Run all the steps to build an installer.

fetch_python()
Fetch the MSI for the specified version of Python.

It will be placed in the build directory.

fetch_pylauncher()
Fetch the MSI for PyLauncher (required for Python2.x).

It will be placed in the build directory.

write_script(*entrypt*, *target*, *extra_preamble=''*)
Write a launcher script from a 'module:function' entry point

py_version and py_bitness are used to write an appropriate shebang line for the PEP 397 Windows launcher.

prepare_shortcuts()
Prepare shortcut files in the build directory.

If entry_point is specified, write the script. If script is specified, copy to the build directory. Prepare target and parameters for these shortcuts.

Also copies shortcut icons

prepare_packages()
Move requested packages into the build directory.

If a pynsist_pkgs directory exists, it is copied into the build directory as pkgs/ . Any packages not already there are found on sys.path and copied in.

copy_extra_files()
Copy a list of files into the build directory, and add them to install_files or install_dirs as appropriate.

write_nsi()
Write the NSI file to define the NSIS installer.

Most of the details of this are in the template and the *nsist.nsiswriter.NSISFileWriter* class.

run_nsis()
Runs makensis using the specified .nsi file

Returns the exit code.

Writing NSIS files

class nsist.nsiswriter.**NSISFileWriter**(*template_file*, *installerbuilder*, *definitions=None*)
Write an .nsi script file by filling in a template.

__init__(*template_file*, *installerbuilder*, *definitions=None*)
Instantiate an NSISFileWriter

Parameters

- **template_file** (*str*) – Path to the .nsi template
- **definitions** (*dict*) – Mapping of name to value (values will be quoted)

write(*target*)
Fill out the template and write the result to 'target'.

Parameters **target** (*str*) – Path to the file to be written

Copying Modules and Packages

class nsist.copymodules.**ModuleCopier**(*py_version*, *path=None*)
Finds and copies importable Python modules and packages.

There is a Python 3 implementation using importlib, and a Python 2 implementation using imp.

copy(*modname*, *target*)
Copy the importable module 'modname' to the directory 'target'.

modname should be a top-level import, i.e. without any dots. Packages are always copied whole.

This can currently copy regular filesystem files and directories, and extract modules and packages from appropriately structured zip files.

nsist.copymodules.**copy_modules**(*modnames*, *target*, *py_version*, *path=None*, *exclude=None*)
Copy the specified importable modules to the target directory.

By default, it finds modules in sys.path - this can be overridden by passing the path parameter.

Example applications

Simplified examples

The repository contains a number of simple examples for building applications with different frameworks:

- A console application
- A tkinter application
- A PyQt4 application
- A PyQt5 application
  - PyQt5 with QML
  - PyQt5 with OpenCV
- A PyGTK application
  - PyGTK with Numpy and Matplotlib (32 bit, Python 2.7)
- A PyGI (or PyGObject) application with Numpy and Matplotlib (64 bit, Python 3.4)
- A pygame application
- A pyglet application
- A pywebview application

Real-world examples

These may illustrate more complex uses of pynsist.

- The author's own application, Taxonome, is a Python 3, PyQt4 application for working with scientific names for species.
- Spreads is a book scanning tool, including a tkinter configuration system and a local webserver. Its use of pynsist (see buildmsi.py) includes working with setuptools info files.
- InnStereo is a GTK 3 application for geologists. Besides pygi, it uses numpy and matplotlib.

Design principles

or *Why I'm Refusing to Add a Feature*

There are some principles in the design of Pynsist which have led me to turn down potentially useful options. I've tried to explain them here so that I can link to this rather than summarising them each time.

1. Pynsist is largely a **simplifying wrapper** around NSIS: it provides an easy way to do a subset of the things NSIS can do. All simplifying wrappers come under pressure from people who want to do something just outside what the wrapper currently covers: they'd love to use the wrapper, if it just had one more option. But if we keep adding options, eventually the simplifying wrapper becomes a convoluted layer of extra complexity over the original system.
2. I'm very keen to **keep installers as simple as possible**. There are all sorts of clever things we could do at install time. But it's much harder to write and test the NSIS install code than the Python build code, and errors when the end user installs your software are a bigger problem than errors when you build it, because you're better able to understand and fix them. So Pynsist does as much as possible at build time so that the installer can be simple.
3. Pynsist has a **limited scope**: it builds Windows installers for Python applications. Mostly GUI applications, but it does also have support for adding command-line tools. I don't plan to add support for other target platforms or languages.

If you need more flexibility

If you want to do something which Pynsist doesn't support, there are several ways it can still help you:

- **Generate an nsi script**: You can run Pynsist once with the `--no-makensis` option. In the build directory, you'll find a file `installer.nsi`, which is the script for your installer. You can modify this and run `makensis installer.nsi` yourself to build the installer.
- **Write a custom template**: Pynsist uses *Jinja* templates to create the nsi script. You can write a custom template and specify it in the *Build section* in your config file. Custom templates can inherit from the templates in Pynsist and override blocks, so you have a lot of control over the installer this way.
- **Cannibalise the code**: Pull out whatever pieces are useful to you from Pynsist and use them in your build scripts. There are the installer templates, code to find and download wheels from PyPI, to download Python itself, to create command-line entry points, to find `makensis.exe` on Windows, and so on. You can take specific bits to reuse, or copy the whole thing and apply some changes.

See also the examples folder in the repository.

The API is not yet documented here, because I'm still working out how it should be structured. The functions and classes have docstrings, and you're welcome to use them directly, though they may change in the future.

See also:

pynsist source code on Github

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

n

Index